CamlG4: une bibliothèque de calcul parallèle pour Objective Caml

J. Falcou[1] & J. Sérot[2]

1: ISIMA,Campus des Cézeaux, 63177 Aubière CEDEX
`joel.falcou@poste.isima.fr`
2: LASMEA - UMR 6602 CNRS/UBP, Campus des Cézeaux, 63177 Aubière CEDEX
`Jocelyn.Serot@lasmea.univ-bpclermont.fr`

Résumé

Ce papier présente CAMLG4, une bibliothèque de calcul parallèle pour Objective Caml destinée aux plateformes PowerPC G4. Cette bibliothèque fournit une interface de haut-niveau aux fonctionnalités offertes par l'unité de calcul vectoriel *AltiVec* intégrée à cette famille de processeurs. L'utilisation de CAMLG4 permet d'obtenir, moyennant un effort modéré de reformulation, des gains en performances significatifs (de 2 à 7 typiquement) pour des applications de calcul numérique intensif. On présente ici les objectifs de CAMLG4, les principaux traits relatifs à son implantation et les premiers résultats obtenus sur un ensemble d'exemples.

1. La Technologie AltiVec

L'unité de traitement AltiVec [1, 3] a été développée par la firme Motorola dans le cadre du projet Apple Velocity Engine [2] en 1995. Le but du projet était la mise en œuvre des principes du parallélisme SIMD[1] au sein de processeurs destinés au marché grand public. Cette intégration devait ainsi augmenter notablement et pour un faible coût les performances des microprocesseurs équipant les micro-ordinateurs de la marque Apple. La première implantation d'AltiVec fut réalisée dans le processeur MPC 7400 (G4). Initialement dédié au seul traitement multimédia[2], l'AltiVec s'est imposé comme un outil très bien adapté à de nombreuses tâches de traitement du signal.

L'AltiVec est une unité de calcul vectoriel opérant en mode SIMD. Cette unité est distincte des unités de calcul scalaire et flottante, possède son propre banc de registres (32), sa propre unité de lecture/écriture mémoire et peut fonctionner concurremment avec les unités scalaires[3] (cf. figure. 1). Les instructions AltiVec opèrent sur des *vecteurs* (`vector`) de 128 bits, chaque vecteur pouvant contenir, au choix :

- 4 flottants 32 bits,
- 4 entiers 32 bits (signés ou non),
- 8 entiers 16 bits (signés ou non),
- 16 entiers 8 bits (signés ou non),
- 128 booléens,
- 8 pixels au format RGB (sur 16 bits) ou
- 4 pixels au format RGBA (sur 32 bits).

[1]Single Instruction Multiple Data.
[2]A l'instar des extensions MMX, puis SSE du Pentium.
[3]Sans requérir de mécanisme de commutation de contexte, comme avec le MMX du Pentium.

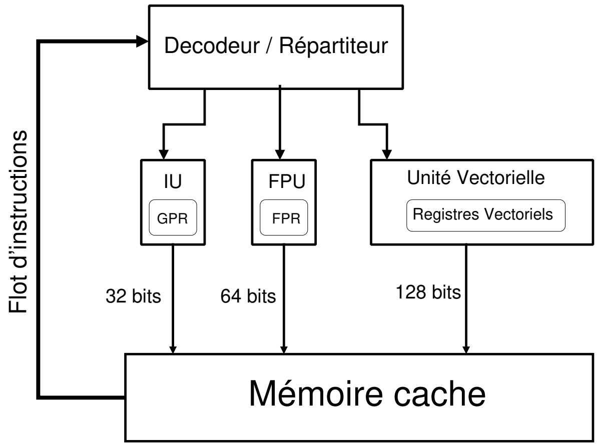


FIG. 1 – Synoptique du processeur PPC 74xx

Une instruction AltiVec opérant sur un vecteur effectue simultanément[4] une même opération sur tous les éléments de ce vecteur (Fig. 2). Les accélérations maximales théoriques, en fonction du type de données traitées, vont donc de 4 à 16. Le jeu d'instructions AltiVec comporte 162 instructions couvrant, entre autres, les opérations arithmétiques et logiques inter et intra-vecteurs, les opérations de comparaison et les opérations de manipulation du cache.

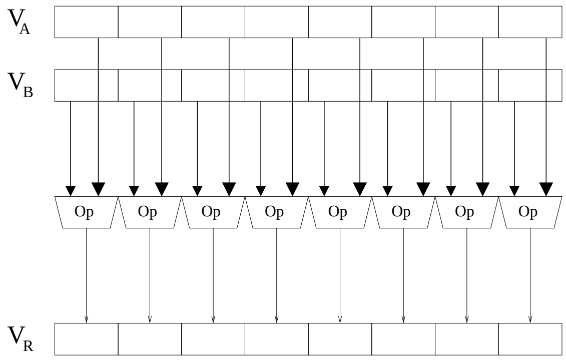


FIG. 2 – Schéma de principe des instructions AltiVec

1.1. Programmer l'AltiVec

Une particularité intéressante de l'AltiVec est de proposer une interface de programmation non seulement en assembleur PowerPC mais aussi en langage évolué (C, C++ et Fortran). Pour le langage C, Motorola et Apple fournissent ainsi une version modifiée du compilateur *gcc* supportant, sans recours à aucune bibliothèque supplémentaire, les types de données et la quasi-totalité des instructions AltiVec sous la formes de fonctions C[5]. C'est cette particularité — qui découle du fait qu'unités scalaires et vectorielle fonctionnent de manière indépendante dans le PowerPC G4 — qui rend possible le développement de bibliothèques de calcul vectoriel sous forme de modules Caml externes — via le mécanisme d'appel de fonctions externes écrites en C — sans avoir à modifier le générateur de code.

