Université de Montréal

Calcul en n-dimensions sur GPU

par
Arnaud Bergeron

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

avril, 2013

RÉSUMÉ

Le calcul scientifique sur processeurs graphiques (GPU) est en plein essor depuis un certain temps, en particulier dans le domaine de l'apprentissage machine. Cette thèse présente les efforts pour établir une structure de données de tableau multidimensionnel de manière efficace sur GPU.

Nous commençons par faire une revue de ce qui est actuellement similaire dans le domaine et des désavantages d'avoir une multitude d'approches. Nous nous intéresserons particulièrement aux calculs fait à partir du langage Python.

Nous décrirons des techniques intéressantes telles que la réduction d'ordre et le calcul asynchrone automatique.

Pour terminer nous présenterons l'utilisation du module développé dans le cadre de cette thèse.

Mots clés: calcul scientifique, python, GPGPU

ABSTRACT

Scientific computing on GPU (graphical processing units) is on the rise, specifically in machine learning. This thesis presents the implementation of an efficient multidimensional array on the GPU.

We will begin by a review of what currently implements similar functionality and the disadvantage of a fragmented approach. We will focus on packages that have a Python interface.

We will explain techniques to optimize execution such as order reduction and automatic asynchronous computations.

Finally, we will present the functionality of the module developed for this thesis.

Keywords: scientific computing, python, GPGPU

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

REMERCIEMENTS

Je tiens à remercier les personnes qui m'ont aidé et inspiré dans l'écriture de ce mémoire.

- Frédéric Bastien, pour l'instigation du projet ;
- Pascal Vincent, pour ses conseils et suggestions ;
- Constance Cardin, pour ses corrections et son support ;
- Jean Bergeron, pour ses corrections et son support.

CHAPITRE 1

INTRODUCTION : CONTEXTE, OBJECTIF ET SURVOL DE L'ÉTAT DE L'ART

Python est un langage assez populaire dans la communauté du calcul scientifique pour l'interface avec l'utilisateur. La plupart des données traitées sont manipulées avec la bibliothèque NumPy [7] avant de se retrouver dans les engins de calcul. Cette bibliothèque est répandue parce qu'elle offre un format d'échange de données flexible sous la forme de son module `ndarray`. Ce module offre un tableau multidimensionnel, ou tenseur dont les vecteurs et les matrices sont des cas particuliers, avec une implémentation efficace sur processeur (CPU) pour les opérations numériques les plus courantes. Ils peuvent contenir plusieurs types de données structurées ce qui permet une grande variété d'algorithmes. La bibliothèque offre des fonctions de conversion pour transformer les données d'un format à un autre ce qui est parfois très utile pour intégrer plusieurs bibliothèques ensemble.

Récemment, les processeurs graphiques (GPU) ont augmenté en popularité pour le calcul scientifique dû au très bon rapport performance/prix qu'ils offrent. Pour des problèmes qui se conforment bien à leur architecture massivement parallèle, les calculs peuvent être 100 fois plus rapides que sur un CPU. Leur utilisation reste tout de même dans le domaine de l'arcane pour plusieurs puisqu'aucune grande bibliothèque de fonctionnalités similaire à ce qu'offre numpy n'est encore disponible. Ce vide est renforcé par la multitude de projets qui exposent un sous-ensemble de fonctionnalités, généralement incompatibles avec les autres projets existants. Cette fracture dans le support nuit grandement au partage de code et d'algorithmes entre les personnes intéressées. Nous cherchons donc à établir un module assez complet pour être utilisé comme implémentation de base, tout en restant relativement simple d'utilisation. Pour éviter de tomber dans le piège des nouveaux standards unifiants qui ne font qu'ajouter à la confusion, nous avons déjà contacté les gens qui maintiennent quelques-unes des implémentations existantes et qui désirent remplacer leur code par le notre.

1.1 Objectif

Afin de pouvoir exploiter facilement la puissance de calcul des GPU pour le calcul scientifique, nous avons besoin d'une bibliothèque qui offre une structure de tableau multidimen-

sionnel efficace, ayant la flexibilité du `ndarray` de NumPy, mais pour les GPU. Étant donné que la fonctionnalité de cette dernière est assez large nous allons nous concentrer sur certaines capacités clés. Ce sont les fonctionnalités de base dont nous avons besoin et sur lesquelles nous pourrons éventuellement construire des fonctionnalités plus avancées.

Types de données

La première de ces capacités est le support pour les éléments du tableau d'une grande variété de types de données et si possible des types fournis par l'utilisateur. Les premières implémentations de calcul sur carte graphiques ne supportaient que des données en virgule flottante de simple précision, mais depuis quelques années toutes les plateformes supportent une variété de types tels que les entiers et les nombres à virgule flottante double précision. Il faut donc que la bibliothèque supporte le plus de types de données possibles. Les types supportés sur une carte en particulier seront limités par les capacités du matériel.

Ordre et dimensions

L'*ordre* est le nombre de dimensions d'un tenseur ou tableau. Par exemple une matrice a un ordre de 2 puisqu'elle est en 2 dimensions. Les *dimensions* font référence à la taille de chaque dimension, ce qui dans le cas d'une matrice serait le nombre de lignes et de colonnes. Ensemble, ces deux informations déterminent le nombre exact d'éléments contenu dans le tableau. Nous voulons que le module supporte un ordre arbitraire avec des dimensions de taille arbitraire[1]. Cela permet à l'utilisateur d'exprimer son problème de la manière la plus naturelle possible, par exemple pour une liste d'images. Bien sûr, avec un ordre arbitraire, il est possible d'utiliser des vecteurs ou des matrices.

Décalage et pas

Certains algorithmes demandent de travailler sur un sous-ensemble des données, par exemple une sous-matrice. Afin de supporter cela, NumPy expose des vues qui sont un tableau qui fait référence à un sous-ensemble des éléments d'un autre tableau. Cette opération n'implique pas de copie, les deux tableaux partageant l'espace mémoire qui contient les données. Cela veut aussi dire que si les données sont modifiées à travers l'un des tableaux, les modifications vont

[1] dans les limites de la mémoire disponible

apparaitre dans l'autre. Pour réaliser ces vues les tableaux doivent supporter deux propriétés qui conditionnent tous les accès à leurs éléments, un décalage et des pas.

Le *décalage* est la distance entre le début de la région mémoire et le premier élément du tableau. Supposons que l'on a un vecteur $\mathbf{a} = \begin{bmatrix} 0 & 1 & 2 & 4 & 3 & 0 & 1 & 2 \end{bmatrix}$ et que l'on veuille travailler sur la deuxième moitié. Sans décalage, il faut faire quelque chose comme illustré sur

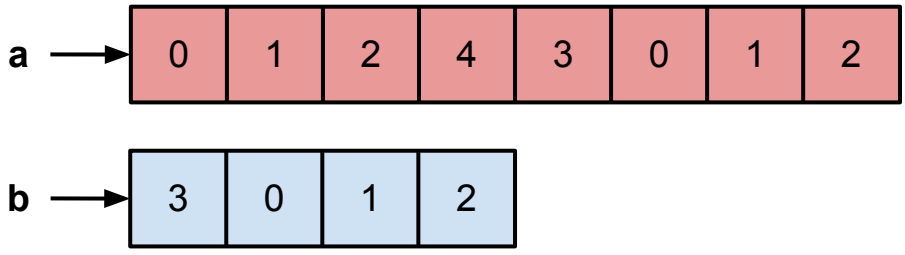


Figure 1.1 – Copie des éléments de **a**

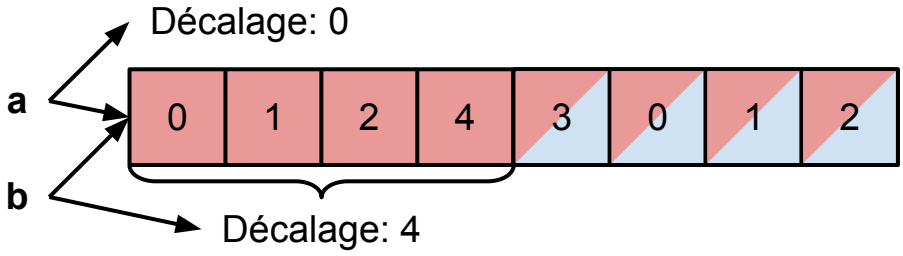


Figure 1.2 – Vue sur les éléments de **a**

la figure 1.1 où l'on copie les éléments sur lesquels on veut travailler. Avec un décalage on peut faire comme illustré sur la figure 1.2 où les deux vecteurs partagent la même région mémoire mais **b** ne contient qu'un sous-ensemble de **a**.

Le *pas* est la distance entre deux éléments du tableau dans une dimension donnée. Sur la figure 1.3 on voit l'illustration des pas sur une matrice 2x3 organisée selon la convention C[2]. Il peut être plus simple de comprendre le pas si l'on considère la version linéarisée du stockage en mémoire[3] comme à la figure 1.4.

[2]On indexe la ligne d'abord et la colonne ensuite.

[3]Cette vue est également celle que l'ordinateur a pour toute forme de stockage.

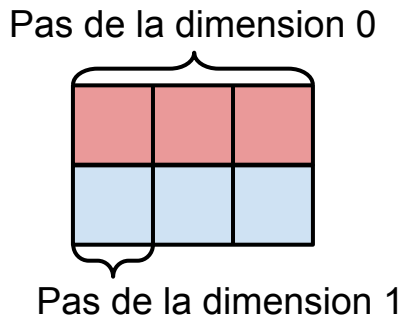


Figure 1.3 – Illustration des pas sur une matrice 2x3

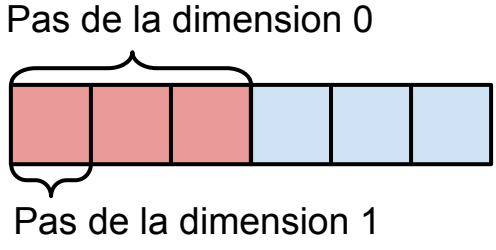


Figure 1.4 – Illustration des pas sur le stockage linéarisé d'une matrice 2x3

Si on combine ces deux propriétés on peut obtenir une vue sur une sous matrice comme illustré par la figure 1.5. La matrice **a** est de taille 4x5, son décalage est de 0 et ses pas sont

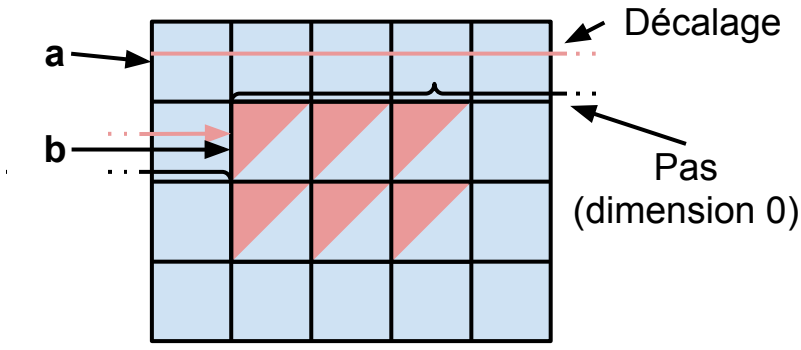


Figure 1.5 – Illustration des vues

5 et 1. La matrice **b**, qui représente le milieu de **a**, est de taille 2x3, son décalage est de 6, et ses pas sont 5 et 1. Le décalage et le pas de la dimension 0 sont indiqués pour la matrice **b** afin de bien illustrer ce à quoi ils correspondent. On peut également avoir un pas négatif ce qui, combiné à un décalage approprié, permet d'avoir une vue dans l'ordre inverse des éléments d'une dimension.

1.2 Implémentations existantes

Nous avons fait le tour des projets qui offrent un objet similaire à ce qui est décrit plus haut sur carte graphique. Étant donné que l'on fait un objet commun qui pourra être partagé à travers plusieurs langages, nous cherchions une implémentation en C, qui est un peu une langue véhiculaire[4] pour l'informatique moderne. Nous avons évité les implémentations commerciales et non-libres puisqu'elles peuvent difficilement remplir le rôle de base commune étant donné leur distribution restreinte.

Comme nous n'avons trouvé aucune implémentation de structure similaire à ce que l'on cherche, même en comptant les implémentations C++ pour lesquelles une interface C peut être réalisée, nous avons étendu notre recherche aux implémentations qui offraient une interface Python, puisque que c'est le premier langage ciblé. Pour ce langage nous avons trouvé les implémentations qui suivent.

[4]Langue véhiculaire : «langue de communication entre des communautés d'une même région ayant des langues maternelles différentes.» `http://www.larousse.fr/dictionnaires/francais/véhiculaire/81267`

cudanumpy

Ce projet [3] étend les structures de NumPy pour pouvoir exécuter certaines opérations sur les cartes NVIDIA. Il utilise le même modèle de calcul que NumPy et la performance de calculs complexes souffre, comme pour NumPy, de l'utilisation de tableaux temporaires. De plus, ce projet est intimement liés aux structures internes de NumPy et serait très difficile à adapter à d'autres langages.

Cudamat/gnumpy

Cudamat [5] est un projet Python qui permet l'utilisation de matrices de nombres à virgule flottante simple précision sur cartes NVIDIA et quelques opérations mathématiques avec celles-ci. La structure implémentée manque de flexibilité et le projet serait très difficile à adapter pour supporter un ordre arbitraire ou d'autres types pour les éléments.

Gnumpy [9] se base sur cudamat pour offrir un ordre arbitraire et présenter une interface similaire à NumPy. Tout comme cudanumpy, ce projet souffre de l'utilisation de tableaux temporaires dans la plupart des calculs. Il est entièrement implémenté en Python ce qui rend difficile sont adaptation à d'autres langages.

PyCUDA/PyOpenCL

Les deux projets [4] offrent un module `gpuarray` qui permet la manipulation de tableaux contigus uniquement. Comme ils sont faits pour refléter en Python l'interface spécifique de la bibliothèque qu'ils exposent, respectivement CUDA et OpenCL, il est difficile d'avoir une seule interface commune utilisant les deux projets interchangeablement.

CudaNdArray (Theano)

CudaNdArray (inclus dans Theano [2]) est l'implémentation qui se rapproche le plus d'un tableau multidimensionnel avec des vues flexibles tel qu'on le désire. La principale déficience est qu'elle ne supporte que les nombres à virgule flottante en simple précision et qu'elle ne fonctionne que sur CUDA. Ce projet est également fortement lié à NumPy donc serait difficile à utiliser à partir de d'autres langages que Python.

1.3 Description des chapitres suivants

Nous voyons qu'aucune des implémentations survolées n'offre l'ensemble des fonctionnalités de base désirées. Nous avons donc entrepris le développement d'une bibliothèque satisfaisant toutes ces exigences. Dans les chapitres qui suivent, nous commencerons par une introduction à l'architecture d'une carte graphique du point de vue de son utilisation pour le calcul. Nous présenterons ensuite la bibliothèque développée dans le but de supporter les fonctionnalités désirées avec une brève introduction à son utilisation et une présentation des optimisations mises en place. Nous terminerons par présenter l'utilisation de l'interface Python, développée dans le cadre du mémoire.

