INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Réseaux Biologiques

# Accélération des simulations stochastiques en utilisant les capacités de parallélisation des processeurs graphiques.


Auteur :
Poquillon Titouan

Enseignant :
Hidde de JongL

Département Biosciences
BioInformatique et Modélisation


14 décembre 2020

# 1 Introduction

## 1.1 Contexte

La biologie des systèmes est une discipline qui peut être particulièrement gourmande en calcules. Certaines simulations peuvent prendre plusieurs heures, voire plusieurs jours. La fréquence de calcul des processeurs limite le nombre d'opérations qu'un programme séquentiel peut réaliser en un temps donné, et n'augmente pas aussi vite que la complexité des systèmes que l'on cherche a résoudre dans cette discipline. Les simulations stochastiques en particulier sont particulièrement gourmandes en calcul car elles demandent à être réalisé sur un grand nombre d'échantillons, chaque simulation pouvant donner un résultat différent. On s'intéressera au cas particulier de l'algorithme de Gillespie

Pour dépasser ces limites, la parallélisation des algorithmes s'est imposée comme une solution particulièrement intéressante, en permettant d'augmenter la puissance de calcul avec le nombre de processeurs. Il existe différentes approches pour paralléliser un code, et les plus classiques reposent sur la multiplications des CPU (Central Processing Units). Ces méthodes ont l'avantage de ne demander que peu de modifications aux codes existants, mais demandent des équipements coûteux (cluster de calcul ) ou dont il est difficile de garantir la sécurité (serveur cloud).

Une alternative existe grâce aux GPU (graphical processing units) qui sont devenus plus abordables financièrement (fig 1), un seul GPU fonctionnant comme un cluster de plusieurs centaines de processeurs. L'inconvénient des GPU, vient de leur architecture SIMD (Single Instruction, Multiple Device) qui pose une contrainte importante aux algorithmes exécutés sur des processeurs GPU, les processeurs voisins devant effectuer la même tâche.

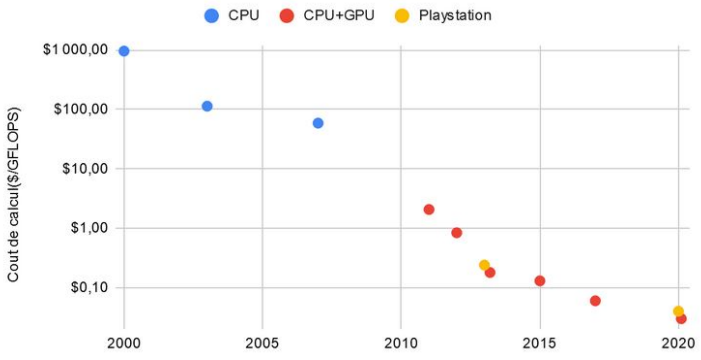



FIGURE 1 – Évolutions du coût de la puissance de calcul depuis les années 2000. [8]

Un autre avantage des GPU provient de leur forte démocratisation. Suite à l'accroissement de la demande en carte graphique dans l'industrie du jeu vidéo, il est maintenant courant de trouver sur des ordinateurs portables des cartes graphiques plusieurs dizaines de fois plus puissantes que leur processeur associé (c'est mon cas). Même si cette puissance est relative, comme on le verra par la suite.

## 1.2 L'Algorithme de Gillespie

Le GSSA (Gillespie Stochastique Simulation Algorithme) est une simulation de Monte-Carlo d'une trajectoire unique à partir de l'équation maîtresse chimique de systèmes réactifs spatialement homogènes. En considérant un système de réaction avec une liste d'espèces S et une liste de réactions possibles R qui décrivent l'interaction entre les espèces, l'algorithme est le suivant

— Initialisation du temps $t_{sim} = 0$ et des espèces à leur quantités initiale $S_i = S_0$

— A chaque étape, tant que $t_{sim} < t_final$ :

