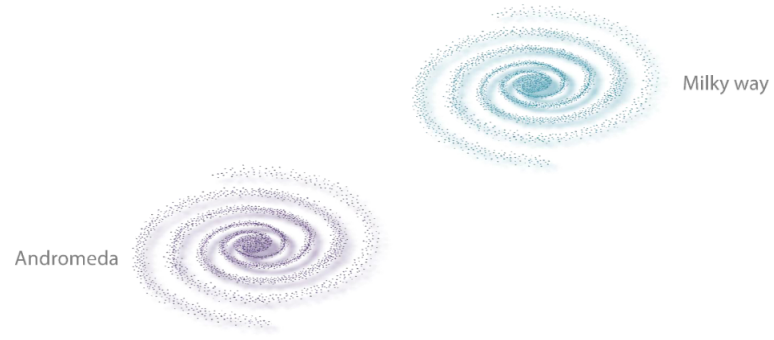
Bordeaux INP
ENSEIRB
MATMECA

Filière électronique

Systèmes Embarqués

HPEC - Galaxeirb



Safae Ouajih
Ghita El Moussi
2020 - 2021

Encadrant : M. J. Crenne

Sommaire

Partie 1

Introduction

1.1 Présentation du projet

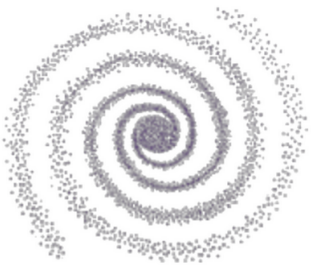
Le but de ce projet est l'implémentation d'une application intensive en mathématique sur un SoC. Il s'agit d'une simulation de collision entre deux galaxies (la Voie Lactée et Andromède). Cette simulation de systèmes à N particules vise à trouver le mouvement futur de chaque particule en prenant en compte les forces agissant sur cette dernière. Chaque particule (corps céleste) est définie par sa position, sa vitesse et sa masse. Le mouvement des particules est influencé par la masse du corps, la vitesse initiale ainsi que la position et la masse des autres corps dans la simulation. L'algorithm de calcul des nouvelles positions est le suivant :

```
1   FOR each step DO
2     FOR each particle DO
3       particle acceleration ← 0
4       FOR each neighbor particle DO
5         particle acceleration ← Add Acceleration ( particle, neighbor particle )
6       END
7     END
8     FOR each particle DO
9       Update Particle Positions ( particle, particle acceleration )
10    END
11    Do Anything Useful ( )
12  END
```

Figure 1.1 – Algorithm de calcul

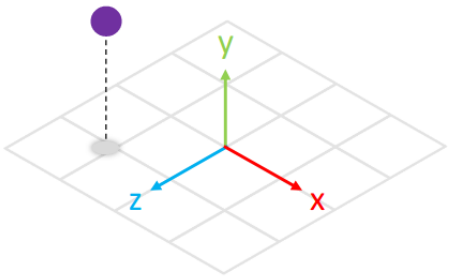
1.2 Modélisation des galaxies

Les deux galaxies simulées sont constituées de particules. Une galaxie est définie par un bulge (8192 particules), un disk (16384 particules) et un halo (16384 particules). Chaque particule possède une position et une vitesse initiales ainsi qu'une masse propre.



1.3 Modélisation des particules de galaxy

Nous considérons toute particule comme un corps à accélération propre nulle, il est subi aux forces extérieures exercées par les autres corps du système. La représentation d'une particule dans l'espace est la suivante :



Les accélérations sont calculées comme suit : $\vec{a_i} = \sum_{j \neq i, j=0}^{n-1} \vec{\Delta_{ij}} * \xi * \frac{1}{d_{ij}^3} * M * m_j$

tel que :

M = 10

ξ = 1 ;

$\vec{\Delta_{ij}} = \vec{p_j} - \vec{p_i}$

$d_{ij} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2}$

Le but est donc d'optimiser le temps de calcul pour avoir une simulation rapide de la collision.

