SIMULATION RAPIDE DE MODÈLES DE NEURONES IMPULSIONNELS SUR CARTE GRAPHIQUE

Alexandre Chariot, Renaud Keriven
ODYSSÉE LAB.
ECOLE DES PONTS
6, rue Blaise Pascal
F-77455 Marne-la-Vallée cedex 2
emails : {chariot,keriven}@certis.enpc.fr

Romain Brette
ODYSSÉE LAB.
ECOLE NORMALE SUPÉRIEURE
45, rue d'Ulm
F-75230 Paris cedex 05
email : brette@di.ens.fr

Résumé
Poussés par l'industrie du jeu vidéo, les développements récents des cartes graphiques, tant en puissance qu'en souplesse de programmation, ont comme retombée inattendue leur utilisation en simulation numérique. Les processeurs de ces cartes, les GPUs[1] *sont désormais des coprocesseurs parallèles rapides et très peu coûteux, de type SIMD/CREW*[2]*. La simulation de réseaux de neurones de taille importante fait généralement appel à des machines parallèles de type grappe et il serait aisé d'équiper chaque noeud d'une carte graphique. Considérant ici un seul de ces noeuds, nous proposons un schéma de simulation de réseaux de neurones impulsionnels adapté aux GPUs et son implémentation pratique. Ce schéma à temps discret tient compte des délais de transmission. Avec un GPU milieu de gamme, des simulations allant jusqu'à* $2.5\ 10^5$ *neurones pour 250 connexions aléatoires par neurone, montrent un gain de 20 par rapport à un CPU usuel dans le temps d'intégration du schéma. Stockées sur CPU, les connexions restent prises en charge par ce dernier, ce qui ramène le gain à un facteur réduit mais encourageant de 2. Ce premier pas fait porter nos futurs efforts vers une simulation* tout-GPU, *envisageable sur les cartes de dernière génération.*

Mots clés
Parallélisme, Processeurs graphiques, Réseaux de neurones, Neurones impulsionnels

1 Motivations

Les neurosciences computationnelles reposent en partie sur la simulation de grands ensembles de neurones (voir par exemple [1]). Les modèles de neurones sont typiquement des systèmes hybrides, consistant en des équations différentielles et des impulsions discrètes. L'approche classique consiste à simuler les équations différentielles à l'aide d'un schéma d'approximation en temps discret (e.g. Euler, Runge-Kutta) et transmettre des impulsions aux neurones voisins lorsqu'une condition de seuil est satisfaite [2].

En notant N le nombre de neurones, p le nombre de synapses par neurones, F la fréquence de décharge des neurones et dt le pas de temps, le temps de la simulation se décompose en 1) l'intégration des équations, d'ordre N/dt, et 2) la transmission des impulsions, d'ordre $F \times N \times p$ [3]. Pour simuler de grands réseaux complexes (temps élevé pour la partie 1) et/ou à forte connectivité (temps élevé pour la partie 2), l'approche actuelle consiste à distribuer le calcul sur une grappe de processeurs [4]. Il s'agit d'équipements lourds, tant en termes de coût que de maintenance.

Or il existe sur la plupart des ordinateurs actuels de véritables machines parallèles : les processeurs de carte graphique (GPU). Ceux-ci sont au départ prévus pour effectuer des tâches liées au rendu des images, mais ils peuvent être détournés de leur fonction. Grâce au développement de l'industrie du jeu vidéo, ces processeurs graphiques sont puissants et économiques. En termes de puissance de calcul brute, les GPUs sont aujourd'hui environ 10 fois plus rapides que les CPUs, et leur croissance elle-même est plus rapide (voir figure 1). A notre connaissance, seuls des réseaux de neurones analogiques [5] et certains modèles non génériques de réseaux impulsionnels [6] ont été portés sur GPU. Nous présentons ici des algorithmes permettant de simuler rapidement des réseaux de neurones impulsionnels génériques sur GPU.