1. Pour chaque réaction $r_i$ on calcule la probabilité de transition (propension) $a_i = k_i[s_i]$ (dans le cas de réaction mono espèce)
2. On calcule le temps $dt$ qu'a mis la réaction à avoir lieu. $dt$ suit une loie exponentielle de moyenne $\sum(a_i)^{-1}$
3. On choisit aléatoirement la réaction qui à lieu sachant que la probabilité qu'une réaction $r_i$ ai lieu est $P(r_i) = \frac{a_i}{\sum a_i}$
4. On met à jour la population en fonction de la réaction choisie
5. On met à jour le temps $t_{sim} = t_{sim+dt}$

Cet algorithme permet de simuler une trajectoire spécifique. De par son aspect stochastique, un tel algorithme ne donnera pas les mêmes résultats à chaque exécution. On réalise donc généralement plusieurs milliers de simulation différentes, ce qui permet d'obtenir une carte de probabilité des états que peuvent prendre les systèmes biologiques simulés. Pour cette raison, cette simulation est particulièrement gourmande en calcul.

## 1.3 État de l'art

L'utilisation de GPU pour accélérer des simulations de processus stochastiques à déjà été montré expérimentalement, par exemple par Y. Zhou [1] et al en 2011. Le langage CUDA (Compute Unified

Device Architecture) développé par le constructeur de cartes graphiques NVIDIA a permis de aux programmeurs de développer des codes capables d'exploiter les GPU pour du calcul scientifique : [2] Cependant, à cette époque, bien que moins chères que les alternatives proposées par les CPU, les cartes graphiques utilisées dans ces expériences étaient des cartes professionnelles et non disponibles pour le grand public.

Ces 10 dernières années, l'augmentation du rapport qualité-prix des cartes publiques poussées par l'industrie du jeu vidéo on pousser certains laboratoires ont d'ailleurs déjà franchi le pas avec la génération des cartes GTX :Y.Fu et al. [3], A.Tangherloni et al.[4].

**Table 6.** GPU-powered tools for dynamic simulation, along with the speed-up achieved and the solutions used for code parallelization

| Simulation of the spatio-temporal dynamics and applications in Systems Biology | | | | |
|---|---|---|---|---|
| | Tool name | Speed-up | Parallel solution | Reference |
| Coarse-grain deterministic simulation with Euler method | – | 63× | GPU | [99] |
| Coarse-grain deterministic simulation with LSODA | cupSODA | 86× | GPU | [100] |
| Coarse-grain deterministic and stochastic simulation with LSODA and SSA | cuda-sim | 47× | GPU | [101] |
| Coarse-grain stochastic simulation with SSA (with CUDA implementation of Mersenne-Twister RNG) | – | 50× | GPU | [102] |
| Coarse- and fine-grain stochastic simulation with SSA | – | 130× | GPU | [103] |
| Coarse-grain stochastic simulation with SSA | – | – | GPU | [104] |
| Fine-grain stochastic simulation of large scale models with SSA | GPU-ODM | – | GPU | [105] |
| Fine-grain stochastic simulation with τ-leaping | – | 60× | GPU | [106] |
| Coarse-grain stochastic simulation with τ-leaping | cuTauLeaping | 1000× | GPU | [107] |
| RD simulation with SSA | – | – | GPU | [108] |
| Spatial τ-leaping simulation for crowded compartments | STAUCC | 24× | GPU | [109] |
| Particle-based methods for crowded compartments | – | 200× | GPU | [110] |
| Particle-based methods for crowded compartments | – | 135× | GPU | [111] |
| ABM for cellular level dynamics | FLAME | – | GPU | [112] |
| ABM for cellular level dynamics | – | 100× | GPU | [113] |
| Coarse-grain deterministic simulation of blood coagulation cascade | coagSODA | 181× | GPU | [114] |
| Simulation of large-scale models with LSODA | cupSODA*L | – | GPU | [115] |
| Parameter estimation with multi-swarm PSO | – | 24× | GPU | [116] |
| Reverse engineering with Cartesian Genetic Programming | cuRE | – | GPU | [95] |
| Parameter estimation and model selection with approximate Bayesian computation | ABC-SysBio | – | GPU | [117] |

FIGURE 2 – Recensement des articles traitant de l'accélération GPU en biologie des systèmes avant 2017. [5]

En 2017, la revue Briefing in Bioinformatics [5] (fig 2) recensait une vingtaine d'articles sur l'accélération des calculs par GPU en biologie des systèmes On peut donc se demander si ces outils, développés ces 10 dernières années, sont exploitables avec un ordinateur portable.

