Réduction parallèle pour le calcul du maximum sur GPU

Soufiane EL AMRANI, Sami AGOURRAM

16 mai 2025

Résumé

Ce rapport présente l'implémentation et l'analyse comparative de quatre stratégies de réduction parallèle pour le calcul du maximum sur GPU avec CUDA. Nous évaluons les performances, la complexité et les limitations de chaque méthode pour différentes tailles de données, et identifions l'approche optimale selon le contexte d'utilisation. Les résultats montrent que les méthodes optimisées peuvent atteindre des accélérations jusqu'à 14× par rapport à l'approche naïve, avec la méthode utilisant la mémoire partagée offrant les meilleures performances pour les grands ensembles de données.

1 Introduction

La réduction parallèle est une opération fondamentale en programmation GPU qui consiste à combiner un ensemble de valeurs en une seule à l'aide d'une opération binaire associative (ici, le maximum). Cette opération apparaît fréquemment dans les algorithmes parallèles et son optimisation est cruciale pour de nombreuses applications scientifiques et d'analyse de données.

Ce projet implémente et compare quatre approches distinctes pour effectuer la réduction du maximum sur GPU :

1. **Méthode naïve** avec mémoire globale
2. **Réduction utilisant la mémoire partagée**
3. **Réduction en deux étapes** (intra-bloc puis inter-bloc)
4. **Réduction avec registres internes** (niveau thread)

Chaque méthode exploite différents aspects de l'architecture hiérarchique des GPU CUDA, notamment la mémoire globale, la mémoire partagée et les registres. Notre objectif est d'identifier quelle méthode offre les meilleures performances selon la taille des données et les caractéristiques du matériel.

2 Méthodologie

2.1 Environnement de test

Les expériences ont été réalisées dans l'environnement Kaggle avec les configurations suivantes :
— GPU : NVIDIA Tesla T4
— Capacité de calcul CUDA : 7.5
— Tailles de tableau testées : 1 000, 10 000, 100 000, 1 000 000, 10 000 000 éléments

2.2 Description des méthodes implémentées

2.2.1 Méthode 1 : Réduction naïve avec mémoire globale

Cette approche utilise exclusivement la mémoire globale et des opérations atomiques. Chaque thread calcule son maximum local sur une portion des données, puis met à jour la valeur maximale globale avec une opération atomique `atomicMaxFloat`. L'inconvénient principal est la contention sur cette unique adresse mémoire globale.

2.2.2 Méthode 2 : Réduction avec mémoire partagée

Cette méthode exploite la mémoire partagée ultrarapide disponible pour chaque bloc. Les threads d'un même bloc chargent d'abord les données en mémoire partagée puis effectuent une réduction parallèle avec synchronisation. Cette approche réduit considérablement les accès à la mémoire globale plus lente.

2.2.3 Méthode 3 : Réduction en deux étapes

Cette approche divise le problème en deux kernels : le premier effectue une réduction au sein de chaque bloc (similaire à la méthode 2), tandis que le second combine les résultats de tous les blocs en un seul résultat final. Cette méthode est conçue pour éviter les transferts CPU-GPU nécessaires dans d'autres approches.

2.2.4 Méthode 4 : Réduction avec registres internes

Cette méthode exploite les registres ultrarapides de chaque thread. Chaque thread traite plusieurs éléments, stocke le maximum local dans ses registres, puis participe à une réduction en mémoire partagée au niveau du bloc, réduisant davantage les accès à la mémoire globale.

2.3 Mesure des performances

Les performances sont mesurées à l'aide des événements CUDA (`cudaEvent_t`) qui permettent de chronométrer précisément l'exécution des kernels. Chaque méthode est évaluée sur :

— Sa correction (comparaison avec le résultat CPU)
— Son temps d'exécution en millisecondes

3 Résultats et analyse

3.1 Performances brutes

La Figure 1 présente les temps d'exécution des quatre méthodes pour différentes tailles de tableau.

