Examen du cours I3
Programmation hybride et multicœurs : Introduction à la programmation CUDA

19 février 2016

Aucun document autorisé. Durée 1h30.

1 Questions de cours

Donnez des réponses **précises**, si possible **courtes** et autant que possible **complètes**.

1. Nous avons énuméré dans le cours quelques unes des différences fondamentales en terme d'architecture matérielle qui existent entre les CPU et GPU actuels. Citer précisément au moins deux de ces différences.
2. Qu'est ce qu'un kernel CUDA ? Quel est le mot clé permettant de déclarer un kernel CUDA et quelle est la syntaxe spécifique pour lancer l'exécution d'un kernel CUDA ?
3. Définir la notion de *warp*.
4. Quelles sont les variables intrinsèques du modèle de programmation CUDA ? Combien y en a t'il ? Aidez-vous d'un schéma pour rappeller leur signification *géométrique*.
5. Enumérer les différents types de mémoire que l'on trouve dans le modèle de programmation CUDA, et donner brièvement leurs caractéristiques spécifiques (matérielles, taille, latence, ...).
6. Le modèle de programmation CUDA définit **deux** niveaux de hiérarchie (bloc de *threads* et grille de blocs). Pourquoi a-t-on choisi d'avoir deux niveaux ? A quoi servent-ils ?
7. Définir brièvement la notion d'*accès coalescent à la mémoire*. De quel type de mémoire parle-t-on ? Externe, interne au GPU ?
8. Faire une courte liste des spécificités incorporées dans le language de programmation CUDA par rapport au C (mots clés, syntaxe, ...).
9. Les GPU actuels sont souvent qualifiés d'architecture massivement *multi-thread*. Rappeller pourquoi il *faut* utiliser beaucoup de *threads* pour obtenir de bonnes performances sur les GPU.
10. En annexe vous trouverez un extrait du résultat d'exécution de la commande `deviceQuery`. Quelle est la bande passante maximale entre le GPU et sa mémoire externe ? Citer des techniques vues en cours et TP pour améliorer l'utilisation de la mémoire dans un code GPU dit *memory bound* ?

2 Implantation d'un algorithme de transformée en ondelettes dans CUDA

On souhaite implanter dans le modèle de programmation CUDA un opérateur de transformée en ondelettes de type CDF [1] (notamment utilisée dans le standard de compression d'images JPEG2000).

Un signal $1D$ est représenté par un tableau de réels `data` de taille `size`. On considère deux variantes d'un kernel CUDA qui permet de calculer les coefficients d'ondelettes (codes donnés dans les annexes B et C). Les kernels sont exécutés avec une grille contenant 1 seul bloc de threads, et ce bloc contient `size` threads. Les résultats sont renvoyés dans `data` (en écrasant donc l'entrée).

1. On considère le kernel `cdf53_kernel1`. Faire un schéma où est représenté l'état de la mémoire partagée (tableau `shared`) après chaque exécution de la commande `__syncthreads()`. On utilisera un tableau en entrée de taille `size=16`. Le tableau `data` en mémoire globale est initialisé avec les valeurs suivantes

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data[i]	0	4	8	12	16	12	8	4	0	-4	-8	-12	-16	-12	-8	-4

2. (question bonus) Remarquez-vous quelque chose de particulier sur les valeurs de sortie de l'algorithme (tableau `data` à la fin de `cdf53_kernel1`) ? En quoi cela permet-il de faire de la compression de données ?
3. Parmi les deux variantes (`cdf53_kernel1` et `cdf53_kernel2`), une seule présente des problèmes d'accès coalescent à la mémoire. Laquelle ? S'agit-t-il d'un problème de coalescence en lecture ? En écriture ? On pourra justifier la réponse en s'aidant d'un schéma illustratif.
4. En s'aidant du cours (voir l'exemple de la transposition de matrice), proposer et mettre en œuvre une solution permettant d'éviter le problème d'accès coalescent à la mémoire. Écrire le code correspondant.

