Programmation Parallèle

Owain Biddulph, Adnane Hamid et Lucas Petit

Mars 2020

1 TP1

1.1 Analyse des sorties de cache

L'objectif de cette première partie du TP est la prise en main du système. On commence par se connecter à la machine en ssh avec la commande suivante.

```
$ ssh progpar22@fusion.centralesupelec.fr
```

On peut utiliser *git clone* {*reporsitory-name*} pour obtenir le dossier du TP, mais l'ayant déjà en local nous avons envoyé le dossier avec *scp*. On peut alors se déplacer dans le sous-dossier *00-Validation* pour cet exercice. Le code étant déjà écrit, on peut lancer la compilation avec la commande suivante.

```
$ make clean; make -j;
```

Quand la compilation est terminée, on peut lancer le job avec la commande suivante.

```
$ qsub subValidation.sh
```

Le fichier *subValidation.sh* contient les informations nécessaires pour le bon traitement du job (path de l'output, path de l'executable, un mail pour suivre le traitement du job etc.).

On peut voir l'évolution du job avec la commande :

```
$ qstat
```

1.2 Moyenne de trois vecteurs

Nous changeons dans le fichier *Makefile* :

```
FILE = tp1_ex2.cpp
PRODUCT = tp1_ex2_vec.exe
```

Nous changeons dans le fichier *Makefile_novec* :

```
FILE = tp1_ex2.cpp
PRODUCT = tp1_ex2_novec.exe
```

Nous changeons dans les fichiers *tp1_ex2.sh* :

```
#PBS -N tp1_ex2
#PBS -m abe -M lucas.petit@student.ecp.fr
```

Puis, on envoie le job au mésocentre :

```
$ qsub tp1_ex2.sh
```

Voici le graphe que l'on obtient lorsque l'on trace le temps écoulé divisé par la taille du vecteur pour une taille de vecteur en i^2 pour i allant de 32 à 4096 avec un pas de 16.

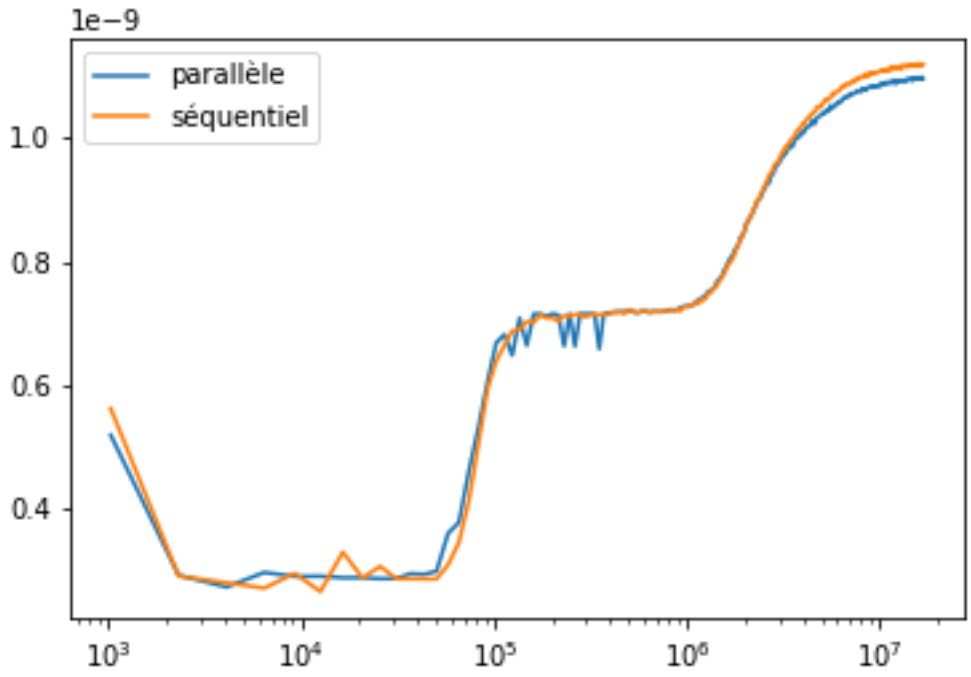

Figure 1: Temps unitaire pour la moyenne de trois vecteurs en fonction de leur taille avec la vectorisation

La valeur élevée pour une taille petite est due au fait que la mesure du temps ne doit pas être précise pour de petits vecteurs. Ensuite, on distingue bien 3 phases pour chaque cache. Le cache L1 est utilisé pour une longueur de vecteur allant jusqu'à 10^5 ; le cache L2 est utilisé pour une longueur de vecteur allant de 10^5 jusqu'à 10^6 ; le cache L3 est utilisé pour une longueur de vecteur allant de 10^6 jusqu'à 10^7.

On remarque bien sûr que le compilateur se charge tout seul de paralléliser le code séquentiel. Voyons maintenant ce qu'il se passe sans la vectorisation.

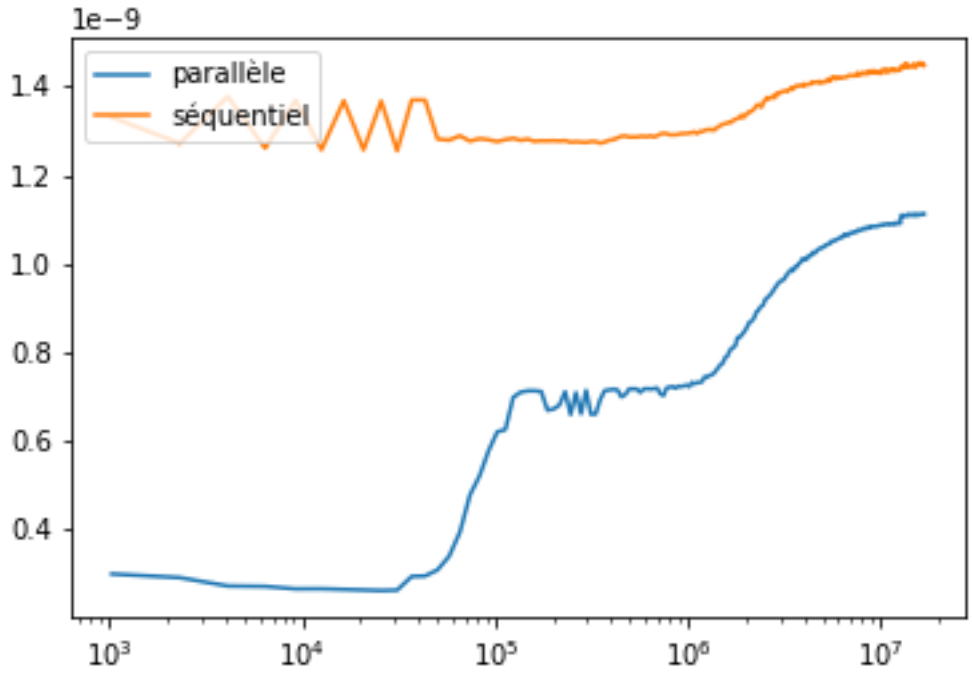

Figure 2: Temps unitaire pour la moyenne de trois vecteurs en fonction de leur taille sans la vectorisation

On voit bien que l'exécution séquentielle du code est plus longue que l'exécution parallèle.

