Optimisation parallèle de l'algorithme de Floyd-Warshall

Romain DULERY

10 avril 2018

Table des matières

1 Présentation générale du projet, objectifs et travail réalisé

L'objectif du projet est d'optimiser l'algorithme de Floyd-Warshall à l'aide d'une implémentation parallèle, en OpenMP et OpenCL. L'accent sera mis sur l'optimisation spatiale de la mémoire, étant donné que l'algorithme est relativement simple et classique.

Les technologies utilisées sont C++11, OpenMP, Python3 pour les scripts, et Valgrind pour les analyses d'optimisation.

J'ai réalisé une plateforme de test de performance, pour générer automatiquement les données d'entrée, et les performances respectives des différents algorithmes.

Dans la partie suivante, sont présentées les méthodes de cette plateforme, le format de données utilisé, les scripts et un exemple d'exécution.

2 Présentation de la plateforme de développement

Elle se présente de la manière suivante :

```
romain@romain-ThinkPad-T420:~/Documents/sauvegarde_ubuntu/Proyecto_Integrador$ ls
Cachegrind     Graphs         Makefile             Scripts_Python  Unit_Tests
Calcul_Times   Ground_Truth   Output_Files         Sizes
Documentation  Input_Files    Performance_Analysis  src
```

FIGURE 1 – Arborescence utilisée

2.1 Données d'entrée

Celles-ci se trouvent dans le dossier Input_Files.

2.1.1 Format de données

Le format choisi est le suivant :

La première ligne indique le nombre de sommets et d'arêtes. Ensuite, chaque ligne représente une arête, avec le numéro du sommet initial, le numéro du sommet d'arrivée, et le poids associé à l'arête.

Les numéros de sommet commencent à 0.

Par exemple, ce graphe :

Est représenté de la manière suivante :

2.1.2 Génération de données d'entrée

Dans le dossier Scripts_Python se trouvent les scripts **Input_Files_Generator.py** et **Special_File_Generator.py**, qui créent des données d'entrée.

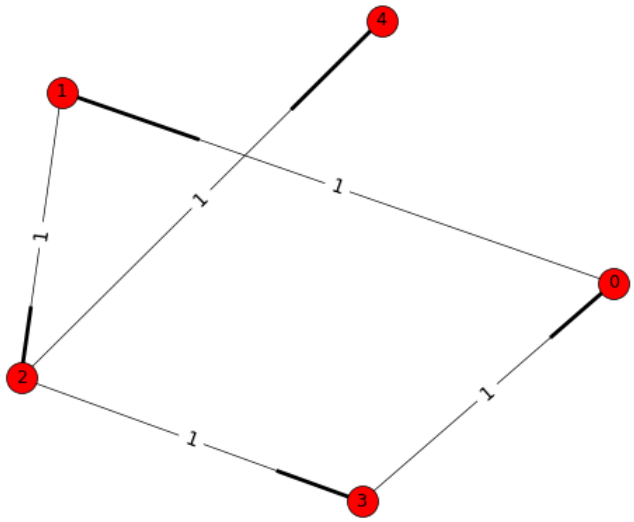


FIGURE 2 – Graphe d'exemple

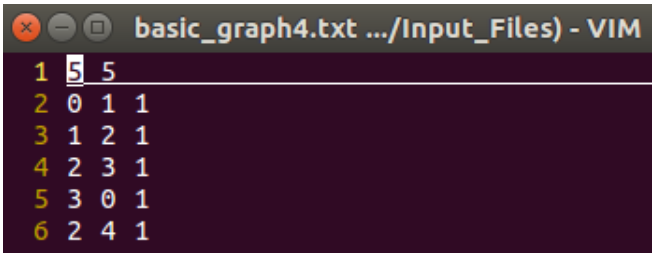


FIGURE 3 – Données d'entrée correspondantes

Le premier génère 30 fichiers d'entrée. Les arguments sont le nom des fichiers générées, le nombre de sommets (aléatoire entre V/2 et V où V est l'argument), et le nombre d'arêtes en troisième argument, bien que celui-ci n'influe pas la complexité de l'algorithme, il sera quand même utile pour certains algorithmes. Le poids de chaque arête est aléatoire entre 1 et 100.

Le second script est le même que le premier, pour un seul fichier, avec un nombre exact de sommets et d'arêtes. Il sert à générer des graphes de référence.

2.1.3 Données utilisées

Dans tout le rapport, V représente le nombre de sommets, et E le nombre d'arêtes.

J'ai utilisé des graphes ordonnés en 7 catégories :

— Basiques (basic_graphX)

  Ils servent à tester le fonctionnement basique des algorithmes. Le graphe de la figure 2 est le basic_graph4.

— Simples (simple_graphX)

V entre 5 et 20 E entre 10 et 30

— Moyens (medium_graphX)

V entre 100 et 1000 E entre 200 et 5000

— Complexes (complex_graphX)

V entre 2000 et 5000 E entre 10000 et 50000

— Enormes (huge_graphX)

V entre 5000 et 10000 E entre 50000 et 150000

— Références (reference_graphX)

Ce sont 6 graphes avec V = 400, 800, 1200, 1600, 2000 et 3200

Cela reste des graphes relativement petits pour pouvoir réaliser une comparaison pertinente des performances, notamment pour conjecturer le comportement lorsque V tend vers l'infini.