[4]En 1 à 5 cycles, selon la complexité de l'instruction.

[5]Les fonctions AltiVec C sont en fait directement *inlinées* en assembleur dans le code final, ce qui n'ajoute aucun surcoût par rapport à une programmation directement en assembleur.

A titre d'exemple, considérons une fonction calculant le produit scalaire de deux vecteurs de quatre entiers 32 bits. Une implantation en C de cette fonction est donnée figure 3.

```
long DotProduct(long* A, long* B)
{
   long R = 0;
   for( i = 0; i < 4; i++ ) R += A[i] * B[i];
   return R;
}
```

FIG. 3 – Calcul d'un produit scalaire en C

La même fonction, ré-écrite en C AltiVec apparaît sur la figure 4.

```
signed long DotProduct(vector signed long A, vector signed long B)
{
   /* Création de deux vecteurs R[1..4] et T[1..4]  initialisés à 0 */
   register vector signed long R = vec_splat_s32(0);
   register vector signed long T = vec_splat_s32(0);

   /* T := [ T1+A1*B1 ; T2+A2*B2 ; T3+A3*B3 ; T4+A4*B4 ]; */
   T = vec_madd( A, B, T );

   /* R := R + T1 + T2 + T3 + T4; */
   R = vec_sums( T, R );

   return R[3];
}
```

FIG. 4 – Calcul d'un produit scalaire en C AltiVec

La fonction C requiert 4 additions, 4 multiplications, 4 écritures en mémoire et 12 lectures en mémoire. Son équivalent AltiVec ne nécessite que 2 opérations de création de vecteur (instruction `vec_splat`) et 2 opérations de calcul vectoriel : `vec_madd` (*multiply-add*) et `vec_sums` (*sum-across*). Le gain en performance est donc significatif. Dans le cas où la taille des vecteurs est supérieure à quatre[6], le code AltiVec procède en itérant sur les vecteurs. Le nombre d'itérations étant alors divisé par 4, 8 ou 16 par rapport à la version C originale[7].

La reformulation des traitements afin de tirer parti de l'unité AltiVec, telle que nous l'avons esquissée dans l'exemple précédent – délibérément très simplifié –, même si elle reste considérablement plus simple qu'avec d'autres extensions SIMD (comme MMX), n'est toutefois pas une opération triviale. D'où l'idée de proposer une interface haut niveau aux fonctionnalités offertes par l'AltiVec en *encapsulant* ces reformulations. L'intérêt d'utiliser pour cela un langage fonctionnel comme Objective Caml [4] est alors double.

D'une part, et à court terme, cette approche doit permettre d'accélérer de manière significative certaines applications écrites en Caml et s'exécutant sur plateforme G4. Dans cette optique, le module CAMLG4 décrit ici, s'inscrit dans la même démarche que celle qui conduirait à recourir à des fonctions externes écrites en C pour accroître les performances d'une application, mais avec une différence

[6] Ou à 8 pour des entiers 16 bits, ou 16 pour des entiers 8 bits.

[7] Il est en fait souvent possible, compte-tenu des particularités de l'unité de lecture/écriture mémoire, d'améliorer encore les performances en dépliant d'un facteur 2 à 6 les boucles vectorielles.

significative : le programmeur n'a plus ici à écrire de code C, ni à se soucier de son interfacage avec Caml (gestion des représentations mémoire des données, coopération avec le *garbage collector* notamment).

D'autre part, et dans un optique méthodologique à plus long terme, l'encapsulation de fonctionnalités SIMD au sein d'un module Caml doit permettre d'aborder la problématique de la *parallélisation* de code séquentiel de manière plus systématique qu'avec une formulation en C, en tirant parti de fonctionnalités propres aux langages fonctionnels comme la pleine fonctionnalité et le polymorphisme. On reviendra sur ce point au paragraphe 4.

2. Implantation

Deux questions ont guidé les choix d'implantation du module CamlG4 :

1. Sous quelle forme les données vectorielles doivent elles être représentées en Caml ?
2. Quelles fonctionnalités de l'AltiVec ce module doit il offrir (« faire remonter ») ?

2.1. Représentation des données vectorielles

Les deux principaux critères à prendre en compte sont ici :
- d'un point de vue « ergonomique », ne pas trop s'écarter des structures de données utilisées traditionnellement pour le calcul numérique en Caml,
- d'un point de vue efficacité, minimiser les coûts liés aux conversions des représentations mémoire lorsque l'on passe du coté C au coté Caml et *vice versa*.