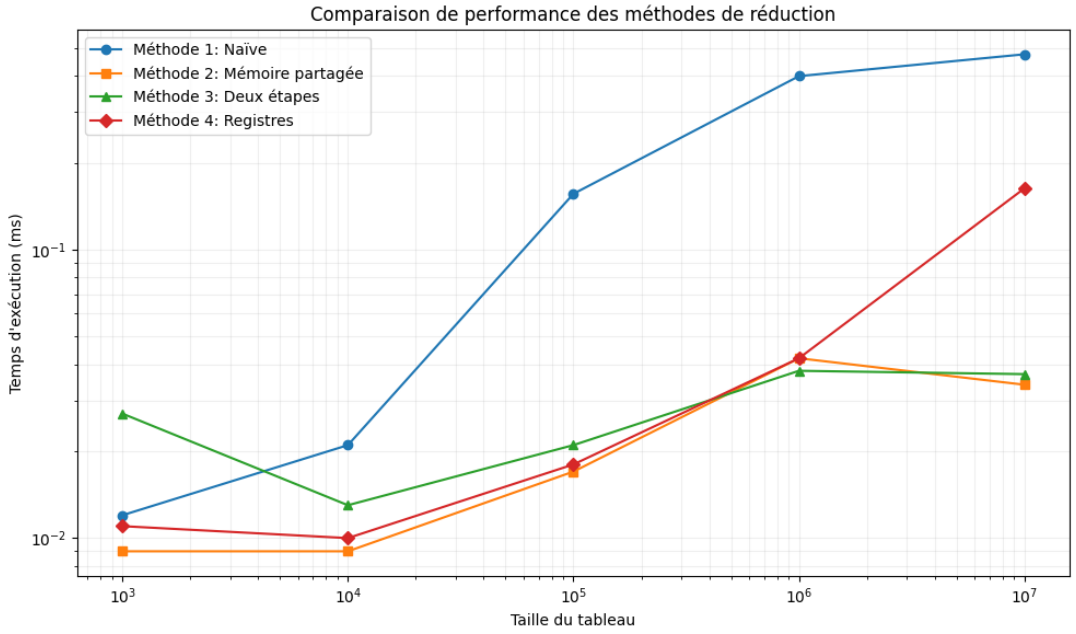

FIGURE 1 – Comparaison des temps d'exécution des différentes méthodes (échelle log-log)

On observe que :

— La méthode naïve devient progressivement la plus lente à mesure que la taille des données augmente
— Pour les petites tailles (1 000 - 10 000 éléments), toutes les méthodes ont des performances similaires
— Pour les tailles moyennes et grandes (100 000 - 10 000 000), les méthodes optimisées surpassent nettement la méthode naïve
— La méthode avec mémoire partagée (Méthode 2) offre les meilleures performances pour les très grandes tailles
— La méthode avec registres (Méthode 4) montre une dégradation de performance pour les très grandes tailles

TABLE 1 – Temps d'exécution (ms) pour chaque méthode et taille de tableau

Taille	Naïve	Mémoire partagée	Deux étapes	Registres
1 000	0.012	0.009	0.027	0.011
10 000	0.021	0.009	0.013	0.010
100 000	0.156	0.017	0.021*	0.018
1 000 000	0.399	0.042	0.038*	0.042
10 000 000	0.475	0.034	0.037*	0.163

3.2 Accélération relative

La Figure 2 illustre l'accélération obtenue par chaque méthode optimisée par rapport à la méthode naïve.

