Génération automatique d'architectures de calcul pour des opérations linéaires : application à l'IDCT sur FPGA

Nicolas Boullis, Arnaud Tisserand

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Génération automatique d'architectures de calcul pour des opérations linéaires : application à l'IDCT sur FPGA

Nicolas Boullis et Arnaud Tisserand

N° 4486

Mars 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4486--FR

Génération automatique d'architectures de calcul pour des opérations linéaires : application à l'IDCT sur FPGA

Nicolas Boullis et Arnaud Tisserand

Thème 2 — Génie logiciel
et calcul symbolique
Projet Arénaire

Rapport de recherche n° 4486 — Mars 2002 — 10 pages

Résumé : Cet article présente une méthode de génération automatique d'opérateurs arithmétiques matériels pour des calculs basés sur des multiplications par des constantes et des additions. A partir d'un recodage des nombres et d'algorithmes particuliers de recherche de sous-expressions communes, on arrive à diminuer sensiblement la surface des opérateurs réalisés. Cette méthode à été implantée dans un générateur de code VHDL et testée dans le cas de l'IDCT sur des FPGA de la famille Virtex de Xilinx. Sur cette application particulière, on note un facteur de 5,8 sur l'amélioration du produit surface $\times$ délai.

Mots-clés : génération automatique d'opérateurs matériels de calcul, multiplication par des constantes, opérations linéaires, DCT, FPGA.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Automatic generation of linear operations architectures: application to IDCT on FPGA

Abstract: This article presents an automatic generation method for hardware operators based on multiplications by constants and additions. Using number recoding and decated common sub-expressions factorisation algorithms, one can significantly reduce the area of the generated operators. This method was implemented into a VHDL generator and tested on an IDCT application on Virtex FPGA from Xilinx. For this specific application, a 5.8 improvement factor was obtained for the area$\times$delay product.

Key-words: automatic generation of hardware arithmetic operators, multiplication by constants, linear operators, DCT, FPGA.

1 Introduction

De nombreux calculs courants, en particulier en traitement du signal ou en multimédia, peuvent s'écrire sous forme de multiplications par des constantes et d'additions. C'est le cas par exemple de nombreuses transformations comme la transformée de FOURIER ou la transformée en cosinus, des filtres, des convolutions... Mathématiquement, ces opérations peuvent s'écrire sous la forme d'une multiplication d'un vecteur d'entrée par une matrice constante. On parle donc d'opérations linéaires.

Il est souvent utile de réaliser de telles opérations en matériel. Les opérateurs réalisant des opérations linéaires seront appelés opérateurs linéaires. Etant donné que toutes les multiplications en jeu sont des multiplications par des constantes, il est intéressant, en terme du produit surface $\times$ délai, de les implémenter par des multiplieurs par des constantes plutôt que par des multiplieurs généralistes. De même, il est alors intéressant d'essayer de combiner certains calculs, pour partager au maximum des expressions communes entre plusieurs parties. On améliore alors encore le produit surface $\times$ délai en réduisant la « quantité de calcul » effectuée.

Le problème de la multiplication par une constante a été étudié depuis longtemps, comme par exemple avec le célèbre recodage de BOOTH [1], ou plus récemment dans l'article de LEFÈVRE [3]. Il existe de nombreux algorithmes pour la multiplication par une constante, ou des problèmes proches comme l'élévation à une puissance constante. Différentes techniques visant à découper une multiplication en additions sont connues dans la bibliographie sous le terme *"distributed arithmetic"*. En revanche, le problème de la multiplication par une matrice constante a été nettement moins étudié, mais certaines idées se retrouvent par exemple dans les articles [4] et [5]. Historiquement, de telles méthodes ont été utilisées dans les compilateurs pour des processeurs cibles ne disposant pas d'un multiplieur entier efficace.

Ce travail consiste en le développement d'un outil d'exploration et de génération automatique d'opérateurs linéaires. A partir d'une description du résultat du calcul à effectuer, le programme optimise, à l'aide de plusieurs améliorations de l'algorithme de LEFÈVRE, l'opérateur de calcul. La description de l'opérateur optimisé peut ensuite être traduite automatiquement en VHDL synthétisable. Les programmes réalisés ont été utilisés pour des opérations de DCT/IDCT. Les descriptions VHDL obtenues ont été synthétisées sur des FPGA Virtex de Xilinx. L'objectif principal est une diminution significative de la « quantité de calcul ». Les résultats semblent prometteurs; en particulier nous avons un gain important pour le produit surface $\times$ délai par rapport à une réalisation proposée par Xilinx dans [2].