Le premier critère plaide en faveur d'un « clone » du module `Array`. Le second en faveur d'une variante du module `Bigarray`, ce dernier permettant d'éviter les (coûteuses) recopies mémoire lors du changement d'espace mémoire[8]. Dans notre cas, le choix du module `Bigarray` s'est imposé, l'utilisation efficace des instructions AltiVec requérant un contrôle précis et strict du *mapping* mémoire des données vectorielles. Les instructions de lecture/écriture de vecteurs en mémoire supposent en effet que les données accédées aient leur adresse alignée sur un multiple de seize[9]. Par ailleurs, afin de tirer parti des spécificités du cache du G4 – et d'améliorer ainsi les performances – les routines C encapsulées dans le module CamlG4 utilisent massivement le dépliage de boucles. Afin de faciliter ce mécanisme, les tableaux à traiter sont toujours créés avec une taille multiple du facteur de dépliage[10]. Ces contraintes d'alignement peuvent être aisément prises en compte en modifiant légèrement le code effectuant l'allocation mémoire au sein du module `Bigarray`, comme indiqué sur la figure 5. Cette modification intègre le calcul de l'espace réellement nécessaire ainsi que l'utilisation de la commande `memalign` en lieu et place de `malloc` afin d'allouer un bloc de mémoire aligné et de taille multiple d'un facteur donné.

Le module `Bigarray` (dans sa version modifiée, présentée ci-dessus) est donc utilisé comme conteneur de données pour le module CamlG4. Cette représentation permet d'éviter complètement les recopies mémoire lors du passage des données vectorielles entre le code Caml de l'application et les fonctions C effectuant des traitements AltiVec sur ces données.

Pour le programmeur d'application, ces données vectorielles appartiennent à un type `Vecarray` qui se présente, du point de vue de la création et l'accès aux éléments, comme un clone du type

[8]Rappelons que si les tableaux de flottants sont bien représentés sous forme *unboxed* en Caml, ce n'est pas le cas des tableaux d'entiers, qui nécessitent une indirection par élément, et ne peuvent donc pas être passés « en bloc » aux fonctions AltiVec.

[9]On peut ne pas respecter cette contrainte, mais il faut alors procéder manuellement au ré-alignement par décalage, ce qui grève de manière significative les performances de l'unité.

[10]Cette contrainte d'alignement supplémentaire permet de ne pas avoir à traiter de cas particuliers, donc de régulariser le code, et par là d'améliorer les performances.

```
/* Calcul original de la taille */
size   = num_elts;
size  *=  bigarray_element_size[flags & BIGARRAY_KIND_MASK];

/* Taille ajustée  */
new_size = ( size & ~63 ) + 64;

/* Allocation alignée */
data     = memalign(16,new_size);
```

FIG. 5 – Modification du module Bigarray

Bigarray. En particulier, les types des données que l'on peut mettre dans un Vecarray appartiennent à un ensemble restreint, sous-ensemble de celui associé aux Bigarrays[11]. La figure 6 résume les correspondances possibles, compte-tenu des possibilités de l'unité de calcul AltiVec.

Types Bigarray	Type Vecarray
float32	float32
float64	n/a
complex32	n/a
complex64	n/a
char	intu8
int8_signed	ints8
int8_unsigned	intu8
int16_signed	ints16
int16_unsigned	intu16
int32	ints32
int	ints32
int64	n/a
nativeint	n/a

FIG. 6 – Correspondance entre types de données pour Bigarray et Vecarray

A titre d'exemple, la création d'un vecteur de 1024 entiers 8 bits signés se fait de la manière suivante avec CAMLG4 :

```
let a = Vecarray.create Vecarray.ints8 1024
```

2.2. Fonctionnalités

La question du choix des fonctionnalités associées au type Vecarray est légitime car il n'est pas raisonnable (ni même souhaitable) de « faire remonter » telles quelles au niveau de l'interface Caml les 162 fonctions AltiVec. Notre démarche a donc été guidée par deux principes.

D'une part, masquer les dissymétries du jeu d'instruction AltiVec. Certaines instructions ne sont en effet disponibles que pour un type de vecteur particulier. C'est le cas par exemple de l'instruction de multiplication qui n'existe directement que pour des flottants 32 bits[12]. Il est donc nécessaire de

[11] Ceci explique pourquoi certains types supportés par AltiVec (booléens et pixels notamment) ne se retrouvent pas dans la table de la figure 6. Leur émulation avec les types fournis ne pose toutefois aucun problème.

[12] La multiplication d'entiers est effectuée en se ramenant à une multiplication flottante.

mettre en place un mécanisme d'émulation afin d'offrir au programmeur un ensemble d'opérations opérant – sauf impossibilité théorique - sur tous les types de vecteurs de la bibliothèque.

D'autre part, éliminer les instructions trop spécifiques ou dont la sémantique n'admet pas de projection en Caml. Il s'agit en particulier des certaines instructions arithmétiques gérant des configurations spécifiques de vecteurs ou destinées à des algorithmes précis, des instructions de comparaison-sélection vectorielles et des instructions de manipulation du cache.

Le jeu d'opération proposé par CAMLG4 est résumé dans le tableau de la figure 7. L'annexe A donne par ailleurs un extrait de la signature du module CAMLG4.