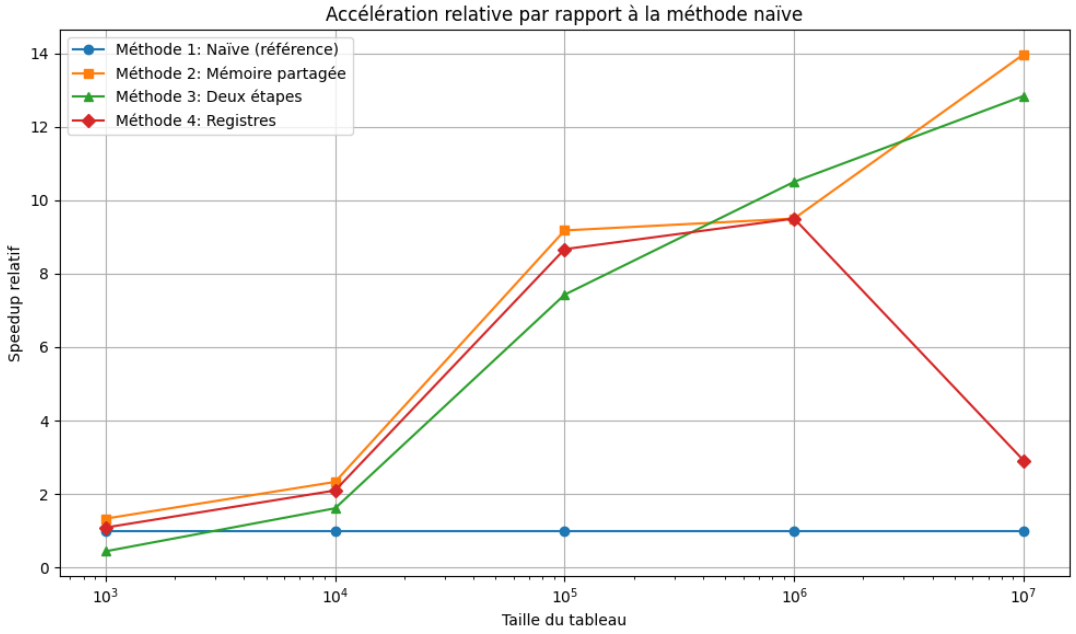

FIGURE 2 – Accélération relative par rapport à la méthode naïve

Les résultats montrent que :

— Les méthodes optimisées atteignent des accélérations impressionnantes, jusqu'à 14× pour la méthode avec mémoire partagée
— L'accélération augmente avec la taille des données jusqu'à 10 millions d'éléments
— La méthode avec registres montre une accélération qui diminue pour les très grandes tailles (10 millions d'éléments)
— La méthode avec mémoire partagée offre de façon constante les meilleures accélérations pour les grandes tailles

3.3 Analyse de scaling

La Figure 3 montre comment le temps d'exécution évolue avec la taille du problème, normalisé par rapport à la plus petite taille.