CHAPITRE 2

ARCHITECTURE D'UNE CARTE GRAPHIQUE

Une carte graphique est une unité indépendante à l'intérieur d'un ordinateur possédant son propre processeur et sa propre mémoire. Le processeur d'une carte graphique est composé d'un ensemble de multiprocesseurs ayant chacun un certain nombre d'unités d'exécution. À l'intérieur d'un multiprocesseur, chaque unité d'exécution exécute la même instruction avec des données différentes de manière similaire au SIMD[1]. La différence est que les multiprocesseurs des cartes utilisent le SIMD pour toutes les opérations qu'ils réalisent. Pour donner une idée des quantités impliquées, une carte graphique moderne, la GeForce GTX 680 a 48 multiprocesseurs comprenant chacun 32 unités d'exécution pour un total de 1536.

Les cartes graphiques se programment en écrivant une fonction dite *noyau* de calcul. L'exécution d'un noyau est composée d'un ensemble d'*instances de noyau* qui partagent les mêmes instructions mais reçoivent un indice différent ce qui leur permet de travailler sur des régions différentes de la mémoire. Ces instances sont organisées dans des *blocs* qui sont eux-mêmes organisés sur une *grille* (voir la figure 2.1 pour une illustration).

Chaque bloc est donné à un multiprocesseur qui répartit ensuite les instances de noyau du bloc sur ses unités d'exécutions. La tâche de répartition est assez complexe puisqu'elle implique de changer d'instances de noyau, possiblement entre chaque instruction, afin de masquer le temps des lectures et écritures mémoire. On peut simplifier cette tâche en ayant beaucoup d'instances par bloc ce qui donne beaucoup de flexibilité à la répartition.

Même si l'écriture de noyaux de calcul se fait dans un langage de plus haut niveau fait pour cacher certains de ces détails, il faut en tenir compte pour écrire du code performant.

2.1 Anatomie d'une exécution

Lorsqu'on veut réaliser un calcul sur une carte graphique, on doit d'abord écrire un noyau de calcul. Celui-ci sera envoyé, avec les données que l'on veut traiter, sur la mémoire de la carte pour être exécuté à un moment ultérieur. Un programme doit lancer l'exécution du noyau en

[1] Single Instruction Multiple Data : une instruction qui fait le même travail sur plusieurs données en même temps

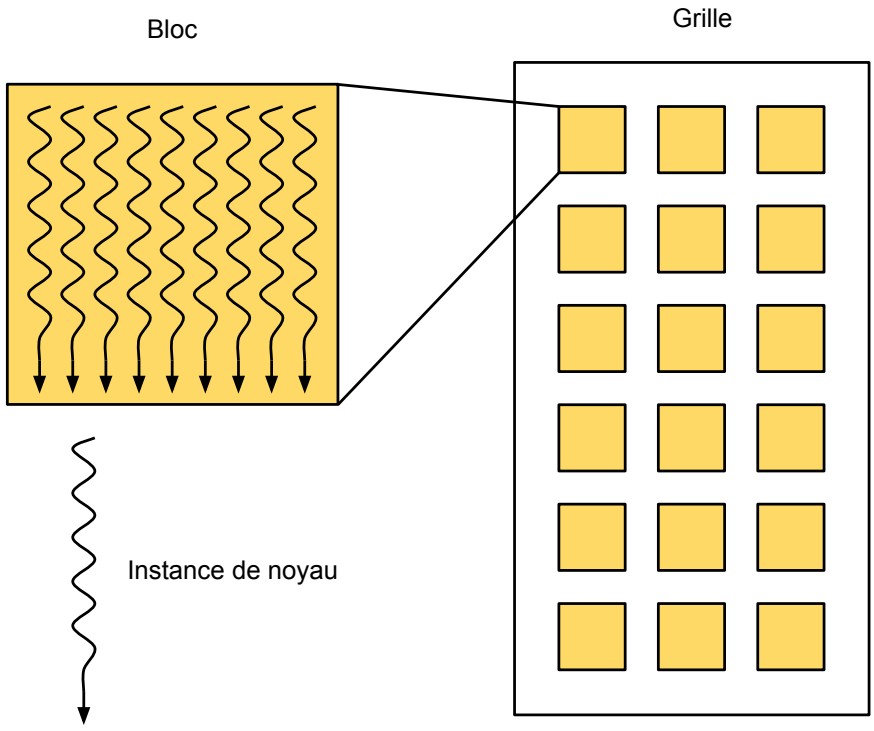


Figure 2.1 – Illustration des blocs, des grilles et des instances de noyau

fournissant ses paramètres d'exécution (arguments, taille de la grille et des blocs). Le calcul se fait de manière asynchrone et indépendante du processeur central une fois lancé. Une fois le calcul complété, le programme peut soit lancer d'autres calculs soit récupérer les résultats en les copiant en mémoire centrale.

2.1.1 Écriture de noyaux

Les deux plateformes principales en ce moment sont OpenCL[8], un standard ouvert adopté par la plupart des vendeurs de processeurs et de cartes graphiques, et CUDA[6], la plateforme propriétaire de NVIDIA pour le calcul sur ses cartes graphiques. Les deux utilisent un dialecte de C pour l'écriture des noyaux avec quelques fonctions ajoutées qui exposent les spécificités du matériel et de l'environnement.

Les noyaux sont écrits comme un calcul sur une petite portion des données avec la compréhension qu'il sera exécuté plusieurs fois en parallèle afin de faire le calcul sur toutes les données. Un noyau ne retourne pas de résultat comme pour une fonction ordinaire mais écrit

plutôt dans un ou plusieurs de ses arguments, selon le problème. La communication avec les autres instances du noyau n'est possible que pour des sous-ensembles d'instances, il est donc impossible de faire de la synchronisation globale sans séparer le calcul en plusieurs noyaux.

2.1.2 Transfert vers la carte

Une fois que le noyau est écrit et transféré sur la carte, il faut aussi transférer les données sur lesquelles il va travailler. Cette étape est généralement simple à réaliser mais peut prendre beaucoup de temps à cause des délais de transfert. Pour avoir une bonne performance il est essentiel de limiter les transferts entre la mémoire centrale et la carte graphique. Il est possible à cet effet de réutiliser des résultats d'une exécution précédente comme entrée pour une nouvelle exécution.

2.1.3 Lancement de l'exécution

Une fois que tout est transféré, on doit configurer le lancement du noyau. On spécifie la taille désirée pour les blocs et la grille[2]. La taille de ces blocs est idéalement un multiple de la taille des unités d'exécutions. Il est possible de choisir une autre taille mais la performance sera moindre.

2.1.4 Résultats

Une fois le calcul lancé, il faut attendre qu'il soit terminé. Le programme sur le CPU peut faire autre chose durant ce temps, comme préparer le calcul suivant. Une fois que l'on a vérifié que le calcul sur la carte graphique est terminé on peut transférer les résultats en mémoire centrale afin de les utiliser. On peut aussi les laisser sur la carte pour les utiliser comme entrée dans un calcul ultérieur. Il faut faire attention de bien vérifier que les calculs soient terminés avant de manipuler les tableaux d'entrée et de sortie pour éviter des résultats erronés.

2.2 Exemple de noyau

Le code de noyau suivant effectue le calcul $a+1$ sur un tableau contigu.

[2] à l'intérieur des limites du matériel

```
KERNEL void ap1(const unsigned int n,
                float *a, float *res) {
    const unsigned int idx = LDIM_0 * GID_0 + LID_0;
    const unsigned int numThreads = LDIM_0 * GDIM_0;
    unsigned int i;

    for (i = idx; i < n; i += numThreads) {
        res[i] = a[i] + 1;
    }
}
```

Avant de lancer le noyau, il faut transférer les données sur lesquelles ont veut travailler et allouer un tableau de résultats. Ensuite on passe ces tableaux en arguments au noyau. Dans le code, `GID_0` représente l'indice de grille et `LID_0` l'indice de bloc. `GDIM_0` et `LDIM_0` représentent respectivement la taille de la grille et des blocs. Ces valeurs sont abstraites par des macros pour permettre au même code de noyau de fonctionner sous CUDA et OpenCL, mais correspondent à des paramètres implicites que reçoivent chaque instance de noyau.

Le noyau comprend une boucle afin de s'assurer de traiter tous les éléments du tableau d'entrée (dont la taille est fournie avec le paramètre `n`) dans le cas où ce tableau aurait plus d'éléments que la taille maximale de la grille et des blocs. Le calcul de l'adressage respectant le décalage et les pas est plus complexe et sera présenté à la section 3.2.

2.3 Fonctions disponibles sur carte graphique

Voici une liste des fonctions mathématiques que l'on peut employer dans les noyaux sur tout type de carte : `acos`, `acosh`, `asin`, `asinh`, `atan`, `atanh`, `cbrt`, `ceil`, `copysign`, `cos`, `cosh`, `cospi`, `erf`, `erfc`, `exp`, `exp10`, `exp2`, `expm1`, `fabs`, `fdim`, `floor`, `fma`, `fmax`, `fmin`, `fmod`, `frexp`, `hypot`, `ilogb`, `isinf`, `isnan`, `ldexp`, `lgamma`, `log`, `log10`, `log1p`, `log2`, `logb`, `modf`, `nan`, `nextafter`, `pow`, `remainder`, `remquo`, `rint`, `round`, `rsqrt`, `sin`, `sincos`, `sinh`, `sinpi`, `sqrt`, `tan`, `tanh`, `tgamma`, `trunc`. En plus de ces fonctions tous les opérateurs C usuels (comparaison, logique, ternaire) sont disponibles.

CHAPITRE 3

BIBLIOTHÈQUE IMPLÉMENTÉE

La structure implémentée pour ce mémoire est en deux niveaux. Le premier niveau s'occupe de l'interface avec l'environnement d'exécution qui permet les calculs sur la carte. Le deuxième utilise le premier pour offrir des fonctionnalitées plus avancées. Cette structure permet d'adapter rapidement le code pour supporter un nouvel environnement de calcul si nécessaire.

3.1 Premier niveau : interface avec l'environnement de calcul

Le premier niveau sert d'adapteur pour offrir une interface identique vers les différents environnements de calculs tels que CUDA et OpenCL. Il expose un ensemble de fonctions permettant de réaliser les opérations nécessaires à l'utilisation d'un dispositif de calcul SIMD tel qu'offert par les cartes graphiques.

Voici une brève exposition des opérations sous la forme d'introduction, à leur utilisation. Pour une description complète et détaillée des différentes fonctions voir l'annexe I (en anglais).

La première tâche est de sélectionner l'implémentation que l'on utilise, pour obtenir un vecteur d'opérations.

```
compyte_buffer_ops *ops = compyte_get_ops("opencl");
```

Les deux implémentations existantes sont `"cuda"` et `"opencl"` correspondant aux deux environnements principaux pour le calcul. Une fois ce vecteur obtenu on doit initialiser un contexte.

```
void *ctx = ops->buffer_init(0, NULL);
```

Avec ce contexte on peut réaliser les diverses opérations offertes par l'interface. La première devrait logiquement être d'allouer des blocs mémoire dans le contexte (ici on alloue 4 mégaoctets).

```
gpudata *buf = ops->buffer_alloc(ctx, 1024*1024*4, NULL);
gpudata *out = ops->buffer_alloc(ctx, 1024*1024*4, NULL);
```

On peut ensuite transférer des données de la mémoire centrale vers le bloc alloué.

```
void *data = ...
ops->buffer_write(buf, 0, data, 1024*1024*4);
```

Ou encore on peut initialiser la mémoire à une valeur fixe.

```
ops->buffer_memset(out, 0, 0);
```

On peut également créer des noyaux à partir de code source.

```
const char *KERNEL_SOURCE =
    "KERNEL void cpy(GLOBAL_MEM float *out,"
    "GLOBAL_MEM float *buf) {"
    "const unsigned int i = LDIM_0 * GID_0 + LID_0;"
    "out[i] = buf[i];"
    "}";
gpukernel *k = ops->buffer_newkernel(ctx, 1,
                                     &KERNEL_SOURCE,
                                     NULL, "cpy",
                                     GA_USE_CLUDA, NULL);
```

Avant d'appeler ce noyau, il faut spécifier ses arguments.

```
ops->buffer_setkernelarg(k, 0, GA_BUFFER, out);
ops->buffer_setkernelarg(k, 1, GA_BUFFER, buf);
```

On peut maintenant appeler le noyau.

```
ops->buffer_callkernel(k, 1024, 1024);
```

Finalement on récupère les résultats.

```
void *res = ...
ops->buffer_read(res, out, 0, 1024*1024*4);
```

Quelques autres opérations sont implémentées à ce niveau, notamment pour permettre l'interopération avec une base de code existante.

3.2 Deuxième niveau : tableau multidimensionnel

Le deuxième niveau utilise le premier pour implémenter une structure de tableau complète. C'est à ce niveau que l'on a l'information sur l'ordre et la taille des dimensions, le décalage et

les pas à l'intérieur d'un bloc de mémoire.

```
typedef struct {
    gpudata *data;
    compyte_buffer_ops *ops;
    size_t *dimensions;
    ssize_t *strides;
    unsigned int nd;
    int flags;
    int typecode;
} GpuArray;
```

L'allocation prépare les éléments de la structure et réserve un bloc mémoire sur la carte graphique. On peut ensuite lire et écrire dans le tableau, copier ses données dans un autre, prendre des vues et changer l'ordre et la taille des dimensions. C'est également à ce niveau que l'on peut faire de l'indexation qui permet d'avoir une vue sur une sous-région d'un autre tableau.

La création de noyaux est assez similaire au premier niveau, excepté que l'on peut enregistrer ses propres types de données pour les utiliser. Par contre dans les manipulations de la mémoire que l'on fait à l'intérieur du noyau il faut tenir compte de la structure du tableau. Il est possible que les éléments soit disposés de manière assez complexe en mémoire. On peut retrouver leur position grâce à un *calcul d'adressage*, qui retrouve la position en mémoire d'un élément du tableau à partir d'un indice de grille et de bloc qui est automatiquement transmis à toutes les instances de noyau lors de l'exécution. Ce calcul est réalisé par l'algorithme 1.

Comme le nombre de dimensions est toujours connu par les diverses fonctions qui génèrent des noyaux, la boucle sur d est toujours déroulée dans le code des noyaux. Puisque les instances de noyau ne peuvent pas communiquer entre elles, chacune doit réaliser son propre calcul d'adressage. Ceci fait que le temps utilisé pour réaliser ce calcul a un impact important sur la performance.