Types de fonctions	Exemples
Création et manipulation	`create`, `of_array`, `fill`, ...
Opération arithmétiques et logiques	`add`, `scale`, `land`, `lsl`, ...
Décalages vectoriels	`svl`, `svr`, ...
Calculs d'approximations.	`invsqrt`, `trunc`, ...
Utilisation de table d'équivalence.	`lookup`

FIG. 7 – Aperçu du jeu de fonctions CAMLG4

3. Utilisation

On donne dans ce paragraphe quelques exemples très simples d'utilisation de CAMLG4. Le but est ici d'illustrer le style de programmation offert par cette bibliothèque et de donner une idée des gains obtenus (accélérations) par rapport à une programmation en Caml « standard ». Toutes les mesures ont été obtenues sur un PowerPC G4 à 450 MHz, disposant de 64M de mémoire, sous Linux 2.4.18 (Yellow Dog). Les programmes ont été compilés en mode natif avec Objective Caml 3.06. Le code C de CAMLG4 a été compilé avec le compilateur `gcc-altivec` version 2.95.2 avec les options `-fvec -fno-schedule-insns`. Pour chaque programme test, les résultats reportés ont été obtenus en moyennant les mesures obtenus sur un grand nombre d'exécutions (10000) afin de s'affranchir d'éventuelles fluctuations dues à la charge du système.

3.1. Traitements élémentaires

Chaque programme test n'utilise ici qu'une seule fonction CAMLG4. Les deux versions de chaque programme, avec et sans CAMLG4, apparaissent sur la figure 8.

Les accélérations obtenues pour différentes tailles de tableau (valeur de `sz` sur la figure 8) et différents types de données (la figure 8 ne traite que le cas des tableaux de flottants) sont reportées sur la figure 9. La figure 9a met en évidence le comportement asymptotique de CAMLG4, l'accélération observée tendant vers une valeur limite lorsque la taille des tableaux traités augmente. Cette accélération maximale est voisine de 7 pour des entiers 8 bits, de 4 pour des entiers 16 bits et de 2 pour des entiers ou des flottants 32 bits, soit, approximativement, la moitié de l'accélération maximale théorique (qui vaut respectivement 16, 8 et 4 pour les types sus-cités). Pour comparaison, la figure 10a donne les accélérations obtenues pour des traitements et des types de données équivalents écrits en C et en C+AltiVec. On remarquera que, *modulo* un facteur constant d'environ 1.8 – qui correspond grossièrement au surcout introduit par CAMLG4–, les deux jeux de courbes présentent des allures remarquablement similaires. La figure 9b est un agrandissement de la première partie de la figure 9a montrant le comportement de CAMLG4 pour des tableaux de faible taille. Il apparaît clairement que l'usage de notre bibliothèque n'est intéressant que pour des tableaux de taille supérieure à quelques Ko. En dessous de cette valeur, le surcoût du aux mécanismes d'interfaçage Caml / C / AltiVec devient prépondérant. Cet effet est bien entendu nettement moins marqué avec la version C,

Caml	Camlg4

`let sz = ...`
`let a = Array.make sz 10.0`
`let b = Array.make sz 10.0`
`let c = Array.make sz 0.0`
	`let sz = ...`
	`let a = Vecarray.create int_s16 sz`
	`let b = Vecarray.create int_s16 sz`
	`let c = Vecarray.create float_32 sz`
	
	`Vecarray.fill a 10.0`
	`Vecarray.fill b 10.0`

Test 1. Addition membre à membre	

`for i = 0 to sz-1 do`
`  c.(i) = a.(i) +. b.(i)`
`done;;`
	`let c = Vecarray.add a b`

Test 2. Multiplication membre à membre	

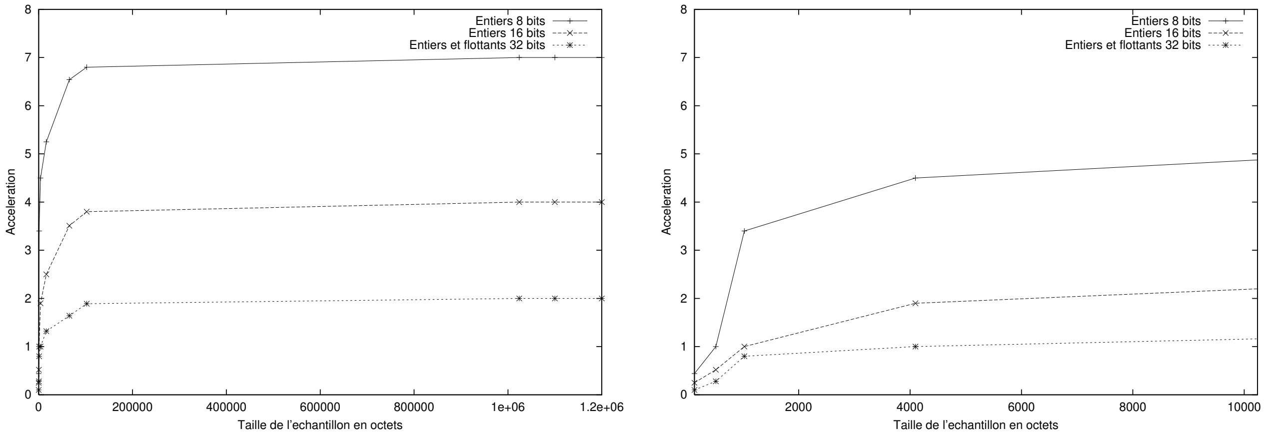
`for i = 0 to sz-1 do`
`  c.(i) = a.(i) *. b.(i)`
`done`
	`let c = Vecarray.mul a b`

