Modèle de programmation pour noyaux de calcul irréguliers sur accélérateur reconfigurable dans un système distribué hétérogène

Erwan Lenormand, Thierry Goubier, Loïc Cudennec, Henri-Pierre Charles

Compas'2021 : Parallélisme / Architecture/ Système/ Temps Réel
MILC - Lyon, France, 5-9 juillet 2021

Modèle de programmation pour noyaux de calcul irréguliers sur accélérateur reconfigurable dans un système distribué hétérogène

Erwan Lenormand[1], Thierry Goubier[1], Loïc Cudennec[2],Henri-Pierre Charles[3]

[1]Université Paris-Saclay, CEA, LIST, F-91191, Gif-sur-Yvette, France, erwan.lenormand@cea.fr
[2]DGA Maîtrise de l'Information, BP 7, 35998 Rennes, France
[3]Univ Grenoble-Alpes, CEA, LIST, F-38000, Grenoble, France

Résumé
Cet article présente un modèle de programmation pour les systèmes distribués hétérogènes intégrant des accélérateurs reconfigurables. Ce modèle de programmation cible les applications de calcul dont les motifs d'accès mémoire sont irréguliers. Il repose sur l'intégration des accélérateurs dans une mémoire virtuellement partagée logicielle leur permettant d'initier les accès aux données distribuées. Un partitionnement en *chunks* est utilisé pour abstraire l'irrégularité des structures de données. Ce partitionnement permet de parcourir les données sous forme de flux de *chunks*. Le modèle de programmation proposé permet : de régulariser les accès mémoire d'applications irrégulières, d'éviter le transfert de données inutiles grâce à une fine granularité d'accès et de cacher les latences d'accès aux données distribuées en superposant implicitement le flux de données transférées avec le flux de données traitées. Nous avons utilisé deux cas d'études : la multiplication de matrices creuses et un code de simulation de tsunami. Les résultats montrent la capacité du modèle de programmation à cacher les latences d'accès en atteignant des performances proches de celles d'un cas idéal.

Mots-clés : Mémoire Virtuellement Partagée, Système Hétérogène, Accélérateur Reconfigurable

1. Introduction

La fin de la mise à l'échelle de Dennard a conduit à une problématique de consommation énergétique des circuits intégrés. L'hétérogénéité apporte une solution à ce problème [8]. Pour cette raison, les systèmes HPC deviennent plus hétérogènes, notamment en intégrant des coprocesseurs (principalement des GPU). Ces derniers permettent d'augmenter l'efficacité FLOP/W du système en exécutant les portions d'une application, appelées noyaux de calcul, dont l'intensité arithmétique est élevée. Cependant tel que l'illustre le banc de test *High Performance Conjugate Gradient* (HPCG) [1], pour des noyaux de calcul irréguliers où l'intensité arithmétique est plus faible et les accès mémoire sont irréguliers, l'hétérogénéité de ces systèmes n'apporte pas de gains en efficacité. Cette contre performance s'explique par la nature *Memory bound* de ces applications qui stresse les systèmes mémoires et les systèmes d'interconnexions, et par la complexité de porter des noyaux de calcul irréguliers sur des GPU. L'architecture des GPU étant optimisée pour traiter en parallèle des vecteurs de données denses, l'exécution de noyaux de calcul irréguliers réduit leur efficacité. L'architecture reconfigurable des *Field-programmable gate arrays* (FPGA) leur permet d'exécuter efficacement des noyaux de calcul irréguliers [9]. Cependant, porter ces noyaux sur FPGA demeure une tâche complexe, notamment parce que leurs systèmes mémoires ne sont pas optimisés pour réaliser des accès aléatoires. Pour répondre à ce problème, nous proposons un modèle de programmation pour noyaux de calcul irréguliers ciblant les systèmes hétérogènes distribués intégrant des accélérateurs reconfigurables (FPGA). Ce dernier s'appuie sur une Mémoire Virtuellement Partagée (MVP) logicielle. L'intégration des accélérateurs dans la MVP leur permet d'initier les accès aux données distribuées. De cette manière, l'ensemble des unités de traitement peuvent réaliser des accès aléatoires avec une granularité fine.

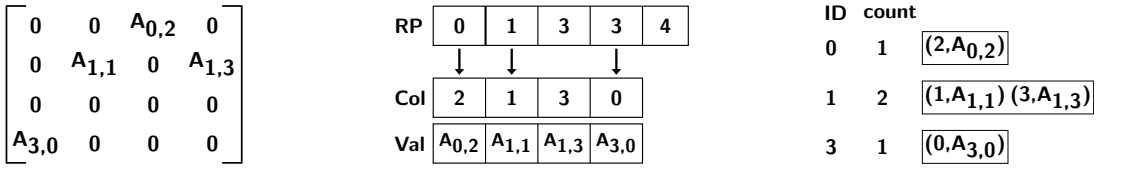
Cet article montre que le modèle de programmation proposé permet de régulariser les accès mémoire de noyaux de calcul d'algèbre linéaire creuse et de méthode des éléments finis en partitionnant leurs structures de données en *chunks*. En abstrayant les structures de données, ce partitionnement permet de précharger les données sous forme de flux. Nous montrons qu'il est possible de cacher les latences d'accès aux données en superposant, de manière implicite, les flux de données préchargées avec les flux de données traitées. Pour valider le modèle de programmation, nous avons utilisé deux cas d'études : la multiplication de matrices creuses comme application d'algèbre linéaire creuse et un code de simulation de tsunami comme application de méthode des éléments finis.
L'article s'articule de la façon suivante : la Section 2 décrit le fonctionnement de la MVP, la Section 3 présente le modèle de programmation proposé, la Section 4 décrit les expérimentations menées pour valider le modèle de programmation, la Section 5 indique des références sur des travaux connexes, enfin, la Section 6 conclut l'article.

