M2IIA : PROJET TUTEURÉ

Université Bourgogne Europe
UFR Sciences et Techniques, Département I.E.M.
Master Informatique - parcours IIA Image et Intelligence Artificielle

Rapport de projet tuteuré

Auteur/Autrice
Nathan Jacquinet

Génération de fractales sur GPU

Christian Gentil

2024-25

Table des matières

1 Introduction 1
2 Préliminaire 1
3 Première approche - Juin 2024 1
3.1 Modélisation par arbre 1
3.2 Encodage des feuilles 2
3.3 Décodage des feuilles 3
3.4 Premier résultat 4
4 États des outils disponible 4
4.1 API programmation GPGPU 5
4.2 API graphique 5
4.3 Choix des outils 6
4.4 Pipeline graphique 6
5 Automate 8
5.1 Encodage des feuilles 9
5.2 Décodage des feuilles 10
5.3 Validation 10
6 Implémentation sur GPU 13
6.1 Formatage de l'automate pour le GPU 13
6.2 Décodage des feuilles sur GPU 14
7 Résultats 15
7.1 Optimisation 17
8 Conclusion 19
8.1 Amélioration possible 20
8.2 Liens annexes 20

Résumé La génération de fractales est plutôt simple, surtout grâce à l'informatique. Nous pouvons créer des structures complexes à l'aide de règles très simples. Le problème est que ces méthodes manquent de flexibilité et ont un domaine d'usage restreint. Générer des fractales plus complexe, avec des contraintes, ou des points de contrôle complexifie énormément le processus. Tant bien qu'algorithmiquement, qu'en temps de traitement, les rendant très peu exploitables pour de l'édition en temps réel. Ce projet de fin d'études, proposé par Christian Gentil, cherche à explorer une nouvelle façon de générer celles-ci en utilisant un GPU, une unité de traitement massivement parallèle, dans un but d'accélérer la génération et la visualisation de ces fractales.

Mots-clés Informatique graphique, GPGPU, Fractales, Calcul massivement parallèle, Visualisation en temps réel, Modélisation procédurale, Théorie des graphes

1 Introduction

Ce projet tuteuré de Master 2 Image et Intelligence Artificielle a débuté en novembre, et se termine en mars. Mon tuteur est M. Christian Gentil, membre du laboratoire d'Informatique de Bourgogne, LIB, dans la spécification Modélisation Géométrique. Nous avons travaillé dans la salle des doctorants, et effectué des rendez-vous ainsi que des petits rapports hebdomadaires. Il a pu m'aider grâce à son expertise dans son domaine principal qui est les fractales.

Le travail recherché durant ce projet tuteuré est d'enrichir la méthode explorée en juin 2024 3 qui portait sur la génération de fractales en utilisant un IFS rigide 3. Pour améliorer la flexibilité de la génération de ces fractales, il est plus intéressant de modéliser des automates à états, de toutes les formes possibles : cyclique, multipoint d'entrée, non-uniforme, etc. Chaque état possède des transformations, en quantité équivalente (uniforme), ou non (non-uniforme) des autres états. Le code issu de ce projet tuteuré est disponible sur GitHub.

2 Préliminaire

Mon travail durant ce projet de fin d'études est le prolongement de mon travail fait en juin 2024, au Laboratoire d'Informatique de Bourgogne (LIB). Pour un court stage d'un mois après l'année de Master 1. Durant ce temps, j'ai travaillé sur une méthode permettant de décharger entièrement la génération d'un IFS simple sur le GPU. Ce travail est toujours disponible sur mon répertoire GitHub.

3 Première approche - Juin 2024

3.1 Modélisation par arbre

Nous allons revoir la première approche explorée. La méthode utilisée pour générer des fractales est un IFS. Une façon plus logique de visualiser ces IFS est de les modéliser sous la forme d'un arbre comme montré dans 1. Où chaque branche représente une transformation appliquée à un nœud. Chaque nœud est une ancienne feuille de l'itération d'avant.

Chaque chemin unique menant à une feuille forme un morceau de l'attracteur fractale. En les combinant, on forme la fractale finale.

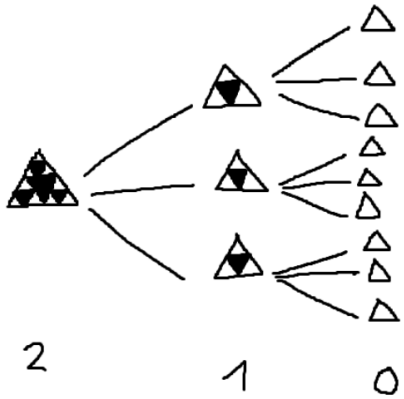


FIGURE 1 – Modélisation de l'arbre représentant l'IFS

On remarque que cet arbre est récursif : l'ensemble résultant pour l'itération i, a besoin du résultat de l'itération *i-1*. (Pour $i \mathrel{!}= 0$) Mais en prenant l'arbre dans l'autre sens, c'est-à-dire en partant des feuilles, et en remontant jusqu'à la racine, chaque feuille produira un morceau de la fractale finale.

On peut remarquer beaucoup de redondance dans cet arbre. Ce qui pourrait être sujet à une optimisation. Une approche a été explorée durant le travail de juin 2024, en utilisant la mémoïsation sur CPU.

On peut aussi voir que chaque feuille possède un chemin unique et indépendant, ce qui est une propriété importante pour la parallélisation massive. Et comme chaque feuille de l'arbre est la structure géométrique initiale, on pourrait envoyer une seule et unique fois la structure originale. Il suffirait alors de calculer parallèlement chaque chemin unique, et de les appliquer à l'objet original. Malgré une redondance des calculs, les GPUs sont très adaptés pour ce type de calculs.

3.2 Encodage des feuilles

Maintenant, que nous avons une modélisation qui semble compatible avec un traitement massivement parallèle, le travail s'est précisé sur la façon de modéliser ces chemins unique. Une approche, qui sera gardée pour la suite du projet de fin d'études, et d'encoder chacune de ces branches par un codage arithmétique. L'idée est d'encoder chaque chemin par une valeur flottante entre 0 et 1. Le nombre de valeurs encodées est égal au nombre de feuilles, et associé à chacune d'elles, comme dans la figure 2. Ensuite, chaque feuille sera décodée itérativement de manière parallèle. Une fois le chemin entier décodé, il sera appliqué à l'objet original, puis affiché, ce qui formera la fractale finale.

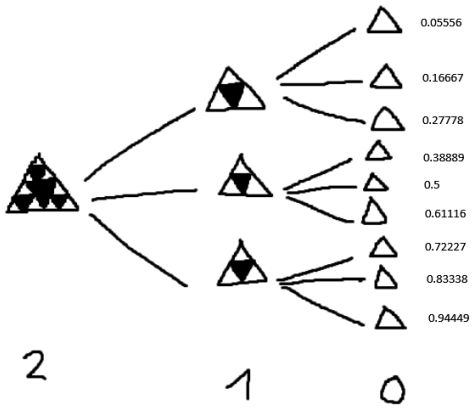


FIGURE 2 – Modélisation de l'arbre représentant l'IFS avec codage arithmétique

3.3 Décodage des feuilles

Pour l'IFS de 2, nous avons trois transformations. Ainsi, nous découpons notre segment unitaire en trois sous-segments uniforme pour les trois transformations, respectivement [0 ; 0.3333].

