

IPLG : Un outil pour la fusion d'opérateurs en Traitement d'Images

Stéphane Piskorski*, Lionel Lacassagne** et Daniel Etiemble*

*Laboratoire de Recherche en Informatique,

**Institut d'Electronique Fondamentale,
Université Paris-Sud 11

Résumé

Nous présentons un outil de génération et de transformation de codes pour le Traitement d'Images. A partir d'une description simple du graphe de flux de données, l'outil génère plusieurs codes C correspondant, en fusionnant les nids de boucle et les opérateurs successifs, et en appliquant automatiquement les optimisations traditionnelles de rotations de variables, réduction ou déroulage de boucle. Un gain significatif d'un facteur deux est obtenu pour des processeurs généralistes PowerPC et Intel et pour des processeurs embarqués customisables NiosII. Ce gain est obtenu pour un effort minimum de codage de la part du traicteur d'images.

Mots-clés : traitement d'images, fusion de boucles, FPGA

1. Introduction

Les optimisations, tant matérielles que logicielles, jouent un rôle majeur pour l'amélioration des performances des applications de traitement d'images et des applications multimédias, à la fois sur les processeurs d'usage général et sur les processeurs enfouis ou embarqués.

Parmi les optimisations matérielles classiques, l'ajout d'instructions spécialisées (customization) est une technique classique dans les processeurs généralistes ou dans les processeurs configurables. La fameuse instruction SIMD calculant la somme des valeurs absolues des différences de huit pixels, que l'on trouve dans les jeux d'instructions IA-32 (SSE) ou PowerPC (AltiVec) utilisé en estimation de mouvement en est un exemple typique. Dans [4], nous avons montré l'intérêt d'instructions SIMD flottantes 16 bits en traitement d'images.

Si les extensions SIMD sont un bon exemple d'ajouts d'instructions dans les processeurs configurables, il reste que ces instructions ne sont facilement utilisables que si les formats des données SIMD sont les mêmes en entrée et en sortie. C'est le cas avec les formats flottants. Ce n'est pas le cas avec les formats entiers. Dès qu'une fonction travaillant sur un format entier effectue un calcul arithmétique, les instructions effectuant les calculs intermédiaires travaillent avec un format différent du format des entrées et de la sortie, avec généralement un nombre supérieur de bits. Dans ce cas, il faut optimiser la fonction globalement, et non telle ou telle instruction utilisée par la fonction.

C'est ce problème d'optimisation au niveau fonctions qui est abordé dans cet article sur un exemple significatif d'application de traitement d'images. Dans tous les cas, il s'agit de trouver le meilleur compromis entre le temps de calcul (opérateurs matériels utilisés) et le temps d'accès aux données (performances de la hiérarchie mémoire).

Il peut être abordé d'un point de vue strictement logiciel, sur un processeur généraliste (superscalaire ou VLIW) ou sur un processeur configurable pour minimiser le temps d'exécution global en fonction de la décomposition de la fonction en différentes sous-fonctions. Dans ce cas, il s'agit de trouver la meilleure décomposition selon les instructions et les ressources matérielles du processeur (opérateurs de calcul), ainsi que les performances de la hiérarchie mémoire (taille et latence des caches).

Il peut également être abordé d'un point de vue matériel, notamment pour les processeurs configurables. Au problème « logiciel » précédent s'ajoute un degré de liberté supplémentaire avec la possibilité d'augmenter le parallélisme d'opérateurs pour exécuter les différentes sous-fonctions. Par exemple, la possibilité de « compiler du matériel » permet de résoudre le problème des instructions SIMD sur des

données entières et de faire des calculs « pseudo-SIMD » en calculant en parallèle des sous-fonctions, comme nous l’avons montré dans [7].

La suite de cet article est organisé de la manière suivante. Le chapitre 2 présente les optimisations habituellement utilisées pour le codage d’un filtre en traitement de signal ; ce sont ces optimisations que l’outil se propose d’automatiser. Le chapitre 3 présente l’outil du point de vue de l’utilisateur, et décrit la démarche amenant jusqu’à la génération automatique du code C. Ensuite, le chapitre 4 présente la méthodologie utilisée pour évaluer l’impact de la fusion d’opérateurs réalisée par l’outil, notamment le *benchmark* utilisé et les conditions expérimentales. Enfin le chapitre 5 présente les résultats obtenus sur le filtre d’exemple pour différentes architectures.

2. Optimisations

Le déroulage de boucle (*loop unrolling*) est une transformation classique pour les compilateurs optimisants qui a pour but principal d’améliorer le fonctionnement du pipeline du processeur. Il en va de même pour le déroulage avec mélange (*unroll and jam*) qui s’attaque au problème de la dépendance de données. Si l’ordre de déroulage est quelconque dans le cas général et peut être choisi par l’utilisateur ou le compilateur, en Traitement du Signal et des Images (TSI), il existe pour chaque opérateur un ordre de déroulage optimal.

2.1. Optimisations en TS

Soit le filtre h non récursif de taille m défini par l’équation 1 :

$$y(n) = \sum_{k=0}^{m-1} x(n-k) \times h(k) \quad (1)$$

Algorithm 1: filtre initial

```

1 for i = 0 + m - 1 to n do
2   y(i) = 0
3   for k = 0 to m do
4     y(i) ← y(i) + x(i - k) × h(k)

```

Le nombre d’accès mémoire est égal à $4m$ si on accumule dans la case mémoire $y(n)$ ou $2m + 1$ si on accumule dans un registre temporaire. La première transformation à réaliser est de dérouler entièrement la boucle interne (*loop unwinding*) qui calcule une valeur en sortie $y(n)$. Dans le cas d’un filtre de taille $m = 3$ nous obtenons (Eq. 2) :

$$y(n) = x(n-0) \times h(0) + x(n-1) \times h(1) + x(n-2) \times h(2) \quad (2)$$

La deuxième étape consiste à dérouler également la boucle externe d’un ordre m (Eq. 5) :

$$y(n+0) = x(n+0) \times h(0) + x(n-1) \times h(1) + x(n-2) \times h(2) \quad (3)$$

$$y(n+1) = x(n+1) \times h(0) + x(n+0) \times h(1) + x(n-1) \times h(2) \quad (4)$$

$$y(n+2) = x(n+2) \times h(0) + x(n+1) \times h(1) + x(n+0) \times h(2) \quad (5)$$

