ENSEIRB-MATMECA

TROISIÈME ANNÉE, DÉPARTEMENT ÉLECTRONIQUE

SYSTÈMES EMBARQUÉS

Calcul haute performance pour les systèmes embarqués (HPEC)

- Galaxeirb -

Auteurs:
FERIAT Abdessamad
MOUHAGIR Ayoub

Professeur:
Jérémie CRENNE

09 Décembre 2019

Table des matières

1 Introduction

Ce module a pour but d'implémenter une application de calcul mathématiques intensif sur un système embarqué (System On Chip). L'application choisie consiste à simuler le déplacement de deux galaxies en fonction de leur gravitation. Il s'agit de la collision entre la voie lactée "Milky Way" et Andromède "Andromeda" qui, selon les observations actuelles, se produira dans approximativement quatre milliards d'années entre les deux plus grandes galaxies du Groupe local. Pour ce faire, on dispose d'une base données listant les composants des deux galaxies, leurs positions, leurs masses et leurs vélocités.

La carte de développement qui sera utilisé (Figure 1)est une carte Nvidia Jetson K1 embarquant quatre coeurs de processeur Cortex A15 (architecture ARMv7) et 192 coeurs CUDA GK20A ainsi que 2GB de RAM.

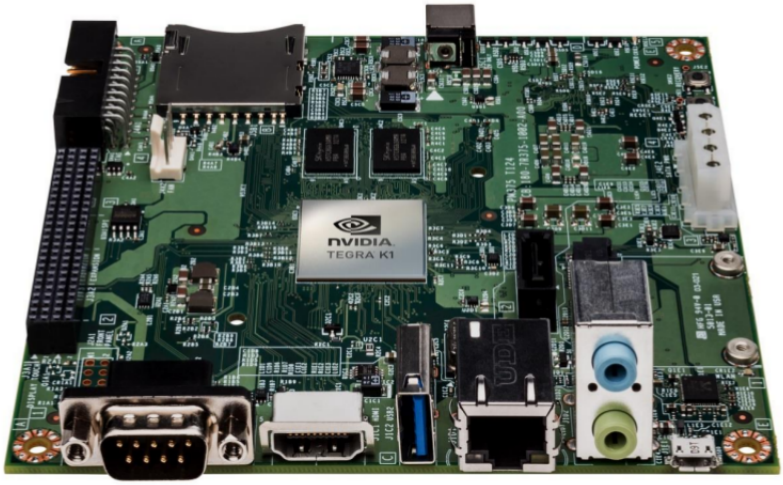

FIGURE 1 – Carte de développement NVIDIA Jetson TK1

Le développement s'effectuera en plusieurs étapes. Tout d'abord une implémentation basique des algorithmes de calculs. Ensuite, on procédera à une optimisation algorithmique concentré dans un premier temps sur le CPU puis dans un second temps sur l'utilisation du GPU. Afin de réaliser ce projet, nous avons à notre disposition plusieurs librairies tel que SDL, OpenGL, OpenMP, CUDA, etc.

2 Présentation de la simulation

Le modèle de simulation adopté pour ce projet est le **N-Body Simulation** ou **Problème à N-Corps**. Ce modèle consiste à résoudre les équations du mouvement de Newton de N corps interagissant gravitationnellement, connaissant leurs masses ainsi que leurs positions et vitesses initiales. Dans notre simulation, les deux galaxies seront simulées par un ensemble de particules caractérisées par une position initiale, une masse et une vélocité. Toutes ces informations initiales sur les particules sont accessibles depuis le fichier *dubinski.tab* fournie par John Dubinski.

