

MSI3 PROJET 2: PROGRAMMATION SUR GPU

Clément Baillet

Le projet précédent nous avait permis d'adapter un code de résolution d'un problème de Poisson en y ajoutant une composante hybride MPI/OpenMP, afin d'exploiter au mieux les différents cœurs et threads disponibles sur un processeur. Toutefois, bien que les CPU aient gagné en parallélisme ces dernières années, ils restent moins performants sur cet aspect précis que les cartes graphiques. En effet, les GPU ont été conçus initialement pour effectuer du calcul parallèle, par exemple du calcul matriciel pour l'affichage graphique. Il est donc naturel de vouloir utiliser ces dispositifs pour du calcul scientifique.

Initialement, le projet 2 consistait en l'implémentation du code Poisson en CUDA, pour l'exécuter sur carte graphique Nvidia. Toutefois, disposant d'un appareil Apple muni d'une puce Apple Silicon M4 (avec 24 Go de mémoire unifiée entre CPU et GPU), j'ai jugé intéressant de proposer une implémentation utilisant l'API Metal combinée à la librairie `metal-cpp`, afin de conserver un code en C++. J'ai également rédigé une version utilisant Kokkos, une API générique qui permet de cibler au choix OpenMP, CUDA ou HIP (API utilisée par AMD pour le calcul sur GPU). Pour tester ce code, je me suis servi de la machine Rhum fournie par l'ENSTA, qui possède un GPU Nvidia (une Tesla K80, avec 2×12 Go de VRAM). La Figure 1 montre une section de la solution séquentielle de référence, visualisée avec le logiciel ParaView.

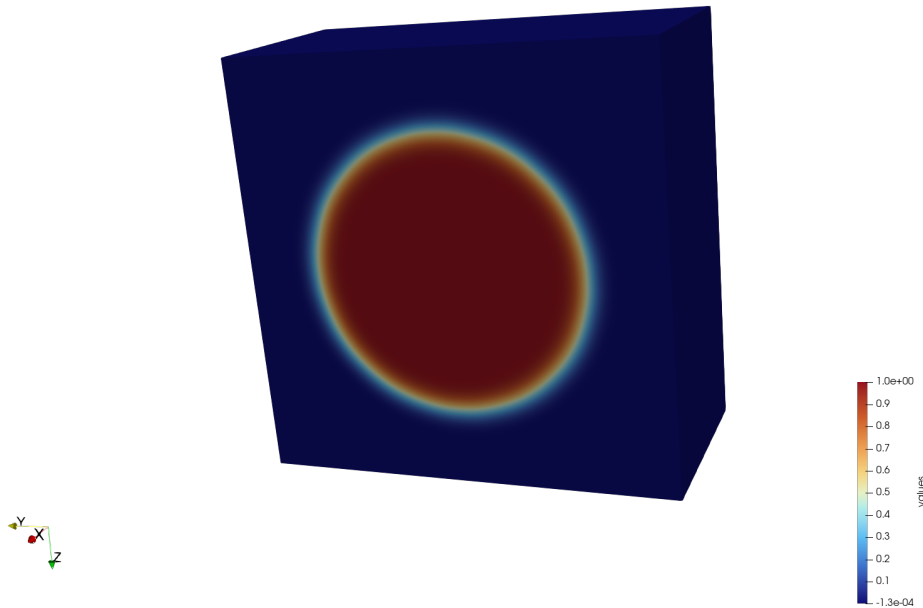


Figure 1: Section du cube dans lequel on résout le problème de Poisson

1 Version Metal

L'implémentation Metal du projet s'appuie sur le squelette fourni pour la version CUDA, que j'ai cherché à respecter autant que possible. J'ai notamment créé un dossier `metal` regroupant les fichiers dédiés aux appels à l'API Metal.

L'architecture adoptée repose sur une séparation nette entre, d'une part, la logique de gestion du matériel et, d'autre part, les noyaux de calcul. Au cœur de ce dispositif se trouve la classe `Context`, définie dans les fichiers `Context.cxx` et `Context.hxx`, qui joue un rôle central. Implémentée sous la forme d'un Singleton, elle est responsable de l'initialisation unique du `MTLDevice`, interface logicielle vers le GPU, ainsi que de la création de la `CommandQueue`, la file d'attente par laquelle transitent toutes les instructions de calcul. Cette centralisation permet d'éviter les coûts de réinitialisation du périphérique à chaque itération de la boucle de temps.

La communication des données entre le processeur principal (CPU) et la carte graphique (GPU) est assurée par des structures communes alignées, définies dans le fichier `SharedStructs.h`. Ce fichier est inclus à la fois par le code C++ et par le code Metal, ce qui garantit une interprétation cohérente des paramètres de simulation (dimensions de la grille, pas de temps, etc.) des deux côtés de la mémoire. Les fonctions de calcul, écrites en Metal Shading Language (MSL), sont regroupées dans le fichier `kernels.metal`. On y trouve par exemple les noyaux `k_iteration`, `k_init` et `k_boundaries`, qui définissent les opérations mathématiques exécutées en parallèle par chaque thread sur le GPU.

