Programmation sur GPU avec CUDA

Initiation

François Cuvelier

Laboratoire d'Analyse Géométrie et Applications
Institut Galilée
Université Paris XIII.

24 mars 2014

Avant d'oublier !

Ecole Doctorale / Module Calcul Scientifique

la séance du 10 avril est décalée au mardi 29 avril !

Plan

GPU

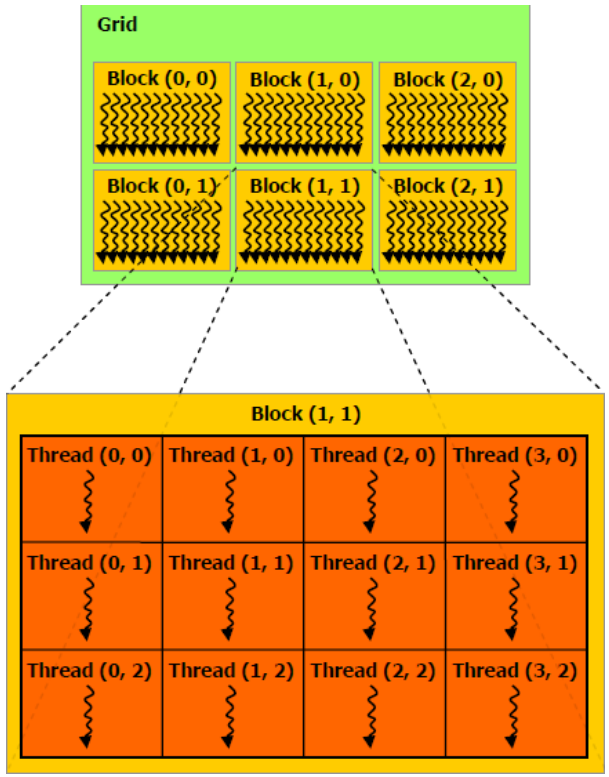
GPU : *Graphics Processing Unit* ou processeur graphique.

- Massivement parallèle (plusieurs centaines de coeurs),
- Opérations sur les vecteurs, matrices et textures à grandes vitesses,
- outils de programmation : AMD APP SDK, Nvidia CUDA, OpenCL, ...

Plan

CUDA

- Le CPU ou **host** demande au GPU ou **device** d'effectuer certains calculs.
- Un **kernel** CUDA est une portion parallèle de code à exécuter sur le GPU par un ensemble de **threads** (processus, tâche, ...)
  - Tous les **threads** exécute le même code.
  - Chaque **thread** peut être identifier de manière unique.
  - Un **kernel** lance une **grid** (ou grille) constituée de **blocks** (ou blocs). Chaque **block** est constitué de **threads**.



Plan

Thèse de N. Belaribi (2012)

- *Sur une classe d'EDP à coefficients irréguliers : Existence, unicité, représentation probabiliste, propagation du chaos*, Thèse dirigée par F. Russo soutenue le 29 novembre 2012.
- *Probabilistic and deterministic algorithms for space multidimensional irregular porous media equation* Belaribi, N., Cuvelier, F. and Russo, F. *Stochastic Partial Differential Equations : Analysis and Computations* March 2013, Volume 1, Issue 1, pp 3-62. DOI 10.1007/s40072-013-0001-7
- *A probabilistic algorithm approximating solutions of a singular PDE of porous media type* Belaribi, N., Cuvelier, F. and Russo, F. *Monte Carlo Methods and Applications*, Volume 17 (2011), Issue 4, Pages 317-369. DOI 10.1515/mcma.2011.014
- Quel rapport avec les GPU ?

Thèse de N. Belaribi (2012)

- Extrait simplifié d'un code Matlab (EDP 2D en espace)

```
x1=-5:1/10:5;x2=x1;
for k=1:nt % Boucle en temps
    X1=X1+... % vecteur aleatoire dimension N>>1
    X2=X2+... % vecteur aleatoire dimension N>>1
    z=ksdensity2D(X1,X2,x1,x2,...)
    ...
end
```

Thèse de N. Belaribi (2012)

- Extrait simplifié d'un code Matlab (EDP 2D en espace)

```
x1=-5:1/10:5;x2=x1;
for k=1:nt % Boucle en temps
    X1=X1+... % vecteur aleatoire dimension N>>1
    X2=X2+... % vecteur aleatoire dimension N>>1
    z=ksdensity2D(X1,X2,x1,x2,...)
    ...
end
```

- **problème majeur :** durée d'éxécution $>$ semaine !

