UNIVERSITÉ
LAVAL

TP 4 - OpenACC & OpenCL

Réalisé par **l'équipe 1** constitué de :

Rémi Huguet - 536 795 651
Gabriel Phillipon - 536 795 815

Dans le cadre du cours :
GIF-7104 – Programmation parallèle et distribuée

Travail présenté à :
Marc Parizeau

Remis le :
7 Avril 2021

Objectif

L'objectif de ce TP était d'inverser une matrice (comme dans le TP précédent) mais en utilisant cette fois ci OpenCL et OpenACC, et en utilisant les capacités de parallélisation d'un GPGPU. Le but est également de mesurer le Speedup entre l'implémentation séquentielle et l'implémentation massivement multithreads de ce programme.

Solution

Dans un premier temps, pour utiliser OpenACC sans avoir d'erreur ni de gêne nous avons dû reprendre la solution séquentielle et l'implémenter sans la classe Matrix fourni. Pour la suite du TP, nous avons décidé de n'utiliser la classe Matrix que pour les initialisations, pour ensuite travailler avec des tableaux dynamiques.

Solution openACC

Notre première solution visait à prendre l'algorithme séquentiel et y ajouter les directives openACC au niveau de chaque boucle. Cependant nous nous sommes vite rendu compte que les boucles possédaient des dépendances ce qui empêchait la parallélisation (En effet, les informations fournient par `pgc++` avec l'option `-Minfo=all,accel,intensity,ccff`, nous indiquait de faible optimisations ; certaines intensités de boucle étaient inférieur à 1).

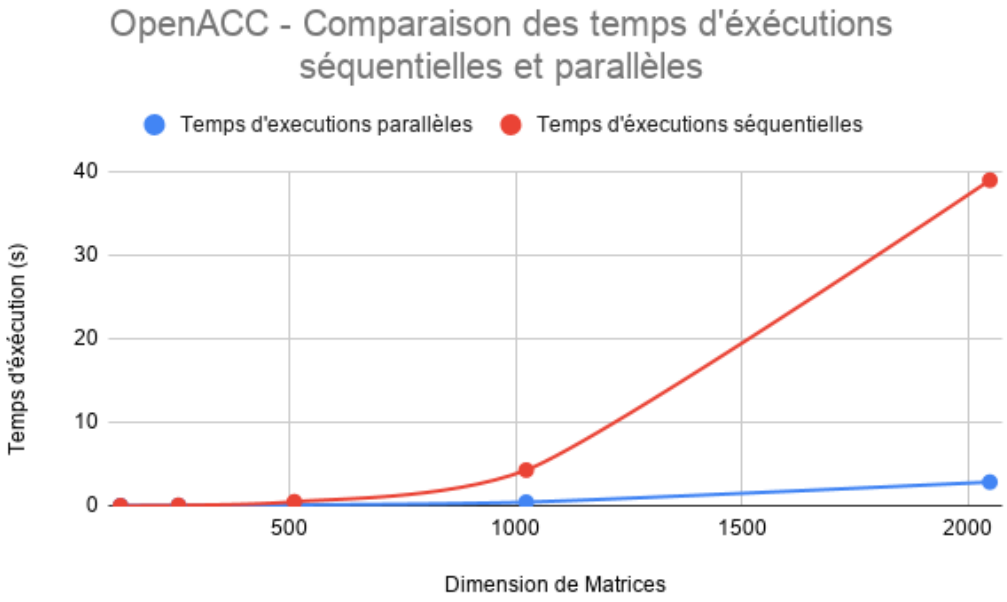
Nous avons décidé de réaliser un algorithme certes plus gourmand en calcul (plus d'opération), mais maximisant l'indépendance des boucles et des instructions. Tous comme avec le programme séquentiel fourni, la boucle principale (qui itère sur chaque rangé) ne peut pas être parallélisé. Cependant, le corps de la boucle peut entièrement être placé dans un kernel (`#pragma acc kernels`). Puis, de cette façon nous avons pu placer des directives `#pragma acc for independent` sur chaque boucle annexe.