2. Mémoire virtuellement partagée logicielle

Une mémoire virtuellement partagée est un système qui permet d'agréger des mémoires physiquement distribuées dans un espace logique partagé. Dans ces travaux nous considérons une MVP logicielle, appelée *Software-Distributed Shared Memory* (S-DSM), qui a été développée pour les micro-serveurs hétérogènes [6]. Celle-ci permet à des tâches logicielles de faire des allocations et des accès mémoire dans un espace logique partagé. La S-DSM est organisée comme un réseau semi-structuré où des processus clients sont connectés à un réseau pair-à-pair de processus serveurs. Les clients exécutent le code utilisateur et les serveurs gèrent les données partagées et les métadonnées associées. L'unité atomique de mémoire gérée par la S-DSM, dont la taille maximale peut être définie par l'application, est appelée *chunk*. Les accès aux *chunks* s'effectuent sous le contrôle d'un protocole de cohérence, qui est en charge de la localisation et du transfert des données. Chaque *chunk* est identifié par un numéro unique (*chunk ID*). L'intégration de noyaux de calcul implémentés sur un FPGA dans l'environnement de la S-DSM nécessite de créer une interface entre un serveur, qui est un processus logiciel, et la logique programmable de l'accélérateur. Cette interface repose sur un processus logiciel, appelé (*FPGA-client*), et un module matériel, appelé (*FPGA-server*). Le rôle du *FPGA-server* est de reproduire les services de la S-DSM pour les noyaux de calcul. Chaque noyau est associé à des flux de données partagées. Un flux est une interface avec le *FPGA-server* permettant de demander l'accès aux *chunks* et de communiquer la taille des données. Chaque flux est associé à une FIFO. La mémoire du FPGA est segmentée en emplacement de la taille d'un *chunk*. Le *FPGA-client* est un processus client de la S-DSM, qui sert de mandataire pour le FPGA en transférant les requêtes provenant des noyaux de calcul à un serveur de la S-DSM.

3. Modèle de programmation pour noyaux de calcul irréguliers

Le modèle de programmation que nous proposons repose sur la capacité des tâches accélératrices à initier des accès aléatoires aux données distribuées avec une granularité fine. Les systèmes mémoires et d'interconnexions des FPGA étant optimisés pour transférer les données par flux, les accès mémoire des noyaux irréguliers doivent être régularisés. Pour ce faire, nous proposons d'appliquer un partitionnement des données en *chunks*. Le *chunk* est un objet courant en informatique. Son concept de base est d'utiliser des métadonnées pour décrire un morceau de données mémorisé dans sa structure. Cette caractéristique permet d'abstraire des structures de données irrégulières. Nous utilisons la S-DSM, qui manipule nativement des *chunks*, pour partager les données aux différentes unités de traitement.
Les applications d'algèbre linéaire creuse opèrent sur des matrices (ou des vecteurs) pour lesquelles la majorité des éléments sont nuls. Le format Compressed Sparse Row (CSR), illustré en Figure 1b, est fréquemment utilisé pour compresser des matrices creuses. Les indices de colonnes et la valeur des éléments non nuls (*NNZ*) sont enregistrés dans l'ordre des rangées dans les tableaux *Col* et *Val*. Le tableau *RP* contient les indices du premier élément de chaque rangée. Telle que le représente la Figure 1c, nous avons adapté ce format à l'utilisation des *chunks*. L'indice de colonne et la valeur des éléments sont colocalisés pour former des pairs. L'ensemble des pairs d'une rangée est stocké dans le même *chunk*. Nous utilisons les métadonnées des *chunks* pour indiquer le nombre d'éléments contenus dans une rangée. Ce format permet de réduire le nombre d'accès nécessaire pour lire ou écrire une rangée.



(a) Format dense. (b) Format CSR. (c) Format CSR à base de chunks.

FIGURE 1 – Format de matrices.

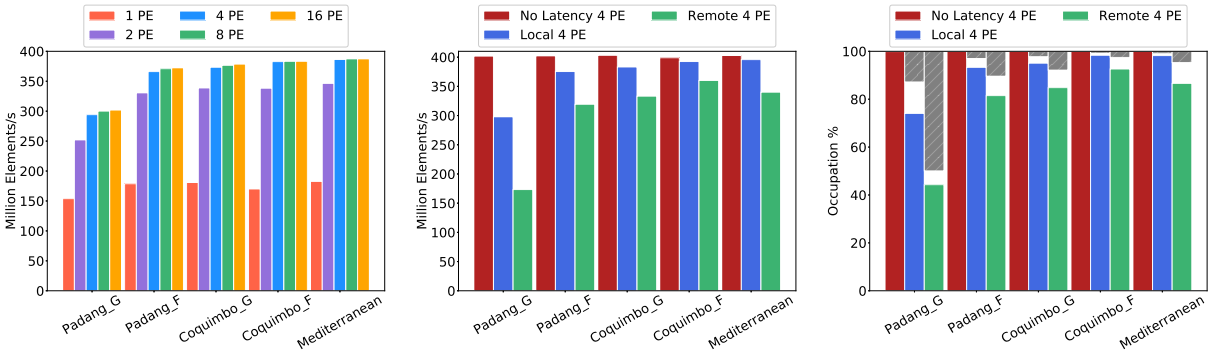
Parcourir une matrice consiste à demander l'accès à chaque rangée, ce qui a pour effet de charger les données de la rangée dans la mémoire du FPGA, puis à demander le transfert des données entre la mémoire du FPGA et le noyau de calcul. En dissociant les requêtes de transferts et les requêtes d'accès, il est possible de précharger les données dans la mémoire du FPGA afin de cacher les latences d'accès.
Les applications de méthode des éléments finis itèrent sur les éléments, les nœuds ou les arêtes d'un maillage. Pour beaucoup d'applications HPC, le maillage est non structuré avec des éléments triangulaires en 2D ou tétraédriques en 3D. Le traitement réalisé nécessite d'accéder aux données en fonction de la topologie du maillage, ce qui implique des accès mémoire par indirections. La topologie du maillage et la façon dont ses données sont indexées ont un impact déterminant sur les performances de l'application. Par exemple, si la différence d'indices des sommets des éléments est grande, un système mémoire à base de cache souffrira d'une faible utilisation. Les courbes de remplissage permettent d'améliorer la localité des accès aux données [15]. Cette méthode consiste à tracer une courbe de remplissage dans l'espace géométrique du maillage et d'ordonnancer les nœuds en fonction de l'ordre dans lequel ils rencontrent la courbe. Elle permet d'apporter une bonne localité des données. Seules les données situées à l'intersection de plusieurs espaces géométriques éloignés souffrent d'une mauvaise localité. Nous utilisons une courbe de Hilbert pour ordonnancer des maillages, puis nous partitionnons les données en *chunks*, de tailles constantes, contenants les données d'indices consécutifs. De cette manière, il est possible de parcourir la majorité des données d'un maillage irrégulier à travers une fenêtre glissante de *chunks* limités en nombre. Nous utilisons ces observations pour concevoir des noyaux qui itèrent sur les données de maillages irréguliers. La fenêtre glissante est implémentée par une FIFO adressable, qui est alimentée par un flux constant de *chunks*. Nous utilisons des tampons mémoire pour accéder aux données qui ne peuvent pas être contenues dans la fenêtre. Ces données sont facilement identifiables en analysant le tracé de la courbe, ce qui permet d'alimenter les tampons par un flux de *chunks* prédéfinis.