Algorithme 1 Calcul d'adressage

$ii \leftarrow idx$
$addr \leftarrow base + décalage$
for $d = \text{nd} - 1 \rightarrow 0$ **do**
  if $d > 0$ **then**
    $pos \leftarrow ii \bmod \mathbf{dim}_d$
    $ii \leftarrow ii / \mathbf{dim}_d$
  else
    $pos \leftarrow ii$
  end if
  $addr \leftarrow addr + pos \cdot \mathbf{str}_i$
end for

3.3 Optimisation

Pour masquer le coût du calcul d'adressage et autres sources de temps perdu, nous avons recours à plusieurs optimisations. Certaines, comme la réduction d'ordre sont spécifique au calcul élément par élément et d'autres, comme l'exécution asynchrone, s'appliquent de manière plus générale.

Dans cette section la plupart des optimisations sont activées ou désactivées explicitement ce qui empêche le code qui gère les noyaux de faire ses décisions heuristiques habituelles. Cette fonctionnalité est très pratique pour faire des mesures fiables, mais donne parfois des cas où certaines optimisations nuisent à la performance. Généralement, il est beaucoup plus avisé de laisser les heuristiques faire pour la sélections d'optimisations appropriées à la situation.

3.3.1 Procédure de mesure

Toutes les mesures dans les sections qui suivent sont effectuées à chaud et ne comptent ni le temps de compilation des noyau, ni le temps de transfert des entrées et sorties des divers noyaux. La procédure commence par quelques exécutions du noyau à mesurer pour mettre le code en cache et s'assurer que tout est prêt. Ensuite une boucle de calibration est effectuée pour obtenir une estimation du temps d'exécution du noyau. Le nombre d'exécutions pour la boucle de mesure est choisi pour avoir un temps total entre 1 et 10 secondes. Cette boucle est effectuée 5 fois et la fonction rapporte la moyenne, la variance ainsi que les extrémums. Les graphiques sont dessinés en utilisant la moyenne et l'écart-type.

3.3.2 Environnement de mesure

Les machines utilisées pour les mesures ont étés choisies pour représenter une variété d'environnements :

Machine 1 est une station de travail Linux (FC14) avec un processeur Intel Core i7-2600K (3.4Ghz) et 16 Go de mémoire. Le dispositif de calcul mesuré est une carte Nvidia GeForce GTX 680 avec 2 Go de mémoire, CUDA version 5.0.35 et la version 319.17 des pilotes. La carte mesurée n'est pas utilisée pour l'affichage de l'écran.

Machine 2 est une machine de jeux roulant sous Windows 7 (64-bit) avec un processeur AMD Athlon II X3 455 (3.3Ghz) et 4 Go de mémoire. Le dispositif de calcul mesuré est une carte AMD Radeon HD 6870 avec 1 Go de mémoire, AMD APP SDK version 2.8 et la version 13.1 des pilotes. La carte mesurée est utilisée pour l'affichage de l'écran.

Machine 3 est une machine de travail sous Mac OS X 10.7.5 avec un processeur Intel Core 2 Duo (E8335 @ 2.93 Ghz) et 8 Go de mémoire. Le dispositif de calcul mesuré est le processeur par l'entremise de l'implémentation d'OpenCL qui vient avec le système d'exploitation. Le système n'était pas chargé durant les mesures, mais n'était pas autrement inactif.

3.3.3 Expressions mesurées

Chaque mesure est effectuée sur trois expressions afin d'avoir un éventail de possibilités.

- ```
  ElemwiseKernel(None, "float *o, float *a",
                 "o[i] = a[i] + 1")
  ```

  $a+1$, sert surtout à voir l'effet des optimisations sur le temps occupé en dehors des calculs noyaux. Ce temps est surtout remplit par les calculs d'adressage, mais contient aussi les préparations au lancement du noyau et l'application de certaines optimisations.

- ```
  ElemwiseKernel(None, "float *d, float *m,"
                 "float *a, float *b",
                 "d[i] = floor(a[i] / b[i]),"
                 "m[i] = fmod(a[i], b[i])")
  ```

divmod, sert à démontrer un calcul un peu plus complexe. Elle calcule dans le même appel de noyau la division et le modulo pour deux tableaux de nombres et retourne les deux résultats. Le temps de ce noyau est plus balancé entre le calcul et le reste.

- `ElemwiseKernel(None, "float *o, float *x",`
  `"o[i] = x[i] - pow(x[i], 3)/6 +"`
  `"pow(x[i], 5)/120 - pow(x[i], 7)/5040 +"`
  `"pow(x[i], 9)/362880")`

  série de taylor pour sinus, démontre l'effet des optimisations sur un long calcul. La majorité du temps de ce noyau est passée dans le calcul lui-même.

3.4 Réduction d'ordre

La réduction d'ordre permet, avant l'exécution d'un noyau élément par élément, de fusionner les dimensions contiguës pour réduire la complexité du calcul d'adressage[1]. On trouve la taille globale du calcul, qui est égale à la taille d'un des tableaux impliqués dans le calcul, puisque tous les tableaux doivent avoir la même taille pour cette sorte de noyau.

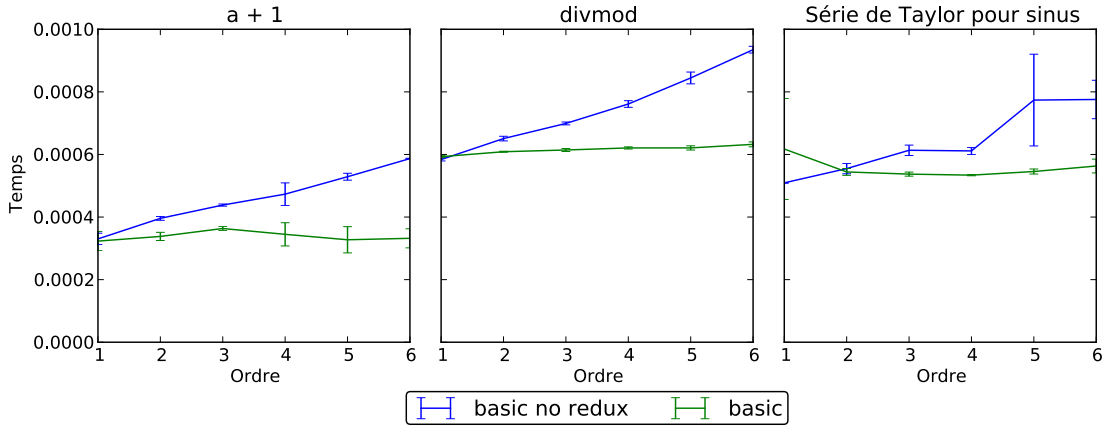
Ensuite, on commence par éliminer les dimensions de taille 1. Donc, pour une matrice 1x4, on la considère comme un vecteur de taille 4 pour les fins de calculs. Puis, on élimine les dimensions qui se suivent et dont les éléments sont contigus en mémoire pour les remplacer par une seule dimension qui a comme taille le produit de toutes celles qu'elle remplace. Par exemple, une matrice 2x3 contiguë sera remplacée par un vecteur de taille 6 (une autre vue sur les mêmes éléments).

À la fin, on obtient des tableaux avec un ordre moindre qui font référence aux mêmes données et dans le même ordre que les tableaux originaux. Comme la structure précise des éléments importe peu pour le calcul élément par élément, on y gagne par l'élimination de calculs d'adressage inutiles.

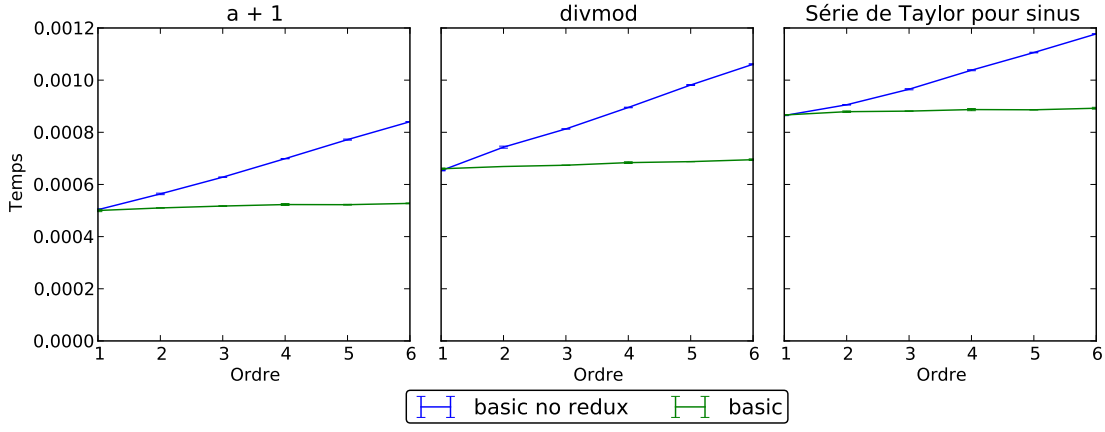
Si on examine la figure 3.1 on voit que l'avantage de la réduction d'ordre croit linéairement en fonction du nombre de dimensions qui peuvent être éliminées. Pour ces mesures les tableaux d'entrée sont tous contigus et peuvent donc être réduits à une vue de une seule dimension. Pour des tableaux non-contigus, l'amélioration du temps d'exécution serait similairement proportionnelle au nombre de dimensions ayant pu être éliminées.

Figure 3.1 – Mesure de l'efficacité de la réduction d'ordre

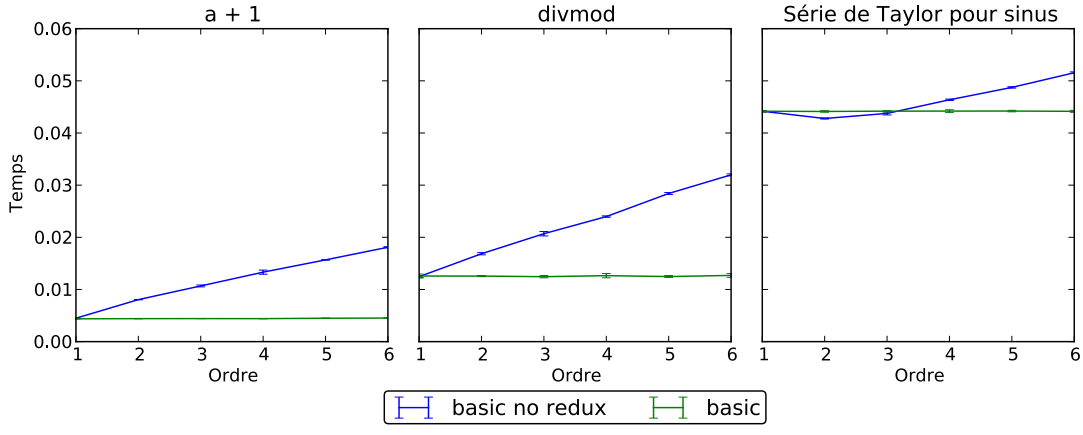
(a) Machine 1 - GTX 680



(b) Machine 2 - Radeon HD 6870



(c) Machine 3 - Intel Core 2 Duo



3.5 Spécialisation

Lors de la génération de noyaux, il est possible de considérer comme constant une partie des paramètres dont dépendent l'adressage. Générer du code en fonction de valeurs spécifiques permet une exécution plus efficace, mais empêche de réutiliser le code pour un calcul avec des paramètres différents. Pour ce faire, plusieurs niveaux de spécialisation (ou patrons) sont disponibles :

basic Ce patron considère uniquement l'ordre du tableau comme constant laisse varier tous les autres paramètres, (taille des dimensions, pas des dimensions, décalage du tableau). Le même code peut être partagé par tous les tableaux ayant le même ordre ce qui permet une réutilisation assez large.

dimspec Ce patron considère comme constant la taille des dimensions en plus de l'ordre. Le même code peut être partagé par tous les tableaux ayant les mêmes tailles de dimensions.

specialized Ce patron considère comme constant tous les paramètres d'un tableau (excepté les données, bien sûr). Le même code ne peut être partagé qu'entre tableaux dont la disposition mémoire est identique. Comme il connait tous les paramètres d'un tableau lors de la génération du code d'adressage, il peut complètement omettre des morceaux de code inutiles comme l'ajout du décalage si celui-ci est de zéro.

contig Ce patron considère que le tableau est contigu et évite ainsi de faire les calculs d'adressages. Le même code peut être partagé pour tous les tableaux contigus, indépendamment de leur ordre ou de la taille de leurs dimensions.

Les appels ordinaires à un noyau font une sélection du meilleur patron à utiliser en fonction des caractéristiques des arguments. Si tous les arguments sont des tableaux contigus, la version **contig** est utilisée. Sinon, lors d'un premier appel, on utilise la version **basic** puisqu'elle est réutilisable. Après quelques appels avec le même ordre et les même tailles de dimensions, la version **dimspec** est employée. Si plusieurs appels se suivent avec des paramètres identiques, on choisit la version **specialized**. Cette gradation est faite dans le but d'éviter de recompiler des noyaux pour chaque appel, ce qui peut être très coûteux et annulerait complètement l'avantage qu'ont les versions plus spécialisées. Par contre, si l'utilisateur emploie un noyau dans une

boucle avec des arguments similaires, après quelques itérations, le code devrait utiliser automatiquement la version la plus rapide. Si l'utilisateur connait d'avance son type d'utilisation, il est possible de choisir directement une des versions.

On peut voir l'effet des différents niveaux de spécialisation à la figure 3.2. On peut voir que le niveau **specialized** est parfois plus rapide pour un petit ordre, mais il ne faut pas oublier que ces mesures n'incluent pas le temps de compilation des noyaux. Comme sur certaines plateformes le temps de compilations peut atteindre quelques secondes, il faut quelques milliers d'appels avant de commencer à voir des avantages.

3.6 Exécution asynchrone

Tous les noyaux exécutés par la bibliothèque le sont de manière asynchrone. Comme une partie de l'exécution est passée à préparer les arguments et s'assurer que tout est conforme pour le lancement, cette partie du travail peut être effectuée alors que le noyau de l'appel précédant termine de s'exécuter.

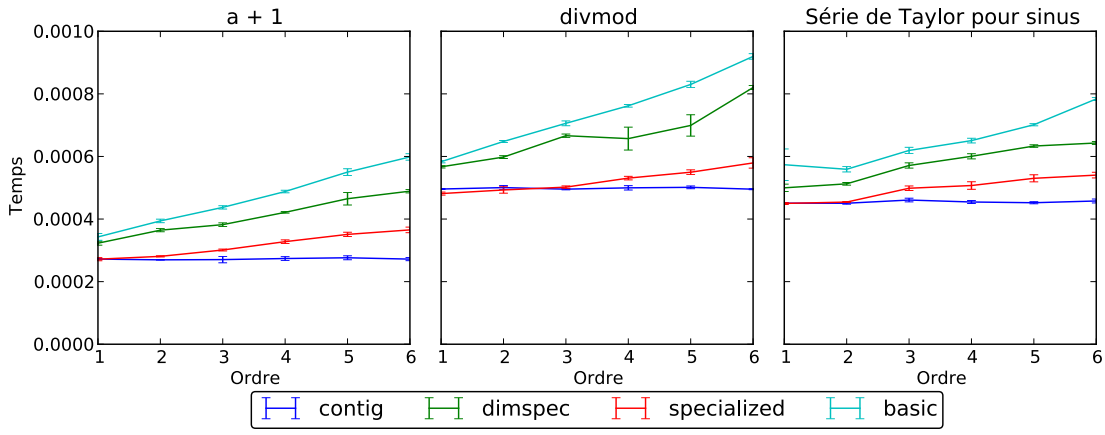
Une synchronisation est automatiquement effectuée lorsque nécessaire pour s'assurer de préserver l'illusion de travail synchrone. Ainsi les noyaux qui travaillent sur des tableaux en commun sont automatiquement marqués pour qu'ils s'exécutent dans le même ordre que celui où ils sont soumis. Les transferts entre la mémoire centrale et le dispositif se font de manière synchrone et attendent que toutes les opérations précédentes impliquants les tableaux concernés soit terminées.