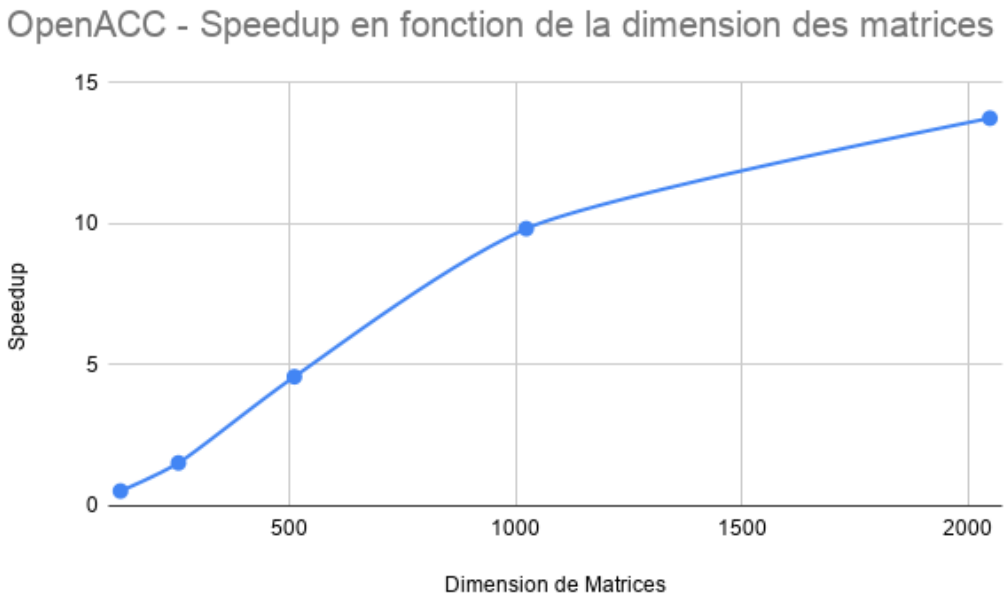
```
Result: Inversed Matrix : result
for each matrix row do
    scale = 1 / matrix[row][row]
    for each element of current row do
        matrix[current row][element] *= scale result[current row][element] *= scale
    end
    for each matrix row do
        if row != current row then
            currentScale = matrix[row][current row] for each element of row do
                result[row][element] -= currentScale * result[row][element]
                matrix[row][element] -= currentScale * matrix[row][element]
            end
        end
    end
end
```

Algorithm 1: Pseudocode de l'algorithme d'inversion de matrice

Afin d'obtenir de meilleur résultats nous avons décidé de laisser openACC gérer automatiquement la mémoire en ajoutant l'option de compilation `managed` dans le flag `-ta=tesla:managed`

Résultats OpenACC





Analyse OpenACC

Avec OpenACC nous avons obtenue de bon speedup. Ceux-ci augmentent avec la taille des matrices. Tel qu'attendu, plus la dimension de la matrice est grande, plus le speedup est important et donc plus la rentabilité d'utilisation d'un GPGPU est rempli.

Solution OpenCL

Pour openCL nous avons décidé de transcrire notre fonction d'inversion openACC en kernel openCL. Tout comme avec OpenACC la boucle principale ne peut pas être parallélisé. Nous l'intégrons donc sur le Host (voir ci-dessous) et nous passons l'index de chaque rangé (`i`) comme argument pour la fonction du kernel.

```
/// Iteration for each row that can't be parallelized
for (int i = 0; i < matrixDimension; i++) {
    status |= clSetKernelArg(kernel, 3, sizeof(cl_int), &i);
    status = clEnqueueNDRangeKernel(cmdQueue,kernel,1,nullptr,globalWorkSize,
   nullptr, 0, nullptr, &event);
    ...
}
```

Listing 1 – Boucle principale sur le host

L'utilisation des `Thread id (idx)` nous permet d'éliminer deux boucles annexes (dans le kernel). En effet, nous nous servons de ces IDs en remplacement des indices de boucles. Cette optimisation est possible car nous créons un nombre de threads égal à la dimension de la matrice.