Nous allons tout d'abord présenter ici quelques algorithmes classiques pour la multiplication par une constante, et l'algorithme de LEFÈVRE. Nous présenterons ensuite des améliorations de cet algorithme, et l'implémentation que nous en avons faite. Enfin, nous étudierons l'application de cet algorithme à l'IDCT pour FPGA, et comparerons nos résultats avec d'autres de la litérature.

2 Algorithmes de multiplication par des constantes

Pour des implantations matérielles, il peut être avantageux d'implanter des multiplications par des constantes entières par des additions (et soustractions) et des décalages. Le principe s'applique tout naturellement aux opérateurs linéaires à coefficients entiers, l'extension aux opérateurs linéaires à coefficients à virgule fixe étant alors triviale. Bien que l'idée soit simple, la mise en œuvre efficace est loin d'être évidente, du fait de la combinatoire du problème.

Nous décrivons ici quelques algorithmes de multiplication d'un nombre entier par une constante entière. Les premiers s'étendent trivialement aux opérateurs linéaires à coefficients entiers, ce qui n'est pas le cas de l'algorithme de BERNSTEIN.

Au cours de cette présentation des différents algorithmes, la notation $x \ll k$ désigne le nombre x décalé de k bits vers la gauche, c'est-à-dire multiplié par 2^k, tous les nombres étants représentés en base 2. En outre, la présentation des algorithmes s'appuiera sur un exemple unique: le calcul de p comme produit de la variable d'entrée x par la constante $c = 111463 = 11011001101100111_2$. Enfin, les opérations d'addition et de soustraction étant équivalentes, nous compterons systématiquement les soustractions comme des additions.

2.1 Les algorithmes simples

L'algorithme le plus simple consiste à écrire la ou les constantes en base 2, et à utiliser la distributivité de la multiplication. Ainsi, si l'écriture binaire de la constante c contient n occurrences du chiffre 1, la multiplication par c se fera en $n-1$ additions. Ainsi, dans notre exemple, le calcul se fait en 11 additions:

$$111463x = x \ll 16 + x \ll 15 + x \ll 13 + x \ll 12 + x \ll 9 + x \ll 8 + x \ll 6 + x \ll 5 + x \ll 2 + x \ll 1 + x$$

Une amélioration simple consiste à utiliser le recodage canonique de BOOTH [1] de la constante. Il s'agit d'un recodage en base 2, avec l'ensemble de chiffres $\{\overline{1},0,1\}$, où $\overline{1} = -1$, qui minimise le nombre de chiffres non nul. Pour notre exemple, on a $111463 = 11011001101100111_2 = 100\overline{1}0\overline{1}0100\overline{1}0\overline{1}0100\overline{1}_2$ et le calcul se fait alors en 8 additions:

$$111463x = x \ll 17 - x \ll 14 - x \ll 12 + x \ll 10 - x \ll 7 - x \ll 5 + x \ll 3 - x$$

2.2 L'algorithme de BERNSTEIN

Avec les algorithmes précédents, la variable d'entrée est décalée plusieurs fois (multipliée par des puissances de 2), puis les résultats de ces décalages sont tous additionnés. Chaque valeur calculée ne sert donc qu'une seule fois. Un progrès serait donc de permettre la réutilisation de valeurs intermédiaires. Par exemple, pour calculer $q = 165x = 10100101_2\, x$, il serait avantageux de calculer $z = 5x = 101_2\, x$ puis $q = 33z = 100001_2\, z$, réduisant ainsi le nombre d'additions de 3 à 2.

L'algorithme de BERNSTEIN permet des recodages de ce type. Son principe est de rechercher, par une exploration d'arbre, le meilleur recodage d'une constante, en utilisant des opérations élémentaires. Le coût de chacune de ces opérations peut être précisé, pour orienter les résultats en fonction de paramètres technologiques. Les opérations permises sont les suivantes:

- décalage $t_{i+1} = t_i \ll k$;
- addition de la variable d'entrée $t_{i+1} = t_i \pm x$;
- addition d'un nombre avec lui-même après décalage $t_{i+1} = t_i \ll k \pm t_i$.