Thèse de N. Belaribi (2012)

- Extrait simplifié d'un code Matlab (EDP 2D en espace)

```
x1=-5:1/10:5;x2=x1;
for k=1:nt % Boucle en temps
    X1=X1+... % vecteur aleatoire dimension N>>1
    X2=X2+... % vecteur aleatoire dimension N>>1
    z=ksdensity2D(X1,X2,x1,x2,...)
    ...
end
```

- **problème majeur :** durée d'éxécution > semaine !
- **raison :** fonction ksdensity2D très couteuses (95% du temps CPU)

Thèse de N. Belaribi (2012)

- Extrait simplifié d'un code Matlab (EDP 2D en espace)

```
x1=-5:1/10:5;x2=x1;
for k=1:nt % Boucle en temps
   X1=X1+... % vecteur aleatoire dimension N>>1
   X2=X2+... % vecteur aleatoire dimension N>>1
   z=ksdensity2D(X1,X2,x1,x2,...)
   ...
end
```

- **problème majeur :** durée d'éxécution > semaine !
- **raison :** fonction ksdensity2D très couteuses (95% du temps CPU)
- **objectif :** utiliser, de manière transparente, un **GPU** pour ce calcul.

Thèse de N. Belaribi (2012)

For a bivariate random sample $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_N$ drawn from a density f, the kernel density estimate is defined by

$$\mathcal{F}_{\boldsymbol{X}}(\boldsymbol{x}; \mathbb{H}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{K}_{\mathbb{H}}(\boldsymbol{x} - \boldsymbol{X}_i)$$

where $\boldsymbol{x} = (x_1, x_2)^t$ and $\boldsymbol{X}_i = (X_{i,1}, X_{i,2})^t$, $\forall i \in [\![1, N]\!]$.
Here $\mathcal{K}_{\mathbb{H}}(\boldsymbol{x}) = |\mathbb{H}|^{-1/2} \mathcal{K}(\mathbb{H}^{-1/2}\boldsymbol{x})$ and $\mathcal{K}$ is the kernel which is a symmetric probability density function, $\mathbb{H}$ is the bandwidth matrix which is symmetric and positive-definite. We take $\mathcal{K}(\boldsymbol{x}) = \frac{1}{2\pi} \exp(-\frac{1}{2} \langle \boldsymbol{x}, \boldsymbol{x} \rangle)$ the standard normal density.

Thèse de N. Belaribi (2012)