## 1.4 Problématique

*Les approches de parallélisation existante de simulations stochastiques, appliquées au GSSA sont-elles intéressantes en utilisant un GPU commercial, que l'on peut trouver dans des ordinateurs grand publiques, ou des ordinateurs portables ?*

| Nom | Accélération théorique | Publication | Algorithme | Processeur |
|---|---|---|---|---|
| cuda-sim | 12x | 2011 [1] | GSSA | Tesla C2050 GPU |
| GPU-ODM | ∼10x | 2012 [6] | GSSA | NVIDIA 480GTX GPU |
| cuTauLeaping | 1000x | 2014 [7] | TauLeaping | GeForce GTX 590 |

TABLE 1 – Solutions existantes sélectionnées pour tester le GSSA sur GPU

### 1.5 Approche envisagé

Dans un premier temps, on réalisera une évaluation des performances des GPU et CPU de l'ordinateur sur lequel on réalisera nos expérimentations de parallélisation. On utilisera les données constructeurs, un logiciel de benchmark (Geekbench 4 [9]) et enfin un algorithme de construction des fractales de Mandelbrot. Cette évaluation nous permettra de nous assurer que le GPU est fonctionnel et de mesurer ses performances dans un contexte neutre.

On essaiera ensuite d'exploiter trois solutions existantes d'accélération de l'algorithme de Gillespie (table 1) pour reproduire les expériences de performance comparative réalisées des résultats obtenus et pour identifier si ces solutions sont adaptées aux GPU non professionnels modernes.

Enfin, en fonction des résultats de l'étape précédente, notamment si on ne réussi pas à exploiter les solutions, on essaiera de construire notre propre implémentation parallèle de l'algorithme de Gillespie en python à l'aide de la library numba-cuda

## 2 Évaluation du matériel

### 2.1 Matériel utilisé

GPU : RTX 2060 CUDA 11.1 CPU : Intel Core i5-9300H Windows10 Python 3.7

### 2.2 Données constructeur et Geekbench 4

GPU :

6451.6 GFLOPS (Floating Point Operation Per Second, peak single precision)

GB4 SFFT (Sparse Fast Fourier Transform) : 93.82 GFLOPS

CPU :

301 GFLOPS (peak single precision)

GB4 SFFT : 9.2/ 43.7 GFLOPS (single/multithread)

