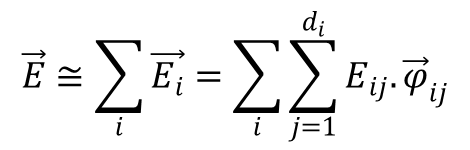
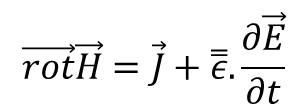
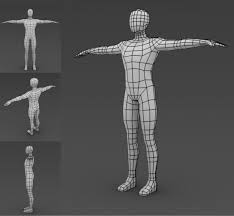
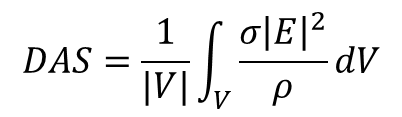
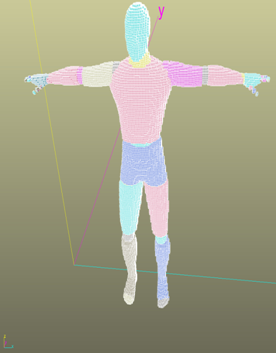
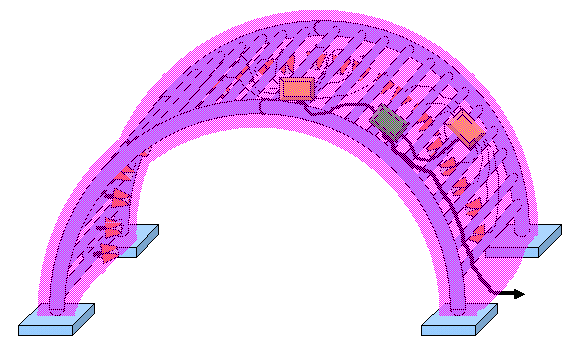
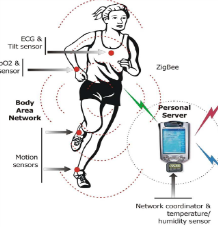
*Utilisation des H****PC pour*** *l’O****ptimisation des*** *R****adiocommunications des*** *O****bjets*** *C****onnectés***

*proches de l’H****omme***

CONVENTION N°P5344



HPC



##### PROJET HOROCH

##### **RAPPORT TECHNIQUE**

« T121 Rapport d’étude et d’évaluation des performances sur la parallélisation hybride»

Résumé : Ce rapport présente les premiers résultats obtenus sur l’optimisation d’un code DG au moyen du parallélisme par tâches. Nous passons d’abord en revue quelques outils de parallélisation automatique ou non. Ensuite nous présentons des résultats d’optimisation de la méthode DG au moyen du support d’exécution StarPU. Dans le cas d’un schéma implicite, les résultats sont très satisfaisants. Pour le schéma explicite, le code a été validé, mais des optimisations restent à faire afin d’obtenir les performances attendues.

**Evolutions**

|  |  |
| --- | --- |
| INDICE DE REVISION | DESCRIPTION |
| -- | Création du document pour diffusion le 30/09/2015 |
| -A |  |
| -B |  |
| -C |  |
| -D |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ind + Date | --  30/09/15 | -A | -B | -C | -D |
| Etabli par |  |  |  |  |  |
| Vérifié par |  |  |  |  |  |
| Vérifié par RT | A. Guena |  |  |  |  |
| Approuvé par PM | M. Viguier |  |  |  |  |

[1.Principes du parallélisme par tâches 4](#__RefHeading___Toc984_1470904692)

[1.1 Généralités 4](#__RefHeading___Toc998_1470904692)

[1.2 Outils existants 4](#__RefHeading___Toc1000_1470904692)

[1.2.1 OpenMP 4](#__RefHeading___Toc1002_1470904692)

[1.2.2 POCC, PLUTO 5](#__RefHeading___Toc1004_1470904692)

[1.2.3 Autres outils 6](#__RefHeading___Toc1006_1470904692)

[1.3 Description de StarPU 6](#__RefHeading___Toc1008_1470904692)

[2.Exemple d’application : schéma DG implicite 7](#__RefHeading___Toc986_1470904692)

[2.1 Schéma DG implicite pour le transport 7](#__RefHeading___Toc1010_1470904692)

[2.2 Graphe de dépendance 8](#__RefHeading___Toc1012_1470904692)

[2.3 Implémentation StarPU 9](#__RefHeading___Toc1014_1470904692)

[2.4 Résultats 10](#__RefHeading___Toc1016_1470904692)

[3.Application à DG explicite 11](#__RefHeading___Toc988_1470904692)

[3.1 Implémentation OpenMP 11](#__RefHeading___Toc1018_1470904692)

[3.2 Implémentation StarPU 12](#__RefHeading___Toc1020_1470904692)

[4.Résultats 12](#__RefHeading___Toc990_1470904692)

[4.1 Comparaison OpenMP/OpenCL 12](#__RefHeading___Toc1022_1470904692)

[4.2 Comparaisons C/StarPU-C 13](#__RefHeading___Toc1024_1470904692)

[4.3 Comparaisons StarPU-C/StarPU-C+OpenCL 14](#__RefHeading___Toc1026_1470904692)

[5.Optimisations OpenCL CPU 15](#__RefHeading___Toc992_1470904692)

[5.1 Axes d’optimisation 15](#__RefHeading___Toc1028_1470904692)

[5.2 Résultats 15](#__RefHeading___Toc1030_1470904692)

[6.Conclusion et perspectives 17](#__RefHeading___Toc994_1470904692)

[Références 17](#__RefHeading___Toc996_1470904692)

# Principes du parallélisme par tâches

## 1.1 Généralités

Actuellement, la plupart des codes de calculs intensifs exploitant les supercalculateurs reposent sur les technologies OpenMP et MPI. OpenMP est utilisé pour attaquer le parallélisme à l’intérieur d’un nœud de calcul, qui est généralement constitué de plusieurs CPU multicœurs partageant une mémoire commune. MPI permet d’échanger des données entre les nœuds de calcul.