2.2 Données de sortie

Les données de sortie sont simplement la matrice de distances entre chaque sommet. Elles peuvent être sauvegardées dans des fichiers via une option du main, présentée dans la sous-partie suivante.

2.3 Makefile et main

Le main se génère via le **make**. Il peut être utilisé comme **make debug** ou **make sinopti**, pour générer le main en version débug, ou en version sans optimisation O3 du compilateur.

Le main demande un fichier d'entrée, et exécute chaque algorithme, en affichant les temps d'exécution de chaque algorithme sur la sortie standard.

```
romain@romain-ThinkPad-T420:~/Documents/sauvegarde_ubuntu/Proyecto_Integrador$ ./main
Choose the input file : basic_graphX, simple_graphX, medium_graphX, complex_graphX, very_comple
with X between 0 and 29 : complex_graph4
-------------------------------------------------
File : complex_graph4.txt
V = 1965;
FWSeq :: Floyd_Warshall Sequential :
        Time elapsed :: 5.99586s ::  using 1 threads;

FWSqM :: Floyd_Warshall Sequential with Memoization :
        Time elapsed :: 6.06106s ::  using 1 threads;

FWSMS :: Floyd_Warshall Sequential with Memoization and Skipping infinities :
        Time elapsed :: 1.22181s ::  using 1 threads;

FWP1L :: Floyd_Warshall Parallel 1D lines :
        Time elapsed :: 1.40006s ::  using 4 threads;

FWSTB :: Floyd_Warshall Sequential Tiled Implemented Cache-Aware, with adapted memory layout :
        Time elapsed :: 2.68123s ::  using 1 threads;

FWPTB :: Floyd_Warshall Parallel Tiled Implemented Cache-Aware, with adapted memory layout :
        Time elapsed :: 1.0263s ::  using 4 threads;

FWS1D :: Floyd_Warshall Sequential with 1d matrix :
        Time elapsed :: 1.24552s ::  using 4 threads;

FWP1D :: Floyd_Warshall Parallel with 1d matrix :
        Time elapsed :: 0.589512s ::  using 4 threads;

Do you want to do another graph ? (y/N) :
```

FIGURE 4 – Une exécution du main

Il possède également plusieurs options pour modifier le comportement, comme par exemple :

- -a : pour choisir un seul algorithme
- -d : pour sauvegarder les données de sortie de chaque algorithme dans des fichiers, dans le dossier Output_Files
- -t : pour sauvegarder les temps d'exécution pour chaque entrée dans des fichiers, dans le dossier Calcul_Times
- -c : pour exécuter les algorithmes sur tous les fichiers d'entrée d'une catégorie (moyen, complexe..)
- -g : pour choisir le graphe d'entrée (sans que le main le demande, afin d'automatiser les résultats)
- -h : pour afficher l'aide
- -s : pour choisir le pas de certains algorithmes, présentés dans la partie suivante.

3 Présentation de chaque algorithme

Chaque algorithme est représenté par un code d'entrée et un code de sortie.

Le code d'entrée sert pour l'option -a du main, et le code de sortie sert pour l'option -t du main.

3.1 Floyd-Warshall séquentiel

Code d'entrée : FW_SEQ
Code de sortie : FWSeq

```cpp
// Basic Floyd_Warshall algorithm
std::vector<std::vector<unsigned short>> floyd_warshall_seq(std::vector<std::vector<unsigned short>> graph)
{
    unsigned short V = graph[0].size();
    for (int k = 0; k < V; k++)
    {
        for (int i = 0; i < V; i++)
        {
            for (int j = 0; j < V; j++)
            {
                if (graph[i][j] > graph[i][k] + graph[k][j]) {
                    graph[i][j] = graph[i][k] + graph[k][j];
                }
            }
        }
    }
    return graph;
}
```

FIGURE 5 – Algorithme de Floyd-Warshall

C'est l'algorithme basique de Floyd-Warshall.

3.2 Floyd-Warshall séquentiel avec mémoïzation

Code d'entrée : FW_SEQ_MEM
Code de sortie : FWSqM

```cpp
// Basic Floyd_Warshall algorithm with memoization
std::vector<std::vector<unsigned short>> floyd_warshall_seq_mem(std::vector<std::vector<unsigned short>> graph)
{
    int i,j,k;
    unsigned short V = graph[0].size();
    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            unsigned short distIK = graph[i][k];
            for (j = 0; j < V; j++)
            {
                if (graph[i][j] > distIK + graph[k][j]) {
                    graph[i][j] = distIK + graph[k][j];
                }
            }
        }
    }
    return graph;
}
```

FIGURE 6 – Algorithme de Floyd-Warshall avec mémoïzation

C'est l'algorithme basique, avec mémorisation de la distance IK, qui est recalculée à chaque pour rien dans la dernière boucle sur j dans la version initiale.

Cela permet également de réduire le nombre de défauts de cache.

