Partie 1.1 : optimisation ILP

Exercice 1 En considérant les définitions de fonctions suivantes :

```
long min(long x, long y) {return x<y ? x :y;}
long max(long x, long y) {return x>y ? x :y;}
void incr(long* xp, long v){ *xp+=v;}
long square(long x) { return x*x;}
```

ainsi que les bouts de codes suivants :

[

```
for(i=min(x,y); i<max(x,y); incr(&i,1))
    t+= square(i);
```

B

```
for(i=max(x,y)-1; i>=min(x,y); incr(&i,-1))
    t+= square(i);
```

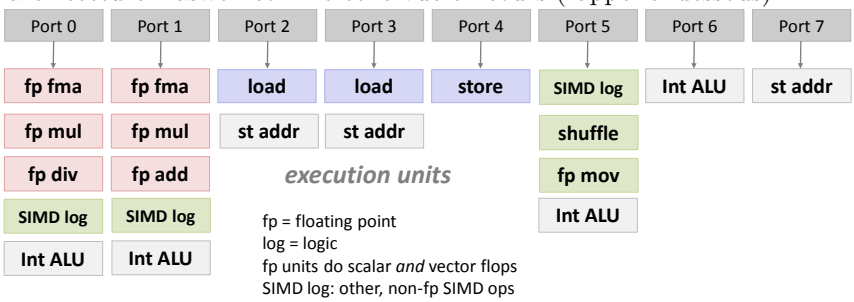
C

```
  long low= min(x,y);
  long high= max(x,y);
  for(i=low; i<high; incr(&i,1))
    t+= square(i);
```

Remplir le tableau ci-dessous indiquant le nombre d'appels de fonctions pour chacun des bouts de code. Vos réponses doivent être exprimées en fonction de x et de y.

Code	`min`	`max`	`incr`	`square`
A				
B				
C				

Exercice 2 Nous souhaitons déterminer une borne inférieur sur le nombre de cycles d'un programme sur une architecture Haswell comme celle vue en cours (rappel ci-dessous).



Execution Unit (fp)	Latency [cycles]	Throughput [ops/cycle]	Gap [cycles/issue]
fma	5	2	0.5
mul	5	2	0.5
add	3	1	1
div (scalar)	14-20	1/13	13
div (4-way)	25-35	1/27	27

- Gap = 1/throughput
- *Intel calls gap the throughput!*
- Same exec units for scalar and vector flops
- Same latency/throughput for scalar (one double) and AVX vector (four doubles) flops, except for div

On se donne le programme suivant en considérant que `u,x,y,z` sont des vecteurs de `double` de taille n.

```
for (size_t i=0;i<n;i++)
    z[i]=z[i]+u[i]*u[i]*u[i]+x[i]*y[i]*z[i];
```

1. Donner la complexité exacte ce programme (sans compter la gestion de la boucle)
2. En s'appuyant sur le mapping des opérations et de leurs débits sur l'architecture Haswell, donner une borne inférieure sur le nombre de cycle de ce programme. Vous ferez deux analyses : une qui n'utilise pas l'opération FMA et l'autre oui. On ne prend pas en compte la dépendance des données.

Exercice 3 Dans la suite de vos TP, nous aurons besoin d'évaluer les performances de vos programmes. Pour cela nous allons définir une classe nous permettant de faire des mesures de performance. Écrire un classe C++ `EvalPerf` qui permet de manipuler le temps ainsi que les cycles CPU. En particulier, on désire que cette classe soit utilisée comme ceci :

```
#include "eval-perf.h"
int main(){

  EvalPerf PE;

  PE.start();
  ma_fonction();
  PE.stop();
  std::cout<<"nbr cycles: "      <<PE.nb_cycle()<<std::endl;
  std::cout<<"nbr secondes: "    <<PE.second()<<std::endl;
  std::cout<<"nbr millisecondes: "<<PE.millisecond()<<std::endl;
  std::cout<<"CPI="<<PE.CPI(N)<<std::endl;;

  return 0;
}
```

Pour l'évaluation du temps vous utiliserez la bibliothèque C++ `chrono` disponible avec `#include <chrono>`. En particulier, il vous faudra utiliser les types de données prédéfinis : `std::chrono::time_point` et `std::chrono::high_resolution_clock>`. Pour le comptage des cycles, vous vous appuierez sur le bout de code suivant :

```
#include <x86intrin.h>
uint64_t rdtsc(){
    unsigned int lo,hi;
    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
    return ((uint64_t)hi << 32) | lo;
}
```

qui retourne un entier représentant la valeur du compteur de nombre de cycles. En particulier, si on appelle cette fonction à deux moments distincts dans le programme, la différence des valeurs vous donnera le nombre de cycles exécutés entre ces deux moments.
Vous écrirez un programme de test pour tester votre classe. Pour vérifier si votre classe est correcte vous comparerez vos résultats avec ceux générés par l'utilisation de l'utilitaire `perf` de linux. En particulier la commande `perf stat -e cycles ./prog` vous donne le nombre de cycles exécutés par la programme `prog` ainsi que la durée d'exécution.

