Sorbonne Université

Polytech – EI-SE5

HPC – TP2 : Programmation avec des instructions SIMD

Dans ce TP nous allons nous focaliser sur l'écriture d'un code mono-cœur efficace. Pour y parvenir, nous allons utiliser les instructions SIMD présentent dans la plupart des CPU modernes. De plus, nous appliquerons aussi certaines transformations de code source comme dans le TP1. L'optimisation sera focalisée sur un algorithme intensif en calcul (*compute intensive* en anglais). Cela nous permet d'adresser l'optimisation des calculs sans avoir de limitations provenant des débits mémoires. Gardez à l'esprit que les instructions SIMD peuvent être utilisées dans beaucoup d'autres domaines applicatifs.

1 Mise en bouche

1.1 Récupérer la version C++ d'EasyPAP

Il est nécessaire de re-télécharger EasyPAP sur votre disque avant de commencer le TP. Des modifications ont été apportées à EasyPAP par rapport au dernier TP. Notamment, dans cette nouvelle version, le code est compilé en C++ au lieu d'un compilateur C. Cela permet l'utilisation de la bibliothèque MIPP pour la vectorisation du code. De votre côté, cela ne change rien, les codes que nous allons regarder sont toujours en C et il vous est demandé d'écrire du code source en C (même si le compilateur compile en C++). Le C++ « englobe » le C : qui peut le plus peut le moins !
Le seul prérequis est d'avoir installé un compilateur C++ sur sa machine. Sous Linux Ubuntu l'installation se fait très simplement :

```
sudo apt install g++
```

Ensuite vous devez cloner le dépôt Git d'EasyPAP. La commande suivante va aussi récupérer MIPP automatiquement pour vous (option `--recursive` de Git) :

```
git clone -b simd --recursive https://gitlab.lip6.fr/cassagnea/easypap-se.git easypap-se-simd
```

Pour compiler, rien n'a changé :