La troisième étape est dépendante du domaine applicatif et consiste à prendre en compte la nature du filtre pour réaliser la *scalarisation* de manière optimale. Lorsque le filtre est appliqué une seconde fois en $n + 1$, tous les points participant au calcul sont connus, sauf le dernier qui est le seul à devoir être lu. De même les coefficients du filtre étant invariants durant la boucle, ils sont pré-chargés dans des registres. Posons $x_k = x(n - k)$ (en début de boucle) et $h_k = h(k)$. Il est aussi nécessaire de réaliser l’initialisation

Algorithm 2: filtre doublement déroulé

```
1  $h_0 \leftarrow H(0), \quad h_1 \leftarrow H(1), \quad h_2 \leftarrow H(2)$ 
2  $x_2 \leftarrow X(0), \quad x_1 \leftarrow X(1)$ 
3 for  $i = 2$  to  $n$  do
4    $x_0 \leftarrow X(i+0) \Rightarrow Y(i+0) \leftarrow x_0 \times h_0 + x_1 \times h_1 + x_2 \times h_2$ 
5    $x_2 \leftarrow X(i+1) \Rightarrow Y(i+1) \leftarrow x_2 \times h_0 + x_0 \times h_1 + x_1 \times h_2$ 
6    $x_1 \leftarrow X(i+2) \Rightarrow Y(i+2) \leftarrow x_1 \times h_0 + x_2 \times h_1 + x_0 \times h_2$ 
```

des différents registres dans un prologue. Lorsque le registre le plus ancien n'est plus utile, il est recyclé pour le calcul du prochain point. Nous obtenons ainsi le pseudo-code de l'algorithme 2 :

Il faut bien sûr que les registres jouent le même rôle lors de la prochaine itération de la boucle, ce qui n'est possible que lorsque l'ordre de déroulage est égal à l'ordre du filtre. Dans cet exemple (algo. 2), x_1 qui est recyclé à la fin de la boucle joue bien le rôle de $x(n'-1)$ dans la boucle suivante (avec $n' = n+3$). Ainsi chaque registre x_k participe à m calculs. D'un point de vue flot de donnée, nous obtenons un fonctionnement *systolique* à chaque fois qu'un nouveau point est consommé, un nouveau point est produit.

2.2. Optimisations en TI

Dans le cas de filtres à deux dimensions appliqués à des images, en plus des techniques précédentes qui s'appliquent, il existe une technique supplémentaire propre à ce domaine applicatif. Il est fréquent que les filtres utilisés soient des filtres 2D séparables en deux filtres 1D, que ce soient des filtres passe-bas (suppression du bruit) ou passe-haut (détection de contours). Soit B_3 le filtre Binomial 3×3 séparable :

$$B_3 = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \frac{1}{4} [1 \quad 2 \quad 1] \quad (6)$$

Au lieu d'appliquer le même schéma de calcul que dans le cas mono-dimensionnel, des valeurs *réduites* par colonne sont calculées (en utilisant le filtre 1D vertical) et stockées en registres. C'est sur ces registres que s'applique alors le déroulage de boucle et le calcul final provient de la combinaison (en utilisant le filtre 1D horizontal) des différentes valeurs réduites.

Cas général : filtre 2D de taille $m \times m$ avec $m = 2k + 1$ indices coefficients centrés en 0.

Algorithm 3: implantation basique d'un filtre 2D de taille $k \times k$

```
1 for  $i = k$  to  $n - 1 - k$  do
2   for  $j = k$  to  $n - 1 - k$  do
3      $y \leftarrow 0$ 
4     for  $\delta_i = -k$  to  $k$  do
5       for  $\delta_j = -k$  to  $k$  do
6          $y \leftarrow y + X(i + \delta_i, j + \delta_j) \times H(\delta_i, \delta_j)$ 
7      $Y(i, j) \leftarrow y$ 
```

Les multiplications par 1 sont volontairement laissées afin de faire apparaître les coefficients des filtres horizontaux et verticaux. Dans une implantation optimisée, les multiplications par 1 seraient supprimées et les multiplications entières par 2 remplacées par des décalages.

On notera la rotation des coefficients r_a , r_b et r_c (algo. 4, lignes 6, 8 et 10) qui jouent chacun alternativement le rôle de colonne de gauche, colonne centrale et colonne de droite.

Dans le cas général, cette combinaison de transformations logicielles et algorithmiques permet d'obtenir un gain proportionnel à m (Tab. 1). Sans ces transformations algorithmiques, le gain ne se serait situé qu'au niveau des accès mémoire.

Algorithm 4: Implantation optimisée du filtre binomial 3×3

```

1 for i = 1 to n - 1 do
2    $x_{a0} \leftarrow X(i-1,0), \quad x_{a1} \leftarrow X(i,0), \quad x_{a2} \leftarrow X(i+1,0), \quad \Rightarrow \quad r_a \leftarrow 1x_{a0} + 2x_{a1} + 1x_{a2}$ 
3    $x_{b0} \leftarrow X(i-1,1), \quad x_{b1} \leftarrow X(i,1), \quad x_{b2} \leftarrow X(i+1,1), \quad \Rightarrow \quad r_b \leftarrow 1x_{b0} + 2x_{b1} + 1x_{b2}$ 
4   for j = 1 to n - 1 step 3 do
5      $x_{c0} \leftarrow X(i-1,j+1), \quad x_{c1} \leftarrow X(i,j+1), \quad x_{c2} \leftarrow X(i+1,j+1) \quad \Rightarrow \quad r_c \leftarrow 1x_{c0} + 2x_{c1} + 1x_{c2}$ 
6      $Y(i,j) \leftarrow 1r_a + 2r_b + 1r_c$ 
7      $x_{a0} \leftarrow X(i-1,j+2), \quad x_{a1} \leftarrow X(i,j+2), \quad x_{a2} \leftarrow X(i+1,j+2) \quad \Rightarrow \quad r_a \leftarrow 1x_{a0} + 2x_{a1} + 1x_{a2}$ 
8      $Y(i,j+1) \leftarrow 1r_b + 2r_c + 1r_a$ 
9      $x_{b0} \leftarrow X(i-1,j+3), \quad x_{b1} \leftarrow X(i,j+3), \quad x_{b2} \leftarrow X(i+1,j+3) \quad \Rightarrow \quad r_b \leftarrow 1x_{b0} + 2x_{b1} + 1x_{b2}$ 
10     $Y(i,j+2) \leftarrow 1r_c + 2r_a + 1r_b$ 