Pour notre exemple $p = c \times x$, l'algorithme de BERNSTEIN trouvera la solution suivante, en 5 additions:

$$\begin{aligned} t1 &= ((x \ll 3 - x) \ll 2) - x \\ t2 &= t1 \ll 7 + t1 \\ p &= (((t2 \ll 2) + x) \ll 3) - x \end{aligned}$$

Malheureusement, cet algorithme ne permet pas de partage de sous-expressions communes entre plusieurs résultats. En outre, la solution construite reste très linéaire: une valeur calculée n'est utilisée que pour calculer la valeur suivante. Ceci peut limiter la capacité de l'algorithme à trouver les meilleurs recodages, et ne permet pas d'utiliser le parallélisme inhérent aux implantations matérielles. En outre, la recherche du meilleur recodage est une opération très longue, probablement trop pour être applicable aux opérateurs linéaires.

2.3 L'algorithme de LEFÈVRE

L'algorithme de LEFÈVRE permet de trouver des recodages où une même valeur peut être réutilisée « à volonté ». En outre, cet algorithme est directement conçu pour la multiplication d'une variable par un jeu de constantes, en permettant des partages de sous-expressions communes entre les différentes constantes. Seul le principe de fonctionnement simplifié de cet algorithme est donné ici, on pourra se référer à l'article de LEFÈVRE [3] pour plus de détails.

Avant d'exposer l'algorithme, quelques définitions préliminaires sont nécessaires. Un motif est une suite de chiffres $0,1,\overline{1}$. On dit qu'un motif apparaît dans une constante avec un décalage k si, en décalant le motif de k vers la gauche, on a un 1 dans la constante en face de chaque 1 du motif et un $\overline{1}$ en face de chaque $\overline{1}$. On définit aussi le poids d'un motif comme le nombre de chiffres non nuls du motif. Par exemple, le motif $1000\overline{1}$ est de poids 2, et il apparaît deux fois dans la constante $84 = 1010\overline{1}0\overline{1}_2$ (avec des décalages de 0 et de 2). En outre, un motif peut apparaître dans une constante sous sa forme complémentée. Ainsi, le motif 101 apparaît aussi deux fois dans la constante $84 = 1010\overline{1}0\overline{1}_2$ (décalé de 4, et sous forme négative décalé de 0).

Le principe de l'algorithme de LEFÈVRE est de rechercher un motif commun, de poids maximal, dans les écritures des différentes constantes. Ce motif doit apparaître plusieurs fois, soit à l'intérieur d'une même constante, soit dans des constantes différentes. On parle alors de motif maximal. Ce motif est alors soustrait des endroits où il apparaît, et rajouté comme nouvelle constante. L'algorithme recommence alors jusqu'à ce qu'il n'y ait plus de motif de poids supérieur ou égal à 2.

Sur notre exemple, en partant de la constante $111463 = 11011001101100111_2 = 100\overline{1}0\overline{1}0100\overline{1}0\overline{1}0100\overline{1}_2$, l'algorithme de LEFÈVRE trouve tout d'abord le motif $100\overline{1}0\overline{1}$, avec des décalages de 5 et 12. Après soustraction du motif, il reste les deux constantes $7 = 100\overline{1}_2$ et $26 = 100\overline{1}0\overline{1}_2$. L'algorithme trouve ensuite le motif commun $100\overline{1}$, et finit avec les deux constantes $7 = 100\overline{1}_2$ et $1 = 1_2$. Ainsi, l'algorithme de LEFÈVRE trouve cette décomposition, en seulement 4 additions:

$$t1 = (x \ll 3) - (x \ll 0)$$
$$t2 = (t1 \ll 2) - (x \ll 0)$$
$$p = (t2 \ll 12) + (t2 \ll 5) + (t1 \ll 0)$$

3 Algorithmes proposés

Notre idée est d'utiliser l'algorithme de LEFÈVRE pour optimiser des opérateurs linéaires. Cet algorithme conçu pour optimiser des multiplications d'une variable par une ou plusieurs constantes s'adapte facilement aux opérations linéaires. Mais quelques améliorations peuvent être proposées.

