Accélération SIMD d'un calcul matriciel : comparaison de trois plateformes

Matthijs Douze
IRIT/ENSEEIHT — douze@enseeiht.fr

Résumé

L'intérêt d'effectuer le même calcul en parallèle sur des données différentes (technique SIMD) est reconnu depuis longtemps. Les constructeurs de processeurs « de bureau » intègrent depuis quelques années des instructions appropriées dans leurs puces. Cependant, leur utilisation n'est pas transparente pour le programmeur. Nous examinons quelles sont les librairies et les interfaces de programmation qu'il peut utiliser, et quels sont les bénéfices qu'il peut en attendre. Les trois plateformes auxquelles nous nous intéressons sont : PowerPC Altivec sous Mac OS X, Pentium 4 SSE sous Linux et UltraSparc VIS sous Solaris.

Mots Clef

Calcul vectoriel, SIMD, développement logiciel.

Introduction

Suivant la loi de Moore, la vitesse des processeurs croît de manière exponentielle. On a amélioré leurs possibilités de différentes manières, plus ou moins contraignantes pour le programmeur et le compilateur :

– en augmentant la fréquence, une accélération sans effort du compilateur et du programmeur ;
– en augmentant le nombre de processeurs, ce qui nécessite d'adapter les algorithmes ;
– en décomposant les instructions pour que les unités de traitement soient toutes occupées en même temps (exécution superscalaire), le compilateur doit ordonancer efficacement les instructions ;
– en dupliquant certaines unités de calcul dans le processeur, là aussi, c'est le compilateur qui travaille ;
– en ajoutant un *pipeline* d'exécution auxiliaire (*hyperthreading*) ;
– en ajoutant une unité SIMD (*Single Instruction, Multiple Data*).

Dans ce dernier cas, c'est le plus souvent le programmeur qui doit faire l'effort de vectoriser son code.

Dans notre contexte de vision par ordinateur, nous utilisons un algorithme de suivi très rapide, mais qui nécessite une initialisation coûteuse. Pendant les longues périodes durant lesquelles l'ordinateur calculait, nous nous sommes dit qu'il serait intéressant d'exploiter pleinement ses possibilités, parmi lesquelles le SIMD.

Comme tout le code est en C (sauf les librairies en Fortran), nous avons cherché à rester le plus possible dans ce langage.

1 Les opérations SIMD

1.1 Principe

Un processeur SIMD contient des registres larges (64 ou 128 bits). Les données contenues dans ces registres sont considérées comme des vecteurs de nombres entiers ou flottants. On peut les combiner membre à membre par des opérations arithmétiques ou logiques. On peut réordonner les éléments, faire des tests dessus (il y a un registre de contrôle dédié), trouver leur maximum ou leur minimum.

Ainsi, un registre de 128 bits peut contenir 2 entiers de 64 bits (signés ou pas), 4×32 bits, 8×16 bits, ou 16×8 bits. Il peut aussi contenir 4 flottants IEEE de 32 bits (`float`) ou 2 flottants IEEE de 64 bits (`double`).

Toutes les opérations ne sont pas possibles sur tous les types de données. On ne peut pas faire d'arithmétique sur les entiers de 128 bits, seulement des opérations booléennes ou de réordonnancement.

Il y a deux types de comportement pour les dépassements de capacité sur les opérations arithmétiques entières :

– modulo : le fonctionnement classique, on calcule modulo une puissance de 2 ;
– saturation : lorsque le résultat d'une opération est plus grand (respectivement petit) que le maximum (minimum) représentable, on le remplace par cet extrémum.

Le mode saturation perturbe moins la forme d'un signal échantillonné quand on effectue des opérations arithmétiques.

L'unité vectorielle a des instructions dédiées pour lire et écrire des données en mémoire. Il faut alors faire attention à l'alignement (sur 16 octets pour un registre 128 bits). En particulier, les espaces alloués par `malloc` ne sont pas nécessairement suffisament alignés.

A priori on n'a pas besoin de support de la part du système d'exploitation pour utiliser les instructions vectorielles : toute la gestion peut être faite en mode utilisateur. Toutefois, il est indispensable que le système sauvegarde les registres vectoriels lors des commutations de tâche.

1.2 Utilisation

Initialement, les opérateurs SIMD ont été utilisés en traitement du signal. Un registre de 128 bits peut contenir 128 pixels d'une image en noir et blanc, 4 pixels d'une image en RGBA, 8 échantillons sonores, etc.