Exercice 4 Écrire un programme de test qui calcule la somme préfixe d'un tableau d'entier (pas d'optimisation pour l'instant). Votre fonction prendra un tableau d'entier A et le remplacera par sa somme préfixe. Rappel : `sommePrefixe([1,3,10,2]) = [ 1,4,14,16 ] = [1, 1+3, 1+3+10, 1+3+10+2])`. Vous utilisez votre classe `EvalPerf` pour évaluerez les performances de votre fonction. En particulier, vous afficherez le temps et le CPI. Afin de voir l'impact des optimisations du compilateur vous mesurerez les performances avec différentes options de compilation : -O0,-O1, -O2 et -O3.
Attention, votre code devra utiliser plusieurs fois votre fonction pour en évaluer une performance moyenne. L'idée est que pour chaque fonction vous jouerez avec la taille du vecteur et le nombre de fois que la fonction est appelée. A minima il vous faudra reporter les résultats dans un fichier pour pouvoir comparer les différents résultats.

Exercice 5 Soit un polynôme $P(X) = p_0 + p_1X + \cdots + P_nX^n$, l'évaluation de $P(\alpha)$ pour un α donné consiste à calculer la valeur $P(\alpha) = p_0 + p_1\alpha + \cdots + P_n\alpha^n$. Écrire deux fonctions qui calculent l'évaluation d'un polynôme P en un point α quelconque. Les coefficients du polynômes seront stockés dans un vecteur de taille $(n+1)$ pour un polynôme de degrés n.
La première fonction devra calculer toutes les puissances successives de α alors que la seconde utilisera la méthode de Horner (cf. Méthode de Ruffini-Horner sur Wikipedia).

Calculer le nombre exact d'additions et de multiplications de chacun des algorithmes. Évaluer les performances de vos fonctions à la fois avec des coefficients et α qui sont soit des entiers soit des nombres flottants. Est-ce que le décompte du nombre d'opérations est corrélé aux performances obtenues ? Essayer de donner une explication (vous pouvez regarder le code assembleur).

Exercice 6 Reproduire les codes vus en cours pour le calcul de la fonction `reduce` avec les opérateurs $+$ et $\times$.

1. Implémenter l'ensemble des algorithmes et tester leur efficacité. Vous essaierez pour chacun des codes de donner une estimation *apriori* du nombre moyen de cycles par instruction (CPI).
2. Proposer de nouveaux algorithmes qui déroulent les boucles sur plus de deux itérations. Trouver expérimentalement le nombre optimal d'itérations à dérouler.

Exercice 7 Nous allons nous intéresser à optimiser le code de la fonction suivante `slowperformance1`, en proposant plusieurs niveau d'optimisation et en comparant le nombre de cycles d'exécution des versions optimisées. Pour cet exercice vous utiliserez l'option de compilation `-O3 -fno-tree-vectorize` et vous avez droit à faire des transformations arithmétiques qui ne garantissent pas le même calcul flottant. On pourra faire l'hypothèse que si les résultats sont identiques à 10^{-3} prêts alors ils sont identiques. Vous comparer les nombres de cycles de chacune de vos optimisations.

```
#include <cmath>
#define C1 0.2f
#define C2 0.3f

void slowperformance1(float *x, const float *y, const float *z, int n) {
  for (int i = 0; i < n - 2; i++) {
    x[i] = x[i] / M_SQRT2 + y[i] * C1;
    x[i+1] += z[(i % 4) * 10] * C2;
    x[i+2] += sin((2 * M_PI * i) / 3) * y[i+2]; }
}
```

1. Afin d'avoir une valeur de référence, faite un programme de test qui génère trois vecteurs de float X, Y, Z ayant des coefficients entiers pris aléatoirement entre 0 et 100. Calculer le nombre de cycles pour exécuter la fonction `slowperformance1` avec des vecteurs de taille 10 000.
2. Faire une fonction `slowperformance2` qui ne fait que la mise à jour de x[i] à chaque itération (il faudra gérer les deux premières et les deux dernières cases du tableau en dehors de la boucle). Comparer le nombre de cycles avec la fonction `slowperforance1`
3. Faire une fonction `slowperformance3` qui déroule la boucle 3 fois et remplace l'appel à la fonction sin par l'utilisation de valeurs constantes (vous devrez faire un peu de trigonométrie). Comparer le nombre de cycles.
4. Faire une fonction `slowperformance4` qui supprimer les divisions et qui regroupe les multiplications ayant une même opérande (si possible utiliser des constantes comme opérande pour certaines multiplication). Comparer le nombre de cycles.
5. Faire une fonction `slowperformance4` qui déroule la boucle 12 fois et qui supprime la lecture de la valeur `z[(i % 4)% * 10]`. Pour que cela soit plus simple, repartez du code de la fonction `slowperformance3` et dupliquer le 4 fois.