Ce découpage repose sur l’hypothèse de calculs très uniformes dont il est possible de connaître à l’avance le coût élémentaire. Il faut aussi que les échanges de données soient relativement faibles comparés au volume des calculs. Par ailleurs, la stratégie est efficace pour une résolution qui n’utilise pas de marche en temps locale, qui repose sur des maillages relativement uniformes et des calculs qui ne nécessitent pas trop de traitements particuliers (par exemple des calculs ou des transferts de données auxiliaires).

La conception actuelle de Teta repose sur cette hypothèse. La seule différence avec l’approche OpenMP/MPI est que dans Teta, le parallélisme à l’intérieur du noeud est réalisé au moyen d’OpenCL afin de pouvoir utiliser n’importe quel type d’accélérateurs, coeurs de CPU aussi bien que des GPU.

Actuellement, de nombreux outils ont été développés par les chercheurs en informatique pour rendre la parallélisation de codes plus facile. Il existe des logiciels qui permettent par exemple de produire automatiquement du code OpenMP à partir d’un code C séquentiel. Il existe aussi des outils, appelés “supports d’exécution” ou “runtime” qui permettent de soumettre des tâches de calcul à une file d’attente. C’est le runtime qui se charge alors de distribuer en parallèle les tâches sur les accélérateurs en fonction des dépendances entre les tâches. Dans les systèmes les mieux conçus, c’est le runtime qui calcule les dépendances entre les tâches.

Le but de cette partie du projet a été d’évaluer divers types outils automatiques afin de mettre en œuvre une parallélisation hybride.

## 1.2 Outils existants

Dans ce paragraphe, on liste quelques outils automatiques. Il est impossible d’être exhaustif, car c’est un domaine où les développements sont très actifs à la fois en termes de recherche et de développement technologique.

### 1.2.1 OpenMP

OpenMP est un standard qui est aujourd’hui très répandu dans le domaine du HPC. Il est intéressant pour le développeur, car il permet d’introduire du parallélisme de manière incrémentale dans un code séquentiel déjà existant au moyen de “directives” : il suffit d’annoter le code là où il doit être parallélisé. Si le compilateur n’est pas compatible avec OpenMP, les annotations sont tout simplement ignorées, car elles sont considérées comme de simples commentaires. La méthode la plus classique consiste à vectoriser les boucles dont chaque terme peut-être calculé indépendamment des autres. Il existe aussi des directives pour calculer des réductions parallèles (somme des termes d’un vecteur, par exemple). Depuis les normes 3 et 4 d’OpenMP il est également possible de décrire des tâches et leurs dépendances. Depuis OpenMP 4 il est possible de planifier des tâches sur des accélérateurs de type GPU. HMPP et OpenACC sont d’autres outils basés sur des directives qui permettent d’accéder à des GPU.

Avec OpenMP il est en général assez facile d’accélérer un code d’un facteur deux à dix. Mais au-delà de dix cœurs, la gestion des accès mémoire devient très importante. Il faut tenir compte de ces accès et une réorganisation du code devient nécessaire. Par ailleurs la coexistence de deux logiques différentes, séquentielle et parallèle au sein d’un même programme rend rapidement le code source difficile à lire. Comme pour tout développement de code parallèle, le débogage est délicat, car c’est au programmeur de gérer les conflits d’accès aux données.

En conclusion, OpenMP semble offrir le standard le plus stable à moyenne et longue échéance pour décrire du parallélisme. Mais modifier incrémentalement un code séquentiel permet rarement une accélération optimale. De toute façon, le parallélisme doit être intégré à la conception initiale du logiciel. Selon la littérature que nous avons pu consulter, la parallélisation par directive permet d’accélérer d’un facteur de l’ordre de 4 un code en déportant une partie des calculs sur GPU. Mais le facteur limitant est imposé par les transferts CPU/GPU et les accès mémoire.

Lorsque l’on vise des performances très élevées, il n’est donc pas tellement plus efficace d’utiliser OpenMP plutôt qu’un langage dédié comme OpenCL ou StarPU.

### 1.2.2 POCC, PLUTO

Il existe depuis quelques années des outils qui permettent de générer du code OpenMP à partir d’un code C séquentiel. Nous avons évalué deux de ces outils :

* POCC “Polyhedral Compiler Collection” [1](file:///home/helluy/schnaps/old_schnaps/doc/temp/horoch_juin20162.html" \l "fn1x0)
* PLUTO “An automatic parallelizer and locality optimizer” [2](file:///home/helluy/schnaps/old_schnaps/doc/temp/horoch_juin20163.html" \l "fn2x0)

Lors des tests (début 2015) il est apparu que ces deux outils ne sont pas capables de générer du code OpenMP à partir du code source de Teta. L’analyseur de PLUTO par exemple s’arrête ou ne détecte pas de parallélisme. Nous sommes allés discuter avec un des développeurs de PLUTO (V. Loechner) qui est un collègue informaticien de l’Université de Strasbourg. Sur un exemple que nous lui avons fourni, il a été capable de faire fonctionner PLUTO. Mais pour cela il a dû restructurer notre exemple de code et modifier dans un second temps le code OpenMP généré par PLUTO. Et comme souvent, plus que le parallélisme des calculs, c’est la réorganisation des données en mémoire (par une technique de “tiling” pour améliorer l’utilisation de la mémoire cache) qui a permis d’obtenir les performances optimales. L’intervention et l’expertise de l’utilisateur restent donc très importantes. Il faudrait cependant tester à nouveau PLUTO, car c’est un logiciel qui évolue rapidement. Selon son développeur principal de PLUTO (Uday Bondhugula, rencontré en juin 2016) la nouvelle version est beaucoup plus facile à utiliser.

