DE LA RECHERCHE À L'INDUSTRIE

PTC-INCOME

Intégration de lois de Comportement Mécanique sur GPU

Salem Khellal

www.cea.fr

6 Novembre 2023

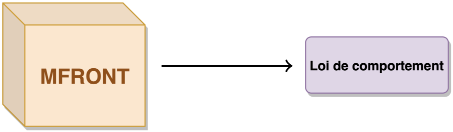
Maison de la Simulation

Loi de comportement

Loi de comportement

Relation mathématique qui décrit comment un matériau solide réagit aux charges, déformations et contraintes.

Mise en contexte



Loi de comportement

Relation mathématique qui décrit comment un matériau solide réagit aux charges, déformations et contraintes.

MFRONT

Générateur de code simplifiant l'écriture de propriétés matériaux, lois de comportement mécanique et modèles physico-chimiques.

Loi de comportement

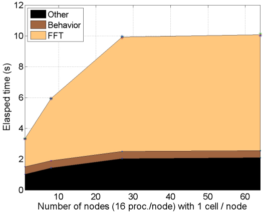
Relation mathématique qui décrit comment un matériau solide réagit aux charges, déformations et contraintes.

MFRONT

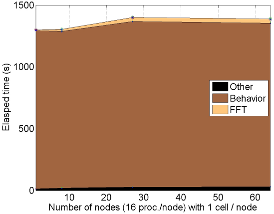
Générateur de code simplifiant l'écriture de propriétés matériaux, lois de comportement mécanique et modèles physico-chimiques.

AMITEX

Solveur à base de Transformées de Fourier rapide (FFT).



Loi simple



Loi complexe

Objectifs

- Réaliser le portage sur GPU des lois de comportement dans MFRONT avec différents Programming Models (CUDA / Kokkos / SYCL)
- Proposer une solution flexible et performante sur GPU pour toutes les lois de comportement.

Points d'intégration

- Calculs indépendants entre chaque point d'intégration (embarassingly parallel) $\rightarrow$ Utilisation de functors
- Possibilité de déséquilibrage de charge entre les points d'intégration

Structures de données avec MGIS/MFRONT

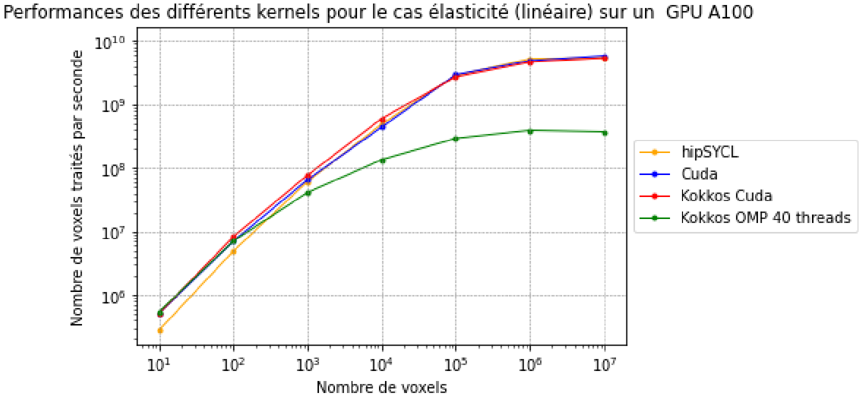
- Transferts mémoires CPU/GPU
- Gérer les structures propres à MGIS et à MFRONT dans le portage GPU
- Passage AOS (Array of Structs) vers SOA (Struct of Arrays) $\rightarrow$ rendre les accès mémoires coalescents

La loi de Hooke

Calcul du tenseur des contraintes σ à partir du tenseur des déformations ϵ dans chaque point d'intégration avec la relation suivante (cas linéaire):

$$\underline{\boldsymbol{\sigma}} = \lambda \mathrm{trace}(\underline{\epsilon})\mathbf{I} + 2\mu\underline{\boldsymbol{\epsilon}}$$