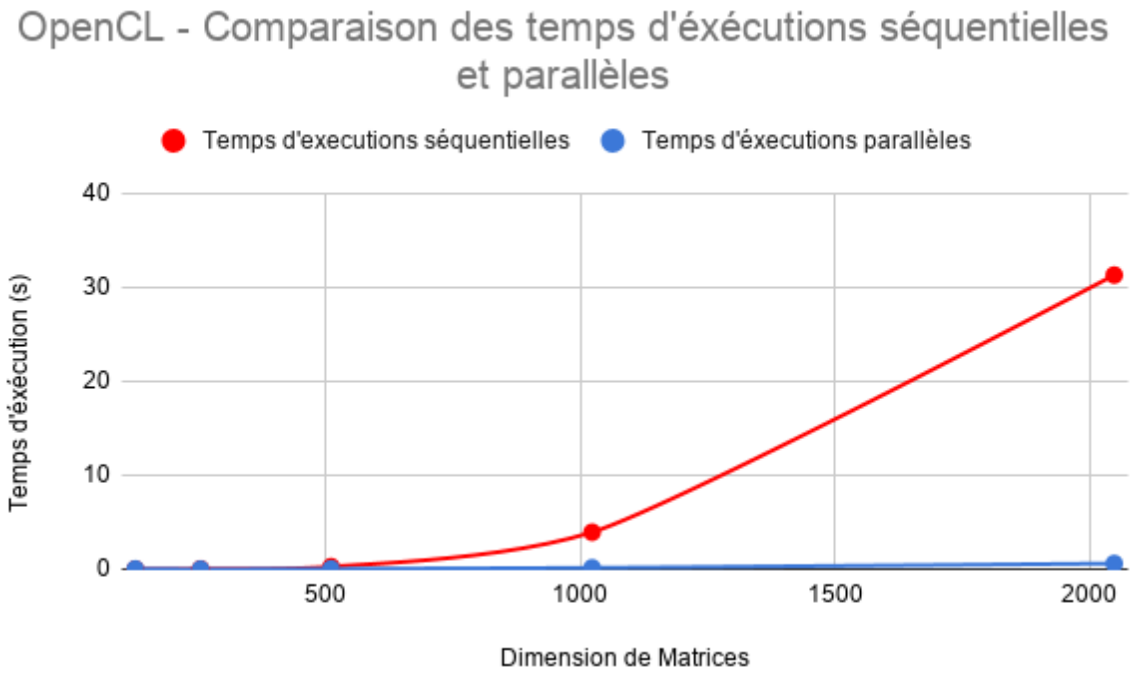
Result: Inversed Matrix : result
scale = 1 / matrix[row*size + row]
matrix[i * size + id_thread] *= scale
result[i * size + id_thread] *= scale
if *thread_id != i* **then**
  currentScale = matrix[thread_id*size + i]
  for *each element of row* **do**
    result[thread_id * size + element] -= currentScale * result[i * size + element]
    matrix[thread_id * size + element] -= currentScale * matrix[i * size+ element]
  end
end