### 1.2.3 Autres outils

Il existe d’autres outils de parallélisation automatiques ou semi-automatiques que nous n’avons pas eu le temps d’évaluer. Il est important de suivre les développements dans ce domaine, qui sont fréquents et rapides. Cependant il faut aussi avoir à l’esprit que certains outils disparaissent aussi vite qu’ils sont apparus. Dans ce domaine, le manque de normes stables semble être la norme !

* Intel TBB : c’est une bibliothèque C++ pour décrire des tâches parallèles. Pour l’instant TBB n’adresse que les CPU multicoeurs.

## 1.3 Description de StarPU

StarPU fait l’objet d’une description à part, car c’est l’outil actuel qui nous a semblé le plus intéressant pour le développement d’applications hybrides.

Depuis quelques années, l’Inria développe le runtime StarPU <http://starpu.gforge.inria.fr/>. C’est une bibliothèque qui permet de décrire un calcul par tâche. Pour décrire une tâche, le programmeur fournit une ou plusieurs programmations de cette tâche dans des morceaux de code source appelés “codelettes”. Certaines codelettes peuvent utiliser CUDA ou OpenCL pour s’exécuter sur GPU. Le programmeur doit aussi décrire quels sont les paramètres constants de cette tâche ainsi que les données d’entrée et de sortie. Les tâches sont soumises à StarPU dans un ordre séquentiel. Le support d’exécution analyse en temps réel les dépendances entre les tâches et détermine celles qui peuvent être réalisées en parallèle. StarPU est aussi capable de choisir à l’exécution quelles sont les codelettes les plus efficaces. Plusieurs méthodes d’ordonnancement peuvent être testées. StarPU gère également les copies temporaires et les transferts mémoire afin d’améliorer les performances.

L’approche proposée par StarPU est extrêmement intéressante pour plusieurs raisons :

* Elle permet au développeur de continuer à penser à son algorithme de façon séquentielle ;
* mais l’incite aussi à découper ses données et ses calculs en pièces élémentaires qui vont permettre le parallélisme.
* L’amélioration incrémentale reste possible en ajoutant au fur et à mesure des codelettes pour GPU ou des codelettes optimisées avec OpenMP.
* StarPU propose de nombreux outils d’analyse et de “profiling” du code : diagrammes de Gantt, graphe des tâches local, activation ou non des CPU, des GPU, etc.
* Il est possible d’essayer plusieurs stratégies différentes d’ordonnancement et même de décrire son propre ordonnanceur.
* StarPU existe aussi en version MPI pour soumettre le graphe des tâches à un supercalculateur ou à un gros cluster de CPU multicoeur.

Actuellement, à notre connaissance, il n’existe pas d’outil de ce type possédant toutes ses caractéristiques [3](file:///home/helluy/schnaps/old_schnaps/doc/temp/horoch_juin20164.html" \l "fn3x0) . Aussi il nous a semblé important d’évaluer et d’exploiter cet outil.

# Exemple d’application : schéma DG implicite

Dans cette partie, nous décrivons les tests que nous avons réalisés avec StarPU pour la parallélisation d’un algorithme DG implicite de l’équation de transport. Nous avons choisi ce schéma, car sa parallélisation est très difficile en utilisant uniquement MPI ou OpenMP. En effet, à cause des dépendances, au début du calcul seule une tâche est active. Au fur et à mesure de la progression, le calcul devient de plus en plus parallèle. Le parallélisme se réduit à nouveau la fin du calcul. Cet exemple permet de bien comprendre les mécanismes en jeu dans StarPU et d’évaluer l’efficacité de l’outil.

## 2.1 Schéma DG implicite pour le transport

Nous considérons le schéma implicite suivant pour résoudre une équation de transport



où l’inconnue est une fonction *f*(*x,t*) qui dépend d’une variable spatiale tridimensionnelle *x* et du temps *t*. La vitesse *v* est supposée constante, pour simplifier. Cette équation constitue le système hyperbolique le plus simple possible. Teta et SCHNAPS peuvent résoudre n’importe quel système hyperbolique et les équations de Maxwell sont un cas particulier.

Nous considérons un maillage constitué de “macrocellules” hexagonales, chaque macrocellule étant elle-même découpée en sous-cellules de taille plus petite. Dans chaque sous-cellule, nous considérons une base de fonctions polynomiales de Lagrange *ψiL*.

L’équation est approchée par le schéma suivant : pour toute sous-cellule *L* et toute fonction de base *i*,

|  |  |
| --- | --- |
|  | (1) |

où :

* *fn* désigne la solution au temps *t* *n* = *n*Δ*t* et *fn*−1 la solution au temps précédent ;
* *R* désigne la cellule voisine de *L* le long de sa frontière ∂*L* (voir Figure [1](#x1-90021)) ;
* *v* ⋅ *n*+ = max(*v* ⋅ *n,* 0)*, v* ⋅ *n*− = min(*v* ⋅ *n,* 0)*.*
* *nLR* désigne le vecteur normal unitaire sur ∂*L* orienté de *L* vers *R*.