L'envoi des instructions au GPU suit un enchaînement précis, principalement orchestré par des fichiers C++ comme `iteration.metal.cxx`. Lorsqu'une étape de calcul est demandée, le programme charge la bibliothèque de fonctions Metal compilées et crée un *Pipeline State Object* (PSO), qui fixe la configuration du GPU pour la tâche considérée. Un *Compute Command Encoder* est ensuite instancié pour enregistrer la suite de commandes. Le programme associe alors les buffers contenant les champs de température aux index attendus par le shader, configure les constantes via la structure partagée, puis définit la topologie de la grille de calcul. Cette dernière étape consiste à spécifier la taille des groupes de threads (*threadgroups*) ainsi que le nombre de groupes nécessaires pour couvrir l'ensemble du domaine spatial. Une fois l'encodage terminé, le *Command Buffer* est soumis à la file d'attente du contexte, déclenchant une exécution massivement parallèle et asynchrone sur le processeur graphique.

2 Version Kokkos

L'implémentation basée sur la librairie Kokkos vise avant tout la portabilité des performances, en permettant d'exécuter le même code source sur différentes architectures matérielles sans modifications majeures. La configuration de l'environnement d'exécution est centralisée dans le fichier `paramKokkos.hxx`. Ce dernier définit les espaces de mémoire et d'exécution, en adaptant automatiquement les types `MemSpace` et `ExecSpace` selon que la compilation active ou non le support CUDA via des directives du préprocesseur. C'est également dans ce fichier que sont définies les politiques d'exécution, telles que `range_policy`, qui déterminent la manière dont les itérations de boucle sont réparties sur les ressources matérielles disponibles.

La définition physique du problème, incluant les conditions initiales et les termes sources, se trouve dans le fichier `user.hxx`. Les fonctions mathématiques y sont précédées de la macro `KOKKOS_INLINE_FUNCTION`, ce qui indique au compilateur de générer du code exécutable aussi bien pour l'hôte que pour le périphérique accélérateur. Cette approche permet de conserver une base de code scientifique unique, indépendante des particularités du jeu d'instructions de l'architecture ciblée.

Contrairement à une approche GPU native explicite, l'envoi des instructions de calcul au GPU est ici entièrement pris en charge par le modèle de programmation de Kokkos. Le code utilise des foncteurs ou des lambdas C++ standards, qui sont ensuite distribués sur les unités de calcul via des primitives de parallélisme telles que `Kokkos::parallel_for`. La librairie se charge de traduire ces appels en lancements de noyaux CUDA ou en threads OpenMP selon la configuration choisie, et calcule automatiquement la taille des grilles et des blocs afin de maximiser l'occupation du matériel. De même, la gestion des données repose sur les conteneurs `Kokkos::View`, qui abstraient l'allocation mémoire et garantissent un agencement des données (par exemple *LayoutLeft* ou *LayoutRight*) adapté à l'architecture sous-jacente.

3 Résultats

Avant de présenter les résultats, par souci d'honnêteté intellectuelle, il est bon de préciser que les tests Séquentiels/Metal et Kokkos OpenMP/Cuda n'ayant pas été effectués sur la même machine, il est difficile d'en tirer des conclusions définitives. En effet, les composants de la machine Rhum ont plus de 10 ans, et ne sont donc pas nécessairement au niveau d'une puce datant de 2025 (bien que le GPU Tesla K80 soit tout de même très performant, et surpassant même sur certains points la partie graphique de la puce M4 avec quasiment 5000 coeurs Cuda contre un peu moins de 1300 ALU pour le SoC d'Apple).

