Rapport
GPU

Chen Marc-Yong
Taleb Aslan

Exercice 1 :

Le but de cet exercice est de créer une carte permettant de lire la visibilité d'une carte depuis un point d'observation. Pour cela nous avons une image nous donnant la hauteur des points de la carte et un point d'observation.

Pour effectuer la première version naïve sur CPU, nous avons retranscrit les étapes fournis dans le TP, on parcourt chaque point de notre carte (h_in) et on calcule la dda entre chaque point et le point d'observation fournis en paramètre ici le point d'observation C (245, 497).

```
for(int Py = 0; Py < MapHeight; Py++)
{
    for(int Px = 0; Px < MapWidth; Px++)
    {

    // DDA entre le point c (Cx, Cy) et le point P (Px, Py);
        float Dx, Dy, Dz,  D;

        Dx = Px - Cx;   // delta x
        Dy = Py - Cy;   // delta y
        Dz = h_in[Py * MapWidth + Px] - h_in[Cy * MapWidth + Cx];   // delta z
        D = max(abs(Dx), abs(Dy));  // delta positif max entre Dx et Dy
        float angle, angle_ref = atan(Dz / sqrt((Dx * Dx) + (Dy * Dy)));

        float incX = (Dx / D);
        float incY = (Dy / D);
```

On définit toutes les variables pour calculer les dda, le delta x, delta y et le delta sur la hauteur, le nombre de points qui compose la dda (D) qui correspond au max entre la valeur absolue de delta x et delta y.

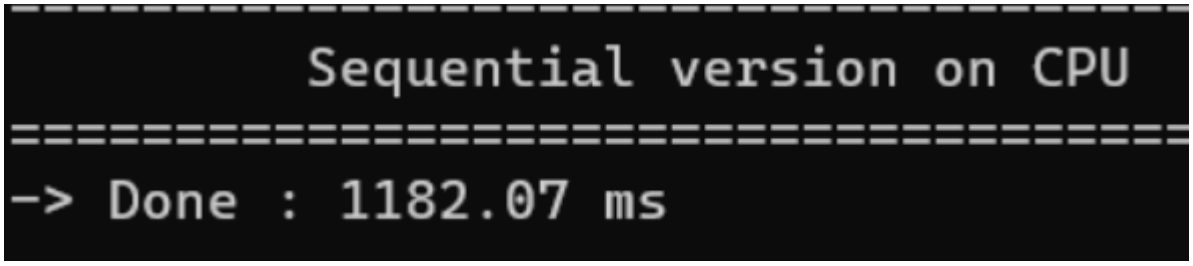
```
for (int i = 0; i < D - 1; i++)
{
    int x = Cx + incX * i;
    int y = Cy + incY * i;
    Dx = Px - x;
    Dy = Py - y;
    Dz = h_in[Py * MapWidth + Px] - h_in[y * MapWidth + x];
    // Calcule Angle
    angle = atan(Dz / sqrt((Dx * Dx) + (Dy * Dy)));
```

Pour obtenir tous les points de la dda, on boucle sur le nombre D et on calcule les coordonnées du point (x, y) ensuite on calcule à nouveau nos delta avec le point obtenu.
Nous calculons l'angle formé entre le nouveau point et le point d'observation.

```
if (angle_ref >= angle)
{
    //h_out.setPixel(Px, Py, 0);
    h_out[Py * MapWidth + Px] = 0;
    break;
}
```

on va comparer tous les angles obtenus avec l'angles de référence qui correspond à l'angle formé entre le point de base et le point d'observation, si l'angle de référence est plus grand qu'un angle formé dans notre dda alors il n'y a pas de visibilité entre notre point et notre point d'observation.

Pour lancer le kernel, on prend en entrée le tableau des hauteurs et un tableau sortie vide de taille correspondante au premier. On alloue la mémoire sur le GPU pour les deux tableaux, et on copie les données du premier tableau vers le GPU, on effectue le kernel, on copie le tableau de sortie dans le CPU et on libère la mémoire.