### 2.3 Test comparatif à l'aide de l'algorithme de Mandelbrot

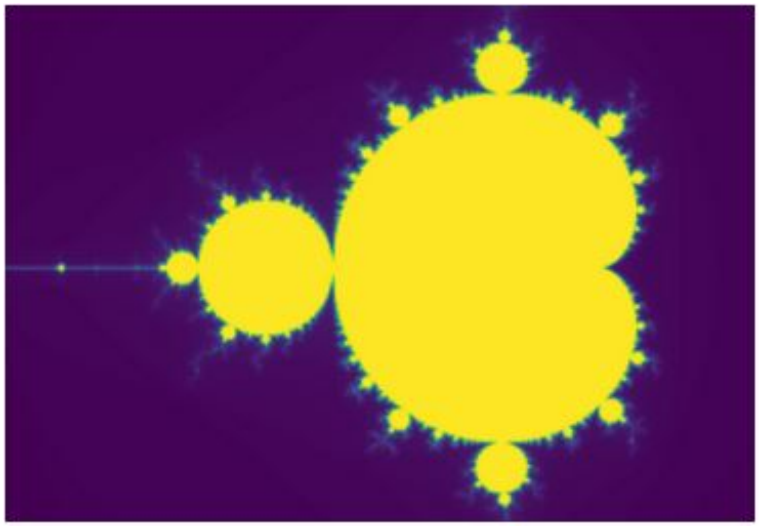


FIGURE 3 – Fractales de Mandelbrot

L'ensemble de Mandelbrot se compose de nombres complexes c pour lesquels la fonction fc (z) = z 2 + c ne diverge pas lorsqu'il est itéré d'une valeur initiale de z = 0, et reste borné en valeur absolue. Les images produites par ces complexes ont une frontière infiniment complexe avec des détails récursifs de plus en plus fins à des résolutions importantes. C'est une courbe fractale (fig3).

L'algorithme permettant de calculer ces fractal est particulièrement simple à implémenter et à paralléliser. En modifiant légèrement une implémentation proposée [10] (Le code disponible **ici**), on a pu évaluer évaluer de façon simple les performances comparatives des GPU et CPU de l'ordinateur sur des système de plus en plus complexes, vérifier le bon fonctionnement de CUDA, et du package python numba cuda (fig 4).

Grâce au test réalisé sur cet ensemble , on remarque notamment que le ratio de performances dépend de la taille des systèmes étudiés. Si le GPU est systématiquement plus rapide que le CPU dans ce cas, on peut remarquer que la ration passe d'un facteur 3 à un facteur 10 lorsque le système augmente de taille. On est relativement proche des ratio de performances observées dans la section précédente

## 3 Test des solutions existantes

Malheureusement aucune des solutions existantes n'a pu être expérimentée.

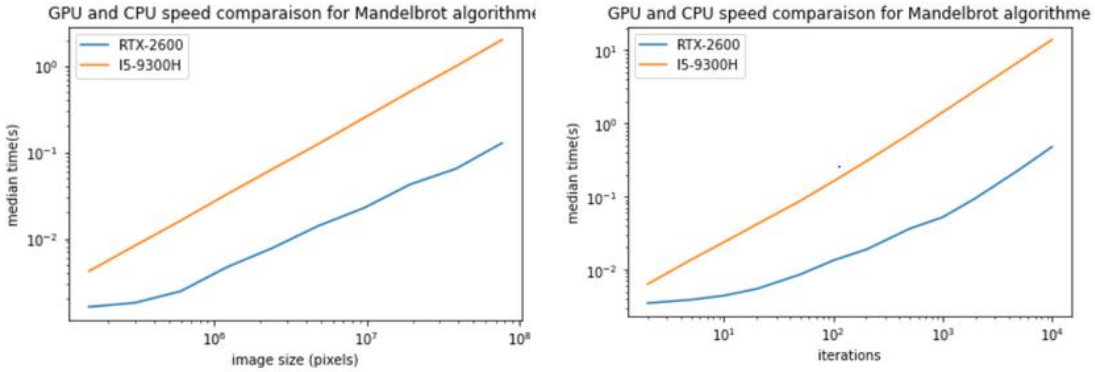



FIGURE 4 – Performance médiane sur des échantillons de 10 tests entre les implémentations de calcul de l'ensemble de Mandelbrot CPU et CPU+GPU. Évaluation en augmentant a) la résolution de l'image, b) le nombre d'itération pour assurer la convergence.

Le package cuda-sim à été développé sous python2. Cette version n'est plus supportée depuis janvier 2020. Il aurait peut être été possible d'installer l'ensemble des dépendances nécessaire pour faire tourner le programme, on a préféré passer aux solutions suivantes

Bien que plus récents, GPU-ODM et cuTauLeaping présentent un autre type de problème. Ces deux logiciels ont été écrits pour utiliser une architecture qui n'est plus supportée depuis la version 9.0 de CUDA (version actuelle : 11.1). L'utilisation de ces solutions aurait donc nécessité ou une réécriture partielle, ou l'utilisation version plus ancienne de l'architecture de nos machines.