Pour encoder la première transformation [0.3334 ; 0.6666] pour la deuxième, et [0.6667 ; 1.0] pour la troisième.

Nous partons alors des neuf valeurs encodée, comme montré dans 2 sur chacune des feuilles. Une note importante est qu'il faut connaître en avance le nombre de feuilles produites par notre IFS, à l'itération voulue.

On calcule alors dans quel intervalle la valeur se situe, puis on accumule la transformation associée. Ensuite, une étape d'étirement de valeur est faite.

Voici le pseudo-code du décodeur. Il est modélisé de façon séquentielle pour plus de clarté.

Algorithm 1 Algorithme décodeur d'IFS

```
1: Entrée : Ensemble des valeurs encodées E, Nombre d'itération iteration
2: Sortie : Ensemble des transformations accumulées S
3: function DECODE(E)
4:     for code dans E do
5:         accu ← Identité                          ▷ Initialiser l'accumulateur
6:         for i = 1 to iteration do
7:             transfoID ← TransformationAssociee(code)
8:             accu ← accu × transfoID
9:             code ← ReMappage(transfoID)          ▷ Re-mappage du code
10:        end for
11:        S ← S ∪ {accu}                           ▷ On ajoute l'accumulateur calculé
12:    end for
13: end function
```

Cet IFS est simple, et uniforme, dans le sens que pour chaque itération, il y a la même distribution des branches. Plus précisément, à chaque nœud, il y a toujours le même nombre

de branches sortantes, ainsi la distribution de l'encodage arithmétique est uniforme, ce qui permet de calculer le codage directement sur GPU. En effet, l'ensemble du codage de l'arbre, est simplement $E = \left\{ \frac{i}{N} - \frac{1}{2N} \mid i \in \mathbb{N},\ 0 \leq i < N \right\}$

3.4 Premier résultat

Ce décodage a été implémenté en utilisant l'API Vulkan pour le rendu et l'exécution des shaders.

TABLE 1 – Comparaison des algorithmes en termes de temps d'exécution et d'empreinte mémoire, tiré du premier rapport

Nombre d'itérations	**CPU**		**GPU**		**Speed up**	
	Temps	**Mémoire**	**Temps**	**Mémoire**	**Temps**	**Mémoire**
3	76800 ns	972 B	< 0.1 ms	0 B	N/A	Inf
6	2,08 ms	26,24 kB	< 0.1 ms	0 B	N/A	Inf
9	19.27 ms	708,59 kB	0.2 ms	0 B	96 350	Inf
12	521,90 ms	19,14 MB	0.6 ms	0 B	870	Inf
15	13,86 s	516,56 MB	16.8 ms	0 B	825	Inf
16	42,51 s	1,55 Go	50.2 ms	0 B	846	Inf
17	N/A	N/A	151.4 ms	0 B	N/A	Inf
18	N/A	N/A	454.6 ms	0 B	N/A	Inf
19	N/A	N/A	1.367 s	0 B	N/A	Inf
20	N/A	N/A	4.116 s	0 B	N/A	Inf

Ce tableau compare un algorithme générant le modèle géométrique final de la fractale exécuté sur CPU, avec un algorithme décodant l'arbre directement sur GPU juste avant l'affichage. Dans la colonne CPU, il manque des valeurs, car elle dépassait la limite d'une allocation mémoire contiguë possible dans la VRAM du GPU.

Il est ensuite décodé avant l'affichage de manière massivement parallèle, d'où l'accélération de 800 fois.

3.4.1 Première conclusion

Cette approche en utilisant un encodage, même si implicite, puis de décoder cet arbre directement sur GPU est prometteur. Dans ce travail préliminaire, le cas d'application est simple, mais cela a été assez prometteur pour s'étendre à mon sujet de projet tuteuré.

4 États des outils disponible

Dans un premier temps, il m'a été demandé de prendre le temps d'énumérer les différents outils disponibles en informatique pour répondre à la problématique : on veut accélérer la génération et visualisation des fractales en utilisant un GPU. De sorte à sélectionner ceux qui semblent les plus adéquats.

Pour travailler avec des GPUs, nous avons à notre disposition différents outils permettant de générer, compiler et exécuter du code sur GPU, comme Cuda (Exclusif à NVidia), ROCm (exclusif à AMD), et d'autre outils fonctionnant partout comme OpenCL, SyCL. Nous pouvons également utiliser des API graphiques comme OpenGL, Vulkan, Direct3D, qui nous exposent les GPUs à travers une abstraction appelée pipeline graphique, mais aussi un accès plus bas niveau à l'aide de nuanciers appelé Compute Shader.

4.1 API programmation GPGPU

Les API de programmation GPGPU comme cité dans l'introduction de cette partie nous permettent d'écrire et d'exécuter facilement du code sur GPU. Ils sont principalement axés pour la conception de programmes de calculs hautement parallèle et ne sont donc pas principalement fait pour faire du rendu, bien que nous pouvons toujours en faire. Pour la plupart. Ils exposent également des mécanismes d'interopérabilité avec des API de rendu, permettant d'utiliser un buffer stockant le résultat d'un programme écrit avec SyCL, dans une pipeline graphique de Vulkan par exemple.

Utiliser une de ces API reviendrait à exécuter la génération de la géométrie, puis d'utiliser le résultat pour un affichage en temps réel en utilisant une API graphique.

4.1.1 Cuda et ROCm

Ce sont des APIs permettant d'écrire et d'exécuter du code sur des GPUs strictement réservés aux constructeurs respectifs NVidia et AMD. Bien qu'il existe des couches permettant de convertir du code de l'un vers l'autre, ce n'est pas optimal et ces principes sont plus des solutions de secours pour éviter de réécrire un programme. Cependant, ils ont l'avantage de pouvoir générer un programme plus facilement optimisé pour l'architecture des GPUs nativement supportés.

4.1.2 OpenCL et SyCL

OpenCL est une API permettant d'écrire et de compiler des programmes pouvant s'exécuter nativement sur des GPUs de différents constructeurs. SyCL est une API qui est construite au-dessus d'OpenCL, qui permet de simplifier l'écriture de programmes et permet de faciliter l'interopérabilité sur des systèmes multi-GPU hétérogène.

4.2 API graphique

Les API graphiques permettent d'écrire des programmes principalement pour le rendu (en temps réel ou non), en utilisant une abstraction du GPU sous forme de graphique pipeline. Cependant, nous pouvons également avoir un accès bas niveau grâce aux nuanciers de calculs.

Direct3D (utilisé sous DirectX9, 11, 12) est une API graphique fonctionnant exclusivement sur Windows. OpenGL ainsi que Vulkan fonctionnent sur n'importe quel GPU compatible, sur n'importe quel système d'exploitation compatible. (Cf liste périphérique conforme) Ils sont deux APIs conçu principalement pour faire du rendu. Mais exposent également la possibilité

d'écrire des programmes similairement aux API GPGPU, à l'aide de Compute Shaders.