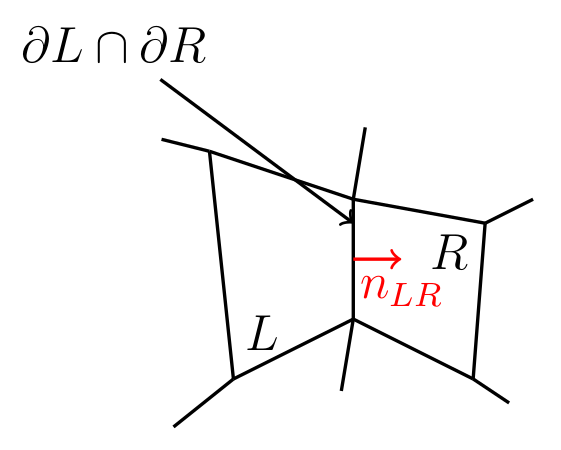


FIGURE 1: Maillage : conventions de notation.

Théoriquement pour calculer *fn* à partir de *fn*−1 en appliquant le schéma DG ([1](#x1-9001r1)) il faudrait résoudre un système linéaire ce qui peut-être très coûteux en 3D. Heureusement en utilisant deux propriétés du schéma DG : discontinuité de la solution numérique et décentrage du flux, nous constatons qu’il est possible de résoudre le système linéaire de proche en proche en parcourant le maillage dans le sens de la vitesse *v*. Plus nous avançons dans le maillage, plus le calcul devient parallèle.

## 2.2 Graphe de dépendance

Afin d’expliquer ce fait, nous considérons le maillage décrit dans la figure [2](#x1-100012) où seules les macrocellules sont représentées. Pour calculer *fn* dans la macrocellule 0, il suffit d’appliquer les conditions aux limites. Ensuite, nous pouvons calculer en parallèle les valeurs de *fn* dans les cellules 1 et 3 à partir des résultats de la cellule 0, etc. Les dépendances des calculs peuvent être schématisées par un graphe orienté également indiqué sur la figure [2](#x1-100012).

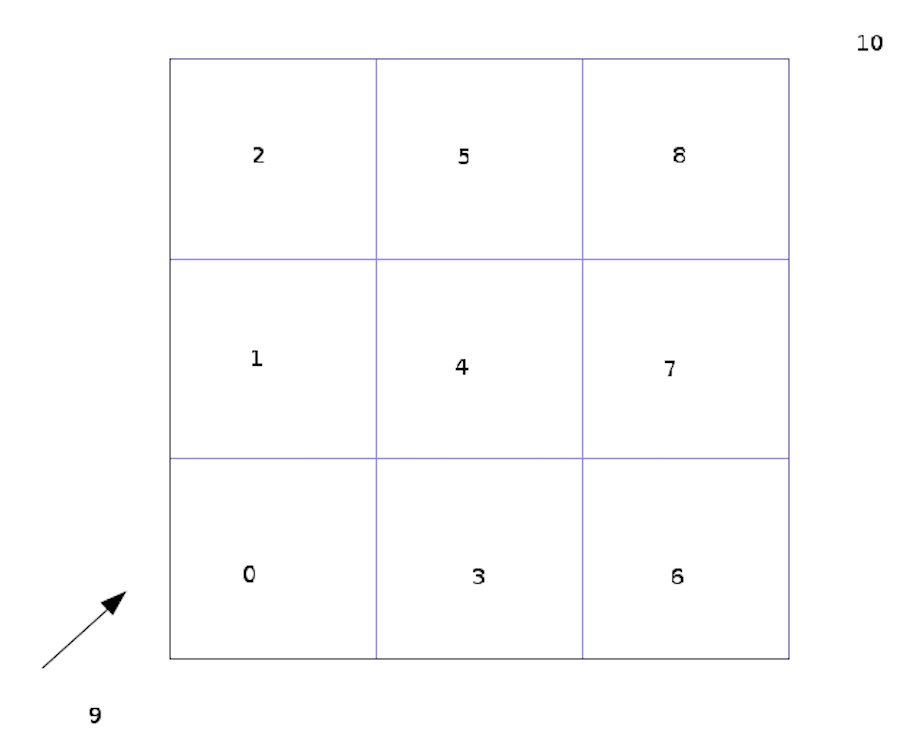
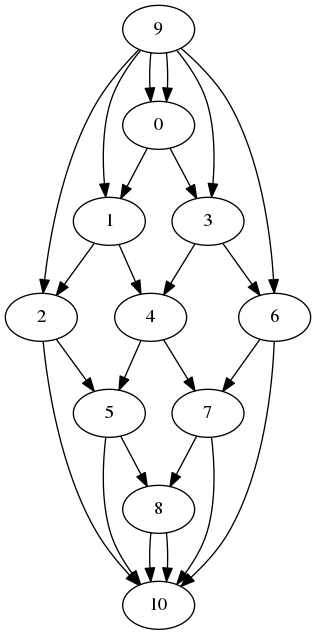
 

FIGURE 2: Parcours d’un maillage dans le sens de la vitesse. À gauche : un exemple de maillage et de vitesse. À droite : graphe de dépendance correspondant. Cette construction se généralise à des maillages déstructurés.

L’organisation des calculs et le parallélisme qui en découle sont très naturels. Mais il est aussi délicat à programmer simplement et efficacement : comment répartir les calculs sur les coeurs de calcul disponibles ?

## 2.3 Implémentation StarPU

StarPU permet de décrire de façon simple et efficace ce type d’algorithme. L’algorithme consiste d’abord à construire le graphe de dépendance des macrocellules. Ensuite il faut effectuer un tri, dit topologique, des nœuds du graphe. Ensuite, pour chaque macrocellule parcourue dans l’ordre topologique, il faut :

* calculer les termes volumiques du schéma DG ([1](#x1-9001r1)) ;
* calculer les flux amont des macrocellules précédemment calculées ou venant des conditions aux limites ;
* résoudre un petit système linéaire local à la macrocellule ;
* extraire les données vers les macrocellules aval.