Une première amélioration consiste à ne pas rechercher un motif de poids maximal à chaque itération. En effet, lorsqu'on soustrait un motif maximal, on est sûr de ne pas générer de nouveau motif du même poids. Au lieu de rechercher un nouveau motif maximal à chaque fois, on peut donc rechercher tous les motifs maximaux, et les garder « en réserve ». Ainsi, pour un poids de motif donné, on cherche tous les motifs de ce poids une fois pour toutes. On peut alors économiser de nombreuses recherches de motifs, ce qui permet une accélération considérable de l'algorithme. On gagne ainsi, en temps d'exécution de l'algorithme, un facteur de l'ordre de 10. En plus, le fait de considérer tous les motifs d'un poids donné permet d'avoir une vision plus globale, ce qui permettra d'autres améliorations, comme nous allons le voir par la suite.

3.1 Améliorations de l'algorithme de LEFÈVRE

L'algorithme de LEFÈVRE souffre d'une lacune en ce qui concerne le choix du motif de poids maximal. En effet, il y a souvent plusieurs motifs possibles qui ont le même poids maximal. Le choix entre ces motifs n'est pas spécifié dans l'algorithme de LEFÈVRE. Il a été observé que différentes implémentations de cet algorithme conduisent parfois à des résultats différents. Il pourrait donc être intéressant d'avoir une heuristique permettant de choisir le « bon » motif.

Ainsi, plusieurs heuristiques ont été testées. Il semblait difficile de déterminer complètement le choix d'un motif maximal. Du coup, lorsque plusieurs possibilités semblent équivalentes, le choix est aléatoire. Ainsi, avec un grand nombre d'essais, des statistiques peuvent être établies, dans le but de déterminer la meilleure heuristique.

Dans la description de ces heuristiques, on parle de conflit entre deux motifs lorsqu'ils annulent le même chiffre d'une constante. Ainsi, s'il y a un conflit entre deux motifs, il sera impossible d'appliquer les deux; réciproquement, s'il n'y a pas de conflit, les deux seront applicables simultanément. Considérons, par exemple, le nombre $85 = 10\overline{1}010\overline{1}_2$; on y trouve les motifs maximaux $M_1 = 10\overline{1}$ et $M_2 = 10001$, comme indiqué en figure 1 avec leurs conflits.

La première heuristique, la plus proche de l'algorithme original, consiste simplement à choisir l'un des motifs de poids maximal au hasard. Cette heuristique est appelée par la suite « aléatoire ».

Les deux autres heuristiques construisent le graphe des conflits de tous les motifs de poids maximal et en extraient un sous-ensemble indépendant maximal (au sens de l'inclusion). L'heuristique dite « graphe-heuristique » choisit systématiquement l'un des motifs qui a le moins de conflits, puis le retire, ainsi que tous les motifs avec lesquels il est en conflit.

Enfin, l'heuristique dite « graphe-optimal » construit tous les sous-ensembles indépendants maximaux et choisit l'un de ceux qui permettent le plus grand nombre d'applications de motifs. Ainsi, en reprenant

Constante:		$10\overline{1}010\overline{1}$
Motifs:	$M_1 =$	$10\overline{1}$
	$M_2 =$	10001
Occurences:		$10\overline{1}010\overline{1}$
	$(M_1,0,+)$	$10\overline{1}$
	$(M_1,2,-)$	$\overline{1}01$
	$(M_1,4,+)$	$10\overline{1}$
	$(M_2,0,-)$	$\overline{1}000\overline{1}$
	$(M_2,2,+)$	10001

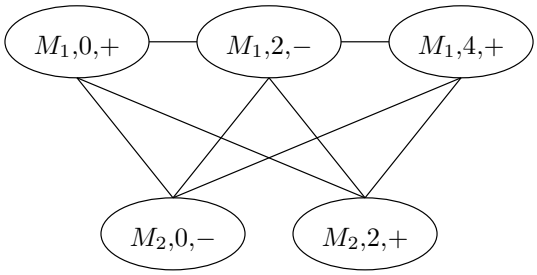

FIG. 1 – *Motifs maximaux et conflits dans la constante* $85 = 10\overline{1}010\overline{1}_2$.

l'exemple de 85, on trouve 3 sous-ensembles indépendants maximaux: le premier contient seulement le motif M_1, sous sa forme négative, décalé de 2 bits; le deuxième contient le motif M_1, sous sa forme positive, décalé de 0 et 4 bits; le troisième contient le motif M_2, sous sa forme positive, décalé de 2 bits, et sous sa forme négative décalé de 0 bit. L'heuristique « graphe-optimal » choisira alors le deuxième ou le troisième des sous-ensembles indépendants maximaux.