Les types flottants 32 bits ont été introduits pour manipuler des coordonnées en visualisation 3D. Les flottants de 64 bits peuvent être utilisés pour le calcul scientifique.

plateforme	mac	pc	sun
registres entiers	32×32 bits	6×32 bits(1)	32×64 bits(2)
registres flottants	32×64 bits	8×80 bits	32×64 bits
registres vectoriels	32×128 bits	8×128 bits	32×64 bits(3)
caches (L1/L2/L3)	32+32 kB/256 kB/2 MB	48+8 kB/256 kB	32 kB+64 kB/8 MB
profondeur du *pipeline*	7	20	14
accès non-alignés	erreur silencieuse	SIGSEGV	SIGSEGV
modèle testé	G4 866 MHz	Pentium 4, 1.8 GHz	UltraSparc III Cu, 900 MHz
date de livraison	octobre 2001	décembre 2001	octobre 2002
système d'exploitation	Mac OS X 10.2.4	Linux 2.4.18	Solaris 9.0
compilateur(s)	GCC 3.1	GCC 3.2, Intel CC 7.0	Sun Workshop Pro 6.2
licence du compilateur	GPL	GPL, licence personelle	gratuit pour l'enseignement

(1) Les registres ne sont pas banalisés. Par exemple, une multiplication ne peut donner son résultat que dans `EDX.EAX`. (2) Dont 24 sont organisés par paquets (« fenêtres ») sélectionnables, et utilisés pour contenir les paramètres et résultats de fonctions. (3) Partagés avec les registres flottants.

TAB. 1: Détails sur les trois plateformes.

D'une manière générale, le SIMD est utile quand les données sur lesquelles on travaille sont organisées sous forme de *tableau de scalaires*, et qu'on fait des traitements *indépendants* sur ces éléments.

1.3 Les trois processeurs étudiés

Il n'y a pas de consensus entre les constructeurs de processeurs sur ce que doit pouvoir faire une unité vectorielle. Les trois processeurs sont donc assez différents entre eux. Voici un résumé de l'histoire des trois machines. Le tableau 1 récapitule leurs principales caractéristiques. Les trois machines sont comparables parce qu'elles ont été achetées à peu près au même moment (sauf la Sun, qui est nettement plus jeune) dans les mêmes classes de prix.

Sun Sur le processeur 64 bits SPARC, la solution choisie est simple : les registres flottants en double précision servent de registres vectoriels 64 bits, et les opérateurs vectoriels sont implantés dans l'unité de calcul flottant.

Les opérateurs proposés sont sourtout axés sur le traitement d'image. Il y a par exemple un opérateur qui calcule la distance en norme ∞ de deux triplets RGB, et des opérateurs pour faire efficacement des calculs d'adresses dans des tableaux en 3D. Il n'y a pas de vecteurs de flottants, mais il y a un type en virgule fixe.

PC La première extension vectorielle de Intel a été introduite en avec le Pentium MMX (MultiMedia eXtentions). Elle ressemble beaucoup à celle du Sparc. Avec le Pentium III, Intel a fait une vraie unité vectorielle (nommée SSE, *Streaming SIMD Extensions*) avec ses propres registres de 128 bits [5].

Il y a un grand nombre d'opérations de conversion entre types scalaires ou vectoriels, et entre flottants et entiers. Ceci offre une telle souplesse que les registres vectoriels sont parfois utilisés par le compilateur pour compenser la pauvreté de l'architecture en registres entiers.

Mac Motorola a introduit l'extension vectorielle AltiVec avec le G4 [4].

Les unités flottantes et vectorielles sont indépendantes de l'unité de calcul entier : pour copier entre les différents types de registres il faut passer par la mémoire. Comme sur les deux autres unités, l'unité vectorielle offre une instruction de multiplication-addition. Il n'y a pas de types vectoriels (entiers ou flottants) de 64 bits.

2 Les interfaces de programmation

Il s'agit maintenant d'exploiter aussi simplement que possible les opérateurs vectoriels. Dans le processus de compilation, ils peuvent être introduites à différents stades (figure 1).

2.1 Bibliothèques

Les constructeurs de processeurs fournissent aux programmeurs des bibliothèques optimisées qui tirent parti des instructions vectorielles. Ces paquetages sont constitués de fichiers `.h` avec les prototypes des fonctions, des fichiers archives ou librairies dynamiques correspondantes (point (1) de la figure 1), et d'une documentation de référence (en PDF).