Chacune de ces étapes est décrite avec des tâches et des codelettes StarPU. Ces tâches sont ensuite soumises dans un ordre correct au support d’exécution, grâce à la numérotation topologique. C’est StarPU qui se charge de la parallélisation et des transferts de données en mémoire.

## 2.4 Résultats

Nous comparons le solveur linéaire direct du système complet avec le solveur de proche en proche pour diverses tailles des macrocellules. En effet, la gestion du graphe des tâches par StarPU a un coût. Il faut donc soumettre des tâches suffisamment grosses pour que ce surcoût soit amorti. Nous constatons dans nos expériences qu’il existe un découpage optimal entre les macro et les sous-cellules pour lequel le parallélisme de StarPU est très efficace. Les résultats sont alors très proches du parallélisme optimal auquel on peut s’attendre pour ce type d’algorithme de progression par front (sur un maillage *n* × *n*, au meilleur de la progression, il est possible de calculer en parallèle *n* macrocellules).

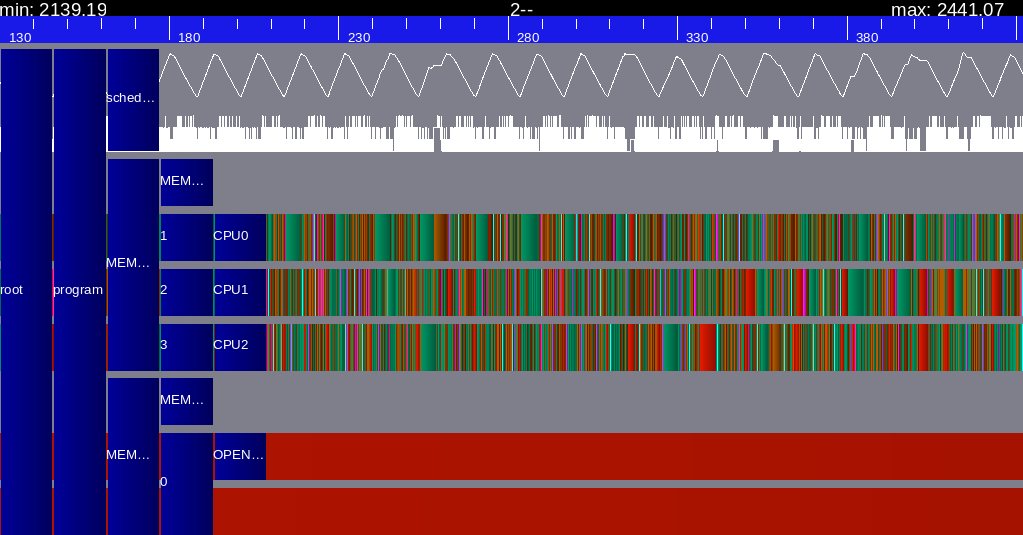
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| nb cores | 0 | 1 | 2 | 4 | 8 | 16 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 10 × 10 × 8 × 8 direct | 30 | 144 | - | - | - | - |
|  |  |  |  |  |  |  |
| 10 × 10 × 8 × 8 upwind | - | 32 | 19 | 12 | 7 | 6 |
|  |  |  |  |  |  |  |
| 20 × 20 × 4 × 4 upwind | - | 41 | 26 | 17 | 12 | 17 |
|  |  |  |  |  |  |  |
| 20 × 20 × 8 × 8 upwind | - | 120 | 72 | 40 | 28 | 20 |
|  |  |  |  |  |  |  |
|  |  | | | | | |

TABLE 1: Étude d’accélération StarPU pour le solveur frontal. AMD Opteron 16 cores, 2.8 Ghz. Temps en secondes secondes pour 200 itérations de la méthode temporelle. En fonction du découpage des macrocellules, le parallélisme est plus ou moins efficace.

StarPU offre aussi la possibilité de visualiser le graphe des tâches. Un exemple est donné figure [3](#x1-120023). Il est aussi possible de visualiser le diagramme de Gantt de l’ordonnancement des tâches. Un tel diagramme est représenté Figure [4](#x1-120034). Nous y comparons deux stratégies d’ordonnancement. Dans la première approche, nous imposons un point de synchronisation à la fin de chaque pas de temps. Nous voyons alors que la file d’attente se vide périodiquement. Dans la seconde approche, nous n’imposons aucun point de synchronisation. La file d’attente se remplit donc pendant une première phase, avant de se vider dans la seconde partie du calcul. Cette deuxième approche est légèrement plus efficace que la première (gain de l’ordre de 10%). Sans StarPU, tester la seconde approche n’aurait pas été aussi simple.