Algorithm 2: Pseudocode du kernel

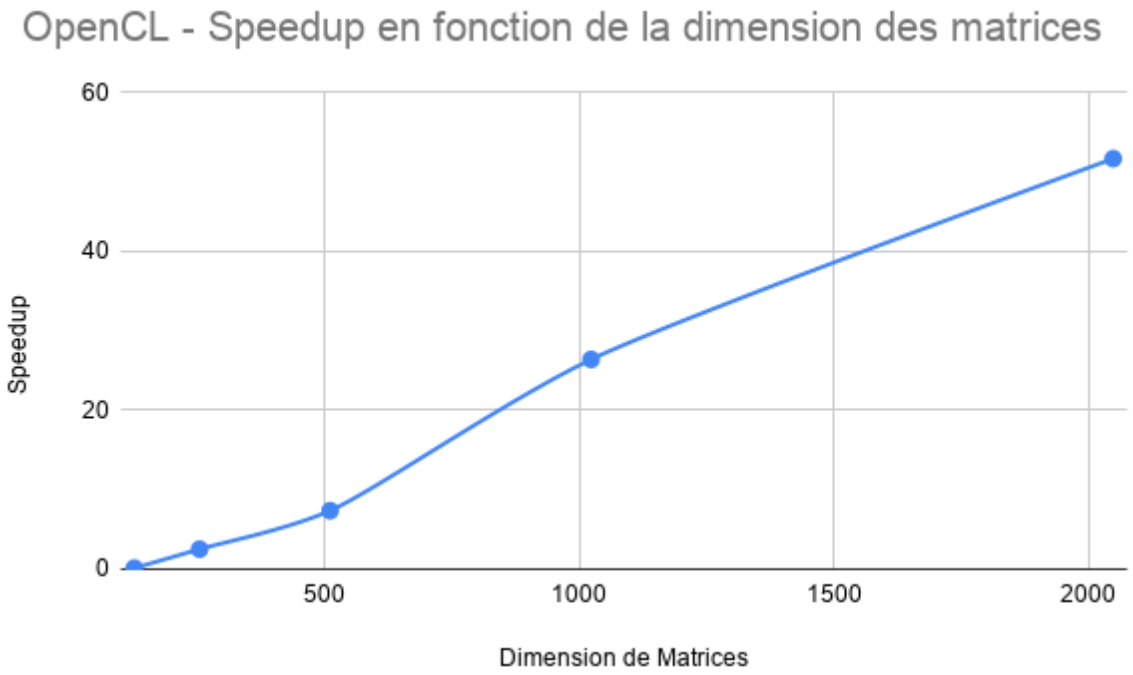
Pour minimiser au maximum les échanges et copie de données entre le `host` et le `device`, on permet au kernel d'écrire directement dans la matrice résultat (avec le `flag` `CL_MEM_USE_HOST_PTR`). Ainsi, on ne recopie pas les matrices sur le `device`.

Comme nous écrivons et lisons directement dans les matrices sur le `host` nous avons décidé de ne pas utiliser de workgroup.