1.3 Produit scalaire de deux vecteurs

Le code en séquentiel est le suivant :

```
float s = 0.0;
for (unsigned long int i = 0; i < size; i++) {
    s += A[i] * B[i];
}
```

Le code utilisant SIMD est le suivant :

```
__m256 s = _mm256_set1_ps(0.0);
for (unsigned long i = 0; i < size; i+=8) {
    __m256 a = _mm256_loadu_ps(&A[i]);
    __m256 b = _mm256_loadu_ps(&B[i]);
    __m256 t = _mm256_mul_ps(a, b);
```

```
    s = _mm256_add_ps(s, t);
}

float dotProduct = 0.0;
for (unsigned long i=0; i<8; i++)
{
    dotProduct += s[i];
}
```

Il est à noter que la vectorisation est désactivée pour les résultats dans la figure 3.

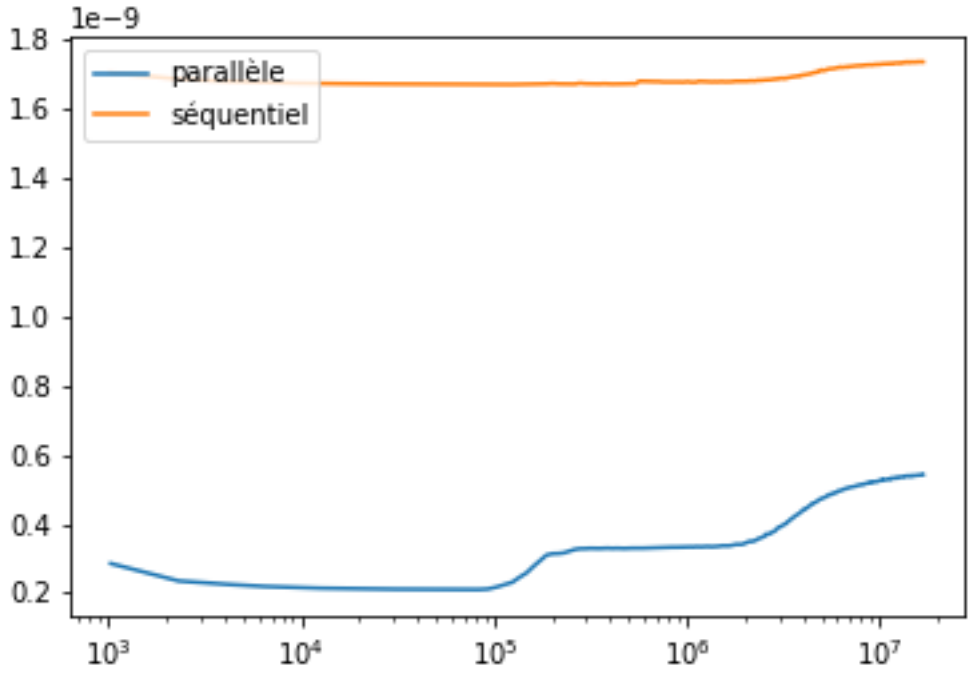

Figure 3: Temps unitaire pour le produit scalaire de deux vecteurs en fonction de leur taille

Outre l'accélération due à l'utilisation d'un paradigme parallèle, on remarque pour la courbe du temps unitaire avec SIMD :

- L'influence de l'overhead au début de la courbe, temps nécessaire à la parallélisation
- Les deux sauts correspondants aux passages L1-L2 puis L2-L3

1.4 Recherche du maximum et du minimum d'un vecteur

Sequenciellement, une recherche de maximum et de minimum peut se faire très simplement par un parcours du vecteur, en comparant à chaque fois l'élément du vecteur avec le maximum et le minimum. Le SIMD va permettre d'effectuer des comparaisons en parallèle sur plusieurs éléments à la fois, avec les fonctions *_mm256_max_ps* et *_mm256_min_ps* qui agissent chacun sur 8 objects *float*. On peut donc initialiser deux vecteurs de comparaison avec le premier élément de vecteur d'entrée avec la fonction *_mm256_set1_ps(A[0])* où A est le vecteur en entrée. Ces vecteurs sont donc comparés au 8 premiers éléments de A, puis au 8 suivant etc. Il faudra alors chercher séquentiellement le maximum ou le minimum de ces vecteurs après les avoir fait glisser sur le vecteur A.

La figure 4 est quasiment exactement la même que pour le produit scalaire de deux vecteurs, les même remarques peuvent être faites.