4.3 Choix des outils

Le but de ce projet est de trouver une nouvelle approche pour générer et afficher des fractales. Utiliser une API graphique est essentiel pour faire l'affichage des géométries sur une fenêtre. Malgré cela, nous avons accès à une certaine flexibilité avec l'option des nuanciers de calculs que ces API nous exposent. Il semble alors naturel de se pencher sur l'utilisation d'un d'entre eux, même si l'utilisation des Compute Shader peut être un peu plus longue à mettre en place qu'un programme OpenCL. Mais ce coût en travail sera largement récompensé, car on travaillera sur une seule et unique API, et on n'aura pas de mécanisme d'interopérabilité à gérer, sans oublier les potentiels formats de donné différents entre les différentes API.

OpenGL est très repandu et plutôt simple d'utilisation et de rapide à mettre en place. Tandis que Vulkan est une API de bas niveau, elle donne un contrôle total aux périphériques compatibles. De plus, elle introduit tout un système de synchronisation explicite (à la charge du développeur), ce qui permet d'écrire explicitement et de manière très fine les règles de synchronisation. Ce qui est très intéressant lorsque nous utilisons par exemple des nuanciers de calculs en plus de le pipeline graphique.

J'ai décidé d'utiliser OpenGL, même si je sacrifie le contrôle des synchronisations fine, cela sera suffisant pour développer et avoir un résultat pendant ce projet tuteuré.

4.4 Pipeline graphique

Comme mentionné avant, une API graphique comme OpenGL nous expose le GPU sous une abstraction pour faire du rendu, appelé pipeline graphique. Cette abstraction est un pipeline globalement fixe, mais nous avons la possibilité de programmer les étapes les plus importantes.

Nous allons discuter des étapes de ce pipeline qui serait les plus appropriés à utiliser.

4.4.1 Pipeline Graphique

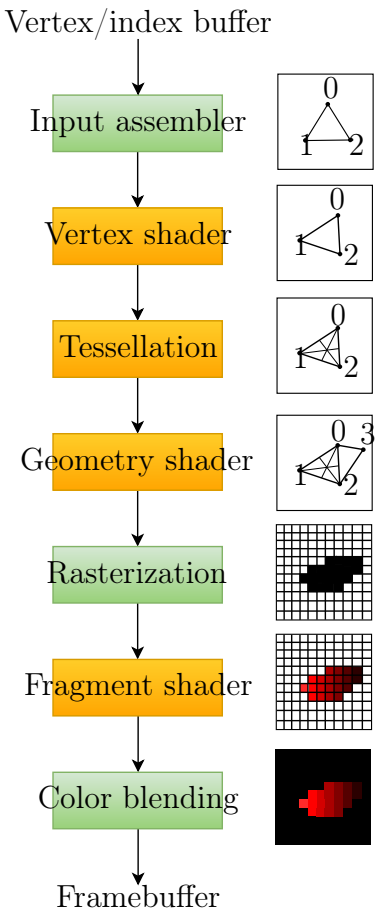


FIGURE 3 – Schéma de la pipeline graphique, tiré du site Vulkan-Tutorial

Dans la figure 3, en vert, ce sont les étapes qui ne sont pas programmables. (implicitement géré par le driver/API graphique) Et en orange, les étapes que le développeur peut programmer à l'aide de nuanciers (shaders)

4.4.2 Vertex Shader

Le vertex shader est l'étape du pipeline où nous pouvons appliquer des transformations pour chaque point regroupé selon ce qui a été défini dans l'Input Shader. (Point, lines, triangle) Cette étape est exécutée en parallèle pour chacune des primitives. C'est cette étape qu'on transforme les points de l'objet géométrique dans l'espace de la fenêtre.

4.4.3 Tesselation

Cette étape de le pipeline permet de créer de la tesselation sur la primitive. Cette étape permet de décider le type tesselation et le type de transformation à appliquer sur le patch. On peut décider de tesseler uniformément ou pas, un patch. Le problème, c'est que l'on manque de contrôle. On ne peut que tesseler de manière fine.

4.4.4 Geometry Shader

Cette étape permet de rajouter des primitives à la primitive traité. Étant donné que les propriétés de la primitive originale seront partagé, il semble être compliqué d'appliquer la génération de fractales à cette étape.

4.4.5 Fragment Shader

Elle est souvent utilisée pour générer des figures fractales comme l'ensemble de Mandelbrot ou un ensemble de Julia. Cette étape du pipeline utilise principalement des textures, qui auraient pu être générées préalablement par différents moyens.

4.4.6 Compute Shader

Une dernière possibilité est celle des shader de calculs (compute shader), elle n'apparaît pas dans le pipeline graphique, car c'est l'abstraction la plus primitive du GPU qui est exposé par ces API graphique. Ces shaders nous permettent de faire du calcul plus flexible, mais les données et le workflow globale de l'exécution est à la charge du développeur.

4.4.7 Choix final

J'ai donc choisi d'utiliser OpenGL, un pipeline graphique pour afficher le résultat, donc une implémentation simple. Mais la phase de décodage de l'automate sera exécutée avec un compute shader, et sera directement utilisé pour le rendu.

Une note important est qu'il existe depuis peu de temps les mesh shaders, qui sont une nouvelle façon de gérer la première partie géométrique du pipeline graphique. On en reparlera plus loin dans le rapport, mais il semble possible d'en faire quelque chose d'intéressant, malheureusement l'équipement dans les ordinateurs du laboratoire n'est pas compatible avec ceci, car trop ancien.

5 Automate

Le premier travail effectué est de développer une interface pour pouvoir modéliser des automates.

Trois classes ont été développées, Automaton, State et Transition. Une Transition est représentée par une transformation, ainsi que l'état suivant. Un State (état) représente un état de l'automate, et est composé d'un ensemble de Transition. Et enfin, un Automaton est composé d'un ensemble de State.

Les parties intéressantes dont nous allons discuter sont dans la classe Automaton. Avant de passer l'exécution sur GPU, il est largement plus pratique de développer si possible notre implémentation sur CPU, car nous avons un accès à des débugueurs. Les algorithmes ont été développés sur CPU, puis validés avec une méthodes que nous verrons juste après.

5.1 Encodage des feuilles

Pour encoder un automate, la méthode utilisée est de traverser celui-ci un nombre d'itération maximale, tout en gardant entre chacune d'elle l'état courant de chaque feuilles. La méthode est appelée *encode*.

Pour représenter un intervalle, il y a deux façons de faire, soit nous les modélisons avec la borne inférieur et supérieur, soit on la modélise avec une valeur, représentant le centre de l'intervalle, ainsi qu'avec une taille.

Ainsi, l'intervalle [0.25 ; 0.75] sera représenté par la structure 0.5 ; 0.5. La première valeur est la valeur centrée de l'intervalle, et la deuxième est sa longueur, appelée son amplitude dans le code. C'est cette approche que j'ai choisie, car cela permet de plus facilement travailler avec les intervalles. De plus, il n'y a pas besoin de calculer le centre des intervalles pour l'encodage final, puisque cette information est déjà contenu dans la structure.

Le fonctionnement est assez simple, on parcourt toutes les feuilles à l'état i, puis pour chacune des transitions, on modifie le codage de la feuille en centrant l'intervalle représentant la transition à prendre, puis on sauvegarde cette nouvelle feuille dans un second buffer, on itère sur chaque feuille.