3.3 Floyd-Warshall séquentiel avec mémoïzation et sauts d'infinis

Code d'entrée : FW_SEQ_MEM_SKIP
Code de sortie : FWSMS

```cpp
// Basic Floyd_Warshall algorithm with memoization and skip infinities
std::vector<std::vector<unsigned short>> floyd_warshall_seq_mem_skip(std::vector<std::vector<unsigned short>> graph)
{
    int i,j,k;
    unsigned short V = graph[0].size();
    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            unsigned short distIK = graph[i][k];
            if (distIK == infinity)
                continue;
            for (j = 0; j < V; j++)
            {
                if (graph[i][j] > distIK + graph[k][j]) {
                    graph[i][j] = distIK + graph[k][j];
                }
            }
        }
    }
    return graph;
}
```

FIGURE 7 – Algorithme de Floyd-Warshall avec mémoïzation et sauts d'infinis

C'est l'algorithme basique, avec mémoïzation de IK, et sauts d'infinis. Les sauts d'infinis font alors dépendre le temps de calcul du nombre d'arêtes, alors que normalement seul le nombre de sommets interviennent dans le calcul de la complexité.

Chaque algorithme qui suit utilisera les deux techniques de mémoïzation et sauts d'infinis.

3.4 Floyd-Warshall parallélizé sur les lignes

Code d'entrée : FW_PAR1_L
Code de sortie : FWP1L

L'algorithme se parallélize facilement car ne possède pas de dépendance temporelle et spatiale contraignante.
On peut alors simplement le paralléliser selon les lignes.

Le site Visualisation de l'exécution permet de visualiser les appels mémoires réalisés durant l'exécution de l'algorithme.

```
// Strip-mined 1D lines version of Floyd_Marshall algorithm
std::vector<std::vector<unsigned short>> floyd_warshall_par1_l(std::vector<std::vector<unsigned short>> graph, int step) {
    int i, j, k;
    int V = graph[0].size();
    for (k = 0; k < V; ++k)
    {
        int ii;
#pragma omp parallel for private(i) num_threads(4)
        for (i = 0; i < V; i += step)
        {
            for (j = 0; j < V; j++)
            {
                unsigned short distKJ = graph[k][j];
                if (distKJ == infinity)
                    continue;
                for (ii = i; ii < i + step; ii++)
                {
                    if (ii == V)
                        break;
                    if (graph[ii][j] > graph[ii][k] + distKJ) {
                        graph[ii][j] = graph[ii][k] + distKJ;
                    }
                }
            }
        }
    }
    return graph;
}
```

FIGURE 8 – Algorithme de Floyd-Warshall parallèle sur les lignes

La matrice est décomposée en bloc horizontaux, et chaque thread travaille sur un bloc.

3.5 Floyd-Warshall parallélizé sur les colonnes

Code d'entrée : FW_PAR1_C
Code de sortie : FWP1C

```
// Strip-mined 1D columns version of Floyd_Marshall algorithm
std::vector<std::vector<unsigned short>> floyd_warshall_par1_c(std::vector<std::vector<unsigned short>> graph, int step) {
    int i, j, k;
    int V = graph[0].size();
    for (k = 0; k < V; ++k)
    {
        int jj;
#pragma omp parallel for private(j) num_threads(4)
        for (j = 0; j < V; j += step)
        {
            for (i = 0; i < V; i++)
            {
                unsigned short distIK = graph[i][k];
                if (distIK == infinity)
                    continue;
                for (jj = j; jj < j + step; jj++)
                {
                    if (jj == V)
                        break;
                    if (graph[i][jj] > distIK + graph[k][jj]) {
                        graph[i][jj] = distIK + graph[k][jj];
                    }
                }
            }
        }
    }
    return graph;
}
```

FIGURE 9 – Algorithme de Floyd-Warshall parallèle sur les colonnes

Idem que le précédent, sur les colonnes.

3.6 Floyd-Warshall séquentiel, avec une matrice coupée en blocs carrés

Code d'entrée : FW_SEQ_TILED
Code de sortie : FWSTI (Floyd-Warshall séquentiel "tiled-implemented")

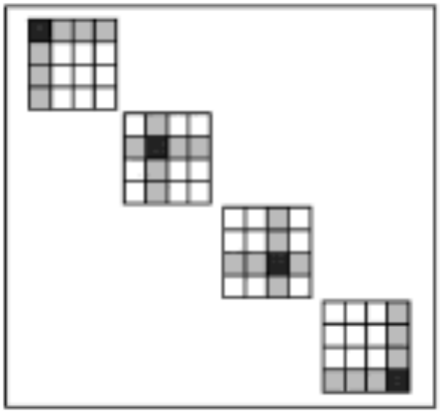
```
// Tiled implementation of Floyd_Marshall algorithm,
// step = bloc width
std::vector<std::vector<unsigned short>> floyd_warshall_seq_tiled(std::vector<std::vector<unsigned short>> graph, int step) {
    int V = graph[0].size();
    int i,j;
    for (int kmin = 0; kmin < V; kmin += step)
    {
        int kmax = MIN(V, kmin + step);
        for (int k = kmin; k < kmax; k++) {
            // Bloc (k,k)
            for (i = kmin; i < kmax; i++) {
                unsigned short distIK = graph[i][k];
                if (distIK == infinity)
                    continue;
                for (j = kmin; j < kmax; j++) {
                    if (graph[i][j] > distIK + graph[k][j]) {
                        graph[i][j] = distIK + graph[k][j];
                    }
                }
            }
        }
        for (int k = kmin; k < kmax; k++) {
            // Line (k,j)
            for (i = kmin; i < kmax; i++) {
                unsigned short distIK = graph[i][k];
                if (distIK == infinity)
                    continue;
                for (j = 0; j < kmin; j++) {
                    if (graph[i][j] > distIK + graph[k][j]) {
                        graph[i][j] = distIK + graph[k][j];
                    }
                }
                for (j = kmax; j < V; j++) {
                    if (graph[i][j] > distIK + graph[k][j]) {
                        graph[i][j] = distIK + graph[k][j];
                    }
                }
            }
            // Column (i,k)
            for (j = kmin; j < kmax; j++) {
                unsigned short distKJ = graph[k][j];
                if (distKJ == infinity)
                    continue;
                for (i = 0; i < kmin; i++) {
                    if (graph[i][j] > graph[i][k] + distKJ) {
                        graph[i][j] = graph[i][k] + distKJ;
```

