École Nationale Supérieure
d'Ingénieurs de Caen

6, Boulevard Maréchal Juin
F-14050 Caen Cedex, France

TP $n^o 4$

Niveau	**2ème année**
Parcours	**Informatique**
Unité d'enseignement	**2I2AC3 - Architectures parallèles**
Responsables	**Emmanuel Cagniot** **Emmanuel.Cagniot@ensicaen.fr** **Hugo Descoubes** **Hugo.Descoubes@ensicaen.fr**

Problème

Nous nous intéressons à l'optimisation d'un petit noyau de code numérique (ici un produit matrice-vecteur) sur un processeur mono-cœur au travers de deux techniques :

- le déroulage de boucle qui consiste à exploiter simultanément plusieurs opérateurs pipelinés du processeur ; cette technique ne peut être mise en œuvre que sur des processeurs super-scalaires c'est à dire doté de plusieurs unités d'exécution ;
- les jeux d'instructions vectoriels. Cette technique ne peut être mise en œuvre que sur un processeur super-scalaire doté d'au moins une unité d'exécution SIMD.

Les mesures de performances ainsi que le benchmarking sont assurés par un outil standard du noyau Linux : `perf`. Ce dernier propose différents services permettant de tracer l'exécution d'un code. La figure 1 présente l'utilisation du service `stat` (le seul service que nous utiliserons dans ce TP) permettant d'obtenir des informations relatives à l'exploitation du hardware. Pour plus d'information sur cet outil vous pouvez consulter le lien :

`https://perf.wiki.kernel.org/index.php/Main_Page`.

Dans cet exemple, nous profilons la commande Unix `whoami` permettant d'afficher le login de l'utilisateur courant (les affichages produits par cette commandes sont redirigés vers `/dev/null` pour ne pas apparaîtrent sur la sortie standard). Afin d'obtenir des statistiques pertinentes (valeurs moyennes) nous demandons à ce que la commande soit exécutée dix fois de suite grâce à l'option `-r`. L'option `-B` permet quant à elle d'afficher les valeurs numériques avec une virgule ou un point selon la langue utilisée par le système d'exploitation.

Les statistiques collectées montrent que `whoami` a nécessité $1,009420$ ms de temps CPU (`task-clock:u`) et $29,338667$ ms de temps utilisateur (`time elapsed`). On rappelle que :

```
perf stat -B -r 10 whoami > /dev/null

 Performance counter stats for 'whoami' (10 runs):

          1,009420      task-clock:u (msec)       #    0,034 CPUs utilized            ( +-  1,06% )
                 0      context-switches:u        #    0,000 K/sec
                 0      cpu-migrations:u          #    0,000 K/sec
                68      page-faults:u             #    0,068 M/sec                    ( +-  0,70% )
           284 578      cycles:u                  #    0,282 GHz                      ( +- 18,83% )  (83,88%)
           313 131      instructions:u            #    1,10  insn per cycle           ( +-  0,00% )
            65 680      branches:u                #   65,067 M/sec                    ( +-  0,00% )
     <not counted>      branch-misses:u                                               ( +- 13,21% )  (53,55%)

       0,029338667 seconds time elapsed
```

FIGURE 1 – Exploitation du service `stat`.

— le temps CPU correspond au temps effectivement passé dans le CPU ;
— le temps utilisateur correspond au temps CPU augmenté de celui lié à sa charge c'est à dire le partage avec les autres processus, les attentes liées aux entrées-sorties, aux accès réseaux, les migrations entre les différents CPU de la machine s'il s'agit d'un multiprocesseur (ou multicœur), les fluctuations de l'horloge, les cache misses, TLB misses, predication branch misses, etc. Par conséquent, le temps utilisateur est généralement beaucoup plus élevé que le temps CPU lui-même.