2.1 Modèle de Galaxies : La voie lactée & Andromède

Les deux galaxies sont constituées d'un ensemble de particules (40960 particules chacune) caractérisant les trois parties qui forment une galaxie (Figure 2) :

- **Disque** : Disque galactique est le plan dans lequel se trouvent les bras spiraux et le disque des galaxies à disque. (16384 particules)
- **Halo** : Halo galactique est une région de l'espace entourant les galaxies spirales. (16384 particules)
- **Bulge** : Bulge galactique est la partie centrale des galaxies spirales, située dans le disque et entourant le noyau galactique. (8192 particules)

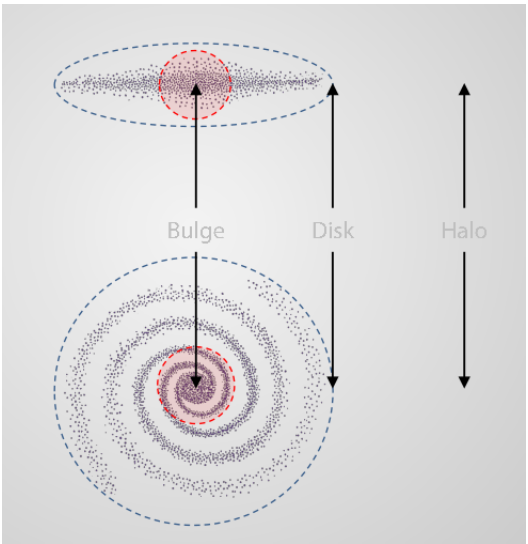

FIGURE 2 – Les différentes parties constituant une galaxie à disque

2.2 Modèle de particules

Chaque particule est caractérisée par sa **masse**, sa **vitesse** et sa **position**. On utilisera le système de coordonnées cartésiennes pour représenter les particules.

Le mouvement de la particule dépend non seulement de sa masse et sa vélocité, mais aussi de celles des particules voisines. Il faut prendre en compte alors la contribution de toutes les particules dans le même corps pour calculer la position

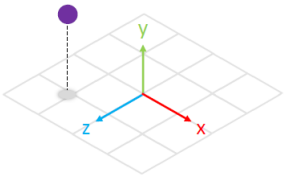

FIGURE 3 – Représentation d'une particule en utilisant le système des coordonnées cartésiennes

d'une seule particule. Ensuite il faut itérer cette opération pour toutes les particules. D'où le principe de la simulation **N-Body**.

La nouvelle position de chaque particule est calculée à partir de sa vélocité. Cette dernière est quant à elle calculée à partir de l'accélération de la particule. Donc pour une particule i, le calcul est effectué de la manière suivante :

$$\vec{p_i} += \vec{v_i} \times DT \tag{1}$$

$$\vec{v_i} += \vec{a_i} \times M \times \varsigma \tag{2}$$

$$\vec{a_i} += \sum_{j=0,\ j\neq i}^{n-1} \vec{\Delta}_{ij} \times M \times \varsigma \times m_j \times \frac{1}{d_{ij}^3} \tag{3}$$

où :

- $\vec{p_i} = (x_i, y_i, z_i)$: Vecteur de position de la particule i.
- $\vec{v_i} = (vx_i, vy_i, vz_i)$: Vecteur de vélocité de la particule i.
- $\vec{a_i} = (ax_i, ay_i, az_i)$: Vecteur d'accélération de la particule i.
- $\vec{\Delta}_{ij} = \vec{p_i} - \vec{p_j}$: Vecteur différence entre les les particules i et j.
- $M = 10$: Constante de masse.
- $\varsigma = 1$: Facteur d'amortissement.
- d_{ij} : Distance entre les particules i et j calculée à partir de la relation suivante : $d_{ij} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2}$.
- m_j : Masse de la particule j.
- n : Nombre de particules.

3 Avancement

3.1 Simulation sur le CPU

Dans un premier temps, on définit toute fonction nécessaire pour réaliser la simulation, notamment celle qui permet de récupérer les coordonnées de chaque particule et les stocker dans des listes prédéfinies pour pouvoir les manipuler ensuite *nom de la fonction*. Ainsi qu'une autre qui permet d'afficher ces particules dans une fenêtre *ShowGalaxies(particule_t *p, int size)*.