FIGURE 10 – Algorithme de Floyd-Warshall séquentiel en "tiled-implemented"

C'est un algorithme de transition vers l'algorithme suivant.

L'idée est de couper la matrice en blocs carrés, et de la traiter selon un ordre particulier :

1. Le bloc (k, k)
2. La ligne et la colonne du bloc, qui dépendent seulement du bloc (k,k)
3. Ce qu'il reste



Cet algorithme est fait pour qu'il soit cache-oblivious, de manière à ce qu'un bloc soit de la taille du cache. En effet, pour calculer chaque bloc, cela nécessite de charger seulement 1 bloc en étape 1, 2 bloc en étape 2, et 3 blocs en étape 3.

3.7 Floyd-Warshall parallèle, avec une matrice coupée en bloc carrés

Code d'entrée : FW_PAR_TILED
Code de sortie : FWPTI (Floyd-Warshall parallèle tiled-implemented)

C'est la version parallèle de l'algorithme précédent.

3.8 Floyd-Warshall séquentiel, avec une matrice coupée en blocs carrés, avec une disposition adaptée de la mémoire

Code d'entrée : FW_SEQ_TILED_LAYOUT
Code de sortie : FWSTB (Floyd-Warshall séquentiel tiled-bloc implemented)

```
// Tiled implementation of Floyd_Marshall algorithm, with adapted memory layout in blocs
// blocSize1D = bloc width
std::vector<std::vector<unsigned short>> floyd_warshall_seq_tiled_layout(std::vector<std::vector<unsigned short>> adjacency_matrix, int blocSize1D) {
    int V = adjacency_matrix[0].size();
    int blocsPerWidth = V / blocSize1D;
    std::vector<std::vector<unsigned short>> graph = graphToBlocLayout(adjacency_matrix, blocSize1D, true);
    int i, ii, j, jj, k;

    for (int bloc = 0; bloc < blocsPerWidth; bloc++) {
        int numQuad = bloc * (1 + blocsPerWidth);
        for (k = 0; k < blocSize1D; k++) {
            // Bloc (k,k)
            for (i = 0; i < blocSize1D; i++) {
                int distIK = graph[numQuad][i*blocSize1D + k];
                if (distIK == infinity)
                    continue;
                for (j = 0; j < blocSize1D; j++) {
                    if (graph[numQuad][i*blocSize1D + j] > distIK + graph[numQuad][k*blocSize1D + j])
                    {
                        graph[numQuad][i*blocSize1D + j] = distIK + graph[numQuad][k*blocSize1D + j];
                    }
                }
            }
        }
        int numQuadLeft = bloc * blocsPerWidth;
        for (k = 0; k < blocSize1D; k++) {
            // Line (k,j)
            for (i = 0; i < blocSize1D; i++) {
                int distIK = graph[numQuad][i*blocSize1D + k];
                if (distIK == infinity)
                    continue;
                for (j = 0; j < blocsPerWidth; j++) {
                    if (j == bloc)
                        continue;
                    int numCurrentQuad = numQuadLeft + j;
                    for (jj = 0; jj < blocSize1D; jj++) {
                        if (graph[numCurrentQuad][i*blocSize1D+jj] > distIK + graph[numCurrentQuad][k*blocSize1D+jj])
                        {
                            graph[numCurrentQuad][i*blocSize1D+jj] = distIK + graph[numCurrentQuad][k*blocSize1D+jj];
                        }
                    }
                }
            }
            // Column (i,k)
            for (i = 0; i < blocsPerWidth; i++) {
                if (i == bloc)
                    continue;
```

C'est le même algorithme que précédemment, avec une disposition adaptée de la mémoire en blocs. C'est ce que la ligne 3 de l'algorithme fait.

L'idée est de réduire énormément le nombre de défauts de cache.

3.9 Floyd-Warshall parallèle, avec une matrice coupée en blocs carrés, avec une disposition adaptée de la mémoire

Code d'entrée : FW_PAR_TILED_LAYOUT
Code de sortie : FWPTB (Floyd-Warshall parallel tiled-bloc implemented)

C'est la version parallèle de l'algorithme précédent. Cela devrait être le meilleur algorithme. Les algorithmes précédents ont été réalisés pour introduire cet algorithme.