3.2 Implémentation

Ces différentes améliorations de l'algorithme de LEFÈVRE ont toutes été implémentées en C++. Un système de *plugin* permet, à l'exécution, de générer plusieurs types de sorties. Il est ainsi possible de générer du code VHDL synthétisable pour un opérateur totalement combinatoire, ou un opérateur série avec différentes bases possibles. Il est aussi possible de générer un programme C++ permettant d'utiliser plusieurs *plugin* de sortie pour le même opérateur sans devoir recommencer l'optimisation.

Comme les différentes améliorations considérées de l'algorithme de LEFÈVRE gardent toutes une partie aléatoire, il est intéressant, pour un même opérateur, et avec une même version de l'algorithme, de l'optimiser plusieurs fois. On génère alors un nouveau programme pour chaque optimisation. Il ne reste alors plus qu'à choisir les programmes qui utilisent le plus petit nombre d'additions, et à les utiliser pour générer des codes VHDL à synthétiser.

Cette phase fait que les solutions qui ont le plus petit nombre d'opérations ne sont pas forcément exactement celles qui conduiront aux opérateurs les plus petits, en particulier à cause de la taille des nombres considérés, et des optimisations du synthétiseur. C'est pour cela qu'il est préférable de ne pas choisir une unique solution avant la synthèse; il vaut mieux en conserver plusieurs, et choisir la meilleure après synthèse, à condition que les temps de synthèse soient raisonnables.

4 Application à l'IDCT sur FPGA

Nous avons choisi d'appliquer notre méthode à des opérations d'IDCT sur FPGA. La transformée en cosinus discrète (DCT) et sa transformée inverse (IDCT) sont utilisées dans un grand nombre d'algorithmes de compression avec pertes d'image ou de son, comme pour les formats JPEG, MPEG ou MP3. L'IDCT est définie par la relation ci-dessous. Les coefficients de sa forme matricielle sont indiqués en table 1. On y remarque certaines régularités évidentes, en particulier dans la première et la quatrième colonnes; ces régularités sont parfaitement détectées et exploitées par notre méthode.

4.1 Résultats avant synthèse

L'étude des résultats avant synthèse (la métrique utilisée est le nombre d'additions) permet tout d'abord de comparer les différentes améliorations de l'algorithme de LEFÈVRE. Elle permet aussi de comparer cet algorithme avec celui proposé par POTKONJAK, SRIVASTAVA et CHANDRAKASAN dans [5].

$$IDCT[i] = c_j \sum_{j=0}^{n-1} x_j \cos \frac{(2i+1)\, j\pi}{2n}$$

$$\text{avec} \quad c_j = \begin{cases} \frac{1}{\sqrt{n}} & \text{si } j = 0 \\ \frac{2}{\sqrt{2n}} & \text{si } j \neq 0 \end{cases}$$

0.354	0.490	0.462	0.416	0.354	0.278	0.191	0.098
0.354	0.416	0.191	-0.098	-0.354	-0.490	-0.462	-0.278
0.354	0.278	-0.191	-0.490	-0.354	0.098	0.462	0.416
0.354	0.098	-0.462	-0.278	0.354	0.416	-0.191	-0.490
0.354	-0.098	-0.462	0.278	0.354	-0.416	-0.191	0.490
0.354	-0.278	-0.191	0.490	-0.354	-0.098	0.462	-0.416
0.354	-0.416	0.191	0.098	-0.354	0.490	-0.462	0.278
0.354	-0.490	0.462	-0.416	0.354	-0.278	0.191	-0.098

TAB. 1 – *Formules de l'IDCT 1D et coefficients de l'IDCT 1D à 8 points.*

La comparaison des différentes heuristiques a été faite sur l'IDCT bidimensionnelle en 8×8 points, et en utilisant 14 bits fractionnaires dans l'écriture des coefficients. Les tailles relevées vont de 1060 à 1147 additions, ce qui représente une différence d'environ de 8%. Les courbes en figure 2 représentent les distributions des tailles obtenues avec les différentes versions de l'algorithme: on mesure le pourcentage des opérateurs générés qui ont une certaine taille en nombre d'additions. On remarque que l'algorithme « aléatoire » donne les plus mauvais résultats, alors que les deux autres sont équivalents.