Pour l'instant, on a un affichage visuelle de la totalité des particules des deux galaxies. On procédera alors à la simulation de la collision entre ces deux galaxies. Pour ce faire, on s'intéressera à un ensemble de 1024 particules pour ne pas saturer

le CPU. Ces 1024 particules seront choisies d'une manière à couvrir les trois parties de la galaxie (disque, bulge et halo).

Ensuite, on met à jour la vélocité de chaque particule à l'aide de la fonction *update_velocity(particule_t *p, int size)* qui implante les équations 3. Puis, on affiche ces particules à chaque itération. Le schéma 4 résume les étapes de la réalisation de la simulation.

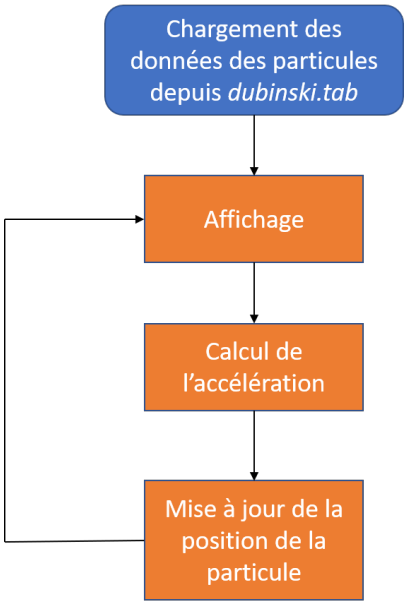

FIGURE 4 – Étapes de la simulation

On charge les particules dans les listes prédéfinies. Puis dans une boucle infinie, on affiche ces particules. Ensuite on procède à la mise à jour de la position de chaque particule en déterminant son accélération à partir de l'effet des autres particules voisines.

Ce premier code nous a permis à la fois de vérifier le bon fonctionnement de toutes les fonctions définies qui nous serviront à la suite et d'avoir un point de départ au niveau de son efficacité sur lequel on se base pour l'optimisation de calcul. L'efficacité de notre solution est mesurée par le facteur FPS (Frame per second ou image par seconde) qui désigne le nombre d'images affichées en une seconde. Plus le nombre d'images est élevé, plus l'animation semble fluide. Pour l'instant avec ce premier code, on arrive à un FPS qui égale à 3.

3.1.1 Optimisations sur le CPU

Pour l'instant, on stocke les informations des particules (masse, position et vélocité) dans différents tableaux. Afin d'utiliser qu'un seul tableau, on introduit une structure de particule appelée `particule_t` comme suit :

```
typedef struct particule
{
        float m;
        float x;
        float y;
        float z;
        float vx;
        float vy;
        float vz;

} particule_t;
```

FIGURE 5 – Listing de la structure **Particule_t**

De plus, au lieu de charger toutes les 81920 particules pour pouvoir en tirer que 1024 qui servent dans la simulation. On pourra parcourir le fichier *dubunski.tab* de telle sorte qu'à charger nos 1024 particules.

La deuxième optimisation réalisée est au niveau du calcul de l'accélération en utilisant la fonction *sqrtf()* au lieu de *sqrt()* :

```
for (i = 0; i!=j && i<N; ++i)
{
    dx = p[i].x-p[j].x;
    dy = p[i].y-p[j].y;
    dz = p[i].z-p[j].z;

    d=sqrt(dx*dx+dy*dy+dz*dz);
    if ( d < 1.0 ) d = 1.0;
    fact=p[i].m/(d*d*d);
    ax += dx*p[i].m/(d*d*d);
    ay += dy*p[i].m/(d*d*d);
    az += dz*p[i].m/(d*d*d);

}
```

```
for (i = 0; i!=j && i < N; ++i)
{
    dx = p[i].x-p[j].x;
    dy = p[i].y-p[j].y;
    dz = p[i].z-p[j].z;

    d  = dx*dx+dy*dy+dz*dz;
    if ( d < 1.0 ) d = 1.0;
    fact=p[i].m/(d*sqrtf(d));
    ax += dx*fact;
    ay += dy*fact;
    az += dz*fact;

}
```

FIGURE 6 – Listing de modification du code : Avant optimisation (gauche), Après optimisation (Droite)

Une troisième optimisation possible concerne les flags de la compilation. On s'intéresse au degré d'optimisation **-Ox** avec x varie de 1 à 3 ou encore x=fast. Avec ce flag, le compilateur `gcc` optimise le code source et le modifie de façon à ce que le programme ait le même résultat par le chemin le plus court. Plus le degré est fort, plus l'optimisation est bonne, mais plus la compilation est longue et gourmande en mémoire.