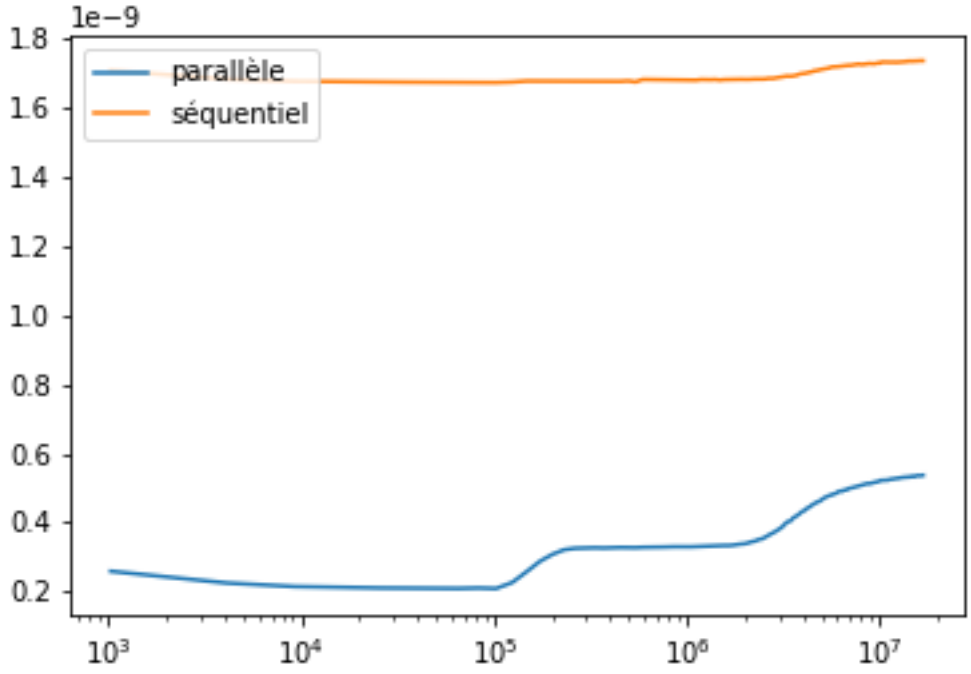

Figure 4: Temps unitaire pour la recherche du maximum et du minimum d'un vecteur en fonction de sa taille

1.5 Filtrage gaussien

Pour effectuer un filtrage gaussien en SIMD, on utilise le code suivant.

```
float parallele(float* A, float* R, unsigned long int size) {
    __m256 two = _mm256_set1_ps(2);
    for (unsigned long i = 1; i < size-1; i+=8) {
        __m256 ai = _mm256_loadu_ps(&A[i-1]);
        __m256 ai1 = _mm256_loadu_ps(&A[i]);
        __m256 ai2 = _mm256_loadu_ps(&A[i+1]);
        __m256 s = _mm256_add_ps(ai, _mm256_mul_ps(ai1, two));
        __m256 t = _mm256_add_ps(s, ai2);
        _mm256_storeu_ps(&R[i-1], t);
    }
    return R[5];
}
```

On charge donc 3 vecteurs décalés avec un pas de 1 du vecteur en entrée (A). On peut alors multiplier celui du milieu par 2 et les additioner pour déterminer 8 éléments consécutifs de la matrice de sortie. La figure 5 donne la vitesse d'exécution en séquentielle et en SIMD du filtrage.

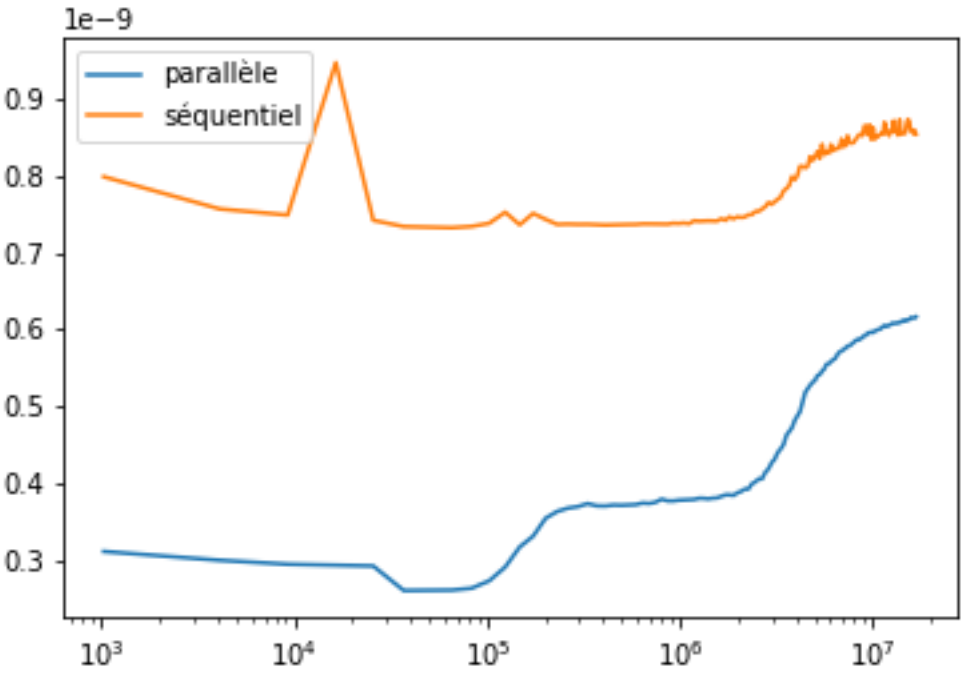

Figure 5: Temps unitaire pour la recherche du maximum et du minimum d'un vecteur en fonction de sa taille

2 TP2 - OpenMP

2.1 Moyenne de deux vecteurs

Le code en séquentiel est le suivant :

```
for (unsigned long int i = 0; i < size; i++) {
    S[i] = (A[i] + B[i])/2;
}
return S[0];
```

Le code utilisant OpenMP est le suivant :

```
#pragma omp parallel for schedule(static)
for (unsigned long int i = 0; i < size; i++) {
    S[i] = (A[i] + B[i])/2;
}
return S[0];
```

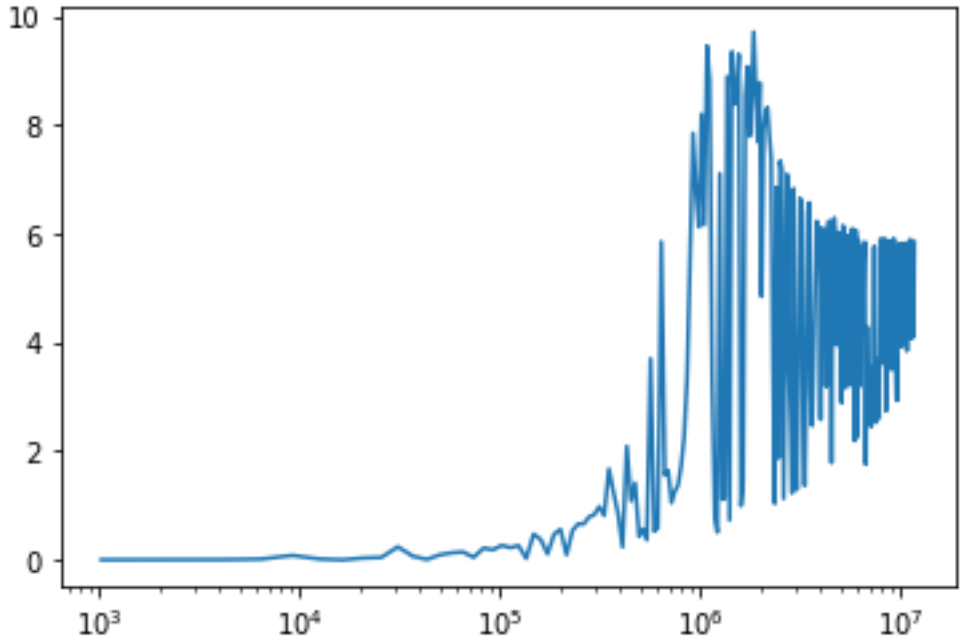

Figure 6: Gain Parallèle/Sequentiel en fonction de la taille des vecteurs moyennés