Test 3. Calcul de l'inverse	

`for i = 0 to sz-1 do`
`  c.(i) = 1.0 /. a.(i)`
`done`
	`let c = Vecarray.inv a`

FIG. 8 – Programmes tests élémentaires

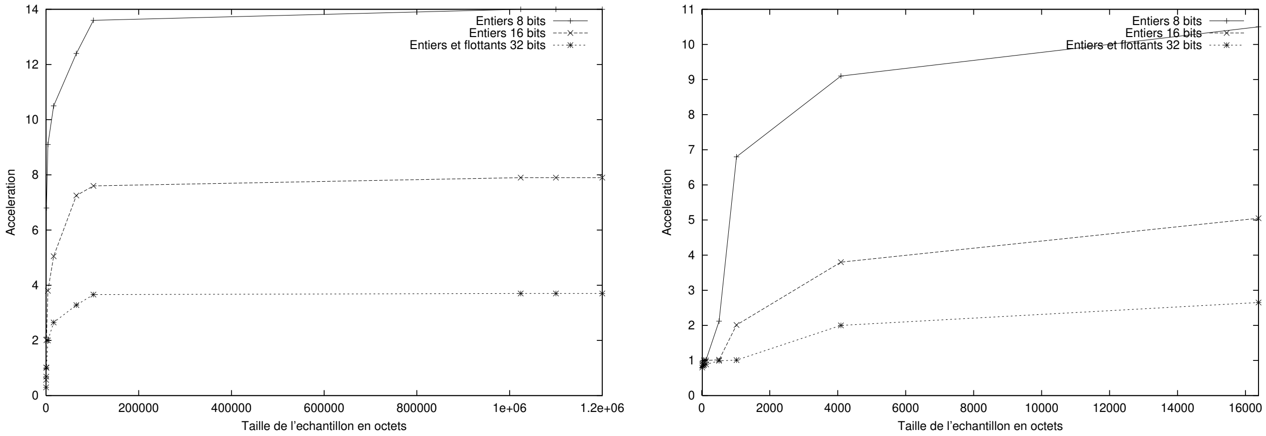
comme le montre la figure 10b. Un chronométrage fin permet d'estimer ces coûts d'interfaçage : le décodage des arguments et l'aiguillage vers la fonction AltiVec prend de l'ordre de 6 μs et la création du `Vecarray` résultat de l'ordre de 5 μs. Les temps de calcul AltiVec proprement dit et de création du tableau C résultat (pour les tests reportés ici) sont donnés dans le tableau 11. Nous pensons que le « saut » observé pour le temps de calcul AltiVec entre 1000 et 10000 est probablement du aux effets de cache, mais de disposant pas d'outils d'analyse de trace d'exécution, nous n'avons pu confirmer cette hypothèse[13]



(a) (b)

FIG. 9 – Accélérations CAMLG4 *vs.* CAML pour les tests élémentaires

[13]De tels outils sont disponibles avec les outils de développement de Mac OS, mais pas sous Linux.



(a) (b)

Fig. 10 – Accélérations C+AltiVec *vs.* C pour les tests élémentaires

Nombre d'éléments traités	Calcul AltiVec	Création du tableau C
10	2	1
100	4	9
1000	8	14
10000	110	30
100000	950	37
1000000	11000	42
10000000	125000	82

Fig. 11 – Temps de calcul AltiVec (μs)

3.2. Filtre RIF

On décrit ici une implémentation, avec CamlG4, d'un filtre monodimensionnel à réponse impulsionnelle finie d'ordre 3. Il s'agit d'une opération très fréquemment utilisée en traitement de signal. Son implantation met en jeu plusieurs fonctions élémentaires (3 multiplications, 2 additions et 2 décalages). Ce test nous permet donc d'évaluer le comportement de notre bibliothèque dans un contexte plus réaliste que les tests de la section précédente. Les résultats sont donnés ici pour des données formées d'entiers 8 bits. Le code source (avec et sans AltiVec) apparaît sur la figure 12. Les fonctions `Vecarray.scale`, `Vecarray.svr` et `Vecarray.svl` permettent respectivement de multiplier un vecteur par un scalaire et de décaler les éléments d'un vecteur de n éléments vers la droite et vers la gauche.

L'accélération mesurée est donnée sur la figure 13. Elle est de l'ordre de 5 dès que la tableau traité contient plus de 16K échantillons, ce qui confirme que la composition, au sein d'un même programme, de plusieurs fonctions CamlG4, ne dégrade pas trop les performances obtenues pour une seule fonction. Pour comparaison, l'accélération obtenue en déportant la boucle de calcul dans une fonction externe écrite en C (sans AltiVec), interfacée via `Bigarray`, est voisine de 1,8. La plus-value offerte par AltiVec peut donc efficacement exploitée depuis Objective Caml. Signalons toutefois que cette accélération – de CamlG4 par rapport à Caml, donc – est ici sensiblement plus faible que celle mesurée pour le même filtre écrit en C (avec et sans AltiVec), cette dernière variant de 6 pour un tableau de 1K à 13 pour un tableau de 10K. Enfin, la manière de coder les opérations dans le code CamlG4 semble toutefois avoir une influence sensible sur ces performances. Ainsi, si on laisse le compilateur Caml