Avec ces différentes optimisations, on arrive pour l'instant à un FPS de l'ordre de 40. Je tiens à préciser que ces valeurs de FPS représentent le pique obtenue dans la simulation.

Une dernière optimisation à réaliser pour le calcul en CPU, c'est l'utilisation de la bibliothèque **OpenMP** qui permet de rendre l'exécution des portions de code

parallèle au lieu de séquentiel. Notre CPU sur la carte NVIDIA Jetson TK1 est constitué de quatre coeurs de calcul Cortex-A15. Alors, à l'aide de la bibliothèque `OpenMP`, on peut effectuer le calcul de l'accélération en parallèle sur un maximum de 4 threads.
Pour exploiter cette bibliothèque, on apporte les modifications suivantes :

- Inclusion de la librairie `omp.h`.
- Ajout d'une directive de compilation `#pragma omp parallel for` avant la boucle for à paralléliser. dans notre cas, il concerne celle qui effectue le calcul de l'accélération dans la fonction *update_velocity()*.
- Possibilité à fixer le nombre de threads à utiliser à l'aide de la fonction *omp_set_num_threads(4)*.
- Ajout de l'option de compilation `-fopenmp` pour le compilateur `gcc` au niveau de `Makefile`.

```
1 void update_velocity(particule_t *p, int size){
2         int i,j,k=0;
3 
4         #pragma omp parallel for
5         for (j = 0; j < size; ++j)
6         {......
7          }
8 }
```

FIGURE 7 – listing d'utilisation de la bibliothèque **OpenMP**

Cette dernière optimisation nous a permis d'atteindre un FPS qui est au alentours de 120 images par seconde.

3.2 Simulation sur le GPU

La simulation sur GPU permet d'exploiter les coeurs de calcul graphique pour un calcul plus rapide. L'architecture du GPU est composée de grilles de blocs constitués de threads. Ces threads sont exécutées en parallèle, et sont exécutés à travers des fonctions appelés **kernels**. Par ailleurs, Les threads n'ont accès qu'à la mémoire du GPU. Donc, il faut à chaque fois copier les données à partir de la mémoire du CPU vers celle du GPU. Ceci se fait à travers les ponts CPU/GPU (bridges host/device). La figure 8 illustre l'architecture du GPU.