Algorithm 2 Algorithme encodeur d'IFS

Entrée : Automate A, Nombre d'itération *iteration*
Sortie : Ensemble des codages de chaque feuilles C
function ENCODE(E)
    $feuilleCalculee \leftarrow$ Une feuille initiale 0, 0.5, 1.0 ▷ Première feuille à l'état 0 centré sur l'intervalle [0 ; 1.0]
    for i allant de 0 à *iteration* **do**
        $feuilleSuivante \leftarrow$ vide
        for chaque $feuille$ dans $feuilleCalculee$ **do**
            for *translation* **dans** $translationsEtat($ **do**
                $newCode \leftarrow$ calculNouveauCode($feuille$)
                $newAmplitude \leftarrow$ calculNouvelAmmplitude($feuille$)
                $nouvelleFeuille \leftarrow \{\text{translation.nextState}, newCode, newAmplitude\}$
                $feuilleSuivante \leftarrow feuilleSuivante \cup \{nouvelleFeuille\}$
            end for
            $feuilleCalculee \leftarrow feuilleSuivante$
        end for
    end for
    $C \leftarrow$ extraireCode($feuilleCalculee$)
end function

L'algorithme d'encodage est plutôt simple 2, et est nécessaire pour gérer tous les types d'automate, en particulier ceux qui ne sont pas uniformes. En effet, l'implémentation proposée en 3 peut être vu comme un automate, à un seul état, et donc uniforme. L'uniformité permet

alors de calculer directement la distribution de l'encodage, car le pas entre chaque codage est uniforme. On pourrait aussi utiliser cet ancien encodage avec des automates à multi-états uniforme. Tandis qu'avec des automates à plusieurs états non-uniforme, et cyclique, la distribution de l'encodage est plus difficilement prévisible, et j'ai donc opté pour cette exécution simple.

5.2 Décodage des feuilles

L'algorithme de décodage dans *Automaton* est le même que celui utilisé en juin 2024 1.

5.3 Validation

Pour valider le principe, j'ai implémenté une fonction qui permet de traverser l'automate en générant toutes les transformations finales pour chaque branche, appelé *compute*. En effet, ce résultat sera utilisé pour valider le système en comparant les matrices générées par le système encodeur-décodeur, avec celles générées par la fonction *compute*.

5.3.1 Résultat

Voici l'automate de test :

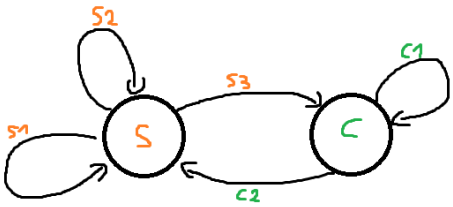


FIGURE 4 – Automate de test : cyclique et non uniforme.

L'automate de 4 commence à l'état S. La génération et la comparaison ont été faite jusqu'à l'itération 14 :

```
i=0 OK CodeSize:1
i=1 OK CodeSize:4
i=2 OK CodeSize:14
i=3 OK CodeSize:48
i=4 OK CodeSize:164
i=5 OK CodeSize:560
i=6 OK CodeSize:1912
i=7 OK CodeSize:6528
i=8 OK CodeSize:22288
i=9 OK CodeSize:76096
i=10 OK CodeSize:259808
i=11 OK CodeSize:887040
i=12 NONOK CodeSize:3028544
i=13 NONOK CodeSize:10340096
i=14 NONOK CodeSize:35303296
i=15 NONOK CodeSize:120532992
```

FIGURE 5 – Résultat du test de validation.

5.3.2 Erreur de précision

Nous pouvons remarquer un problème dans les résultats de 5, à partir de l'itération 12 pour cet automate, nous n'avons pas les mêmes résultats. Les premiers soupçons de la cause sont les valeurs flottantes (ici 32 bits), en effet, cette représentation peut poser problème lorsque l'on a besoin de précision.

Revenons à l'implémentation faite en juin 2024, qui était comme un automate uniforme. Dans ce cas, nous avons une distribution uniforme de l'encodage, c'est même la répartition qui permet de maximiser la répartition sur le segment $[0\,;\,1]$. Avec un automate, non-uniforme, la réparation est bien différents. Il y a des zones du segment avec une densité élevée de codage, et par manque de précision des flottantes, certaines valeurs flottante sont tronquées par les FPUs, et nous avons alors potentiellement les mêmes valeurs. Ce qui compresse alors avec perte les feuilles de l'automate.

5.3.3 Solution

Une solution simple serait d'utiliser un double, qui est un flottant en 64 bits. Les fonctions ont été spécialisées par des templates, de sorte à laisser l'utilisateur choisir la précision de l'encodage.

```
i=0 OK CodeSize:1
i=1 OK CodeSize:4
i=2 OK CodeSize:14
i=3 OK CodeSize:48
i=4 OK CodeSize:164
i=5 OK CodeSize:560
i=6 OK CodeSize:1912
i=7 OK CodeSize:6528
i=8 OK CodeSize:22288
i=9 OK CodeSize:76096
i=10 OK CodeSize:259808
i=11 OK CodeSize:887040
i=12 OK CodeSize:3028544
i=13 OK CodeSize:10340096
i=14 OK CodeSize:35303296
i=15 OK CodeSize:120532992
```

FIGURE 6 – Résultat du test de validation avec float 64 bits.

En utilisant des flottant de 64 bits, le test est valide bien plus loin dans les itérations. Je n'ai pas pu aller plus loin par manque de mémoire vive, car il faut les matrices test et type pour les comparer, ce qui prend énormément de mémoire, mais la théorie est validée.

Il est dur de prédire quand il faut utiliser des flottant de plus grande précision, mais une chose est sûr, ce n'est pas forcément la quantité de feuilles encodées, mais surtout la réparation de ces codes. Par curiosité, j'ai voulu analyser cette répartition :

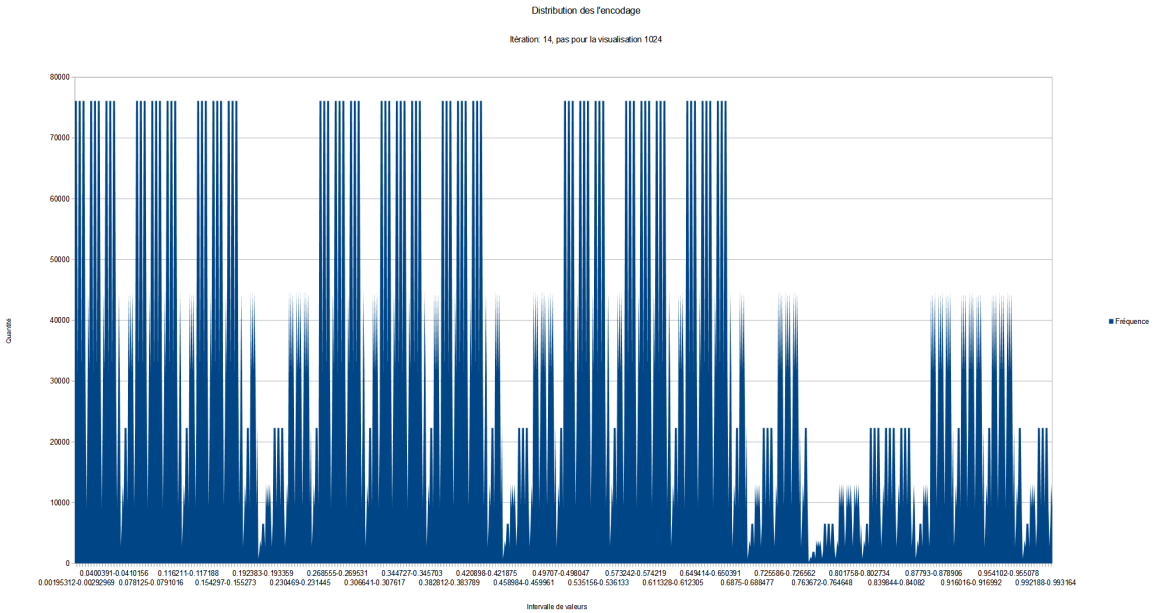
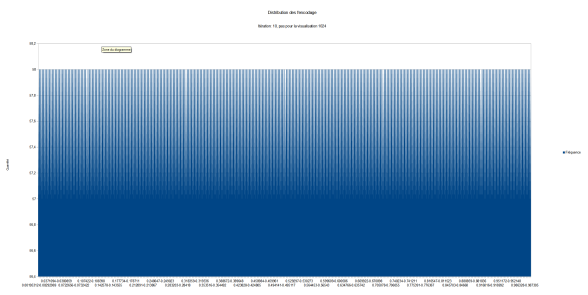