3.10 Floyd-Warshall séquentiel, avec une matrice en une dimension

Code d'entrée : FW_SEQ_1D
Code de sortie : FWS1D

```cpp
// Tiled parallel implementation of Floyd_Marshall algorithm, with a 1 dimension matrix
std::vector<std::vector<unsigned short>> floyd_warshall_seq_1d(std::vector<std::vector<unsigned short>> adjacency_matrix) {
    int V = adjacency_matrix[0].size();
    std::vector<unsigned short> graph(V*V);
    int i, j, k;
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            graph[i * V + j] = adjacency_matrix[i][j];
        }
    }

    for (k = 0; k < V; k++) {
        // Bloc (k,k)
        for (i = 0; i < V; i++) {
            int distIK = graph[i*V + k];
            if (distIK == infinity)
                continue;
            for (j = 0; j < V; j++) {
                if (graph[i*V + j] > distIK + graph[k*V + j])
                {
                    graph[i*V + j] = distIK + graph[k*V + j];
                }
            }
        }
    }

    std::vector<std::vector<unsigned short>> output_matrix(V, std::vector<unsigned short>(V));
    for (i = 0; i < V*V; i++) {
        output_matrix[i / V][i % V] = graph[i];
    }

    return output_matrix;
}
```

C'est l'algorithme basique, avec une représentation 1D de la matrice.

Il peut se voir comme un cas particulier de l'algorithme précédent, quand il n'y a qu'un seul bloc.

3.11 Floyd-Warshall parallèle, avec une matrice en une dimension

Code d'entrée : FW_PAR_1D
Code de sortie : FWP1D

```
// Tiled parallel implementation of Floyd_Marshall algorithm, with a 1 dimension matrix
std::vector<std::vector<unsigned short>> floyd_warshall_par_1d(std::vector<std::vector<unsigned short>> adjacency_matrix) {
    int V = adjacency_matrix[0].size();
    std::vector<unsigned short> graph(V*V);
    int i, j, k;
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            graph[i * V + j] = adjacency_matrix[i][j];
        }
    }

    for (k = 0; k < V; k++) {
#pragma omp parallel for private(i,j) num_threads(4)
        // Bloc (k,k)
        for (i = 0; i < V; i++) {
            int distIK = graph[i*V + k];
            if (distIK == infinity)
                continue;
            for (j = 0; j < V; j++) {
                if (graph[i*V + j] > distIK + graph[k*V + j])
                {
                    graph[i*V + j] = distIK + graph[k*V + j];
                }
            }
        }
    }

    std::vector<std::vector<unsigned short>> output_matrix(V, std::vector<unsigned short>(V));
    for (i = 0; i < V*V; i++) {
        output_matrix[i / V][i % V] = graph[i];
    }

    return output_matrix;
}
```

FIGURE 11 – Algorithme de Floyd-Warshall parallèle avec une matrice 1D

La version parallèle de l'algorithme précédent.

3.12 Floyd-Warshall parallèle, implémentation GPU

Code d'entrée : FW_GPU
Code de sortie : FWGPU

ALGORITHM OPENCL_PARALLEL_FW(A, N)

```
1    for k = 1 to n do
2        for all elements in matrix A, where 1≤ i, j ≤ n in
         parallel do
3        call FW_KERNEL (A, k)
4        end for
     end for
```

Fig. 4. Pseudo code for OpenCL parallel FW algorithm

KERNEL FW_KERNEL(A, K)

```
1    (i, j) ← getThreadID
2  A[i, j] ← min(A[i,  j], A[i,  k] + A[k,  j])
```

FIGURE 12 – Idée de l'implémentation GPU de Floyd-Warshall

La version OpenCL de l'algorithme précédent.

4 Garantie de l'exactitude des sorties

4.1 Vérité Terrain

J'ai commencé par vérifier les sorties de l'algorithme basique, qui peuvent être supposées correctes pour tout graphe si elles sont correctes sur les graphes basiques, étant donné que le code de l'algorithme est classique et se trouve facilement sur internet.

Le script Python **Verify_Output.py** permet en effet de vérifier que les sorties pour les graphes basiques sont les mêmes que celles de la vérité terrain du dossier Ground_Truth.

Pour vérifier les sorties basiques également, le script **Graph_Generator.py** permet de générer des représentations visuelles de graphes, comme celui de la figure 2. Il a un intérêt que pour les graphes basiques.

4.2 Script de comparaison des sorties

Après l'utilisation de l'option -d du main (pour écrire les sorties des algorithmes dans des fichiers), il est possible de comparer les sorties de chaque algorithme.

Le script à côté permet de faire des différences entre les fichiers de sortie du -d, pour vérifier que les algorithmes produisent bien la même sortie.