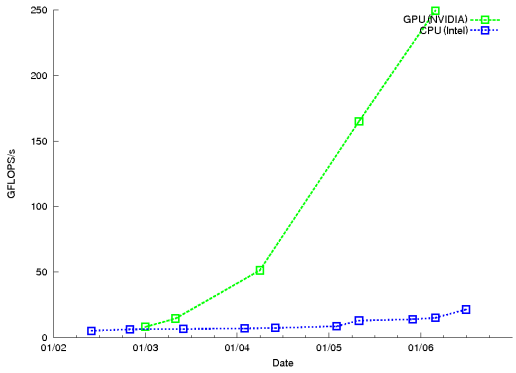


FIG. 1. Evolution comparée GPU Nvidia - CPU Intel

[1] *Graphical Processing Units*
[2] *Simple Instruction Multiple Data/Concurrent Read Exclusive Write*

2 Programmation Générique sur GPU

Par Programmation Générique sur Processeur Graphique (GPGPU), on désigne l'utilisation des GPUs pour faire d'autres calculs que l'affichage d'objets tridimensionnels, affichage pour lesquels ils sont initialement conçus. Un grand nombre de simulations numériques sont adaptées à ce traitement. Nous décrivons brièvement l'architecture actuelle des GPUs et comment ils sont généralement utilisés en GPGPU. Pour plus de détails, nous invitons le lecteur à consulter [7, 8].

2.1 Architecture d'un GPU

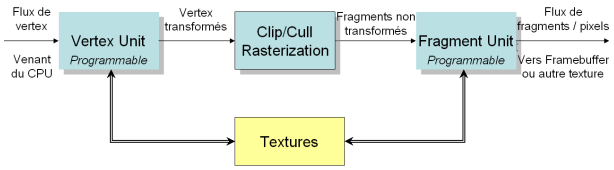


FIG. 2. Schéma simplifié du pipeline d'un GPU

Tout comme les CPUs, les GPUs sont organisés en pipeline, ce qui est déjà une forme de parallélisme. La puissance des GPUs vient, entre autres, de la présence de plusieurs pipelines identiques (typiquement 24) travaillant en parallèle. Le principe d'un pipeline est le suivant (figure 2) : (i) Un flux de sommets (*vertices*) est fourni en entrée par le CPU. Ces sommets sont reliés entre eux pour former des triangles. (ii) Une unité (*vertex unit*), modifie la position des sommets et éventuellement d'autres attributs qui leurs sont affectés (normale, couleur, etc.). Cette unité est programmable et le programme qui lui est affecté est appelé *vertex shader*. (iii) Une deuxième unité, non programmable, prend ensuite le relais, principalement pour transformer chaque triangle en pixels (ou *fragments*). On parle de *rasterization*. (iv) Enfin, une dernière unité (*fragment unit*) calcule pour chaque pixel une couleur et l'affiche à l'écran. Cette unité est, elle aussi, programmable, via un *fragment shader*.

On notera que : (i) Chaque pixel peut aller lire des couleurs dans des tableaux (*textures*) en fonction de *coordonnées de texture* qui lui sont affectées. (ii) Le rendu peut se faire dans une texture : la couleur finale des pixels ne s'affiche pas mais peut être stockée dans une texture. (iii) Pour un rendu donné, c'est le même vertex (resp. fragment) shader qui est exécuté pour tous les sommets (resp. pixels) : c'est en cela qu'un GPU est SIMD[3].

2.2 Utilisation en GPGPU

L'implémentation de simulations numériques sur GPU ignore généralement les vertex shaders. On peut remarquer que, si on dessine un simple rectangle, alors la carte calcule en parallèle :

$$\forall (i,j),\ T_{\text{out}}(i,j) = \mathcal{F}(T_{\text{in}_1}, T_{\text{in}_2}, ..., T_{\text{in}_n}, i, j)$$