FIGURE 7 – Histogramme de la répartition de l'encodage sur l'automate 4, à l'itération 14. Histogramme de 1024 colonnes

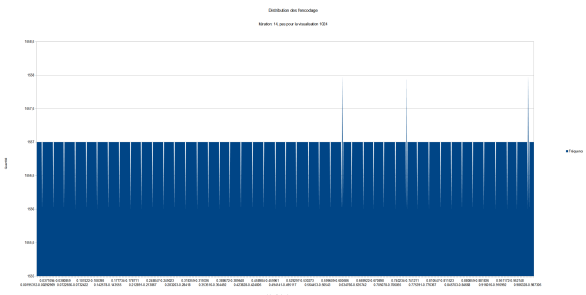
Dans cet histogramme 7, on peut observer qu'il y a de grandes disparités dans la distribution des valeurs encodées.

D'ailleurs, on peut observer une distribution fractale, on remarque les trois transformations de l'état S de 4 qui renvoie sur S. On voit alors que la partie la plus à droite (celle qui encode en premier l'étape C) possède moins de codes, car il y a que deux transformations, et on retrouve le pattern des trois pics ce secteur.

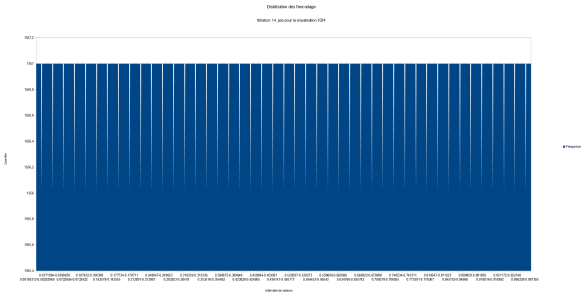
Revenons à notre problème de distribution, pour comparer, voici la répartition 8 des valeurs codée, sur un automate **uniforme** à trois transitions, aux itérations 10, et 13. Sachant qu'à 10 l'encodage-décodage est équivalent à la vérité, et à 13 ce n'est plus le cas pour des flottant 32 bits.



(a) Histogramme de la répartition de l'encodage un automate uniforme, à l'itération 10. Histogramme de 1024 colonnes



(b) Histogramme de la répartition de l'encodage un automate uniforme, à l'itération 13. Histogramme de 1024 colonnes



(c) Histogramme de la répartition de l'encodage un automate uniforme, à l'itération 13 en utilisant des **doubles**. Histogramme de 1024 colonnes

FIGURE 8 – Comparaison et visualisation des problèmes d'arrondis

Dans la figure 8, qui possède une répartition uniforme, comme dans 8a qui est valide, cependant ce n'est pas le cas pour 8b, qui à l'itération 13 n'est plus valide, et nous pouvons observer un problème de répartition dans les valeurs flottantes. Et enfin, dans l'image 8c, qui est le même automate, mais en utilisant cette fois-ci des flottant 64 bits, la répartition est de nouveau uniforme.

6 Implémentation sur GPU

Maintenant que le système encodeur-décodeur est validé, tout en ayant conscient des limitations, une implémentation sur GPU est envisageable.

6.1 Formatage de l'automate pour le GPU

Un travail est nécessaire pour transférer l'automate, et particulièrement ses différents états, ainsi que ses transitions. Ensuite, le CPU garde le rôle d'encodeur.

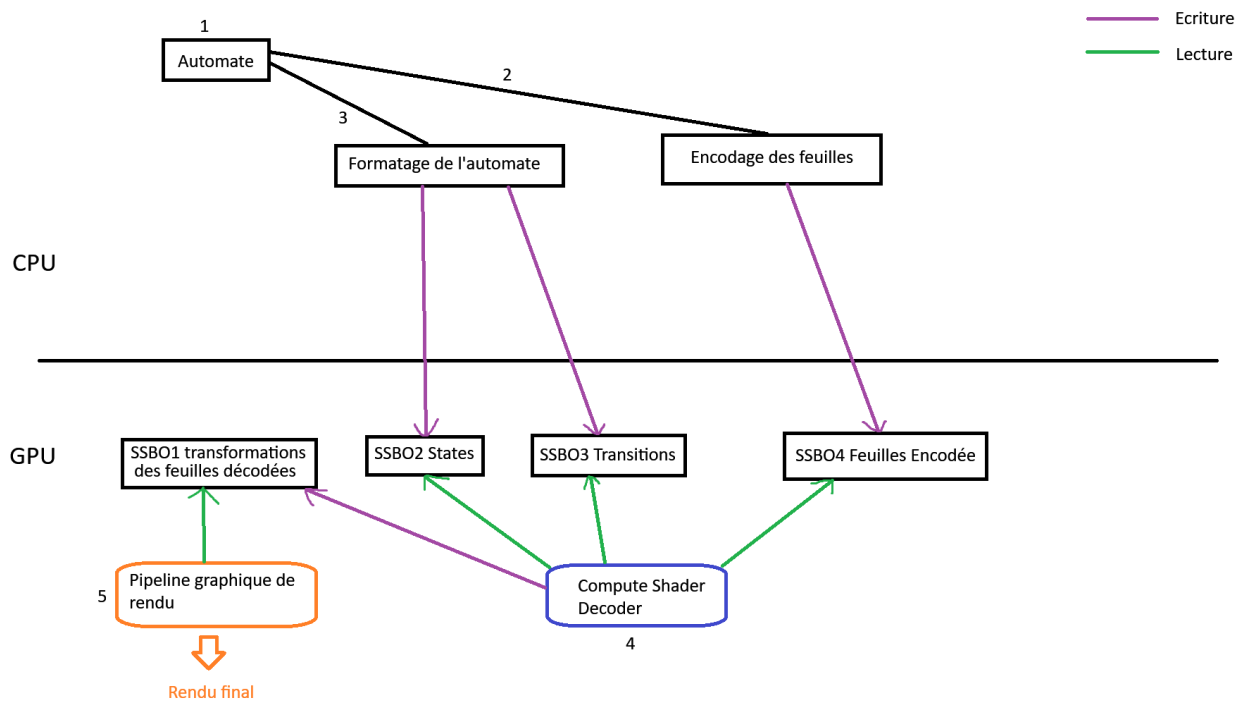


FIGURE 9 – Schéma général de la procédure de génération et de rendu final d'un automate.