1.4 Matériel utilisé

Nous utilisons la carte de développement NVIDIA Jetson TK1 (Ubuntu 14.04 LTS). L'architecture de processeur graphique Kepler utilise 192 Cœurs CUDA pour des capacités graphiques avancées

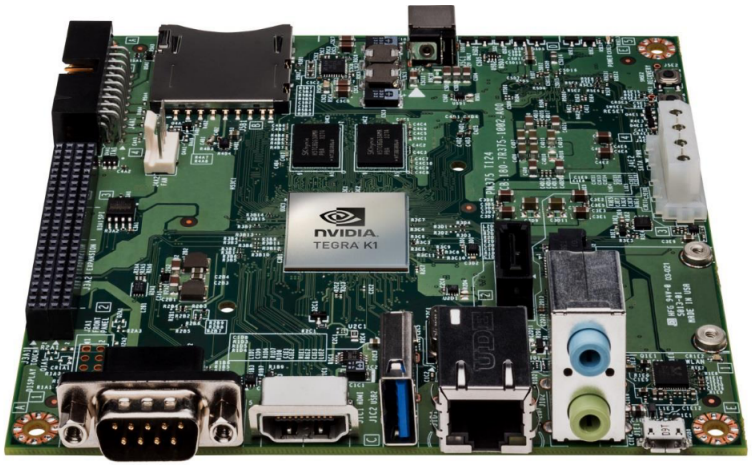


FIGURE 1.2 – Cible : NVIDIA JetsonTK1

Les APIs utilisées dans le projet sont :

— **SDL (Simple Directmedia Layer)** : pour gérer les périphériques (clavier, souris), ainsi que le zoom.

— **OpenMP (Open Multi-Precessing)** : pour faire du parallélisme sur CPU.

— **OpenGL (Open Graphics Library)** : Un ensemble normalisé de fonctions de calcul d'images 2D ou 3D

— **CUDA (Compute Unified Device Architecture)** : pour réaliser des calculs (parallèles) sur le GPU de la carte (NVIDIA).

Partie 2

Première simulation de la collision

2.1 CPU

Nous allons initialement essayer de simuler la collision des deux galaxies en utilisant le CPU de la carte NVIDIA. Le calcul se fait à l'aide de quatre coeurs C-A15 dont chacun possède deux mémoires caches de 32 Ko, un microcontrôleur de 32 bits (architecture ARMv7) et une unité de calcul en virgule flottante.

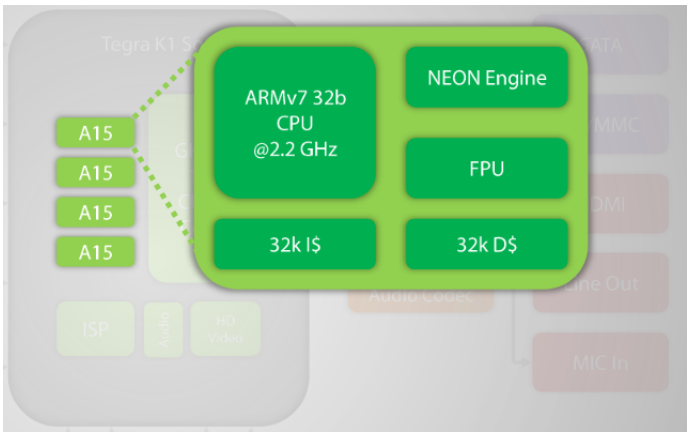


Figure 2.1 – Architecture CPU

2.2 Préparation

Pour simuler la collision des galaxies, nous avons une data base initiale indiquant la position et la vitesse initiale de chaque particule. Les étapes de préparation des données sont :

- Ouvrir le fichier "dubinski.tab"
- Extraire les données dans des variables internes
- Itérer sur les données pour garder 1024 particules

Nous avons utilisé des variables flottantes distinctes pour stocker la masse, la position suivant x, la position suivant y, la position suivant z, la vitesse suivant x,la vitesse suivant y et la vitesse suivant z. 3 variables accélérations de type float sont créées pour calculer les accélérations des particules.

2.3 Calcul

Après avoir préparé les données initiales, nous allons commencer cette étape de calcul qui va permettre de calculer les nouvelles positions de chaque particule dans le système. Les étapes de simulation sont comme suit :

- Mettre les accélérations initiales à 0.
- Afficher la position des particules.
- Calculer les accélérations.
- Calculer les vitesses suivantes.
- Calculer les prochaines positions des particules.
- Afficher la position des particules.

Les équations utilisée pour le calcul des viesses et des positions sont :

vitesse(t)=vitesse(t-1)+accélération(t)

position(t)=vitesse(t)*0.1

2.4 Simulation

Pour afficher les positions des particules nous utilisons l'API OPENGL ;

```
glBegin(GL_POINTS);
for ( j = 0; j < size2; j++) {
        accelx [j]  =0;
        accely [j]  =0;
         accelz [j]  =0;

        glColor3f( 0.0f, 1.0f, 0.0f );
        glVertex3f( posx[j], posy[j], posz[j] );

}
glEnd();
```

FIGURE 2.2 – Affichage avec OpenGl

Dans le fichier opengl/src/main.c, on trouve plusieurs fonctions prédéfinis comme DrawGridXZ qui affiche la grille, ShowAxes qui affiche les axes, et le main où est configurée l'interface graphique.

