Bordeaux INP – ENSEIRB-MATMECA

Systèmes Électroniques Embarqués

Rapport de projet

Galaxeirb

MI205

HPEC (High Performance Embedded Computing)

Par Martin AUCHER et Kevin PEREZ

20 Novembre 2019

Supervisé par Jérémie CRENNE

Sommaire :

I) Approche du projet :

1. Objectifs du projet :

Le cannibalisme galactique est un phénomène peu rare dans l'univers qui consiste en un grand nombre de collisions de deux galaxies entre elles. La voie lactée, galaxie dans laquelle nous vivons, va subir une collision avec la galaxie la plus proche : Andromède. Dans environ 3 milliard d'année, ces deux galaxies n'en formeront plus qu'une : une galaxie elliptique.

Ce phénomène peut être simulé sur machine en ayant une base de données contenant les différents corps célestes qui composent les deux galaxies étudiés. Dans le cadre de ce projet, nous allons utiliser le fichier **dubinski.tab** qui contient 81920 particules. Chacune de ces particules possède 7 propriétés :

- Position (x, y et z)
- Vitesse (vx, vy, vz)
- Masse

Voici un extrait du fichier **dubinski.tab** :

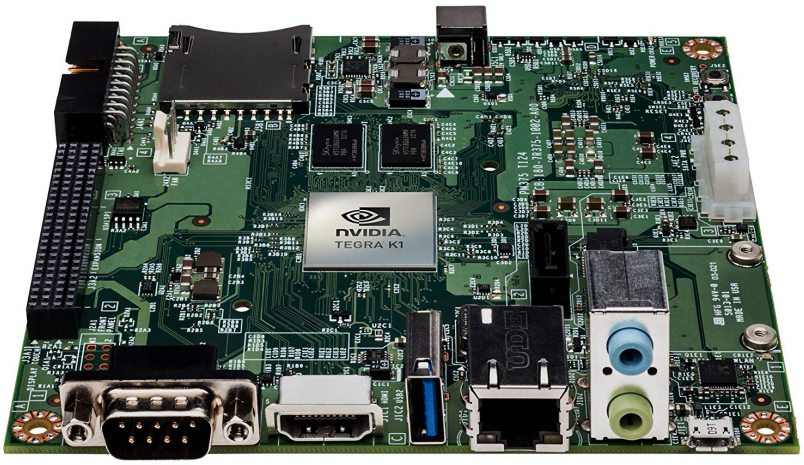
```
4.98914e-05 1.74668 -19.0253 6.79166 -0.420658 0.597319 -0.37661
4.98914e-05 1.70056 -19.0035 6.83187 -0.305732 0.958783 -0.149375
4.98914e-05 1.77301 -19.013 6.80016 -0.226915 0.402915 -0.187241
4.98914e-05 1.81599 -19.0531 6.82844 -0.456958 0.639559 -0.435593
4.98914e-05 1.76612 -19.0548 6.83716 -0.695444 0.931998 -0.759405
4.98914e-05 1.76562 -19.0317 6.84715 -0.483433 0.362284 -0.433775
4.98914e-05 1.69742 -19.0138 6.73435 -0.443397 0.535203 -0.415921
4.98914e-05 1.74097 -19.0844 6.75588 -0.737709 0.845112 -0.827197
4.98914e-05 1.70505 -19.0661 6.82502 -0.572025 1.37572 -0.16096
```

Extrait du fichier dubinski.tab

La simulation N-Body entre Andromède et la Voie Lactée nécessite de réaliser un grand nombre de calcul. Afin d'obtenir une simulation qui apparait le plus fluide possible sur un écran, le code doit être optimisé en terme de performances.

2. La carte de travail :

Le projet Galaxeirb va être compilé et testé sur une carte conçue et fabriquée par NVIDIA : la NVIDIA Jetson TK1.