Il est également possible de synchroniser manuellement, si l'interopération avec une bibliothèque existante est désirée. La figure 3.3 montre l'incidence de l'exécution asynchrone sur le temps mesuré d'exécution des noyaux.

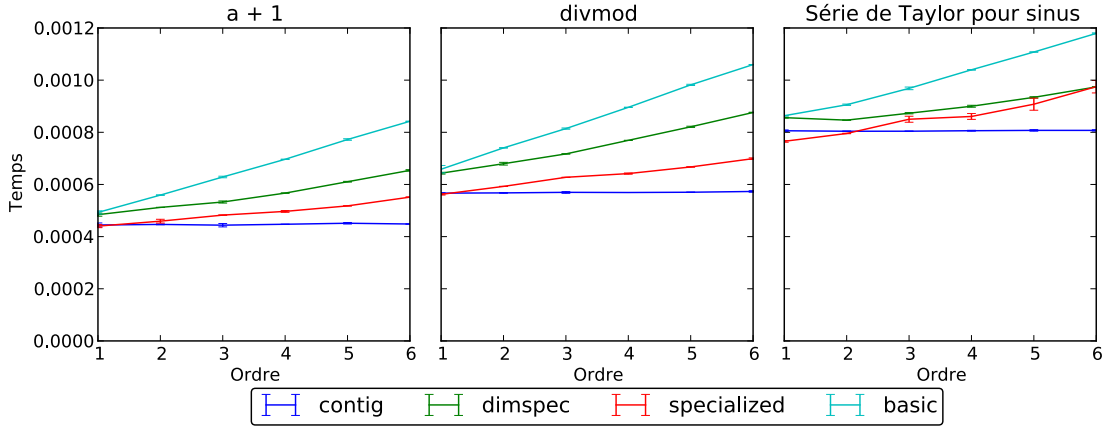
On peut voir aussi que les exécutions sur la machine 3 profitent beaucoup moins de l'exécution asynchrone puisque toutes la parties du programme roulent sur le processeur et donc ne peuvent pas trop se chevaucher. On voit aussi que plus les tableaux impliqués sont grand plus l'avantage est réduit puisque que le temps de préparation est fixe et diminue proportionnellement par rapport au temps d'exécution des noyaux. Si une application effectuait un autre travail qu'appeler des noyaux dans une boucle, les autres opérations pourraient aussi s'effectuer en même temps que les noyaux.

Figure 3.2 – Mesure de l'efficacité des différents niveaux de spécialisation

(a) Machine 1 - GTX 680



(b) Machine 2 - Radeon HD 6870



(c) Machine 3 - Intel Core 2 Duo

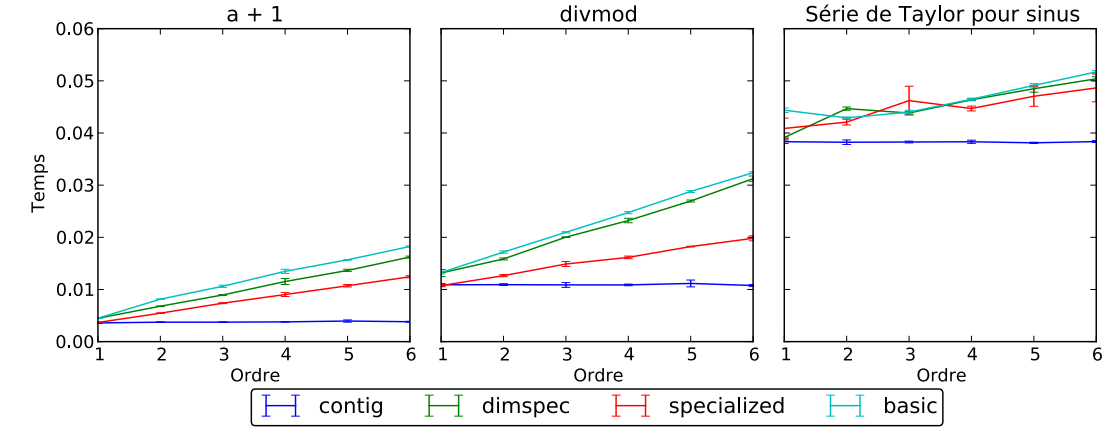
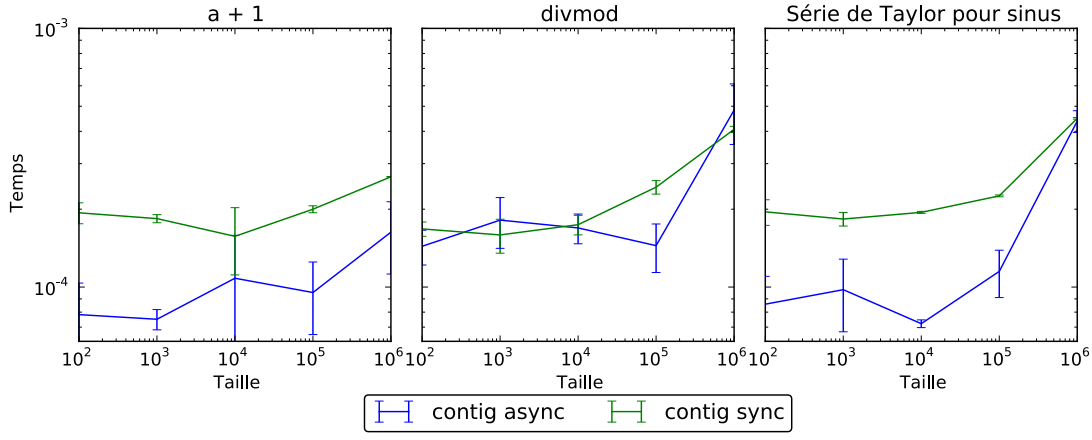
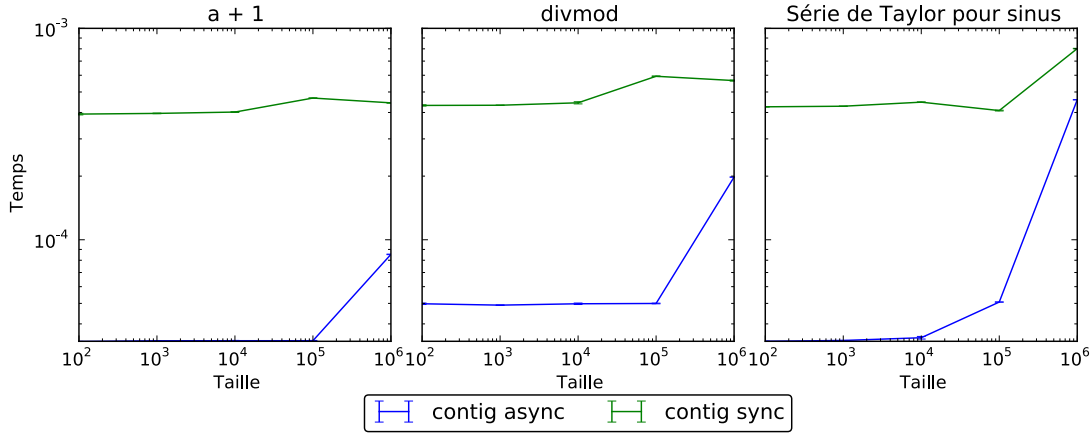


Figure 3.3 – Mesure de l'incidence de l'exécution asynchrone

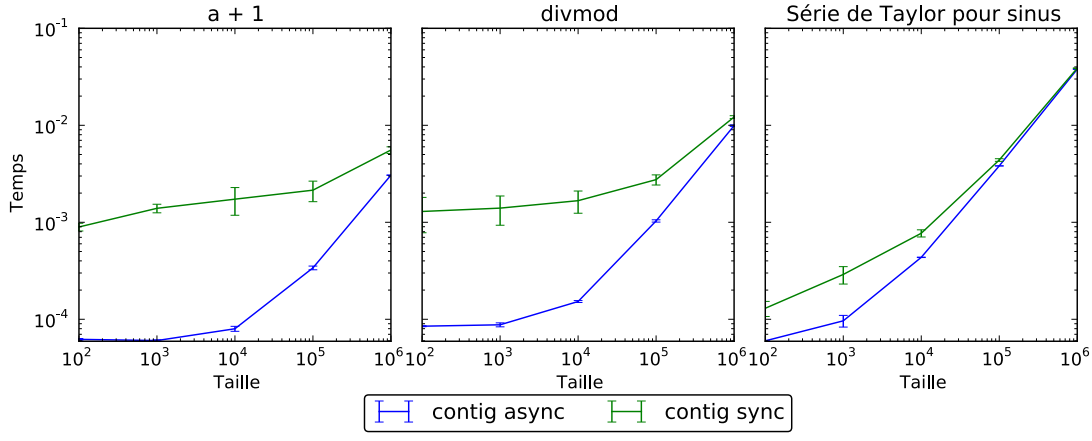
(a) Machine 1 - GTX 680



(b) Machine 2 - Radeon HD 6870



(c) Machine 3 - Intel Core 2 Duo



3.7 Adaptation automatique

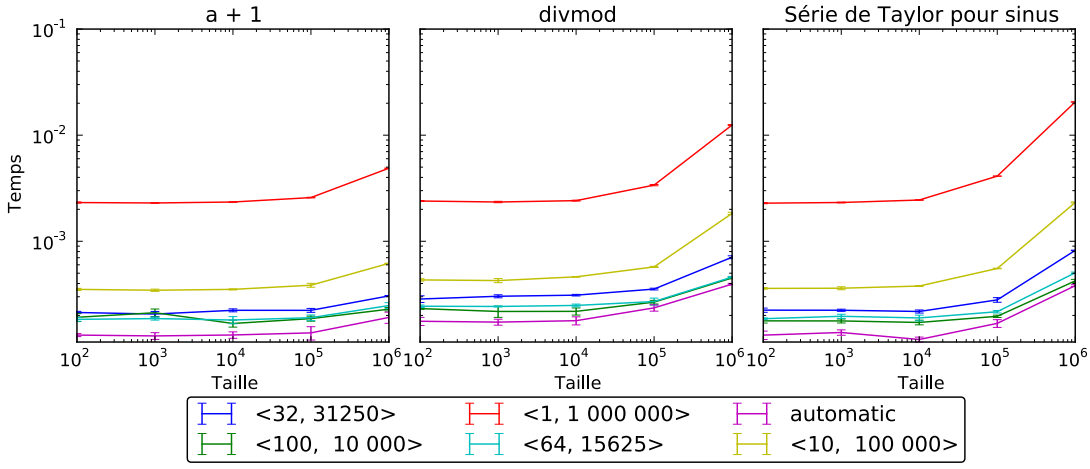
La dernière optimisation est l'adaptation automatique aux divers dispositifs de calculs. On spécifie le nombre d'instances du noyau que l'on veut rouler et la bibliothèque s'occupe de choisir une taille de bloc et de grille optimale pour le dispositif actuel selon une heuristique. Il faut faire attention en utilisant cette fonctionnalité puisqu'il est possible que les tailles choisies donnent lieu à plus d'instances que le nombre demandé, pour des raisons de performance. Par exemple si on effectue des calculs sur un tableau de 1000 éléments, l'heuristique va fort probablement choisir un grille de taille 1 avec des blocs de taille 1024, ce qui nous donne 24 éléments de plus que la taille du tableau. Ces éléments additionnels sont insérés pour respecter le multiple demandé de la carte qui est habituellement au moins 32. Il faut donc que les noyaux connaissent la taille de la tâche à accomplir et s'assurent d'être à l'intérieur des limites avant d'effectuer un travail. Il est également possible de spécifier uniquement la taille des blocs ou de la grille et de laisser la bibliothèque remplir les valeurs manquantes. Si un algorithme a des contraintes spécifiques sur les tailles de ses blocs ou de sa grille, toutes les propriétés utilisées dans le calcul sont exposées afin de permettre une heuristique différente.

La figure 3.4 donne une idée de la différence de performance que peut donner une mauvaise sélection de taille ou une sélection fixe. On peut voir que pour des cas extrêmes où une très mauvaise sélection est faite, l'exécution peut prendre 10 fois plus de temps sur les cartes graphiques. La performance sur processeur est moins directement affectée par cette sélection et dépend surtout de ce que le système fait en plus du noyau. Dans un système d'exploitation général il est difficile de contrôler le reste de la machine afin d'avoir un graphique cohérent.

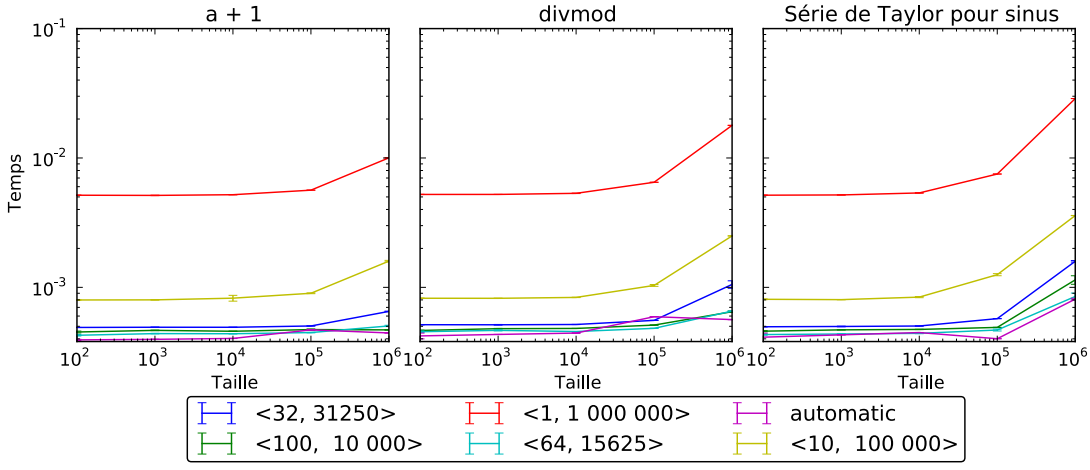
L'adaptation automatique permet d'avoir une bonne performance, tout en restant flexible, évitant un problème courant de noyaux restreints à certains dispositifs à cause des ressources qu'ils consomment, ou souffrant de performance inadéquate sur les dispositifs plus récents parce que leurs paramètres de lancement sont fixes.