```

| | filtre basique | filtre optimisé | gain optimisations |
|---------------|-----------------------------|---------------------------|--------------------|
| accès mémoire | m^2 LOAD + 1 STORE | m LOAD + 1 STORE | $m/2$ |
| arithmétique | m^2 MUL + $(m^2 - 1)$ ADD | $2m$ MUL + $2(m - 1)$ ADD | m |

TABLE 1 – Complexité par point (sans / avec) optimisation : un filtre $m \times m$

Cette méthode dite de *réduction par colonne* est aussi applicable aux filtres récursifs. Ce type de filtre reste bien plus difficile à optimiser par les compilateurs, à cause des dépendances de données [1] [2] et doit, actuellement, toujours être transformé manuellement le temps que les compilateurs optimisants intègrent des optimisations spécifiques et *scriptables* [3].

2.3. Fusion d'opérateurs en TI

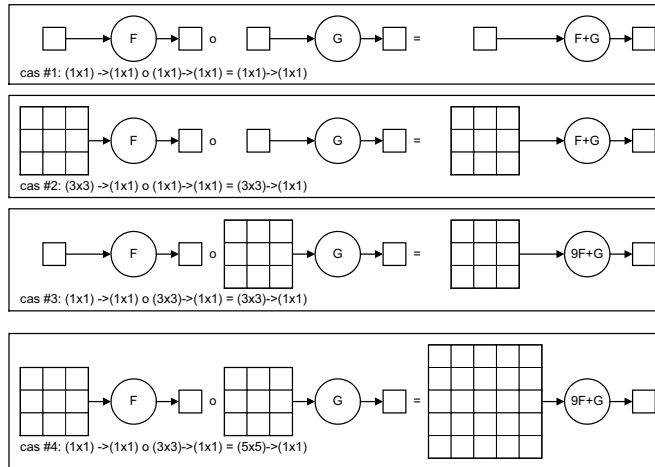


FIGURE 1 – Quatre cas de chaînage

Avant de réaliser le chaînage de deux opérateurs, il est nécessaire d'adapter leurs motifs de consommation et production mémoire. L'adapter permet de déterminer le nombre de fois que le premier opérateur devra être répliqué. Notons $F : (a \times b) \rightarrow (c \times d)$; l'opérateur F qui consomme un motif de taille $(a \times b)$ et qui produit un motif de taille $(c \times d)$. La figure 1 présente quatre cas de chaînage.

1. chaînage de deux opérateurs ponctuels : $[(1 \times 1) \rightarrow (1 \times 1)] \circ [(1 \times 1) \rightarrow (1 \times 1)] = [(1 \times 1) \rightarrow (1 \times 1)]$: les motifs d'accès mémoire sont déjà adaptés et ne changent pas. Le nombre d'appel aux opérateurs ne change pas non plus.

2. chaînage d'un opérateur à bords avec un opérateur ponctuel : $[(3 \times 3) \rightarrow (1 \times 1)] \circ [(1 \times 1) \rightarrow (1 \times 1)] = [(3 \times 3) \rightarrow (1 \times 1)]$: les motifs d'accès mémoire sont déjà adaptés et ne changent pas. Le nombre d'appel au opérateurs ne change pas non plus.
3. chaînage d'un opérateur ponctuel avec un opérateur à bords : $[(1 \times 1) \rightarrow (1 \times 1)] \circ [(3 \times 3) \rightarrow (1 \times 1)] = [(3 \times 3) \rightarrow (1 \times 1)]$: les motifs d'accès mémoire doivent être adaptés. Le premier opérateur doit être dupliqué 3×3 fois. Les motifs de production - consommations des cas 2 et 3 sont donc identiques, mais le nombre d'appel d'opérateur est différent.
4. chaînage de deux opérateurs avec bords : $[(3 \times 3) \rightarrow (1 \times 1)] \circ [(3 \times 3) \rightarrow (1 \times 1)] = [(5 \times 5) \rightarrow (1 \times 1)]$: comme dans le cas précédent, les motifs d'accès mémoire doivent être adaptés et premier opérateur doit être dupliqué 3×3 fois.

Les deux premiers cas sont toujours des transformations améliorant les performances, par ce qu'il y a suppression de l'accès mémoire intermédiaire, et un meilleur fonctionnement du pipeline. Les deux derniers cas ne le sont pas nécessairement car le premier opérateur est appelé 9 fois, générant donc 9 fois plus de calculs pour ne produire qu'un seul point en sortie.

2.4. Réduction par colonne et fusion d'opérateurs en TI

Combiner *réduction par colonne* et fusion d'opérateurs limite l'explosion des calculs. En reprenant les expressions des complexités données dans la table 1 et en remplaçant m par 3 ou 5, nous obtenons les complexités de deux filtres 3×3 ou d'un filtre 5×5 avec et sans optimisation.

| | 2 filtres 3×3 non optimisés | 1 filtre 5×5 non optimisé | 2 filtres 3×3 optimisés | 1 filtre 5×5 optimisé |
|---------------|--------------------------------------|------------------------------------|----------------------------------|--------------------------------|
| accès mémoire | 18 LOAD + 2 STORE | 25 LOAD + 1 STORE | 6 LOAD + 2 STORE | 5 LOAD + 1 STORE |
| arithmétique | 36 MUL + 16 ADD | 25 MUL + 24 ADD | 12 MUL + 12 ADD | 10 MUL + 8 ADD |

TABLE 2 – Complexité de deux filtres 3×3 vs. un filtre 5×5 optimisé

Nous pouvons observer plusieurs points intéressants :

- si le filtre 5×5 non optimisé a une complexité arithmétique plus petite que deux filtres 3×3 non optimisés, sa complexité mémoire est plus grande ce qui est disqualifiant puisque ces filtres sont de type *memory bound*,
- les complexités après optimisations sont proches, même si le filtre 5×5 optimisé est un peu meilleur. Il est donc nécessaire de tester tous les cas.