Si nous divisons le temps CPU par le temps utilisateur, nous obtenons la valeur $0,034405789$ qui correspond au taux d'utilisation de notre CPU. Dans cet intervalle 284578 cycles (`cycles:u`) ont été consommés. Si nous divisons cette valeur par le temps CPU exprimé en secondes, nous obtenons une fréquence d'environ $0,282$ GHz. En divisant la valeur 313131 représentant le nombre d'instructions exécutées (`instructions:u`) par le nombre de cycles consommés, nous obtenons la valeur 1.10 représentant le nombre moyen d'instructions exécutées au cours d'un cycle CPU. En prenant l'inverse de ce rapport, nous constatons que chaque instruction nécessite approximativement 1 cycle CPU.

L'archive `tp4.tar.gz`, accessible via la page web décrivant l'unité d'enseignement, contient un squelette incomplet de l'application à réaliser. Sa structure est la suivante :

`src/include/ :` contient les déclarations des différentes versions de notre produit matrice-vecteur ;

`src/ :` contient les définitions des différentes versions de notre produit matrice-vecteur ainsi que le programme principal `bench.c` ;

`CMakeLists.txt :` script permettant de générer le makefile de l'application via l'utilitaire `cmake`. Cinq exécutables différents sont générés grâce à l'utilisation conjointe de `bench.c` et de directives préprocesseur :
— `dry_run` qui représente le programme sans les instructions relatives au produit matrice-vecteur ;
— `bench` qui représente le programme exploitant une version canonique du produit matrice-vecteur ;
— `bench_r4` qui représente le programme exploitant une version du produit matrice-vecteur basée sur un déroulage de boucle ;
— `bench_sse_r4` qui représente le programme exploitant une version du produit matrice-vecteur basée sur un jeu d'instructions SIMD.

`Lisezmoi.txt :` fichier texte décrivant la procédure de génération du makefile et celle de la configuration du fichier `CMakeCache.txt` produit par `cmake`. Une fois ce fichier modifié, le compilateur est autorisé à exploiter les caractéristiques de votre processeur (option `-march=native`) et toutes les optimisations possibles (option `-O3`) si vous lui en laissez la possibilité.

La figure 2 présente la forme canonique de notre produit matrice-vecteur $b = A \times x$ (fichiers sources `matvec.h` et `matvec.c`) c'est à dire la forme non optimisée. Les éléments de A, b et x sont de type `float` c'est à dire des nombre flottants simple précision occupant quatre octets (32 bits) en mémoire.

Notre matrice A, de taille `size` $\times$ `size` (nous l'avons choisie carrée mais elle aurait tout aussi bien pu être rectangulaire pourvu que son nombre de colonnes soit compatible avec la longueur des vecteurs b et x), est représentée sous la forme d'un tableau 1D et non 2D : par conséquent, l'élément $A(i, j)$ correspond à l'entrée `A[i` $\times$ `size + j]` de notre tableau. Le mot-clé `const` désigne les paramètres qui ne sont accédés qu'en lecture seule, les autres (ici le tableau `b`) pouvant l'être aussi bien en lecture qu'en écriture.

Le symbole pré-processeur `DRY_RUN` est utilisé pour générer les exécutables `dry_run` et `bench`. S'il n'est pas défini alors la fonction `matvec` contient les instructions de calcul du produit matrice-vecteur. Inversement, s'il ne l'est pas alors le corps de la fonction est vide. Le rôle de l'exécutable `dry_run` est de mesurer le temps d'exécution du programme sans l'algorithme de multiplication matrice-vecteur c'est à dire celui de son enveloppe.