<table>
<tr><th>Caml</th><th>CamlG4</th></tr>
<tr><td><pre>
let sz = ...
let c1,c2,c3 = ...

let s = Array.make sz 0
let r = Array.make sz 0;

for i = 1 to sz-2 do
  r.(i) = c1*s.(i-1)+c2*s.(i)+c3*s.(i+1)
done
</pre></td><td><pre>
let sz = ...
let c1,c2,c3 = ...

let s = create int_s8 sz

let r1 = Vecarray.svl s 1
let r2 = Vecarray.scale c1 r1
let r3 = Vecarray.scale c2 s
let r4 = Vecarray.svr s
let r5 = Vecarray.scale c3 r4
let r6 = Vecarray.add r2 r3
let r = Vecarray.add r6 r5
</pre></td></tr>
</table>

FIG. 12 – Programme de test 2

gérer tout seul les variables intermédiaires, en écrivant par exemple

```
let r2 = Vecarray.scale c1 (Vecarray.svl s 1)
```

ou

```
let r = Vecarray.add (r2 (Vecarray add r3 r5))
```

l'accélération peut diminuer nettement (jusqu'à un facteur 2 dans certains cas[14]). Nous pensons que cet effet est lié au comportement du cache, dont l'influence est très sensible sur les performances de l'unité AltiVec. Il semblerait qu'un enchaînement explicite des calculs comme dans le code de la figure 12 favorise la localité des données et donc une exploitation optimale du cache[15].

3.3. Décodeur base64

Le dernier exemple présenté concerne un algorithme de décodage de fichiers encodés en base 64. Cet algorithme est utilisé principalement par les clients de courrier électronique pour permettre l'envoi et la réception de fichiers attachés. Lors de l'envoi d'un tel fichier, son contenu est accolé à la suite de l'éventuel message textuel. Or, dans le cas d'un fichier binaire, le flot de caractère correspondant s'interromprait à la première occurrence de la valeur 27 — indiquant en ASCII la fin de fichier — ou de la valeur 0 — indiquant la fin d'une chaîne de caractère. Pour éviter de tronquer ainsi les attachements, le fichier est encodé par l'algorithme dit de de BASE64 [5] pour éliminer du fichier transmis les caractères 27 ou 0. L'encodage procède par transformation de groupes de 24 bits contigus, chaque paquet de 3 fois 8 bits étant converti en 4 groupes de 6 bits. Une fois ce découpage effectué, les valeurs obtenues sont remplacées par des caractères « neutres » (différents de 0 et 27) via une table de correspondance.

L'algorithme décrit ici réalise le décodage de fichiers encodés selon le principe décrit ci-dessus. Il procède en trois étapes :

1. Détermination de la taille du fichier décodé. Cette taille est égale à 3/4 de la taille du fichier encodé.

[14]La baisse la plus sensible a été observée avec une formulation totalement infixée de l'algorithme, dans laquelle on écrivait directement `let r=(scale c1 (s<<<1 )+++(scale c2 s)+++(scale c3 (s>>>1)`, où `+++`, `>>>` et `<<<` sont les versions infixes des opérations `add`, `svr` et `svl`.

[15]Nous n'avons pu, en l'absence d'outil d'analyse de trace avec les outils de développement dont nous disposions, confirmer cette interprétation.

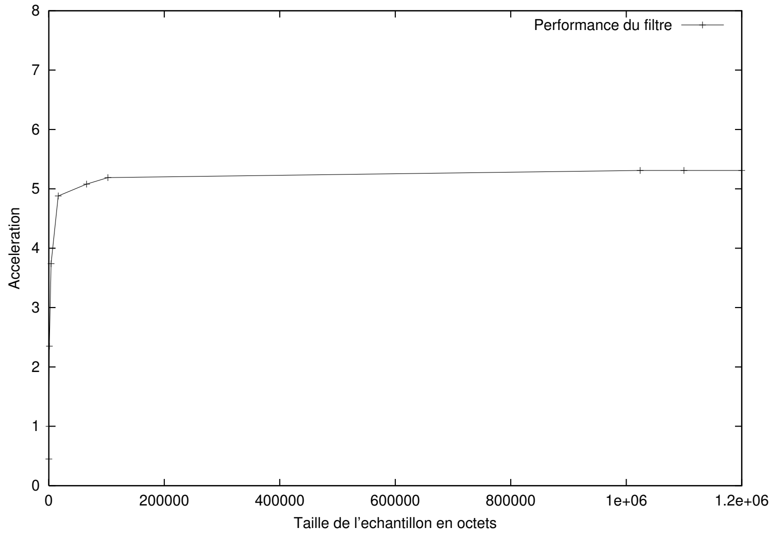


Fig. 13 – Accélérations mesurées pour le filtre

2. Décodage du fichier. Un parcours itératif est effectué au cours duquel chaque octet du fichier d'entrée est traité via une table de décodage. Une fois ce décodage effectué, le fichier est prêt à être reconstruit.

3. Reconstruction du fichier. Par une suite de décalage bits à bits et de masquage binaire, les groupes de 4 fois 6 bits sont remis en forme de 3 blocs de 8 bits.