Le temps pour effectuer la méthode naïve sur CPU

Pour faire une première méthode naïve sur GPU, chaque thread de notre grille va faire le même travail que sur CPU pour chaque point de la carte, c'est-à-dire que chaque thread va effectuer une dda et calculer chaque angle et les comparer avec son angle de référence.

```
__global__ void kernelMapV1(uint8_t *h_in, uint8_t *h_out, const int MapWidth, const int MapHeight, const int Cx, const int Cy)
{
    for(int Py = blockDim.y * blockIdx.y + threadIdx.y; Py < MapHeight; Py += blockDim.y * gridDim.y)
    {
        for(int Px = blockDim.x * blockIdx.x + threadIdx.x; Px < MapWidth; Px += blockDim.x * gridDim.x)
        {
            int Dx, Dy, Dz,  D;
            Dx = Px - Cx;   // delta x
            Dy = Py - Cy;   // delta y
            Dz = h_in[Py * MapWidth + Px] - h_in[Cy * MapWidth + Cx];   // delta z
            D = max(abs(Dx), abs(Dy));  // delta positif max entre Dx et Dy
            float incX = ((float)Dx / D), incY = ((float)Dy / D);
            float angle_ref = atan(Dz / __fsqrt_rn((Dx * Dx) + (Dy * Dy)));
            h_out[Py * MapWidth + Px] = 255;
            for(int i = 0; i < D; i++)      // boucle dans la dda
            {
                int x = Cx + i * incX;
                int y = Cy + i * incY;
                Dx = Px - x;
                Dy = Py - y;
                Dz = h_in[Py * MapWidth + Px] - h_in[y * MapWidth + x];
                float angle = atan(Dz / __fsqrt_rn((Dx * Dx) + (Dy * Dy)));
                if (angle_ref > angle)
                {
                    h_out[Py * MapWidth + Px] = 0;
                    break;
```

La seul différence avec la méthode sur CPU est dans la double boucle imbriqué Py correspond à l'identifiant du thread suivant l'axe y et si la hauteur de l'image d'entré (MapHeight) est plus grand que le nombre de threads dans la grille alors les threads pourront effectuer plusieurs calculs de dda, pour cela avec Py += blockDim.y * gridDim.y qui incrémente Py du nombre de threads selon l'axe y de notre grille.
On fait la même opération avec Px sur l'axe x et on suit le même procéder que sur la méthode sur CPU à la seul différence que l'utilisation de la méthode sqrt fait le calcule sur le CPU, nous ne pouvons donc pas utiliser cette méthode sur GPU alors on a utilisé __fsqrt_rn.

```
==========================================
          Parallel version on GPU
==========================================
5513 5513
Done kernelMap : 325.960022 ms
```

Le temps d'exécution du kernel pour la méthode naîve sur GPU.

Maintenant pour optimiser le temps de calcul il faut essayer de réduire le nombre de calculs que l'on effectue dans le kernel, pour le moment nous effectuons un calcul de l'angle pour chaque point dans chaque dda, comme un point de la carte est traversé par plusieurs dda, cela veut que nous effectuons le calcul d'angle plusieurs fois le même points. Nous pouvons essayer d'optimiser ça en utilisant un kernel qui est dédié à calculer les angles entre tous les points de la carte et le point d'observation. On stocke tous les angles obtenus dans un tableau.