Figure 3.4 – Mesure de la performance pour diverses tailles de grille et blocs

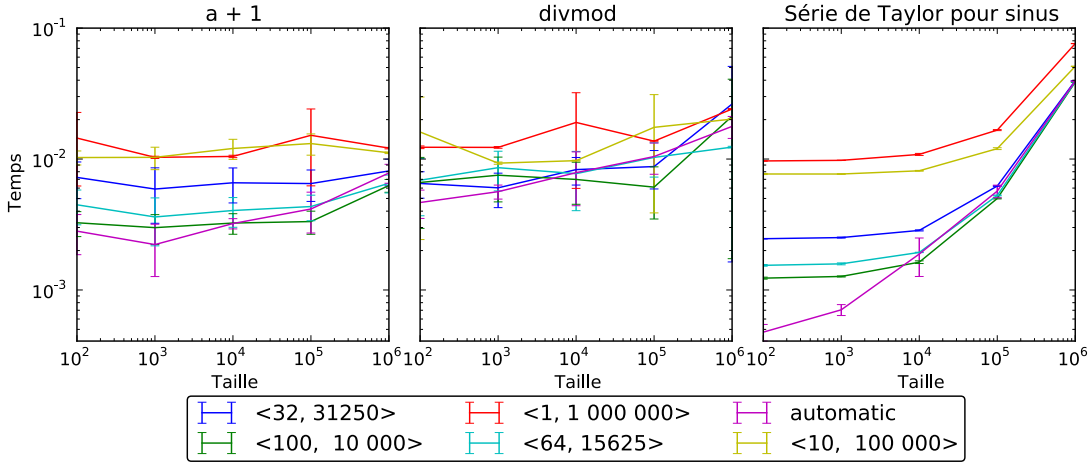
(a) Machine 1 - GTX 680



(b) Machine 2 - Radeon HD 6870



(c) Machine 3 - Intel Core 2 Duo



CHAPITRE 4

UTILISATION DU MODULE PYTHON

Ce chapitre introduit l'utilisation du module `pygpu` sous la forme d'un tutoriel, en exposant les fonctionnalités. Pour un description plus poussée des diverses fonctions et classes introduites, faire référence à l'annexe II.

Les sections devraient généralement être lues dans l'ordre puisqu'elles font référence aux notions exposées dans les précédentes.

4.1 Initialisation

Avant d'utiliser le module, il faut se créer un contexte. La version simplifiée utilise des chaînes pour identifier un dispositif. Les chaines sont du genre `"cuda0"` ou `"opencl0:1"`. La première portion de la chaîne (`"cuda"` ou `"opencl"`.) indique l'environnement que l'on veut utiliser et les chiffres après indiquent le numéro du dispositif à utiliser pour le contexte.

On crée un contexte avec le code suivant :

```
ctx = pygpu.init("cuda1")
```

L'objet retourné, `ctx` n'est pas modifiable, mais a certaines propriétés que l'on peut consulter, notamment `ctx.devname` qui est une chaîne représentant le nom du dispositif désigné par ce contexte.

Lorsque l'on travaille avec un seul contexte, on peut appeler

```
pygpu.set_default_context(ctx)
```

afin d'éviter de passer en paramètre le contexte avec lequel presque toutes les autres fonctions travaillent.

Il est possible de fonctionner avec plusieurs contextes dans la même application mais ils fonctionnent comme des silos séparés.

4.2 Allocation

Pour travailler avec des données, il faut ensuite s'allouer un ou plusieurs tableaux. La manière la plus simple est d'utiliser la fonction `pygpu.array` qui convertit une structure ressem-

blant à un tableau en tableau sur l'appareil. On peut bien sûr lui passer tout ce qui est accepté par la fonction `numpy.array`, incluant des tableaux NumPy. Le code suivant

```
tab1 = pygpu.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]],
                   dtype='int32', context=ctx)
```

a pour effet d'allouer un tableau de deux dimensions contenant 9 éléments. Les tableaux retournés par cette fonction ne remplacent pas très bien des tableaux NumPy. Si on veut des tableaux supportant tous les opérateurs python habituels pour les opérations mathématiques et les comparaisons, comme le font les tableaux NumPy, on doit utiliser la classe `pygpu.array.gpuarray`. On peut le faire en passant la classe comme argument à la méthode `pygpu.array`

```
tab2 = pygpu.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]],
                   dtype='int32', context=ctx,
                   cls=pygpu.array.gpuarray)
```

Il est également possible d'écrire sa propre classe d'extension pour ajouter ou modifier des opérations aux tableaux, comme `pygpu.array.gpuarray`.

On peut également allouer des tableaux de zéros avec `pygpu.zeros` ou encore des tableaux vides (non-initialisés) avec `pygpu.empty`.

4.3 Manipulations

Après le transfert des données, on peut les manipuler de plusieurs manières. La première méthode est l'indexation, qui consiste à sélectionner un sous-ensemble des données d'un tableau. Chaque dimension peut être indexée indépendamment avec un début, une fin et un pas. La syntaxe est celle qui est habituellement utilisée pour les objets Python. Par exemple l'indexation `tab1[::-1,0]` nous donnerait la première valeur de chaque rangée dans l'ordre inverse, donc pour le tableau `tab2` défini plus haut `[7, 4, 1]`. Il est important de noter que cette indexation ne comporte aucune copie ou transfert de données et que le tableau résultant réfère aux mêmes données que l'original.

Pour copier des données, on peut utiliser l'opération

```
tab_copy = tab1.copy()
```

Par défaut, la copie retournée est contiguë et les dimensions sont arrangées selon l'ordre C, mais il est possible de changer cela avec des paramètres optionnels.

On peut également créer des vues avec l'opération

```
tab1.view()
```

La méthode `view()` accepte une classe comme argument optionel pour spécifier le type retourné. Une vue est une opération relativement peu coûteuse puisqu'elle ne nécessite aucune copie des données sur le dispositif.

Pour convertir les données sous un autre format comme de passer d'entier à nombre en virgule flottante avec

```
tab_float = tab1.astype('float32')
```

Cette opération implique une copie des données.

Si on veut s'assurer d'une certaine disposition des données, on peut appeler une des fonctions `ascontiguousarray()` et `asfortranarray()` pour obtenir respectivement, un tableau contigu selon la norme C et un tableau contigu selon la norme Fortran.

4.4 Calculs

Nous en arrivons à la partie la plus importante, qui est la réalisation de calculs sur cartes graphiques. L'interface principale est la classe `GpuKernel`. Cette interface nécessite l'écriture manuelle de noyau. On crée une instance en lui passant le code source de l'opération que l'on veut réaliser. Pour une opération simple, la plupart du code s'occupe de l'adressage des éléments. Voici un exemple de code pour l'opération `a += b` effectuée élément par élément sur un tableau à deux dimensions avec cette interface.

```
a_p_b = """
KERNEL void calcul(const unsigned int n,
                   const unsigned int dim1,
                   GLOBAL_MEM char *a_data,
                   const unsigned int a_offset,
                   const int a_str_0, const int a_str_1,
                   GLOBAL_MEM char *b_data,
                   const unsigned int b_offset,
                   const int b_str_0, const int b_str_1)
{
```

```
    const unsigned int idx = LDIM_0 * GID_0 + LID_0;
    const unsigned int numThreads = LDIM_0 * GDIM_0;
    unsigned int i;

    a_data += a_offset;
    b_data += b_offset;

    for (i = idx; i < n; i += numThreads) {
        unsigned int pos;
        GLOBAL_MEM char *a_p = a_data;
        GLOBAL_MEM char *b_p = b_data;
        pos = i % dim1;
        a_p += pos * a_str_1;
        b_p += pos * b_str_1;
        pos = i / dim1;
        a_p += pos * a_str_0;
        b_p += pos * b_str _0;
        GLOBAL_MEM float *a = (GLOBAL_MEM float *)a_p;
        GLOBAL_MEM float *b = (GLOBAL_MEM float *)b_p;
        a[0] += b[0];
    }
}
"""
noyau = GpuKernel(a_p_b, 'calcul', context=ctx)
```

Pour appeler le noyau (en supposant que `a` et `b` sont déjà alloués et d'ordre 2) on utilise

```
noyau(numpy.asarray(a.size, dtype='uint32'),
      numpy.asarray(a.shape[1], dtype='uint32'),
      a, numpy.asarray(a.offset, dtype='uint32'),
      numpy.asarray(a.strides[0], dtype='int32'),
      numpy.asarray(a.strides[1], dtype='int32'),
      b, numpy.asarray(b.offset, dtype='uint32'),
```

```
    numpy.asarray(b.strides[0], dtype='int32'),
    numpy.asarray(b.strides[1], dtype='int32'),
    n=a.size)
```

Les arguments de type tableau peuvent être passés tels quels, mais les arguments de type scalaire doivent être accompagnés d'un marqueur de type qui est le rôle joué par la structure NumPy. Ce marqueur de type est nécessaire pour que le passage des arguments se fasse correctement. Si les mauvais types sont indiqués, il est fort probable que le noyau reçoive des valeurs corrompues. Il n'est malheureusement pas encore possible de détecter automatiquement les bons types d'arguments.

On peut voir que dans ce cas assez simple, il y a tout de même beaucoup de code à écrire. Heureusement, une interface simplifiée est disponible pour les opérations élément par élément. Cette interface s'occupe de générer le code d'adressage approprié pour les arguments reçus. La même opération avec cette autre interface est implémentée par le code suivant

```
noyau = ElemwiseKernel(ctx, "float *a, float *b",
                       "a[i] += b[i]")
```

Elle s'occupe également de passer les arguments requis avec les marqueurs de types nécessaires lors de l'appel. L'appel se fait avec ce code

```
noyau(a, b)
```

qui est beaucoup plus simple. Cette interface supporte aussi des valeurs scalaires comme paramètres pour réaliser des opérations du genre $c = ax + b$, mais il faut qu'au moins un des paramètres soit un tableau.

La première interface reste disponible pour réaliser d'autres sortes d'opérations que celles élément par élément.

4.5 Lecture

Une fois que tous les calculs désirés sont effectués, on veut retransférer les données vers la mémoire centrale. Pour ce faire on appelle :

```
a_cpu = numpy.asarray(a)
```

et `a_cpu` devient un tableau NumPy qui contient une copie exacte des éléments de `a` mais en mémoire centrale. Une copie temporaire dans la mémoire du dispositif est effectuée durant l'opération si `a` n'est pas contigu.

CHAPITRE 5

CONCLUSION ET TRAVAUX FUTURS

Entre la première version de ce projet, présentée dans [1] en décembre 2011 et la version actuelle présentée dans ce mémoire, il y a eu d'énormes progrès. La bibliothèque offre maintenant un plein support pour les deux principaux environnements de calculs.

L'implémentation supporte une grande variété de types et permet aux utilisateurs d'y ajouter leurs propres types de données si les besoins ne sont pas couverts. Le support pour un ordre et des tailles de dimension est complet mais n'impose pas une implémentation complexifiée de noyaux puisque des fonctions de conversion de format sont fournies. Ces fonctions permettent aussi la réutilisation de noyaux existants. Pour compléter les objectifs que nous nous étions fixés, l'implémentation comporte aussi un décalage et des pas pour chaque dimension qui régissent l'accès aux données. Encore une fois, les fonctions de conversion de format permettent d'éviter de ce soucier de ces paramètres lors de l'écriture de noyaux.

Pour obtenir cette abstraction certaines possibilités ont du être restreintes. La limitation principale est le support de dispositifs uniques, c'est-à-dire que l'on ne peut travailler qu'avec un seul dispositif de calcul à la fois dans un contexte donné. Ainsi, même si une application peut travailler avec une multitude de GPUs, ceux-ci ne peuvent partager les tableaux qu'en copiant leurs données en mémoire centrale de manière explicite.

En cours de route, certains problèmes ont du être résolus ou contournés pour avancer. Chaque camps a ses problèmes, mais certains d'entre eux en on plus que d'autres.

Pour commencer, avec l'environnement de Nvidia, le seul problème majeur que j'ai eu est le manque de support pour l'indexation par des entiers négatifs sur les plateformes 64 bits. La manière dont les cartes font le calcul mène à un dépassement dans les registres temporaires et ajoute 2^{32} à l'adresse en plus de soustraire la quantité désirée. J'ai contourné le problème en faisant la somme des déplacement dans toutes les dimensions avant de l'additionner au pointeur. Comme cette somme est toujours positive, le problème ne se présente pas.

Sinon du côté d'AMD, même si leurs implémentations respectent le standard OpenCL, je ne peux m'empêcher de penser que c'est une considération de seconde classe. Toutes sortes de limites arbitraires (sur le nombre de queues asynchrones totales, sur les décalages minimaux de

sous-blocs) posées aux capacités de l'interface m'ont forcé à revoir plusieurs choix de design et à compliquer l'implémentation. En plus de tout cela, pour faire fonctionner une carte sous Linux, il faut nécessairement l'utiliser pour l'affichage graphique de la machine et avoir accès à cet affichage[1]. Ce dernier point particulièrement me fait douter de la capacité à utiliser des cartes AMD pour en faire une grappe de calcul.

Du côté des environnements de Apple et Intel je n'ai pas rencontré de problèmes majeurs.

Le travail accompli a permit d'établir des bases solides sur lesquelles certaines fonctionnalités intéressantes ont déjà étés bâties. Il en reste encore une partie à faire pour atteindre un éventail similaire à ce qu'offre numpy. Ce qui est le plus pressant est la réduction, une opération qui applique un calcul commutatif aux éléments d'un tableau afin d'en obtenir une rangée sommaire.

Il reste également à terminer l'intégration à Theano, qui est déjà entamée. Cette intégration sera ma priorité dans les mois à venir puisque qu'elle bénéficiera non seulement à mon travail par un ensemble de tests solides basé sur une utilisation diverse, mais également à Theano en permettant l'utilisation de nouvelles cartes de calculs. Les noyaux employés par theano bénéficieront aussi des optimisations offertes par la bibliothèque.

Une fois que l'intégration sera bien avancée, il restera à convaincre d'autres projets d'adopter notre plateforme pour bâtir leur fonctionnalité.

[1] rouler un serveur X utilisant cette carte et y avoir les droits d'accès

BIBLIOGRAPHIE

[1] Frédéric Bastien, Arnaud Bergeron, Andreas Klöckner, Pascal Vincent et Yoshua Bengio. A common gpu n-dimensional array for python and c. Dans *Big Learn workshop, NIPS'11*, 2011.

[2] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley et Yoshua Bengio. Theano : a CPU and GPU math expression compiler. Dans *Proceedings of the Python for Scientific Computing Conference (SciPy)*, juin 2010. URL `http://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf`. Oral Presentation.