FIGURE 3: Zoom sur le graphe des taches générées par StarPU



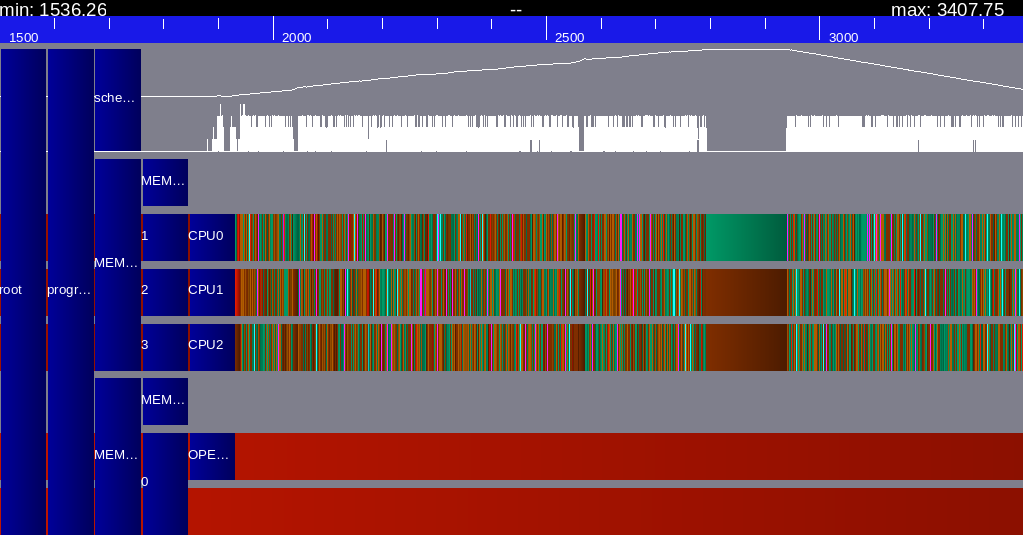


FIGURE 4: Deux stratégies d’ordonnancement. Diagramme de Gantt généré par StarPU. En haut : synchronisation à la fin de chaque pas de temps. En bas : pas de synchronisation.

# Application à DG explicite

## 3.1 Implémentation OpenMP

Nous avons programmé une version C/OpenMP d’un code DG explicite sur maillage structuré. Ce code a été optimisé à la fois en termes de parallélisme (vectorisation des boucles) mais aussi en termes d’accès mémoire grâce à une technique dite de tiling afin d’exploiter au mieux la mémoire cache du CPU. Ces optimisations sont sans doute très proches du maximum de ce qu’on peut obtenir avec un processeur multicore. Nous avons été assistés dans ce travail par V. Loechner, collègue informaticien à Strasbourg [[1](#Xhelluy2015asynchronous)].

## 3.2 Implémentation StarPU

Nous avons programmé dans SCHNAPS une version StarPU de l’algorithme DG explicite. Pour cela, comme dans Teta, nous avons découpé le calcul en tâches élémentaires et programmé les codelettes correspondantes.

Un gros avantage de la programmation StarPU par rapport à un code purement OpenCLest que nous n’avons pas à expliciter à la main les dépendances entre tâches. Il suffit de préciser quelles sont les données en lecture (R), écriture (W) ou lecture-écriture (RW) de chaque tâche. Par ailleurs la possibilité de tracer le graphe des tâches de StarPU ainsi que des diagrammes de Gantt de l’ordonnancement est très utile pour le débogage ou l’amélioration des performances.

Nous avons aussi pour chaque tâche programmée une codelette C pour CPU et une codelette OpenCL pour GPU. Ainsi StarPU peut choisir à l’exécution de lancer telle ou telle tâche sur CPU ou GPU. Nous avons aussi pour certaines codelettes C écrit plusieurs versions.

# Résultats

## 4.1 Comparaison OpenMP/OpenCL

Dans ce test, nous comparons le programme DG OpenMP avec une implémentation OpenCL du même algorithme. L’avantage de la version OpenCL est qu’elle peut être indifféremment être exécutée sur CPU ou GPU .Voir tableau [2](#x1-170012).

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Implementation | Time | Speedup |
|  |  |  |
|  |  |  |
| OpenMP (Intel CPU 12 cores) | 717 s | 1 |
|  |  |  |
| OpenCL (Intel CPU 12 cores) | 996 s | 0.7 |
|  |  |  |
| OpenCL (NVIDIA K20) | 45 s | 16 |
|  |  |  |
| OpenCL (AMD HD7970) | 38 s | 19 |
|  |  |  |
| OpenCL + MPI (4 x NVIDIA K20) | 12 s | 58 |
|  |  |  |
|  |  | |

TABLE 2: Comparaison des différentes implémentations de la méthode DG sur une grille régulière. Métériel : 2× Intel(R) Xeon(R) E5-2630 (6 cores, 2.3GHz), AMD Radeon HD 7970, NVidia K20m. Sur les CPU Intel l’hyperthreading est désactivé.

Dans ce tableau, nous constatons que l’implémentation OpenMP est la plus efficace sur CPU. Cependant la version OpenCL n’est pas loin derrière sur CPU avec une perte de seulement 30%. Cette perte s’explique par le fait que le code OpenCL a été optimisé pour GPU. Certains transferts en mémoire cache sont inutiles dans le cas du CPU et ralentissent les calculs sur ce type d’architecture.

Il est possible de spécialiser les kernels OpenCL et de choisir à l’exécution la version la plus efficace selon que l’on est sur CPU et GPU. Cette stratégie permettrait d’atteindre la même efficacité avec le code OpenCL qu’avec le code OpenMP, y compris sur CPU. Une étude en ce sens est proposée par exemple dans [[2](#Xshen2012performance)].

Sur GPU l’accélération du code OpenCL est bien sûr très efficace.

## 4.2 Comparaisons C/StarPU-C

Afin d’évaluer les performances de l’implémentation StarPU pour le schéma explicite, nous comparons un code C non optimisé avec le code StarPU où seules les codelettes C sont autorisées. Ceci peut se faire en positionnant la variable d’environnement STARPU\_NOPENCL à 0.

Le cas de calcul est celui d’une onde électromagnétique plane se propageant dans un domaine de forme cylindrique. Le maillage est constitué de 40 macrocellules DG de degré 2. Nous comparons les performances obtenues avec divers ordonnanceurs (“eager” ou “dmda”) et pour diverses tailles du maillage (23 sous-cellules ou 33 sous-cellules par macrocellule). Nous obtenons les résultats du tableau [3](#x1-180013).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| nb cores | 1 (c) | 1 (spu) | 2 (spu) | 4 (spu) | 8 (spu) | speedup (c/spu) |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 23, eager | 48 | 34 | 18 | 11 | 5 | 6.8 |
|  |  |  |  |  |  |  |
| 23, dmda | 48 | 32 | 20 | 12 | 6 | 5.3 |
|  |  |  |  |  |  |  |
| 33, eager | 214 | 187 | 109 | 59 | 23 | 8.1 |
|  |  |  |  |  |  |  |
| 33, dmda | 214 | 181 | 118 | 65 | 25 | 7.2 |
|  |  |  |  |  |  |  |
|  |  | | | | | |

TABLE 3: Comparaison des temps de calcul C/Starpu-C

StarPU permet d’étudier l’efficacité de l’ordonnanceur. On constate par exemple sur la figure [5](#x1-180025) que tous les “workers” StarPU (c’est à dire les coeurs de CPU) sont bien exploités : peu d’attentes (représentées en rouge). Les transferts mémoires (flèches blanches) sont également limités.