L'implantation avec CamlG4 de l'algorithme de décodage suppose une reformulation SIMD de ce schéma. Comme pour l'exemple de la section précédente, ces formulations visent à remplacer les itérations par des opérations « en masse » sur des vecteurs. Les opérations effectués restent les mêmes (calculs, décodage, décalages et masquages) mais se déroulent à l'échelle du fichier – stocké dans un vecteur. Le code complet du décodeur ré-écrit avec CamlG4 est donné en annexe B.

Les performances mesurées sont résumées dans le tableau 3.3.

Taille du fichier	Accélération
128	0.20
512	0.48
1024	1
4096	1.85
16384	2.30
65536	3.51
102400	3.80
1024000	4.02

Fig. 14 – Performances de décodeur B64

4. Conclusion et perspectives

Nous avons présenté une bibliothèque de calcul parallèle en Objective Caml dédiée aux plateformes PowerPC G4[16]. Les premiers résultats obtenus avec cette bibliothèque sont encourageants, les gains en performances, par rapport à un code écrit en Caml seul, s'échelonnant entre 2 et 7 pour des applications simples de calcul numérique. Par comparaison, les gains obtenus par le seul recodage en C (sans AltiVec) des fonctions de calcul, appelées depuis Caml, n'excèdent jamais 2 pour ces applications. Un premier prolongement du travail décrit ici consiste à confirmer ces résultats sur des applications plus complexes : calcul matriciel ou traitement d'images, par exemple. Dans ces contextes, l'utilisation de CAMLG4 doit notamment être comparée avec l'usage direct de bibliothèque de routines spécialisées écrites en C[17]. Les deux approches ne se situent en effet pas au même niveau de « granularité » et ce niveau joue clairement sur le compromis performances / expressivité.

Nous pensons toutefois *a posteriori* que CAMLG4 n'a pas forcément vocation à être utilisée telle quelle par le programmeur d'application. Son utilisation suppose en effet une reformulation (vectorielle) des traitements. Si cette reformulation est assez immédiate pour des traitements simples, comme illustré aux sections 3.1 et 3.2, elle peut devenir moins triviale pour des algorithmes plus complexes, comme celui présenté à la section 3.3. Une solution consiste alors à accroître le niveau d'abstraction en offrant au programmeur, plutôt que des opérations élémentaires sur des vecteurs, des « squelettes » encapsulant des séquences complètes d'opérations et correspondant à des stratégies de calcul parallèle fréquemment utilisées. On rejoint là des problématiques de vectorisation (automatique ou semi-automatique) de code, problématiques qui font l'objet de nombreuses recherches par ailleurs. L'originalité d'une approche fondée CAMLG4 consisterait ici à aborder ces problèmes dans le cadre d'un formalisme fonctionnel, au sein duquel les « squelettes de parallélisation » s'interprètent naturellement comme des fonctions d'ordre supérieur. Nous avons déjà eu l'occasion de juger de l'intérêt d'une telle d'approche dans le cadre de la parallélisation d'applications sur machines parallèles à mémoire distribuée (MIMD-DM) [6]. Son application à la vectorisation de code pour AltiVec nous semble un axe de recherche intéressant. Dans cette optique, la bibliothèque CAMLG4 apparaîtrait comme un langage intermédiaire (une forme d' « assembleur ») pour un formalisme plus abstrait dans lequel ne figurerait que des schémas algorithmiques complets opérant sur des vecteurs.

Enfin, d'un point de vue plus technique, signalons que l'implantation de CAMLG4 a permis de confirmer l'intérêt et l'efficacité des mécanismes de coopération Caml / C fournis par Objective Caml. Dans notre cas, l'usage du module `Bigarray` a notamment permis de réduire le surcoût associé au code d'interface (*stub-code*) au strict minimum tout en autorisant la prise en compte des contraintes d'allocation mémoire imposées par l'AltiVec.

Références

[1] http ://www.simdtech.org/altivec

[2] http ://developer.apple.com/hardware/ve

[3] C. Hunter The AltiVec difference Published on the O'Reilly network http ://www.oreillynet.com/pub/a/mac/2002/04/05/altivec.htm

[4] X. Leroy and D. Doligez and J. Garrigue and D. Rémy and J. Vouillon The Objective Caml System release 3.05 - Documentation and user's manual http ://caml.inria.fr/ocaml/htmlman/

[5] N. Borenstein and N. Freed. Internet RFC/STD/FYI/BCP Archives. http ://www.faqs.org/rfcs/rfc1521.html, Septembre 1993.

[16] Un prototype de cette bibliothèque peut être obtenu à l'url suivante : `http ://wwwlasmea.univ-bpclermont.fr/Personnel/Jocelyn.Serot/camlg4.html`.

[17] Utilisant ou non AltiVec !