En plus de l'accélération due au calcul parallèle, on remarque encore l'influence de l'overhead pour des petites tailles pour lesquelles la parallèlisation est moins intéressante. Lorsque la taille dépasse 1 million, le code parallèle est généralement plus performant que le code séquentiel, avec quelques exceptions.

2.2 Produit scalaire de deux vecteurs

Le code en séquentiel est le suivant :

```
double s = 0;
for (unsigned long int i = 0; i < size; i++) {
    s += A[i] * B[i];
}
return s;
```

Le code utilisant OpenMP est le suivant :

```
double s = 0;
#pragma omp parallel for reduction(+:s)
for (unsigned long int i = 0; i < size; i++) {
    s += A[i] * B[i];
 }
return s;
```

On obtient l'évolution de gain suivante.

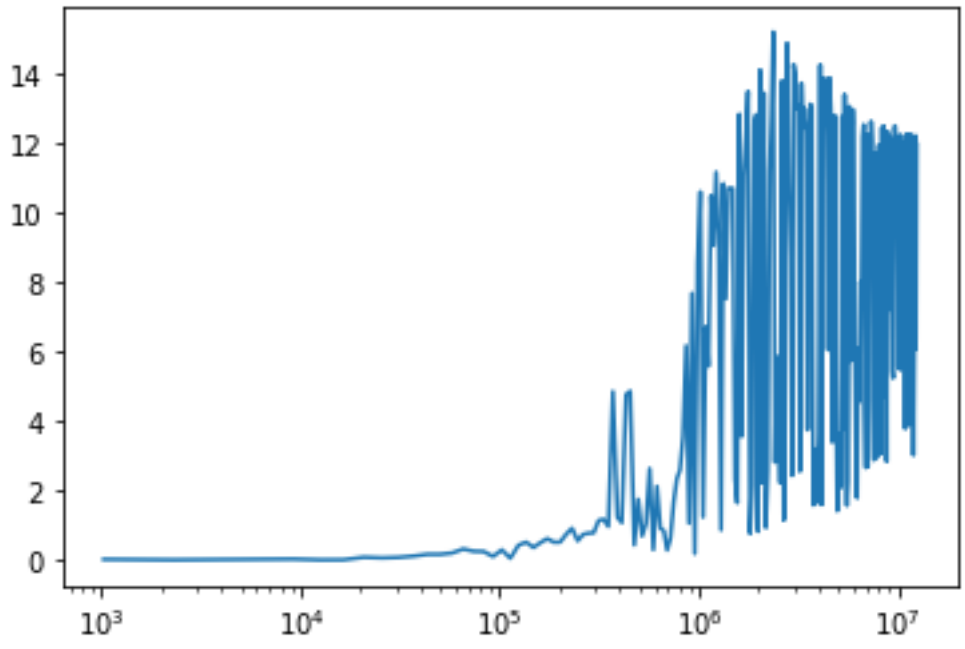

Figure 7: Gain Parallèle/Sequentiel en fonction de la taille des vecteurs pour le produit scalaire avec 20 threads

On fait l'expérience de réduire le nombre de threads à 8 avec la commande `omp_set_num_threads(8)`.

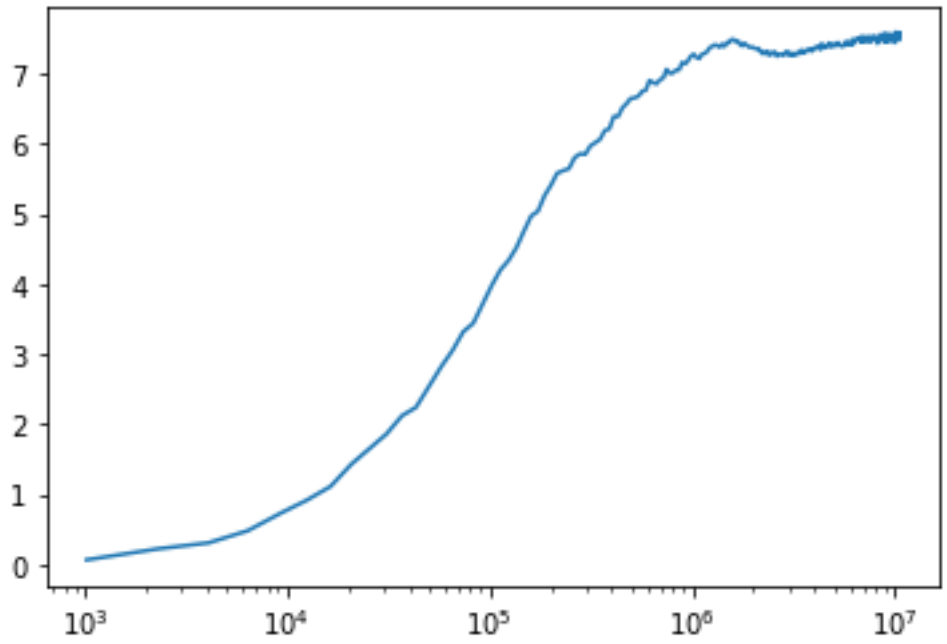

Figure 8: Gain Parallèle/Sequentiel en fonction de la taille des vecteurs pour le produit scalaire avec 8 threads

La courbe d'accélération est cette fois plus lisse. Le gain est aussi beaucoup plus proche du nombre de threads, cela peut s'expliquer par la réduction d'efficacité par cpu attendue avec l'augmentation du nombre de threads.