4. Validation du modèle de programmation

Pour valider le modèle de programmation nous avons mené des expériences à l'aide d'un outil de simulation que nous avons développé [13]. L'objectif était d'évaluer, à partir d'une modélisation haut niveau, les effets des latences d'accès aux données sur la capacité d'accélération des noyaux de calcul. L'évaluation des performances repose sur l'analyse de l'activité du noyau de calcul simulé et sur la génération des latences relatives à cette activité. L'outil utilise une méthode hybride : l'activité du noyau est générée par un moteur de simulation et les latences sont produites en mesurant les latences des requêtes S-DSM réellement exécutées sur l'architecture physique. Le moteur de simulation est un processus client de la S-DSM qui peut être exécuté sur un nœud de calcul différent de celui qui exécute le processus serveur. De cette manière il est possible d'étudier différentes topologies de système correspondant à des profiles de latences différents. Dans le reste de la section, trois topologies sont présentées : *No Latency* qui correspond au cas idéal où les requêtes d'accès sont immédiatement satisfaites, *Local* qui correspond au cas où le FPGA est localisé sur le nœud exécutant le serveur S-DSM, et *Remote* qui correspond au cas où le FPGA et le serveur S-DSM sont sur des nœuds différents. La topologie *Local* induit des latences moyennes (383 µs pour des requêtes de lecture et 207 µs pour des requêtes d'écriture). La topologie *Remote* induit des latences élevées (1311 µs pour des requêtes de lecture et 533 µs pour des requêtes d'écriture). Le FPGA modélisé est un Xilinx Virtex-7 fonctionnant à 200 MHz et dont la bande passante mémoire théorique maximale entre le contrôleur mémoire et la logique programmable est de 12.8 Go/s. Les expérimentations ont porté sur l'accélération d'un noyau de calcul d'un code de simulation de tsunami et sur l'accélération d'une opération de multiplication de matrices creuses. Les simulations étant non-déterministes, les résultats présentés sont les valeurs médianes parmi 10 exécutions.

4.1. Cas d'étude 1 : Simulation de tsunami

Le code à l'étude provient du code de simulation TsunAWI, qui résout des équations de Saint-Venant avec une méthode des éléments finis, et dont les résultats sont utilisés pour le système d'alerte de tsunami indonésien et sous contrainte temps-réel dans le projet européen LEXIS [10]. La structure de données de base est un maillage non structuré triangulaire. Ce code a été optimisé pour les performances, notamment au regard de l'ordonnancement du maillage [11]. Nous avons conçu un noyau pour accélérer le calcul du gradient, une des opérations du code à l'étude. Cette opération consiste, pour chaque élément, à faire la somme aux trois sommets de la hauteur de la surface de l'eau multipliée par une fonction de base. Les valeurs de la fonction de base sont précalculées et sont uniques pour chaque sommet de chaque élément. Le noyau est implémenté avec plusieurs unités de traitement (*Processing Element (PE)*) indépendantes. Chaque PE traite une partie différente du maillage. La taille de la fenêtre glissante est de 1024 mots (4 octets). Pour les expérimentations nous avons utilisé cinq maillages, dont les caractéristiques sont présentées dans le Tableau 1. La mémoire du FPGA a été configurée à 64 Mio.



(a) Vitesse de calcul en MElements/s en fonction du niveau de parallélisme.

(b) Vitesse de calcul en MElements/s en fonction de la topologie du système.

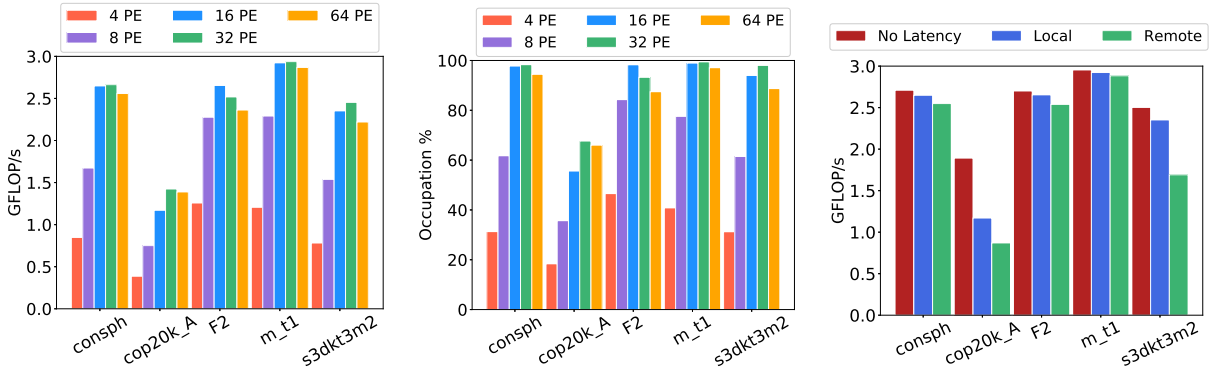
(c) Activité du contrôleur mémoire (taux d'occupation) en fonction de la topologie du système. Les barres grises représentent le temps de chargement de la première donnée.