S'il aurait peut être été possible de trouver un moyen pour faire fonctionner ces solutions, l'échec des tests met en évidence les difficultés que l'on peut rencontrer à essayer de reproduire et réutiliser les résultats de publications en bioinformatique, à cause de l'évolution rapide des logiciels, pilotes et du matériel. Sans support et mise à jour régulière, les algorithmes et les codes produits par la recherche scientifique risquent de devenir rapidement obsolètes.

Suite à ce constat, on a décidé de consacrer le temps du projet à implémenter directement une solution parallèle pour le GSSA.

# 4 Implémentation Python

## 4.1 Numba-Cuda

Numba(version 0.5) est une librairie python [11] qui permet de transformer des fonctions écrites en python en un code compilé (par défaut, python est un langage interprété) à l'aide d'un décorateur(@numba jit), ce qui permet d'accélérer grandement un algorithme python.

Numba.cuda est un module de Numba permettant de transformer des fonctions pour les rendre exécutables par un GPU à l'aide d'un décorateur semblable (@numba.cuda jit). Cependant, ces fonctions ne peuvent exploiter que certaines fonctionnalités de python (tous les types ne sont pas utilisables, seul certaines opérations peuvent être réalisées), et présentent des contraintes structurelles fortes (réorganisation de l'algorithme autour des kernels du GPU).

## 4.2 Contexte de l'expérience

Ce travail s'inspire du travail réalisé par Justin Bois et Michael Elowitz pour Caltech en 2019 pour Caltech [12], ou est présenté une implémentation non parallèle accéléré grâce à @numba jit du GSSA. Cette implémentation est illustré par le système simple de la production de protéine : DNA→mRNA→protein Par soucis de simplicité, et pour pouvoir comparer facilement les résultats obtenu, on a choisi de travailler sur le même système, en utilisant les même paramètres et la même structure de résultats

## 4.3 Première approche

La première approche que l'on a essayé d'appliquer pour rendre le code consistait à décorer à l'aide des décorateurs de cuda et à modifier légèrement le code de J.Bois et M.Elowitz pour construire une structure parallélisable que l'on puisse lancer sur le GPU

Cette approche a échoué à tous les niveaux : à cause des opération limites permises par CUDA, et à cause de la structure de l'implémention (parallélisation impossible avant le plus haut niveau)

## 4.4 Solution proposée

On a décidé de réécrire complètement l'algorithme pour qu'il s'adapte aux contraintes imposées par les GPU. Cette réécriture a imposé de coder en dur les paramètres du système. Il existe très probablement une solution élégante pour rendre l'algorithme polyvalent, il s'agit d'une piste d'amélioration pour le futur.

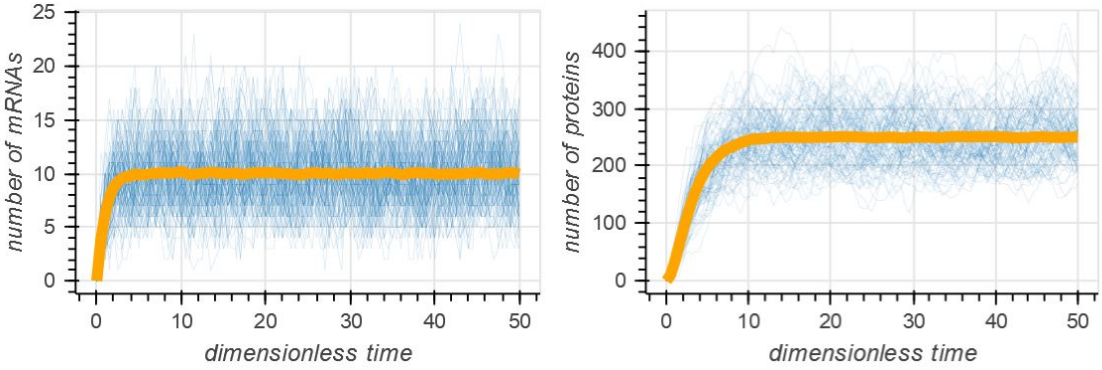



FIGURE 5 – Résultat moyens (en orange) de 5000 simulations (en bleu) de 50 unité de temps arbitraire pour le système de production de protéines, de paramètres ($\beta_p = \beta_m = 10$ et $\gamma = 0.4$)