Si tous les algorithmes produisent la même sortie que l'algorithme basique, on suppose qu'ils sont exactes, même s'il peut y avoir des différences sur des cas limites, qui ne nous intéressent pas particulièrement dans ce projet.

```
#!/bin/bash
a="FW_seq.txt"
b="FW_par1_l.txt"
c="FW_par1_c.txt"
d="FW_par2.txt"
e="FW_seq_tiled.txt"
f="FW_par_tiled.txt"
g="FW_seq_tiled_layout.txt"
h="FW_par_tiled_layout.txt"
j="FW_seq_tiled_layout_memPos.txt"
k="FW_seq_tiled_layout2.txt"
l="FW_seq_mem.txt"
m="FW_seq_1d.txt"
n="FW_par_1d.txt"

echo "simple_graph analysis..."
for i in {0..29};
do
    diff "simple_graph$i$a" "simple_graph$i$b"
    diff "simple_graph$i$a" "simple_graph$i$c"
    diff "simple_graph$i$a" "simple_graph$i$e"
    diff "simple_graph$i$a" "simple_graph$i$f"
    diff "simple_graph$i$a" "simple_graph$i$g"
    diff "simple_graph$i$a" "simple_graph$i$h"
    diff "simple_graph$i$a" "simple_graph$i$l"
    diff "simple_graph$i$a" "simple_graph$i$m"
    diff "simple_graph$i$a" "simple_graph$i$n"
done

echo "medium_graph analysis..."
for i in {0..29};
do
    diff "medium_graph$i$a" "medium_graph$i$b"
    diff "medium_graph$i$a" "medium_graph$i$c"
    diff "medium_graph$i$a" "medium_graph$i$e"
    diff "medium_graph$i$a" "medium_graph$i$f"
    diff "medium_graph$i$a" "medium_graph$i$g"
    diff "medium_graph$i$a" "medium_graph$i$h"
    diff "medium_graph$i$a" "medium_graph$i$l"
    diff "medium_graph$i$a" "medium_graph$i$m"
    diff "medium_graph$i$a" "medium_graph$i$n"
done

echo "complex_graph analysis..."
for i in {0..29};
do
    diff "complex_graph$i$a" "complex_graph$i$b"
    diff "complex_graph$i$a" "complex_graph$i$c"
    diff "complex_graph$i$a" "complex_graph$i$e"
    diff "complex_graph$i$a" "complex_graph$i$f"
    diff "complex_graph$i$a" "complex_graph$i$g"
    diff "complex_graph$i$a" "complex_graph$i$h"
    diff "complex_graph$i$a" "complex_graph$i$l"
    diff "complex_graph$i$a" "complex_graph$i$m"
    diff "complex_graph$i$a" "complex_graph$i$n"
done
```

5 Comparatif de performances des algorithmes CPU

Les comparatifs se font sur les graphes de taille au moins moyenne.

La configuration utilisée est la suivante :

— Processeur : Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz
— 4 threads
— Taille de cache : 3072 KB
— Compilateur : gcc 6.2
— Optimisation : O3

5.1 Graphes moyens

Algorithme	Performances meilleures que l'algo basique	Temps total	Meilleur cas	Pire cas
FWSeq		6.41		
FWSqM	29	6.24	118% meilleur	0%, égual
FWSTB	15	5.33	2308% meilleur	68% pire
FWSTI	23	2.95	4433% meilleur	64% pire
FWP1C	28	2.95	3640% meilleur	38% pire
FWP1L	30	2.80	4084% meilleur	16% meilleur
FWS1D	23	2.66	3913% meilleur	63% pire
FWSMS	30	2.48	17600% meilleur	17% meilleur
FWPTB	30	2.32	3379% meilleur	6% meilleur
FWPTI	29	1.39	5745% meilleur	60% pire
FWP1D	30	1.25	6946% meilleur	100% meilleur

5.2 Graphes complexes

Algorithme	Performances meilleures que l'algo basique	Temps total	Meilleur cas	Pire cas
FWSeq		94.24		
FWSqM	25	93.97	2% meilleur	7% pire
FWSTB	16	74.94	4671% meilleur	47% pire
FWP1C	30	60.23	747% meilleur	1% meilleur
FWP1L	30	49.36	3891% meilleur	11% meilleur
FWSTI	30	44.19	6010% meilleur	12% meilleur
FWSMS	30	38.79	11861% meilleur	24% meilleur
FWS1D	30	38.69	6952% meilleur	24% meilleur
FWPTB	30	28.76	6318% meilleur	57% meilleur
FWPTI	30	20.88	7547% meilleur	100% meilleur
FWP1D	30	17.73	11506% meilleur	100% meilleur

5.3 Graphes très complexes

Algorithme	Performances meilleures que l'algo basique	Temps total	Meilleur cas	Pire cas
FWSeq		1007		
FWSTB	2	1535	263% meilleur	54% pire
FWP1C	7	1243	33% meilleur	37% pire
FWSqM	20	1016	3% meilleur	8% pire
FWP1L	13	1006	107% meilleur	30% pire
FWSTI	30	729	232% meilleur	6% meilleur
FWSMS	30	706	215% meilleur	11% meilleur
FWS1D	30	694	263% meilleur	12% meilleur
FWPTB	29	589	641% meilleur	9% meilleur
FWPTI	30	365	585% meilleur	36% meilleur
FWP1D	30	335	647% meilleur	100% meilleur

5.4 Analyse

Cette partie est relativement uniquement à ma machine. Les résultats peuvent être très différents sur des machines plus puissantes.

L'algorithme parallèle avec la matrice en une dimension (et les deux optimisations basiques de mémoïzation et sauts d'infinis) est le meilleur.

C'est un cas particulier de l'algorithme en blocs, qui devrait théoriquement être meilleur. Par ailleurs l'algorithme FWPTI, qui est un FWPTB sans changer la représentation en mémoire de la matrice, fournit de meilleurs résultats. C'est un résultat étranger étant donné que l'algorithme est conçu pour utiliser une représentation spatiale différente des données, de plus, avec gprof il est possible de voir que les fonctions qui changent la représentation de la matrice en mémoire ne prennent pas énormément de temps.