Sun Sun fournit la librairie MediaLib, qui comprend des fonctions d'algèbre linéaire (sur des types de données entiers), des opérateurs de traitement d'image (conversion, opérations pixel à pixel, opérations morphologiques, convolution, Fourier), de traitement de son (rééchantillonnage, analyse statistique, convolution, diverses filtres), et de traitement vidéo (ondelettes, compensation de mouvement sur des blocs, compression JPEG).

PC Intel propose IPP (Intel Performance Primitives) qui permet de faire du traitement de signal 1D (réchantillonnage, Fourier et ondelettes, convolution, fonctions de reconaissance de parole, (dé)codage MP3), du traitement d'image (Fourier, ondelettes, convolution, opérations morphologiques, (dé)codage MPEG-1/2, décodage MPEG-4 et H.263). La bibliothèque est payante, mais il y a une version d'évaluation valable 30 jours.

Mac Apple fournit `vecLib.framework` avec ses outils de développement. Il contient une version de BLAS, des opérateurs de transformation de Fourier en 1D, 2D et nD, une bibliothèque de manipulation de nombres de grande taille.

Ces bibliothèques sont optimales dans leur exploitation

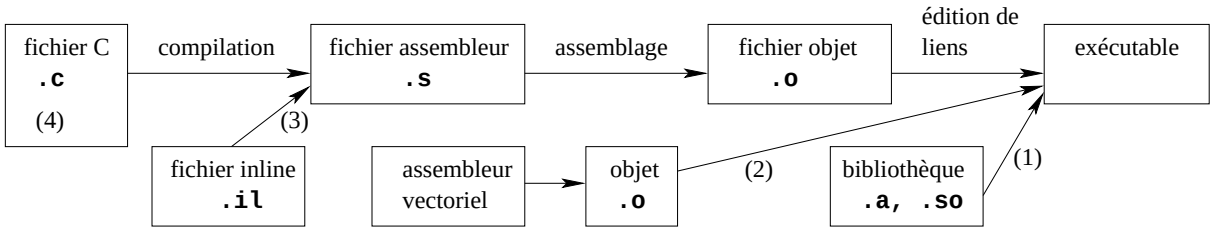

FIG. 1: Étapes de la compilation (chaîne en haut) et points auxquels on peut introduire du code vectoriel (numéros entre parenthèses).

du processeur, donc très utiles quand on a besoin d'un opérateur qu'elles offrent. Cependant, quand ce n'est pas le cas, comme ici (voir section 3), elles sont inexploitables. D'autant plus que leur code source n'est pas disponible, donc on ne peut pas l'adapter.

ATLAS [2] est une bibliothèque intéressante optimisée pour les processeurs vectoriels (AltiVec et SSE). Elle fournit des routines optimisées pour BLAS et LAPACK.

Sa particularité est qu'elle utilise une technique de AEOS (*Automated Empirical Optimization of Software*). Lors de la compilation, la librairie effectue des mesures de performances et en déduit automatiquement quels algorithmes et quels paramètres sont les plus rapides sur la plateforme.

2.2 Assembleur

L'approche au plus bas niveau de la programmation est le langage d'assemblage. On est obligé d'y revenir si le compilateur ignore les registres ou les opérations vectorielles.

Fonctions en assembleur On peut isoler le code vectoriel dans des fonctions écrites entièrement en assembleur. On les met dans un fichier indépendant qui fournit un fichier objet (point (2) de la figure 1).

Pour l'interfacer correctement, il faut savoir quel est le protocole de passage des arguments de fonction en C pour la plateforme donnée (sont-ils sur la pile ? dans des registres ?). En pratique, on peut compiler une fonction de même prototype mais dont le corps est vide, et adapter l'assembleur généré.

Cette méthode oblige à gérer les paramètres, les variables locales, etc. soi-même, ce qui est fastidieux et, à moins de connaître intimement le processeur, sous-optimal.

Assembleur *inline* La plupart des compilateurs de C proposent un moyen d'intégrer de l'assembleur dans le code, sans faire d'appel de fonction. Ceci permet de réduire la quantité de langage machine, sans la pénalité due à un appel de fonction. Le point délicat est d'établir la correspondance entre les variables.