NVIDIA Jetson TK1

Performances de la carte :

- 192 Coeurs CUDA, 300 gigaflops
- CPU 4 Coeurs
- 2Go de RAM
- Connecteur HDMI
- Connecteur Ethernet avec carte réseau
- Capacité de calcul de 32 bits à virgule flottante

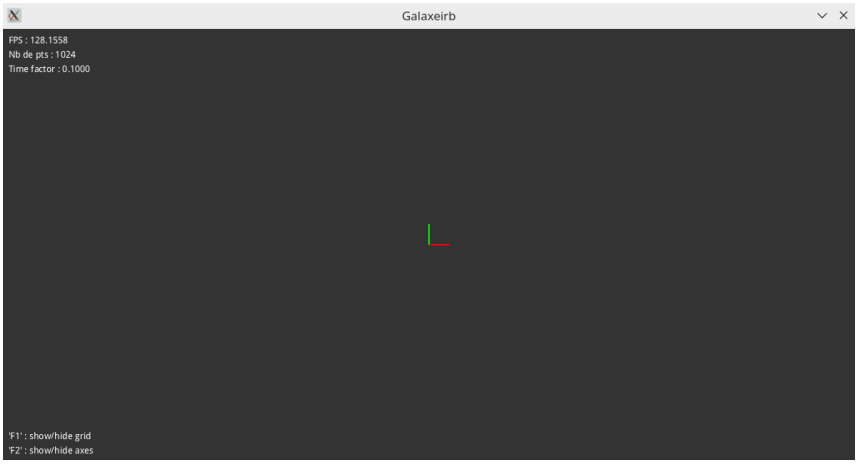
II) Code de base :

1. Création de l'interface :

Pour réaliser ce projet, nous avons d'abord commencé par implémenter une interface à l'aide de SDL. Cette librairie nous permet de créer facilement une fenêtre de type canvas ainsi que de gérer les actions effectuées sur le clavier et la souris. Durant ce projet, le clavier va nous permettre d'exécuter des commandes (avec les touches F1, F2, ECHAP et R). La souris quand à elle va nous permettre de se déplacer autour des deux galaxies qui vont entrer en collisions.

Logo SDL

La création de l'interface se fait grâce à la fonction **`SDL_CreateWindow`** et **`SDL_GL_CreateContext`**. On définit une taille par défaut pour notre fenêtre. C'est sur cette fenêtre que l'on va faire apparaitre les différentes valeurs présentes dans notre bases de donnée : **`dubinski.tab`**.