Ces complexité sont données dans le cas général où on considère qu'il y a m^2 multiplications et $m^2 - 1$ additions pour chaque filtre. Cela peut être considéré comme pessimiste car de nombre filtres ont des coefficients symétriques par rapport à celui du milieu (permettant de factoriser certains calculs), ou un certains nombre de coefficients à 1 ou 0 (permettant d'éliminer certains termes).

Dans la section 4.1 nous donnerons les complexités des filtres considérés.

3. Outils de fusion automatique

Le but de l'outil IPLG est de permettre la génération automatique du code C implémentant un filtre de traitement d'images régulier, donné par l'utilisateur sous la forme d'un graphe décrit par un fichier XML. Les opérateurs sont également décrits par l'utilisateur sous la forme d'un pseudo-code (C enrichi) dont nous verrons un exemple.

A partir de ces données, l'outil va s'attacher à optimiser le code généré par deux méthodes complémentaires. La première va consister à « fusionner » les opérateurs, c'est à dire à les regrouper au sein d'une même double-boucle de parcours de l'image, comme vu au paragraphe 2.3. Ceci afin de maximiser la localité des calculs, et de minimiser le nombre d'accès mémoire. L'optimalité est atteinte quand on trouve le regroupements d'opérateurs présentant le bon compromis entre nombres d'accès mémoires nécessaires au rangement de calculs intermédiaires, et complexité du code généré.

Une fois listés les regroupements possibles, l'outil va dans un deuxième temps réaliser automatiquement les techniques de rotation de variables, de réduction et de déroulage de boucle vues au paragraphe 2.2, afin de réduire le nombre de calculs (réduction) et le nombre de variables temporaires à conserver par un noyau de calcul à un instant t (déroulage). Ce nombre sera d'autant plus important que le regroupement traité comportera de nombreux opérateurs fusionnés. En cas de taille insuffisante du banc de registres, leur rangement sur la pile (*spill code*) viendra contrebalancer les gains obtenus par la réduction du nombre d'accès aux images à traiter.

L'ensemble des regroupements possibles d'opérateurs, avec pour chacun plusieurs types d'optimisations possibles, forment alors un panel d'implémentations du même filtre, que l'on va évaluer afin de trouver la plus efficace. Cette exploration peut se faire de manière exhaustive, ou sur un sous-ensemble de regroupements donnés par l'utilisateur (choisis manuellement ou via un outil extérieur calculant leur complexité en termes d'accès mémoires et de calculs).

3.1. Représentation du graphe

La première donnée d'entrée fournie par l'utilisateur est un fichier XML représentant le graphe de flot de données (voir figure 2) à implémenter. Chacun des noeuds représentant un calcul implémente un TCL (Traitement Combinatoire Local), et ce calcul devra rester simple. En effet l'outil travaillant à la recombinaison de ces noeuds pour générer du code, le calcul stocké dans ces noeuds représente la granularité à laquelle travaille l'outil. Des calculs plus simples (un grain plus fin) peuvent potentiellement présenter de meilleures optimisations (voir par exemple la réduction 2.4 permise par la décomposition des noyaux 3×3 en deux noyaux 1×3 et 3×1).

```
<?xml version="1.0" encoding="UTF-8"?>
<dataflow name="HARRIS" architecture="x86">
  <flowinput name="I0" datatype="F32" width="512" height="512" />
  <flowoutput name="O0" />

  <data name="GX" /> <data name="GY" />
  ...
  <operator name="Gradient" type="SOBEL2D33_F32">
    <input E="I0" /> <output GX="GX" /> <output GY="GY" />
  </operator>
  <operator name="CarreX" type="SQR_F32">
    <input E="GX" />
    <output S = "IXX" />
  </operator>
  ...
  <operator name="K" type="HARRIS_F32">
    <input XX="GIXX" /> <input XY="GIXY" /> <input YY="GIYY" />
    <output K="O0" />
  </operator>
</dataflow>
```

TABLE 3 – Description XML du filtre de Harris (extrait)

3.2. Représentation des opérateurs

Les opérateurs utilisés sont tels qu'un point de sortie est fonction des points d'entrée situés aux mêmes coordonnées et des points situés dans leur voisinage immédiat. Soit en notant S l'image de sortie, E_e une image en entrée ($e \in [1..n_e]$), j et i l'abscisse et l'ordonnée du point de S que l'on souhaite calculer,

$$S[i][j] = f(\dots, E_e [i - \delta_{i0_e} .. i + \delta_{i1_e}] \times [j - \delta_{j0_e} .. j + \delta_{j1_e}] , \dots)_{e \in [1..n_e]}$$

Pour chaque image d'entrée, on définit un *voisinage* de points, représentant les points nécessaires au calcul du point de sortie. Ce voisinage s'étend respectivement de $\delta_{i0_e}, \delta_{i1_e}, \delta_{j0_e}$ et δ_{j1_e} pixels à gauche, droite, au-dessus et au-dessous du point de référence $E_c[i][j]$.

L'outil s'intéressant essentiellement à augmenter la localité des calculs, seuls le motif d'accès à la mémoire de l'opérateur est important (c.-à-d. la taille du voisinage) et la fonction f implémentée peut être quelconque. Cependant dans les exemples simples les plus courants, cette fonction sera une convolution comme vue au paragraphe 2.2.

L'utilisateur saisit un opérateur sous la forme d'un fichier XML (exemple 4) dans lequel on trouve entre autres :

- une description des images en entrée/sortie de l'opérateur, avec le type (entiers, flottants, triplets RVB) des données stockées ;
- une ligne de pseudo-code - du code C annoté dans lequel la référence à un point est repérée par le symbole $@nom_image[\delta_i][\delta_j]$, où δ_i et δ_j sont les coordonnées du point au sein du voisinage du point (i, j) en cours de calcul. La taille du voisinage est extraite par recherche des δ_i et δ_j maximum présents dans le pseudo-code.