Avec le compilateur Sun, on écrit le code assembleur dans un fichier *inline* (point (3) de la figure 1). Les paramètres sont passés de la même manière que pour les fonctions, c'est-à-dire dans des registres fixes.

Inclusion de code assembleur GCC (et ICC) proposent une technique plus souple pour faire cette mise en correspondance. On introduit le code assembleur dans le code C, dans un bloc `asm` (point (4) de la figure 1). Par exemple (assembleur PowerPC non-vectoriel) :

```
int plus35(int a) {
  int b;
  asm volatile (
    "  addi %1,%0,35\n" /* add immediate */
    : "=r" (b)          /* sortie */
    : "r" (a)           /* entrée */
  );
  return b;
}
```

Le compilateur traite le code assembleur comme une chaîne de caractères. Il remplace les $\%n$ par les registres correspondant aux variables entre parenthèses. Chaque variable est accompagnée d'une *contrainte* (comme `"=r"`) qui indique si elle doit être fournie dans un registre entier, flottant ou en mémoire, si elle est lue et/ou écrite par le code assembleur.

Cette approche a l'avantage de pouvoir être utilisée même si le compilateur ne connaît pas du tout les registres vectoriels. Cependant, il est aussi incapable d'ordonnancer efficacement les instructions, par exemple en entrelaçant les opérations vectorielles et de lecture/écriture.

2.3 *Intrinsics*

Ici, le compilateur connaît les types de données vectoriels. On peut donc déclarer des variables de ce type, le compilateur les range soit en mémoire, soit dans les registres vectoriels. Chaque instruction vectorielle a une fonction ou une macro correspondante en C.

Les quatre compilateurs que nous avons examiné offrent des intrinsics (voir les exemples, figure 2). Le compilateur Sun et ICC sont *faiblement typés* : ils ne distinguent pas les types selon le contenu des registres. Par exemple, un `vis_d64` peut contenir aussi bien 2 entiers de 32 bits que 4 entiers de 16 bits. À l'inverse, l'interface C pour Altivec (définie par Motorola), et le modèle à base de `__attribute__` de GCC sont *fortement typés* : il faut faire un *cast* pour passer de `vector float` à `vector short`.

2.4 Auto-vectorisation

Il est souhaitable pour le confort du programmeur que le compilateur prenne son code scalaire et le tranforme automatiquement en assembleur vectoriel. Cette tâche nécessite une analyse fine des dépendances de données (figure 3). D'une manière générale, une boucle est vectorisable si le résultat est indépendant de l'ordre des exécutions du corps de la boucle (condition suffisante).

Les compilateurs ICC et Sun peuvent générer automatiquement du code vectoriel à base de *thread*s de calcul pour des multiprocesseurs à mémoire partagée (ils supportent

Code scalaire :

```
void vsum(int n,short *a,
          short *b,short *c) {
  int i;
  for(i=0;i<n;i++)
    c[i]=a[i]+b[i];
}
```

Code Mac (à compiler avec `cc -faltivec`). Il suffit de convertir les types en vectoriel :

```
void vsum(int n,short *a,
          short *b,short *c) {
  int i;
  vector short *va=(void*)a,*vb=(void*)b;
  vector short *vc=(void*)c;
  for(i=0;i<n/8;i++)
    vc[i]=vec_add(va[i],vb[i]);
}
```

Code Sun, à compiler avec `cc -DVIS=0x200 vis_64.il -xarch=v9`. Ici, il faut traiter les données par paquets de 4 :

```
#include <vis_types.h>
#include <vis_proto.h>

void vsum(int n,short *a,
          short *b,short *c) {
  int i;
  vis_d64 *va=(void*)a,*vb=(void*)b;
  vis_d64 *vc=(void*)c;
  for(i=0;i<n/4;i++)
    vc[i]=vis_fpadd16(va[i],vb[i]);
}
```

Code PC à compiler avec ICC.

```
#include <emmintrin.h>

void vsum(int n,short *a,
          short *b,short *c) {
  int i;
  __m128i *va=(void*)a,*vb=(void*)b;
  __m128i *vc=(void*)c;
  for(i=0;i<n/8;i++)
    vc[i]=_mm_add_epi16(va[i],vb[i]);
}
```