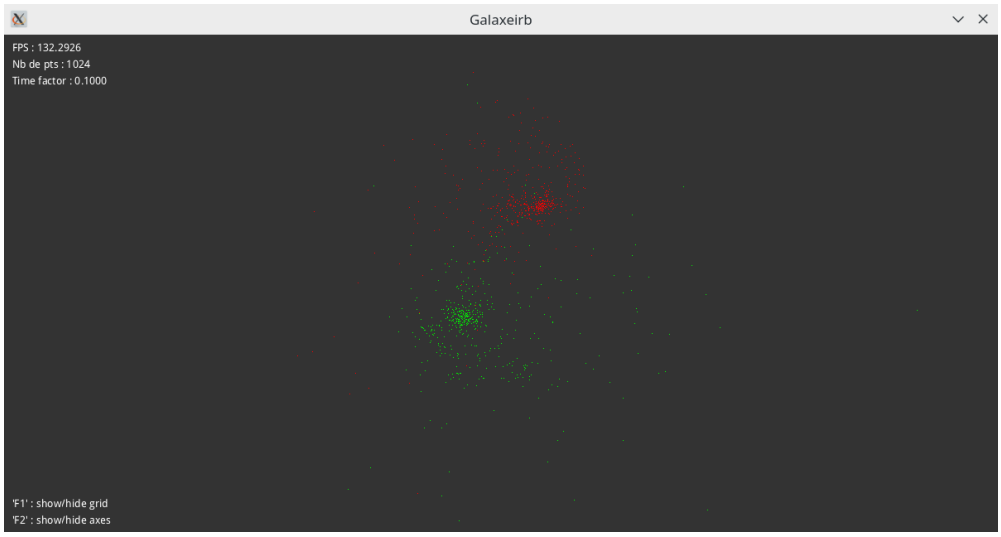
Interface du programme Galaxeirb

2. Affichage des constellations :

Les constellations vont quand à elles être affichée à l'aide de l'API OpenGL (**Open G**raphics **L**ibrary). Cette API (**A**pplication **P**rograming **I**nterface) est compatible avec SDL et est très couramment utilisé pour faciliter la conception d'un environnement 3D. Elle permet de de déclarer la géométrie d'objets sous forme de points, de vecteurs ou encore de polygones. Dans notre cas, elle sera très utile afin d'afficher des points correspondant à un système solaire dans notre fenêtre de travail.

Logo OpenGL

Afin de ne pas manipuler une quantité astronomique de points, on va se concentrer sur l'affichage et l'animation de 1024 points du fichier **`dubinski.tab`** qui en contient 81290.



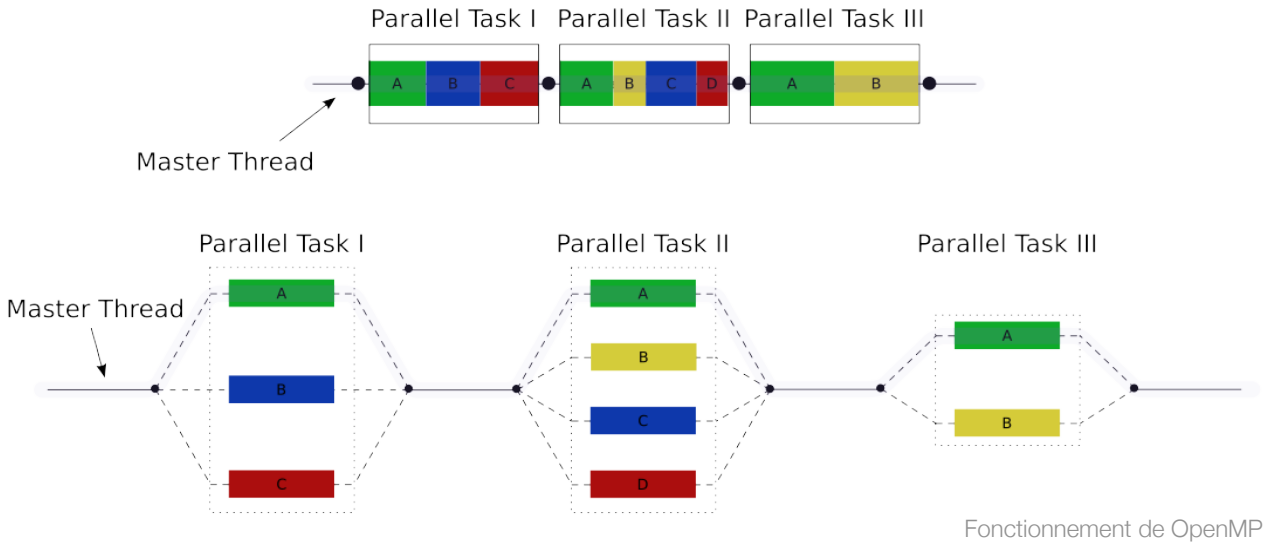
1024 points affichés

II) Optimisation des performances du programme :

1. OpenMP :

OpenMP est une API utilisée pour écrire des application multi-thread. Cela permet principalement de paralléliser les boucles et ainsi réduire drastiquement le temps de calcul en répartissant la charge sur chaque thread alloué. La méthode utilisée pour paralléliser une boucle **`for`** est le **`fork-join`**. Le thread principal lance plusieurs autres threads avec **`fork`** et leurs fait faire une partie des calculs. (1/4 de la charge pour 4 threads par ex). Lorsque tous les threads ont terminé leurs calculs, le thread maitre est chargé de rassembler tout les résultats partiels des calculs pour former le résultat final, c'est le **`join`**. A la fin du rassemblement, le programme peut continuer normalement et utiliser les resultat calculés dans les différents threads.

