Adaptation automatique de codes de calcul mécanique pour matériels hétérogènes

Hugo Leclerc

Adaptation automatique de codes de calcul mécanique pour matériels hétérogènes

H. Leclerc[1]

[1] LMT-Cachan (ENS Cachan/CNRS/UPMC/PRES UniverSud Paris)
61 avenue du Président Wilson, F-94235 Cachan Cedex, France
leclerc@lmt.ens-cachan.fr

Résumé — Les évolutions récentes des architectures grand public (*many-core*, évolution puissance arithmétique / bande passante...) posent des problèmes majeurs d'implémentation pour les codes de calcul scientifique. Les modèles de programmation conventionnels sont aujourd'hui mis en défaut. L'approche présentée ici se base sur le concept de "librairie active" et exploite en conjonction l'évaluation paresseuse, le calcul symbolique et la génération automatique de code. Cette approche permet, en facilitant la *modularisation complète* et en *simplifiant* sensiblement l'implémentation (ex : gestion des discontinuités, ...), d'accéder à de *meilleurs temps d'exécution* ainsi qu'à un meilleur usage des mémoires.

Mots clés — GPU, calcul symbolique, discontinuïtés, génération automatique de code.

1 Contexte

Beaucoup des méthodes et des façons de programmer le calcul scientifique nous viennent d'un âge informatique où demander une assistance complexe de la part d'un compilateur était prohibé : la méta-programmation[1] était impraticable il y a 15 ans, de même que la Programmation Orientée Objet, même dans sa version dynamique[1], était impraticable il y a 25 ans. Il a pourtant été prouvé que ce dernier modèle de programmation permettait d'obtenir des gains significatifs, en terme de simplicité d'écriture et de modularité. Parmi les exemples de réussite dans ce domaine, on pourra citer ZéBuLon[2], OOFEM[3], GetFem++[4], etc...

Avec la montée en puissance des compilateurs, des outils d'assistance et des optimisations supplémentaires ont été rendues possibles. Il est par exemple possible de s'arranger pour que les compilateurs propagent des informations connues à la compilation pour générer un code adapté à chaque cas particulier sans que l'utilisateur n'ait à ré-écrire quoi que ce soit. Comme exemples triviaux mais significatifs, on pourra citer les adaptations automatiques sur la taille des vecteurs (ex : une fois choisi le nombre de dimensions de l'espace, le compilateur sera en mesure de connaître la taille des vecteurs position, déplacement, ...) ou celles concernant le type de scalaire (simple précision, complexe, etc...).

Dans un genre plus avancé, ces évolutions ont permis de développer le concept de *librairie active*[5]. Une librairie est dite active lorsqu'elle prend part au processus de compilation, permettant une forme supplémentaire d'adaptation automatique (par rapport à celle que les compilateurs fournissent déjà) : l'utilisateur aura *moins* de code à réecrire pour que son application contienne tel ou tel type d'optimisation, spécialisée pour tel ou tel type d'architecture par exemple.

[1] c'est à dire dans une version où les adaptations, fonction du type des objets, sont réalisées pendant l'exécution, permettant de décharger le compilateur.

De fait, toutes (ou presque) les librairies d'algèbre linéaire programmées en C++ utilisent le concept d'*expression template*[6] qui permet d'optimiser bon nombre d'opérations courantes sans changer le code. Exemple : lorsque l'utilisateur écrit $V = A + B + C$ plutôt que d'effectuer chaque opération de façon gloutonne (i.e. en faisant $T = A + B$, avec T un vecteur temporaire puis $T + C$), la librairie (et non le compilateur, qui en est bien incapable) repère qu'il vaut mieux utiliser $V_i = A_i + B_i + C_i$ ce qui permet de ne pas avoir à créer de vecteur temporaire et donc gérer mieux le cache, factoriser les instructions pour faire les boucles, ... Ces librairies cherchent les optimisations potentielles uniquement pendant le temps de compilation, c'est à dire que jamais elles ne nuisent au temps d'exécution. Quand les optimisations sont possibles les gains sont souvent considérables.