Code PC à compiler avec `gcc -msse2`. Les intrinsics de GCC 3.2 ne supportent pas encore bien SSE, donc il faut définir le type vectoriel (`m128`) et une fonction `add` correspondante. La contrainte `xm` correspond à un registre SSE *ou* un emplacement mémoire.

```
/* V4SI = vecteur de 4 int */
typedef int m128
    __attribute__ ((__mode__(__V4SI__)));

inline m128 add(m128 a,m128 b) {
  asm volatile (
    "  paddw %1,%0\n"
    : "+x" (a)
    : "xm" (b));
  return a;
}

void vsum(int n,short *a,
          short *b,short *c) {
  int i;
  m128 *va=(void*)a,*vb=(void*)b;
  m128 *vc=(void*)c;
  for(i=0;i<n/8;i++)
    vc[i]=add(va[i],vb[i]);
}
```

FIG. 2: Vectorisation d'une fonction qui ajoute deux vecteurs d'entiers 16 bits. Pour simplifier, on suppose que les tableaux sont alignés, et que leur taille est un multiple de 8.

OpenMP[8]), mais seul ICC est capable d'exploiter automatiquement les opérateurs vectoriels.

On peut aider la génération de code vectoriel avec ICC de deux manières :

- le code `__assume_aligned(a,n);` permet de préciser que le pointeur `a` est aligné sur `n` octets ;
- le qualificateur `restrict` (intoduit dans C99) permet de préciser qu'un tableau n'en recouvre pas un autre (exemple A de la figure 3).

Il faut compiler le code avec les options `-axW -vec_report3 -restrict`. L'exemple A est vectorisée correctement. L'exemple C est vectorisé, mais le code généré n'est pas optimal (il n'utilise pas de vecteur d'accumulateurs).

3 Le problème

Nous allons chercher à vectoriser un algorithme que nous utilisons intensément dans notre recherche [9].

3.1 Un problème d'optimisation en traitement du signal

À un certain moment, on se retrouve avec le problème aux moindres carrés :

$$\widehat{A} = \underset{A \in \mathbb{R}^{n \times p}}{\operatorname{argmin}} \|GA - M\|^2$$

où :

- $M \in \mathbb{R}^{N \times p}$;
- $G \in \mathbb{R}^{N \times n}$ est une matrice de différences d'échantillons de niveaux de gris ;
- $p = 8$, n de l'ordre de 400, N de l'ordre de 5000.

A : boucle vectorisable à condition que les tableaux ne se recouvrent pas (c'est l'exemple de la figure 2) :

```
for(i=0;i<n;i++)
  c[i]=a[i]+b[i];
```

B : boucle non vectorisable :

```
a[0]=0;
for(i=1;i<n;i++)
  a[i]=a[i-1]+1;
```

C : boucle vectorisable si on se rend compte que l'addition est associative (on remplace `s` par un vecteur d'accumulateurs) :

```
s=0;
for(i=0;i<n;i++)
  s+=a[i];
```

FIG. 3: Exemples de boucles plus ou moins vectorisables automatiquement (en négligeant les problèmes d'alignement).

À condition que G soit de rang plein n, la solution de ce problème est $\widehat{A} = G^{+}M$, en notant $G^{+} = (G^{\top}G)^{-1}G^{\top}$ la pseudo-inverse.
En pratique, on préfère utiliser la méthode des équations normales [1]. Elle se déroule en plusieurs étapes, qui consomment chacune un certain nombre d'opérations flottantes (flop) :

1. calculer $M' = G^{\top}M$ ($2pnN$ flops) ;
2. calculer $G' = G^{\top}G$, matrice symétrique (n^2N flops) ;
3. calculer la décomposition de Cholesky ($n^3/3$ flops) de G', ce qui fournit C, triangulaire supérieure, telle que $G' = CC^{\top}$;
4. résoudre $CY = M'$ en Y (pn^2 flops) ;
5. résoudre $C^{\top}A = Y$ en A (pn^2 flops).

Puisque $N > n > p$, l'opération la plus coûteuse est le point 2. Nous allons donc chercher à accélérer vectoriellement la multiplication $G^{\top}G$.
La matrice G est remplie de différences de niveaux de gris, donc de valeurs entières comprises entre -255 et 255. Nous manipulons donc cette matrice comme un tableau de `short`, entiers signés 16 bits.
Ce problème est à mi-chemin entre un problème de traitement du signal (données entières) et de calcul scientifique (multiplication de matrice). Il n'y a donc pas de fonction exactement appropriée dans les librairies optimisées proposées par les fabriquants de processeurs.