où T_{out} est une texture de sortie, les T_{in_k} des textures d'entrée, et (i,j) les coordonnées du pixel, et $\mathcal{F}$ le fragment shader chargé dans la carte. Les textures sont des tableaux bidimensionnels mémorisant des couleurs mais il est aisé de simuler tout type de donnée. On remarquera que $\mathcal{F}$ peut accéder à tout emplacement des textures d'entrée, qui ne sont d'ailleurs pas nécessairement de même taille. En revanche, un pixel ne peut évidemment changer la couleur de sortie des autres pixels. C'est en cela qu'un GPU est de type CREW. Ainsi, on peut par exemple effectuer $T_{\text{out}}(i,j) = T_{\text{in}}(i+1,j)$, mais pas $T_{\text{out}}(i+1,j) = T_{\text{in}}(i,j)$. Ce point est crucial dans notre cas : *un neurone qui émet une impulsion ne peut prévenir directement ses successeurs ; il faudrait à l'inverse que chaque neurone aille demander à ses prédécesseurs s'ils ont émit une impulsion*. Pour des réseaux limités, une telle démarche est tenable [6]. Ici, pour des réseaux quelconques, nous résolvons ce problème en reportant sur CPU cette partie de la simulation.

3 Réseau de neurones impulsionnels et implémentation

3.1 Modèle utilisé

Pour tester notre algorithme, nous considérons un réseau de N neurones excitateurs, chacun étant connecté à 250 voisins tirés au hasard (ainsi le nombre de neurones postsynaptiques est fixe, mais pas le nombre de neurones présynaptiques). Nous avons choisi pour les neurones un modèle impulsionnel simple, un intègre-et-tire [9] avec des courants synaptiques excitateurs exponentiels. Ainsi l'état de chaque neurone est donné par la valeur des variables V, le potentiel membranaire, et I, l'entrée synaptique totale, qui sont régies par les équations différentielles suivantes :

$$\begin{aligned} \tau_V \frac{dV}{dt} &= -V + I + I_0 \\ \tau_I \frac{dI}{dt} &= -I \end{aligned}$$

où $\tau_V = 10$ ms est la constante de temps membranaire, $\tau_I = 3$ ms est la constante de temps synaptique, et $I_0 = 1.1$ est une entrée constante. Une impulsion est produite lorsque V atteint à l'instant t le seuil, égal à 1 par convention[4]. On réinitialise alors V à 0 et l'impulsion est transmise aux voisins après un délai constant d fixé : à l'instant $t+d$, $I_i \leftarrow I_i + c_0$ pour chaque neurone cible i ($c_0 = 0.5/250$). L'objet de l'étude n'étant pas la qualité de l'approximation, les équations différentielles sont implémentées simplement selon un schéma d'Euler (pour ce modèle linéaire, une intégration exacte est possible, mais ce n'est pas le cas en général).

[3] Sur les derniers GPUs, les vertex units sont MIMD

[4] On note que les neurones déchargent spontanément, il s'agit donc d'un réseau d'oscillateurs couplés.

3.2 Implémentation

L'algorithme 1 implémente le schéma précédent avec les notations supplémentaires suivantes : C est la matrice de connexion ($C_{ij} = c_0$ si i est une cible de j, 0 sinon), $m = d/\Delta t$ (supposé entier), T mémorise les temps d'impulsion, X est une variable intermédiaire, et les Y^k cumulent les effets des impulsions pour l'instant futur $t + k\Delta t$.

Algorithme 1 Euler avec délai

1: $Y^k \leftarrow 0$ $(k = 1 \text{ to } m)$
2: **loop**
3: $T_i \leftarrow 0$
4: **for** $k = 1$ to m **do**
5: **for** $i = 1$ to N **do**
6: $V_i \leftarrow V_i + \frac{\Delta t}{\tau_v}(-V_i + I_i + I_0)$
7: $I_i \leftarrow I_i + \frac{\Delta t}{\tau_i}(-I_i + Y_i^k)$
8: **if** $V_i > 1$, **then**
9: $V_i \leftarrow 0, T_i \leftarrow k$
10: **end if**
11: **end for**
12: **end for**
13: **for** $k = 1$ to m **do**
14: $X_i \leftarrow \{T_i = k\}$ $(i = 1 \text{ to } N)$
15: $Y^k \leftarrow C.X$
16: **end for**
17: **end loop**

Les lignes 3 à 12 sont effectuées sur le GPU, les lignes 13 à 16 sur le CPU. Ce découpage est motivé par la nécessité de transférer le moins fréquemment possible des données entre le CPU et le GPU. D'où le principe consistant à calculer m pas de temps du schéma d'Euler, avant de gérer les impulsions. Ici les transferts sont limités à : (i) envoyer Y^k vers le GPU entre les lignes 2 et 3, et (ii) récupérer T sur le CPU entre les lignes 12 et 13.

