1 Optimisation ILP/IPC

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 :

A

```
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 On se donne un processeur superscalaire qui à les caractéristique suivantes :

- gestion *out of order* des instructions
- 2 multiplication par cycle, exécutées sur les ports 1 et 2
- 1 addition (ou soustraction) par cycle, exécutée sur le port 1

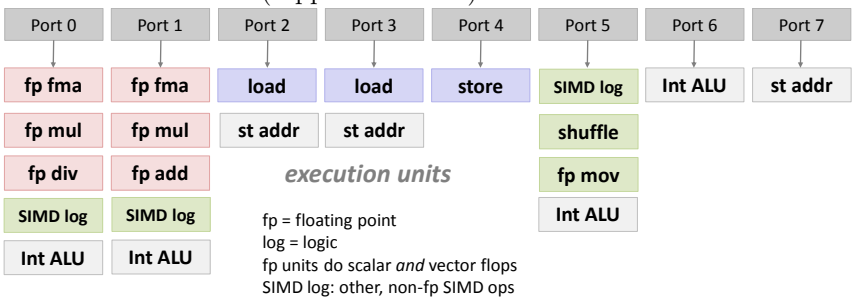
On se donne les trois fonctions ci-dessous qui calculent toutes le même résultat.

```

void f1(const float a){
    return 1 - a*a*a*a;
}
void f2(const float a){
    float x=1-a;
    float y=1+a;
    return (1+a*a) * x * y;
}
void f3(const float a}{
    float x=a*a;
    return  (1 - x) * (1 + x);

}
```

On considère également que le compilateur ne fait aucune optimisation sur l'exécution des instructions (les instructions sont exécutées tel que décrit dans le code) ; les opérations entières pour la gestion de la boucle ainsi que le chargement des données dans les registres seront à ignorer dans la suite.

1. Calculer l'ILP pour chacune de ces fonctions.
2. Calculer l'IPC minimal pour chacune de ces fonctions en considérant le processeur décrit.
3. On modifie notre processeur pour qu'il puisse faire 2 additions/soustractions par cycle sur les ports 1 et 2 ; et 1 multiplication par cycle sur le port 1. Quelles sont les fonctions qui obtiennent un meilleur IPC ?

Exercice 3 Nous souhaitons estimer les performances 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) div (4-way)	14-20 25-35	1/13 1/27	13 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]+x[i]*y[i]*z[i];
```

On considère que le compilateur est capable de réordonner les opérandes de toutes les opérations pour améliorer les performances (par exemple il peut changer $a+b+c=(a+b)+c$ en $(b+c)+a$). On n'autorise pas l'utilisation de l'opération FMA.

1. Donner la complexité exacte ce programme (sans compter la gestion de la boucle).
2. Calculer l'ILP de ce programme.
3. En s'appuyant sur le mapping des opérations et de leurs débits sur l'architecture Haswell, calculer l'IPC de ce programme. Vous ferez deux analyses : une ou le compilateur n'est pas capable de dérouler la boucle et une autre ou il le peut.

Exercice 4 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 une 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(){
  int n,N;
  EvalPerf PE;

  PE.start();
  ma_fonction(n);
  PE.stop();
  N=flops_ma_fonction(n);
  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;;
  std::cout<<"IPC="<<PE.IPC(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 5 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 utiliserez votre classe `EvalPerf` pour évaluerez les performances de votre fonction. En particulier, vous afficherez le temps et le CPI/IPC. 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. Vous ne mesurerez ni le temps pour allouer/désallouer le tableau ni celui pour lui affecter des valeurs (aléatoires ou pas). L'idée est qu'en fonction de la taille du tableau vous exécuterez un nombre suffisant de fois votre fonction pour que le temps global soit de l'ordre de la seconde. A minima il vous faudra reporter les résultats dans un fichier pour pouvoir comparer les différents résultats.

Exercice 6 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 pour des polynômes à coefficients entiers (32 bits) ou des nombres flottants (64 bits). Bien entendu le type pour α sera le même que celui des coefficients du polynôme. 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 7 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 8 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 9 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 10 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 11 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 12 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 13 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 matrice 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 14 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 `int64_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 aux 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`. Aide : il faudra utiliser les instructions SIMD de manipulation de données dans les registres.

   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. Une fois que vous en êtes la, il vous faudra utiliser la fonction `_mm256_permute2x128_si256` pour permuter les données par paquet de 128 bits. En faisant cela on minimise le croisement des 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 15 Nous souhaitons évaluer les performances des algorithmes de tri de tableau. 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 tableaux contenant 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>`.