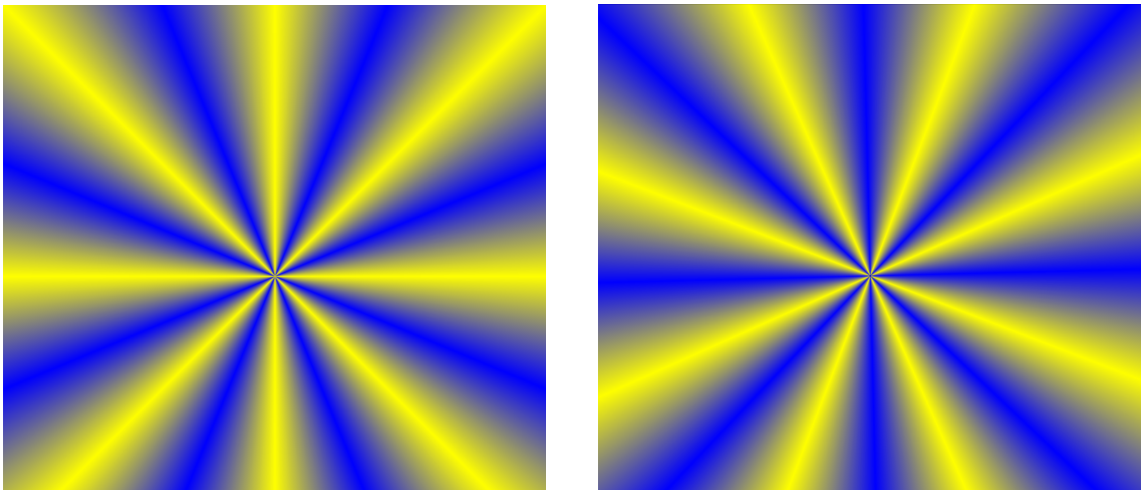
```
cd easypap-se-simd; make -j4
```

1.2 Le noyau de calcul `spin`

Dans ce TP nous allons utiliser EasyPAP et plus particulièrement nous allons travailler sur le noyau de calcul (ou *kernel*) `spin`. Ce noyau de calcul a pour effet de faire tourner l'image au fur et à mesure des itérations. Chaque pixel, en fonction de sa position dans l'image et d'un angle, va se voir attribuer une couleur entre le jaune et le bleu. L'angle est commun à tous les pixels et il est incrémenté d'un degré à chaque itération. La Figure 1 illustre le *kernel* `spin`.
Le Code 1 présente l'implémentation actuelle de l'algorithme `spin`. Comme vous pouvez le voir, la fonction `spin_compute_seq` est appelée en premier (c'est le point d'entrée du *kernel* EasyPAP), puis les fonctions `compute_color` et `rotate` sont appelées ensuite.
Vous pouvez dès à présent lancer le noyau de calcul dans EasyPAP :

```
./bin/easypap -k spin -i 500 -v seq
```

La commande précédente permet de lancer 500 itérations du *kernel* `spin`. De plus, nous avons spécifié la version avec l'option `-v` (ou `--version` en version longue). Cela permet d'appeler automatiquement `spin_compute_seq` ligne 18 dans le Code 1.
Dans ce TP nous allons écrire plusieurs variantes de la fonction `spin_compute_seq`. Pour cela vous serez invités à copier coller cette même fonction et à remplacer la fin du nom (ici `seq`) par le nouveau nom. Vous pourrez ensuite comparer les temps d'exécution des différentes versions en utilisant le paramètre `-v` d'EasyPAP.



(a) Image originale. (b) Rotation de 25 degrés (25 itérations).

FIGURE 1 – Illustration du noyau de calcul `spin`.

```
1  // Computation of one pixel
2  static unsigned compute_color(int i, int j) {
3    float atan2f_in1 = (float)DIM / 2.f - (float)i;
4    float atan2f_in2 = (float)j - (float)DIM / 2.f;
5    float angle = atan2f(atan2f_in1, atan2f_in2) + M_PI + base_angle;
6    float ratio = fabsf((fmodf(angle, M_PI / 4.f) - (M_PI / 8.f)) / (M_PI / 8.f));
7    int r = color_a_r * ratio + color_b_r * (1.f - ratio);
8    int g = color_a_g * ratio + color_b_g * (1.f - ratio);
9    int b = color_a_b * ratio + color_b_b * (1.f - ratio);
10   int a = color_a_a * ratio + color_b_a * (1.f - ratio);
11   return rgba(r, g, b, a);
12 }
13
14 static void rotate(void) {
15   base_angle = fmodf(base_angle + (1.f / 180.f) * M_PI, M_PI);
16 }
17
18 unsigned spin_compute_seq(unsigned nb_iter) {
19   for (unsigned it = 1; it <= nb_iter; it++) {
20     for (unsigned i = 0; i < DIM; i++)
21       for (unsigned j = 0; j < DIM; j++)
22         cur_img(i, j) = compute_color(i, j);
23
24     rotate(); // Increase the base angle (+ 1 degree)
25   }
26
27   return 0;
28 }
```

Code source 1 – Code du noyau de calcul `spin` dans EASYPAP.

Le traitement effectué par le *kernel* `spin` fait intervenir des notions de trigonométrie. Vous pouvez vous en rendre compte car il y a des appels à la fonction `atan2f` (définie dans le fichier d'en-têtes `math.h`). Comme dans le précédent TP, les différentes composantes d'un pixel sont le rouge, le vert, le bleu et le canal alpha (= la transparence). Chacune de ces composantes est stockée sur 8 bits. Toutes ces composantes sont regroupées dans un même entier 32-bit. La fonction `rgba` produit un pixel dans l'image finale à afficher (ligne 11 du Code 1).

Travail à faire Lancer le *kernel* `spin` sur votre machine, comprendre le code.

1.3 Temps d'exécution en fonction des options d'optimisation

La compilation du code EasyPAP est définie dans le ficher `Makefile` à la racine. Les options de compilations sont définies à la ligne 57 :

```
CFLAGS += -O3 -march=native -Wall -Wno-unused-function -std=c++11
```

Comme vous pouvez le voir l'option `-O3` est présente, c'est l'option d'optimisation de niveau 3.

Travail à faire 1 Compiler le code avec les différents niveaux d'optimisation disponibles (`-O0`, `-O1`, `-O2` et `-O3`) puis comparer et noter les différents temps d'exécution pour 500 itérations. Pour voir le temps d'exécution il faut utiliser l'option `--no-display` d'EasyPAP.