L'écriture de la ligne de pseudo-code, et donc l'efficacité du code généré, est ici dépendante du niveau d'optimisation que l'utilisateur y injecte. Cette ligne n'est pas modifiée par l'outil ; il faut donc à ce stade faire attention, de fournir un code optimisé pour le compilateur ciblé, par exemple en factorisant les expressions, en utilisant des décalages au lieu de multiplications entières, ou en insérant des instructions *intrinsic* de calcul SIMD (SSE, AltiVec). Dans ce dernier cas, l'insertion d'un attribut indiquant à l'outil que cet opérateur produit ou consomme les points par paquets de n points est prévu.

```
<?xml version="1.0" encoding="UTF-8"?>
<operator name="GAUSS33_F32" architecture="x86">
  <input name="E" type="F32" />
  <output name="S" type="F32" />
  <code>
    <![CDATA[
      @S[0][0]=
        4.0f*@E[0][0]
        +2.0f * (@E[-1][0]+@E[1][0]+@E[0][-1]+@E[0][+1])
        + (@E[-1][-1]+@E[-1][+1]+@E[+1][-1]+@E[+1][+1]);
      @S[0][0]*= (float)0.0625;
    ]]>
  </code>
</operator>
```

TABLE 4 – Description XML d'un filtre de Gauss 3x3
(opérateur entièrement déroulé - sans boucle sur les coefficients)

3.3. Génération de code

A partir du graphe modélisant le filtre à implémenter, les différentes phases vont consister à :

- déterminer quelles sont les opérations de calcul à regrouper. Dans la version actuelle de l'outil, tous les regroupements possibles (réalisant le calcul correct) sont testés.
- générer les nids de boucles de calcul correspondant aux groupes d'opérateurs retenus. Plusieurs versions de ces nids de boucle sont écrites, en tentant d'appliquer automatiquement quelques optimisations standards bien connues en traitement du signal.

3.4. Recherche des opérateurs à fusionner

Le but de cette phase est de déterminer quels sont les opérateurs qui vont être regroupés et appliqués ensemble au sein d'une même double-boucle de traitement, et donc quelles images intermédiaires ne seront pas stockées. Cette phase de recherche est itérative. En appelant *mapping* un certain regroupement

d'opérateurs, on suppose initialement un *mapping* où chaque opérateur est appliqué séparément. Puis, pour chaque couple de groupes d'opérateurs, on va tester :

- si ces deux groupes partagent au minimum une image en entrée/sortie
- si aucune image de sortie du groupe fusionné potentiel ne possède dans la liste des noeuds images dont il dépend une des entrées du groupe fusionné.

Cette recherche se termine quand on arrive à un *mapping* d'un seul groupe contenant tous les opérateurs du graphe.

L'outil génère tous les regroupements d'opérateurs que le graphe permet. Dans l'idéal, une heuristique déterminant les opérateurs à regrouper à partir d'une description de l'architecture de la machine permettrait un gain de temps dans la recherche du *mapping* le plus efficace. Une telle heuristique pourrait par exemple évaluer le ratio entrées-sorties sur calculs, permettant d'estimer si le filtre généré est de type *memory-bound* ou *computation-bound*. Il faudrait alors confronter cette quantité à des paramètres architecturaux comme les taille et latence du banc de registres et des caches. Cependant, tenter de déterminer par avance si telle ou telle version d'un filtre améliore la localité des données et plus généralement les performances, nécessiterait une connaissance très fine de l'architecture qu'il est trop difficile d'obtenir. C'est pourquoi il est tout à fait acceptable, au vu de la taille des filtres à implémenter, d'évaluer exhaustivement par *benchmark* les performances de chaque *mapping* et de conserver le plus rapide à l'issue de ces tests.

Cette approche permet par ailleurs de s'affranchir d'une machine particulière, les optimisations proposées (déroulage de boucles, rotations) étant communes à toutes les machines considérées dans cet article. Explorer de façon quasi-exhaustive un panel d'optimisations choisies empiriquement est une solution que l'on retrouve dans des projets d'optimisation automatique de bibliothèques, telles que Atlas [9], FFTW [8] ou Spiral [6], quand un temps important passé à optimiser un programme est largement compensé par le temps gagné à l'exécution. Un filtre de traitement d'images rentre typiquement dans ce cas de figure, par le fait qu'il est destiné à être exécuté de façon intensive (25 à 50 fois par seconde sur une longue période dans le cas de traitements de flux vidéo).

3.5. Génération de code pour un groupe d'opérateurs

Pour chaque groupe d'opérateurs présents dans les différents mappings à tester, une double-boucle *POUR* de traitement est écrite. Celle-ci balaie les images à traiter ligne par ligne (on suppose un stockage mémoire où deux pixels consécutifs d'une ligne sont consécutifs en mémoire). Le code C implémentant le coeur de boucle est généré en remontant son graphe de la sortie vers les entrées, et en écrivant le pseudo-code des opérateurs rencontrés. Dans ce pseudo-code (du code C annoté), les références aux points en cours de calcul sont alors remplacées par les noms des images réellement traitées, indicées par les compteurs de la double-boucle. Par analyse des motifs d'accès mémoire et des dépendances entre calculs de points, l'outil recense pour chaque nid de boucle généré les points qui feront l'objet d'un chargement, d'un stockage, d'une rotation, etc. Il remplit alors un squelette de code commun à toutes les implémentations.