[6] J. Sérot and D. Ginhac Skeletons for parallel image processing : an overview of the SKiPPER project Parallel Computing, 28(12) :1785–1808, 2002

5. Annexe A. Un extrait de la signature du module Vecarray

Pour des raisons de place, seule les fonctions les plus représentatives sont reproduites ici, les fonctions « similaires » étant simplement listées.

```
module type Vecarray = sig

  type ('a, 'b) t

  type  ('a, 'b) kind = ('a, 'b) Bigarray.kind

  val create:  ('a, 'b) kind -> int -> ('a, 'b) t
    (* [create contents dim] cree un vecteur de type [contents]
        et de taille [dim]   *)

  val dim : ('a, 'b) t -> int
    (* [dim v] renvoie la taille (le nombre d'éléments) du vecteur [v] *)

  val get : ('a, 'b) t -> int -> 'a
    (* [get i v] renvoie l'élément d'index [i] du vecteur [v] *)

  val of_array : ('a, 'b) kind -> 'a array -> ('a, 'b ) t
    (* Crée un vecteur à partir d'un [array] *)

  external add: ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t = "CAMLG4add"
    (* [add v1 v2] effectue la somme, élément par élément des
       vecteurs [v1] et [v2] et renvoie le vecteur résultat.
       Fonctions similaires : [sub], [mul] *)

  external  land: ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t = "CAMLG4and"
    (* [land v1 v2] effectue le et logique, élément par élément des
       vecteurs [v1] et [v2] et renvoie le vecteur résultat.
       Fonctions similaires : [lor], [lxor], [lnot], [lnand], ... *)

  external round: ('a, 'b) t -> ('a, 'b) t = "CAMLG4round"
    (* [round v] calcule l'arrondi élément par élément du
       vecteur [v]. Fonctions similaires : [abs], [inv], [floor], ... *)

  external  svl: ('a, 'b) t -> int -> ('a, 'b) t = "CAMLG4shiftleft"
    (* [svl v n] décale le vecteur [v] de [n] éléments vers la gauche
       et renvoie le vecteur résultat. Fonctions similaires : [svr] *)

  external  lsl: ('a, 'b) t -> int -> ('a, 'b) t = "CAMLG4l_shiftleft"
    (* [lsl v n] décale chaque élément du vecteur [v] de [n] bits vers la gauche
       et renvoie le vecteur résultat. Fonctions similaires : [lsr] *)

  external  fill:    ('a, 'b) t ->'a -> unit = "CAMLG4fill"
```

```
    (* [fill v e] remplit le vecteur [v] avec la valeur [e] *)

  external  mask:    ('a, 'b) t -> int -> ('a,'b) t = "CAMLG4mask"
    (* [mask v n] retourne le vecteur [w] tel que [w(i)=v(i)] pour [i=0,k,2*j,...]
       et [w(i)=0] ailleurs *)

  external  lookup:   ('a, 'b) t -> ('a, 'b) t -> int -> ('a, 'b) t = "CAMLG4lookup"
    (* [lookup v1 v2 n] réalise la permutation du vecteur [v1] par le vecteur [v2].
       Si [r] est le vecteur résultat, on a [r(i)=v2(v1(i))].
       La taille du vecteur [v2] (passée en argument [n]) doit être égale à 32,
       64, 128 ou 256 *)
end
```

6. Annexe B. Code source du décodeur base64

```
(************************************************)
(*      Table de transcodage des octets.        *)
(************************************************)

let table = [
  62, 0, 0, 0, 0,63,52,53,54,55,56,57,58,59,60,61,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7,
   8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
  24,25, 0, 0, 0, 0, 0, 0,26,27,28,29,30,31,32,33,
  34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
  50,51, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0];;

(************************************************)
(*           Debut du decodeur base 64          *)
(************************************************)

let decode t =

  (* Allocation du tableau resultat *)

  let l_t = VecArray.dim t in
  if l_t mod 4 <> 0 then failwith "decodeB64";

  (* Determination du nombre de caracteres de padding *)

  let pad_chars =
    if l_t > 0 then begin
      if VecArray.get t (l_t - 2) = "=" && VecArray.get t (l_t - 2)  = "=" then 2
      else if VecArray.get t (l_t - 1) 1 = "=" then 1
      else 0
    end
    else 0 in

  (* Transformation de table en VecArray *)
```

```
let l_s = (l_t / 4) * 3 - pad_chars
and let avtable = VecArray.of_array table in

(* Resolution des equivalences des caracteres encodés *)

let c = VecArray.lookup t avtable 128 in

(* Récupération des octets k*4+1 par decalage et masquage successifs *)

let x0 = VecArray.lsl c 2
and x1 = VecArray.lsr c 4
and x2 = VecArray.lsl c 4
and x3 = VecArray.lsr c 2
and x4 = VecArray.lsl c 6 in

let r0 = VecArray.svl x1 1 in

let v0 = VecArray.lor x0 r0
and m1 = VecArray.fill l_t 0xF0 in

(* Récupération des octets (k*4)+2 *)

let r1 = VecArray.svl x2 1 in
let r2 = VecArray.land m1 r1 in
let v1 = vecArray.lor r2 x3
and m2 = VecArray.fill l_t 0xC0 in

let r3 = VecArray.svl x3 1 in
let r4 = VecArray.land m2 r3 in
let v2 = VecArray.lor r4 x4

(* Récuperation des octets (k*4)+0 *)

and m0 = VecArray.mask_odd v0 4 0
and m1 = VecArray.svr (VecArray.mask_odd v1 4 0) 1
and m2 = VecArray.svr (VecArray.mask_odd v2 4 0) 2 in

(* Récupération des octets (k*4)+3 *)

let m3 = VecArray.svl m0 1 in

(* Somme des resultats pour former le resultat final *)

let s0 = VecArray.add m0 m1
and s1 = VecArray.add m2 m3 in

(* Fin du decodage *)
let s = VecArray.add s0 s1;
s
```