J'imagine que plus les graphes sont grands, plus l'écart de performance se réduit entre FWPTB et FWP1D. De plus il est raisonnable d'imaginer que le FWPTB possède des meilleurs performances avec un cache plus grand.

Nous pouvons également voir que l'algorithme basique avec mémoïzation possède environ les mêmes performances que l'algorithme basique. Cela s'explique par le O3 de gcc 6.2. En effet j'ai travaillé quasiment tout le temps avec le O3 de gcc 4.8.4, pour lequel la mémoïzation est un grand gain de temps.

On peut en conclure que le O3 est devenu suffisamment intelligent pour gérer seul la mémoïzation.

D'autre part, les résultats au meilleur cas peuvent s'expliquer avec les sauts de valeurs infinis. Les 30 graphes étant générés de manière aléatoire, s'il y a très peu d'arêtes en proportion (des arêtes infinies donc), les performances seront excellentes. Ainsi E n'influe pas la complexité mais influence les performances.

De plus, l'algorithme FWP1L est meilleur que l'algorithme FWP1C, car les lignes de la matrice sont localizées dans la mémoire (même avec std : :vector, qui utilise la même représentation spatiale que des tableaux 2D). Ils sont cependant plutôt mauvais, car possèdent des pires performances que le FWSMS en utilisant les mêmes méthodes.

6 Analyse comparative synthétique des principaux algorithmes

Voici les temps de calcul des principaux algorithmes, exprimés en secondes, sur des graphes de référence.

6.1 Avec O3

Algorithme	**V = 400**	**V = 800**	**V = 1200**	**V = 1600**	**V = 2000**	**V = 3200**
Basique	0.062	0.40	1.32	3.15	6.12	24.36
Basique avec mémoïzation	0.059	0.40	1.31	3.16	6.11	24.38
Basique avec mémoïzation et sauts d'infinis	0.051	0.30	0.77	1.33	1.96	10.37
Séquentiel 1d	0.069	0.33	0.81	1.41	2.02	10.54
Parallèle 1d 2 threads	0.039	0.18	0.45	0.75	1.09	5.67
Parallèle 1d 4 threads	0.032	0.15	0.39	0.64	0.95	5.08

6.2 Sans O3

Algorithme	**V = 400**	**V = 800**	**V = 1200**	**V = 1600**	**V = 2000**	**V = 3200**
Basique	1.03	8.08	27.1	64.29	125.9	514
Basique avec mémoïzation	0.73	5.76	19.28	45.65	89.1	352
Basique avec mémoïzation et sauts d'infinis	0.63	4.03	10.47	17.35	25.3	145
Séquentiel 1d	0.42	2.64	6.86	11.37	16.6	94
Parallèle 1d 2 threads	0.40	1.57	4.13	6.93	11.73	56
Parallèle 1d 4 threads	0.23	1.46	3.75	6.25	9.91	52

7 Comparatif de performances GPU vs CPU

La configuration utilisée dans cette partie est différente, mon ordinateur habituel n'ayant pas accès à OpenCL.
Voici les caractéristiques du CPU :

- Processeur : Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz
- 4 threads
- Taille de cache : 6144 KB
- Compilateur : g++ 5.4.0
- Optimisation : O3

Voici celles du GPU :

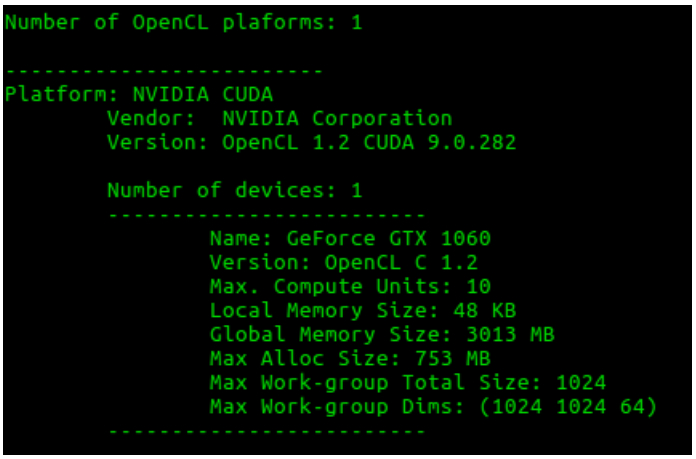


FIGURE 13 – Caractéristiques du GPU

Les résultats sont les suivants, sur le graphe de référence numéro 5 :

Algorithme	**V = 3200**
Basique	22.4745
Basique avec mémoïzation	22.826
Basique avec mémoïzation et sauts d'infinis	9.8378
Séquentiel 1d	10.7011
parallèle 1d 4 threads	5.4638
FWPTB	5.4165
Parallèle GPU OpenCL	1.4488

On constate que l'implémentation GPU possède de meilleures performances, ce qui était relativement prévisible étant donné le nombre de threads impliqué.