4. Benchmark et conditions expérimentales

4.1. Opérateur de Traitement d'Images : détecteur de points d'intérêts Harris

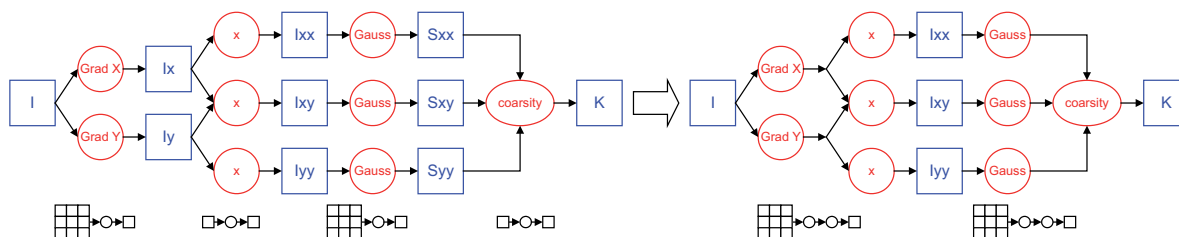


FIGURE 2 – Transformation *Halfpipe* de l'opérateur de Harris

L'opérateur de Harris a été choisi car il met en évidence l'impact du chaînage d'opérateurs. Etant composée de quatre phases successives de calculs (gradient, produits des dérivées premières, lissage des produits et coarsité), il implique dans son implantation classique, l'accès à huit mémoires intermédiaires (fig. 2). Il est ainsi possible de chaîner les opérateurs Sobel et Mul d'une part et les opérateurs Gauss et Coarsité d'autre part, car leur motifs de consommation et production mémoire sont compatibles : $(3 \times 3) \rightarrow (1 \times 1)$ et $(1 \times 1) \rightarrow (1 \times 1)$. Cette transformation est appelée *Halfpipe*. Il est aussi possible de chaîner entièrement les opérateurs (fig. 3) c'est la version *Fullpipe*. Dans ce cas il est alors nécessaire d'adapter les motifs de consommation et production : pour produire 1×1 point en sortie, il est nécessaire d'en consommer 5×5 et d'exécuter 3×3 fois l'ensemble Sobel+Mul. Notons aussi que la version *Fullpipe* est bien plus complexe que les autres : jusqu'à 9 fois plus de calculs dans sa version basique. Une fois optimisées, les versions *Halfpipe* et *Fullpipe* ont une complexité de calcul très proches (5), tandis que la version *Fullpipe* a une complexité mémoire bien moindre. La version *Halfpipe* reste donc *memory-bound* alors que la version *Fullpipe* pourra devenir *computation-bound* en fonction de l'architecture (taille du banc de registres, vitesse de l'ALU, de la FPU, etc.).

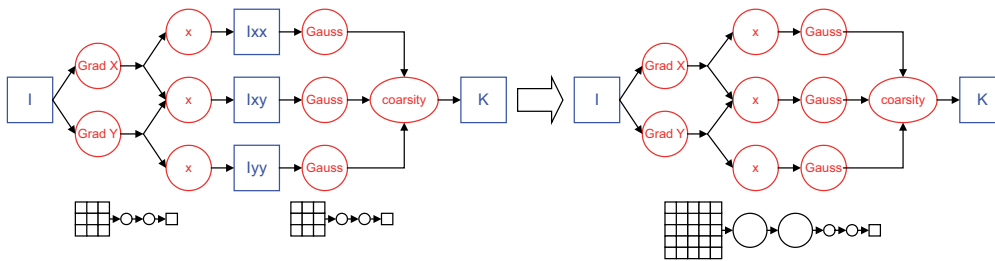


FIGURE 3 – Transformation *Fullpipe* de l'opérateur de Harris

| | no Pipe | Halfpipe optimisé | Fullpipe optimisé |
|---------------|-------------------|------------------------|-------------------------|
| accès mémoire | 54 LOAD + 9 STORE | 18 LOAD + 4 STORE | 5 LOAD + 1 STORE |
| arithmétique | 24 MUL + 35 ADD | 16 MUL + 19 ADD | 26 MUL + 31 ADD |

TABLE 5 – Complexité de l'opérateur de Harris

4.2. Conditions de test

Nous avons testé les performances de différentes décompositions sur trois processeurs différents, deux processeurs généralistes et un processeur « logiciel » implanté sur FPGA. Dans les trois cas, les résultats pour chaque configuration correspondent à la valeur minimale sur un certain nombre d'exécution (au moins 5). Une mesure, exprimée en nombre de cycles d'horloge par pixel ou CPP, est déduite du nombre de cycles d'horloge écoulés, mesurés via l'instruction processeur idoine. Le processus est placé en priorité haute. Les données de calcul sont accédées via une double indexation. Pour chaque image, un bloc contigu de données est alloué, puis un index est construit sous la forme d'un tableau de pointeurs pointant sur le début de chaque ligne de l'image (gestion mémoire par offset de type Numerical Recipes).

Processeur Intel

La machine de test est équipée d'un processeur Intel Xeon E5450 (Quad Core) où un coeur accède à 128 Ko de cache de données de premier niveau et 6 Mo de cache L2. Les codes évalués ont été compilés avec le compilateur C++ Intel version 11.0.069, avec l'option O3, sans parallélisation automatique.

Processeur Power PC

Les mesures ont été effectuées sur un PowerPC 970MP, avec 1Mo de cache L2. Le code est compilé avec XLC version 9.0, option -O3.

Processeur NiosII (Altera)

Le processeur NiosII est un processeur « logiciel » implantable sur les FPGA d’Altera. Nous l’avons utilisé dans sa version f, avec multiplication matérielle utilisant les blocs DSP, division matérielle (qui n’est pas utilisée dans l’algorithme), un cache instructions de 4 Ko et un cache de données de 2 Ko avec des lignes de 4 octets. Il fut testé sur la version EP2S60 du kit Stratix II, qui contient jusqu’à 60 k éléments logiques, 2 Mb de SRAM et 36 blocs DSP permettant d’implanter jusqu’à 144 multiplieurs 18b x 18b. La fréquence d’horloge du processeur est 50 MHz. Les codes évalués sont compilés avec l’environnement de développement NIOS IDE.

5. Résultats expérimentaux

| Intitulé | Regroupement opéré |
|-----------------|--|
| <i>Basique</i> | (GradXv,GradXh) (GradYv,GradYh) (ProduitXY) (CarreX) (CarreY) (GaussXXv,GaussXXh) (GaussXYv,GaussXYh) (GaussYYv,GaussYYh) (Coarcity) |
| <i>Halfpipe</i> | (GradXv,GradXh,GradYv,GradYh,ProduitXY,CarreX,CarreY) (GaussXXv,GaussXXh,GaussXYv,GaussXYh,GaussYYv,GaussYYh,Coarcity) |
| <i>Fullpipe</i> | (GradXv,GradXh,GradYv,GradYh,ProduitXY,CarreX,CarreY, GaussXXv,GaussXXh,GaussXYv,GaussXYh,GaussYYv,GaussYYh,Coarcity) |
| Sym1 | (GradXv,GradXh,CarreX) (GradYv,GradYh,CarreY) (ProduitXY) (GaussXXv,GaussXXh) (GaussXYv,GaussXYh) (GaussYYv,GaussYYh) (Coarcity) |
| Sym2 | (GradXv,GradXh,GradYv,GradYh,ProduitXY,CarreX,CarreY) (GaussXXv,GaussXXh) (GaussXYv,GaussXYh) (GaussYYv,GaussYYh) (Coarcity) |
| Asym1 | (GradXv,GradXh,CarreX) (GradYv,GradYh,ProduitXY,CarreY) (GaussXYv,GaussXYh) (GaussYYv,GaussYYh) (GaussXXv,GaussXXh,Coarcity) |
| Asym2 | (GradXv,GradXh,GradYv,GradYh,ProduitXY,CarreX,CarreY) (GaussXYv,GaussXYh) (GaussYYv,GaussYYh) (GaussXXv,GaussXXh,Coarcity) |

TABLE 6 – Liste des principaux regroupements intéressantes, architectures confondues

La table 6 présente quelques uns des regroupements possibles des opérateurs du filtre de Harris.