L'objet de ce papier est de montrer comment ce modèle a pu être étendu pour prendre en charge des aspects habituellement liés au calcul symbolique. Après quelques explication sur le schéma de pensée développé (section (2)), parmi les multiples conséquences, on abordera dans cette étude celles concernant la *simplicité d'écriture* (section (3)), notamment en présence de discontinuïtés (XFEM, ...). Dans la section (4), on s'intéressera à la *vitesse d'exécution* et en particulier avec utilisation de processeurs de cartes graphiques (GPU).

2 Évaluation paresseuse

Les propositions présentées ici se basent largement sur le concept d'*évaluation paresseuse* : l'exécution effective des taches est retardée au maximum, dans le but d'accumuler des informations habituellement inaccessibles aux compilateurs, mais nécessaires pour obtenir les meilleurs temps d'exécution, avec une utilisation minutieuse des mémoires. Ce concept est connu et utilisé notamment dans le domaine des langages fonctionnels[7], pour les listes infinies, le *copy-on-write*... Dans notre cas, jusqu'à quand les objets sont-ils utilisés, pourquoi, dans quel contexte matériel, ... sont des questions importantes pour générer du code de qualité, mais aussi pour distributer et ordonnancer correctement.

2.1 Première étape de simplification

Si, pour prendre un exemple simple, on souhaite calculer $V = MNV + OPU$ avec (M,N,O,P) des matrices et (V,U) des vecteurs, le langage permet d'obtenir le graphe de la figure 1.

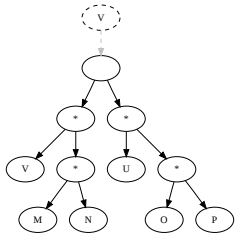

Figure 1 – Graphe généré automatiquement par l'écriture de `V = M*N*V + O*P*U`.

Une librairie ou un environnement standard (tel que MATLAB™, ...) calculerait *séquentiellement* l'expression de droite à gauche, en stockant MN dans une matrice temporaire puis multiplierait la matrice temporaire par V, etc... Mais dans ce cas, l'évaluation paresseuse permet automatiquement de repérer qu'il vaut mieux faire NV (qui donne un vecteur) puis ensuite multiplier M par le résultat.

2.2 Ordonnancement et répartition automatiques

L'évaluation paresseuse permet en plus de repérer automatiquement qu'il est possible de faire *MNV* et *OPU en parallèle*. L'ordonnanceur en tache de fond peut donc décider, fonction du contexte matériel et du positionnement des variables dans les mémoires, d'exécuter l'opération sur plusieurs threads (comme dans [8]) ou *plusieurs machines* (puisqu'on sait tranférer les données, grace à un langage introspectif). Dans ce cas, MPI est utilisé comme une librairie, mais pas comme un modèle de programmation : les étapes de transfert et de synchronisation, forcément dans un style "programmation impérative" sont gérées en interne.

Une des conséquences est qu'il devient possible d'utiliser les résultats très nombreux de la recherche en répartition et ordonnancement (cf. [9]). Il existe bon nombre d'heuristiques pour les problèmes d'algèbre linéaire [10]. Pour les problèmes type décomposition de domaine (où on souhaite répartir des sous-structures), la plupart des heuristiques fonctionnent sans difficultés. Ce point n'est pas détaillé dans ce papier.

En outre, certaines heuristiques fournissent des outils pour gérér efficacement la mémoire [11] quand elle devient une ressource rare, ou pour gérer les pannes ou les changements dynamiques de contexte [12].

2.3 Calcul symbolique

Les graphes générés fournissant des informations mathématiques, l'évaluation paresseuse permet d'obtenir un vision "symbolique" des programmes. Il devient possible automatiquement d'intégrer, de dériver, de substituer, de minimiser, etc... des résultats obtenus par des programmes de longueurs quasi-quelconques (cf. figure 2).