- λ est le coefficient de Lamé
- μ est le module de cisaillement

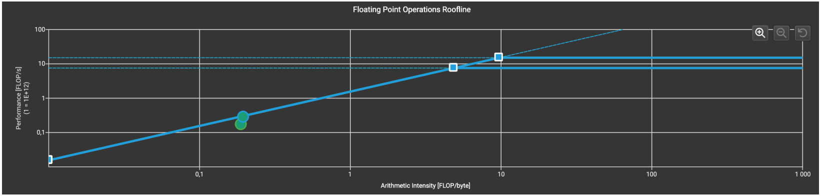


Performances des différents kernels pour le cas élasticité (linéaire) sur un GPU A100
Nombre de voxels traités par seconde
10^{10}
10^9
10^8
10^7
10^6
10^1
10^2
10^3
10^4
10^5
10^6
10^7
Nombre de voxels
hipSYCL
Cuda
Kokkos Cuda
Kokkos OMP 40 threads

Comparaison de performances pour la loi élasticité avec 10^7 voxels

	Kokkos OMP (CPU)	Kokkos CUDA (GPU)	CUDA	hipSYCL
Kernel	0.03s	0.002s	0.002s	0.002s
H $\rightarrow$ D	-	0.16s	0.16s	0.21s
D $\rightarrow$ H	-	0.11s	0.15s	0.15s

- x10 speedup pour l'exécution du kernel,
- mais le transfert memoire est pénalisant sur GPU.



Avec des données coalescents, on est beaucoup plus proche du roof (performance maximale théorique).

Plasticité isotrope

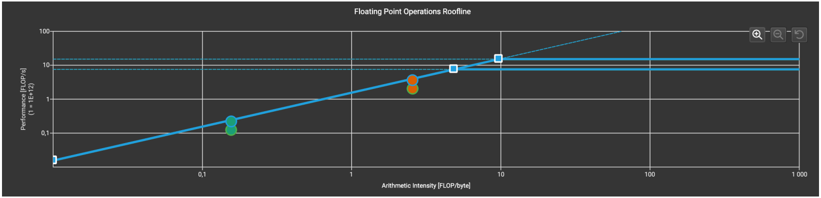
Résolution d'une equation non-lineaire en utilisant un algorithme de Newton-Raphson scalaire en chaque point d'intégration:

$$\begin{cases} \underline{\boldsymbol{\epsilon}}^{to} = \underline{\boldsymbol{\epsilon}}^{el} + \underline{\boldsymbol{\epsilon}}^{p} \\ \underline{\boldsymbol{\sigma}} = \underline{\boldsymbol{D}} : \underline{\epsilon}^{el} \\ \underline{\dot{\boldsymbol{\epsilon}}}^{p} = \dot{p}\underline{n} \\ f(\sigma_{eq}, p)\dot{p} = 0, avec \quad f(\sigma_{eq}, p) \leq 0 \end{cases}$$

- $\underline{\boldsymbol{\epsilon}}^{to}, \underline{\boldsymbol{\epsilon}}^{el}, \underline{\boldsymbol{\epsilon}}^{p}$ sont respectivement les déformations totale, élastique et plastique
- $\underline{n} = \frac{3}{2}\frac{\underline{s}}{\sigma_{eq}}$ est la direction d'écoulement
- $\underline{s}$ est le déviateur des contraintes
- σ_{eq} est la norme de Von Mises.

	Kokkos OMP (CPU)	Kokkos CUDA (GPU)	CUDA	hipSYCL
Kernel	0.09s	0.002s	0.002s	0.003s
H $\rightarrow$ D		0.15s	0.18s	0.21s
D $\rightarrow$ H		0.12s	0.17s	0.17s

- x45 speedup pour l'exécution du kernel.



Avec des données coalescents, on est beaucoup plus proche du roof comme dans le cas précédent .