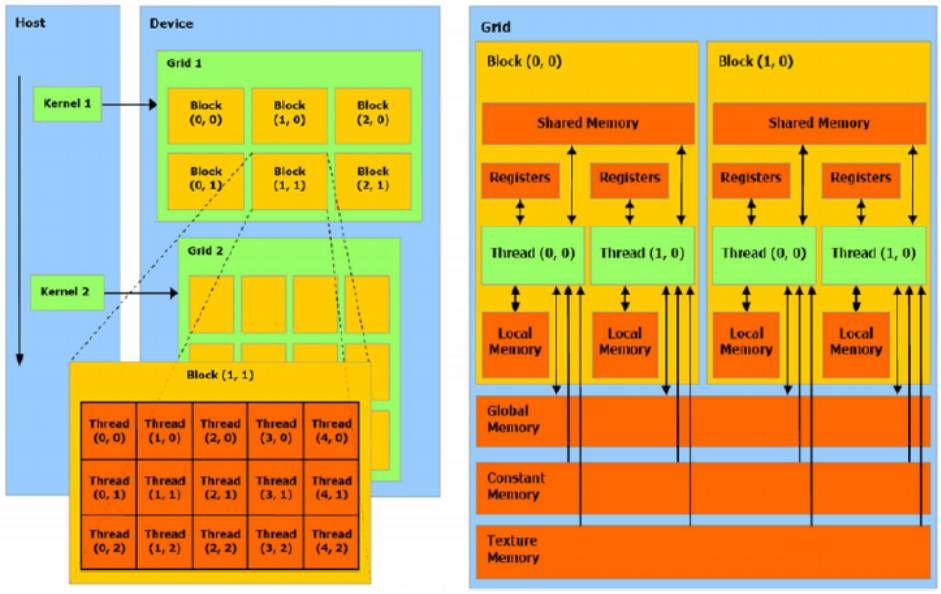

FIGURE 8 – Architecture GPU

Le calcul au niveau du GPU passe les étapes suivantes ;

- Allocation de mémoire GPU.
- Copie des données de la mémoire CPU vers la mémoire GPU.
- Définition des grilles et des blocs qui vont effectuer le calcul.
- Exécution du kernel.
- Copie des données de la mémoire GPU vers la mémoire CPU.
- Libération de la mémoire GPU.

Une première étape consiste à reproduire le code utilisé sur CPU auparavant et l'exécuter sur GPU, en respectant les étapes ci-dessus. Le listing du code ci-dessous montre l'implémentation de ces étapes.

```
1  ////Copy inputs to device
2  CUDA_MEMCPY(d_part, part, sizeof(particule_t) * N,
       cudaMemcpyHostToDevice);
3
4  ////Update acceleration in GPU
5  update_acc( numBlocks, N , d_part, d_acc, N);
6
7  cudaStatus = cudaDeviceSynchronize();
8  if ( cudaStatus != cudaSuccess ) {
9          printf( "error: unable to synchronize threads\n");
10 }
11
12 //Copy Result back to Host
13 CUDA_MEMCPY(acc, d_acc, sizeof(particule_t) * N,
       cudaMemcpyDeviceToHost);
```

FIGURE 9 – Listing du code

La fonction `CUDA_MEMCPY()` permet la copie des données d'une mémoire à une autre. Le dernier argument de la fonction permet de déterminer la mémoire source et de destination :

— cudaMemcpyDeviceToHost : Pour copier de la mémoire GPU vers la mémoire CPU.

— cudaMemcpyHostToDevice : Pour copier de la mémoire CPU vers la mémoire GPU.

La fonction `update_acc()` est le kernel développé sur le GPU dans le fichier *kernel.cu*. il est écrit sous la forme suivante :

```
1  __global__ void kernel_update_acc( particule_t *p, vector_t *acc,
       int size ) {
2          int j = blockIdx.x * blockDim.x + threadIdx.x;
3       ...
4       ..
5       .
6  }
7
8  void update_acc( int nblocks, int nthreads, particule_t *p,
       vector_t *acc, int size) {
9          kernel_update_acc<<<nblocks, nthreads>>>( p, acc, size);
10 }
```

FIGURE 10 – Les fonctions utilisées dans CUDA

En utilisant les variables "built-in" `threadIdx`, `blockIdx` et `blockDim`, on détermine le nombre de blocs qu'on souhaite utiliser. Le mot `__global__` signifie que cette fonction peut être appelé à partir du CPU. D'où son utilisation dans le listing de la figure 9. Dans cette étape, on obtient une simulation à 80 FPS.

3.2.1 Optimisation

Une première étape d'optimisation consiste à calculer l'accélération des particules dans le GPU, puis mettre à jour les positions des particules en utilisant le CPU. Cette étape consiste à transporter le code dans une fonction dans le fichier *kernel.cu*. Cette étape permet d'augmenter le FPS à 160 puisqu'on attribue au GPU une grande partie du calcul qui est le calcul de l'accélération.

La deuxième étape d'optimisation consiste à effectuer le calcul de la position des particules aussi dans le GPU. Ceci est effectué en créant une nouvelle fonction `update_position()` dans le kernel. Le fait de rajouter cette partie du calcul au GPU permet de gagner en performance et d'obtenir une simulation à 250 FPS.