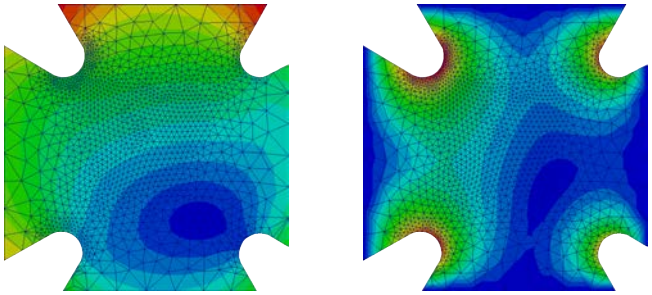

Figure 2 – Résultat et dérivée (par rapport au coefficient de Poisson) d'une simulation d'un essai élastique de bi-traction d'une eprouvette en composite. Le code pour la dérivée est obtenu automatiquement à partir d'un code qui n'était pas prévu à cet effet mais est optimal en terme d'efficacité, la matrice n'étant factorisée qu'une fois.

La description des éléments, des formulations et des loi de comportement s'en trouvent considérablement simplifiées et modularisées : on ne spécifie qu'un minimum d'informations (la position des points dans l'espace de référence, l'énergie libre non dérivée, ...) et la librairie pré-calcule le reste (les fonctions de formes, la matrice tangente, etc...). On pourra noter de façon générale que le calcul symbolique permet d'optimiser de façon interprocédurale une quantité significative de calcul : les traitements symboliques permettent des simplifications et des *transformations de formules* qu'aucun humain ne s'amuserait à faire, et ce, de façon automatique, fonction de l'architecture sous-jacente, et éventuellement en vue d'utiliser une architecture logicielle particulière (logiciel commercial, ...). Ces outils fournissent donc une base solide pour s'adapter aux architectures matérielles, même les plus récentes.

3 Une illustration sur la simplicité d'utilisation

Avec l'avénement des level-sets [13] et de l'enrichissement [14], les traitements des fonctions non polynômiales et notamment des discontinuïtes est devenu de première importance. Lorsque les éléments d'un maillage sont traversés par une unique discontinuïté, plusieurs auteurs (ex : [15]) ont proposé des solutions pour continuer à intégrer numériquement les matrices et les vecteurs dans une approche identique à celle qui est programmée dans la quasi-totalité des codes actuels (les poins "de gauss"). On propose ici d'intégrer symboliquement et automatiquement les résidus que l'utilisateur propose, avec un nombre quelconque de discontinuïtés par élément pour *ensuite* effectuer les transformations qui permettent d'obtenir les matrices et les seconds membres.

On constate simplement que (avec H la fonction de heaviside)

$$\int_{[a,b]} f(H(g(x)),x)\,dx = \int_{[a,b]\cap\{x|g(x)\geq 0\}} f(1,x)\,dx + \int_{[a,b]\cap\{x|g(x)<0\}} f(0,x)\,dx$$

Le moteur de calcul symbolique cherche donc les discontinuités dans les formules à intégrer (typiquement : une énergie dans une formulation). Il cherche ensuite les zéros des fonctions "$g(x)$" qui en pratique sont souvent des polynômes de degré 1 (dans le cas par exemple d'un level-set avec des tetrahèdres), pour diviser les intervalles d'intégration, substituer et appeler de nouveau les fonctions d'intégrations de façon récursive.

La discontinuité peut donc se situer à n'importe quel niveau (élément, formulation, ...) le moteur de calcul la repérera automatiquemene et regénérera l'intégralité du code nécessaire. Il est par exemple possible de générer un code adapté pour la visualisation en plus du code utile pour calculer les opérateurs.

3.1 Frontière mobile

Les dérivations étant faites après les intégrations, la gestion des frontières mobiles (de façon inconnue ou pas) est complétement automatique. La figure 3 en montre un exemple.