[3] Mads Mayntz Kierulf. Numpy for cuda. Mémoire de maîtrise, Department of Computer Science, University of Copenhagen, juin 2010.

[4] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov et Ahmed Fasih. PyCUDA and PyOpenCL : A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012. ISSN 0167-8191.

[5] V. Mnih. Cudamat : a CUDA-based matrix class for python. Rapport technique UTML TR 2009-004, Department of Computer Science, University of Toronto, novembre 2009.

[6] John Nickolls, Ian Buck, Michael Garland et Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, mars 2008. ISSN 1542-7730. URL `http://doi.acm.org/10.1145/1365490.1365500`.

[7] Travis E. Oliphant. *Guide to NumPy*. Provo, UT, mars 2006. URL `http://www.tramy.us/`.

[8] John E. Stone, David Gohara et Guochun Shi. Opencl : A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, mai 2010. ISSN 0740-7475. URL `http://dx.doi.org/10.1109/MCSE.2010.69`.

[9] T. Tieleman. Gnumpy : an easy way to use GPU boards in Python. Rapport technique UTML TR 2010-002, University of Toronto, Department of Computer Science, 2010.

Annexe I

Référence libcompyte (en anglais)

CHAPTER

TWO

C LIBRARY REFERENCE

struct **compyte_buffer_ops**
#include <compyte/buffer.h>

Function table that a backend must provide.

Public Members

void *(* **buffer_init**)(int dev, int *ret)

Create a context on the specified device.

Warning

This function is not thread-safe.

Return

An opaque pointer to the created context or NULL if an error occured.

Parameters

- `dev` - the device number. The precise meaning of the device number is backend-dependent
- `ret` - error return location. Will be ignored if set to NULL.

void(* **buffer_deinit**)(void *ctx)

Destroy a context.

This removes the external reference to the context and as soon as all buffer and kernels associated with it are free all its resources will be released.

Do not call this function more than once on a given context.

Parameters

- `ctx` - a valid context pointer.

gpudata *(* **buffer_alloc**)(void *ctx, size_t sz, int *ret)

Allocates a buffer of size `sz` in context `ctx`.

Return

A non-NULL pointer to a gpudata structure. This structure is intentionally opaque as its content may change according to the backend used.

Parameters

- `ctx` - a context pointer
- `sz` - the requested size
- `ret` - error return pointer

void(* **buffer_free**)(gpudata *b)

Free a buffer.

Release all ressources associated with `b`.

If this function is called on a buffer that is in use by a kernel the results are undefined. (The current backends either block until kernel completion or maintain a reference to the buffer, but you should not rely on this behavior.)

int(* **buffer_share**)(gpudata *a, gpudata *b, int *ret)

Check if two buffers may overlap.

Both buffers must have been created with the same backend.

Parameters

- `a` - first buffer
- `b` - second buffer
- `ret` - error return pointer

Return Value

- `1` - The buffers may overlap
- `0` - The buffers do not overlap.
- `-1` - An error was encoutered, `ret` contains a detailed error code if not NULL.

int(* **buffer_move**)(gpudata *dst, size_t dstoff, gpudata *src, size_t srcoff, size_t sz)

Copy the content of a buffer to another.

Both buffers must be in the same context. Additionally the buffer must not overlap otherwise the content of the destination buffer is not defined.

Return

GA_NO_ERROR or an error code if an error occurred.

Parameters

- `dst` - destination buffer
- `dstoff` - offset inside the destination buffer
- `src` - source buffer
- `srcoff` - offset inside the source buffer
- `sz` - size of data to copy (in bytes)

int(* **buffer_read**)(void *dst, gpudata *src, size_t srcoff, size_t sz)

Transfer data from a buffer to memory.

Return

GA_NO_ERROR or an error code if an error occurred.

Parameters

- `dst` - destination in memory
- `src` - source buffer
- `srcoff` - offset inside the source buffer
- `sz` - size of data to copy (in bytes)

int(* **buffer_write**)(gpudata *dst, size_t dstoff, const void *src, size_t sz)

Transfer data from memory to a buffer.

Return

GA_NO_ERROR or an error code if an error occurred.

Parameters

- `dst` - destination buffer
- `dstoff` - offset inside the destination buffer
- `src` - source in memory
- `sz` - size of data to copy (in bytes)

int(* **buffer_memset**)(gpudata *dst, size_t dstoff, int data)

Set a buffer to a byte pattern.

This function acts like the C function memset() for device buffers.

Return

GA_NO_ERROR or an error code if an error occurred.

Parameters

- `dst` - destination buffer
- `dstoff` - offset into the destination buffer
- `data` - byte value to write into the destination.

gpukernel *(* **buffer_newkernel**)(void *ctx, unsigned int count, const char **strings, const size_t *lengths, const char *fname, int flags, int *ret)

Compile a kernel.

Compile the kernel composed of the concatenated strings in `strings` and return a callable kernel. If lengths is NULL then all the strings must be NUL-terminated. Otherwise, it doesn't matter (but the lengths must not include the final NUL byte if provided).

Return

Allocated kernel structure or NULL if an error occured. `ret` will be updated with the error code if not NULL.

Parameters

- `ctx` - context to work in
- `count` - number of input strings
- `strings` - table of string pointers
- `lengths` - (optional) length for each string in the table
- `fname` - name of the kernel function (as defined in the code)

- `flags` - flags for compilation (see *ga_useflags*)
- `ret` - error return pointer

void(* **buffer_freekernel**)(gpukernel *k)

Free a kernel.

Releases all resources associated with `k`.

If you free a kernel while it is running, the results are undefined.

int(* **buffer_setkernelarg**)(gpukernel *k, unsigned int index, int typecode, const void *val)

Set a kernel argument.

`val` is a pointer to the actual argument value except for the pseudo-type GA_BUFFER where val is the gpudata pointer itself.

Return

GA_NO_ERROR or an error code if an error occurred.

Parameters

- `k` - kernel
- `index` - argument index (0-based)
- `typecode` - argument type
- `val` - pointer to argument value

int(* **buffer_callkernel**)(gpukernel *k, size_t bs, size_t gs)

Call a kernel.

All arguments must have been specified by a previous call to *buffer_setkernelarg()*. Arguments may be reused between call.

Return

GA_NO_ERROR or an error code if an error occurred.

Parameters

- `k` - kernel
- `bs` - block size for this call (also known as local size)
- `gs` - grid size for this call (also known as global size)

int(* **buffer_sync**)(gpudata *b)

Synchronize a buffer.

Waits for all previous read, writes, copies and kernel calls involving this buffer to be finished.

This call is not required for normal use of the library as all exposed operations will properly synchronize amongst themselves. This call may be useful in a performance timing context to ensure that the work is really done, or before interaction with another library to wait for pending operations.

int(* **buffer_extcopy**)(gpudata *input, size_t ioff, gpudata *output, size_t ooff, int intype, int outtype, unsigned int a_nd, const size_t *a_dims, const ssize_t *a_str, unsigned int b_nd, const size_t *b_dims, const ssize_t *b_str)

Copy non-contiguous buffers.

Specialized kernel to copy memory from a generic array structure to another. May be used to perform casts on the data and/or change data layout.

This function requires that the input and output buffers have the same number of items.

Return

GA_NO_ERROR or an error code if an error occurred.

Parameters

- `input` - input data buffer
- `ioff` - offset into input buffer
- `output` - output data buffer
- `ooff` - offset into output buffer
- `intype` - data type of input
- `outtype` - data type of output
- `a_nd` - order of input (number of dimensions)
- `a_dims` - dimensions of input
- `a_str` - strides of input
- `b_nd` - order of output (number of dimensions)
- `b_dims` - dimensions of output
- `b_str` - strides of output

int(* **buffer_property**)(void *ctx, gpudata *buf, gpukernel *k, int prop_id, void *res)

Returns a property.

Can be used to get a property of a context, a buffer or a kernel. The element corresponding to the property category must be given as argument and the other two are ignored. The currently defined properties and their type are defined in *Properties*.

Return

GA_NO_ERROR or an error code if an error occurred.

Parameters

- `ctx` - context
- `buf` - buffer
- `k` - kernel
- `prop_id` - property id (from *Properties*)
- `res` - pointer to the return space of the appropriate type

const char *(* **buffer_error**)(void *ctx)

Get a string describing the last error that happened.

This function will return a string description of the last backend error to happen on the specified context.

If you need to get a description of a error that occurred during context creation, call this function using NULL as the context. This version of the call is not thread-safe.

Return

string description of the last error

Parameters

- `ctx` - context for which to query the error

struct **compyte_type**
#include <types.h>

Structure that holds the properties of a type.

Public Members

const char * **cluda_name**

Type name to use in the buffers.

size_t **size**

Size of one element (in bytes).

size_t **align**

Alignement requirement for the type.

int **typecode**

Code for the type.

struct **GpuArray**
#include <array.h>

Main array structure.

Public Members

gpudata * **data**

Device data buffer.

compyte_buffer_ops * **ops**

Backend operations vector.

size_t * **dimensions**

Size of each dimension. The number of elements is *nd*.

ssize_t * **strides**

Stride for each dimension. The number of elements is *nd*.

size_t **offset**

Offset to the first array element into the device data buffer.

unsigned int **nd**

Number of dimensions.

int **flags**

Flags for this array (see *Array Flags*).

int **typecode**

Type of the array elements.

struct **GpuKernel**
#include <kernel.h>

Kernel information structure.

Public Members

gpukernel * **k**

Device kernel reference.

compyte_buffer_ops * **ops**

Backend operations vector.

file **array.h**
#include <compyte/buffer.h>

Array functions.

Defines

GA_C_CONTIGUOUS

Array is C-contiguous.

GA_F_CONTIGUOUS

Array is Fortran-contiguous.

GA_OWNDATA

Array owns the *GpuArray::data* element (is responsible for freeing it).

GA_ENSURECOPY

Unused.

GA_ALIGNED

Buffer data is properly aligned for the type (currently this is always assumed to be true).

GA_WRITEABLE

Can write to the data buffer. (This is always true for array allocated through this library).

GA_BEHAVED

Array data is behaved (properly aligned and writable).

GA_CARRAY

Array layout is that of a C array.

GA_FARRAY

Array layout is that of a Fortran array.

GpuArray_OWNSDATA(a)

Checks if the array owns its data (is responsible for freeing it when freed itself).

Return

true if `a` owns its data

Parameters

- `a` - array

GpuArray_ISWRITEABLE(a)

Checks if the array data is writable.

Return

true if the data area of `a` is writable

Parameters

- `a` - array

GpuArray_ISALIGNED(a)

Checks if the array elements are aligned.

Return

true if the elements of `a` are aligned.

Parameters

- `a` - array

GpuArray_ISONESEGMENT(a)

Checks if the array elements are contiguous in memory.

Return

true if the data area of `a` is contiguous

Parameters

- `a` - array

GpuArray_ISFORTRAN(a)

Checks if the array elements are laid out if Fortran order.

Return

true if the data area of `a` is Fortran-contiguous

Parameters

- `a` - array

GpuArray_ITEMSIZE(a)

Retrive the size of the elements in the array.

Return

the size of the array elements.

Parameters

- `a` - array

Enums

ga_order enum

Type used to specify the desired order to some functions

Values:

- `GA_ANY_ORDER = =-1` - Any order is fine.
- `GA_C_ORDER = =0` - C order is desired.
- `GA_F_ORDER = =1` - Fortran order is desired.

Functions

int **GpuArray_CHKFLAGS**(*GpuArray* * a, int flags)

Checks if all the specified flags are set.

Return

true if all flags in `flags` are set and false otherwise.

Parameters

- `a` - array
- `flags` - flags to check

int **GpuArray_empty**(*GpuArray* * a, *compyte_buffer_ops* * ops, void * ctx, int typecode, unsigned int nd, size_t * dims, *ga_order* ord)

Initialize and allocate a new empty (uninitialized data) array.

Return

A return of GA_NO_ERROR means that the structure is properly initialized and that the memory requested is reserved on the device. Any other error code means that the structure is left uninitialized.

Parameters

- `a` - the *GpuArray* structure to initialize. Content will be ignored so make sure to deallocate any previous array first.
- `ops` - backend operations to use.
- `ctx` - context in which to allocate array data. Must come from the same backend as the operations vector.
- `typecode` - type of the elements in the array
- `nd` - desired order (number of dimensions)
- `dims` - size for each dimension.
- `ord` - desired layout of data.

int **GpuArray_zeros**(*GpuArray* * a, *compyte_buffer_ops* * ops, void * ctx, int typecode, unsigned int nd, size_t * dims, *ga_order* ord)

Initialize and allocate a new zero-initialized array.

Return

A return of GA_NO_ERROR means that the structure is properly initialized and that the memory requested is reserved on the device. Any other error code means that the structure is left uninitialized.

Parameters

- `a` - the *GpuArray* structure to initialize. Content will be ignored so make sure to deallocate any previous array first.
- `ops` - backend operations to use.
- `ctx` - context in which to allocate array data. Must come from the same backend as the operations vector.
- `typecode` - type of the elements in the array
- `nd` - desired order (number of dimensions)
- `dims` - size for each dimension.
- `ord` - desired layout of data.

int **GpuArray_fromdata**(*GpuArray* * a, *compyte_buffer_ops* * ops, *gpudata* * data, size_t offset, int typecode, unsigned int nd, size_t * dims, ssize_t * strides, int writeable)

Initialize and allocate a new array structure from a pre-existing buffer.

The array will be considered to own the gpudata structure after the call is made and will free it when deallocated. An error return from this function will deallocate `data`.

Return

A return of GA_NO_ERROR means that the structure is properly initialized. Any other error code means that the structure is left uninitialized and the provided buffer is deallocated.

Parameters

- `a` - the *GpuArray* structure to initialize. Content will be ignored so make sure to deallocate any previous array first.
- `ops` - backend that corresponds to the buffer.
- `data` - buffer to user.
- `offset` - position of the first data element of the array in the buffer.
- `typecode` - type of the elements in the array
- `nd` - order of the data (number of dimensions).
- `dims` - size for each dimension.
- `strides` - stride for each dimension.
- `writeable` - true if the buffer is writable false otherwise.

int **GpuArray_view**(*GpuArray* * v, *GpuArray* * a)

Initialize an array structure to provide a view of another.

The new structure will point to the same data area and have the same values of properties as the source one. The data area is shared and writes from one array will be reflected in the other. The properties are copied and not shared and can be modified independantly.

Return

GA_NO_ERROR if the operation was succesful.

an error code otherwise

Parameters

- `v` - the result array
- `a` - the source array

int **GpuArray_sync**(*GpuArray* * a)

Blocks until all operations (kernels, copies) involving `a` are finished.

Return

GA_NO_ERROR if the operation was succesful.

an error code otherwise

Parameters

- `a` - the array to synchronize

int **GpuArray_index**(*GpuArray* * r, *GpuArray* * a, ssize_t * starts, ssize_t * stops, ssize_t * steps)

Returns a sub-view of a source array.