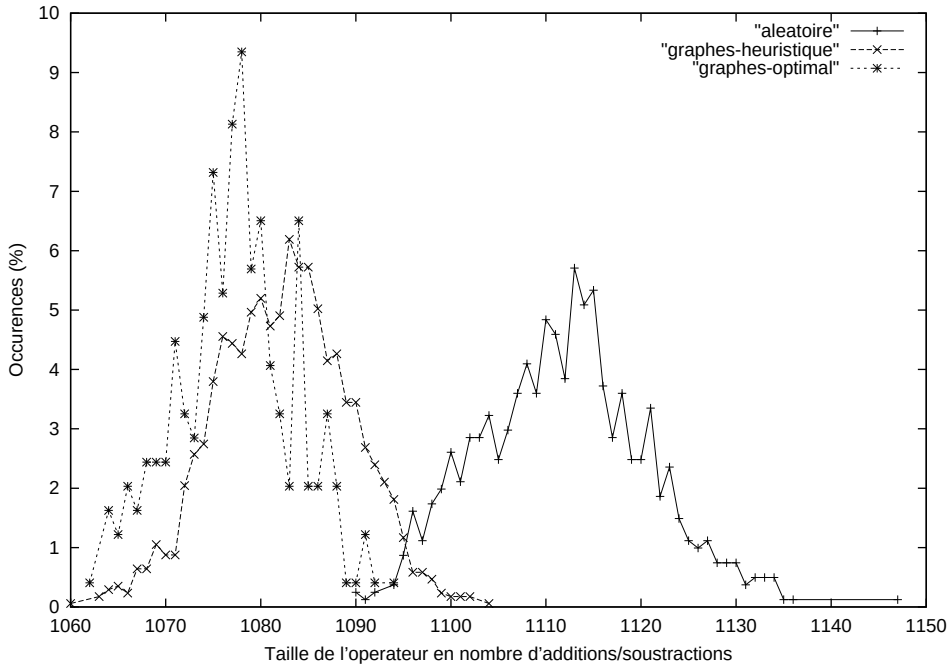

FIG. 2 – *Distribution des tailles obtenues avec les différents algorithmes. (Sur un grand nombre d'optimisations d'un même opérateur, on mesure le pourcentage d'opérateurs qui ont une taille donnée.)*

Le temps de calcul des différents algorithmes est présenté en table 2 dans le cas de la génération d'un opérateur d'IDCT 2D à 8×8 points. On constate que le gain en vitesse de calcul est supérieur à un ordre de grandeur en pratique. Le temps d'exécution de l'algorithme « graphe-optimal » est très variable suivant les cas. Quelques exécutions ont été interrompues après plusieurs heures de calcul, d'où la valeur symbolique ∞.

En revanche, aucune étude théorique de la complexité des différents algorithmes n'a été menée. En effet, le temps de calcul dépend grandement des entrées, et une telle étude nécessiterait donc des modélisations probabilistes très complexes.

Nous avons aussi voulu comparer nos résultats avec ceux de la méthode MCM, présentée dans [5], qui utilise des idées similaires. Parmi les résultats présentés, on trouve les nombres d'additions obtenus pour calculer une DCT en 8 points avec 8, 12, 16 et 24 bits fractionnaires. Nous avons essayé d'optimiser les mêmes opérateurs avec l'algorithme de LEFÈVRE, étendu par notre heuristique « aléatoire », pour comparer les deux

version	LEFÈVRE	aléatoire	graphe-heuristique	graphe-optimal
temps de calcul (min)	15	1	1	$3 \rightarrow \infty$