On a aussi créé une méthode __device__ CalcAngle qui retourne l'angle.

```
__device__ float calcAngle(int Dx, int Dy, int Dz)
{
        return atan(Dz / __fsqrt_rn((Dx * Dx) + (Dy * Dy)));
}
```

```
__global__ void kernelAngle(uint8_t *h_in, uint8_t *h_out, float *angle, const int MapWidth, const int MapHeight, const int Cx, const int Cy)
{

    int Dx, Dy, Dz;
    for(int Py = blockDim.y * blockIdx.y + threadIdx.y; Py < MapHeight; Py += blockDim.y * gridDim.y)
    {
        for(int Px = blockDim.x * blockIdx.x + threadIdx.x; Px < MapWidth; Px += blockDim.x * gridDim.x)
        {
            Dx = Px - Cx;
            Dy = Py - Cy;
            Dz = h_in[Py * MapWidth + Px] - h_in[Cy * MapWidth + Cx];
            angle[Py * MapWidth + Px] = atan(Dz / __fsqrt_rn((Dx * Dx) + (Dy * Dy)));
        }
    }
}
```

Dans le kernelAngle, on prend en entré deux tableaux, le premier correspond au tableau d'entrée des hauteurs et le deuxième correspond au tableau de sortie qui contiendra tous les angles.

On effectue une double boucle imbriqué dans laquelle chaque thread va effectuer un ou plusieurs calculs d'angle.

Avec ce kernel nous pouvons utiliser le tableau contenant tous les angles et juste effectuer une comparaison au lieu de devoir calculer à nouveau les angles, ce qui réduit le temps d'exécution.

```
==========================================
          Parallel version on GPU
==========================================
5513 5513
-> Done : 183.10 ms en moyenne
```

Ensuite nous pouvons réduire le nombre de dda à effectuer, pour le moment on effectue une dda sur tous les points, quand on fait une dda on remarque que l'on obtient les dda de tous les points qui sont composés, par exemple si on a une liste de points formant une dda suivante L = { P0, P1, P2, …, Pn }, on voit que dans cette liste on a une dda entre P0 et Pn mais on remarque que l'on a aussi la dda entre P1 et Pn, et ainsi de suite jusqu'à Pn, donc plus la liste de points est grande, plus on obtient de dda à l'intérieur de notre liste. *

Il faut chercher à obtenir un minimum de dda composé d'un maximum de points et dont tous les points de la carte seront accessibles au moins une fois.
En faisant une dda sur chaque point sur les bords de la carte, chaque point de la carte sera accessible au moins une fois en un minimum de dda.

```cpp
__global__ void kernelMapOptimize(uint8_t *h_in, uint8_t *h_out, float *angle, const int MapWidth, const int MapHeight, const int Cx, const int Cy)
{
    int Dx, Dy, Dz, D;

    for(int Py = blockDim.y * blockIdx.y + threadIdx.y; Py <= MapHeight; Py += blockDim.y * gridDim.y)
    {
        for(int Px = blockDim.x * blockIdx.x + threadIdx.x; Px <= MapWidth; Px += blockDim.x * gridDim.x)
        {
            if (Px == 0 || Py == 0 || Px == MapWidth || Py == MapHeight)
            {
                D = max(abs(Dx), abs(Dy));
                float incX = ((float)Dx / D), incY = ((float)Dy / D);
                float angle_max = - M_PI/2;         // min de arctan
                for (int i = 0; i < D; i++)
                {
                    int hx = Cx + i * incX;
                    int hy = Cy + i * incY;
                    float current_angle = angle[hy * MapWidth + hx];         //calcAngle(Dx, Dy, Dz);
                    if (angle_max < current_angle)
                    {
                        h_out[hy * MapWidth + hx] = 255;
                        angle_max = current_angle;
```

On met une condition supplémentaire qui effectue les dda uniquement si notre point sur l'axe x est égale à 0 ou à la largeur de la carte ou y est égale à 0 ou à la longueur de la carte.

En changeant la façon dont on utilise les angles de la manière suivante, on déclare une variable qui va stocker l'angle maximum parcouru dans la dda en débutant par le point le plus proche du point d'observation.