La troisième étape d'optimisation consiste à utiliser les types de variables **float3** et **float4** au lieu de la structure **particule_t**. Ce sont des types de variables "built-in" qui permettent d'optimiser le calcul sur GPU. On utilisera alors une variable **float3** pour la position de la particule, et une variable **float4** pour la vélocité et la masse de la particule. En plus, dans l'algorithme de calcul de l'accélération de la particule, on peut remarquer que la position d'une particule i est constante au

sein de la boucle qui parcourt les autres particules. On peut donc la stocker dans une variable différente (lignes 10, 11 et 12 de la figure 11).

```
1  ..
2  .
3  int i=0;
4  float3 P;
5  float dx,dy,dz,d,fact;
6  if ( j < size ) {
7          acc[j].x = 0.0f;
8          acc[j].y = 0.0f;
9          acc[j].z = 0.0f;
10         P.x =  p[j].x;
11         P.y =  p[j].y;
12         P.z =  p[j].z;
13
14         for (i = 0; i < size; ++i)
15         {
16                 dx = p[i].x-P.x;
17                 dy = p[i].y-P.y;
18                 dz = p[i].z-P.z;
19 ..
20 .
```

FIGURE 11 – utilisation d'une nouvelle variable pour la particule pour laquelle l'accélération est calculée

Cette étape permet de passer à 450 FPS.

La quatrième étape d'optimisation consiste à utiliser la mémoire partagée entre les threads d'un bloc, au lieu de la mémoire globale. Ceci se fait en utilisant le mot **__shared__** avant la définition de la variable partagée qu'on souhaite utiliser dans le kernel. Ensuite, à l'aide de la fonction `make_float()`, on copie les données de la mémoire GPU vers la variable partagée (en l'occurrence la structure particule). Puis on effectue le calcul de l'accélération et de la position de la particule avec cette nouvelle variable. Finalement, on recopie le contenu de la variable partagée vers le GPU. Cette étape permet de passer à 900 FPS environs.

La dernière étape d'optimisation est l'utilisation de la "pinned memory" du GPU. En effet, le GPU ne peut pas accéder directement à la mémoire du CPU. Ainsi lors d'un transfert de données de la mémoire CPU vers la mémoire GPU, on doit d'abord allouer de la mémoire "pinned". L'objectif ici est d'allouer directement de la mémoire "pinned" pour minimiser le coût du transfert de données du CPU vers le GPU. Ceci se fait à l'aide de la fonction `cudaMallocHost()` pour l'allocation de la mémoire "pinned" et `cudaMemcpyAsync()` pour la copie des données vers le GPU. Cette étape permet d'atteindre un affichage à 1400 FPS environs.

4 Bilan et Conclusion

Tout au long de ce projet, nous avons découvert les rudiments du calcul haute performance pour les systèmes embarqués et l'importance de la gestion de données et des accès mémoires en programmation. En plus, ce projet nous a permis de se familiariser avec l'utilisation de CUDA au niveau d'un GPU et de connaître son intérêt pour ce genre d'application (Problème N-Body). Une bonne maîtrise de cette structure permet d'augmenter les performances du calcul d'une façon remarquable.

Vous trouverez ci-dessous le bilan des optimisations réalisées dans le projet pour atteindre un maximum de nombre d'images par seconde (Je tiens à préciser que ces valeurs de FPS représentent le pique des performances obtenues dans la simulation) :

CPU		
Sans Optimisation	sqrtf(), flag –Ofast, etc	OpenMP
3 FPS	40 FPS	160 FPS

GPU				
Calcul d'accélération en GPU / Mise à jour des positions en CPU	Calcul d'accélération et Mise à jour des positions en GPU	Utilisation des types « builtins » float3 et float4	Utilisation de la mémoire partagée	Utilisation de la <<pinned>> mémoire
160 FPS	250 FPS	450 FPS	900 FPS	1400 FPS

FIGURE 12 – Performances obtenues

Les performances obtenues ne sont pas maximales. En effet, le code peut être encore optimisé en exploitant de plus les ressources de CUDA.