La loi Norton

Résolution d'un système non-lineaire en utilisant un algorithme de Newton-Raphson vectoriel en chaque point d'intégration:

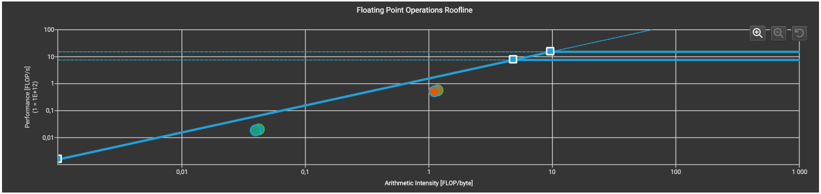
$$\begin{cases} \underline{\boldsymbol{\epsilon}}^{to} = \underline{\boldsymbol{\epsilon}}^{el} + \underline{\boldsymbol{\epsilon}}^{vis} \\ \underline{\boldsymbol{\sigma}} = \underline{\underline{\boldsymbol{D}}} : \underline{\boldsymbol{\epsilon}}^{el} \\ \dot{\underline{\boldsymbol{\epsilon}}}^{vis} = \dot{p}\underline{n} \\ \dot{p} = A\sigma_{eq}^{m} \end{cases}$$

- $\underline{\boldsymbol{\epsilon}}^{to}, \underline{\boldsymbol{\epsilon}}^{el}, \underline{\boldsymbol{\epsilon}}^{vis}$ sont respectivement les déformations totale, élastique et visqueuse
- $\underline{n} = \frac{3}{2}\frac{\underline{s}}{\sigma_{eq}}$ est la direction d'écoulement
- $\underline{s}$ est le déviateur des contraintes
- σ_{eq} est la norme de Von Mises.

Comparaison de performances pour la loi Norton avec 10^7 voxels

	Kokkos OMP (CPU)	Kokkos CUDA (GPU)	CUDA	hipSYCL
Kernel	1.60s	0.29s	0.30s	0.33s
H $\rightarrow$ D		0.05s	0.05s	0.08s
D $\rightarrow$ H		0.05s	0.05s	0.07s

- x5 speedup pour l'exécution du kernel.
- Réduction de la quantité des données à copier à chaque itération $\rightarrow$ Overhead lié au transdert mémoire devient moins prononcé



Dans ce cas plus complexe, rendre les données coalescents ne permet pas de vraiment se rapprocher de la limite.

Newton vectoriel

Dans un algorithme de Newton vectoriel, on résout un système lineaire à chaque itération lorsque c'est possible:

$$F'(x_k)(x_{k+1} - x_k) = -F(x_k)$$

- $F'(x_k)$ la matrice Jacobienne à l'itération k
- x_k le vecteur contenant les variables internes à l'itération k

Hypothèse 1: Optimiser la décomposition LU ?

La Décomposition LU est l'étape qui prend le plus de temps.
En testant un kernel qui réalise uniquement la décomposition LU, on a constaté avec une Roofline que ce n'est pas cette étape le problème.

Hypothèse 2: Grand nombre de registres par thread

Dans un algorithme de Newton vectoriel, on résout un système lineaire à chaque itération lorsque c'est possible:

$$F'(x_k)(x_{k+1} - x_k) = -F(x_k)$$

- $F'(x_k)$ la matrice Jacobienne à l'itération k nécessite le stockage de 12x12 double
- x_k le vecteur contenant les variables internes à l'itération k nécessite le stockage de 12 double

Dans l'algorithme actuel toutes ces données sont stockées dans la pile $\rightarrow$ Grand nombre de registres = problème sur GPU
Idée: déplacer le stockage de ces données vers la mémoire globale

Conclusion

Performances

- Loi simple (élasticité , plasticité): CPU $>$ GPU
- Loi plus complexe (Norton): GPU $>$ CPU

Perspectives

- Poursuivre l'optimisation du cas Norton (diminution du nombre de registres par thread)
- Tester les cas avec déséquilibrage de charge en intégrant AMITEX
- Tester d'autres lois plus complexes

Contributeurs

Merci pour votre contribution au PTC !
Thomas Helfer
Lionel Gélébart
Raphael Prat
Yushan Wang