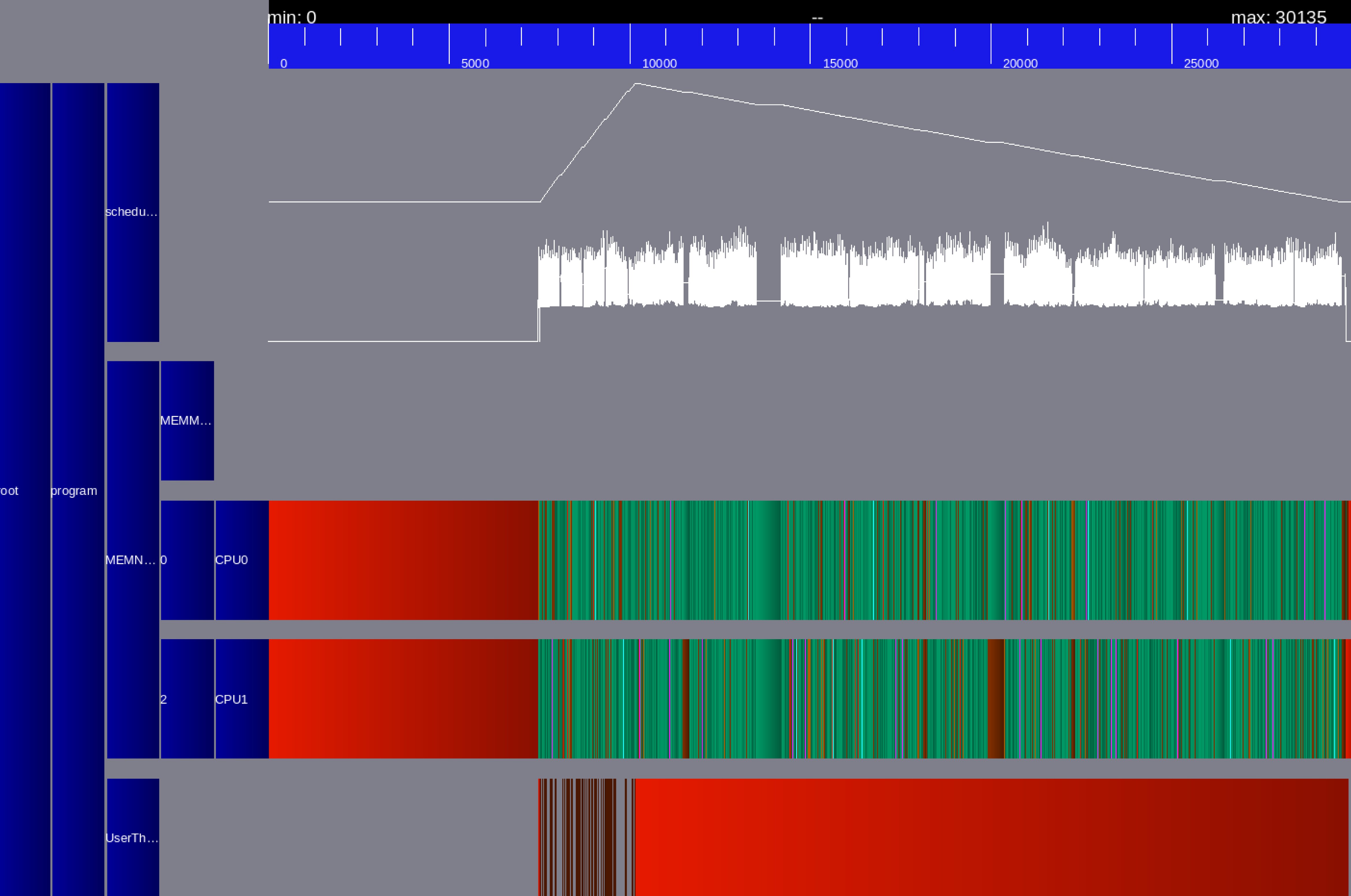


FIGURE 5: Exécution en parallèle sur deux cœurs CPU. Diagramme de Gantt généré par StarPU.

## 4.3 Comparaisons StarPU-C/StarPU-C+OpenCL

Dans ce test on cherche à évaluer l’apport de l’activation des GPU dans le calcul StarPU. Pour cela, on positionne la variable d’environnement STARPU\_NOPENCL de 0 à 2 (nombre de GPU à activer). Le maillage est fixé à une taille de 40 macrocellules DG découpées en 33 sous-cellules. Les résultats sont présentés dans le tableau [4](#x1-190014).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | 0 GPU | 1 GPU | 2 GPU | max. speedup |
|  |  |  |  |  |
|  |  |  |  |  |
| C+OpenCL (eager) | 59 | 40 | 36 | 1.6 |
|  |  |  |  |  |
| C+OpenCL (dmda) | 65 | 43 | 38 | 1.7 |
|  |  |  |  |  |
|  |  | | | |

TABLE 4: Comparaison des temps de calcul StarPU-C/StarPU-C+OpenCL

On constate que l’activation des GPU apporte une légère accélération. Cependant, celle-ci est limitée, quel que soit l’ordonnanceur. C’est assez surprenant et décevant, car l’ordonnanceur “dmda” est sensé tenir compte des transferts mémoire pour la distribution des tâches. Notre découpage des données ne devrait impliquer des transferts qu’aux interfaces entre les zones de calcul. Nous allons prendre contact avec les développeurs StarPU pour mieux comprendre ce défaut de performance. Nous avons soumis un projet de recherche au CEMRACS 2016 <http://smai.emath.fr/cemracs/cemracs16/>. Il est prévu que nous rencontrions des experts de StarPU lors de ce projet.