8 Explication des résultats

8.1 L'importance de la gestion du cache en ligne

En C++ la mémoire des tableaux 2D est organisée en lignes, d'où le résultat suivant :

```
File : medium_graph12.txt
V = 356;
FWSeq :: Floyd_Warshall Sequential :
        Time elapsed :: 1.69827s ::  using 1 threads;

==6113==
==6113== I   refs:      499,960,040
==6113== I1  misses:          2,511
==6113== LLi misses:          2,145
==6113== I1  miss rate:        0.00%
==6113== LLi miss rate:        0.00%
==6113==
==6113== D   refs:      136,551,126  (136,203,123 rd   + 348,003 wr)
==6113== D1  misses:      1,507,471  (  1,492,321 rd   +  15,150 wr)
==6113== LLd misses:         20,476  (      6,625 rd   +  13,851 wr)
==6113== D1  miss rate:         1.1% (        1.0%     +     4.3%  )
==6113== LLd miss rate:         0.0% (        0.0%     +     3.9%  )
```

```
for (int k = 0; k < V; k++)
{
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (graph[i][j] > graph[i][k] + graph[k][j]) {
                graph[i][j] = graph[i][k] + graph[k][j];
            }
        }
    }
}
```

```
File : medium_graph12.txt
V = 356;
FWSeq :: Floyd_Warshall Sequential :
        Time elapsed :: 3.69628s ::  using 1 threads;

==6134==
==6134== I   refs:      589,942,613
==6134== I1  misses:          2,510
==6134== LLi misses:          2,145
==6134== I1  miss rate:        0.00%
==6134== LLi miss rate:        0.00%
==6134==
==6134== D   refs:      181,542,053  (181,194,051 rd   + 348,002 wr)
==6134== D1  misses:    101,395,286  (101,380,135 rd   +  15,151 wr)
==6134== LLd misses:         20,480  (      6,628 rd   +  13,852 wr)
==6134== D1  miss rate:        55.8% (       55.9%     +     4.3%  )
==6134== LLd miss rate:         0.0% (        0.0%     +     3.9%  )
```

```
for (int k = 0; k < V; k++)
{
    for (int j = 0; j < V; j++)
    {
        for (int i = 0; i < V; i++)
        {
            if (graph[i][j] > graph[i][k] + graph[k][j]) {
                graph[i][j] = graph[i][k] + graph[k][j];
            }
        }
    }
}
```

Pour expliquer cela, on s'intéresse à la dernière boucle dans chaque cas.

Dans le premier cas, graph[i][j] est une constante de la boucle, seulement graph[i][j] et graph[k][j] dépendent de la variable de la boucle. Ainsi, pour charger toutes les valeurs de la boucle, il n'y a qu'à charger les lignes i et k de la matrice.

Dans le second cas, graph[i][j] et graph[i][k] dépendent de la variable de la boucle. Mais pour chaque valeur de i, il faut charger en mémoire une nouvelle ligne de la matrice.

Cela génère beaucoup d'erreurs de cache, et explique pourquoi le second cas est beaucoup plus lent que le premier.

8.2 L'importance de la représentation spatiale de la mémoire

Cette partie essaie d'expliquer pourquoi l'algorithme avec la matrice en une dimension est le meilleur.

Comme la taille de la matrice des distances est connue à l'exécution, les tableaux sont dynamiques.

C'est pour cela que j'ai utilisé des vecteurs de vecteurs. Ils fonctionnent comme un tableau 2D en C, avec des pointeurs de pointeurs.

Lorsque que la matrice n'est pas en une dimension, la localisation spatiale en mémoire se perd, car chaque ligne peut être loin de la ligne suivante.

9 Possibilités d'amélioration

La première chose à faire est de tester les performances sur une machine plus puissante. Le meilleur algorithme devrait être le FWPTB, avec une taille de bloc égal à la taille du cache. Le script Python **generate_blocsize_analysis.py** peut tester différentes tailles de blocs. Il s'appelle depuis le dossier principal.

D'autre part, il est possible d'implémenter l'algorithme de Floyd-Warshall de manière récursive, avec la représentation en mémoire Z-Morton.

10 Bibliographie

Tutoriel OpenMP :

https ://computing.llnl.gov/tutorials/openMP/

Algorithme de Floyd-Warshall :

https ://www.cs.usfca.edu/galles/visualization/Floyd.html

http ://www.mcs.anl.gov/itf/dbpp/text/node35.html

http ://www.infor.uva.es/diego/docs/ortega13b.pdf

http ://www.cse.buffalo.edu/faculty/miller/Courses/CSE633/Muthuraman-Spring-2014-CSE633.pdf

http ://courses.csail.mit.edu/6.884/spring10/projects/kelleyk-neboat-slides.pdf

http ://citeseerx.ist.psu.edu/viewdoc/download ?doi=10.1.1.40.5378&rep=rep1&type=pdf

https ://prezi.com/z9eiteeaqxmh/parallel-approach-to-floyd-warshall-algorithm/

https ://people.cs.kuleuven.be/george.karachalias/papers/floyd-warshall.pdf

http ://moais.imag.fr/membres/marc.tchiboukdjian/pub/thesis.pdf

http ://escholarship.org/uc/item/9v89p5wv#page-10

http ://repository.upenn.edu/cgi/viewcontent.cgi ?article=1213&context=hms

Z-Morton order :

https ://en.wikipedia.org/wiki/Z-order_curve

Défauts de cache :

http ://valgrind.org/docs/manual/cg-manual.html

http ://igoro.com/archive/gallery-of-processor-cache-effects/