Partie 1.3 : cache et complexité spatiale

Exercice 16 On cherche à déterminer l'intensité opérationnelle d'une fonction qui calcule le produit d'une matrice élévée au carré par un vecteur. Autrement dit, à partir d'une matrice A et d'un vecteur v on souhaite calculer $A^2v = A \times (Av)$. Les élements de la matrice et du vecteur sont des `double` et on utilise un vecteur temporaire pour stocker le résultat intermédiaire (Av).
Vous considérerez un cache de taille γ avec des blocs de taille 64 octets.

1. Déterminer la complexité arithmétique exacte $W(n)$ pour ce calcul avec une matrices carré de taille $n \times n$. Cela correspond au nombre exact d'opérations sur les double durant le calcul.
2. Donner un encadrement pour la valeur de $Q(n)$ qui correspond au nombre d'octets transférés entre la mémoire et le cache de données durant le calcul. Vous ne tiendrez compte que des accès en lecture des données et vous considérerez uniquement des mouvements mémoire de type *cold miss*.
3. Calculer une borne supérieure sur l'intensité opération $I(n)$ de ce calcul.
4. À votre avis pour quelle valeur de n la borne calculée précédemment sera assez précise (en fonction du cache considéré).
5. Le code suivant modifie le calcul de A^2v de telle sorte que l'on applique le calcul sur la sous-matrice de A composée des colonnes d'indice j tel que $j \bmod k = 0$ pour une valeur de k donnée (le calcul prédécent correspond à $k = 1$).

```
void mat2vec(size_t n, const double* A, double *y, double * tmp, size_t k){
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; j+=k)
      tmp[i] += A[i*n + j]*v[j];

  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; j+=k)
      y[i] += A[i*n + j]*tmp[j];
}
```

   Donner une nouvelle borne sur l'intensité opérationnelle de ce calcul dans les cas $k = 7$ et $k = 16$.
6. Programmer la fonction `mat2vec` donnée ci-dessus. Proposer une nouvelle fonction `mat2tvec` similaire qui calcule AA^Tv ou A^T représenter la matrice transposée de A. Bien évidemment, il ne faudra pas calculer explicitement A^T. Comparer les performances de vos deux fonctions pour des matrices carées de taille 2^t pour $t \in \{1, 2, 3, \ldots, 12\}$ et pour des valeurs de $k \in \{1, 7, 16\}$. Essayer de donner une explication aux performances obtenues. Vous pouvez obtenir la taille des caches de vos machines en utilisant la commande unix : `getconf -a | grep CACHE`.

Exercice 17 On considère un cache de données ayant les caractéristiques suivantes :

- block size : B= 16 octets
- associativité : E=8
- nombre de blocs par voie : S=256
- politique de remplacement de bloc : *LRU : Least Recently Used*

ainsi que le code suivant :

```
double sum(double T[65536]){
    double res=0.0;
    for(size_t i=0;i<65536;i++)
        res+=T[i];
    return res;
}
```

On considèrera que le programme stocke la variable `res` dans un registre.

1. Donner le nombre minimum de cache miss que fera ce programme. Expliquer votre calcul.
2. Donner le nombre maximum de cache miss que fera ce programme. Expliquer votre calcul.
3. Donner une borne inférieure sur le taux de cache miss de ce programme.

Maintenant nous considérons le code suivant :

```
float sum(float *T, size_t n){
    float res=0.0;
    for(size_t i=0;i<n;i++)
        res+=T[i];
    return res;
}
```

Donner un borne inférieure sur le taux de cache miss de ce programme.

Exercice 18 On considère l'opération de transposition d'une matrice 2×2 faite à partir du code suivant :

```
void transpose( int32_t A[2][2], const int32_t B[2][2]){
    for(size_t i=0;i<2;i++)
        for(size_t j=0;j<2;j++)
            A[i][j]=B[j][i];
}
```

sur machine ayant les caractéristiques suivantes :

- le tableau A commence à l'adresse 0 et le tableau B commence à l'adresse 16
- il n'y a qu'un seul cache L1 avec des blocs de taille 8 octets
- le cache est *direct-mapped. write-allocate/write-through* d'une taille totale de 16 octets.