The indexing follows simple basic model where each dimension is indexed separately. For a single dimension the indexing selects from the start index (included) to the end index (excluded) while selecting one over step elements. As an example for the array `[ 0 1 2 3 4 5 6 7 8 9 ]` indexed with start index 1 stop index 8 and step 2 the result would be `[ 1 3 5 7 ]`.

The special value 0 for step means that only one element corresponding to the start index and the resulting array order will be one smaller.

Return

GA_NO_ERROR if the operation was succesful.

an error code otherwise

Parameters

- `r` - the result array
- `a` - the source array

- `starts` - the start of the subsection for each dimension (length must be a->nd)
- `stops` - the end of the subsection for each dimension (length must be a->nd)
- `steps` - the steps for the subsection for each dimension (length must be a->nd)

int **GpuArray_setarray**(*GpuArray* * a, *GpuArray* * v)

Sets the content of an array to the content of another array.

The value array must be smaller or equal in number of dimensions to the destination array. Each of its dimensions' size must be either exactly equal to the destination array's corresponding dimensions or 1. Dimensions of size 1 will be repeated to fill the full size of the destination array. Extra size 1 dimensions will be added at the end to make the two arrays shape-equivalent.

Return

GA_NO_ERROR if the operation was succesful.

an error code otherwise

Parameters

- `a` - the destination array
- `v` - the value array

int **GpuArray_reshape**(*GpuArray* * res, *GpuArray* * a, unsigned int nd, size_t * newdims, *ga_order* ord, int nocopy)

Change the dimensions of an array.

Return a new array with the desired dimensions. The new dimensions must have the same total size as the old ones. A copy of the underlying data may be performed if necessary, unless `nocopy` is 0.

Return

GA_NO_ERROR if the operation was succesful.

an error code otherwise

Parameters

- `res` - the result array
- `a` - the source array
- `nd` - new dimensions order
- `newdims` - new dimensions (length is nd)
- `ord` - the desired resulting order
- `nocopy` - if 0 error out if a data copy is required.

void **GpuArray_clear**(*GpuArray* * a)

Relase all device and host memory associated with `a`.

This function frees all host memory, and releases the device memory if it is the owner. In case an array has views it is the responsability of the caller to ensure a base array is not cleared before its views.

This function will also zero out the structure to prevent accidental reuse.

Parameters

- `a` - the array to clear

int **GpuArray_share**(*GpuArray* * a, *GpuArray* * b)

Checks if two arrays may share device memory.

Return

1 if `a` and `b` may share a portion of their data.

Parameters

- `a` - an array
- `b` - an array

void * **GpuArray_context**(*GpuArray* * a)

Retursns the context of an array.

Return

the context in which `a` was allocated.

Parameters

- `a` - an array

int **GpuArray_move**(*GpuArray* * dst, *GpuArray* * src)

Copies all the elements of and array to another.

The arrays `src` and `dst` must have the same size (total number of elements) and be in the same context.

Return

GA_NO_ERROR if the operation was succesful.

an error code otherwise

Parameters

- `dst` - destination array
- `src` - source array

int **GpuArray_write**(*GpuArray* * dst, void * src, size_t src_sz)

Copy data from the host memory to the device memory.

Return

GA_NO_ERROR if the operation was succesful.

an error code otherwise

Parameters

- `dst` - destination array (must be contiguous)
- `src` - source host memory (contiguous block)
- `src_sz` - size of data to copy (in bytes)

int **GpuArray_read**(void * dst, size_t dst_sz, *GpuArray* * src)

Copy data from the device memory to the host memory.

Return

GA_NO_ERROR if the operation was succesful.

an error code otherwise

Parameters

- `dst` - dstination host memory (contiguous block)
- `dst_sz` - size of data to copy (in bytes)
- `src` - source array (must be contiguous)

int **GpuArray_memset**(*GpuArray* * a, int data)

Set all of an array's data to a byte pattern.

Return

GA_NO_ERROR if the operation was succesful.

an error code otherwise

Parameters

- `a` - an array (must be contiguous)
- `data` - the byte to repeat

int **GpuArray_copy**(*GpuArray* * res, *GpuArray* * a, *ga_order* order)

Make a copy of an array.

This is analogue to *GpuArray_view()* except it copies the device memory and no data is shared.

Return

GA_NO_ERROR if the operation was succesful.

an error code otherwise

const char * **GpuArray_error**(*GpuArray* * a, int err)

Get a description of the last error in the context of `a`.

The description may reflect operations with other arrays in the same context if other operations were performed between the occurence of the error and the call to this function.

Operations in other contexts, however have no incidence on the return value.

Return

A user-readable string describing the nature of the error.

Parameters

- `a` - an array
- `err` - the error code returned

void **GpuArray_fprintf**(FILE * fd, const *GpuArray* * a)

Print a textual description of `a` to the specified file descriptor.

Parameters

- `fd` - a file descriptior open for writing
- `a` - an array

int **GpuArray_is_c_contiguous**(const *GpuArray* * a)

int **GpuArray_is_f_contiguous**(const *GpuArray* * a)

file **buffer.h**
#include <sys/types.h>
#include <stdio.h>
#include <stdarg.h>
#include <compyte/compat.h>

This file contains the interface definition for the backends.

For normal use you should not call the functions defined in this file directly.

See

array.h For managing buffers

kernel.h For using kernels

Defines

GA_CTX_PROP_DEVNAME

Get the device name for the context.

Type: `char *`

Note

The returned string is allocated and must be freed by the caller.

GA_CTX_PROP_MAXLSIZE

Get the maximum block size (also known as local size) for a kernel call in the context.

Type: `size_t`

GA_CTX_PROP_LMEMSIZE

Get the local memory size available for a call in the context.

Type: `size_t`

GA_CTX_PROP_NUMPROCS

Number of compute units in this context.

compute units times local size is more or less the expected parallelism available on the device, but this is a very rough estimate.

Type: `unsigned int`

GA_BUFFER_PROP_CTX

Get the context in which this buffer was allocated.

Type: `void *`

GA_KERNEL_PROP_CTX

Get the context for which this kernel was compiled.

Type: `void *`

GA_KERNEL_PROP_MAXLSIZE

Get the maximum block size (also known as local size) for a call of this kernel.

Type: `size_t`

GA_KERNEL_PROP_PREFLSIZE

Get the prefered multiple of the block size for a call to this kernel.

Type: `size_t`

GA_KERNEL_PROP_MAXGSIZE

Get the maximum group size for a call of this kernel.

Type: `size_t`

Typedefs

typedef struct _gpudata **gpudata**

Opaque struct for buffer data.

typedef struct _gpukernel **gpukernel**

Opaque struct for kernel data.

Enums

ga_usefl enum

Flags for *compyte_buffer_ops::buffer_newkernel*.

It is important to specify these properly as the compilation machinery will ensure that the proper configuration is made to support the requested features or error out if the demands cannot be met.

Warning

Failure to properly specify the feature flags will in most cases result in silent data corruption (especially on ATI cards).

Values:

- `GA_USE_CLUDA = = 0x01` - The kernel source uses CLUDA unified language.
- `GA_USE_SMALL = = 0x02` - The kernel makes use of small (size is smaller than 4 bytes) types.
- `GA_USE_DOUBLE = = 0x04` - The kernel makes use of double or complex doubles.
- `GA_USE_COMPLEX = = 0x08` - The kernel makes use of complex of complex doubles.
- `GA_USE_HALF = = 0x10` - The kernel makes use of half-floats (also known as float16)
- `GA_USE_PTX = = 0x1000` - The kernel is made of PTX code.

Functions

const char * **Gpu_error**(*compyte_buffer_ops* * o, void * ctx, int err)

Get the error string corresponding to `err`.

Return

A string description of the error.

Parameters

- `o` - operations vector for the backend that produced the error.
- `ctx` - the context in which the error occured or NULL if the error occured outside a context (like during context creation).
- `err` - error code

compyte_buffer_ops * **compyte_get_ops**(const char * name)

Get operations vector for a backend.

The available backends depend on how the library was built.

Return

the operation vector or NULL if the backend name is unrecognized.

Parameters

- `name` - backend name, currently one of `"cuda"` or `"opencl"`

file **error.h**
#include <compyte/compat.h>

Error functions.

Enums

ga_error enum

List of all the possible error codes.

Values:

- `GA_NO_ERROR = = 0` -
- `GA_MEMORY_ERROR` -
- `GA_VALUE_ERROR` -
- `GA_IMPL_ERROR` -
- `GA_INVALID_ERROR` -
- `GA_UNSUPPORTED_ERROR` -
- `GA_SYS_ERROR` -
- `GA_RUN_ERROR` -
- `GA_DEVSUP_ERROR` -

Functions

const char * **compyte_error_str**(int err)

Returns a user-readable description for most error codes.

Some errors only happen in a context and in those cases *Gpu_error()* will provide more details as to the reason for the error.

file **extension.h**
#include <compyte/compat.h>

Extensions access.

Functions

void * **compyte_get_extension**(const char * name)

Obtain a function pointer for an extension.

Return

A function pointer or NULL if the extension was not found.

file **kernel.h**
#include <compyte/buffer.h>
#include <compyte/array.h>

Kernel functions.

Functions

int **GpuKernel_init**(*GpuKernel* * k, *compyte_buffer_ops* * ops, void * ctx, unsigned int count, const char ** strs, size_t * lens, const char * name, int flags)

Initialize a kernel structure.

`lens` holds the size of each source string. If is it NULL or an element has a value of 0 the length will be determined using strlen() or equivalent code.

Return

GA_NO_ERROR if the operation is successful

any other value if an error occured

Parameters

- `k` - a kernel structure
- `ops` - operations vector
- `ctx` - context in which to build the kernel
- `count` - number of source code strings
- `strs` - C array of source code strings
- `lens` - C array with the size of each string or NULL
- `name` - name of the kernel function
- `flags` - kernel use flags (see *ga_usefl*)

void **GpuKernel_clear**(*GpuKernel* * k)

Clear and release data associated with a kernel.

Parameters

- `k` - the kernel to release

void * **GpuKernel_context**(*GpuKernel* * k)

Returns the context in which a kernel was built.

Return

a context pointer

Parameters

- `k` - a kernel

int **GpuKernel_setarg**(*GpuKernel* * k, unsigned int index, int typecode, void * arg)

Set a scalar argument for a kernel.

Return

GA_NO_ERROR if the operation is successful

any other value if an error occured

Parameters

- `k` - a kernel
- `index` - argument index to set
- `typecode` - type of the argument to set
- `arg` - pointer to the scalar value

int **GpuKernel_setbufarg**(*GpuKernel* * k, unsigned int index, *GpuArray* * a)

Set an array argument for a kernel.

Return

GA_NO_ERROR if the operation is successful

any other value if an error occured

Parameters

- `k` - a kernel
- `index` - argument index to set
- `a` - array argument

int **GpuKernel_call**(*GpuKernel* * k, size_t n, size_t ls, size_t gs)

Launch the execution of a kernel.

You either specify the block and grid sizes (`ls` and `gs`) or the total size (`n`). Set a value to `0` to indicate it is unspecified. You can also specify the total size (`n`) and one of the block (`ls`) or grid (`gs`) size.

If you leave one or both of `ls` or `gs`, it will be filled according to a heuristic to get a good performance out of your hardware. However the number of kernel instances that will be run can be slightly higher than the total size you specified in order to avoid performance degradation. Your kernel should be ready to handle this.

Parameters

- `k` - the kernel to launch
- `n` - number of instances to launch
- `ls` - size of launch blocks
- `gs` - size of launch grid

file **types.h**
#include <sys/types.h>
#include <stddef.h>
#include "compyte/compat.h"

Type declarations and access.

Enums

COMPYTE_TYPES enum

List of all built-in types.

Values:

- GA_BUFFER = = −1 -
- GA_BOOL = = 0 -
- GA_BYTE = = 1 -
- GA_UBYTE = = 2 -
- GA_SHORT = = 3 -
- GA_USHORT = = 4 -
- GA_INT = = 5 -
- GA_UINT = = 6 -
- GA_LONG = = 7 -
- GA_ULONG = = 8 -
- GA_LONGLONG = = 9 -
- GA_ULONGLONG = = 10 -
- GA_FLOAT = = 11 -
- GA_DOUBLE = = 12 -
- GA_QUAD = = 13 -
- GA_CFLOAT = = 14 -
- GA_CDOUBLE = = 15 -
- GA_CQUAD = = 16 -
- GA_HALF = = 23 -
- GA_BYTE2 -
- GA_UBYTE2 -
- GA_BYTE3 -
- GA_UBYTE3 -
- GA_BYTE4 -
- GA_UBYTE4 -
- GA_BYTE8 -
- GA_UBYTE8 -

- GA_BYTE16 -
- GA_UBYTE16 -
- GA_SHORT2 -
- GA_USHORT2 -
- GA_SHORT3 -
- GA_USHORT3 -
- GA_SHORT4 -
- GA_USHORT4 -
- GA_SHORT8 -
- GA_USHORT8 -
- GA_SHORT16 -
- GA_USHORT16 -
- GA_INT2 -
- GA_UINT2 -
- GA_INT3 -
- GA_UINT3 -
- GA_INT4 -
- GA_UINT4 -
- GA_INT8 -
- GA_UINT8 -
- GA_INT16 -
- GA_UINT16 -
- GA_LONG2 -
- GA_ULONG2 -
- GA_LONG3 -
- GA_ULONG3 -
- GA_LONG4 -
- GA_ULONG4 -
- GA_LONG8 -
- GA_ULONG8 -
- GA_LONG16 -
- GA_ULONG16 -
- GA_FLOAT2 -
- GA_FLOAT4 -
- GA_FLOAT8 -
- GA_FLOAT16 -

- `GA_DOUBLE2` -
- `GA_DOUBLE4` -
- `GA_DOUBLE8` -
- `GA_DOUBLE16` -
- `GA_HALF2` -
- `GA_HALF4` -
- `GA_HALF8` -
- `GA_HALF16` -

file **util.h**
#include <*compyte/compat.h*>
#include <*compyte/types.h*>

Utility functions.

Functions

int **compyte_register_type**(*compyte_type* * t, int * ret)

Registers a type with the kernel machinery.

On error this function will return -1.

Return

The type code that corresponds to the registered type. This code is only valid for the duration of the application and cannot be reused between invocation.

Parameters

- `t` - is a preallocated and filled *compyte_type* structure. The memory can be allocated from static memory as it will never be freed.
- `ret` - is a pointer where the error code (if any) will be stored. It can be NULL in which case no error code will be returned. If there is no error then the memory pointed to by `ret` will be untouched.

const *compyte_type* * **compyte_get_type**(int typecode)

Get the type structure for a type.

The resulting structure MUST NOT be modified.

Return

A type structure pointer or NULL

Parameters

- `typecode` - the typecode to get structure for

size_t **compyte_get_elsize**(int typecode)

Get the size of one element of a type.