Résultats OpenCL



2

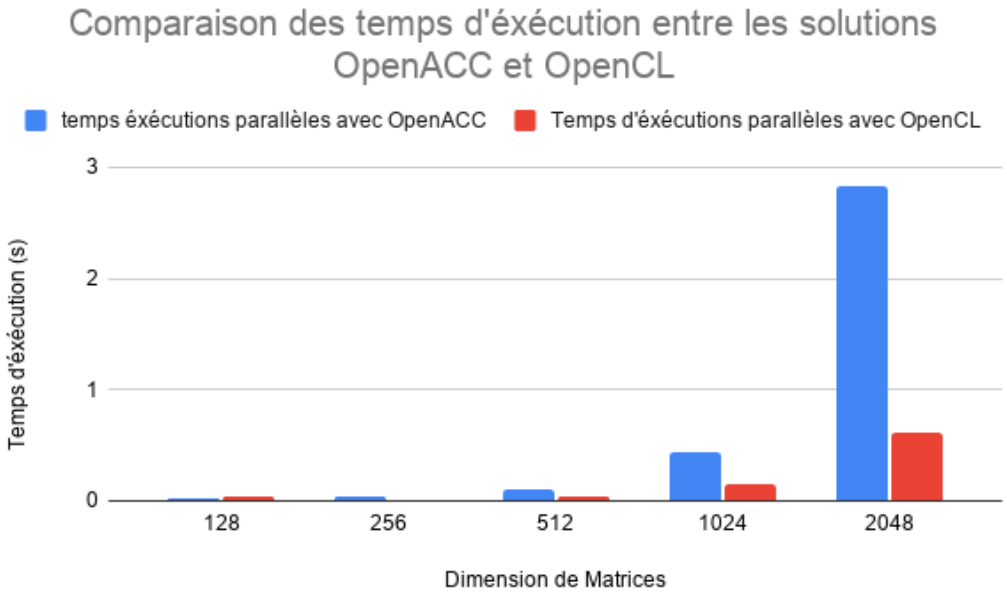


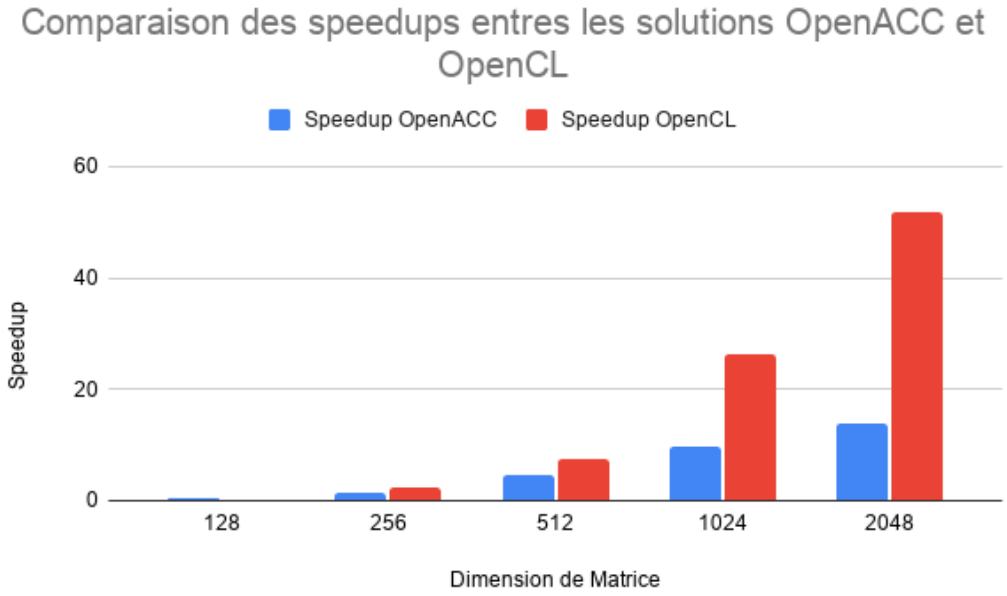
Analyse OpenCL

Avec OpenCL nous obtenons des speedup allant jusqu'à 50 ce qui est relativement bon. Ceux-ci augmente avec la taille des matrices. Comme prévu, plus la dimension de la matrice est grande,

plus le speedup est important et donc plus la rentabilité d'utilisation d'un GPGPU est rempli. En effet, plus la matrice est grande, plus nous créons de thread et plus la parallélisation devient efficace.

OpenACC vs OpenCL





Tous d'abord, on peut remarquer que les speedups d'OpenCL sont bien plus grands que ceux d'OpenACC. En effet, OpenCL nous permet assez de liberté et de précision par rapport aux

optimisations, et permet donc de dégager plus de performance. On remarque ainsi que nous pouvons obtenir des speedup allant jusqu'à 50 avec OpenCL contre 15 pour OpenACC.

En gérant nous même la mémoire et les transferts de données avec la solution OpenACC nous aurions peut être pu nous rapprocher des performances de la solution OpenCL.

Spécification Matériel

— OS : Ubuntu 20.10

— Processeur : Intel® Core™ i7-6700HQ CPU @ 2.60GHz × 8

— GPU : GTX-980M. Driver version : 460.39, CUDA version : 11.2

— RAM : 8 Go

Optimisations

Pour tirer un maximum partie de la machine sur lequel le programme a été exécuté et afin d'optimiser les temps d'exécution, la compilation est faite en `"-O3"` pour OpenCL et en `-O3 -fast -ta=tesla:managed` pour OpenACC.

Pour représenter nos matrices nous nous sommes séparé de la classe `Matrix` fournie et nous avons utilisé des tableaux 2D pour la solution OpenACC et des tableaux 1D pour OpenCL, tous deux contiguë en mémoire.

Piste d'amélioration

En ce qui concerne OpenACC la gestion de la mémoire et des données auraient peut-être été un plus quoique sûrement négligeable. Pour OpenCL, trouver une façon de gérer parallèlement la boucle principale pourrait être intéressant, à la condition que les temps de synchronisation ne rendent pas cette optimisation désuète.

Annexe

Données brutes

Consultable sur un Google Sheet