1. En considérant que les variables `i,j` sont en registres, indiquer le pattern d'accès H/M (hit ou miss) pour A et B
2. même question mais avec un cache de taille 32 octets

Exercice 19 On reprend le principe de la somme prefixe vue en cours pour l'appliquer aux lignes d'une matrice. L'idée est de transformer une matrice carrée A de telle sorte que sa i-ème ligne $A_{i,*} = \sum_{j=i}^{n} A_{j,*}$. Autrement dit, après le calcul on a ajouté à chaque ligne la somme de toute les lignes située en dessous dans la matrice initiale :

$$\begin{bmatrix} 14 & 10 & 3 & 15 \\ 8 & 4 & 4 & 6 \\ 4 & 10 & 16 & 12 \\ 15 & 1 & 1 & 10 \end{bmatrix} \Rightarrow \begin{bmatrix} 41 & 255 & 24 & 43 \\ 27 & 15 & 21 & 28 \\ 19 & 11 & 17 & 22 \\ 15 & 1 & 1 & 10 \end{bmatrix}$$

En considérant un stockage unidimensionnel par ligne des matrices, le code suivant adapte l'algorithme de somme préfixe, en commencant par la dernière ligne et en remontant les lignes au fur et à mesure.

```
void row_prefixsum(int32_t *A, int n){
    for(int i=n-2;i>=0;i--)
        for(int j=0;j<n;j++){
            int32_t r1 = A[i*n+j];
            int32_t r2 = A[(i+1)*n+j];
            A[i*n+j] = r1+r2;
        }
}
```

On considère que ce code s'exécute sur une machine ayant un cache avec une taille de bloc de $B = 32$ octets, et que le cache est *direct mapped* de taille 4096 octets, avec une politique d'écriture de type *write-back/write-allocate*. L'exécution nous garantie que les variables `i,j,r1,r2` ne génèrent aucun défaut de cache et que le tableau `A` commence à l'adresse 0.

1. calculer le taux de cache miss lorsque $n = 256$
2. calculer le taux de cache miss lorsque $n = 4096$
3. calculer le taux de cache miss lorsque $n = 2056$

Exercice 20 On considère le problème de multiplication de matrices carrées de taille 4×4 sur une machine ayant un cache *fully associative* avec une politique d'éviction des blocs de type LFU (*Least Frequently Used*) et une écriture de type *write back/write allocate*. Le cache est de taille 64 octets avec des blocs de taille 16 octets. On considère le produit de matrice $C = A \times B$ avec des entrées de type `double`. Les matrices utilisent un stockage linéaire *row major* (les lignes des matrices les unes après les autres).

1. Donner le taux de cache miss pour la variante `ikj` qui calcule le produit en ordonnant les boucles par les lignes de C, puis les colonnes de A et enfin les colonnes de B.
2. Donner le taux de cache miss pour la variante `jki` qui calcule le produit en ordonnant les boucles par les colonnes de B, puis les colonnes de A et enfin les lignes de C.
3. Donner les taux de cache misse pour les deux variantes précédentes mais cette fois-ci en considérant des matrices contenant des `float`.

Exercice 21 On se place dans le modèle d'analyse de complexité dit du *Ideal Cache Model* avec un cache de taille M et des lignes de cache contenant B mots. On s'intéresse à analyser de manière asymptotique la complexité en cache du problème suivant :

Problème 1 (Transposition de matrices) *Soit A et B deux matrices contenant des* `double` *de tailles $n \times n$ stockées dans le format* row major*, on souhaite écrire dans B la matrice transposée de A, câd $B = A^T$.*