1 Enveloppe et forme canonique

L'enveloppe de l'algorithme de multiplication matrice-vecteur représente l'ensemble des instructions du programme qui ne sont pas liées à l'algorithme lui-même. Séparer l'enveloppe de l'algorithme nous

```
void
matvec(const float A[], const float x[], float b[], const unsigned size) {

#ifndef DRY_RUN

  // Boucle sur les composantes du vecteur cible b et donc les lignes de la
  // matrice A.
  for (unsigned i = 0; i != size; i ++) {

    // La composante b[i] étant utilisée comme accumulateur pour l'addition, il
    // faut commencer par l'initialiser à zéro.
    b[i] = 0.0;

    // Boucle sur les composantes du vecteur source x et donc les colonnes de
    // la matrice A. Comme cette dernière est représentée sous la forme d'un
    // tableau unidimensionnel, il nous faut réaliser la conversion 2D -> 1D.
    // Par conséquent, deux jeux d'indices sont utilisés : j qui parcourt les
    // colonnes de la ligne i dans la matrice A et k qui parcourt les
    // composantes du vecteur source x.
    for (unsigned j = i * size, k = 0; k != size; j ++, k ++) {
      b[i] += A[j] * x[k];
    }

  }

#endif

}
```

FIGURE 2 – Forme canonique (non optimisée) de notre produit matrice-vecteur.

permettra par la suite de mesurer les performances obtenues pour chaque version optimisée de ce dernier.

1.1 Question

Appliquez le service `stat` de l'utilitaire `perf` à l'exécutable `dry_run` puis consignez le temps d'exécution utilisateur $\mathcal{T}_{init}$ en millisecondes ainsi que le nombre de cycles $\mathcal{N}_{init}$ consommés.

Dans la suite de cet énoncé, nous allons calculer les deux quantités suivantes pour chaque version de notre algorithme (vous consignerez le tout dans le tableau de la figure 3) :

$$\mathcal{T}_{algo} = \frac{\mathcal{T}_{total} - \mathcal{T}_{init}}{ITERS}, \tag{1.1}$$

$$\mathcal{N}_{algo} = \frac{\mathcal{N}_{total} - \mathcal{N}_{init}}{ITERS}, \tag{1.2}$$

où $\mathcal{T}_{total}$ et $\mathcal{N}_{total}$ représentent respectivement le temps utilisateur et le nombre de cycles consommés par l'exécutable correspondant à une version particulière de notre algorithme. Ces deux équations représentent respectivement le temps utilisateur et le nombre de cycles consommés par l'algorithme seul. La constante $ITERS$ représente le nombre de répétitions de l'algorithme permettant d'atteindre des durées d'exécutions mesurables. Sa valeur est définie par le biais d'un symbole pré-processeur dans le code source `bench.c`.

	`bench` (canonical form)	`bench_r4` (loop unrolling)	`bench_sse_r4` (vectorization)
$\mathcal{T}_{total}$ (ms)			
$\mathcal{N}_{total}$			
$\mathcal{T}_{algo}$ (ms)			
$\mathcal{N}_{algo}$			
Insn. per cycle			

FIGURE 3 – Caractéristiques des différentes versions de notre algorithme.

L'exécutable `bench` est associé à la forme canonique de notre algorithme c'est à dire la forme non optimisée. Comme le compilateur est autorisé (fichier `CMakeCache.txt`) à mettre en œuvre toutes les optimisations possibles, c'est lui qui déterminera la meilleure façon de procéder. Plus précisément il optera soit pour une technique de déroulage de boucle, soit une vectorisation par le biais d'un jeu d'instructions SIMD, soit une solution mixte si votre processeur présente plusieurs unités d'exécutions SIMD. Par conséquent, la forme canonique d'un algorithme est toujours sa forme la plus portable d'une architecture à une autre mais pas forcément la plus optimisée si le compilateur ignore certaines caractéristiques spécifiques de ces architectures.

1.2 Question

Appliquez le service `stat` de l'utilitaire `perf` à l'exécutable `bench` puis remplissez la colonne correspondante dans le tableau de la figure 3.

2 Déroulage de boucle

La technique du déroulage de boucle consiste à activer plusieurs opérateurs pipelinés simultanément pourvu que le processeur sous-jacent soit superscalaire.