The type MUST exist or the program will crash.

Return

the size

Parameters

- `typecode` - the type to get the element size for

int **compyte_elem_perdim**(char * strs, unsigned int * count, unsigned int nd, const size_t * dims, const ssize_t * str, const char * id)

group **aflags**

Defines

GA_C_CONTIGUOUS

Array is C-contiguous.

GA_F_CONTIGUOUS

Array is Fortran-contiguous.

GA_OWNDATA

Array owns the *GpuArray::data* element (is responsible for freeing it).

GA_ENSURECOPY

Unused.

GA_ALIGNED

Buffer data is properly aligned for the type (currently this is always assumed to be true).

GA_WRITEABLE

Can write to the data buffer. (This is always true for array allocated through this library).

GA_BEHAVED

Array data is behaved (properly aligned and writable).

GA_CARRAY

Array layout is that of a C array.

GA_FARRAY

Array layout is that of a Fortran array.

group **props**

Defines

GA_CTX_PROP_DEVNAME

Get the device name for the context.

Type: `char *`

Note

The returned string is allocated and must be freed by the caller.

GA_CTX_PROP_MAXLSIZE

Get the maximum block size (also known as local size) for a kernel call in the context.

Type: `size_t`

GA_CTX_PROP_LMEMSIZE

Get the local memory size available for a call in the context.

Type: `size_t`

GA_CTX_PROP_NUMPROCS

Number of compute units in this context.

compute units times local size is more or less the expected parallelism available on the device, but this is a very rough estimate.

Type: `unsigned int`

GA_BUFFER_PROP_CTX

Get the context in which this buffer was allocated.

Type: `void *`

GA_KERNEL_PROP_CTX

Get the context for which this kernel was compiled.

Type: `void *`

GA_KERNEL_PROP_MAXLSIZE

Get the maximum block size (also known as local size) for a call of this kernel.

Type: `size_t`

GA_KERNEL_PROP_PREFLSIZE

Get the prefered multiple of the block size for a call to this kernel.

Type: `size_t`

GA_KERNEL_PROP_MAXGSIZE

Get the maximum group size for a call of this kernel.

Type: `size_t`

dir **compyte**

Annexe II

Référence pygpu (en anglais)

CHAPTER

ONE

PYTHON MODULE REFERENCE

class pygpu.gpuarray.**GpuArray**

Device array

To create instances of this class use `zeros()`, `empty()` or `array()`. It cannot be instanciated directly.

You can also subclass this class and make the module create your instances by passing the *cls* agument to any method that return a new GpuArray. This way of creating the class will NOT call your `__init__()` method.

You can also implement your own `__init__()` method, but you must take care to ensure you properly initialized the GpuArray C fields before using it or you will most likely crash the interpreter.

astype(*dtype*, *order='A'*, *copy=True*)

Cast the elements of this array to a new type.

Parameters

- **dtype** (*string or numpy.dtype or int*) – type of the elements of the result
- **order** (*string*) – memory layout of the result
- **copy** (*bool*) – Always return a copy?

This function returns a new array will all elements cast to the supplied *dtype*, but otherwise unchanged.

If *copy* is False and the type and order match *self* is returned.

copy(*order='C'*)

Return a copy if this array.

Parameters **order** (*string*) – memory layout of the copy

dtype

The dtype of the element

flags

Return the flags as a dictionary

gpudata

Return a pointer to the raw gpudata object.

itemsize

The size of the base element.

ndim

The number of dimensions in this object

offset

Return the offset into the gpudata pointer for this array.

reshape(*shape*, *order='C'*)
Returns a new array with the given shape and order.

The new shape must have the same size (total number of elements) as the current one.

shape
shape of this ndarray (tuple)

size
The number of elements in this object.

strides
data pointer strides (in bytes)

sync()
Wait for all pending operations on this array.

This is done automatically when reading or writing from it, but can be useful as a separate operation for timings.

typecode
The compyte typecode for the data type of the array

view(*cls=GpuArray*)
Return a view of this array.

Parameters **cls** – class of the view (must inherit from GpuArray)

The returned array shares device data with this one and both will reflect changes made to the other.

exception pygpu.gpuarray.**GpuArrayException**
Exception used for all errors related to libcompyte.

class pygpu.gpuarray.**GpuContext**
Class that holds all the information pertaining to a context.

```
GpuContext(kind, devno)
```

Parameters

- **kind** (*string*) – module name for the context
- **devno** (*int*) – device number

The currently implemented modules (for the *kind* parameter) are "cuda" and "opencl". Which are available depends on the build options for libcompyte.

If you want an alternative interface check `init()`.

devname
Device name for this context

kind
Module name this context uses

lmemsize
Size of the local (shared) memory, in bytes, for this context

maxlsize
Maximum size of thread block (local size) for this context

numprocs
Number of compute units for this context

ptr
Raw pointer value for the context object

class pygpu.gpuarray.**GpuKernel**

```
GpuKernel(source, name, context=None, cluda=True, have_double=False, have_small=False, have_comp
```

Compile a kernel on the device

Parameters

- **source** (*string*) – complete kernel source code
- **name** (*string*) – function name of the kernel
- **context** (*GpuContext*) – device on which the kernel is compiled
- **cluda** – use cluda layer?
- **have_double** – ensure working doubles?
- **have_small** – ensure types smaller than float will work?
- **have_complex** – ensure complex types will work?
- **have_half** – ensure half-floats will work?

The kernel function is retrieved using the provided *name* which must match what you named your kernel in *source*. You can safely reuse the same name multiple times.

Note: With the cuda backend, unless you use *cluda=True*, you must either pass the mangled name of your kernel or declare the function 'extern "C"', because cuda uses a C++ compiler unconditionally.

The *have_** parameter are there to tell libcompyte that we need the particular type or feature to work for this kernel. If the request can't be satified a `GpuArrayException` will be raised in the constructor.

Warning: If you do not set the *have_* flags properly, you will either get a device-specific error (the good case) or silent completly bogus data (the bad case).

Once you have the kernel object you can simply call it like so:

```
k = GpuKernel(...)
k(param1, param2, n=n)
```

where *n* is the minimum number of threads to run. libcompyte will try to stay close to this number but may run a few more threads to match the hardware preferred multiple and stay efficient. You should watch out for this in your code and make sure to test against the size of your data.

If you want more control over thread allocation you can use the *ls* and *gs* parameters like so:

```
k = GpuKernel(...)
k(param1, param2, ls=ls, gs=gs)
```

If you choose to use this interface, make sure to stay within the limits of *k.maxlsize* and *k.maxgsize* or the call will fail.

call(*n*, *ls*, *gs*)
Call the kernel with the prepered arguments

Parameters

- **n** – number of work elements

- **ls** – local size
- **gs** – global size

Either *n* or *gs* and *ls* must be set (not 0). You can also set *n* and one of *ls* or *gs* and the other will be filled in.

For a friendlier interface try just calling the object (documentation is in the class doc `GpuKernel`)

maxgsize
Maximum global size for this kernel

maxlsize
Maximum local size for this kernel

preflsize
Preferred multiple for local size for this kernel

setarg(*index*, *o*)
Set argument *index* to *o*.

Parameters

- **index** (*int*) – argument index
- **o** (*GpuArray or numpy.ndarray*) – argument value

This overwrites any previous argument set for that index.

The type of scalar arguments is indicated by wrapping them in a numpy.ndarray like this:

```
param1 = numpy.asarray(1.0, dtype='float32')
```

Arguments which are not wrapped will raise an exception.

setargs(*args*)
Sets the arguments of the kernel to prepare for a `call()`

Parameters **args** (*tuple or list*) – kernel arguments

pygpu.gpuarray.**array**(*obj*, *dtype='float64'*, *copy=True*, *order=None*, *ndmin=0*, *context=None*, *cls=None*)
Create a GpuArray from existing data

Parameters

- **obj** (*array-like*) – data to initialize the result
- **dtype** (*string or numpy.dtype or int*) – data type of the result elements
- **copy** (*bool*) – return a copy?
- **order** (*string*) – memory layout of the result
- **ndmin** (*int*) – minimum number of result dimensions
- **context** (*GpuContext*) – allocation context
- **cls** (*class*) – result class (must inherit from GpuArray)

Return type GpuArray

This function creates a new GpuArray from the data provided in *obj* except if *obj* is already a GpuArray and all the parameters match its properties and *copy* is False.

The properties of the resulting array depend on the input data except if overriden by other parameters.

This function is similar to `numpy.array()` except that it returns GpuArrays.

pygpu.gpuarray.**asarray**(*a*, *dtype=None*, *order='A'*, *context=None*)
Returns a GpuArray from the data in *a*

Parameters

- **a** – data
- **dtype** (*string, numpy.dtype or int*) – type of the elements
- **order** (*string or int*) – layout of the data in memory, one of 'A'ny, 'C' or 'F'ortran
- **context** (*GpuContext*) – context in which to do the allocation

Return type GpuArray

If *a* is already a GpuArray and all other parameters match, then the object itself returned. If *a* is an instance of a subclass of GpuArray then a view of the base class will be returned. Otherwise a new object is create and the data is copied into it.

context is optional if *a* is a GpuArray (but must match exactly the context of *a* if specified) and is mandatory otherwise.

pygpu.gpuarray.**ascontiguousarray**(*a*, *dtype=None*, *context=None*)
Returns a contiguous array in device memory (C order).

Parameters

- **a** (*array-like*) – input
- **dtype** (*string, numpy.dtype or int*) – type of the return array
- **context** (*GpuContext*) – context to use for a new array

Return type array

context is optional if *a* is a GpuArray (but must match exactly the context of *a* if specified) and is mandatory otherwise.

pygpu.gpuarray.**asfortranarray**(*a*, *dtype=None*, *context=None*)
Returns a contiguous array in device memory (Fortran order)

Parameters

- **a** (*array-like*) – input
- **dtype** (*string, numpy.dtype or int*) – type of the elements
- **context** (*GpuContext*) – context in which to do the allocation

Return type array

context is optional if *a* is a GpuArray (but must match exactly the context of *a* if specified) and is mandatory otherwise.

pygpu.gpuarray.**cl_wrap_ctx**(*ptr*)
Wrap an existing OpenCL context (the cl_context struct) into a GpuContext class.

pygpu.gpuarray.**cuda_wrap_ctx**(*ptr*)
Wrap an existing CUDA driver context (CUcontext) into a GpuContext class.

pygpu.gpuarray.**dtype_to_ctype**(*dtype*)
Return the C name for a type.

Parameters **dtype** (*numpy.dtype*) – type to get the name for

Return type string

pygpu.gpuarray.**dtype_to_typecode**(*dtype*)
Get the internal typecode for a type.

Parameters **dtype** (*numpy.dtype*) – type to get the code for

Return type int

pygpu.gpuarray.**empty**(*shape*, *dtype='float64'*, *order='C'*, *context=None*, *cls=None*)
Returns an empty (uninitialized) array of the requested shape, type and order.

Parameters

- **shape** (*iterable of ints*) – number of elements in each dimension
- **dtype** (*string, numpy.dtype or int*) – type of the elements
- **order** (*string*) – layout of the data in memory, one of 'A'ny, 'C' or 'F'ortran
- **context** (*GpuContext*) – context in which to do the allocation
- **cls** (*class*) – class of the returned array (must inherit from GpuArray)

Return type array

pygpu.gpuarray.**from_gpudata**(*data*, *offset*, *dtype*, *shape*, *context=None*, *strides=None*, *writable=True*, *base=None*, *cls=None*)
Build a GpuArray from pre-allocated gpudata

Parameters

- **data** (*int*) – pointer to a gpudata structure
- **offset** (*int*) – offset to the data location inside the gpudata
- **dtype** (*numpy.dtype*) – data type of the gpudata elements
- **shape** (*iterable of ints*) – shape to use for the result
- **context** (*GpuContext*) – context of the gpudata
- **strides** (*iterable of ints*) – strides for the results
- **writable** – is the data writable?
- **base** – base object that keeps gpudata alive
- **cls** – view type of the result

Warning: This function is intended for advanced use and will crash the interpreter if used improperly.

Note: This function might be deprecated in a later relase since the only way to create gpudata pointers is through libcompyte functions that aren't exposed at the python level. It can be used with the value of the *gpudata* attribute of an existing GpuArray.

---

pygpu.gpuarray.**init**(*dev*)
Creates a context from a device specifier.

Parameters **dev** (*string*) – device specifier

Return type GpuContext

Device specifiers are composed of the type string and the device id like so:

```
"cuda0"
"opencl0:1"
```

For cuda the device id is the numeric identifier. You can see what devices are available by running nvidia-smi on the machine.

For opencl the device id is the platform number, a colon (:) and the device number. There are no widespread and/or easy way to list available platforms and devices. You can experiement with the values, unavaiable ones will just raise an error, and there are no gaps in the valid numbers.

pygpu.gpuarray.**may_share_memory**(*a*, *b*)
Returns True if *a* and *b* may share memory, False otherwise.

pygpu.gpuarray.**register_dtype**(*dtype*, *cname*)
Make a new type known to the cluda machinery.

This function return the associted internal typecode for the new type.

Parameters

- **dtype** (*numpy.dtype*) – new type
- **cname** (*string*) – C name for the type declarations

Return type int

pygpu.gpuarray.**set_cuda_compiler_fn**(*fn*)
Sets the compiler function for cuda kernels.

Parameters **fn** (*callable*) – compiler function

Return type None

fn must have the following signature:

```
fn(source)
```

It will recieve a python bytes string consiting the of complete kernel source code and must return a python byte string consisting of the compilation results or raise an exception.

Warning: Exceptions raised by the function will not be propagated because the call path goes through libcompyte. They are only used to indicate that there was a problem during the compilation.

This overrides the built-in compiler function with the provided one or resets to the default if *None* is given. The provided function must be rentrant if the library is used in a multi-threaded context.

Note: If the "cuda" module was not compiled in libcompyte then this function will raise a *RuntimeError* unconditionaly.

---

pygpu.gpuarray.**set_default_context**(*ctx*)
Set the default context for the module.

Parameters **ctx** (*GpuContext*) – default context

Return type None

The provided context will be used as a default value for all the other functions in this module which take a context as parameter. Call with *None* to clear the default value.

If you don't call this function the context of all other functions is a mandatory argument.

This can be helpful to reduce clutter when working with only one context. It is strongly discouraged to use this function when working with multiple contexts at once.

pygpu.gpuarray.**zeros**(*shape*, *dtype='float64'*, *order='C'*, *context=None*, *cls=None*)
Returns an array of zero-initialized values of the requested shape, type and order.

Parameters

- **shape** (*iterable of ints*) – number of elements in each dimension
- **dtype** (*string, numpy.dtype or int*) – type of the elements
- **order** (*string*) – layout of the data in memory, one of 'A'ny, 'C' or 'F'ortran
- **context** (*GpuContext*) – context in which to do the allocation
- **cls** (*class*) – class of the returned array (must inherit from GpuArray)

Return type array