TAB. 2 – *Temps d'exécution des différents algorithmes pour une IDCT 2D.*

méthodes. Les résultats sont indiqués dans la table 3. Ces résultats nous montrent bien l'intérêt de l'algorithme de LEFÈVRE étendu qui bat en moyenne la méthode MCM d'un facteur de l'ordre de 1,5. En revanche, le temps de calcul de l'algorithme MCM n'étant pas connu, nous ne pouvons pas comparer le temps de calcul de ces deux méthodes.

nombre de bits fractionnaires	nombre d'additions		facteur d'amélioration
	MCM	LEFÈVRE	
8	94	58	1,62
12	100	70	1,43
16	129	90	1,43
24	212	120	1,77

TAB. 3 – *Comparaison de la méthode MCM avec l'algorithme de* LEFÈVRE*, étendu par notre heuristique « aléatoire », pour la DCT 1D à 8 points.*

4.2 Résultats après synthèse

Les résultats suivants ont été obtenus en synthétisant, avec le logiciel Leonardo (version 2001), et sur une machine Ultra Sparc à 440 MHz et 1 Go de RAM, les fichiers VHDL générés automatiquement. Le placement/routage est effectué par les outils Xilinx, appelés automatiquement par Leonardo; le temps de sythème indiqué comprend donc le temps de placement/routage. Le FPGA cible des synthèses est un XCV1000 de la famille Virtex de Xilinx. Pour information, ce type de FPGA contient une matrice de 64×96 CLB et chaque CLB contient 4 LUT (soit 24576 LUT au total). Dans les données des tables ci-dessous, nous indiquons le nombre de LUT utilisées, ainsi que le pourcentage de surface du Virtex XCV1000 utilisée. Les CLB sont les blocs logiques de base d'un FPGA, et les LUT sont de toutes petites tables utilisées dans les CLB.

La table 4 présente les résultats pour l'IDCT à une dimension pour différentes versions séries et une parallèle. Plusieurs bases de calcul ont été testées pour les versions séries. On constate que l'opérateur réalisé est petit même pour la version parallèle où il ne représente que 5% de la surface totale du FPGA. Les vitesses obtenues permettent de faire tourner le circuit entre 20 et 25 MHz suivant les versions. Pour la version parallèle, il faut remarquer que la version générée actuellement est complètement combinatoire et que, en pipelinant, le temps de cycle doit pouvoir être réduit. Par ailleurs, les temps de synthèse (incluant le placement/routage) sont très faibles.

Version	taille [LUT]	utilisation du FPGA	délais [ns]	temps synthèse [s]
parallèle	1227	5,0%	40	22
série base 2	170	0,7%	22	16
série base 4	305	1,2%	36	15
série base 8	387	1,6%	46	17
série base 16	484	2,0%	47	20
série base 64	697	2,8%	47	26
série base 256	891	3,6%	48	31

TAB. 4 – *Résultats de synthèse pour l'IDCT 1D à 8 points.*

La table 5 présente les résultats de synthèse pour la version à deux dimensions de l'IDCT 8×8. Les opérateurs générés sont bien plus gros mais ils restent implantables en pratique. Le délai augmente par rapport à

la version à une dimension mais ici aussi des techniques de pipeline peuvent être utilisées pour augmenter la fréquence de fonctionnement. Les temps de synthèse augmentent beaucoup mais sont à corréler au fait que le circuit est nettement plus complexe, et qu'ils restent modérés par rapport à d'autres types de circuits de taille équivalente mais beaucoup plus complexes en contrôle.

Version	taille [LUT]	utilisation du FPGA	délais [ns]	temps synthèse
parallèle	17047	69,0%	64	7 min 5 s
série base 16	6518	26,5%	60	15 min 6 s

TAB. 5 – *Résultats de synthèse pour l'IDCT 2D à* 8×8 *points.*

A titre de comparaison, on peut regarder les résultats de Xilinx publiés dans une *"application note"* [2] sur ce même sujet. Ils implantent une IDCT 2D sur un Virtex XCV600 (en utilisant le synthétiseur de Synplify) sur une surface de 13234 LUT pour une fréquence de fonctionnement de 53,2 MHz et 489 ns de latence. Nos résultats sont donc 1,3 fois plus gros mais 7,6 fois plus rapides pour la version parallèle, soit un gain de 5,8 sur le produit surface $\times$ délai. Afin d'être totalement objectif sur les avantages de notre méthode, il faudrait utiliser les mêmes outils de CAO. Cependant, sur des opérateurs simples, ces différents outils donnent généralement des résultats similaires.