Exercice 8 Reprendre les codes d'évaluation de polynômes et essayer d'optimiser les performances. Votre objectif consiste à éliminer les appels de fonction inutiles, d'éviter les écritures/lectures dans la mémoire et de dérouler les boucles pour exhiber de l'ILP dans vos codes.

Partie 1.2 : vectorisation SIMD

Exercice 9 Nous souhaitons écrire des variantes SIMD AVX2 de la fonction `reduce` qui calcule la réduction d'un vecteur par les opérations + ou × (comme celles vues en cours). Bien entendu vos variantes vont dépendre du type de données est de l'opération concernée.

1. Proposer une variante SIMD pour le calcul du produit des éléments d'un vecteur contenant des doubles. Au niveau algorithmique cette variante devra dérouler la boucle 4 fois pour exhiber du calcul SIMD sur 4 voies.
2. Proposer une variante SIMD pour le calcul de la somme des éléments d'un vecteur contenant des `int32_t`. Au niveau algorithmique cette variante devra dérouler la boucle 8 fois pour exhiber du calcul SIMD sur 8 voies.

Vous comparerez les performances avec la fonction `Reduce3` qui ne fait aucun déroulage de boucle mais qui utilise une variable temporaire pour faire les calculs.. Attention, les performances de la fonction `Reduce` doivent être calculées sans l'activation des SIMD, option de compilation `-fno-tree-vectorize`. Par contre, comme vos optimisations utilisent du SIMD, il faudra activer le jeu d'instruction SIMD adéquat (option de compilation `-O3 -mavx -mavx2`.
Refaite vos comparaison en supprimant l'option de compilation `-fno-tree-vectorize`. Que pouvez-vous conclure ?
Si ce n'est pas déjà le cas, essayer d'améliorer les performances de vos version SIMD pour être aussi efficace ou meilleur que la fonction `Reduce3` vectorisée par le compilateur.

Exercice 10 Dans le même esprit que l'exercice précédent écrire une fonction qui calcule le minimum d'un tableau de float. Bien entendu vous devrez utiliser les instructions SIMD adéquates pour ce calcul. Vous comparerez vos performances avec celle d'une fonction naive qui fait de simples comparaisons avec des if. Vous comparerez également vos performances avec la fonction `std::min_element` fournit par la bibliothèque standard STL avec `#include <algorithm>`.

Exercice 11 Nous souhaitons introduire des instructions SIMD pour améliorer les performances de l'évaluation d'un polynôme $P(X)$ en α. Pour cela nous devons trouver un moyen d'exhiber du parallélisme. Une façon simple d'introduire du parallélisme est d'utiliser un changement de variable. En effet, si l'on considère le polynôme $P(X) = p_0 + p_1X + p_2X^2 + p_3X^3 + p_4X^4 + p_5X^5$ on peut le voir comme le polynôme :

$$\bar{P}(X) = (p_0 + p_1X) + X^2(p_2 + p_3X) + X^4(p_4 + p_5X)$$

En substituant X^2 par une nouvelle variable Y on obtient le polynôme

$$\bar{P}(X, Y) = (p_0 + p_1X) + Y(p_2 + p_3X) + Y^2(p_4 + p_5X)$$

que l'on peut réécrire

$$\bar{P}(X, Y) = (p_0 + p_2Y + p_4Y^2) + X(p_1 + p_3Y + p_5Y^2).$$

Par définition, on a $P(\alpha) = \bar{P}(\alpha, \alpha^2)$. En posant $P_1(Y) = p_0 + p_2Y + p_4Y^2$ et $P_2 = p_1 + p_3Y + p_5Y^2$ on trouve que

$$P(\alpha) = P1(\alpha^2) + \alpha P_2(\alpha^2).$$

On peut donc clairement calculer $P_1(\alpha^2)$ et $P_2(\alpha^2)$ en parallèle. Cette méthode se généralise très facilement pour obtenir un parallélisme de k en substituant X^k par Y et en faisant k évaluation de polynômes en α^k. Proposer une version SIMD de la méthode d'évaluation classique (pas Horner). On considérera ici que le polynôme P et le point α contiennent des `double`.