Le code commenté de cette solution est disponible **ici**. Pour résumer, elle s'organise autour de trois fonctions :

— GSSA_gpu - cette méthode tourne sur un processeur du GPU. elle réalise une unique itération du GSSA pour une simulation donnée et met à jour le temps et la population en fonction de la réaction réalisée (ainsi que l'information d'interruption de la simulation, un simulation s'interrompt lorsqu'elle atteint un point temporel ),

— GSSA_kernel - pour chaque simulation, cette méthode attribue un kernel qui va faire tourner une itération de GSSA_gpu. Au bout d'un nombre arbitraire d'itération (1000) cette méthode renvoie l'état de l'ensemble des simulation à GSSA_Master,

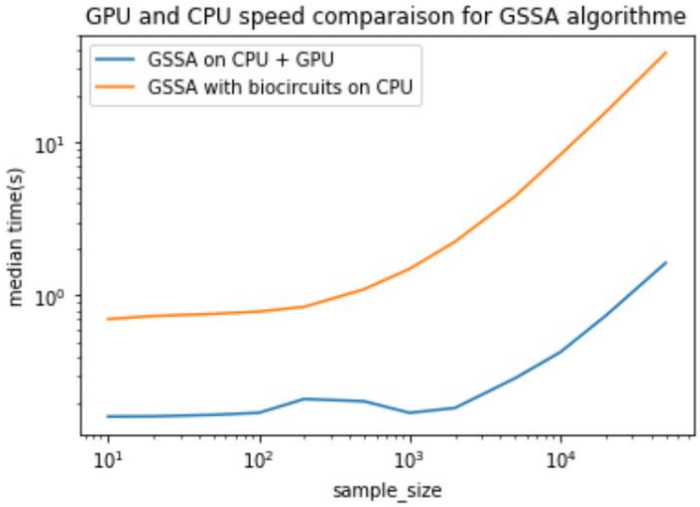
— GSSA_master - cette méthode tourne sur le CPU. Pour chaque étape de simulation (point temporel) elle vérifie si l'ensemble des simulation sont parvenu cette étape et appelle GSSA_kernel tant que ce n'est pas le cas. Ensuite, elle enregistre l'état de l'ensemble des simulation au point temporel, et relance le processus pour le point temporel suivant

Cette solution produit bien les résultat attendu pour une simulation stochastique de la production de protéines (5) :

## 4.5 Évaluation

Pour évaluer les performance de cette implémentation, on a comparé la vitesse d'exécution en fonction de la taille de l'échantillon étudié pour cette implémentation et celle de la library biocircuit

La figure 6 met en évidence l'accélération que permet la parallélisation de l'algorithme de Gillespie avec un GPU, puisque on parvient à multiplier par 20 la vitesse d'exécution de notre simulation




| Samples | 10 | 20 | 30 | 100 | 200 | 500 | 1000 | 2000 | 5000 | 10000 | 20000 | 50000 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GPU speedup | 4 | 4 | 4 | 4 | 4 | 5 | 8 | 11 | 15 | 19 | 22 | 23 |

FIGURE 6 – Test de vitesse médiane d'exécution de l'algorithme GSSA parallélisé (bleu) et de la solution biocircuit (orange) sur des taille d'échantillon des simulations croissante. Les simulations sont réalisées sur 50 unité de temps arbitraire pour un système de production de protéines de paramètres ($\beta_p = \beta_m = 10$ et $\gamma = 0.4$)

lorsque le nombre de d'échantillons devient suffisamment grand (n>10000). Par contre l'accélération est plus faible sur les tout petits systèmes, la parallélisation ne pouvant pas exprimer son plein potentiel. De manière générale, cette implémentation nous fait gagner un ordre de grandeur en termes de vitesse de calcul pour le GSSA. Ce résultat est cohérent avec les chiffres de la littérature scientifiques sur des implémentation classique du GSSA et avec l'évaluation des performance que l'on a réalisé en début de projet.