Rappel 1 *En C et CUDA, l'opérateur % désigne le reste de la division entière. Par exemple `7%2` vaut 1.*

Rappel 2 *En C et CUDA, l'opération $>> 1$ décale les bits d'une unité vers la droite, et permet donc de diviser un entier par 2.*

3 Implantation GPU de la multiplication de matrices

On souhaite implanter dans le modèle de programmation CUDA la multiplication de matrice : $A(n,m) = B(n,p) * C(p,m)$ qui repose sur le code séquentiel suivant :

```
for (i = 0; i < n; ++i)
  for (j = 0; j < m; ++j) {
    sum = 0.0;
    for (k = 0; k < p; ++k)
      sum += b[i+n*k] * c[k+p*j];
    a[i+n*j] = sum;
  }
```

1. Cohen-Daubechies-Feauveau ; famille d'ondelettes biorthogonales.

$A(n, m)$ est une matrice de taille (n, m) représentée en mémoire par tableau de *float*, associée au pointeur a. Les données sont rangées par colonne ($A[i, j]$ désigne l'élément de la i-ème ligne, j-ème colonne ; c'est à dire $a[i + n * j]$ en C).

Pendant les TP, l'exemple *helloworld* vous a montré comment une boucle *for* pouvait être naturellement dépliée dans le modèle de programmation CUDA (via la grille de *threads*).

1. Écrire le code du kernel *kernel_matmul_naif* qui réalise la multiplication de matrice en dépliant les boucles *for* sur i et j grâce au modèle de programmation CUDA (grille 2D de bloc 2D). On suppose que les données sont accédées directement depuis la mémoire externe, on ne se pré-occupe pas de la mémoire partagée.

```
__global__ void kernel_matmul_naif(float* a, float *b, float *c,
                                   int n, int m, int p)
{
  int i = /* ?? */;
  int j = /* ?? */;

  /* A COMPLETER */

}
```

2. Faire un dessin illustrant, pour un bloc donné, le *mapping* entre les indexes de *thread* et les données utilisées dans A, B et C.
3. Si on décidait de faire des blocs 1D de *thread*, qui *mappent* les colonnes de la matrice $A(= B * C)$, et si on devait mettre une partie des données utilisées par un de ces blocs en mémoire partagée pour améliorer les performances, choisiriez-vous de mettre en mémoire partagée les lignes de B ou les colonnes de C ? Justifier la réponse.

A Caractéristiques du GPU K80

```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11520 MBytes (12079136768 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 145 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with
       device simultaneously) >
```

B Kernel 1

```
__global__ void cdf53_kernel1(float *data, int size) {

  __shared__ float shared[SIZE];

  // thread runtime environment, 1D parametrization
  const int x = threadIdx.x;

  // global thread id
  int index = x;

  // read data from global memory
  shared[x] = data[index];
  __syncthreads();

  // Predict 1
  if(x < size) {
    if(x<size/2-1)
      shared[2*x+1]   -= 0.5 * ( shared[2*x] + shared[2*x+2] );
    if (x == size/2-1)
      shared[size-1] -= shared[size-2];
  }
  __syncthreads();

  // Update 1
  if(x < size) {
    if(x<size/2 && x>0)
      shared[2*x] +=  0.25 * ( shared[2*x-1] + shared[2*x+1] );
    if (x == 0)
      shared[0] +=  0.5 * shared[1];
  }
  __syncthreads();

  // Size and Pack back to global memory
  float a = 2.0;
  if(x < size) {
    if(x < size/2)
      data[index] = shared[2*x]*a;
    else
      data[index] = shared[2*x-size+1]/a;
  }
} // end kernel cdf53_kernel1
```

C Kernel 2

```
__global__ void cdf53_kernel2(float *data, int size) {

  __shared__ float shared[SIZE];

  // thread runtime environment, 1D parametrization
  const int x = threadIdx.x;

  // global thread id
  int index = x;

  // read data from global memory
  shared[x] = data[index];
  __syncthreads();

  // Predict 1
  if (x%2==1 && x<size-1)
    shared[x]   -= 0.5 * ( shared[x-1] + shared[x+1] );
  if (x == size-1)
    shared[size-1] -= shared[size-2];
  __syncthreads();

  // Update 1
  if (x%2==0 && x>0 && x<size)
    shared[x] +=  0.25 * ( shared[x-1] + shared[x+1] );
  if (x == 0)
    shared[0] +=  0.5 * shared[1];
  __syncthreads();

  // Size and Pack back to global memory
  float a = 2.0;
  if(x < size) {
    if( x == 2*(x>>1)) {
      index = x/2;
      data[index] = shared[x]*a;
    } else {
      index = x/2 + size/2;
      data[index] = shared[x]/a;
    }
  }
} // end kernel cdf53_kernel2
```