Dans notre cas, nous allons dérouler la boucle de la forme canonique sur une profondeur de 4 c'est à dire que nous allons tenter de calculer simultanément quatre composantes consécutive du vecteur cible b. Cette profondeur n'est pas choisie au hasard car les éléments de nos vecteurs et de nos matrices occupent 32 bits en mémoire. Par conséquent, quatre éléments consécutifs occupent 128 bits en mémoire, ce qui correspond justement à la taille d'un registre `XMM` dans le jeu d'instruction SIMD SSE que nous allons mettre en œuvre dans la troisième partie de ce TP. De fait, puisque le compilateur est autorisé à tout optimiser, il pourra encore choisir entre dérouler la boucle comme nous le demandons ou bien opter pour une vectorisation dans le jeu d'instruction SSE : nous conservons ainsi une certaine portabilité entre architectures. Cependant, contrairement à la forme canonique, nous sommes beaucoup plus intrusifs en imposant une profondeur de déroulage (ici 4) alors que le nombre d'opérateurs pipelinés disponible peut être beaucoup plus important.

La figure 4 présente la définition incomplète de la fonction `matvec_r4` (fichier `matvec_r4.c`) que vous devrez compléter.

```
1  void
2  matvec_r4(const float A[], const float x[], float b[], const unsigned size) {
3
4    // Quatre accumulateurs différents.
5    float acc0, acc1, acc2, acc3;
6
7    // Boucle sur les composantes du vecteur cible b et donc les lignes de la
8    // matrice A.
9    for (unsigned i = 0; i != size; i ++) {
10
11     // Nos accumulateurs sont ré-initialisés pour le calcul de chaque nouvelle
12     // composante du vecteur cible b.
13     acc0 = acc1 = acc2 = acc3 = 0.0;
14
15     // ... à compléter ...
16
17     // La valeur finale de la composante b[i] est la somme de nos quatre
18     // accumulateurs.
19     b[i] = acc0 + acc1 + acc2 + acc3;
20
21   }
22
23 }
```

FIGURE 4 – Fonction `matvec_r4` implémentant un déroulage de boucle sur une profondeur de 4.

Par rapport à la forme canonique, nous introduisons quatre accumulateurs différents afin de rompre la dépendance sur `b[i]` dans le corps de la boucle extérieure (technique mise en œuvre dans le TP-TP n^{o}1). Ce faisant la valeur des quatre accumulateurs peut être calculée simultanément. In fine, la valeur de la composante `b[i]` vaut la somme de ces accumulateurs.

2.1 Question

Complétez la définition de la fonction `matvec_r4`. Une fois l'exécutable `bench_r4` généré, appliquez lui le service `stat` de l'utilitaire `perf` puis remplissez la colonne correspondante dans le tableau de la

figure 3.

3 Vectorisation

Le sigle MMX (MultiMedia eXtended) représente la première tentative d'Intel pour introduire un jeu d'instructions SIMD dans ses processeurs 32 bits. Initié en 1996 sur les Pentium MMX, ce jeu d'instructions prévoit huit nouveaux registres 64 bits baptisés `MM0`, ..., `MM7`, chacun pouvant accueillir au choix :

— huit nombres entiers codés sur 8 bits (type `char`) ;
— quatre nombres entiers codés sur 16 bits (type `short int`) ;
— deux nombres entiers codés sur 32 bits (type `int`) ;
— un nombre entier codé sur 64 bits (type `long int`).

Les instructions MMX peuvent traiter le contenu de ce registre de deux façons :

— **mode decompacté** : le registre ne contient qu'une seule donnée sur laquelle s'applique l'instruction. Il s'agit du modèle d'exécution classique de Von Neuman ;
— **mode compacté** : le registre contient plusieurs données sur lesquelles s'applique simultanément l'instruction. Il s'agit du modèle d'exécution SIMD qui permet d'atteindre un facteur d'accélération proche de n si n représente le nombre de données contenues dans le registre.