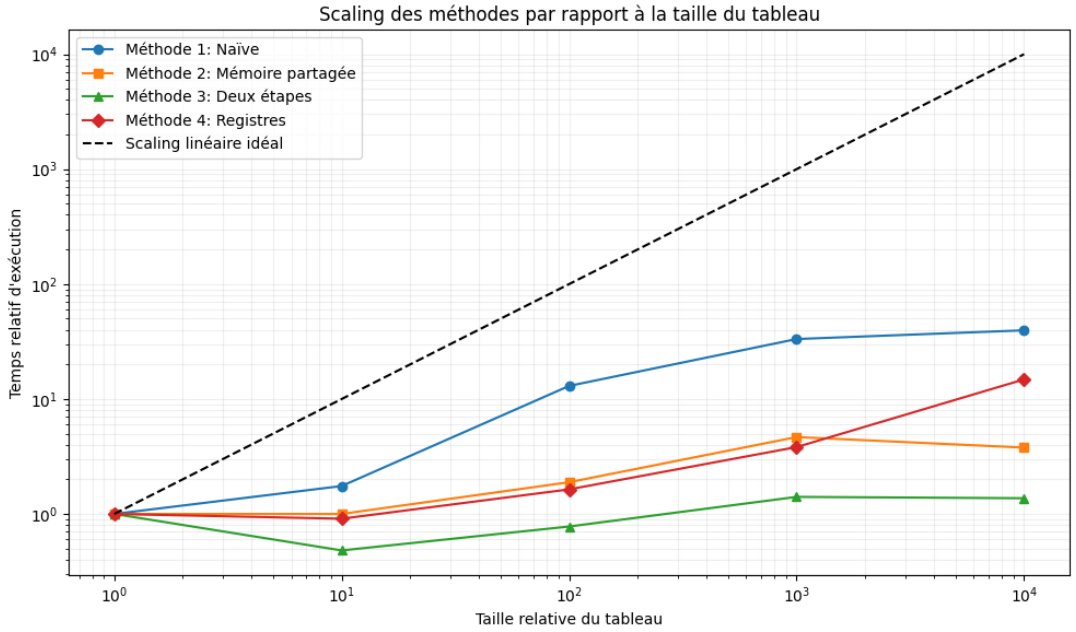

FIGURE 3 – Scaling des méthodes par rapport à la taille du tableau

Cette analyse révèle que :
— Aucune méthode n'atteint le scaling linéaire idéal (ligne pointillée)
— La méthode naïve présente le pire scaling, s'écartant rapidement de l'idéal
— La méthode en deux étapes montre le meilleur scaling, restant presque constant
— La méthode avec mémoire partagée reste très efficace même pour les grandes tailles
— La méthode avec registres montre une inflexion pour la plus grande taille, suggérant une limitation de ressources

4 Discussion et analyse comparative

4.1 Comparaison des méthodes

TABLE 2 – Analyse comparative des méthodes de réduction

Critère	**Naïve**	**M. partagée**	**Deux étapes**	**Registres**
Complexité d'implémentation	Simple	Moyenne	Complexe	Moyenne
Performance petites tailles	Bonne	Excellente	Moyenne	Bonne
Performance grandes tailles	Faible	Excellente	Bonne*	Moyenne
Scaling	Médiocre	Très bon	Excellent*	Bon
Utilisation mémoire	Minimale	Moyenne	Élevée	Moyenne
Robustesse	Élevée	Élevée	Faible*	Élevée

4.2 Complexité théorique

— **Méthode 1 (Naïve)** : O(n) lectures en mémoire globale + O(t) opérations atomiques, où t est le nombre de threads.
— **Méthode 2 (Mémoire partagée)** : O(n) lectures en mémoire globale + O(b·log(k)) opérations en mémoire partagée, où b est le nombre de blocs et k la taille d'un bloc.
— **Méthode 3 (Deux étapes)** : O(n) lectures + O(b·log(k)) opérations en mémoire partagée + O(log(b)) opérations dans le second kernel.
— **Méthode 4 (Registres)** : O(n) lectures en mémoire globale + O(b·log(k)) opérations en mémoire partagée + O(b) opérations atomiques.

4.3 Limitations identifiées

— **Méthode naïve** : Contention sur les opérations atomiques, limitant sévèrement le scaling pour les grands nombres de threads.
— **Méthode avec mémoire partagée** : Limitée par la taille de la mémoire partagée disponible par multiprocesseur, mais très efficace dans la plage de tailles testées.
— **Méthode en deux étapes** : Problème de correction pour les grandes tailles ; surcoût du lancement de deux kernels pour les petites tailles.
— **Méthode avec registres** : Performance qui se dégrade pour les très grandes tailles, suggérant une pression sur les registres ou une occupation réduite des multiprocesseurs.

4.4 Recommandations d'utilisation

En fonction de nos résultats, nous recommandons :
— **Pour les petites tailles** ($< 10\,000$) : La méthode avec mémoire partagée offre le meilleur compromis entre simplicité et performance.
— **Pour les tailles moyennes** (10 000 - 1 000 000) : La méthode avec mémoire partagée reste le choix optimal.
— **Pour les très grandes tailles** ($> 1\,000\,000$) : La méthode avec mémoire partagée offre les meilleures performances, tandis que la méthode avec registres commence à montrer des limitations.
— La méthode en deux étapes présente un potentiel intéressant (bon scaling).

5 Conclusion

Notre analyse comparative des quatre stratégies de réduction parallèle pour le calcul du maximum sur GPU démontre l'importance cruciale d'exploiter efficacement la hiérarchie mémoire dans les architectures CUDA. Les résultats montrent clairement que :
— Les méthodes optimisées peuvent atteindre des accélérations impressionnantes (jusqu'à $14\times$) par rapport à l'approche naïve
— La méthode utilisant la mémoire partagée (Méthode 2) offre le meilleur compromis entre performance, simplicité d'implémentation et robustesse
— L'efficacité des différentes méthodes varie avec la taille des données, confirmant l'importance d'adapter l'approche au contexte d'utilisation

Ce projet illustre parfaitement comment l'exploitation intelligente des différentes hiérarchies mémoire d'un GPU peut conduire à des gains de performance substantiels pour des opérations fondamentales comme la réduction parallèle.