On initialise angle_max à la valeur minimum possible c'est-à-dire -PI/2, on fait le parcours de la dda, si l'angle du point en cours est supérieur à angle_max cela veut dire que le point est visible depuis le point d'observation, on l'inscrit dans notre tableau de sortie et angle_max devient l'angle du point en cours.

```
==========================================
          Parallel version on GPU
==========================================
5513 5513
Done kernelMap : 0.673792 ms
-> Done : 54.65 ms en moyenne
```

Exercice 2:

Le but de cet exercice est de minimiser la taille d'une carte en créant une carte des hauteur de taille 10 * 10 dont chaque point correspond à la hauteur maximum d'une tuile de taille dépendant de la taille de la carte originale, chaque tuile sera de taille (largeur / 10, longueur / 10).

```
void tiledMap(uint8_t *h_in, uint8_t *h_out,const int inMapWidth, const int inMapHeight , const int outMapWidth,
{
    int TiledDimX = int(inMapWidth / outMapWidth);
    int TiledDimY = int(inMapHeight / outMapHeight);

    for(int TiledIdY = 0; TiledIdY < outMapHeight; TiledIdY++)
    {
        for(int TiledIdX = 0; TiledIdX < outMapWidth; TiledIdX++)
        {
            uint8_t maxHeight = 0;
            int location = (TiledIdY * TiledDimY) * (inMapWidth) + TiledIdX * TiledDimX;
            for(int y = 0; y < TiledDimY; y++)
            {
                for(int x = 0; x < TiledDimX; x++)
                {
                    int point = location + y * inMapWidth + x;
                    if (h_in[point] > maxHeight) maxHeight = h_in[point];
                }
            }
            h_out[TiledIdY * outMapWidth + TiledIdX] = maxHeight;
        }
    }
}
```

Pour la méthode naïve sur CPU, comme décrit plus haut le but est de diviser la carte en une grille de 10*10 tuiles, tout d'abord on recherche la taille de chaque tuile, avec ici TiledDimX et TiledDimY.
Ensuite dans chaque tuile, on recherche la hauteur maximum, on va donc boucler sur tous les points de la tuile est comparer la hauteur de chaque point avec la variable qui stocke la hauteur max (maxHeight).

Pour calculer la position de chaque point dans une tuile, on va d'abord déterminer la position du premier point de la tuile avec la variable localisation et ensuite on ajoute la position dans la tuile pour obtenir la position globale dans la carte.

```
==========================================
          Sequential version on CPU
==========================================
-> Done : 0.70 ms
```

```
__global__ void kernelTiled(uint8_t *h_in, uint8_t *h_out,const int inMapWidth, const i
{
    int TiledDimX = int(inMapWidth / outMapWidth);
    int TiledDimY = int(inMapHeight / outMapHeight);

    __shared__ int max;

    int location = (blockIdx.y * TiledDimY) * (inMapWidth) + blockIdx.x * TiledDimX;

    for(int Py =threadIdx.y; Py < TiledDimY; Py += blockDim.y)
    {
        for(int Px =threadIdx.x; Px < TiledDimX; Px += blockDim.x)
        {
            int point = location + Py * inMapWidth + Px;
            atomicMax(&max, int(h_in[point]));
        }
    }
    __syncthreads();
    h_out[blockIdx.y * outMapWidth + blockIdx.x] = max;
}
```

Pour la méthode sur GPU, on va utiliser une mémoire partagée contenant la valeur maximum de la hauteur dans le bloc, ensuite une utilise une opération atomique atomicMax car les threads du bloc peuvent accéder à l'espace mémoire contenant la valeur de variable max en même moment ce qui peut créer des conflits. atomicMax va comparer la valeur stocker dans max et la hauteur du point, si la valeur de la hauteur du point est plus importante que max alors la valeur de max va devenir celle de la hauteur du point.
On utilise __syncthreads() pour synchroniser tous les threads du même bloc et enfin on met dans le tableau de sortie la valeur de max à la position correspondant à la tuile.