2.3 Produit scalaire de deux vecteurs avec déséquilibre entre threads

Le code en séquentiel est le suivant :

```
double s = 0;
for (unsigned long int i = 0; i < size; i++) {
    if(i < size/10){
        s += A[i] * B[i];
    }
}
return s;
```

Pour le code OpenMP, nous allons tester les 6 différents modes de la fonction schedule: le mode par défaut, static, dynamic, guided, auto et runtime. Pour le mode static par exemple le code utilisant OpenMP est le suivant avec un chunk size de size/200.

```
double s = 0;
#pragma omp parallel shared(s)
{
    #pragma omp for reduction(+:s) schedule(static , size/200)
    for (unsigned long int i = 0; i < size; i++) {
        if(i < size/10){
            s += A[i] * B[i];
        }
    }
}
return s;
}
return s;
```

Pour chaque mode où cela est possible, nous testerons 2 chunk size différent, le premier correspondant à la taille du sous vecteur sur lequel se fait le produit scalaire, à savoir size/10. Le deuxième chunk size utilisé correspond à cette taille divisée par le nombre de processeur qui est de 20 ce qui correspond donc à size/200. Ce choix est fait pour qu'il soit possible de répartir le produit effectué sur le sous-vecteur en 20 process. Notons que les modes auto et runtime ne permettent pas de fixer le chunk size de par leurs fonctionnement. Pour que la comparaison soit équitable nous calculons le gain par rapport à une seule et même exécution du code séquentiel pour tous les modes. Soulignons par ailleurs l'importance de demander les résultats obtenus pour chaque mode, par exemple avec un print, pour assurer que l'exécution soit bien faite sans quoi les temps d'exécutions ne sont pas valides.

Plusieurs constats :

- On remarque encore une fois un gain de performance en fonction de la taille du vecteur pour tous les modes sauf runtime.
- Les modes runtime et auto sont les moins performants. Ces deux modes font le choix d'un scheduling de manière automatique selon leur procédure.
- Comme on pouvait s'y attendre, diviser le chunk size par 200 permet d'avoir plus de process dans le sous-vecteur dans lequel se passe la multiplication. On a donc ici une performance améliorée lorsqu'on choisi ce chunk size par rapport à size/10.
- Parmi les modes utilisant un chunk size de size/200 le mode static est le plus performant. C'est cohérent avec le fait que le mode static fait un round robin basique qui fera en sorte que les 20 chunks de la partie des vecteurs multipliés, où se fait le calcul, sera réparti entre 20 processeurs. Ceci n'est pas assuré avec les modes dynamic et guided, ces deux peuvent changer la manière dont est fait le round robin, ainsi le processus 1 pourrait avoir à faire deux chunks dans la partie multipliée ce qui fait qu'un autre processeur ne fera pas d'opération dans cette partie et restera donc oisif.

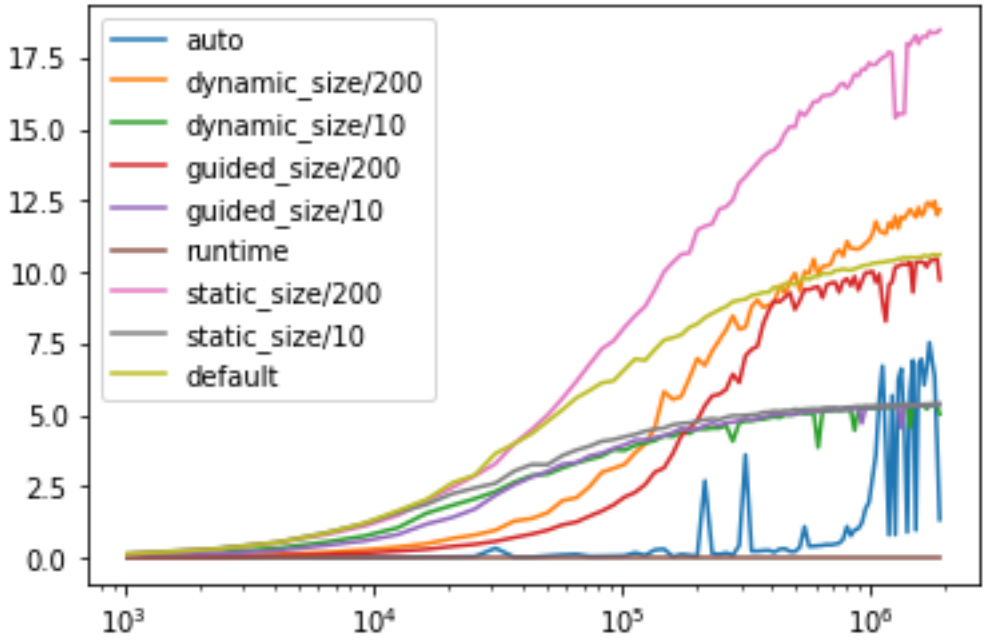

Figure 9: Gain Parallèle/Sequentiel en fonction de la taille des vecteurs pour le produit scalaire partiel, avec des modes et chunk size différents

De plus pour le mode dynamique, le chunk size correspond à la taille minimum.

- La performance du mode défaut est proche de ses deux derniers.
- On a un gain de 5 pour tous les modes pour lesquels on prend une chunk size de size/10 qui est la taille du vecteur multiplié. On n'a donc pas de multiprocessing dans la multiplication. Cependant, le gain peut être lié au fait qu'on parcours les 90/100 restants des vecteurs qui implique à chaque fois de faire une addition sur l'incrément et de tester des conditions. Toutes ces opérations sont accélérées par la parallélisation ce qui explique le gain supérieur à 1.

Pour vérifier le dernier point, refaisons donc l'expérience en bouclant seulement sur les 10/100 premières valeurs des vecteurs au lieu d'avoir une condition à l'intérieur de la boucle. Le code séquentiel prend alors le modèle suivant:

```
double s = 0;
for (unsigned long int i = 0; i < size/10; i++) {
    s += A[i] * B[i];
}
return s;
```

Et on change le code d'openMP de la même manière:

```
double s = 0;

#pragma omp parallel shared(s)
{
    #pragma omp for reduction(+:s) schedule(static , size/200)
    for (unsigned long int i = 0; i < size/10; i++) {
        s += A[i] * B[i];

    }
}
return s;
}
```