3.2 Implantation scalaire

Pour que le tableau G soit accédé autant que possible séquentiellement pendant la multiplication, il faut qu'il soit rangé par colonnes (à la Fortran). Dans ce cas, le calcul se réduit à une série de produits scalaires sur des vecteurs de taille N.

plateforme	mac	pc gcc	pc icc	sun
scalaire entier	3838	1647	1360	4082
scalaire flottant	7280	1449	1459	2491
bibliothèque		518	518	1401
auto-vectorisé			445	
vectoriel	2078	471	445	2783
bloqué	1666	1443	1083	3349
(b_s, b_t)	(16,256)	(2,32)	(4,32)	(16,256)
vectoriel bloqué	542	385	172	1550
(b_s, b_t)	(16,64)	(8,32)	(4,128)	(8,512)
vectoriel déroulé	365	170		
(b_s, b_t)	(4,16)	(4,8)		

TAB. 2: Performances du calcul sur les trois plateformes. Les temps sont en millisecondes (écart-type de l'ordre de 5 ms).

La suite du code (décomposition de Cholesky de Linpack ou Lapack) n'utilise que la partie triangulaire supérieure de G', sous forme d'un tableau de `double`. Ceci nous impose le format de sortie de la fonction.

3.3 Implantation vectorielle

La multiplication de deux entiers sur 16 bits donne un résultat sur 32 bits. Pour un vecteur de taille 8×16 bits (si le registre fait 128 bits) le produit membre à membre est donc un vecteur de taille 8×32 bits. L'opérateur de multiplication doit donc réduire de moitié la taille du résultat pour le ranger dans un registre. Les solutions mises en œuvre consistent à :
– garder les 16 bits de poids fort ;
– garder les 16 bits de poids faible ;
– faire le produit sur la moitié des éléments (que les pairs, ou que les impairs, ou les 4 premiers), et écrire le produit sur 32 bits ;
– faire le produit et aditionner deux valeurs successives sur 32 bits.

La dernière opération est la plus adaptée, puisqu'elle fait déjà une partie de l'addition du produit scalaire stocké dans un accumulateur de 4×32 bits. Elle n'existe pas sur Sun, donc il faut faire le calcul en deux temps : poids faible puis poids fort.

3.4 Implantation vectorielle bloquée

Pour atteindre une vitesse optimale, il ne faut pas négliger les méthodes classiques de calcul matriciel. Il faut en particulier travailler par blocs pour exploiter la hiérarchie de la mémoire [10],

4 Quelques résultats

4.1 Expériences

Nous avons implanté de plusieurs manières la multiplication $G^{\top}G$. Nous avons généré une matrice G de taille 5000×400, et nous avons mesuré les temps de calcul avec `gettimeofday`. La matrice est rangée à la Fortran, avec un paramètre `lda` qui permet de s'assurer que toutes les colonnes sont suffisament alignées. Nous avons fait des scripts /BASH et AWK pour automatiser le processus. Sur Sun, nous avons généré des exécutables en mode 32 bits,

parce qu'ils sont plus rapides.

Version scalaire Le temps de référence est celui de l'algorithme scalaire entier. Nous avons aussi fait l'implantation en mettant le tableau G en `float`. Si l'accumulateur est aussi en `float`, le résultat n'est pas exact car la mantisse de 24 bits est trop petite.

Bibliothèques Les bibliothèques de calcul matriciel (cf. 2.1) existent en `float` (les résultats sont imprécis) ou en `double` (le tableau devient trop gros) ; elles ne sont donc pas adaptées.

Par contre, nous avons utilisé l'opération de produit scalaire 16 bits offerte par les bibliothèques de traitement du signal. Elle remplace la boucle intérieure du produit matriciel.

Version vectorielle Ici aussi, on implante un produit scalaire (comme décrit au 3.3).

Version bloquée On calcule la matrice G' par blocs de taille $b_\mathrm{s} \times b_\mathrm{s}$. Pour chaque bloc, on n'accède qu'à deux séries de b_s colonnes de G. On fait les produits scalaires sur des segments de b_t éléments à la fois.

Nous avons implanté cette stratégie en scalaire et en vectoriel, et nous avons fait varier $(b_\mathrm{s}, b_\mathrm{t})$ par puissances de 2. Nous avons retenu les paramètres les plus rapides.

En guise de test, nous avons aussi déroulé la boucle de calcul du bloc, en générant le code C avec un script Python. Le code devient cependant assez long.