La figure 9 montre le fonctionnement général du système encodage-décodage de ce projet tuteuré.

L'étape 1 est la phase de modélisation de l'automate par l'utilisateur.
L'étape 2 est l'étape de transformation de l'automate, et de son transfert au GPU.
L'étape 3 est l'encodage des feuilles de l'automate à une itération fini. C'est la fonction 2, appelé *encode*. L'ensemble des valeurs encodées est envoyé sur le GPU.
L'étape 4 est le décodage, qui est effectué entièrement sur GPU. Chaque valeur est décodée en parallèle, et les matrices résultantes sont stockées dans un buffer (SSBO1) utilisé pour le rendu.
L'étape 5 est un rendu instancié de la primitive originale, transformée par chacune des transformations précédemment décodées.

Un des grands avantage de ce système, est que l'encodage est toujours le même pour un automate donné et une itération donnée. Ainsi, l'utilisateur peut modifier les transformations composant l'automate, et seulement le décodage devra être exécuté de nouveau pour mettre à jour les transformations de chacune des feuilles.

6.2 Décodage des feuilles sur GPU

Le shader exécuté pour décoder les feuilles est une implémentation quasiment similaire à la version sur CPU. Pour être précis, la première boucle sur CPU itérant sur chaque feuille est supprimée. Car sur le GPU, un thread sera exécuté par feuille encodée, et chacune d'elles sera décodée de manière massivement parallèle.

7 Résultats

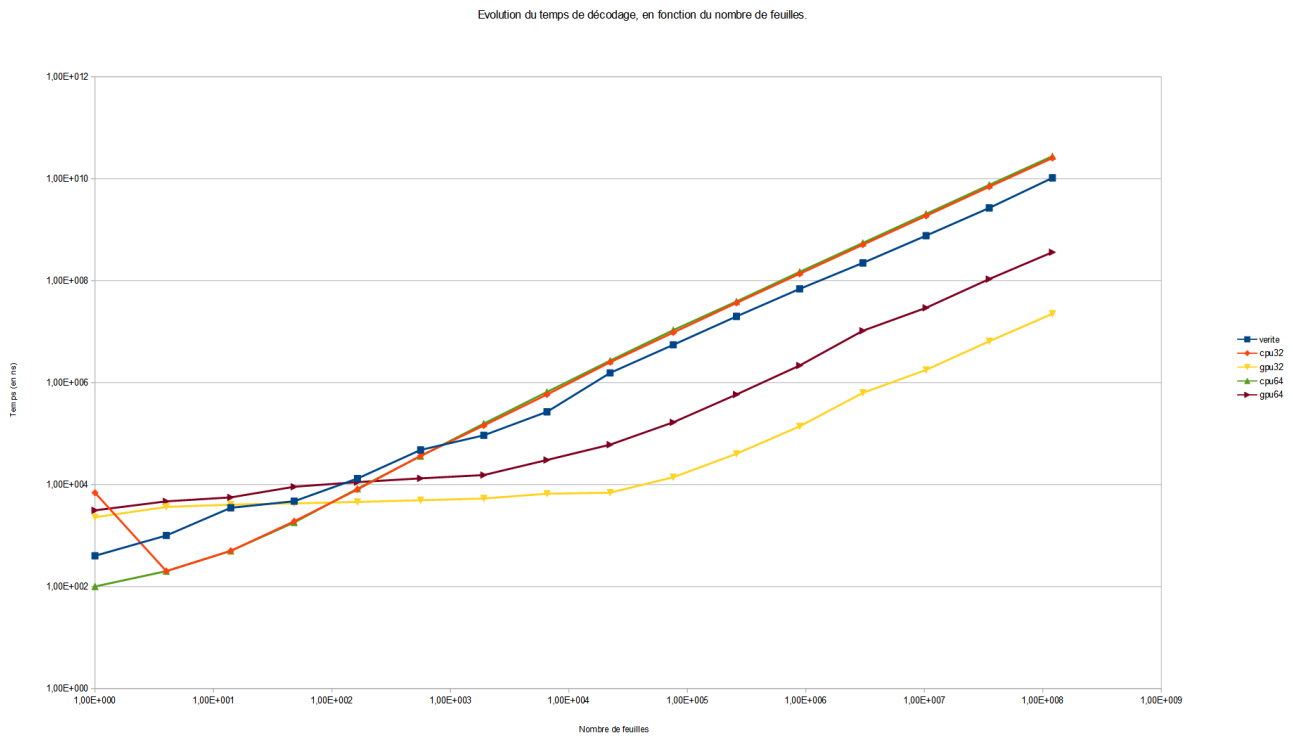


FIGURE 10 – Graphique du temps de décodage d'un automate en fonction du nombre de feuilles.

Les données ont été récoltées sur un Intel Core I7 13700K (@5.4GHz), et le GPU est une AMD Radeon 9070 XT.
Les échelles sont logarithmiques.

Ce graphique 10 met en concurrence le décodage des feuilles sur CPU, et GPU, avec des flottant 32 et 64 bits . Pour enrichir le graphique, une donnée du temps de génération des matrices classique sans encodage-décodage a été récolté. Ainsi, nous pouvons mesurer l'overhead du système.

Le graphique est très intéressant, déjà, nous pouvons observer que l'encodeur-décodeur sur CPU (en orange et vert) produit un overhead, comparé à la méthode naïve de calculs des feuilles (en bleu).

Ensuite, la seconde remarque importante est que les implémentations sur GPUs sont beaucoup plus rapides. On peut observer que pour un nombre de feuilles entre 1 et 500 000, le GPU a une allure assez plate et c'est donc très efficace. Cela peut sûrement s'expliquer également avec la taille du cache disponible sur le GPU, le pattern de lecture étant totalement séquentiel, c'est-à-dire que chaque thread lit et écris dans un emplacement unique, mais surtout qui est contiguë. (Le thread 1 lit et écrit à "l'emplacement" 1, le thread 2 au 2, etc.) Le GPU peut alors maximiser sa bande passante, mais on remarque bien une cassure vers 500 000, car la pression doit être trop élevée sur la VRAM et donc cela ajoute de la latence.

L'utilisation de nombres flottants en 64 bits impacte les performances, bien que l'exécution

reste plus rapide sur GPU que sur CPU. Les GPU disposent d'une répartition non uniforme des unité de traitements de données. Par exemple, le GPU que j'utilise possède un rapport 1 :32 entre les unités de calcul 64 bits et 32 bits, ce qui signifie qu'il y a 32 fois moins de puissance de calcul en 64 bits qu'en 32 bits. [Spécifications (Theoretical Performance)].

À l'inverse, les CPU ont une architecture différente, où les unités de calcul en virgule flottante (FPU) sont partagées entre les opérations 32 bits et 64 bits. Cela explique pourquoi les performances en 64 bits et 32 bits sont relativement similaires, bien que les calculs en 64 bits soient légèrement plus lents. Cette différence de performance est probablement due à une pression mémoire plus élevée. J'aborde ce point plus en détail dans la section 7.1.