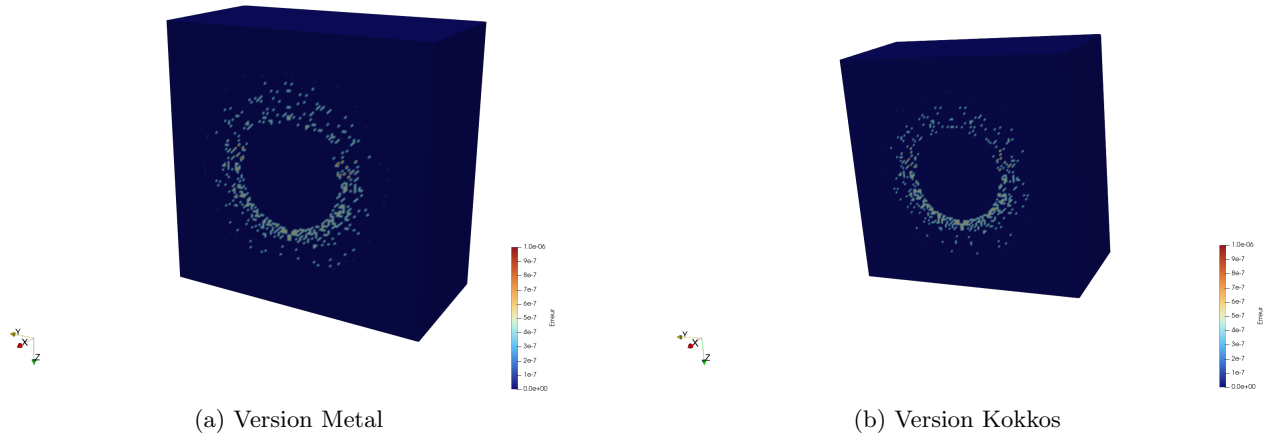


Figure 2: Erreur commise avec les codes Metal et Kokkos par rapport au code séquentiel

On commence par vérifier que les résultats que l'on a obtenu avec chacune des deux versions est correct par rapport au code séquentiel. Pour ce faire, j'ai utilisé le logiciel Paraview qui permet de visualiser les fichiers .vtk produits par les différents codes. Il permet notamment de pouvoir faire la différence entre deux fichiers afin de directement observer l'erreur commise. On fournit la comparaison Séquentiel/Metal ainsi que Séquentiel/Kokkos dans la Figure 2. À noter que comme l'architecture de la partie graphique de la puce M4 ne permet pas nativement de faire des calculs avec des flottants double précision, j'ai ici comparé uniquement les codes en simple précision. On remarque sur les figures une légère erreur sur le contour de la sphère, de l'ordre de $10^{-6}/10^{-7}$, ce qui est cohérent avec l'utilisation de float à la place de double. On peut donc être satisfait de la qualité des résultats obtenus pour les versions Metal et Kokkos.

Taille de grille	Séquentiel	Metal	Kokkos OMP(Rhum)	Kokkos Cuda(Rhum)
64^3	0.14 s	0.1 s	0.11 s	0.17 s
128^3	0.66 s	0.12 s	0.80 s	0.22 s
256^3	5.3 s	0.39 s	5.2 s	0.80 s
512^3	43 s	2.3 s	74 s	4.4 s

Table 1: Benchmark des différentes versions du solveur Poisson pour différentes tailles de grille

Passons maintenant aux performances : on fournit le tableau 1 qui regroupe les temps totaux d'exécution pour toutes les versions, pour différentes tailles de maillage. Comme précisé au début de la section, comme nous avons effectué les tests sur des machines différentes, nous allons surtout nous concentrer sur les comparaisons Séquentiel/Metal et Kokkos OpenMP/Cuda:

- Dès les plus petites tailles de grilles, on observe un net avantage pour la version Metal du solveur, avantage qui ne fait que s'accroître à mesure que la grille grandit (pour une grille 512^3 , le code Metal est environ 20x plus rapide que le code séquentiel).
- Pour la comparaison entre Kokkos OpenMP et Cuda, le résultat est sans appel : Cuda surpasse largement OpenMP, et ce malgré les écritures de données à faire entre le CPU et le GPU.
- À noter que les performances entre Kokkos Cuda et Metal restent tout de même assez proches malgré l'âge qui sépare les deux architectures, avec tout de même un avantage pour Metal qui bénéficie de la mémoire partagée entre le CPU et le GPU, et qui lui permet en particulier d'avoir autant de mémoire vidéo que sur la Tesla K80.

Conclusion

Ce projet m'a permis d'explorer concrètement la programmation sur GPU à travers deux approches complémentaires : une implémentation spécifique à l'écosystème Apple via Metal, et une implémentation portable grâce à Kokkos. Les deux versions reproduisent correctement la solution de référence, avec une erreur relative de l'ordre de $10^{-6}/10^{-7}$ en simple précision, ce qui confirme la validité numérique des adaptations réalisées.

Sur le plan des performances, les tests montrent un gain très significatif par rapport à la version séquentielle, en particulier pour la version Metal, qui profite de la mémoire unifiée de la puce Apple Silicon M4. Du côté

de Kokkos, la comparaison entre l'exécution OpenMP et CUDA met clairement en évidence l'intérêt du GPU pour ce type de problème, malgré le surcoût potentiel des transferts de données.

Au-delà des chiffres, ce travail illustre l'importance de structurer le code pour séparer la logique scientifique de la gestion du matériel, que ce soit via une API bas niveau comme Metal ou via un modèle de programmation abstrait comme Kokkos. Cette séparation facilite la portabilité, la maintenance et l'évolution du code vers de nouvelles architectures. À plus long terme, ce type d'approche ouvre la voie à des solveurs plus complexes, capables de tirer parti efficacement des accélérateurs disponibles sur des plateformes hétérogènes.