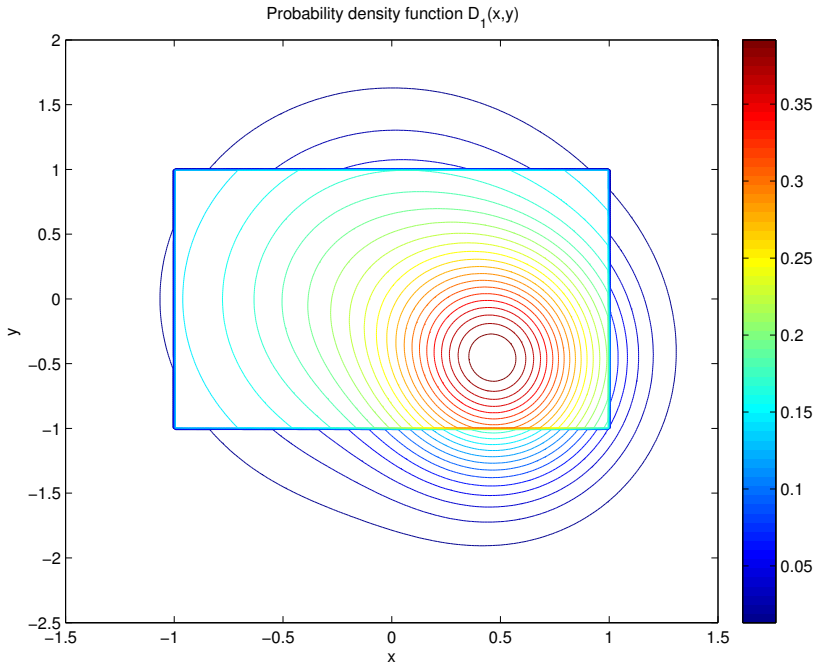
Usual bivariate distributions

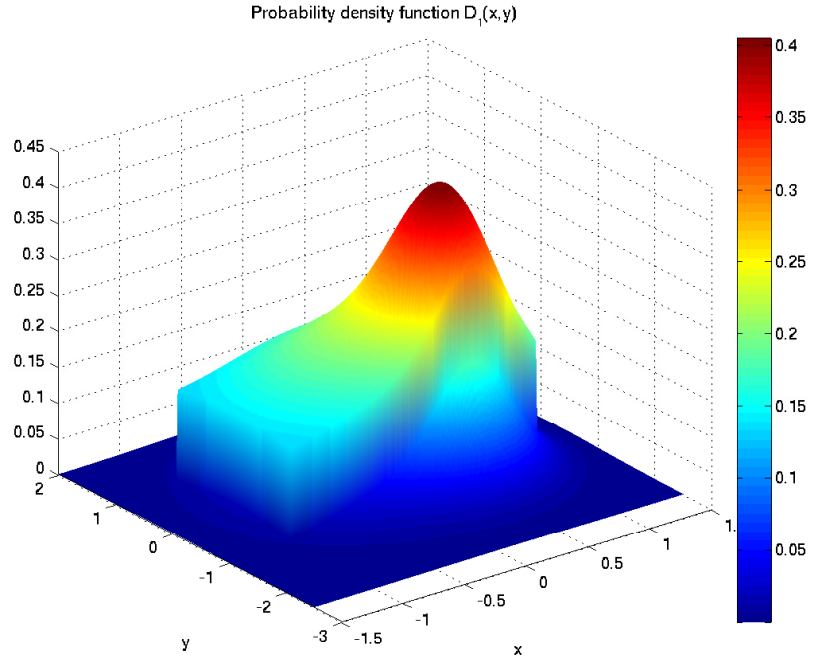
- $\mathcal{U}([a_1, b_1] \times [a_2, b_2])$: bivariate **uniform** distribution over $[a_1, b_1] \times [a_2, b_2]$.
  Density function :
  $$f(\boldsymbol{x}) = 1_{[a_1,b_1]\times[a_2,b_2]}(\boldsymbol{x}).$$
- $\mathcal{N}(\boldsymbol{m}, \mathbb{S})$: bivariate **normal** distribution (mean vector $\boldsymbol{m} \in \mathbb{R}^2$ and covariance matrix $\mathbb{S} \in \mathcal{M}_{2,2}(\mathbb{R})$)
  Density function :
  $$f(\boldsymbol{x}; \boldsymbol{m}, \mathbb{S}) = \frac{1}{2\pi \det(\mathbb{S})^{1/2}} \exp(-\frac{1}{2} \left\langle \mathbb{S}^{-1}(\boldsymbol{x} - \boldsymbol{m}), \boldsymbol{x} - \boldsymbol{m} \right\rangle)$$

Thèse de N. Belaribi (2012)

bivariate distribution sample

$$\begin{aligned}\mathcal{D}_1 &= 0.50\,\mathcal{U}\left([-1.00, 1.00] \times [-1.00, 1.00]\right) \\ &+0.25\,\mathcal{N}\left(\begin{bmatrix}0.00\\0.00\end{bmatrix}; \begin{bmatrix}0.30 & 0.00\\0.00 & 0.70\end{bmatrix}\right) \\ &+0.25\,\mathcal{N}\left(\begin{bmatrix}0.50\\-0.50\end{bmatrix}; \begin{bmatrix}0.10 & 0.00\\0.00 & 0.30\end{bmatrix}\right)\end{aligned}$$





Thèse de N. Belaribi (2012)

Comparison of several versions of `ksdensity2D` function implemented in Matlab :

- `ksdensity2D_OptV0` : code without Toolbox.
- `ksdensity2D_OptV1` : code using `mvnpdf` function from Matlab Statistics Toolbox.
- `ksdensity2D_GpuV0` : code using GPU under Matlab (version 0) → CUDA kernel function + Matlab PCT Toolbox.
- `ksdensity2D_GpuV1` : code using GPU under Matlab (version 1) → CUDA kernel function with **shared** memory + Matlab PCT Toolbox.