4.2 Commentaires

Les résultats sont présentés dans le tableau 2. En premier lieu, on observe que le PC est largement plus rapide que ses deux concurrents. Sa fréquence d'horloge double est un avantage déterminant en calcul entier. GCC est souvent plus lent que ICC, qui a des algorithmes d'ordonnancement des instructions très fins et personalisés pour les machines Intel.

Les versions vectorielles sont nettement plus rapides, sauf sur Sun, sans doute parce que l'interfaçage à base de `.il` n'est pas efficace. La version auto-vectorisée de ICC est aussi rapide que la version vectorisée à la main (l'assembleur généré pour la boucle interne est le même dans les deux cas).

Les bibliothèques (IPP sur PC, Medialib sur Sun) sont certainement accélérées vectoriellement. IPP n'est pas très rapide par rapport à la version vectorielle, sans doute à cause d'un problème d'interfaçage (elle choisit lors de l'exécution selon le processeur quelle implantation elle va utiliser).

L'accès aux matrices par blocs permet encore de gagner en vitesse. Cet effet est encore plus appréciable en combinaison avec le calcul vectoriel, sauf sur Sun, où on est obligé d'utiliser la bibliothèque pour que les performances ne s'effondrent pas. Nous n'avons pas eu le temps de tester la version vectorielle déroulée pour toutes les plateformes.

Nous avons rencontré un certain nombre de bugs avec GCC 3.2 et 3.3 sur PC. Le support des débuggeurs pour les types SIMD est balbutiant.

Conclusion

La meilleure plateforme pour faire du SIMD est globalement le PC, parce que le processeur est le moins cher et le plus rapide. En dehors des considérations de licences, ICC est le meilleur compilateur. L'extension du C de Motorola qui permet de faire du SIMD est la plus élégante, celle de GCC est la plus souple. Sun est largement distancée parce que son unité SIMD n'a pas évolué depuis 1995.
Nous avons cherché à accélérer notre code en faisant le minimum d'efforts. Dans ce contexte, la stratégie la plus efficace est d'utiliser l'auto-vectoriseur de ICC, qui permet de gagner un facteur 3 en ajoutant 4 mot-clés dans le code et 2 options de compilation.
Si on ne dispose pas d'un compilateur sophistiqué, l'utilisation d'*intrinsic*s permet d'obtenir une accélération appréciable sans effort excessif.
La suite de ce travail pourrait être d'utiliser une multiplication de Strassen [11] qui diminue la quantité de calculs à faire dès que les matrices sont suffisament grosses.

Références

[1] Gene H. Golub, Charles F. Van Loan, *Matrix Computations*, John Hopkins University Press, Baltimore, 1989.

[2] R. Clint Whaley, Antoine Petitet, Jack J. Dongarra, *Automated Empirical Optimization Software and the ATLAS Project*, `http://math-atlas.sourceforge.net/`.

[3] IBM and PowerPC, *PowerPC Microprocessor Family : The Programming Environments for 32-bit Microprocessors*, `http://www.ibm.com/chips/products/powerpc`.

[4] Motorola, *AltiVec Technology Programming Environments Manual*, `http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPEM.pdf`

[5] Intel, *IA-32 Intel Architecture Software Developer's Manual, volumes 1,2,3*, `http://www.intel.com/design/pentium4/manuals/245471.htm`.

[6] Davis L. Weaver, Tom Germon et al. *The SPARC Architecture Manual, version 9*, `http://www.sparc.org/standards.html`.

[7] Sun microsystems, *VIS Instruction Set User's Manual*, `http://www.sun.com/processors/vis/`.

[8] OpenMP Architecture Review Board, *OpenMP C and C++ Application Program Interface* `http://www.openmp.org/specs/mp-documents/cspec20.pdf`

[9] Matthijs Douze, Vincent Charvillat, Bernard Thiesse, *Comparaison et intégration de trois algorithmes de suivi de motifs plans*, conférence ORASIS, Géraldmer, mai 2003.

[10] Michel Daydé, Partick Amestoy, Philippe Berger, *Conception des logiciels numériques pour calculateurs haute-performance*, notes de cours ENSEEIHT, 2000.

[11] Richard Crandall, Jason Klivington, *Fast matrix algebra on Apple G4*, `http://developer.apple.com/hardware/ve/pdf/g4matrix.pdf`.