Voici une représentation visuelle du fonctionnement de OpenMP :



Fonctionnement de OpenMP

L'utilisation de OpenMP passe par l'ajout avant une boucle for à paralléliser de l'instruction suivante :

```
#pragma omp parallel for
```

Dans notre programme, nous utilisons un boucle for qui a un impact sur le temps de calcul très important lors de l'actualisation des paramètres intrinsèque à chaque points que l'on affiche sur notre interface. Pour accélérer le temps de réalisation de l'intégralité de cette boucle, nous utilisons OpenMP.

2. Optimisation du compilateur : O3 :

Par défaut l'objectif du compilateur est de réduire le coût de la compilation du programme, que cela soit en terme de temps ou de ressources utilisées. Les breakpoints permettent de débugger notre programme en vérifiant la valeur d'une variable, en lui assignant une nouvelle valeur, etc.

L'optimisation o3 consiste à ajouter un flag lors de la compilation. Ce dernier va permettre au compilateur d'optimiser le code. Cette optimisation va augmenter le coût de compilation en terme de ressource et de temps alloué. Les breakpoints et le débugge seront ignorés.

Il existe d'autre flag que l'on peut utiliser lors de la compilation comme par exemple :

- **`O0`** : c'est le flag par défaut. Le compilateur a pour objectif de réduire le temps de compilation.
- **`O2`** : Ce flag permet d'optimiser le temps de compilation et la taille du programme généré.
- **`O3`** : Optimisation des performances du programmes. Cependant, le binaire aura une taille plus élevé.

Pour ce projet, nous avons choisis d'utiliser le flag de compilation **`-O3`**, qui nous permet d'augmenter les performances de notre programme cible en raison du nombre important de calculs nécessaire à l'animation de la collision. Le rapidité d'exécution du programme est critique dans ce projet afin d'avoir un rendu fluide.

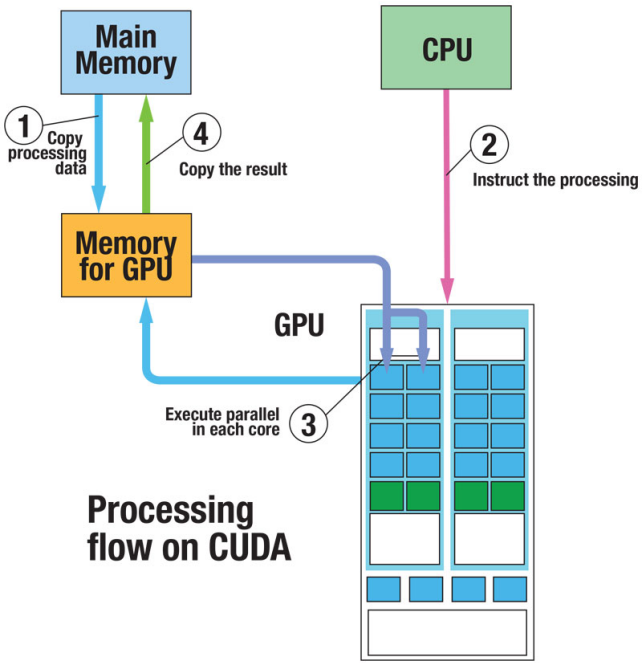
```
14  CFLAGS          = -g -Wall -fopenmp -O3
15  NFLAGS          = -O3 -G -g -arch=compute_32
16  LFLAGS          = -Wall -I. -lm -lGL -lGLEW -lSDL2 -lGLU -lglut -fopenmp -O3
```

Extrait du makefile permettant l'utilisation de O3

Le flag de compilation est ajouté directement dans le makefile permettant la compilation à l'aide de la commande **`make`**.