FIGURE 2 – Performances du noyau de calcul de simulation de tsunami.

Pour la première expérimentation, nous avons évalué les performances du noyau en fonction du niveau de parallélisme en implémentant entre 1 et 16 PE. Les résultats obtenus, illustrés sur la Figure 2a, montrent une grande baisse de l'efficacité au-delà de 2 PE. Cette baisse de rendement traduit une sous-exploitation des unités de traitement due à une famine de données. Pour comprendre ce phénomène, nous avons mené une deuxième expérimentation en faisant varier la topologie du système afin d'évaluer les effets de la latence d'accès aux données sur les performances d'un noyau avec 4 PE. La Figure 2b expose les vitesses de calcul obtenues. Nous pouvons observer que l'augmentation des latences provoque un ralentissement de la vitesse de calcul et que plus le jeu de données est grand, plus le ralentissement est faible. La Figure 2c permet de comprendre le phénomène de famine observé et la corrélation entre la taille du maillage et l'impact des latences sur les performances. Nous pouvons observer que le contrôleur mémoire est occupé en permanence pour la topologie sans latence, ce qui traduit une saturation de la bande passante de la mémoire. Ce goulot d'étranglement explique la famine de données. Le temps de chargement des premières données nécessaire pour que le noyau rentre dans un mode de fonctionnement nominal explique pourquoi l'effet de la latence sur les performances est corrélé à la taille du maillage. Plus le jeu de données est petit, plus ce temps de chargement représente une part importante du temps de traitement. D'une manière générale plus le jeu de données est petit, plus il est compliqué de

	Résolution grossière				Résolution fine			
Zone	Nom	# Éléments	# Nœuds	Taille	Nom	# Éléments	# Nœuds	Taille
Océan Indien	Padang_G	460 119	231 586	14 Mo	Padang_F	2 470 345	1 242 653	74 Mo
Océan Pacifique	Coquibo_G	3 396 755	1 709 506	102 Mo	Coquimbo_F	9 762 027	4 887 927	293 Mo
Mer Méditerranée	Mediterranean	9 917 645	4 999 404	298 Mo				

TABLE 1 – Caractéristiques du jeu de maillages utilisé pour les expérimentations.



(a) Vitesse de calcul en GFLOP/s en fonction du niveau de parallélisme.

(b) Activité du contrôleur mémoire (taux d'occupation) en fonction du niveau de parallélisme.

(c) Vitesse de calcul en GFLOP/s en fonction de la topologie du système.

FIGURE 3 – Performances du noyau de calcul SpGEMM.

cacher les latences d'accès. Ces expérimentations montrent que le modèle de programmation proposé permet de traiter des jeux de maillages avec une taille mémoire de l'accélérateur réduite sans perte de performance et de cacher efficacement les latences d'accès aux données lorsque le temps de traitement est suffisamment important pour minimiser l'impact du temps de chargement des premières données.

4.2. Cas d'étude 2 : Multiplication de matrices creuses

La multiplication de matrices creuses (*General Sparse Matrix-Matrix Multiplication (SpGEMM)*) est fréquemment utilisée pour étudier des méthodes d'accélération pour l'algèbre linéaire creuse. Cette opération est compliquée à optimiser, car elle génère des motifs d'accès mémoire irréguliers avec une faible efficacité en terme d'opérations flottantes par unité de temps. Nous avons conçu le noyau de calcul en utilisant l'algorithme de Gustavson [12] (parcours par rangée). Pour paralléliser les calculs, le noyau est implémenté avec plusieurs unités de traitement (PE). Les premiers étages du noyau accèdent aux rangées de la première matrice source et distribuent les éléments non nuls aux PE. Chaque PE multiplient les éléments reçus par ceux correspondant aux rangées de la seconde matrice source. Enfin, les derniers étages accumulent les résultats partiels et écrivent la matrice résultat. Les indices et les valeurs des éléments des matrices sont encodés avec 4 octets (calculs en précision simple).

Pour évaluer la capacité du modèle de programmation à s'adapter à l'irrégularité, nous avons choisi un jeu de matrices [7], présentées en Table 2, dont les tailles, les densités et les motifs varient. Pour reproduire la situation où la capacité mémoire de l'accélérateur requiert de transférer les données au fil de l'exécution, nous avons adapté la taille de la mémoire simulée en fonction du jeu de matrices. Ainsi, la mémoire de l'accélérateur a été configuré avec 64 Mio.

Pour la première expérimentation, nous avons fait varier le niveau de parallélisme du noyaux de calcul en implémentant entre 4 et 64 unités de traitement sur le nœud local. Les vitesses de calcul obtenues, illustrées sur la Figure 3a, montrent que l'augmentation du parallélisme permet d'accélérer les calculs jusqu'à un niveau limite situé entre 16 et 32 PE. Les résultats mettent en lumière également une forte baisse de l'efficacité au-delà de 8 PE. Cette limitation traduit une sous-exploitation des unités de traitement due à une insuffisante alimentation en données (famine de données). Ce phénomène peut s'expliquer soit par une famine au niveau de la mémoire du FPGA (causée par des latences d'accès trop importantes) ou par un goulot d'étranglement provoqué par la bande passante de la mémoire du

Matrice	rangées	NNZ	Densité (%)	Empreinte mémoire
`consph`	83334	6010480	0.087	294 Mo
`cop20k_A`	121192	2624331	0.018	182 Mo
`F2`	71505	5294285	0.10	383 Mo
`m_t1`	97578	9753570	0.10	427 Mo
`s3dkt3m2`	90449	3753461	0.046	134 Mo

TABLE 2 – Jeu de matrices carrées utilisées pour les expérimentations. Le nombre d'éléments et la densité font référence à la matrice source. L'empreinte mémoire comprend les trois matrices opérandes.