Suivant le délai d et le pas de temps Δt nécessaire à l'intégration[5], il se peut que m devienne trop grand pour pouvoir mémoriser tous les Y^k. Nous avons donc aussi testé un schéma simplifié (algorithme 2) dans lequel l'instant exact k d'impulsion n'est pas pris en compte. Moins consommateur en mémoire, ce schéma rend compte, sur notre exemple, du comportement qualitatif du réseau. Les lignes 3 à 13 sont effectuées par le GPU, la ligne 14 par le CPU.

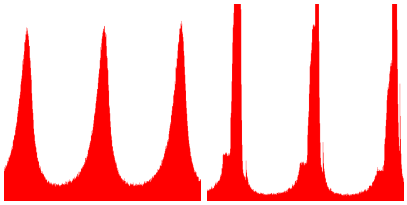
4 Résultats

La figure 3 montre la fréquence de décharge du réseau au cours du temps. Il apparaît que les neurones déchargent de manière régulière et corrélée, comme observé dans [10] (figure 4) pour une architecture complète. Une mesure quantitative montre que les deux algorithmes retrouvent la même fréquence, bien que les profils soient naturellement un peu différents.

[5]Encore une fois, nous avons choisi un modèle simple d'intègre-et-tire, mais il ne s'agit que d'un exemple.

Algorithme 2 Délais de transmission approchés

1: $Y \leftarrow 0$
2: **loop**
3: $I \leftarrow I + Y$
4: $X \leftarrow 0$
5: **for** $k = 1$ to m **do**
6: **for** $i = 1$ to N **do**
7: $V_i \leftarrow V_i + \frac{\Delta t}{\tau_v}(-V_i + I_i + I_0)$
8: $I_i \leftarrow I_i + \frac{\Delta t}{\tau_i}(-I_i)$
9: **if** $V_i > 1$, **then**
10: $V_i \leftarrow 0, X_i \leftarrow 1$
11: **end if**
12: **end for**
13: **end for**
14: $Y \leftarrow C.X$
15: **end loop**



(a) (b)

FIG. 3. Fréquence de décharge en régime stable du réseau simulé. (a) Algorithme 1. (b) Algorithme 2.

La figure 4 montre, pour différentes valeurs de N et de Δt (donc du nombre m de boucles internes), le temps passé par l'algorithme dans le schéma d'Euler (hors calcul des Y, donc). Nous comparons ici un GPU milieu de gamme (GeForce 7800GTX) avec un CPU standard (Pentium 3GHz). Le gain moyen est d'environ 20 ce qui est bien la valeur attendue en cas d'utilisation efficace du GPU.

La figure 5 montre, dans les mêmes conditions, le temps total pour l'algorithme 2. Le gain tombe à 2 environ. Ce gain est prometteur et requiert des commentaires. (i) Ce gain est normal étant donnée la complexité du calcul de Y et la simplicité du modèle de neurone choisi. Plus le modèle sera complexe, plus le gain sera important. (ii) Nous étudions la possibilité de porter le calcul de Y sur GPU. Avec un GPU haut de gamme[6], plus rapide et doté de plus de mémoire, un tel portage est envisageable.

5 Conclusion

Nous avons proposé un schéma de simulation de modèles de neurones impulsionnels sur carte graphique.

[6]La notion de "haut de gamme" est toute relative puisqu'un tel GPU ne coûte que 600 euros pour une puissance 3 fois supérieure et une mémoire 4 fois plus importante (1Go).