Travail à faire 2 Comme le *kernel* fait appel à différentes fonctions mathématiques (`atan2f`, `fmodf` et `fabsf`), il est intéressant de voir l'impact de l'optimisation `-ffast-math`. Pour rappel, cette optimisation fait des sacrifices sur la précision des calculs flottants au profit de la performance. Recompiler votre code en combinant les différents niveaux d'optimisation avec l'option `-ffast-math`. Qu'observez-vous ?

Pour la suite du TP, il faut utiliser le niveau d'optimisation « trois avec l'option *fast math* activée » (`-O3 -ffast-math` = toutes les optimisations).

2 Modifier le code pour diminuer le temps d'exécution

2.1 Approximation de fonctions mathématiques (`approx`)

```
1  // arc tangante, résultat entre -pi/2 et pi/2 radians
2  static float atanf_approx(float x) {
3    return x * M_PI / 4.f + 0.273f * x * (1.f - fabsf(x));
4  }
5  // arc tangante, résultat entre -pi et pi radians
6  static float atan2f_approx(float y, float x) {
7    float ay = fabsf(y);
8    float ax = fabsf(x);
9    int invert = ay > ax;
10   float z = invert ? ax / ay : ay / ax; // [0,1]
11   float th = atanf_approx(z);           // [0,pi/4]
12   if (invert) th = M_PI_2 - th;         // [0,pi/2]
13   if (x < 0) th = M_PI - th;            // [0,pi]
14   if (y < 0) th = -th;
15   return th;
16 }
17 // reste de la division flottante
18 static float fmodf_approx(float x, float y) {
19   return x - trunc(x / y) * y;
20 }
```

Code source 2 – Approximations des fonctions `atanf`, `atan2f` et `fmodf`.

Travail à faire En partant de la version initiale du code (version `seq`), écrire une nouvelle version `approx` dans laquelle vous remplacerez les appels aux fonctions mathématiques par les approximations données dans le Code 2. Testez votre code avec EasyPAP. Que constatez-vous ? Comment l'expliquez-vous ?

2.2 Préparation pour la vectorisation du code (`simd_v0`)

À partir de maintenant vous allez commencer à vectoriser le code. Pour cela vous allez utiliser la bibliothèque SIMD MIPP. Vous n'avez pas besoin de l'installer, cela est déjà fait pour vous. Cependant vous pouvez (voire devez !) vous référer à la documentation qui est disponible dans le *readme* : `https://github.com/aff3ct/MIPP#list-of-mipp-functions`. Cette dernière détaille toutes les opérations SIMD disponibles au travers de la bibliothèque.

```
1  mipp::Reg<int> compute_color_simd_v0(mipp::Reg<int> r_i, mipp::Reg<int> r_j) {
2    // à faire !
3  }
4
5  unsigned spin_compute_simd_v0(unsigned nb_iter) {
6    int tab_j[mipp::N<int>()];
7    for (unsigned it = 1; it <= nb_iter; it++) {
8      for (unsigned i = 0; i < DIM; i++)
9        for (unsigned j = 0; j < DIM; j += mipp::N<float>()) {
10         for (unsigned jj = 0; jj < mipp::N<float>(); jj++) tab_j[jj] = j + jj;
11         int* img_out_ptr = (int*)&cur_img(i, j);
12         mipp::Reg<int> r_result =
13             compute_color_simd_v0(mipp::Reg<int>(i), mipp::Reg<int>(tab_j));
14         r_result.store(img_out_ptr);
15       }
16     rotate();
17   }
18   return 0;
19 }
```

Code source 3 – Implémentation de la fonction `spin_compute_simd_v0`.

