M2 - Architecture et Programmation d'accélérateurs Matériels.

(APM 2024-2025)

TP4

Programmation GPU

OpenMP

julien.jaeger@cea.fr
adrien.roussel@cea.fr

Les objectifs de ce TP sont :
— Mise en pratique du modèle de déport de calcul avec OpenMP.

I Compilation & Vérification

Environnement :
Afin de compiler un programme OpenMP supportant le support de déport de calculs sur accélérateurs type GPU, le compilateur ainsi que le runtime OpenMP doivent le supporter.Pour cela, nous devons charger un environnement de compilation particulier, avec LLVM, depuis le supercalculateur ROMEO
Pour charger l'environnement qui va bien pour utiliser le compilateur clang installé : `module load llvm/14.0.5-gcc_10.2.0-targets_X86_NVPTX`

Compilation :
Nous allons à présent compiler notre premier programme OpenMP qui va vérifier que le support "target" est bien activé pour les GPU Nvidia disponible sur Romeo. Il faut alors rajouter à la ligne de commande, les options suivantes : `-fopenmp-targets=nvptx64-nvidia-cuda -Xopenmp-target -march=sm_60`

Q.1: Le code se trouvant dans le dossier `ONGPU` permet de faire un test simple afin de vérifier que le déport du calcul sur GPU s'est effectué correctement. Créez un fichier `Makefile` afin de prendre en compte les informations de compilation ci-dessus et vérifiez que l'exécution s'est déroulée avec succès. Attention, la compilation se fait uniquement sur les noeuds de login, et l'exécution sur un noeud de calcul disposant d'au moins un GPU.

Q.2: Grâce aux fonctions du runtime OpenMP, affichez le nombre de devices disponibles sur le système courant.

Q.3: Initiez une première ligue d'équipes et affichez le nombre d'équipes grâce aux fonctions du runtime.

II Kernels mathématiques élémentaires

Dans cette partie, nous allons voir la programmation de kernels mathématiques élémentaires tels que `Axpy` et la multiplication de matrice (gemm).

Attention : Pour transférer des données depuis/vers le GPU grâce à la clause map, si la donnée est un pointeur, n'oubliez pas de spécifier la taille de cette donnée ou bien sinon vous aurez une erreur à l'exécution. Si A est un vecteur et N sa taille, procédez de la manière suivante :
`#pragma omp target map(to/from/tofrom: A[:N])`

Q.4: `Axpy` est une opération très répandue en algèbre linéaire. Soit x et y deux vecteurs et a un réel, l'opération `Axpy` effectue le calcul suivant : $y = a \times x + y$. Écrivez dans un premier temps un programme séquentiel décrivant cette opération. Puis, grâce aux directives OpenMP, proposez une parallélisation sur accélérateur. Indice : afin de rendre le programme le plus efficace possible en tirant avantage des accès coalescés du GPU, assurez-vous qu'une équipe s'occupe d'une portion contigüe du vecteur. Pour cela, calculer un chunk d'indices à calculer par équipes, que vous calculerez grâce à la fonction `omp_get_num_teams()`

Q.5: À partir du fichier `gemm.c`, effectuez le même exercice, mais cette fois-ci en effectuant un produit entre 2 matrices.
Indice : vous pouvez utiliser la clause `collapse` de la directive `distribute` pour distribuer un ensemble de boucles à différentes équipes.

III Sparse Matrix Vector kernel (SpMV)

Le kernel de multiplication vecteur-matrice creuse (SpMV) est un des calculs les plus utilisés en algèbre linéaire et dans les applications scientifiques. Il consiste à multiplier un vecteur x par une matrice A tel que $y = xA$. La matrice A est dite *creuse*, ce qui signifie qu'elle contient beaucoup de zéros. Ici, nous voulons explorer différents formats de stockage de la matrice A pour gagner en performance. Les figures 1 et 2 montrent deux formats utilisés pour compresser la matrice par rapport à sa représentation naïve.

Le premier format montré est le format de "ligne creuse et compressée" ou *Compressed Sparse Row* (CSR), ou encore *Compressed Row Storage* (CRS), voire "format Yale". Il consiste à stocker les données non-nulles à l'aide de trois tableaux. En notant N le nombre de lignes de la matrice, on a : un tableau avec les indices des colonnes de chaque éléments non-nuls, un tableau avec les valeurs des éléments non-nuls et un tableau `row` de taille $N+1$ correspondant aux indices dans les deux premiers tableaux du premier élément non-nul de chaque ligne. Ainsi, `row[N]` indique le nombre d'éléments non-nuls de la matrice et `row[i+1] - row[i]` indique le nombre d'éléments non-nuls de la ligne `i`.