1. Donner la complexité en cache de ce problème lorsque les matrices A et B tiennent en cache. Vous expliciterez quelles sont les conditions requises sur n, B et M.
2. De la même manière vous donnerez la complexité en cache de ce problème dans le pire cas, en expliquant les conditions sur n, B et M.
3. Démontrer qu'il est possible d'obtenir une complexité en cache de $\mathrm{MT}(n) = O(n^2/B)$ lorsque $M \geq (n+1)B$. Expliquez pourquoi on peut considérer cette complexité comme optimale.
4. Coder une fonction `MatrixTrans_naif` qui effectue l'algorithme naif de transposition de matrices. En récupérant la taille de votre cache L1 avec la commande unix (`getconf -a | grep CACHE`) trouver la valeur de n qui permet de se trouver dans le cas le plus favorable étudié ci-dessus. À partir de l'exécution de votre code sur cette taille vous démontrerez que celui-ci est optimal. Pour cela il vous suffira de comparer vos performances avec celle du code le plus simple dont on sait qu'il est optimal pour les caches. (Aide : celui-ci a été vu en cours). ATTENTION : en pratique les défauts de cache en écriture sont plus coûteux que ceux en lecture.
5. Comme pour le produit de matrices, il est possible d'améliorer la complexité en cache dans le pire cas à $O(n^2/B)$ en utilisant la technique de blocking. L'idée est de découper la matrice en blocs $b \times b$ de telle sorte que $2b^2 < M$. Dans ce cas, la transposition de chacun des blocs a un complexité en cache de $O(b^2/B)$. Comme il y a $O(n^2/b)$ blocs à transposer, on obtient bien une complexité en cache de $O(n^2/B) = O(n^2/b) * O(b^2/B)$. Coder cette algorithme en laissant la possibilité de choisir la taille b des blocs comme paramètre template. Vous comparerez les performances de ce nouveau code avec la version naive codée précédemment. En particulier, vous essaierez de déterminer de manière empirique la meilleure valeur possible pour b avec des matrices de taille 2^k pour $7 \leq k \leq 10$. Que pouvez-vous conclure ?
6. De la même manière que pour le produit de matrices, il est possible d'obtenir un algorithme *cache oblivious* pour la transposition de matrices atteignant une complexité en cache de $O(n^2/B)$. L'idée est d'utiliser un algorithme diviser pour régner. En effet, le problème s'écrit très facilement de manière récursive :
$$A = \begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \Rightarrow B = A^T = \begin{bmatrix} A_0^T & A_2^T \\ A_1^T & A_3^T \end{bmatrix}$$
Une des difficultés pour implanter cet algorithme est de savoir comment représenter une sous-matrice. L'approche classique pour stocker une matrice $m \times n$ de manière linéaire en mémoire est de stocker les m lignes de tailles n les unes à la suite des autres. Par conséquent pour accéder à l'entrée (i, j) d'une matrice M, il suffit d'accéder à $M[i * n + j]$. Imaginons que nous souhaitons découper une matrice M de taille $2m \times 2n$ comme suit :
$$M = \begin{bmatrix} M_0 & M_1 \\ M_2 & M_3 \end{bmatrix}$$
avec tous les blocs M_0, M_1, M_2, M_3 de taille $m \times n$. À partir de M, si je souhaite accéder à l'élément (i, j) de M_3 il suffit de remarquer que l'entrée $(0, 0)$ de M_3 correspond à $M[m * 2 * n + n]$. En effet, $m * 2 * n$ correspond au nombre d'entrées de M avant l'entrée $(0, 0)$ de M_2 (m lignes de taille $2n$). Pour avoir l'entrée $(0, 0)$ de M_3 il suffit donc de décaler de n entrées car les premières lignes de M_2 et M_3 sont stockées l'une derrière l'autre. Du coup, l'entrée (i, j) de M_3 corespond à $M[m*2*n+n+i*2*n+j]$. On a pris l'entrée $(0, 0)$ de M_3 et on avance de $i*2n$ dans la mémoire pour avoir l'entrée $(i, 0)$ de M_3, puis on avance de j pour avoir l'entrée (i, j) de M_3. En C++, on peut se simplifier la vie avec les pointeurs. Imaginons que M est un pointeur sur l'entrée $(0, 0)$ alors le pointeur $M_3 = M + m*2*n+n$. Maintenant pour accéder à l'entrée (i, j) de M_3 il suffit d'accéder à $M_3[i * 2n + j]$. Une façon simple de formaliser une sous-matrice est donc de fournir :
   — un pointeur sur l'entrée $(0, 0)$ de cette sous-matrice,
   — les dimensions m et n,
   — un entier s qui donne la distance entre 2 éléments successif dans une même colonne.

Le code ci-dessous utilise ce formalisme pour afficher un sous-matrice :

```
void print_submatrix(const double* M, size_t m, size_t n, size_t s){
    for (size_t i=0;i<m;i++){
        for (size_t j=0;j<n;j++)
            cout<<M[i*s+j]<<" ";
        cout<<endl;
    }
}

int main(){
    double A[16*16];
    for(size_t i=0;i<256;i++) A[i]=(i+1);

   // affichage de la sous-matrice A3
   print_submatrix(A+16*8+8 , 8, 8, 16);
   return 0;
}
```

Donner le code de la fonction `MatrixTrans_obliv` qui implante l'algorithme de transposition de manière récursive. La signature de cette fonction doit être la suivante :
`void MatrixTrans_obliv(size_t m, size_t n, const double* A, size_t sa, double * B, size_t sb);` avec `m,n` les dimensions des matrices A et B, A la matrice d'entrée, B la matrice de sortie et sa et sb les distances entre 2 éléments successifs dans une colonne de respectivement A et B. Comparez votre fonction avec le code naif et la fonction qui fait du *blocking*.