Pour vous aider à vous lancer, le code de la fonction `spin_compute_simd_v0` vous est donné. Prenez le temps de le comprendre. Il est **TRÈS IMPORTANT** de voir que la boucle ligne 9 du Code 3 est différente par rapport au code initial. En effet, le pas de la bouche est de `mipp::N<float>()`, le nombre d'éléments dans un registre vectoriel, au lieu de 1 initialement.
Vous remarquerez que le prototype de la fonction `compute_color_simd_v0` (ligne 1 du Code 3) est différent des implémentations scalaires. C'est normal puisqu'à partir de maintenant nous allons travailler sur des registres vectoriels : `mipp::Reg<int>` au lieu de `int` et `mipp::Reg<float>` au lieu de `float`. Pour le moment MIPP ne supporte pas les entiers non-signés alors il faudra **ABSOLUMENT ÉVITER** d'utiliser `mipp::Reg<unsigned>`.

Travail à faire 1 Implémenter la fonction `compute_color_simd_v0`. Pour cela, dans cette première implémentation, écrire une boucle qui itère sur les éléments des registres vectoriels en entrée (`r_i` et `r_j`). Chaque élément d'un registre vectoriel peut être individuellement accédé avec l'opérateur `[]` (comme pour un tableau). Le calcul à l'intérieur de la boucle sera pour le moment séquentiel (le même que dans la fonction `compute_color_approx` implémentée précédemment). Vous sauvegarderez les n pixels dans un tableau d'entiers de taille `mipp::N<int>()` puis vous retournerez un registre que vous initialiserez en chargeant le tableau précédent (voir ligne 13 du Code 3 pour le chargement d'un tableau).

Travail à faire 2 Vérifiez visuellement que le code fonctionne toujours quand vous l'exécutez. Comparez le temps d'exécution par rapport aux versions précédentes. Que constatez-vous ? Comment l'expliquez-vous ?

2.3 « Et zé bartiii » pour la vectorisation (`simd_v1`)

Travail à faire 1 Écrire une version SIMD du reste de la division dans la fonction `fmodf_approx_simd`.

Travail à faire 2 À partir de la version `compute_color_simd_v0` que vous avez écrite précédemment, implémentez une nouvelle version `compute_color_simd_v1` qui fait appel à `fmodf_approx_simd`.

Travail à faire 3 Vérifiez visuellement que le code fonctionne toujours quand vous l'exécutez. Comparez le temps d'exécution par rapport aux versions précédentes. Que constatez-vous ? Comment l'expliquez-vous ?

Tips 1 Il faut scinder la boucle qui itère sur les éléments des registres en deux et il ne doit y avoir qu'un seul appel à la fonction `fmodf_approx_simd`.

Tips 2 Quelques fonctions SIMD MIPP bien utiles :

- `mipp::trunc`,
- `mipp::abs`,
- `mipp::sub` (ou l'opérateur `-`),
- `mipp::mul` (ou l'opérateur `*`),
- `mipp::div` (ou l'opérateur `/`).

2.4 « Un pas de plus » dans la vectorisation (`simd_v2`)

Travail à faire 1 Écrire une version SIMD de la fonction `rgba` dans la fonction `rgba_simd`. Pour rappel, l'implémentation de la fonction scalaire `rgba` se trouve dans le fichier `include/img_data.h`.

Travail à faire 2 À partir de la version `compute_color_simd_v1` que vous avez écrite précédemment, implémentez une nouvelle version `compute_color_simd_v2` qui fait appel à `rgba_simd`.

Travail à faire 3 Vérifiez visuellement que le code fonctionne toujours quand vous l'exécutez. Comparez le temps d'exécution par rapport aux versions précédentes. Que constatez-vous ? Comment l'expliquez-vous ?

Tips 1 La deuxième boucle `for` dans la fonction `compute_color_simd_v1` doit disparaître au profit d'instructions exclusivement SIMD dans la nouvelle version `compute_color_simd_v2`.

Tips 2 Quelques fonctions SIMD MIPP bien utiles :

- `mipp::lshift` (ou l'opérateur `<<`),
- `mipp::orb` (ou l'opérateur `|`),
- `mipp::add` (ou l'opérateur `+`),
- `mipp::cvt<float, int>` (pour convertir un registre vectoriel flottant en registre vectoriel entier).

2.5 « À pieds joints » dans la vectorisation (`simd_v3`)

Travail à faire 1 Écrire des versions SIMD des fonctions `atanf_approx` et `atan2f_approx` dans les fonctions `atanf_approx_simd` et `atan2f_approx_simd`.