- La version *Basique* représente une version naïve telle qu’elle pourrait être écrite manuellement. Elle nous servira de référence pour évaluer la performances des versions fusionnées et optimisées.
- La version *Fullpipe* regroupe tous les calculs dans une même double-boucle. Elle nous permettra d’éventuellement observer les effets d’une surfusion.
- Les autres versions sont celles qui se sont révélées parmi les plus efficaces lors d’une exploration exhaustive des regroupements. Nous n’en présentons ici que quatre par pédagogie, mais l’utilisateur peut choisir de toutes les tester, ou de ne tester qu’un échantillon dicté par son expérience. Notamment la version *Halfpipe* est celle connue pour donner les meilleurs performances lors d’un codage manuel.

5.1. Résultats sur GPP

Sur les deux architectures considérées, l’outil permet de considérablement diminuer le temps de calcul par point. Sur le PowerPC qui dispose d’un banc de registres plus important, c’est la version entièrement fusionnée (*Fullpipe*) qui obtient en moyenne les meilleures performances. Tandis que le processeur Intel obtient le meilleur score pour un découpage en deux parties équilibrées du graphe de calcul (version *Halfpipe*). Dans les deux cas, le gain en performances est fortement accentué par la montée en taille des images. En effet, la fusion permet de diminuer l’empreinte cache des noyaux de calcul générés. Les meilleures versions stockent peu ou pas de points intermédiaires en mémoire, qui peuvent donc rester en mémoire cache L2. Il en est de même pour l’image de sortie produite par le filtre, qui en environnement réel est consommée par la suite par un autre filtre.

| Regroupement | Optimisation | Xeon | | PowerPC | |
|-----------------|--------------|--------------|---------------|---------------|---------------|
| | | N=512 | N=1024 | N=512 | N=1024 |
| <i>Basique</i> | aucune | 78.2 (× 1.0) | 103.8 (× 1.0) | 247.5 (× 1.0) | 254.7 (× 1.0) |
| | rotations | 61.3 (× 1.3) | 84.2 (× 1.2) | 93.4 (× 2.7) | 99.7 (× 2.6) |
| | déroulage | 59.9 (× 1.3) | 84.0 (× 1.2) | 92.5 (× 2.7) | 98.6 (× 2.6) |
| <i>Halfpipe</i> | aucune | 48.9 (× 1.6) | 66.4 (× 1.6) | 170.5 (× 1.5) | 178.7 (× 1.4) |
| | rotations | 40.1 (× 2.0) | 57.0 (× 1.8) | 104.7 (× 2.4) | 114.9 (× 2.2) |
| | déroulage | 38.5 (× 2.0) | 55.7 (× 1.9) | 106.7 (× 2.3) | 116.7 (× 2.2) |
| <i>Fullpipe</i> | aucune | 63.3 (× 1.2) | 67.1 (× 1.6) | 102.7 (× 2.4) | 104.5 (× 2.4) |
| | rotations | 59.6 (× 1.3) | 62.6 (× 1.7) | 47.5 (× 5.2) | 48.2 (× 5.3) |
| | déroulage | 48.9 (× 1.6) | 51.7 (× 2.0) | 42.9 (× 5.8) | 43.7 (× 5.8) |
| Sym1 | aucune | 76.3 (× 1.0) | 101.8 (× 1.0) | 254.1 (× 1.0) | 262.3 (× 1.0) |
| | rotations | 60.3 (× 1.3) | 80.4 (× 1.3) | 106.7 (× 2.3) | 116.7 (× 2.2) |
| | déroulage | 59.3 (× 1.3) | 80.1 (× 1.3) | 105.4 (× 2.4) | 115.5 (× 2.2) |
| Sym2 | aucune | 64.4 (× 1.2) | 78.7 (× 1.3) | 224.7 (× 1.1) | 228.8 (× 1.1) |
| | rotations | 46.2 (× 1.7) | 61.1 (× 1.7) | 88.0 (× 2.8) | 95.2 (× 2.7) |
| | déroulage | 44.2 (× 1.8) | 60.9 (× 1.7) | 86.7 (× 2.9) | 93.7 (× 2.7) |
| Asym1 | aucune | 68.1 (× 1.2) | 80.2 (× 1.3) | 252.5 (× 1.0) | 259.5 (× 1.0) |
| | rotations | 47.8 (× 1.6) | 64.8 (× 1.6) | 106.6 (× 2.3) | 116.9 (× 2.2) |
| | déroulage | 47.2 (× 1.7) | 64.6 (× 1.6) | 103.8 (× 2.4) | 114.1 (× 2.2) |
| Asym2 | aucune | 59.4 (× 1.3) | 67.5 (× 1.5) | 230.7 (× 1.1) | 235.4 (× 1.1) |
| | rotations | 41.0 (× 1.9) | 53.7 (× 1.9) | 95.3 (× 2.6) | 104.7 (× 2.4) |
| | déroulage | 41.1 (× 1.9) | 53.8 (× 1.9) | 92.5 (× 2.7) | 101.6 (× 2.5) |

TABLE 7 – Performances des implémentations de Harris sur GPP
Calcul en flottant simple précision - entre parenthèses, l'accélération par rapport à la version basique
Le partitionnement retenu est celui en gris.