# 5 Discutions

Ce projet permet d'illustrer les gains de performance non négligeable qu'apporte l'utilisation des processeurs graphiques dans le domaine des simulations stochastiques. Gains que l'on a pu estimer à une accélération d'environ un ordre de grandeur sur cet ordinateur.

Cependant, on aura aussi mis en évidence la complexité que peut impliquer l'utilisation de de la parallélisation en utilisant les GPU. Les solutions proposées dans la littérature deviennent

rapidement obsolète à moins d'être régulièrement maintenu, et l'implémentation directe, si elle est possible, impose des contraintes fortes, tant dans la structure que dans les opérations limitées que l'on peut effectuer.

Approcher un problème de simulation stochastique avec une solution de parallélisation sur GPU ne semble donc intéressant que lorsque le temps de calcul est un problème majeur (simulations de plusieurs heures à plusieurs jours), ce qui rentabilise le temps passé à développer (ou trouver) un code adapté. Dans un contexte étudiant, on peut imaginer qu'une telle situation arrive pendant un projet. En travaux pratiques ou en cours, c'est moins probable.

Pour aller plus loin, On pourra essayer d'améliorer notre implémentation pour que les paramètres de notre système n'aient pas à être codés en dur. Cela pourrait permettre d'étudier des système plus complexes que la production de protéines, comme le repressilator décrit par Elowitz et Leibler. On pourra aussi étudier l'impact de la taille du système sur l'accélération engendré par GPU. Enfin, l'évaluation du temps de calcul utilisé à chaque étape du GSSA parallèle montre que c'est le transfère de données du GPU au CPU qui consomme le plus de ressource. On pourrait donc encore améliorer le gain de puissance de notre algorithme en trouvant une solution à ce problème.

## Références

[1] Yanxiang Zhou, Juliane Liepe, Xia Sheng, Michael P. H. Stumpf, Chris Barnes *GPU accelerated biochemical network simulation.* Bioinformatics, Volume 27, Issue 6, 15 March 2011, Pages 874–876, https ://doi.org/10.1093/bioinformatics/btr015

[2] https ://developer.nvidia.com/gpu-accelerated-libraries

[3] You Fu & Wei Zhou *A novel parallel Markov clustering method in biological interaction network analysis under multi-GPU computing environment.* The Journal of Supercomputing volume 76, pages7689–7706(2020)

[4] Andrea Tangherloni, Marco S. Nobile, Daniela Besozzi, Giancarlo Mauri & Paolo Cazzaniga *LASSIE : simulating large-scale models of biochemical systems on GPUs.* BMC Bioinformatics volume 18, Article number : 246 (2017)

[5] Marco S Nobile, Paolo Cazzaniga, Andrea Tangherloni, Daniela Besozzi *Graphics processing units in bioinformatics, computational biology and systems biology.* Briefings in Bioinformatics, Volume 18, Issue 5, September 2017, Pages 870–885, https ://doi.org/10.1093/bib/bbw058

[6] Komarov I, D'Souza RM. *Accelerating the Gillespie exact stochastic simulation algorithm using hybrid parallel execution on graphics processing units.* . PLoS One2012 ;7(11) :e46693

[7] Nobile MS , Cazzaniga P, BesozziD, et al *cuTauLeaping : A GPU-powered tau-leaping stochastic simulator for massive parallel analyses of biological systems.* . PLoS One 2014 ;9(3) :e91963.

[8] https ://en.wikipedia.org/wiki/FLOPS

[9] https ://www.geekbench.com/geekbench4/

[10] https ://github.com/harrism/numba_examples/blob/master/mandelbrot_numba.ipynb

[11] https ://numba.pydata.org/ https ://numba.pydata.org/numba-doc/latest/cuda/index.html

[12] http ://be150.caltech.edu/2019/handouts/12_stochastic_simulation_all_code.html