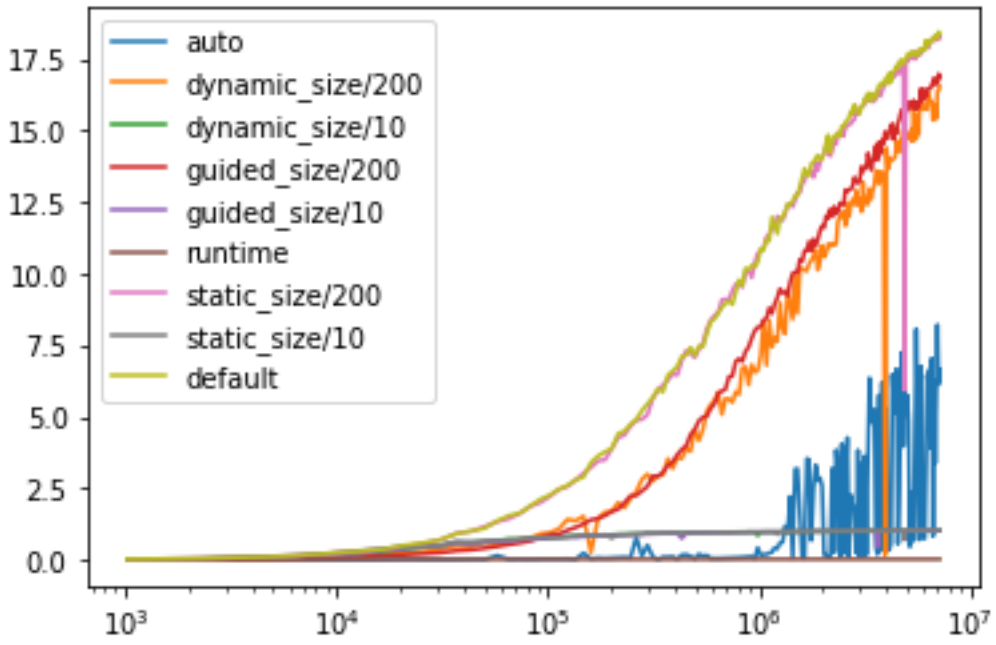

Figure 10: Gain Parallèle/Sequentiel en fonction de la taille des vecteurs pour le produit scalaire partiel, avec des modes et chunk size différents

On remarque que :

- Cette fois tous les modes qui on un chunk size de size/10 on un gain qui tend vers 1, c'est cohérent avec le fait qu'un seul process est reçoit le chunk où les calculs sont fait.
- Les modes guided et dynamic avec un chunk size de size/200 sont plus

performants mais restent inférieurs à static. Ils reconaissent mieux que la partie des vecteurs où ne se font pas les calculs n'a pas besoin de ressources et donc la répartition se fait mieux dans la partie où sont faits les calculs. D'où le gain de performance par rapport au cas précédent.

- La performance du mode défaut est similaire au static, ils ont la meilleur performance parmis les modes dans ce cas.
- La performance des modes auto et runtime ne change pas.

2.4 Calcul de la dérivée sur une image

Le code en séquentiel est le suivant :

```
for (auto i = 1; i < height-1; i++) {
    for (auto j = 1; j < width-1; j++) {
        if ((i==0)||(i==height-1)||(j==0)||(j==width-1)) {Resultat[i][j]=0;}
        else {
            Resultat[i][j]  = std::abs(
            Source[i-1][j-1] + Source[i-1][j] + Source[i-1][j+1]
            - (Source[i+1][j-1] + Source[i+1][j] + Source[i+1][j+1])
            );
            Resultat[i][j] += std::abs(
            Source[i-1][j-1] + Source[i][j-1] + Source[i+1][j-1] -
            (Source[i-1][j+1] + Source[i][j+1] + Source[i+1][j+1])
            );
        }
    }
}
```

Et le code parallèle:

```
#pragma omp parallel shared(Resultat)
{
#pragma omp for schedule(static)
for (auto i = 1; i < height-1; i++) {
    for (auto j = 1; j < width-1; j++) {
        if ((i==0)||(i==height-1)||(j==0)||(j==width-1)) {Resultat[i][j]=0;}
        else {
            Resultat[i][j]  = std::abs(
            Source[i-1][j-1] + Source[i-1][j] + Source[i-1][j+1] -
            (Source[i+1][j-1] + Source[i+1][j] + Source[i+1][j+1])
            );
            Resultat[i][j] += std::abs(
            Source[i-1][j-1] + Source[i][j-1] + Source[i+1][j-1] -
            (Source[i-1][j+1] + Source[i][j+1] + Source[i+1][j+1]));
        }
    }
```

```
    }
}
    return(Resultat[3]);
```

Quelques points sur le code :

- On déclare la variable Resultat comme shared car elle est utilisée par tous les process.
- La condition intérieure à la boucle assure qu'on ne calcule pas de dérivée sur les bords.
- On calcul la dérivée les les lignes puis on ajoute la dérivée sur les colonnes.

On obtient un gain parallèle/séquentiel de 13.7 sur l'image drone et 15.3 sur l'image huge drone.

2.5 Création d'une sous liste

On propose un code séquentiel puis parallèle pour parcourir une liste et stocker ses éléments pairs dans une sous-liste et renvoyer leur nombre.

Code séquentiel:

```
int s = 0;
for (unsigned long int i = 0; i < size; i++) {
    if(A[i]%2 == 0){
        S[s] = A[i];
        s ++;
    }
}
return s;
```

Code parallèle:

```
int s = 0;
#pragma omp parallel for reduction(+:s)
for (unsigned long int i = 0; i < size; i++) {
    if(A[i]%2 == 0){
        S[s] = A[i];
        s ++;
    }
}
return s;
```

Le code suit finalement le même format que pour le produit scalaire avec une réduction. La courbe de gain ci-dessous montre l'intérêt du calcul parallèle dans ce cas également.