7.0.1 Rapport performance

Nous allons nous détacher des échelles logarithmiques, qui permettent de bien voir une allure, mais cela est plus difficile de bien réaliser le résultat. Voici un tableau calculant l'accélération pour le décodage entre le CPU et le GPU.

Nombre Feuilles	CPU vs GPU (32bits)	CPU vs GPU (64bits)
1.000 00	3.026 32	0.032 05
4.000 00	0.054 95	0.042 74
14.000 00	0.123 76	0.089 29
48.000 00	0.443 93	0.199 12
164.000 00	1.754 39	0.743 73
560.000 00	7.439 02	2.726 59
1912.000 00	26.992 48	10.201 82
6528.000 00	88.348 48	21.609 50
22 288.000 00	362.673 41	44.521 45
76 096.000 00	688.908 05	64.466 69
259 808.000 00	911.866 90	66.814 33
887 040.000 00	982.642 73	68.679 16
3 028 544.000 00	804.796 67	52.924 31
10 340 096.000 00	1049.869 55	69.558 94
35 303 296.000 00	1073.869 61	70.222 47
120 532 992.000 00	1133.983 12	76.513 96

Table 2 – Accélération du décodage entre CPU et GPU

Le tableau 2 nous permet de mieux visualiser le gain de performance. En précision 32 bits, le gain est rapidement impressionnant, et rend la modification des transformations d'un automate ainsi que la visualisation en temps réel. Pour plus de valeurs compréhensibles, un tableau est fourni 4.

Pour ce qui est des doubles, le gain est toujours présent, mais est plus de l'ordre 1/16, ce qui est mieux que le traitement théorique dont on avait parlé en 10. Cela montre que la mémoire arrive à cacher de moitié la latence théorique de 1/32 produit par les calcules flottant 64 bits sur ce GPU.

7.1 Optimisation

Durant mes tests, la phase la plus limitante dans tout le pipeline de l'encodeur-décodeur est la partie encodage, ça s'explique naturellement parce qu'il est exécuté sur CPU, mais surtout que la première version ne possède aucun effort de programmation.

J'ai donc réécris une seconde fonction pour encoder les valeurs, en utilisant un buffer ping-pong, ainsi qu'en limitant l'utilisant de structure de donnée standard comme une std : :vector.

J'ai généré ces données d'analyse avec le même automate 4. En itérant jusqu'à les limites en mémoire de mon ordinateur.

Les résultats ont été calculés sur un Intel Core I7 13700K (@5.4Ghz) et 48Go de RAM.

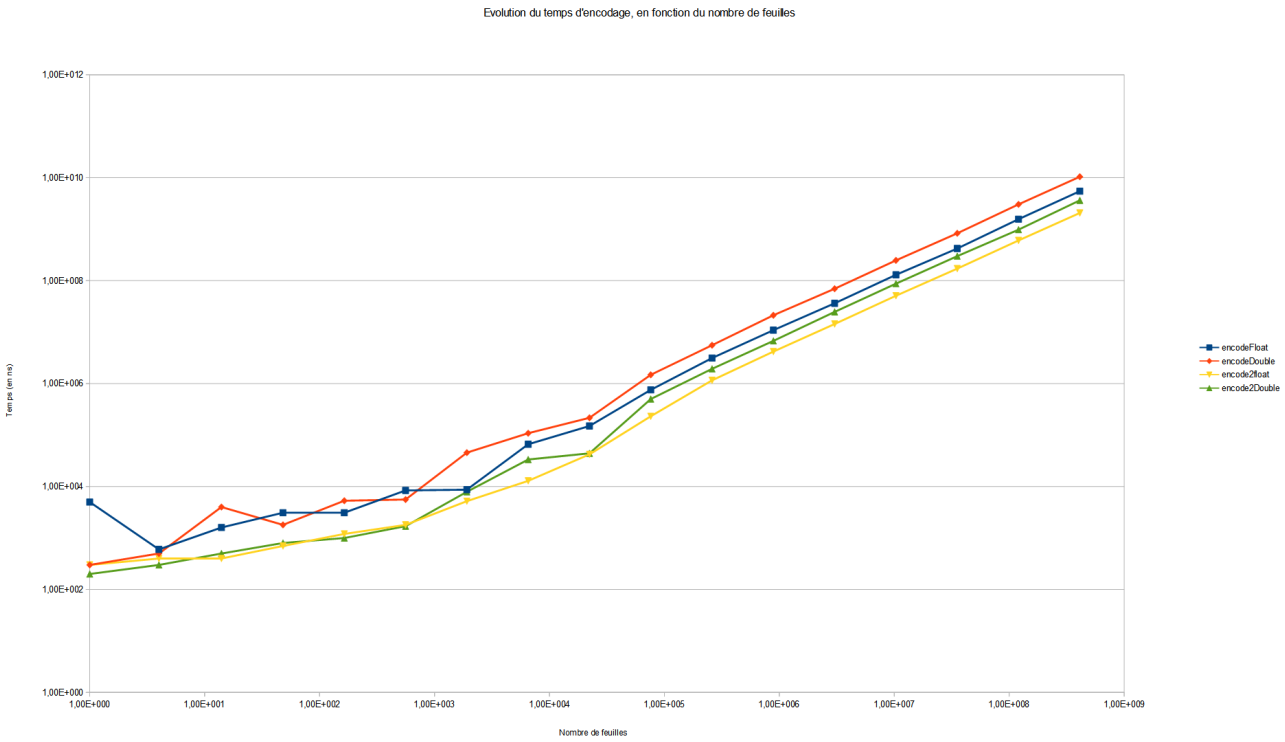


FIGURE 11 – Graphique du temps de l'encodage d'un automate en fonction du nombre de feuilles.

Le début de courbes est assez oscillant, cela s'explique probablement parce que les deux algorithmes travaillent sur des tableaux qui peuvent tenir entièrement dans les différents niveaux de cache du CPU, ce qui introduit pas mal de bruit en fonction de ce qui s'exécute sur la machine. Le graphique 11 met en concurrence les deux implémentations sur CPU, testé sur des flottants 32 bits, et 64 bits. Les deux axes sont des échelles logarithmiques. Si on fait une analyse sur les flottant 32 bits, les courbes bleu et jaune, on voit que la jaune est plus rapide. Dans l'ordre des 3 fois plus rapides. On peut également remarquer que l'impact des doubles est assez contrôlé. Pour le premier encodeur, l'écart est environ 2 fois plus lent en double qu'en

flottant. Cependant, pour le deuxième encodeur, l'écart est en moyenne de 1.68 fois plus lente.

L'encodage utilisant des doubles est plus lent, et est dans l'ordre de 2 fois plus lent, ce qui est cohérent, car il y a deux fois plus de pression, en mémoire et en utilisation de ressources comme les FPU.

Pour plus de lisibilité sur les temps d'exécution, voici le tableau des temps pour les deux analyses.