On atteint une fréquence de **3 FPS**.

2.5 Optimisation

Afin d'optimiser la compilation , nous ajoutons l'option **-O3** (optimisation de compilation de niveau 3) il permet d'optimiser plus la taille du code et le temps d'exécution. Nous avons aussi optimisé le code en éliminant les variables de passage et nous avons optimisé les calculs de manière à ce que les opérations de base (* , /) soient appelées le moins possible.

Aprés optimisation, nous sommes à **36 FPS**

2.6 Utilisation de OpenMP

Cette optimisation est faite à l'aide de la bibliothèque OpenMP. Cette méthode consiste à utiliser au maximum les Threads (4) pour effectuer les traitements comme le calcul d'accélération des particules. Il suffit d'indiquer qu'il faut essayer de faire du parallélisme pour une boucle donnée :

```
#pragma omp parallel for
```

Voici un exemple utilisé :

```
#pragma omp parallel for
for(h=0;h<size2 ; h++){
   if ( h != k ) {
            float mulPosX = posx[h]-posx[k];
           float mulPosY = posy[h]-posy[k];
           float mulPosZ = posz[h]-posz[k];
            float d = sqrtf(mulPosX*mulPosX + mulPosY*mulPosY + mulPosZ*mulPosZ);
           if ( d < 1.0 ) d = 1.0;
            float l = masse[h]* M * (1/(d*d*d));
            accelx [k] += mulPosX*l;
           accely [k] += mulPosY*l;
            accelz [k] += mulPosZ*l;
   }

       }
```

FIGURE 2.3 – Exepmle de parallélisme

Pour indiquer le nombre de Threads utilisé, nous utilisons la fonction ;

```
omp_set_num_threads( 4 );
```

On affiche avec une fréquence de **132 FPS**.
Avec **3 Threads**, nous sommes sur **100 FPS**. Avec **2 Threads**, nous avons **60 FPS**. Le calcul sur CPU reste très limité. Le nombre de Threads est limité à 4 alors que nous sommes en train de simuler 1024 particules. La prochaine étape est l'utilisation de GPU pour pouvoir atteindre une fréquence plus élevée.

Partie 3

Utilisation de CUDA

Dans cette partie, nous allons étudier l'utilisation de GPU pour déplacer la charge de calcul et ainsi libérer le CPU. le coeur NVIDIA Kepler incluant 192 coeurs CUDA :

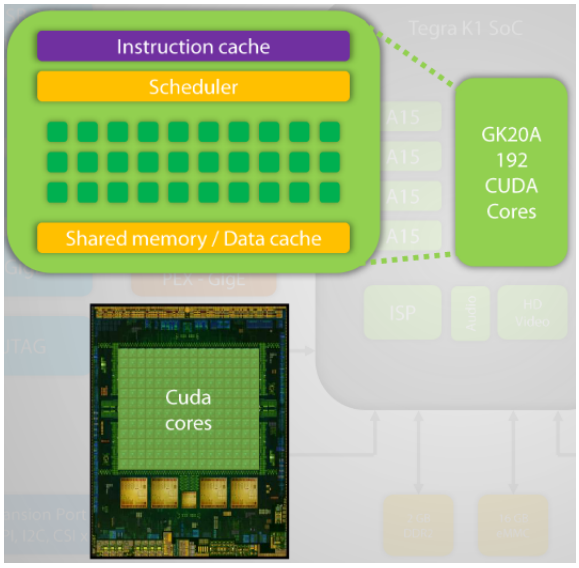


FIGURE 3.1 – Coeurs CUDA

Cette architecture est la plus adaptée pour effectuer les calculs sur notre 1024 particules. Les étapes pour configurer l'utilisation des coeurs CUDA sont :

- Allocation de la mémoire dans le CPU et dans le GPU pour les positions des particules ainsi que leurs vitesses.
- Copie d'élément de la mémoire du CPU vers la mémoire du GPU.
- Effectuer le calcul sur le GPU.
- Copie des résultats de la mémoire du GPU vers la mémoire du CPU pour les afficher.

3.1 Premiers tests avec CUDA

Afin d'utiliser le GPU nous avons suivi les étapes précedente :
- Pour allouer la mémoire dans le GPU, nous avons utilisé la fonction :
CUDA_MALLOC();
- Pour copier à partir du CPU vers le GPU nous avons utilisé la fonction :
CUDA_MEMCPY();
Nous utiliserons la meme fonction pour la copie inverse (GPU -> CPU).