7. Afin d'améliorer votre version *cache oblivious*, il est possible d'arrêter la récurrence avant le cas $n = 1$. Dans ce cas là, il vous suffira d'utiliser une version de transposition naive pour faire le cas de base. Comme pour la version par bloc, proposer une variante de la fonction `MatrixTrans_obliv` qui prend en paramètre template la taille minimale pour les appels récursifs. Essayer de trouver de manière empirique quelle est la meilleure taille possible et comparer votre code avec les version précédentes (naif, *blocking*).

Exercice 22 Nous souhaitons évaluer les performances de l'algorithme de tri par comptage vue en cours, et particulièrement l'amélioration en cache de ce dernier. Nous souhaitons également comparer ces perfoprmances par rapport à des algorithmes de tri classiques. Nous considérerons le problème de tri pas en place, c'est à dire que le résultat sera écrit dans un tableau différent du tableau à trier. Pour simplifier le problème, nous considérerons directement un tableau d'entier de type `size_t`. Cela nous permettra de remplacer la fonction `key(x)` par `x` dans le code. Dans la suite, lorsqu'il vous sera demandé d'évaluer les performances de vos fonctions de tri, vous testerez vos fonctions sur des tableaux de taille 2^k pour $16 \leq k \leq 24$ dont les entiers sont compris entre 0 et 2^k.

1. Dans un premier temps, transcrire en C++ le code naif vue en cours pour le tri par comptage. Vous utiliserez la signature suivante :

```
void  counting_sort_naif(size_t* R,  const size_t *T, size_t n, size_t min, size_t max);
```

où `R` et `T` sont des tableaus de taille n représentant respectivement la sortie et l'entrée. Les valeurs `min` et `max` représentent la valeur maximum et minimum des éléments du tableau `T`.

2. Implanter la variante du tri par comptage vue en cours qui améliore la complexité en cache. Pour le choix du nombre de bucket vous choisirez $m = 64$ si $n \geq 128$ sinon vous prendrez $m = min(4, n)$.
3. Comparez les performances de vos deux fonctions de tri avec les spécifications de test mentionnées au début de l'énoncé.
4. Pensez-vous que le choix de $m = 64$ est judicieux ? Proposer une meilleure valeur en justifiant votre choix et en démontrant une amélioration en pratique.
5. Proposer une amélioration de l'algorithme naif de tri par comptage (sans bucket) qui utilise le fait que `key(x)=x`. Aide : comme les éléments correspondent au clés, on peut directement remplir le tableau résultat à partir de l'histogramme des valeurs. Coder cette version et comparer ses performances aux variantes précédentes. Vous comparerez également les performances avec la fonction de tri disponible dans la bibliothèque STL de C++ (fonction `std::sort` disponible avec `#include <algorithms>`.

Nous considérons maintenant l'algorithme de tri par comparaison *merge sort*. Ce tri utilise une approche diviser pour régner pour obtenir un tri de complexité quasi-linéaire. Le code python ci-dessous donne une description de cet algorithme.

```python
def merge(T1,T2):
    n = len(T1)+len(T2)
    T=[0]*n
    i1,i2,k=0,0,0
    while i1<len(T1) and i2<len(T2) :
        if (T1[i1]<T2[i2]):
            T[k]=T1[i1]
            i1+=1
        else:
            T[k]=T2[i2]
            i2+=1
        k+=1
    while i1<len(T1) :
        T[k]=T1[i1]
        i1+=1
        k+=1
    while i2<len(T2) :
        T[k]=T2[i2]
        i2+=1
        k+=1
    print(T1,T2,T)
    return T

def mergesort(T):
    n=len(T)
    if n ==1 :
        return T
    else :
        return merge(mergesort(T[:n//2]), mergesort(T[n//2:]))
```

6. Quelle est la complexité asymptotique du nombre de comparaisons effectuées par cet algorithme. Vous justifierez votre analyse.
7. Quelle est la complexité en cache de la fonction `merge` qui fusionne deux tableaux triés. On fera l'hypothèse qu'au moins 3 lignes de cache sont disponibles et que l'on connait les caractéristiques du cache B et M. Vous justifierez votre analyse.
8. À partir du résultat de la question précédente, quelle est la complexité en cache de la fonction `mergesort` donnée ci-dessus. Vous justifierez votre analyse.
9. Donner un code C++ pour *merge sort* qui atteint une complexité en cache de $O(n/B \log(M/B))$. Comparez les performances de ce code avec celles du tri par comptage et déduire lequel est meilleur (pour les paramètres de test précisés en début d'énoncé). Vous proposerez un programme de test qui permet d'arriver à votre conclusion.
10. Trouver un exemple qui montre que *merge sort* peut être plus rapide que le tri par comptage. Vous illustrerez votre exemple au travers d'un programme de test qui utilise vos fonctions.