NombreFeuille	encodeFloat	encodeDouble	encode2float	encode2Double
1	5000 ns	300 ns	300 ns	200 ns
4	600 ns	500 ns	400 ns	300 ns
14	1600 ns	4000 ns	400 ns	500 ns
48	3100 ns	1800 ns	700 ns	800 ns
164	3100 ns	5300 ns	1200 ns	1000 ns
560	8400 ns	5600 ns	1800 ns	1700 ns
1912	8700 ns	45600 ns	5200 ns	7900 ns
6528	66300 ns	108600 ns	12900 ns	33500 ns
22288	150300 ns	216700 ns	42200 ns	44200 ns
76096	756500 ns	1.4831 ms	231900 ns	501500 ns
259808	3.1391 ms	5.5776 ms	1.1582 ms	1.9376 ms
887040	10.9241 ms	21.2822 ms	4.1907 ms	6.7647 ms
3028544	36.3091 ms	69.8133 ms	14.4025 ms	24.7677 ms
10340096	129.9143 ms	248.1056 ms	50.9119 ms	87.5565 ms
35303296	421.5587 ms	833.2183 ms	171.3093 ms	299.2871 ms
120532992	1.5660 s	3.0407 s	604.8464 ms	980.917 ms
411525376	5.4801 s	10.4739 s	2.0696 s	3.6375 s

Table 3 – Temps d'encodage en nanosecondes, millisecondes et secondes

NombreFeuille	Vérité	CPU32	GPU32	CPU64	GPU64
1	400 ns	6900 ns	2280 ns	100 ns	3120 ns
4	1000 ns	200 ns	3640 ns	200 ns	4680 ns
14	3500 ns	500 ns	4040 ns	500 ns	5600 ns
48	4700 ns	1900 ns	4280 ns	1800 ns	9040 ns
164	13100 ns	8000 ns	4560 ns	8300 ns	11160 ns
560	47700 ns	36600 ns	4920 ns	36100 ns	13240 ns
1912	92400 ns	143600 ns	5320 ns	156700 ns	15360 ns
6528	267200 ns	583100 ns	6600 ns	655200 ns	30320 ns
22288	1.5491 ms	2.5097 ms	6920 ns	2.698 ms	60600 ns
76096	5.5212 ms	9.5896 ms	13.92 µs	10.7221 ms	166.32 µs
259808	19.8782 ms	36.5841 ms	40.12 µs	39.065 ms	584.68 µs
887040	68.9264 ms	136.3122 ms	138.72 µs	148.7673 ms	2.16612 ms
3028544	223.0653 ms	506.6356 ms	629.52 µs	550.434 ms	10.4004 ms
10340096	763.2586 ms	1.8647 s	1.77616 ms	2.0409 s	29.33996 ms
35303296	2.6692 s	6.95197 s	6.47376 ms	7.54939 s	107.50672 ms
120532992	10.3795 s	25.3749 s	22.37676 ms	27.6642 s	361.55756 ms

TABLE 4 – Temps de décodage en nanosecondes, millisecondes et secondes

Pour l'étape d'encodage 4, nous pouvons observer plus facilement l'impacte de la seconde méthode d'encodage. Son optimisation réduit très bien le temps d'encodage, de l'ordre de 2.5 fois plus rapide.

Pour le décodage, on passe de l'ordre des secondes aux millisecondes. Même avec des flottant 64 bits, cela reste utilisable en temps réel. En regardant le tableau 2, en 32 bits, on a une accélération de l'ordre de 800 fois également, mais légèrement plus élevés que dans 1. Mais attention, le matériel utilisé n'est pas même, pour ce projet tuteuré, j'ai utilisé une AMD Radeon 9070 XT, alors que pour les chiffres de 1, c'était une Nvidia GTX 1080.

Avant l'encodage était directement fait sur le GPU sans utiliser de mémoire. Maintenant, il faut un tampon stockant les feuilles encodées. Aussi, l'implémentation a été changée un peu : Avant les transformations étaient décodé pour chaque rendu, maintenant le décodage est exécuté une seule fois pour remplir un buffer contenant toutes les transformations, et le rendu de chaque image est fait en utilisant celui-ci. La phase de décodage est exécutée uniquement si les transformations de l'automate ont été changées.

8 Conclusion

La généralisation de l'encodage d'un automate permet une accélération intéressante. Bien que l'étape d'encodage soit sur CPU, elle est de l'ordre raisonnable.

L'impact de l'encodage est obligatoire si l'automate n'est pas uniforme. Mais la chose intéressante est que l'encodage est effectué qu'une seule fois, le GPU décode les feuilles à chaque fois que les transformations associées aux transitions ont été modifiées. L'utilisateur peut alors travailler en temps réel pour modifier ses transformations. Si l'utilisateur modifie la

structure de l'automate, comme rajouter un état, une transition, etc. Il faut alors mettre à jour les données de l'automate sur le GPU, et encoder de nouveau les feuilles.

Ce système encodeur-décodeur souffre d'un problème de précision, comme discuté en 5.3.2. Il existe des solutions, qui ont un coût en performance, surtout sur des GPU grand publique comme celle la AMD 9070 XT utilisé pour ces tests. Il faudrait mesurer l'impacte sur des GPU professionnels qui sont beaucoup plus adaptés à ces calculs 64 bits.

De manière personnelle, ce projet tuteuré était très intéressant à faire, il m'a permis de mélanger plusieurs domaines théoriques, comme avec les arbres, et le codage arithmétique utilisé pour la compression, tout en couplant cela à du rendu. Il m'a permis de développer des algorithmes, mais surtout des structures de données compatible pour le GPU, de manière à l'utiliser de la meilleure manière possible. Et le bonus est que les résultats sont assez intéressant. Cela a été enrichissant de travailler sur ce projet.

8.1 Amélioration possible

On pourrait implémenter un système pendant l'encodage qui vérifie que les valeurs N-1 ou N+1 ne soient pas égales à une feuille N, ce qui montrerait qu'il y a eu une troncation de la valeur flottante, et il serait intéressant d'en notifier l'utilisateur. Cependant, cela aura forcément un petit coût en performance.

On pourrait également réfléchir à une implémentation de l'encodeur sur GPU, bien que moins importante que le décodeur, cela pourrait totalement décharger le CPU de cette tâche et d'avoir une accélération importante à la clef.

On pourrait aussi avoir une approche hybride avec l'implémentation de juin 2024 3. Le calcul des transformations est recalculé à chaque fois dans le Vertex Shader, sans stockage en VRAM. Cela gaspille beaucoup de ressources en calculs, mais limite grandement l'utilisation de mémoire. L'idée serait alors de calculer les feuilles à une itération données avec le décodeur, et de stocker ces feuilles en mémoire VRAM. Ensuite, calculer le reste des itération directement dans le vertex shader jusqu'à l'itération final voulu.

Annexes

8.2 Liens annexes

- `https://github.com/Gibus21250/IFSOpenGL`
- `https://lib.ube.fr/`
- `https://en.wikipedia.org/wiki/Arithmetic_coding`
- `https://developer.nvidia.com/tools-ecosystem`
- `https://www.amd.com/fr/products/software/rocm.html`
- `https://www.khronos.org/opencl/`
- `https://www.khronos.org/sycl/`
- `https://www.khronos.org/opengl/`
- `https://registry.khronos.org/vulkan/`
- `https://learn.microsoft.com/en-us/windows/win32/direct3d`
- `https://www.khronos.org/conformance/adopters/conformant-products/vulkan`
- `https://www.khronos.org/blog/mesh-shading-for-vulkan`

— https://en.wikipedia.org/wiki/Floating-point_unit
— https://developer20.com/memory-wall-problem/
— https://www.techpowerup.com/gpu-specs/radeon-rx-9070-xt.c4229