Thèse de N. Belaribi (2012)

N	OptV1	OptV0	GpuV0 (sm_20)	GpuV1 (sm_20)
10000	44.919 (s) x 1.00	23.691 (s) x 1.90	0.743 (s) x 60.42	0.053 (s) x 841.88
20000	94.996 (s) x 1.00	37.137 (s) x 2.56	1.472 (s) x 64.52	0.082 (s) x 1154.91
30000	139.311 (s) x 1.00	55.307 (s) x 2.52	2.201 (s) x 63.30	0.111 (s) x 1255.52
40000	190.900 (s) x 1.00	65.643 (s) x 2.91	2.929 (s) x 65.17	0.140 (s) x 1368.38
50000	235.426 (s) x 1.00	68.924 (s) x 3.42	3.658 (s) x 64.37	0.169 (s) x 1394.52
60000	246.112 (s) x 1.00	68.362 (s) x 3.60	4.386 (s) x 56.11	0.197 (s) x 1247.24
70000	296.531 (s) x 1.00	101.305 (s) x 2.93	5.116 (s) x 57.96	0.226 (s) x 1314.19
80000	368.510 (s) x 1.00	120.110 (s) x 3.07	5.843 (s) x 63.07	0.255 (s) x 1447.94
90000	425.157 (s) x 1.00	115.732 (s) x 3.67	6.575 (s) x 64.67	0.283 (s) x 1502.20
100000	460.035 (s) x 1.00	128.837 (s) x 3.57	7.410 (s) x 62.08	0.314 (s) x 1465.49

Table: ksdensity2D - Probability Density : $\mathcal{D}_1$ - time and speedup

Plan

GPU sur magi

- Sur MAGI, 4 GPU (devices) de type "Nvidia Tesla T10 Processor" de 2008.

GPU	Mém.	coeurs	bande passante	Pic perf.
T10 (2008)	4Go	240	102 GB/s	0.19 Tflops
K40 (2013)	12Go	2880	288 GB/s	1.43 Tflops

- CUDA version 5.5 (répertoire /usr/local/cuda) et exemples dans `/usr/local/cuda/samples/`.
- Compilateur **nvcc**, répertoire `/usr/local/cuda/bin` et librairies dans répertoire `/usr/local/cuda/lib64`.
  $\Rightarrow$ `LD_LIBRARY_PATH` et `PATH` à configurer si besoin.

Objectifs

- Apprendre à compiler et exécuter les exemples fournis avec CUDA
  $\Rightarrow$ Exercices 0 et 1.
- Utiliser le GPU sur une partie d'un code existant
  $\Rightarrow$ Exercice 2
- Utilisation des librairies **Thrust** et **curand**
  $\Rightarrow$ Exercices 3 et 4

Plan

Exo00 : DeviceQuery

- Récupération des codes :

```
$ mkdir exo00
$ cd exo00
$ cp -R /usr/local/cuda/samples/1_Utilities/deviceQuery/* .
$ make clean
```

- Modification du `Makefile` :
  - ligne 103, remplacer par :
    `INCLUDES := -I$(CUDA_PATH)/samples/common/inc`
  - lignes 128 et 129 à commenter.
- Compilation

```
$ make
```

Exo00 : DeviceQuery

- Execution du programme sur magi :

```
$ srun -p GPU --gres=gpu:4 deviceQuery
```

En sortie, on obtient

```
deviceQuery Starting...

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

Detected 4 CUDA Capable device(s)

Device 0: "Tesla T10 Processor"
  CUDA Driver Version / Runtime Version          5.5 / 5.5
  CUDA Capability Major/Minor version number:    1.3
  Total amount of global memory:                 4096 MBytes (4294770688 bytes)
  (30) Multiprocessors, (  8) CUDA Cores/MP:     240 CUDA Cores
  GPU Clock rate:                                1296 MHz (1.30 GHz)
  Memory Clock rate:                             800 Mhz
  Memory Bus Width:                              512-bit
  Maximum Texture Dimension Size (x,y,z)         1D=(8192), 2D=(65536, 32768), 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(8192), 512 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(8192, 8192), 512 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
...
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.5, CUDA Runtime Version = 5.5, NumDevs = 4, D
Result = PASS
```

Exo00 : DeviceQuery

- Compilation avec le compilateur Nvidia **nvcc** :

```
$ nvcc -o deviceQuery deviceQuery.cpp
```

- Compilation avec le compilateur **gcc** :

```
$ g++ -c deviceQuery.cpp -I /usr/local/cuda-5.5/include -I /usr/local/cuda-5.5/samples/common/inc
$ g++ -o deviceQuery deviceQuery.o -L/usr/local/cuda-5.5/lib64 -lcudart
```

