Partie 1.1

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);
```

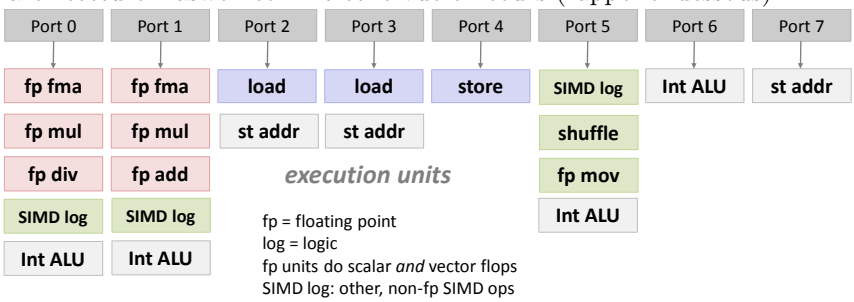
C

```
  long low= min(x,y);
  long max= 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 programe 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 degrès 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 Reprendre les codes de somme préfixe et d'évaluation de polynômes et essayer d'optimiser les performances. Votre objectif consiste à éliminier 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.

Exercice 8 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).

1. Écrire les six fonctions possibles pour faire ce produit de matrices en interchangeant l'ordre de ces trois boucles. Pour l'instant on se contentera du code naif, pas d'optimisation. Vous mesurerez les performances de toutes les fonctions et vous déterminerez laquelle est la meilleure.
2. 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 ? Réécriver le code de votre meilleure fonction avec cette convention de stockage et comparer les performances.
3. Proposer une modification de l'algorithme de multiplication qui déroule les boucles par pas de 2 et essayer de le programmer en minimisant les accès mémoires.

Dans un deuxième temps vous essaierez d'optimiser le code pour avoir un meilleur ILP.

Exercice 9 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 implémentations avec celle de la fonction `sort` disponible dans la bibliothèque STL de C++ (`#include <algorithm>`.