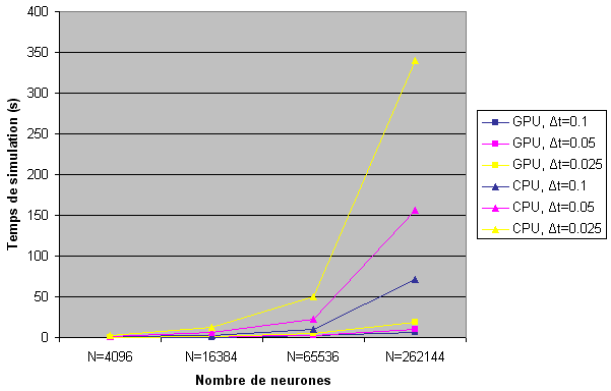


FIG. 4. Temps de calcul CPU/GPU de la partie schéma d'Euler de la simulation (lignes 3 à 12 de l'algorithme 1 ou 3 à 13 de l'algorithme 2)

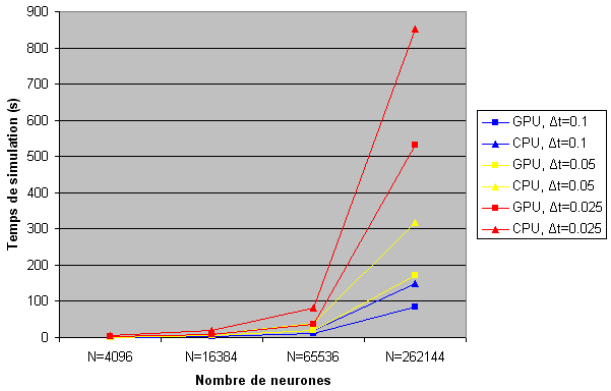


FIG. 5. Temps de calcul CPU/GPU de l'algorithme 2 complet

Son implémentation est efficace et prometteuse et peut servir de base à l'utilisation des GPUs comme co-processeur dans une grappe de calcul ou sur un simple ordinateur personnel. Les pistes suivies actuellement sont la validation de notre schéma avec des modèles complexes de neurones et la recherche d'une simulation entièrement prise en charge par le GPU.

Références

[1] R.D. Traub, D. Contreras, M.O. Cunningham, H. Murray, F.E.N. LeBeau, A. Roopun, A. Bibbig, W.B. Wilent, M.J. Higley and M.A. Whittington (2005). Single-Column Thalamocortical Network Model Exhibiting Gamma Oscillations, Sleep Spindles, and Epileptogenic Bursts. J. Neurophysiol. 93 : 2194-2232.

[2] M.V.Mascagni & A.S.Sherman. In : C.Koch & I.Segev (eds.), Methods in neuronal modeling. Cambridge, MA : MIT Press (1998).

[3] Brette R. (2006) Exact simulation of integrate-and-fire models with synaptic conductances. Neural Comput. 18(8), pp. 2-7 & 9-13.

[4] Morrison, A., Mehring, C., Geisel, T., Aertsen, A. and Diesmann, M. (2005). Advancing the boundaries of high connectivity network simulation with distributed computing. Neural Comput. 17(8), 1776-1801.

[5] Kyoung-Su Oh et Keechul Jung, "'GPU implementation of neural networks"', *Pattern Recognition*, 37, 2004, pp. 1311-1314.

[6] F. Bernhard, R. Keriven, Spiking Neurons on GPU, *International Conference on Computational Science*, Workshop on General Purpose computation on Graphics Processing Units (GPGPU), Readings, UK, 2006.

[7] `http://www.gpgpu.org`, site regroupant un grand nombre de travaux orientés GPGPU, il contient un forum très actif.

[8] Ouvrage collectif, *GPU Gems 2, Programming Techniques for High-Performance Graphics and General-Purpose Computation* (Matt Pharr ed., 2005).

[9] Tuckwell H (1988). Introduction to theoretical neurobiology, vol 1 : linear cable theory and dendritic structure. Cambridge University Press, Cambridge.

[10] Van Vreeswijk C. (1996) Partial synchronization in populations of pulse-coupled oscillators. Phys. Rev. E 54 :5522-5537.