3. L'optimisation GPU : CUDA :

CUDA (Compute Unified Device Architecture) est une technologie développé par NVIDIA permettant d'effectuer des calculs sur le processeur graphique (GPU) au lieux du CPU. Les GPU ont une fréquence d'horloge plus faible que les CPU, cependant ils disposent de beaucoup plus de coeur de de calculs. L'intérêt de cette technologie est donc de paralléliser les calculs afin d'accélérer le temps du calcul final. La carte NVIDIA TK1 possède 128 coeurs CUDA. Ce qui nous permet donc de paralléliser 128 calculs en même temps, la où un processeur 4 coeurs ne permet d'en paralléliser que 4. Le fonctionnement de cette technologie ne fonctionne que sur des GPU NVIDIA.



Pour implémenter CUDA sur notre programme, il faut suivre les étapes suivantes :

1. **`CUDA_MALLOC`** : Qui permet d'allouer un espace mémoire sur la RAM du GPU.
2. **`CUDA_MEMCPY`** : Qui permet de copier des données dont l'adresse est connue, vers la mémoire RAM du GPU.
3. On exécute l'instruction de calcul sur le GPU à l'aide des adresses mémoires GPU connus au préalables.
4. **`CUDA_MEMCPY`** : Qui permet de copier les données de la mémoire GPU vers la mémoire RAP CPU.

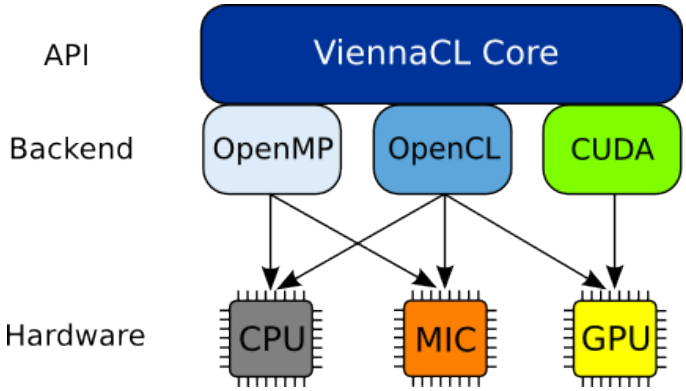
Les fonctions qui permettent d'interagir avec les coeurs CUDA du GPU NVIDIA sont définis dans la librairie CUDA que l'on intègre à notre programme (CUDA.h).

III) Améliorations possibles :

Suite à nos recherches effectuées durant ce projet, on a pu envisager diverses améliorations qu'il est possible d'essayer afin de tester le programme et constater si il y'a ou non une augmentation du nombre de FPS de la simulation.

La première amélioration concerne l'optimisation du compilateur en passant de -O3 à -Ofast. L'optimisation Ofast est identique à l'optimisation O3, mais elle ajoute en plus une accélération de certains calculs mathématiques. Il est donc intéressant pour nous d'évaluer les différents gains de FPS en passant de O3 à Ofast.

La deuxième amélioration concernant l'utilisation de OpenCL à la place de CUDA.



Comme on peut le voir sur ce schéma, CUDA n'est disponible que pour des GPU, qui plus est, fabriqué par NVIDIA. OpenCL (**Open C**omputing **L**anguage) également une API qui est disponible pour plusieurs hardware. On peut donc utiliser OpenCL pour des GPU qui ne sont pas des processeur NVIDIA. Dans notre cas, il serait intéressant de noter la différence de fonctionnement ainsi que de performance de ces deux technologies.

IV) Conclusion :

Ce projet nous a permis d'apprendre à optimiser un code grâce à différentes fonctions, options de compilations ainsi que de librairies (ou API). Nous avons pu tester notre code sur une carte fabriqué par NVIDIA, embarquant notamment un GPU. Ce dernier nous a permis d'utiliser deux processeurs possédant des vitesses d'horloges différentes et un nombre de coeur différent. La première approche du projet nous a permis de coder ce programme sur 1024 points sans piste d'optimisation. Nous avons lancé notre programme et nous avons constaté rapidement que sans optimisation de ce dernier, il serait impossible de le faire fonctionner sur notre carte NVIDIA. Suite à ce constat, nous avons pris conscience de l'importance de l'optimisation dans nos codes. Ce projet nous a permis d'accroître nos compétences en C pour l'embarqué.