# Optimisations OpenCL CPU

## 5.1 Axes d’optimisation

Le solveur Teta a initialement été programmé et optimisé pour être exécuté sur GPU [4](file:///home/helluy/schnaps/old_schnaps/doc/temp/horoch_juin20165.html" \l "fn4x0) . Les kernels OpenCL qu’il contient sont utilisables sur CPU mais ne produisent pas de bons résultats en termes de performances. De manière générale, un kernel OpenCL optimisé GPU ne sera pas efficace sur CPU. Plusieurs axes de développement sont conseillés par les constructeurs de CPU (AMD, Intel), en voici une liste non exhaustive :

* Favoriser la mémoire globale à la mémoire locale : généralement, la mémoire locale d’un CPU est globale ;
* Augmenter la taille des work-groups (pour en diminuer le nombre) et sérialiser les traitements au sein d’un kernel : les changements de work-groups sont coûteux en temps ;
* Favoriser le stockage des données qui seront recalculées ;
* Utiliser les unités de calcul vectoriel SSE ou AVX si disponibles.

Ces axes d’optimisation ont été explorés (en partie) dans le cadre du projet HOROCH et nous verrons les gains obtenus dans la section suivante.

## 5.2 Résultats

Afin d’évaluer les performances suite à l’optimisation des kernels OpenCL pour CPU, nous comparons les temps de chaque kernel d’une même simulation avant et après modifications. Les résultats sont dépendants du matériel et ont été obtenus avec un CPU Intel Core i7-920 2,67 GHz et un GPU NVIDIA GeForce GTX 460. Avec cette configuration, le ratio de temps CPU/GPU avant optimisations était de 18.

Seuls les kernels “Volume” et “Surface” qui représentent respectivement les flux intra et inter éléments ont été modifié. Ces deux kernels représentaient 92% du temps total avant optimisations. Ils ont été modifiés pour ne plus utiliser la mémoire locale, pour qu’un work-item traite un élément complet / une surface complète et non plus un seul point d’interpolation et pour que les données d’interpolation soient stockées à la première itération puis réutilisées.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | avant (%) | après (%) |
|  |  |  |
|  |  |  |
| Volume | 48 | 13 |
|  |  |  |
| Surface | 49 | 71 |
|  |  |  |
| Mass | 2 | 10 |
|  |  |  |
| RK2 | 1 | 6 |
|  |  |  |
|  |  | |

TABLE 5: Comparaison des ratios de temps entre les différents kernels avant et après optimisations OpenCL CPU

Avec ces optimisations, le CPU a connu un speedup de 5,6 par rapport aux temps obtenus avec les kernels optimisés GPU. Ce speedup ramène le ratio de temps CPU/GPU à 3,2. Le tableau [5](#x1-220015) présente l’évolution des ratios de temps entre les différents kernels avant et après modifications.

Suite aux adaptations effectuées dans les fonctions d’interpolation, le GPU a lui aussi connu un impact positif avec un speedup de 1,4. Le ratio de temps CPU/GPU réel est donc de 4,5 en comptant cet impact.

# Conclusion et perspectives

En conclusion de cette tâche “Parallélisation hybride” nous sommes arrivés aux conclusions suivantes :

* Pour des architectures de codes complexes, OpenMP et son système de directives n’est pas forcément très indiqué pour réaliser des conceptions parallèles efficaces.
* L’écriture d’un code OpenCL est un peu plus compliquée, mais apporte l’avantage énorme de pouvoir tourner à la fois sur CPU et GPU. Sur CPU il est possible d’atteindre les mêmes performances qu’avec OpenMP.
* Les outils récents de génération automatique de code parallèle (pocc, PLUTO, DSL, etc.) sont très prometteurs, mais en l’état actuel de la technologie, ils sont encore en phase de recherche et développement. En général ils fonctionnent sur des codes simples ou des exemples assez académiques. Il s’agit néanmoins d’un domaine en évolution très rapide et il faut maintenir une veille technologique sur ces outils (ce à quoi nous nous employons).
* StarPU est le système de parallélisation par tâches le plus avancé à l’heure actuelle. Il est d’emploi assez aisé, efficace. Les outils annexes sont très utiles pour le débogage et l’optimisation.
* Les performances de notre implémentation StarPU de l’algorithme DG sont cependant décevantes lorsque l’on mixe des CPU et des GPU. Ces résultats, obtenus récemment, sont décevants et assez inexplicables pour l’instant, car StarPU a justement été développé pour le calcul hybride. Nous allons prendre contact cet été (2016) avec des développeurs StarPU pour comprendre d’où provient ce défaut de performances.

### Références

[1]   Philippe Helluy, Thomas Strub, Michel Massaro, and Malcolm Roberts. Asynchronous opencl/mpi numerical simulations of conservation laws. In *Proceedings of the 3rd International Workshop on OpenCL*, page 4. ACM, 2015.

[2]   Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance gaps between openmp and opencl for multi-core cpus. In *2012 41st International* *Conference on Parallel Processing Workshops*, pages 116–125. IEEE, 2012.