Par défaut, les données sont accédées en mémoire par blocs de 64 bits. Les architectures Intel étant little endian, la première donnée du paquet est implantée dans la partie basse du registre. Par conséquent, un seul accès mémoire suffit pour descendre plusieurs éléments consécutifs d'un tableau.

Le Pentium MMX souffre d'un gros handicap dû au choix effectué par Intel pour l'implémentation des registres `MMx` : de simples prête-noms pour la partie basse des registres 80 bits du coprocesseur arithmétique `x87`. Par conséquent, l'exécution d'une instruction MMX est mutuellement exclusive avec l'exécution d'une instruction en virgule flottante.

La seconde tentative, baptisée SSE (Streaming SIMD Extension), est lancée en 1999. Les Pentium III, processeurs 32 bits, se voient dotés de huit nouveaux registres 128 bits nommés `XMM0`, ..., `XMM7` et pouvant accueillir, au choix :

— seize nombres entiers codés sur 8 bits (type `char`) ;
— huit nombres entiers codés sur 16 bits (type `short int`) ;
— quatre nombres entiers codés sur 32 bits (type `int`) ;
— deux nombres entiers codés sur 64 bits (type `long int`) ;
— quatre nombres flottants simple précision codés sur 32 bits (type `float`).

Pour cette nouvelle technologie, Intel corrige l'erreur commise sur les Pentium MMX en implantant physiquement les registres `XMMx` sur le processeur. Malheureusement, les données continuent d'être accédées en mémoire par blocs de 64 bits. Par conséquent, deux accès consécutifs sont nécessaires pour remplir les registres `XMMx`, ce qui impose de prévoir des instructions spécifiques permettant de garnir leur partie haute et d'autres leur partie basse.

La technologie SSE2, qui apparaît en 2000 avec les processeurs 32 bits Pentium IV, marque un tournant. D'une part, le bus de données de 128 bits autorise un seul accès pour remplir un registre `XMMx`. D'autre part, ce même registre peut maintenant recevoir, au choix (voir Figure 5) :

— seize nombres entiers codés sur 8 bits (type `char`) ;
— huit nombres entiers codés sur 16 bits (type `short int`) ;
— quatre nombres entiers codés sur 32 bits (type `int`) ;
— deux nombres entiers codés sur 64 bits (type `long int`) ;
— quatre nombres flottants simple précision codés sur 32 bits (type `float`) ;
— deux nombres flottants double précision codés sur 64 bits (type `double`).