FPGA. L'activité du contrôleur mémoire, illustrée sur la Figure 3b, indique un taux d'occupation proche de 100% à partir de 16 PE (hormis pour la matrice `cop20k_A`). Cette activité intense du contrôleur mémoire implique un effet de saturation de la mémoire dû à une limitation de la bande passante.
La seconde expérience visait à étudier les impacts de la topologie sur les performances. Pour ce faire nous avons utilisé une configuration à 16 PE. Les résultats obtenus, illustrés sur la Figure 3c, montrent que pour les matrices les plus volumineuses les différences de performances avec le cas idéal sont très faibles (entre 1% et 6%). Ces écarts peuvent s'expliquer par le temps de chargement des premières données avant d'atteindre le mode de fonctionnement nominal du noyau. Ces résultats montrent la capacité du modèle de programmation à cacher les latences d'accès aux données. Pour les matrices moins volumineuses, leurs faibles densités induisent une intensité arithmétique plus faible. De ce fait, pour ces matrices il est plus compliqué de superposer le flux de données traitées avec le flux de données transférées. Ainsi, la capacité à masquer les latences est moins bonne.

5. Travaux connexes

Des travaux antérieurs ont été menés pour construire une mémoire partagée dans un système distribué avec accélérateurs. Willendberg et coll. [18] ont proposé une infrastructure de communication inter-FPGA compatible avec GASNet [5]. Ces travaux permettent aux unités de traitement implémentées sur un FPGA d'initier des accès direct à la mémoire distante (RDMA) d'un autre FPGA. Unicorn [4] propose un MVP pour grappes de CPU-GPU. Celle-ci est mise en place avec des sémantiques transactionnelles et des synchronisations différées des données en blocs. StarPU [2] est un support exécutif pour systèmes hétérogènes qui utilise une MVP pour gérer la réplication des données, mais elle n'est pas directement exposée à l'utilisateur.
Des travaux récents ont proposé d'appliquer à des matrices creuses un partitionnement en *chunks* pour accélérer des opérations d'algèbre linéaire creuse. Winter et coll. [19] ont proposé un noyau de calcul de matrices creuses pour GPU basé sur des *chunks* adaptatifs. Cette approche consiste à utiliser des *chunks* pour enregistrer les résultats partiels des multiplications, puis d'utiliser les métadonnées des *chunks* pour faire les sommes. Rubensson et Rudberg [14] ont proposé un modèle de programmation, appelé *Chunks and Tasks*, pour paralléliser des applications irrégulières. Ce modèle représente les matrices sous la forme d'arbres quaternaires sporadiques. MatRaptor [17] et REAP [16] utilisent des *chunks* pour adapter le format CSR et utilisent l'algorithme de Gustavson pour implémenter des noyaux de calcul de multiplication de matrices creuses sur FPGA.
Barrio et coll. [3] ont proposé un algorithme de tri pour maillage non structuré permettant le traitement par flux d'applications de méthode des éléments finis. Cet algorithme est utilisé pour étudier l'accélération d'applications scientifiques sur des plateformes CPU-FPGA.

6. Conclusion

Améliorer l'efficacité énergétique des systèmes HPC est devenu un enjeu majeur. Grâce à leurs architectures reconfigurables, les FPGA peuvent augmenter le rendement énergétique d'applications HPC irrégulières. Malheureusement, à cause de leur complexité d'utilisation, les FPGA sont sous employés dans les systèmes HPC. Dans cet article, nous avons proposé un modèle de programmation pour l'accélération de noyaux de calcul irréguliers sur FPGA intégrés dans un système distribué hétérogène. Ce modèle de programmation repose sur une mémoire virtuellement partagée logicielle qui permet aux tâches d'initier l'accès aux données distribuées et sur un partitionnement des données en *chunks* qui permet d'abstraire la structure irrégulière des données. Nous avons montré comment ce modèle de programmation pouvait s'appliquer à des noyaux de l'algèbre linéaire creuse et de méthode des éléments finis. Nous avons conduit des expérimentations avec un outil de simulation hybride, qui exploite l'architecture physique du système pour produire des données précises. Les expériences ont montré que le modèle de programmation permet de cacher efficacement les latences d'accès aux données. Elles ont également montré que le modèle de programmation permet d'accélérer des noyaux de calcul irréguliers, qui sont *memory bound*, avec un espace mémoire limité sans perte de performance. Ces résultats nous encouragent à étudier l'utilisation d'accélérateurs disposant d'un système mémoire à haute bande passante (HBM) pour repousser le goulot d'étranglement observé.