5 Conclusion et travaux futurs

L'optimisation d'opérateurs arithmétiques pour certaines applications a été abordée, à travers la réécriture de multiplications par des constantes sous forme de suites d'additions. En particulier, nous avons développé un générateur de code de VHDL synthétisable pour les opérateurs linéaires. Ce générateur a été utilisé efficacement dans le cas de l'IDCT, et nous comptons l'étendre prochainement à d'autres types de calculs.

Les travaux présentés ici ont montré que l'algorithme de LEFÈVRE, avec nos extensions, est un algorithme de choix pour l'optimisation d'opérateurs linéaires. Il permet d'obtenir un bon produit surface $\times$ délai. Nous pensons étendre ces travaux au cadre des circuits ASIC. En outre, comme la « quantité de calcul » totale est réduite, on peut fortement espérer un gain significatif de la consommation d'énergie. Ceci est important pour les appareils électroniques portatifs, comme, par exemple, les appareils photo numériques ou les lecteurs de MP3. Cet aspect sera étudié prochainement.

En outre, d'autres améliorations futures sont envisageables. Une première piste qui peut être explorée est de modifier le codage des nombres. En effet, dans sa version actuelle, l'algorithme de LEFÈVRE utilise le codage canonique de BOOTH. Cependant, en utilisant d'autres codages, les motifs trouvés seront différents. On peut donc espérer que certains codages permettent d'obtenir de meilleurs résultats.

Enfin, une autre voie possible reste à explorer: les valeurs des coefficients. En effet, pour des opérations comme la DCT ou l'IDCT, les coefficients sont des valeurs réelles, que l'on arrondit (au plus proche) à un certain nombre de bits fractionnaires. Or, dans ce genre de situation, ce n'est pas vraiment le nombre de bits qui importe, mais plutôt l'erreur sur les coefficients. Une idée à explorer serait donc de trouver, pour chaque coefficient, la « bonne » valeur, qui ne dépasse pas une certaine erreur, mais qui n'utilise pas forcément un nombre minimal de bits. A défaut d'une meilleur méthode, cette « bonne » valeur pourrait être trouvée par une recherche aléatoire. Dans ce cas, et pour limiter l'étendue de la recherche, il pourrait être bon de trouver des propriétés mathématiques intrinsèques des coefficients qu'il faudrait préserver. Par exemple, si deux coefficients sont égaux ou opposés, il y a de fortes chances que cette égalité soit utilisée par l'algorithme de LEFÈVRE. Une telle propriété devrait donc être préservée.

6 Remerciements

Nous tenons à remercier tout particulièrement Vincent LEFÈVRE pour les discussions intéressantes à propos de son algorithme.

Nous tenons aussi à remercier chaleureusement le Ministère de la Recherche Français pour son support avec l'ACI Jeunes Chercheurs et la société Xilinx pour son don de circuits FPGA via le *"Xilinx University Program"*.

Références

[1] A. D. BOOTH – "A signed binary multiplication technique", *Quarterly Journal of Mechanics and Applied Mathematics* **4** (1951), no. 2, p. 236–240.

[2] K. CHAUDHARY, H. VERMA & S. NAG – "An inverse discrete cosine transform (IDCT) implementation in Virtex for MPEG video applications", Application note, Xilinx, décembre 1999, `http://www.xilinx.com/xapp/xapp208.pdf`.

[3] V. LEFÈVRE – "Multiplication par une constante", *Réseaux et Systèmes Répartis, Calculateurs Parallèles: numéro spécial sur l'arithmétique des ordinateurs* **13** (2001), no. 4–5, p. 465–484.

[4] M. MELLAL & J.-M. DELOSME – "Multiplier optimization for small sets of coefficients", in *International Workshop Logic and Architecture Synthesis*, décembre 1997, p. 13–22.

[5] M. POTKONJAK, M. B. SRIVASTAVA & A. P. CHANDRAKASAN – "Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **15** (1996), no. 2, p. 151–165.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399