Nous appelons la fonction saxpy() définie dans le fichier "kernel.cu" qui prend en argument le nombre de blocs ainsi que le nombre de Threads à utiliser pour chaque bloc. Une synchronisation est nécessaire pour attendre la fin des calculs avant de passer à l'affichage. nous utilisons la fonction **cudaDeviceSynchronize()** pour la synchronisation. le fichier "kernel.cu" contient la définition de deux fonctions principales. La fonction **saxpy()** qui fait appel à la fonction kernel_acc() en indiquant le nombre de blocs à utiliser ainsi que le nombre de Threads par bloc. Les calculs sont faits dans la fonction kernel_acc(). Nous utilisons la variable :
int i = blockIdx.x * blockDim.x + threadIdx.x ; pour avoir le thread utilisé.
Le calcul de l'accélération est fait pour la particule numéro i, ensuite, la nouvelle position est mise à jour. Nous utilisons 1024 Thread donc i est entre 0 et 1023. Après l'utilisation du GPU, nous sommes à une fréquence plus élevée qui est : **530 FPS**

3.2 Utilisation des registres

Nous avons décidé d'utiliser les registres dans la partie calcul dans GPU vu que le temps d'accès aux registres est beaucoup plus faible. Plusieurs variables sont utilisées pour optimiser le temps d'accès aux variables nécessaires. Nous avons aussi supprimer les flags de debug -G dans le fichier "Makefile". Cette modification permet d'obtenir une fréquence de **600 FPS**.

3.3 Utilisation de la mémoire partagée

Nous allons dans cette partie utiliser la mémoire partagée pour une bonne optimisation du code ; l'accès de mémoire est plus rapide. En effet, les threads du bloc ont accès à une mémoire commune(partagée). L'accès à cette mémoire est plus rapide que l'accès à la mémoire globale. Le but est donc permettre aux Threads d'un bloc d'utiliser une mémoire partagée.Nous avons une taille de 32ko de mémoire partagée.
Pour déclarer une variable qui sera stockée dans la mémoire partagée, nous utilisons le mot clé __shared__ avant sa déclaration. Nous avons décidé de rendre partagées les variables masse et positions des particules. l'accès à ces variables est multiple donc l'utilisation de la mémoire partagée est très utile.

```
__shared__ float m[1024],px[1024],py[1024],pz[1024] ;
```

On atteint une fréquence de **700 FPS**.

3.4 Utilisation des structures

Pour optimiser encore le temps de simulation, nous avons ajouté d'autres optimisation au niveau des variables, Nous avons décidé d'utiliser des types float3 et float4 au lieu de déclarer trois variables. Nous avons décidé d'utiliser des structures pour la représentation des particules ; chaque particule possède trois variables de position, trois vitesses et une masse.

```
5 struct part{
6         float3 pos;
7         float3 vel;
8         float masse;
9 };
```

Ces optimisations nous ont permis d'atteindre **830 FPS**

Partie 4

Conclusion

Le but de ce projet était de simuler la collision des deux galaxies ; La Voie Lactée et Andromède. En fait, nous avons essayé d'optimiser le code de simulation pour augmenter la fréquence des images. Nous avons commencé par une première simulation non optimisée dont nous faisons l'acquisition des données initiales, nous calculons les prochaines positions des particules du système et nous les affichons par la suite. Initialement, la fréquence était : **3 FPS**. Nous avons essayé d'optimiser le code en optimisant l'utilisation des variables ainsi que le mode de compilation. Avec 1024 particules, nous avons réussi à atteindre une fréquence de **36 FPS**. Pour aller plus loin, nous avons décidé d'utiliser le principe du parallélisme avec **OpenMp** ce qui a permis d'atteindre **132 FPS**. Le CPU étant très limité avec seulement quatre threads, nous avons pris la décision de faire les calculs sur le GPU. Avec 192 cœur de calcul, nous avons atteint **530 FPS** et ensuite **600 FPS** en utilisant les registres. L'utilisation de la mémoire partagée était une solution beaucoup plus efficace . En effet, nous avons réussi à atteindre **700 FPS**. Comme dernière optimisation, nous avons changé les types des variables en float3 et nous avons utilisé les structures pour une bonne optimisation. Finalement, nous avons réussi à avoir **830 FPS**. Malheureusement, nous n'avons pas eu le temps d'utiliser la mémoire épinglée "pinned memory" et l'utilisation des blocs à deux dimensions pour pouvoir travailler avec un nombre de particules plus élevé.

A travers ce projet, nous avons découvert plusieurs méthodes d'optimisation de code au niveau de l'utilisation de CPU, GPU et modes de compilation. C'était l'occasion de découvrir OpenGL, OpenMp, SDL et CUDA. Nous avons tant appris sur l'utilisation du CPU et du GPU que sur l'optimisation du code pour atteindre les meilleures performances.