Bibliographie

1. High-Performance Conjugate Gradient (HPCG) Benchmark results, november 2020.
2. Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, vol. 23, n2, 2011.
3. Barrio (P.), Carreras (C.), López (J. A.), Óscar Robles, Jevtic (R.) et Sierra (R.). – Memory optimization in FPGA-accelerated scientific codes based on unstructured meshes. *Journal of Systems Architecture*, vol. 60, n7, 2014, pp. 579–591.
4. Beri (T.), Bansal (S.) et Kumar (S.). – The Unicorn Runtime : Efficient Distributed Shared Memory Programming for Hybrid CPU-GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, n5, 2017.
5. Bonachea (D.) et Jeong (J.). – *GASNet : A Portable High-Performance Communication Layer for Global Address-Space Languages*. – Rapport technique, CS258 Parallel Computer Architecture Project, University of California Berkeley, 2002.
6. Cudennec (L.). – Software-Distributed Shared Memory over heterogeneous micro-server architecture. – In *Euro-Par 2017 : Parallel Processing Workshops*, 2017.
7. Davis (T. A.) et Hu (Y.). – The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, vol. 38, n1, décembre 2011.
8. Eeckhout (L.). – Heterogeneity in Response to the Power Wall. *IEEE Micro*, vol. 35, n4, 2015, pp. 2–3.
9. Escobar (F. A.), Chang (X.) et Valderrama (C.). – Suitability Analysis of FPGAs for Heterogeneous Platforms in HPC. *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, n2, 2016.
10. Goubier (T.), Martinovic (J.), Dubrulle (P.), Ganne (L.), Louise (S.), Martinovic (T.) et Slaninová (K.). – Real-Time Model of Computation over HPC/Cloud Orchestration - The LEXIS Approach. – In *Proceedings of the 14th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS-2020)*, *Advances in Intelligent Systems and Computing*, volume 1194, 2020.
11. Goubier (T.), Rakowsky (N.) et Harig (S.). – Fast Tsunami Simulations for a Real-Time Emergency Response Flow. – In *2020 IEEE/ACM HPC for Urgent Decision Making, UrgentHPC@SC 2020*, pp. 21–26. IEEE, 2020.
12. Gustavson (F. G.). – Two Fast Algorithms for Sparse Matrices : Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.*, vol. 4, n3, 1978, p. 250–269.
13. Lenormand (E.), Goubier (T.), Cudennec (L.) et Charles (H.-P.). – A combined fast/cycle accurate simulation tool for reconfigurable accelerator evaluation : application to distributed data management. – In *2020 International Workshop on Rapid System Prototyping (RSP)*, 2020.
14. Rubensson (E. H.) et Rudberg (E.). – Chunks and Tasks : A programming model for parallelization of dynamic algorithms. *Parallel Computing*, vol. 40, n7, 2014.
15. Sastry (S. P.), Kultursay (E.), Shontz (S. M.) et Kandemir (M. T.). – Improved Cache Utilization and Preconditioner Efficiency through Use of a Space-Filling Curve Mesh Element- and Vertex-Reordering Technique. *Engineering with computer*, vol. 30, n4, 2014.
16. Soltaniyeh (M.), Martin (R. P.) et Nagarakatte (S.). – Synergistic CPU-FPGA Acceleration of Sparse Linear Algebra, 2020.
17. Srivastava (N. K.), Jin (H.), Liu (J.), Albonesi (D. H.) et Zhang (Z.). – MatRaptor : A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. – In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2020.
18. Willenberg (R.) et Chow (P.). – A Remote Memory Access Infrastructure for Global Address Space Programming Models in FPGAs. – In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Association for Computing Machinery, 2013.
19. Winter (M.), Mlakar (D.), Zayer (R.), Seidel (H.-P.) et Steinberger (M.). – Adaptive Sparse Matrix-Matrix Multiplication on the GPU. – In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, 2019.

Annexes

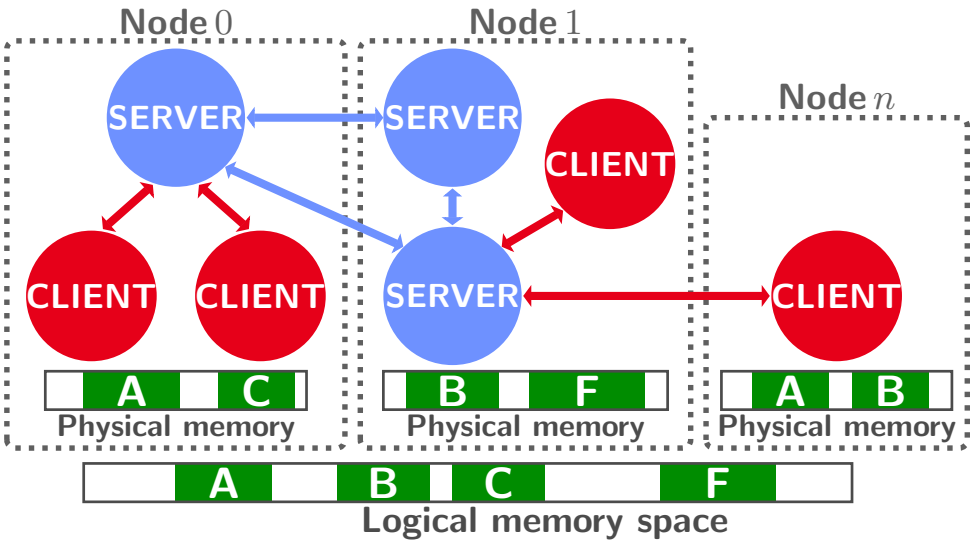
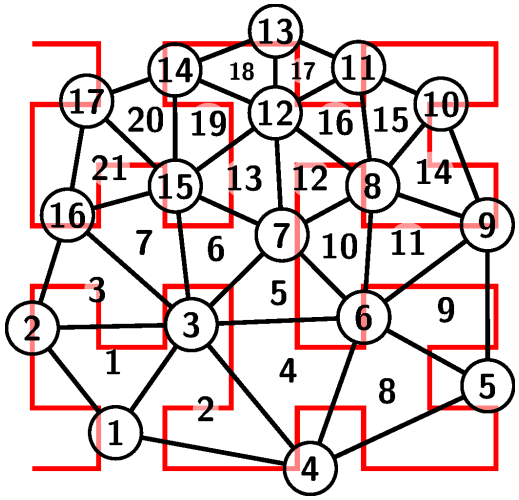


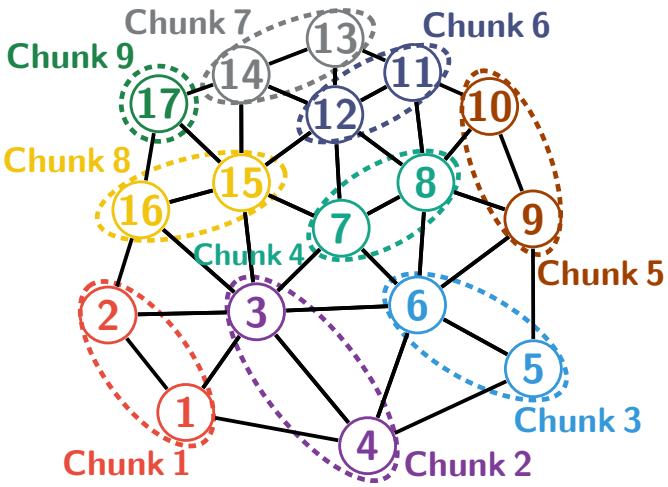
FIGURE 4 – Représentation de l'organisation de la mémoire virtuellement partagée logicielle utilisée dans ces travaux. La topologie du système est un réseau semi-structuré *super-peer*. Les cercles bleus représentent les processus serveurs connectés avec une topologie pair-à-pair. Les cercles rouges représentent les processus clients, où chaque client est connecté à un unique serveur. Les rectangles verts représentes les données partagées, sous forme de *chunks*, unifiées dans un même espace logique et distribuées dans les mémoires physiquement réparties.

```
void compute_gradient(const size_t nTriangles,  // Number of mesh elements
                      const uint32_t ** elem2d, // Mesh topology
                      const float * ssh,        // Nodes sea surface heights
                      const float ** bafu,      // Vertex basis function values
                      float * grad)             // Element computed gradients
{
    for(size_t i = 0; i < nTriangles; ++i)
        grad[i] =  bafu[i][0] * ssh[elem[i][0]]
                 + bafu[i][1] * ssh[elem[i][1]]
                 + bafu[i][2] * ssh[elem[i][2]];
}
```