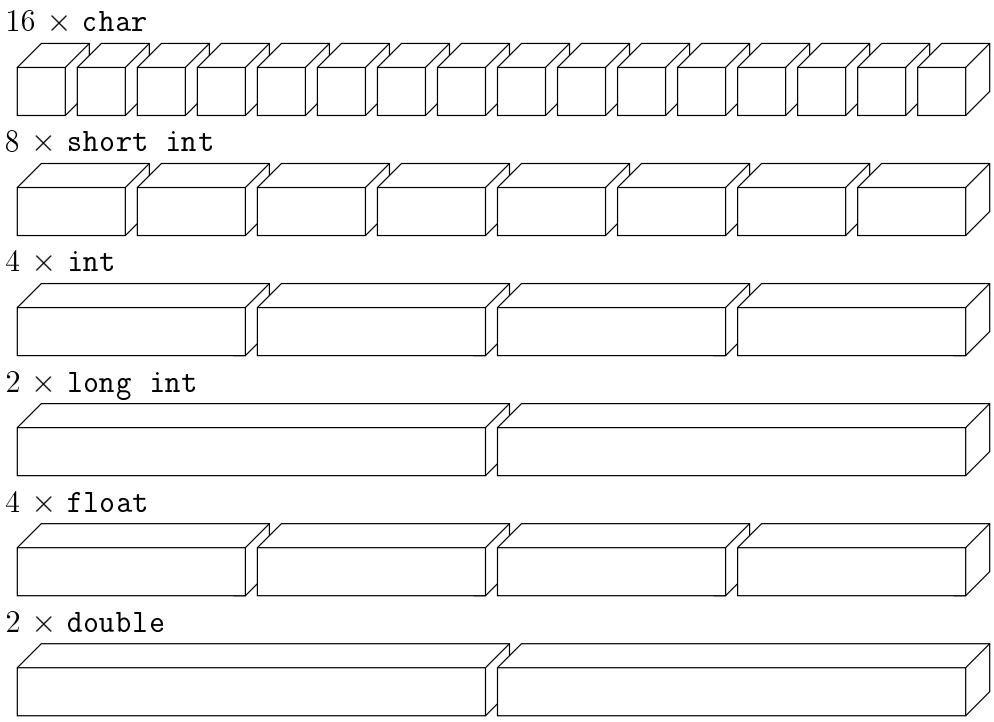

FIGURE 5 – L'un des huit registres 128 bits `XMMx` sur Pentium IV.

La technologie SSE3 apparaît en 2004 avec la seconde génération Prescot des Pentium IV. Elle enrichit les jeux d'instructions SSE et SSE2 par des instructions relatives notamment à la nouvelle technologie INTEL baptisée Hyperthreading. La technologie SSSE3 débarque en 2006 avec les architectures 64 bits qui proposent maintenant seize registres 128 bits nommés `XMM0`, ..., `XMM15`. Elle ne fait qu'enrichir les jeux SSE-SSE2-SSE3 de quelques instructions. Enfin, la technologie SSE4, qui apparaît en 2007, augmente les jeux SSE-SSE2-SSE3-SSSE3 d'une bonne cinquantaine de nouvelles instructions. Courant 2011, INTEL propose, avec les architectures Sandy Bridge, un nouveau jeu d'instructions baptisé AVX (Advanced Vector Extension) et basé sur des registres 256 bits. La longueur de ces derniers est doublée dans le jeu d'instructions AVX-512 qui apparaît en 2013.

Il existe trois manières d'exploiter les jeux d'instructions SIMD :

— en activant les options d'optimisation du compilateur. Cette manière de procéder, généralement satisfaisante, est évidemment la plus simple puisqu'elle ne requiert aucune action particulière de la part du programmeur. Cependant, pour l'écriture de routines très optimisées telles que des BLAS (Basic Linear Algebra Set), ce dernier doit prendre la main ;

— en insérant directement du code assembleur dans les routines critiques. L'efficacité est alors maximale mais la contrepartie à payer est connue : complexité et risques d'erreurs élevé, durée de développement longue, forte difficulté de maintenance, portabilité illusoire, etc. ;

— en exploitant des routines basiques prédéfinies et rédigées en assembleur : les intrinsics. Cette forme de programmation, très efficace également, peut être vue comme une écriture dans un assembleur de haut niveau donc un peu plus portable que de l'assembleur pur et dur. C'est à cette dernière forme d'exploitation que nous nous intéressons, et plus particulièrement au jeu d'instruction SSE permettant de traiter simultanément quatre nombres de type `float`.

Les prototypes des routines intrinsics sont regroupés dans le fichier entête `x86intrin.h`. La cible des instructions SSE étant les tableaux, il est important que ceux-ci soient alignés en mémoire sur des blocs de seize octets. Dans notre cas, cet alignement est imposé lors de l'allocation dynamique dans le fichier

source `bench.c` et vous n'avez donc pas à vous en soucier.