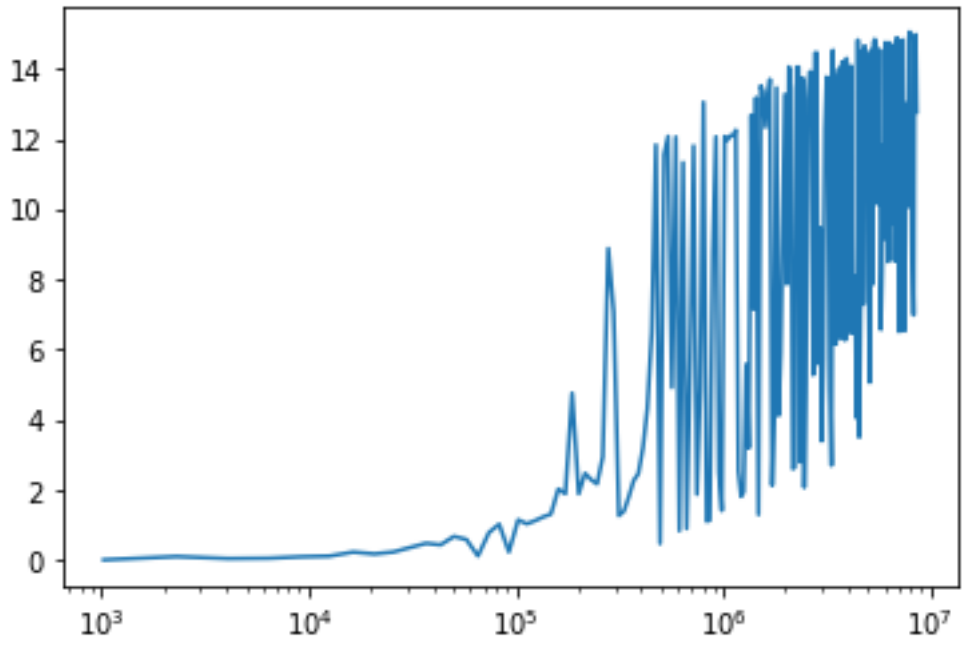

Figure 11: Gain Parallèle/Sequentiel en fonction de la taille des vecteurs pour la création d'une sous-liste des éléments paires.

3 TP3 - Cuda

3.1 Produit scalaire de deux vecteurs

Voici la section de code qui utilise Cuda pour faire le produit scalaire :

```
__global__ void gpu(int n, float *x, float *y, float *s)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) atomicAdd(s, x[i] * y[i]);
}
```

Nous comparons le temps pour effectuer un produit scalaire avec un programme parallèle sur CPU utilisant OpenMP sur 20 threads :

```
void cpu(int n, float *x, float *y, float *s)
{
  #pragma omp parallel for num_threads(20)
  for (int i=0; i<n; i++)
  {
      *s += x[i] * y[i];
  }
```

}

Nous avons ensuite tracé pour le rapport entre le temps nécessaire pour le CPU et le temps nécessaire pour le GPU pour différentes tailles de vecteurs et différents nombres de threads par bloc (32, 512 et 2048).

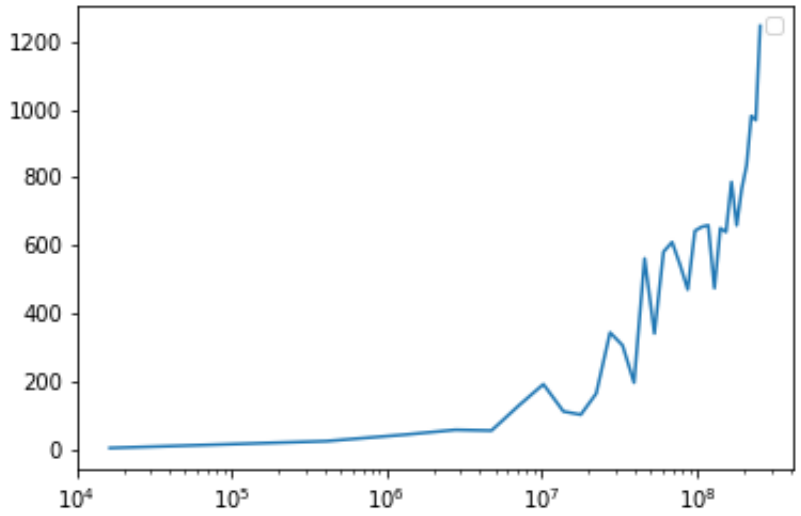

Figure 12: Gain CPU/GPU pour un produit scalaire avec 32 threads par bloc

Plusieurs constats :

- Pour un nombre de threads par bloc fixe, le gain augmente avec la taille des vecteurs ce qui semble logique. Pour des tailles de vecteurs petites, l'overhead (temps nécessaire pour la création des threads) compense le gain de temps sur le calcul.
- Le gain est d'autant plus élevé qu'il y a de threads par bloc, ce qui est cohérent. On arrive même à une accélration de l'ordre de plusieurs dizaines de milliers pour de grands vecteurs avec 2048 threads par bloc.

3.2 Sobel

Dans cette partie, nous allons appliquer un filtre de Sobel (détection de contours) sur différentes images. Cette opération peut bien-sûr être parallélisée. C'est pourquoi nous l'effectuerons sur CPU, sur GPU et sur GPU avec mémoire partagée.

Voici les 3 images avant traitement, la photographie existe en version petite (345 ko) et en version grande (10,7 Mo) :