D'autre part, le format de données Ellpack (surnommé ELL) est basé sur une technique de rembourrage (padding) afin de limiter la consommation mémoire d'une matrice creuse par rapport à un format creux tout en favorisant les accès vectorisés à la structure de données. En mémoire, la largeur de chaque ligne de la matrice est identique pour se caler sur la largeur maximum de la matrice de départ. Si une ligne possède un nombre d'éléments non nuls inférieurs à la largeur fixée (i.e. le max d'éléments non nuls par ligne), on rembourre alors cette ligne avec des éléments nuls. On rajoute un peu de données et du calcul par rapport à un format plus classique tel que le CSR, mais étant donné que le pattern d'accès à cette structure rend la possibilité de vectoriser facilement son code, on gagne en efficacité.

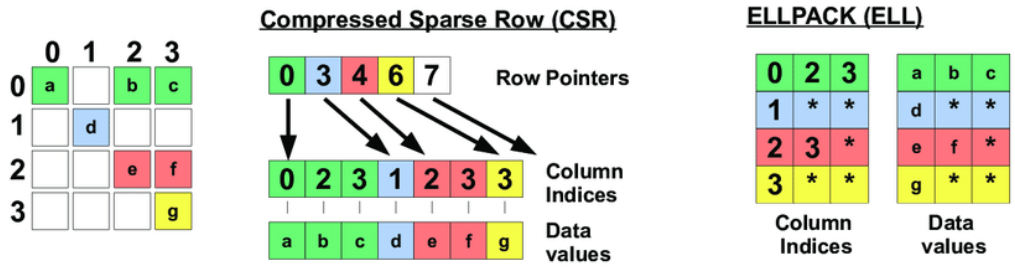
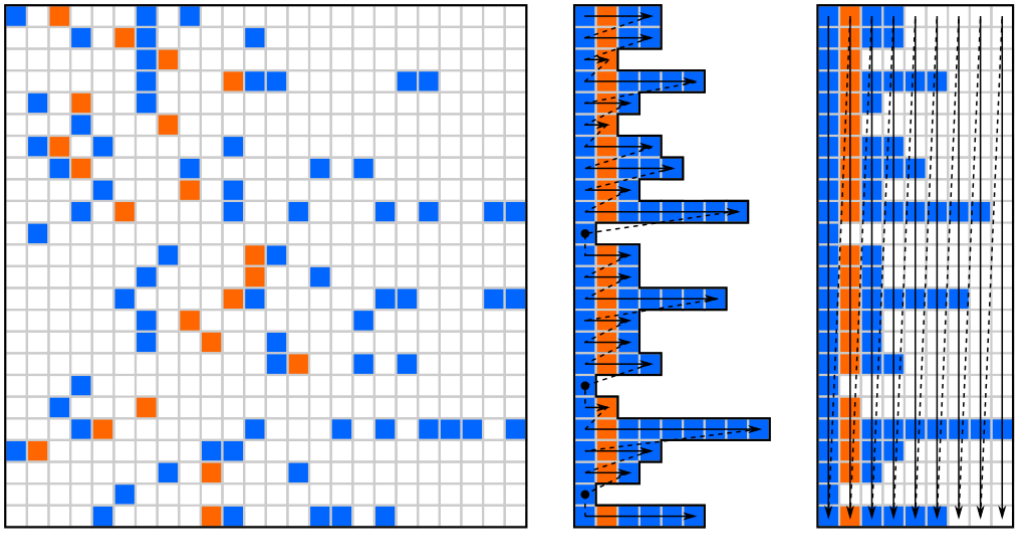


FIGURE 1 – Deux méthodes de compression d'une matrice creuse : CSR et ELL



(a) Source matrix (b) CSR (c) ELLPACK

FIGURE 2 – Pattern d'accès mémoire des formats CSR et ELL. Les flèches indiquent l'ordre de stockage dans la mémoire des valeurs des matrices et des indices des colonnes. Les éléments non-nuls orangés forment une colonne d'éléments dans les formats compressés. Dans le format CSR, ceux-ci ne sont pas stockés de manière successive.

Q.6: À partir du programme donné, écrivez une fonction exécutant le noyau de calcul **SpMV** avec le format CSR, puis en déportant cette fonction sur GPU. Les résultats vous semblent-ils suffisants ? Si non, pourquoi et comment le résoudre ?

Q.7: En prenant en compte l'exemple suivant, écrivez une fonction permettant de convertir une matrice CSR en format Ellpack.

Exemple : La matrice

$$A_{m,n} = \begin{pmatrix} a & 0 & 0 & b \\ 0 & c & 0 & 0 \\ 0 & d & e & 0 \\ f & 0 & g & h \end{pmatrix}$$

aura la représentation suivante :

$$m\_values = \begin{bmatrix} a & b & 0 \\ c & 0 & 0 \\ d & e & 0 \\ f & g & h \end{bmatrix}$$

$$m\_cols = \begin{bmatrix} 0 & 3 & 0 \\ 1 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 2 & 3 \end{bmatrix}$$

Q.8: À partir de ce nouveau format, proposez un code séquentiel effectuant le produit matrice-vecteur (SpMV).

Q.9: Parallélisez ce nouveau noyau grâce au support d'offload OpenMP et des directives target. Comparez par rapport aux résultats obtenus précédemment.