Code 1 – Code de référence utilisé pour la modélisation du noyau de calcul du cas d'étude de simulation de tsunami et interprété du code source TsunAWI (https ://gitlab.awi.de/tsunawi/tsunawi).

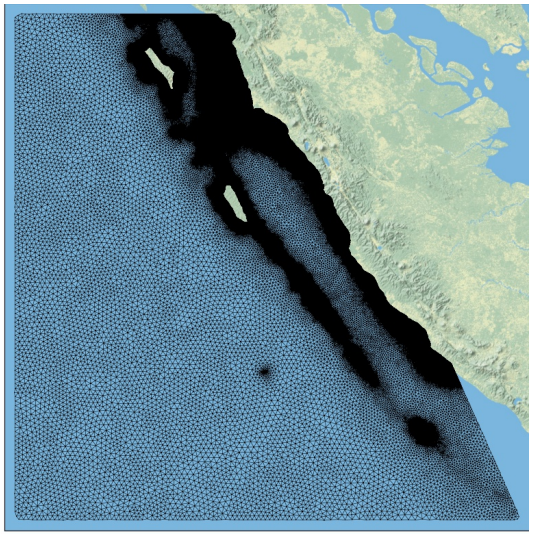


(a) Exemple de tri en suivant le tracé d'une courbe de Hilbert. La courbe représentée en rouge commence en bas à gauche et termine en haut à gauche. Les sommets du maillage sont indexés en suivant le parcours de la courbe. Les triangles sont numérotés en fonction de l'ordre croissant de leurs indices de sommets.

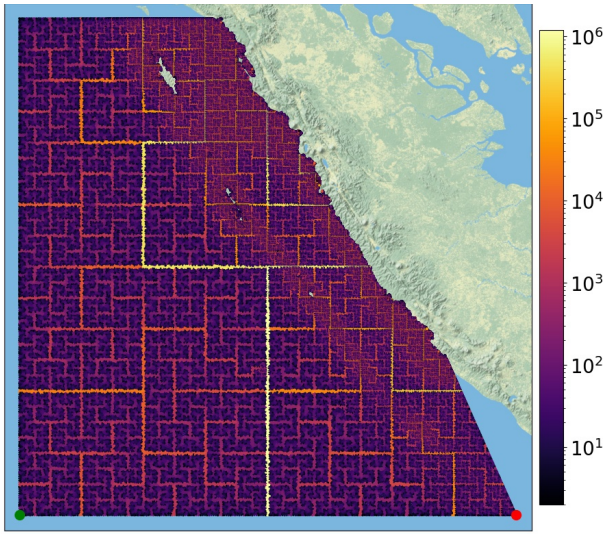


(b) Exemple de partitionnement du maillage en *chunks*. Les valeurs de sommets d'indices consécutifs sont regroupées en *chunks* d'une taille maximale de 2 valeurs.

FIGURE 5 – Exemple de tri et de partitionnement d'un maillage bidimensionnel non-structuré.



(a) Topologie du maillage à résolution grossière (Padang_G). La plus petite longueur d'arête est de l'ordre de la centaine de mètres. La plus grande longueur d'arête est de l'ordre de la quinzaine de kilomètres.



(b) Représentation, sous forme de carte de couleur à échelle logarithmique, de la différence maximale entre l'indice de chaque nœud et celui de ses voisins. Les lignes formées par les points jaunes illustrent les intersections des espaces géométriques de la courbe les plus éloignés.

FIGURE 6 – Représentation du maillage Padang suivant la zone de subduction de la fosse de Java le long de la côte de l'océan Indien de l'île indonésienne de Sumatra. La résolution du maillage est plus élevées sur les zones côtières et terrestres.

Algorithm 1 Algorithme de Gustavson pour multiplier deux matrices creuses. Les matrices A et B sont accédées séquentiellement rangée par rangée. La matrice B est accédée par rangée en fonction des indices de colonnes des éléments non nuls de la matrice A.

Input: $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$
Output: $C \in \mathbb{R}^{m \times p}$
for i = 0,...,m-1 **do**
  For Each $A_{i,k}$ in row $A_{i,*}$ **do**
    For Each $B_{k,j}$ in row $B_{k,*}$ **do**
      if $C_{i,j} == 0$ **then**
        $C_{i,j} = A_{i,k} \times B_{k,j}$
      else
        $C_{i,j}$ += $A_{i,k} \times B_{k,j}$
      end if
    end for
  end for
end for