Exercice 12 Le produit de deux matrices A et B de taille respectivement $m \times k$ et $k \times n$ consiste à faire une triple boucle sur les valeurs m, n, k. En effet, l'entrée de la matrice produit $C = A \times B$ vérifie $C[i, j] = \sum_{i=0}^{k-1} A[i, k] \times B[k, j]$ (en considérant que les indices des éléments dans les matrices commencent à 0). On considère uniquement des matrices à coefficients flottant double précision.

1. On souhaite utiliser la convention de stockage des matrices en C dans un tableau unidimensionnel. Pour cela il suffit de stocker les lignes de la matrices les unes à la suite des autres dans votre tableau à une dimension. Soit v le vecteur de taille mn stockant une matrice à m lignes et n colonnes, comment récupérer l'élément à la ligne i et la colonne j dans la matrice ?

2. Écrire les six fonctions possibles pour faire ce produit de matrices en inter-changeant l'ordre des trois boucles. Pour l'instant on se contentera du code naïf, pas d'optimisation. Vous mesurerez les performances de toutes les fonctions et vous déterminerez laquelle est la meilleure.
3. À partir de votre meilleure fonction, vous proposerez une nouvelle version qui utilise des instructions SIMD. En particulier, vous devrez utiliser l'opération de FMA. Vous comparerez les performances de cette fonction avec celle sans SIMD. Attention, pensez à utiliser l'option de compilation `-fno-tree-vectorize` pour supprimer l'auto vectorisation par le compilateur.
4. Comparez les performances de vos fonction lorsque vous activez l'auto vectorisation (`-O3 -mfma -mavx`). Que pouvez-vous conclure ? Essayer d'améliorer l'ILP de votre fonction avec SIMD pour que les performances soient meilleures.

Exercice 13 L'opération de transposition d'une matrice consiste à échanger ses entrées au position (i, j) et (j, i). Autrement dit

$$\text{transpose}\left(\begin{bmatrix}14 & 10 & 3 & 15\\ 8 & 4 & 4 & 6\\ 4 & 10 & 16 & 12\\ 15 & 1 & 1 & 10\end{bmatrix}\right) = \begin{bmatrix}14 & 8 & 4 & 15\\ 10 & 4 & 10 & 1\\ 3 & 4 & 16 & 1\\ 15 & 6 & 12 & 10\end{bmatrix}$$

1. Écrire une fonction naïve (sans optimisation) qui prend une matrice carré de `int32_t` en entrée et qui la modifie en sa matrice transposée.
2. Proposer une variante spécifique pour le cas des matrices 4×4. L'idée est de représenter la matrice au travers de 4 registres SIMD `r0,r1,r2,r3` qui correspondent au 4 lignes de la matrices. À partir de cette représentation, trouver une succession d'instructions SIMD qui permet de stocker la transposée de cette matrice dans `r0,r1,r2,r3`. En s'appuyant sur l'exemple précédent, la matrice initiale est stockée comme `r0=[14 10 3 15]`, `r1=[8 4 4 6]`, `r2=[4 10 16 12]` et `r3=[15 1 1 10]`. Après l'appel de la fonction, on obtient `r0=[14 8 4 15]`, `r1=[10 4 10 1]`, `r2=[3 4 16 1]` et `r3=[15 6 12 10]`. L'idée est de réussir à regrouper les éléments de telle sorte que les 128 bits de poids fort et poids faible de chaque registre correspondent à des données 128-bits de poids fort ou faible de la sortie finale. En garantissant, cela on évite de croiser les voies 128-bits.

   Par exemple, si on obtient `[10 4 3 4]` tout va bien car `[10 4]` et `[3 4]` correspondent au premier 128 bits de `r1` et `r2`. Par contre si on obtient `[8 4]` il faudra croiser les voies 128-bits car cela correspond au 128-bits centraux de `r1`.
3. Proposer une nouvelle fonction de transposition de matrice qui utilisent votre fonction optimisée. Vous ferez l'hypothèse que les matrices sont carrées de taille un multiple de 4. Comparez les performances de votre nouvelle fonction avec l'ancienne.

Exercice 14 Nous souhaitons évaluer les performances des algorithmes de tri classiques. Pour cela vous aller écrire les codes de deux fonctions de tri. La première utilisera un algorithme ayant une complexité de $O(n^2)$ alors que la seconde utilisera un algorithme ayant une complexité de $O(n \log n)$. On supposera que les éléments peuvent être comparés via les opérateurs de comparaison standard.
Évaluer les performances de vos fonctions sur des tableaux de tailles 2^k pour k allant de 8 à 18. Vous testerez pour des entiers de $8, 16, 32$ et 64 bits. Vous comparerez les performances de vos implantations avec celle de la fonction `sort` disponible dans la bibliothèque STL de C++ (`#include <algorithm>`.