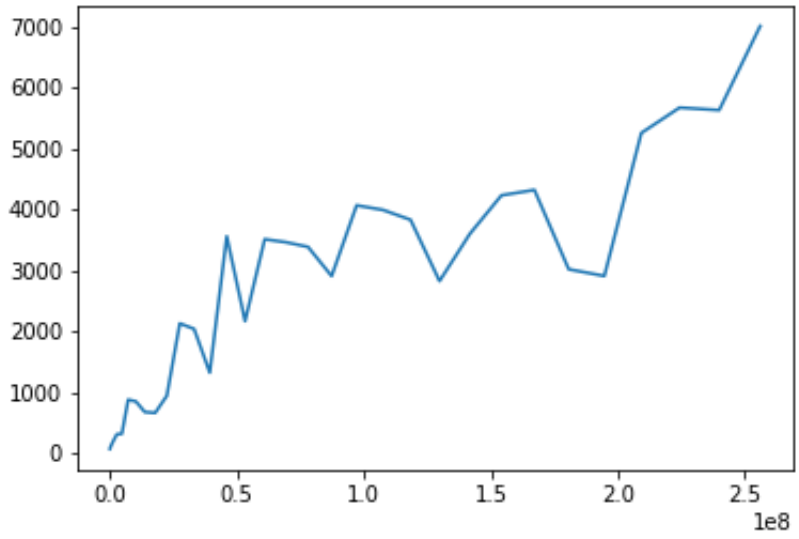

Figure 13: Gain CPU/GPU pour un produit scalaire avec 512 threads par bloc

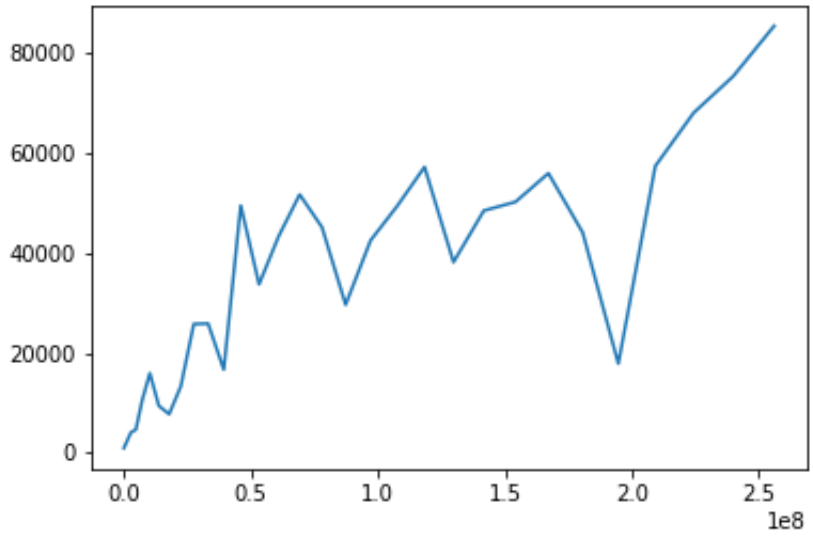

Figure 14: Gain CPU/GPU pour un produit scalaire avec 2048 threads par bloc

Figure 15: Image simple d'un rectangle noir sur fond blanc

Figure 16: Photographie avant traitement

Pour cela, le code C++ était déjà disponible et nous avons écrit ce fichier *.sh* :

```
#!/bin/bash

#PBS -S /bin/bash
#PBS -N tp3_sobel
#PBS -e errorJob.txt
#PBS -j oe
#PBS -l walltime=0:10:00
#PBS -l select=1:ncpus=12:ngpus=1
#PBS -q gpuq
#PBS -m abe -M lucas.petit@student.ecp.fr
#PBS -P progpar

# Go to the current directory
cd $PBS_O_WORKDIR

# Load the same modules as for compilation
```

```
module load gcc/7.3.0
module load cuda/10.2

make clean
make -j

# Run code
echo "Info sur la carte"
nvidia-smi -L


echo "Lancement du bench"
./exe/sobel.exe images/Drone.pgm 10

/gpfs/opt/bin/fusion-whereami
date
time sleep 2
```

De plus, on définit dans le fichier *parameters.cuh* la taille des blocs utilisés pour la parallélisation avec Cuda :

```
#define BLOCKDIM_X 8
#define BLOCKDIM_Y 8
```

Les images renvoyées sont les suivantes :

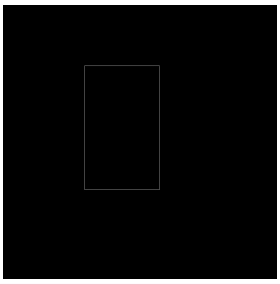

Figure 17: Détection des contours d'un rectangle

Les images qui résultent sont les mêmes sur CPU ou sur GPU (avec la mémoire partagée ou non) et c'est normal car la parallélisation ne change pas le résultat. Intéressons-nous plutôt aux temps d'exécution (en millisecondes) :

	CPU	GPU	GPU (mémoire partagée)
Rectangle	91.46	0.19	0.31
Photo petite	7.47	0.042	0.034
Photo grande	231.75	0.45	0.77

On constate plusieurs choses :

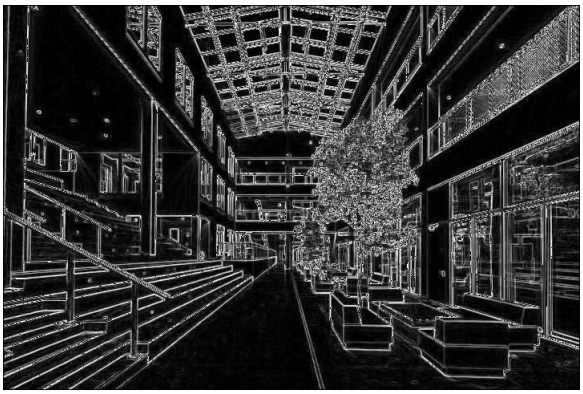

Figure 18: Détection des contours dans une photographie

- Le GPU a encore une fois des performances bien meilleures en parallélisation que le CPU (facteur 1000 en termes de rapidité).
- La mémoire partagée pour le GPU n'est pas nécessairement avantageuse en termes de temps de calcul.

3.3 Transposition d'une image

On s'intéresse aux mêmes images mais on veut cette fois-ci les transposer. Pour ce faire, nous avons écrit le code séquentiel suivant :

```
for (auto i = 1; i < height-1; i++) {
    for (auto j = 1; j < width-1; j++) {
        Resultat[j][i] = Source[i][j];
    }
}
```

Le code utilisant CUDA sans utiliser de mémoire partagée est le suivant :

```
int x = blockIdx.x * TILE_DIM + threadIdx.x;
int y = blockIdx.y * TILE_DIM + threadIdx.y;

for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS) {
    odata[x*height + (y+j)] = idata[(y+j)*width + x];
}
```

Sans oublier de définir certaines variables au début du fichier :

```
#define TILE_DIM 32
#define BLOCK_ROWS 8
```

On peut également faire une transposition en faisant en sorte d'utiliser une mémoire partagée :

```
__shared__ float tile[TILE_DIM][TILE_DIM];

int x = blockIdx.x * TILE_DIM + threadIdx.x;
int y = blockIdx.y * TILE_DIM + threadIdx.y;


for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS) {
    tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];
}

__syncthreads();

x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
y = blockIdx.x * TILE_DIM + threadIdx.y;

for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS) {
    odata[(y+j)*height + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Les images renvoyées sont les suivantes :

Figure 19: Transposée d'un rectangle

Les images qui résultent sont les mêmes sur CPU ou sur GPU (avec la mémoire partagée ou non) et c'est normal car la parallélisation ne change pas le résultat. Intéressons-nous plutôt aux temps d'exécution (en millisecondes) :

	CPU	GPU	GPU (mémoire partagée)
Rectangle	65.57	0.48	0.082
Photo petite	1.1	0.054	0.015
Photo grande	47.19	1.42	0.19

On constate :

- Le GPU est bien plus efficace que le CPU comme pour le filtre de Sobel
- Le GPU avec mémoire partagée est plus efficace que le GPU sans mémoire partagée pour les trois images, ce qui n'était pas le cas pour le filtre de Sobel

Figure 20: Transposée d'une photographie