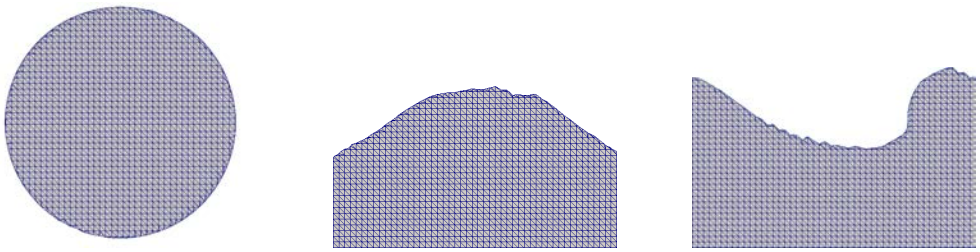

Figure 3 – Navier-stockes avec un level-set advecté. Les discontinuïtés des propriétés matériaux sont gérées automatiquement et l'advection est décrite informatiquement par `phi.diff( time ) + dot( vit, grad( phi ) )`.

3.2 Maillages non conformes

Dans ce cas, les discontinuïtés viennent des bords des éléments. Si `e0` et `e1` sont des éléments non conformes potentiellement en contact, écrire du code comme

```
energy := e1.integration( e0.point_is_inside( e1.pos ) *
     local_energy.subs( e0.var_inter, e0.var_inter_for_pos( e1.pos ) ) )
qe := quadratic_expansion( energy, e0.symbols +++ e1.symbols )
```

permet par exemple d'obtenir de façon symbolique, quelque soient le type des éléments, le nombre de dimensions de l'espace... et sans erreurs d'approximation, des matrices et vecteurs élémentaires de projection (type Mortar, ...).

4 Une illustration sur la vitesse d'exécution

Dans cette section, on s'intéresse au calcul de résidu pour un problème d'élasticité 3D.

4.1 CPU

Si on suppose la matrice K et le second membre F connus, on peut calculer le résidu en U avec $KU - F$. C'est la façon classique de procéder, mais qui conduit à une profonde sous-exploitation du matériel, surtout en présence de CPU multi-cœurs (cf. figure 4) : le bus de tranfert mémoire vers CPU sature rapidement. Une solution (utilisée d'habitude essentiellement pour conserver les ressources mémoires) pourrait consister à recalculer les termes de la matrice à la volée (ce qui revient simplement avec l'évaluation paresseuse à demander la dérivée de l'énergie). De façon classique, si on utilise les matrices gradients, les tenseurs de Hooke représentés par des matrices, ... et tout l'attirail conventionnel élément fini, les temps deviennent encore plus mauvais ($\simeq$10 fois plus lent voire pire).

Mais en utilisant le calcul symbolique (avec les simplifications automatiques) et la génération automatique de code, il est possible déja d'être sensiblement plus rapide sur 1 cœur mais aussi de récupérer un speedup de 63%. C'est le générateur de code qui choisit lui même les représentations intermédiaires. Sur 8 cœurs, il est possible d'être 14 fois plus rapide en laissant la librairie décider, qu'en forçant à utiliser $KU - F$ de façon conventionnelle.

4.2 GPU

Le générateur de code est par ailleurs capable de s'adapter aux GPUs (NVIDIA 8800 ultra dans notre cas). La figure 4 montre le résultat en terme de vitesse d'exécution (gain $\simeq$ 28 en simple précision).