Plan

Exo00 : DeviceQuery

- Récupération des codes :

```
$ mkdir exo01
$ cd exo01
$ cp -R /usr/local/cuda/samples/0_Simple/vectorAdd/* .
$ make clean
```

- Modification du `Makefile` :
  - ligne 103, remplacer par :
    `INCLUDES := -I$(CUDA_PATH)/samples/common/inc`
  - lignes 128 et 129 à commenter.
- Compilation

```
$ make
```

Exo01 : vectorAdd

- Execution du programme sur magi :

```
$ srun -p GPU --gres=gpu:1 vectorAdd
```

En sortie, on obtient

```
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

Exo01 : vectorAdd

- Compilation avec le compilateur Nvidia **nvcc** :

```
$ Exo01 : vectorAdd
```

▷ Que fait le programme ?

Exo01 : vectorAdd

- Compilation avec le compilateur Nvidia **nvcc** :

```
$ Exo01 : vectorAdd
```

▷ Que fait le programme ?

- Allocation mémoire de `h_A`, `h_B` et `h_C` sur le *host* (CPU).

Exo01 : vectorAdd

- Compilation avec le compilateur Nvidia **nvcc** :

```
$ Exo01 : vectorAdd
```

▷ Que fait le programme ?

- Allocation mémoire de `h_A`, `h_B` et `h_C` sur le *host* (CPU).
- Initialisation de `h_A` et `h_B`.

Exo01 : vectorAdd

- Compilation avec le compilateur Nvidia **nvcc** :

```
$ Exo01 : vectorAdd
```

▷ Que fait le programme ?

- Allocation mémoire de `h_A`, `h_B` et `h_C` sur le *host* (CPU).
- Initialisation de `h_A` et `h_B`.
- Allocation mémoire de `d_A`, `d_B` et `d_C` sur le *device* (GPU).

Exo01 : vectorAdd

- Compilation avec le compilateur Nvidia **nvcc** :

```
$ Exo01 : vectorAdd
```

▷ Que fait le programme ?

- Allocation mémoire de `h_A`, `h_B` et `h_C` sur le *host* (CPU).
- Initialisation de `h_A` et `h_B`.
- Allocation mémoire de `d_A`, `d_B` et `d_C` sur le *device* (GPU).
- Copie de `h_A` dans `d_A` et de `h_B` dans `d_B`.

Exo01 : vectorAdd

- Compilation avec le compilateur Nvidia **nvcc** :

```
$ Exo01 : vectorAdd
```

▷ Que fait le programme ?

- Allocation mémoire de `h_A`, `h_B` et `h_C` sur le *host* (CPU).
- Initialisation de `h_A` et `h_B`.
- Allocation mémoire de `d_A`, `d_B` et `d_C` sur le *device* (GPU).
- Copie de `h_A` dans `d_A` et de `h_B` dans `d_B`.
- Utilisation du kernel `vectorAdd` pour calculer `d_C` ("d_A" + "d_B").

Exo01 : vectorAdd

- Compilation avec le compilateur Nvidia **nvcc** :

```
$ Exo01 : vectorAdd
```

▷ Que fait le programme ?

- Allocation mémoire de `h_A`, `h_B` et `h_C` sur le *host* (CPU).
- Initialisation de `h_A` et `h_B`.
- Allocation mémoire de `d_A`, `d_B` et `d_C` sur le *device* (GPU).
- Copie de `h_A` dans `d_A` et de `h_B` dans `d_B`.
- Utilisation du kernel `vectorAdd` pour calculer `d_C` ("d_A" + "d_B").
- Copie de `d_C` dans `h_C`.

Exo01 : vectorAdd

- Compilation avec le compilateur Nvidia **nvcc** :

```
$ Exo01 : vectorAdd
```

▷ Que fait le programme ?

- Allocation mémoire de `h_A`, `h_B` et `h_C` sur le *host* (CPU).
- Initialisation de `h_A` et `h_B`.
- Allocation mémoire de `d_A`, `d_B` et `d_C` sur le *device* (GPU).
- Copie de `h_A` dans `d_A` et de `h_B` dans `d_B`.
- Utilisation du kernel `vectorAdd` pour calculer `d_C` ("d_A" + "d_B").
- Copie de `d_C` dans `h_C`.
- Liberation de la mémoire alluée sur GPU puis sur CPU.

Exo01 : vectorAdd

Sources du kernel `vectorAdd`