La figure 6 présente une utilisation des instructions SSE pour calculer la somme des éléments d'un tableau de type `float`.

```
/*
 * Union permettant d'accéder aux quatre nombre flottants simple précision
 * compactés dans un registre 128 bits.
 */
typedef union {
  __m128 m128_vec;     // Le registre.
  float  m128_f32[4]; // Ce même registre vu comme un tableau de taille 4.
} xmm_t;

float
sum_sse(const float x[], const unsigned n) {

  // Nos quatre accumulateurs.
  xmm_t acc;

  // Mise à zéro des accumulateurs.
  acc.m128_vec = _mm_setzero_ps();

  // Registre permettant d'accueillir quatre éléments consécutifs de x.
  __m128 pk_x;

  // Boucle permettant de traiter quatre éléments à chaque itération.
  for (unsigned i = 0; i != n; i += 4) {
    pk_x          = _mm_load_ps(x + i);
    acc.m128_vec = _mm_add_ps (acc.m128_vec, pk_x);
  }

  // Le resultat final est la somme des quatre accumulateurs.
  return acc.m128_f32[0]
    + acc.m128_f32[1]
    + acc.m128_f32[2]
    + acc.m128_f32[3];

}
```

FIGURE 6 – Somme des éléments d'un tableau.

Le type `__m128` désigne un registre 128 bits accueillant quatre nombres de type `float` (packed data). Pour pouvoir accéder individuellement à chacun de ces nombres, nous utilisons un type `union` permettant de faire partager un même emplacement mémoire à des données de type différents (dans notre cas ces données ont exactement la même taille). Les fonctions intrinsics mises en œuvre dans ce exemple sont :

- `__m128 _mm_setzero_ps()` qui initialise les quatre emplacements d'un registre 128 bits à zéro.
- `__m128 _mm_load_ps(float *p)` qui recopie les quatres nombres de type `float` implantés à partir de l'adresse `p` en mémoire dans un registre 128 bits.

La figure 6 montre qu'un code optimisé avec des fonctions intrinsics ne laisse aucune liberté au compilateur : il s'agit de la forme d'écriture la plus intrusive et bien évidemment la moins portable d'une architecture à une autre. De fait, recourir immédiatement aux jeux d'instructions SIMD n'est pertinent

que si l'on désire tirer un maximum de performances d'une architecture de processeur spécifique.

La figure 7 présente la définition incomplète de la fonction `matvec_sse_r4` (fichier `matvec_sse_r4.c`) que vous devrez compléter (le mot-clé `restrict` indique au compilateur que les tableaux impliqués occupent des emplacements différents en mémoire et qu'ils ne se recouvrent donc pas).

```
1  void
2  matvec_sse_r4(const float A[restrict],
3                const float x[restrict],
4                      float b[restrict],
5                const unsigned size) {
6
7    // Nos quatre accumulateurs.
8    xmm_t acc;
9
10   // A chaque accès mémoire nous allons ramener quatre nombre flottants en
11   // provenance de la matrice A et quatre autres en provenance du vecteur
12   // source. Il faudra ensuite procéder à la multiplication des quatres valeur
13   // en provenance de A par celles en provenance de B. Par conséquent, nous
14   // utilisons trois registres 128 bits différents pour simplifier l'écriture.
15   __m128 AA, xx, AAxx;
16
17   // Boucle sur les composantes du vecteur cible b et donc les lignes de la
18   // matrice A.
19   for (unsigned i = 0; i != size; i ++) {
20
21     // Mise à zéro de nos quatre accumulateurs.
22     acc.m128_vec = _mm_setzero_ps();
23
24     // ... à compléter ...
25
26   }
27
28   // La composante b[i] vaut la somme des quatre accumulateurs.
29   b[i] = acc.m128_f32[0]
30     + acc.m128_f32[1]
31     + acc.m128_f32[2]
32     + acc.m128_f32[3];
33
34   }
35
36 }
```

FIGURE 7 – Fonction `matvec_sse_r4` implémentant une vectorisation dans le jeu d'instructions SSE.

3.1 Question

Complétez la définition de la fonction `matvec_sse_r4`. Une fois l'exécutable `bench_sse_r4` généré appliquez lui le service `stat` de l'utilitaire `perf` puis remplissez la colonne correspondante dans le tableau de la figure 3.