Travail à faire 2 À partir de la version `compute_color_simd_v2` que vous avez écrite précédemment, implémentez une nouvelle version `compute_color_simd_v3` qui fait appel à `atan2f_approx_simd`. Normalement à partir de cette version vous ne devriez plus avoir de boucle dans la fonction `compute_color_simd_v3`.

Travail à faire 3 Vérifiez visuellement que le code fonctionne toujours quand vous l'exécutez. Comparez le temps d'exécution par rapport aux versions précédentes. Que constatez-vous ? Comment l'expliquez-vous ?

Tips Quelques fonctions SIMD MIPP bien utiles :

— `mipp::Msk<mipp::N<float>()>` (déclaration d'un registre vectoriel de type « masque »),

— `mipp::blend` (l'implémentation d'une condition ternaire en vectoriel),

— `mipp::cvt<int, float>` (pour convertir un registre vectoriel entier en registre vectoriel flottant).

2.6 « À fond les ballons » dans la vectorisation (`simd_v4`)

Travail à faire 1 En partant de la version précédente, écrire une nouvelle version `compute_color_simd_v4` dans laquelle vous inlinerez complètement l'appel à la fonction `atan2f_approx_simd`. En d'autres termes, il ne doit plus y avoir d'appels à `atanf_approx_simd` et à `atan2f_approx_simd` dans la fonction `compute_color_simd_v4`.

Travail à faire 2 Certains registres contenant des constantes peuvent être éliminés, supprimez les doublons.

Travail à faire 3 Vérifiez visuellement que le code fonctionne toujours quand vous l'exécutez. Comparez le temps d'exécution par rapport aux versions précédentes. Que constatez-vous ? Comment l'expliquez-vous ?

2.7 « Ça pique les yeux » la vectorisation (`simd_v5`)

Travail à faire 1 En partant de la version précédente, écrire une nouvelle version `spin_compute_simd_v5` dans laquelle vous inlinerez complètement l'appel à la fonction `compute_color_simd`.

Travail à faire 2 Sortez les constantes des boucles, regardez si certains calculs ne sont pas faits « trop de fois ». Par exemple, certains calculs ne dépendent pas de `j`... Vous devez aussi trouver des enchaînements de multiplications et d'additions et les remplacer par des opérations de type FMA (*Fused Multiply and Add*). Ce type d'opérations est maintenant très généralisé dans les jeux d'instructions SIMD et ils permettent théoriquement de doubler l'efficacité.

Travail à faire 3 Vérifiez visuellement que le code fonctionne toujours quand vous l'exécutez. Comparez le temps d'exécution par rapport aux versions précédentes. Que constatez-vous ? Comment l'expliquez-vous ?

Tips Quelques fonctions SIMD MIPP bien utiles :

— `mipp::fmadd(a,b,c)` (FMA : $a \times b + c$),

— `mipp::fnmadd(a,b,c)` (FMA : $-(a \times b) + c$).

2.8 « Cerise (de Groupama) sur le gâteau », unrolling (`simd_v6`)

Dans cette ultime version, spoilée par le titre, nous allons unroller la boucle sur les `j`. Mais avant d'y aller comme des grands malades, il est important de mettre un peu d'ordre sinon vous n'allez pas y arriver... Reprenez la version 5, et faites un copier/coller pour la version 6. Postfixez toutes vos variables qui dépendent de `j` par `_j0`. Par exemple, si vous avez un registre vectoriel `mipp::Reg<float> r_atan2f_in2` vous devez le renommer en `mipp::Reg<float> r_atan2f_in2_j0`.

Travail à faire 1 En partant de la version précédente, écrire une nouvelle version `spin_compute_simd_v6u2` où vous déroulerez la boucle sur les `j` à l'ordre 2.

Travail à faire 2 En partant de la version précédente, écrire une nouvelle version `spin_compute_simd_v6u4` où vous déroulerez la boucle sur les `j` à l'ordre 4.

Travail à faire 3 Vérifiez visuellement que le code fonctionne toujours quand vous l'exécutez. Comparez le temps d'exécution par rapport aux versions précédentes. Que constatez-vous ? Comment l'expliquez-vous ?