```
1  __global__ void
2  vectorAdd(const float *A, const float *B, float *C,
3            int numElements)
4  {
5      int i = blockDim.x * blockIdx.x + threadIdx.x;
6
7      if (i < numElements)
8      {
9          C[i] = A[i] + B[i];
10     }
11 }
```

Appel du kernel `vectorAdd`

```
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
```

- `__global__` : **kernel** exectuable sur le **device** depuis le **host**.
- `gridDim` : nombre de **blocks** dans la **grid**.
- `blockDim` : nombre de **threads** dans un **block**.
- `blockIdx.x` : numéro du **block** dans la **grid**.
- `threadIdx.x` : numéro du **thread** dans le **block**.

Plan

Exo02 : axpbyVector

On dispose d'un code en Langage C effectuant les opération suivantes :

1. Allocation et initialisation de deux *vecteurs* aléatoires $\boldsymbol{x}$ et $\boldsymbol{y}$ (loi uniforme $[0, 1]$) de dimension 2×10^7.
2. Allocation du *vecteur* $\boldsymbol{z}$ de dimension 2×10^7.
3. $\boldsymbol{z} \leftarrow 2 * \boldsymbol{x} + 3 * \boldsymbol{y}$ via l'appel `axpbyVector(N,2.,x,3.,y,z);`
4. Calcul des moyennes de chacun des vecteurs.

Exo02 : axpbyVector

On dispose d'un code en Langage C effectuant les opération suivantes :

1. Allocation et initialisation de deux *vecteurs* aléatoires $\boldsymbol{x}$ et $\boldsymbol{y}$ (loi uniforme $[0, 1]$) de dimension 2×10^7.
2. Allocation du *vecteur* $\boldsymbol{z}$ de dimension 2×10^7.
3. $\boldsymbol{z} \leftarrow 2 * \boldsymbol{x} + 3 * \boldsymbol{y}$ via l'appel `axpbyVector(N,2.,x,3.,y,z);`
4. Calcul des moyennes de chacun des vecteurs.

Objectif : Remplacer la fonction `axpbyVector` par son équivalent sur GPU !

Exo02 : axpbyVector

- Récupération et compilation des codes :

```
$ wget http://www.math.univ-paris13.fr/~cuvelier/docs/↩
Informatique/CUDA/TPs_5.5/exo02/exo02.tar.gz
$ tar zxvf exo02.tar.gz
$ cd exo02
$ make
```

- Execution du code CPU :

```
srun -n 1 ./axpbyVector
```

```
Time to generate datas (N=20000000) : 509.0570(ms)
Time for axpbyVector function (N=20000000) : 155.0580(ms)
mean(x)=0.499947, mean(y)=0.500003, mean(z)=2.4999
Time for means computation : 215.1300(ms)
Memory used by each vector : 160 Mo
```

Exo02 : axpbyVector

Objectif : Remplacer la fonction axpbyVector par son équivalent sur GPU !

1. Allocation et initialisation de deux *vecteurs* aléatoires $\boldsymbol{x}$ et $\boldsymbol{y}$ (loi uniforme $[0, 1]$) de dimension $N = 2 \times 10^7$.
2. Allocation du *vecteur* $\boldsymbol{z}$ de dimension N.
3. $\boldsymbol{z} \leftarrow 2 * \boldsymbol{x} + 3 * \boldsymbol{y}$ via l'appel `axpbyVector(N,2.,x,3.,y,z);`
4. Calcul des moyennes de chacun des vecteurs.

Exo02 : axpbyVector

Objectif : Remplacer la fonction axpbyVector par son équivalent sur GPU !

1. Allocation et initialisation de deux *vecteurs* aléatoires ***x*** et ***y*** (loi uniforme $[0, 1]$) de dimension $N = 2 \times 10^7$.
2. Allocation du *vecteur* ***z*** de dimension N.
3.
   1. Allocation des vecteurs d_x, d_y et d_z sur **GPU** (dimension N)
   2. Transfert des vecteurs **x** et **y** dans respectivement d_x et d_y
   3. d_z $\leftarrow$ 2* d_x+3* d_y via l'appel du kernel : axpbyGPUVector<<<128, 128>>>(N,2.,d_x,3.,d_y,d_z);
   4. Transfert du vecteur d_z dans ***z***
4. Calcul des moyennes de chacun des vecteurs.

Exo02 : axpbyVector

- Execution du code *pure* CPU :