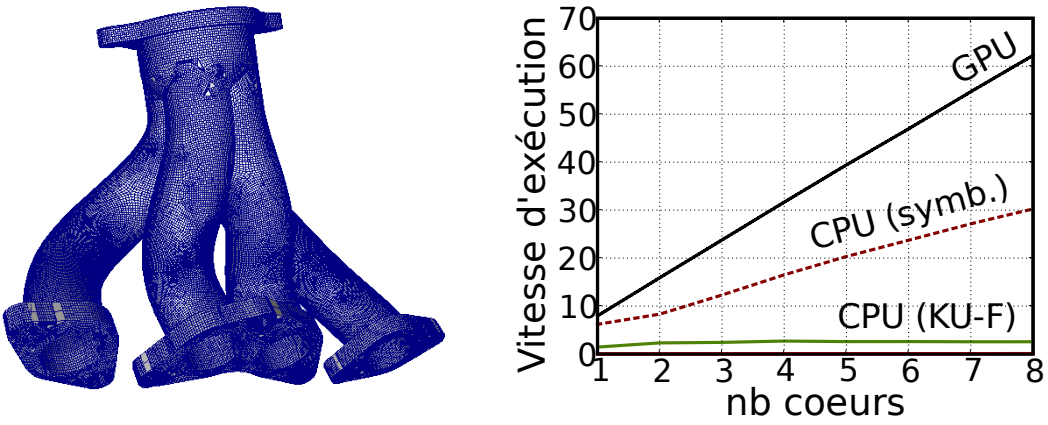

Figure 4 – Vitesse (relative à celle en utilisant $KU - F$ sur 1 cœur) avec ou sans calcul symbolique et GPU (simple précision). L'image de gauche représente le maillage test.

5 Conclusions

Un modèle de programmation adapté au développement de codes pour l'analyse numérique a été proposé. Ce modèle étend un concept présent par ailleurs dans les langages fonctionnels : l'évaluation paresseuse. Il permet automatiquement de gérer la répartition et l'ordonnancement pour le calcul parallèle et la gestion des mémoires, mais aussi de décrire les programmes de façon

symbolique. Les opérations associées traditionnellement aux CAS (Computer Algebraic System) sont accessibles et peuvent fonctionner sur un programme entier avec une efficacité optimale.

Les illustrations ont permis de montrer les conséquences en terme de simplicité et de modularité, mais aussi en terme de vitesse d'exécution, en s'adaptant automatiquement aux architectures matérielles exotiques ou contemporaines.

Références

[1] Todd L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran ? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[2] Frédéric Feyel. Some new technics regarding the parallelisation of zébulon, an object oriented finite element code for structural mechanics. *Mathematical Modelling and Numerical Analysis*, 36(5) :923–935, sep 2002.

[3] B. Patzák and Z. Bittnar. Design of object oriented finite element code. *Advances in Engineering Software*, 32(10-11) :759–767, 2001.

[4] Yves Renard and Julien Pommier. *GetFem++, Short User Documentation*.

[5] Todd L. Veldhuizen and Dennis Gannon. Active libraries : Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[6] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5) :26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[7] H. Conrad Cunningham, Yi Liu, and Hui Xiong. Lazy functional programming in haskell tutorial presentation.

[8] T. Gautier, R. Revire, and F. Zara. Efficient and easy parallel implementation of large numerical simulation. 2003.

[9] Oliver Sinnen. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[10] Olaf Schenk and Klaus Gärtner. Two-level dynamic scheduling in pardiso : Improved scalability on shared memory multiprocessing systems. *Parallel Computing*, 28(2) :187 – 197, 2002.

[11] Emmanuel Agullo. *On the Out-of-core Factorization of Large Sparse Matrices*. PhD thesis, École Normale Supérieure de Lyon, November 2008.

[12] T. Gautier and H.R. Hamidi. Automatic re-scheduling of dependencies in a rpc-based grid. 2004.

[13] Stanley Osher, Stanley Osher, Ronald P. Fedkiw, and Ronald P. Fedkiw. Level set methods : An overview and some recent results. *J. Comput. Phys*, 169 :463–502, 2001.

[14] N. Moes, A. Gravouil, and T. Belytschko. Non-planar 3d crack growth by the extended finite element and level sets part i : Mechanical model. *International Journal for Numerical Methods in Engineering*, 53(11) :2549–2568, 2002.

[15] T. Gautier and H.R. Hamidi. Automatic re-scheduling of dependencies in a rpc-based grid. 2004.