5.2. Résultats sur processeur embarqué

| Regroupement | Optimisation | CPP | accél. |
|-----------------|--------------|--------|----------|
| <i>Fullpipe</i> | rotations | 216.87 | (× 2.15) |
| Asym2 | déroulage | 266.78 | (× 1.75) |
| Asym2 | déroulage | 266.90 | (× 1.75) |
| Asym2 | rotations | 288.45 | (× 1.62) |
| Sym2 | rotations | 288.57 | (× 1.62) |
| Sym2 | déroulage | 288.69 | (× 1.62) |
| Asym1 | déroulage | 303.68 | (× 1.54) |
| <i>Basique</i> | aucune | 466.96 | (× 1.00) |

| Regroupement | Optimisation | CPP | accél. |
|-----------------|--------------|--------|----------|
| <i>Halfpipe</i> | déroulage | 308.46 | (× 1.51) |
| Sym2 | déroulage | 311.86 | (× 1.50) |
| Sym2 | rotations | 311.93 | (× 1.50) |
| <i>Halfpipe</i> | rotations | 318.62 | (× 1.47) |
| Asym1 | rotations | 330.90 | (× 1.41) |
| Asym2 | aucune | 336.22 | (× 1.39) |
| Sym2 | aucune | 356.58 | (× 1.31) |
| <i>Basique</i> | aucune | 466.96 | (× 1.00) |

TABLE 8 – Performance des 14 meilleurs versions sur NiosII
En calcul entier
Accélération donnée par rapport au partitionnement basique sans optimisation

Par rapport à la version naïve initiale du code qui a un CPP de 467, les performances des 14 meilleures configurations sur NiosII pour une image de taille 128x128 sont données dans la table 8. On constate que la meilleure configuration est la version en une passe utilisant la variante de la rotation de registres. Les accélérations obtenues sont supérieures à 2 par rapport à la version initiale. Les résultats ne sont pas très différents pour d'autres tailles d'images. Nous avons testé les tailles 256x256 et 132x132 pour éventuellement mettre en évidence l'impact de la correspondance directe du cache de données avec des tailles puissance de 2.

Il faut souligner ici que les configurations testées n'utilisent pas l'approche SIMD. Les pixels de l'image initiale (en niveaux de gris) sont codés avec des octets non signés et les formats entiers utilisés (16 bits ou 32 bits) dépendent de la précision nécessaire des calculs. Au moment où cet article est rédigé, nous travaillons sur des versions pseudo-SIMD des différents noyaux de calcul utilisé par l'outil dans

la recherche des meilleurs regroupements. Le jeu d'instructions NiosII ne comprend pas d'instructions SIMD, et les accès mémoire sont des accès 32 bits.

Il est cependant possible d'utiliser le logiciel de compilation de matériel C2H fourni par Altera, qui permet d'accélérer des boucles écrites en C. Il est donc possible d'écrire des fonctions accédant simultanément à des groupes de 4 pixels via des accès 32 bits normaux, et d'écrire 4 fois le code nécessaire au traitement des pixels individuels du mot de 32 bits, en extrayant chaque pixel à l'aide de masques et de décalage. L'intérêt serait limité au niveau logiciel puisque les 4 traitements seraient séquentialisés : la seule différence notable serait de diminuer le nombre d'accès cache, ce qui peut être significatif dans le cas de cache à correspondance directe lorsque les accès se font avec des pas qui sont des puissances de deux.

Dans le cas de la compilation de matériel, la situation est différente puisque les extractions d'octets et les décalages sont gratuits (réalisées via les connexions) et qu'il y a vraiment traitement en parallèle des données correspondant à chaque octet des mots de 32 bits. On a donc un traitement pseudo-SIMD, avec le fait que les calculs parallèles intermédiaires se font avec le strict nombre de bits nécessaires à la précision des calculs. Dans le cas de fonctions utilisant des calculs pseudo-SIMD, l'efficacité des regroupements peut être complètement différente de celle des regroupements utilisant des calculs scalaires. Nous présenterons les résultats de cette approche « pseudo-SIMD », actuellement en cours de développement, dans la version finale de l'article.

L'ordre de grandeur du gain obtenu dépend bien évidemment des performances de la version initiale. A titre d'exemple, les accélérations obtenues sur NiosII pour la version en flottant 32 bits ne sont pas significatives.

6. Conclusion et perspectives

Cet outil de génération de code est particulièrement efficace car les transformations automatiques permettent un gain en performances d'un facteur deux. La connaissance du domaine d'application permet de surpasser les performances obtenues uniquement par l'utilisation d'un compilateur optimisant (ICC ou XLC) sur un code basique non fusionné. Cet outil est particulièrement intéressant car ne nécessite aucune expérience en compilation pour un traiteur d'image.

Afin de prendre en compte les évolutions des architectures actuelles (multicoeur avec SIMD), une nouvelle version de l'outil est en cours de conception. Concernant le parallélisme de *thread* pour le multicoeur SPMD, cela peut être fait simplement sur GPP via OpenMP. Concernant l'utilisation d'instructions SIMD, il faut non seulement modifier le granularité des opérateurs et des accès mémoires, mais en plus avoir des règles spécifiques de transformation pour gérer les constructions de mélange intra-registre (`vec_perm` sur Altivec, `mm_shuffle_ps` en SSE) dans le cas d'opérateurs 2D décrits en 1D avec réduction par colonne.

Bibliographie

1. D. Demigny *et al*, *Méthodes et architectures pour le TSI en temps réel*, Chapitre 8 : L. Lacassagne, F. Lohier, P. Garda, "Méthodologie d'optimisation logicielle pour microprocesseurs superscalaire RISC et CISC", pp169-191, Hermes collection IC2, ISBN 2-7462-0327-8, 2001.
2. L. Lacassagne, L. Lohier, P. Garda, *Real time execution of optimal edge detectors on RISC and DSP processors*, ICASSP pp 3101-3104, 1998
3. S. Pop, A. Cohen, C. Bastoul, S. Girbal : *Graphite : Polyhedral Analyses and Optimizations for GCC*. GCC 2008.
4. S. Piskorski, L. Lacassagne, S. Bouaziz, D. Etiemble, *Customizing CPU instructions for embedded vision systems*, IEEE CAMPS Computer Architecture, Machine Perception and Sensors, 2006
5. <http://gcc.gnu.org/wiki/Graphite>
6. M. Püschel *et al*, *SPiRAL : Code Generation for DSP Transforms*, Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation", Vol. 93, No. 2, pp. 232-275, 2005
7. D. Etiemble, S. Piskorski, L. Lacassagne, *Performance evaluation of Altera C2H compiler on image processing benchmarks*, TCHA : Workshop on Tools And Compiler for Hardware Acceleration, 17 septembre, 2006, Seattle
8. M. Frigo, *A fast Fourier transform compiler*, Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp169-180, 1999
9. R. Clint Whaley, A. Petitet, et J. Dongarra, *Automated Empirical Optimization of Software and the ATLAS Project*, Parallel Computing, pp3-35, 2001