```
srun -n 1 ./axpbyVector
```

```
Time to generate datas (N=20000000) : 509.0570(ms)
Time for axpbyVector function (N=20000000) : 155.0580(ms)
mean(x)=0.499947, mean(y)=0.500003, mean(z)=2.4999
Time for means computation : 215.1300(ms)
Memory used by each vector : 160 Mo
```

- Execution du code CPU/GPU :

```
srun -p GPU --gres=gpu:1 axpbyVectorGPU
```

```
Time to generate datas (N=20000000) : 687.9780(ms)
Time for transfert CPU->GPU : 158.8030(ms)
Time for axpbyGPUVector function (N=20000000) : 5.9350(ms)
Time for transfert GPU->CPU : 162.6770(ms)
mean(x)=0.50005, mean(y)=0.500049, mean(z)=2.50025
Memory used by each vector : 160 Mo
```

Exo02 : axpbyVector

- Execution du code *pure* CPU :

```
srun -n 1 ./axpbyVector
```

```
Time to generate datas (N=20000000) : 509.0570(ms)
Time for axpbyVector function (N=20000000) : 155.0580(ms)
mean(x)=0.499947, mean(y)=0.500003, mean(z)=2.4999
Time for means computation : 215.1300(ms)
Memory used by each vector : 160 Mo
```

- Execution du code CPU/GPU :

```
srun -p GPU --gres=gpu:1 axpbyVectorGPU
```

```
Time to generate datas (N=20000000) : 687.9780(ms)
Time for transfert CPU->GPU : 158.8030(ms)
Time for axpbyGPUVector function (N=20000000) : 5.9350(ms)
Time for transfert GPU->CPU : 162.6770(ms)
mean(x)=0.50005, mean(y)=0.500049, mean(z)=2.50025
Memory used by each vector : 160 Mo
```

- Conclusion ?

Plan

Exo03 : Thrust

Thrust : a C++ template library for CUDA based on the Standard Template Library STL.

- version 1.7 fourni en standard avec CUDA,
- exemples fournis sur le site `http://thrust.github.io`
- informations supplémentaires `https://github.com/thrust/thrust/wiki/Quick-Start-Guide`

Exo03 : Thrust

- Téléchargement des exemples :

```
$ wget http://www.math.univ-paris13.fr/~cuvelier/docs/↩
Informatique/CUDA/TPs_5.5/Thrust-1.7-examples.tar.gz
$ tar zxvf Thrust-1.7-examples.tar.gz
$ cd Thrust-1.7-examples
```

- Compilation de *basic_vector.cu*

```
nvcc -o basic_vector basic_vector.cu
```

- Execution de *basic_vector*

```
srun -p GPU --gres=gpu:1 basic_vector
```

Plan

Exo04 : axpbyVectorThrust

Objectif : Intégralité des calculs de l'exemple `axpbyVector` sur GPU avec utilisation des les librairies **Thrust** et **curand** fournies avec CUDA.

- Récupération et compilation des codes :

```
$ wget http://www.math.univ-paris13.fr/~cuvelier/docs/↩
Informatique/CUDA/TPs_5.5/exo04/exo04.tar.gz
$ tar zxvf exo04.tar.gz
$ cd exo04
$ make
```

- Remarque : fichier `axpbyVectorThrust.cu` (GPU) 98 lignes et fichier `axpbyVector.c` (CPU) 85 lignes !
- Performances ?

Exo04 : axpbyVectorThrust

- Execution du code *pure* CPU :

```
srun -n 1 ./axpbyVector
```

```
Time to generate datas (N=20000000) : 509.0570(ms)
Time for axpbyVector function (N=20000000) : 155.0580(ms)
mean(x)=0.499947, mean(y)=0.500003, mean(z)=2.4999
Time for means computation : 215.1300(ms)
Memory used by each vector : 160 Mo
```

- Execution du code GPU :

```
srun -p GPU --gres=gpu:1 axpbyVectorThrust
```

```
Time to generate datas (N=20000000) : 68.46(ms)
Time for saxpby_fast function (N=20000000) : 5.796(ms)
mean(x)=0.500044, mean(y)=0.499966, mean(z)=2.49999
Time for means computation : 7.897(ms)
Memory used by each vector : 160 Mo
```