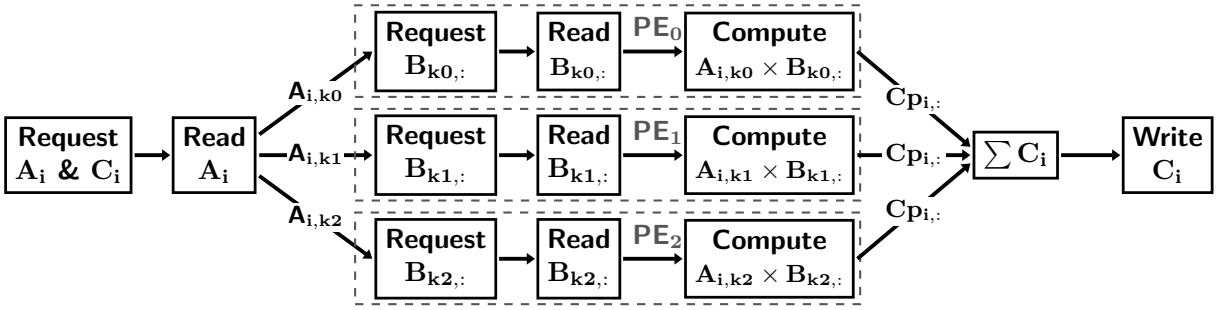
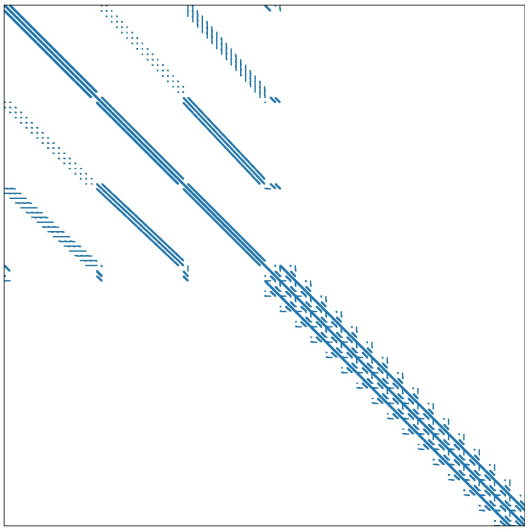
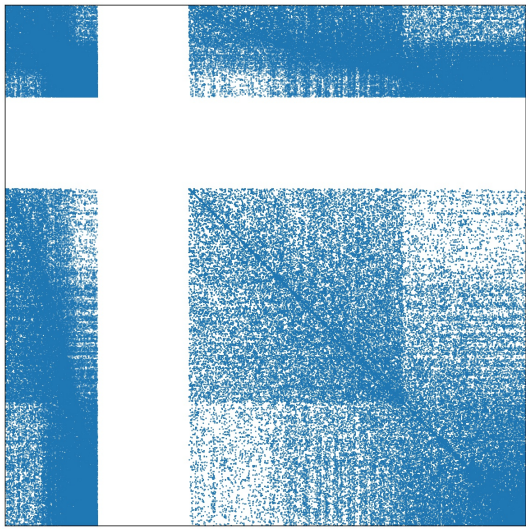


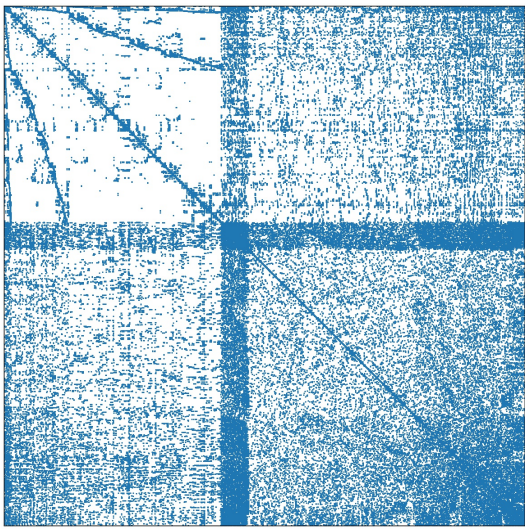
FIGURE 7 – Représentation en flot de données du noyau de calcul conçu pour le cas d'étude de la multiplication de matrices creuses. Le noyau est implémenté avec 3 unités de traitement (PE). Le premier étage demande l'accès aux rangées d'indice i de la matrice A (en lecture) et de la matrice C (en écriture). Le deuxième étage lit les données de la rangée i de la matrice A et les distribue aux PE. Le premier étage de chaque PE demande l'accès à la rangée k correspondant à l'indice de colonne de l'élément non nul de la matrice A en cours de traitement. Le deuxième étage de chaque PE lit les données de la rangée k de la matrice B. Le troisième étage de chaque PE calcul les résultats partiels (multiplication) des éléments lus. L'ensemble des résultats partiels sont sommés par l'avant dernier étage pour former les rangées de la matrice C. Le dernier étage écrit les rangées de la matrice C.



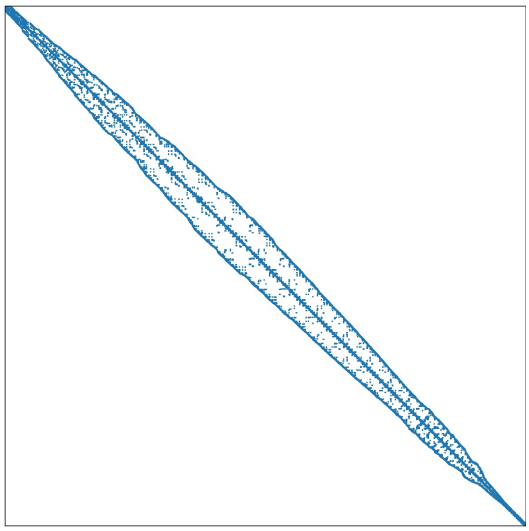
(a) Matrice consph.



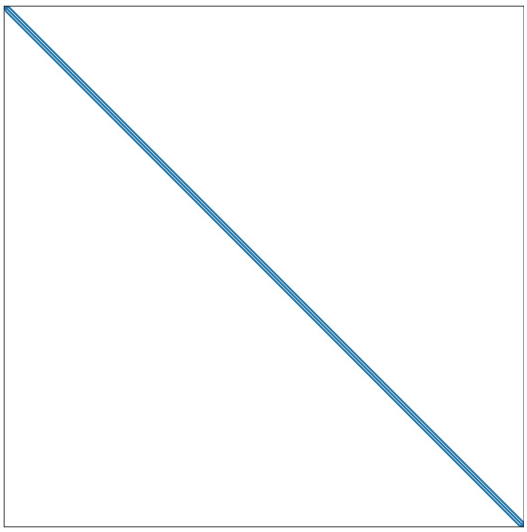
(b) Matrice cop20k_A.



(c) Matrice F2.



(d) Matrice m_t1.



(e) Matrice S3dkt3m2.

FIGURE 8 – Représentation en nuage de points du jeu de matrices utilisé pour les expérimentations du cas d'étude de la multiplication de matrices creuses.