# TP Introduction à la programmation des GPU

P. Kestener

19 mai 2022

**Les TP et le mini-projet devront être faits sur le calculateur local `odette` (Nvidia Ampere A100).**

## 1 Configuration de l'environement

### 1.1 Connection au serveur GPU

```
ssh -Y odette
```

### 1.2 CUDA / Nvidia HPC toolkit

Nous allons utiliser deux compilateurs différents pour les GPU Nvidia :

— `nvcc` pour le codes CUDA

— `nvc++` pour les codes OpenACC

Mettre les lignes suivantes dans le fichier `.bashrc` de votre HOME :

```
source /usr/share/modules/init/bash
module use /opt/modulefiles
module use /opt/nvidia/hpc_sdk_22_3/modulefiles/
```

— pour utiliser le compilateur `nvcc`, tapper `module load cuda/11.6`

— pour utiliser le compilateur `nvc++`, tapper `module load nvhpc/22.3`

### 1.3 python (facultatif)

Si le temps le permet, on pourra faire une introduction à l'usage des GPU à travers python
Je conseille d'installer chacun son propre environement python avec Miniconda3 :

```
mkdir ~/install/python
cd ~/install/python
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
# answer yes when asked if you want to activate miniconda3
```

Redémarrer le shell, et créer un environement spécifique pour le cours :

```
conda create -n cours_gpu
conda activate cours_gpu
# (optionnel) ajouter 'conda activate cours_gpu' à la fin de votre .bashrc
```

Installation de paquets supplémentaires :

```
conda install -c conda-forge numpy matplotlib astropy jupyter jupyterlab
conda install -c conda-forge numba cupy cython
```

### 1.4 configuration pour exécuter un jupyter notebook à distance

But : ouvrir un notebook depuis un navigateur sur la machine locale, mais en l'exécutant sur une machine distante (ici `odette`

Suivre les étapes suivantes :

0. Se connecter par ssh sur `odette`
1. **(À faire une et une seule fois)** Configurer un mot de passe pour `jupyter` sur `odette`

   ```
   jupyter notebook --generate-config
   jupyter notebook password
   ```

2. Lancer le serveur :

   ```
   jupyter notebook --no-browser --port=8889
   ```

   **IMPORTANT :** chacun doit choisir un port différent : 8889, 8890, 8891, .... sinon vous allez éditer le notebook des autres.
   alternativement, vous pouvez lancer un jupyterlab :

   ```
   jupyter lab --no-browser --port=8889
   ```

3. Dans un autre terminal (sur la machine locale), ouvrir un tunnel ssh de la machine locale vers `odette`

   ```
   ssh -f -L localhost:8889:localhost:8889 username@odette sleep 1800
   ```

   le tunnel sera valide 30 minutes et se détruira automatiquement.
4. Ouvrir votre firefox local sur
   - l'url localhost:8889 (pour jupyter notebook)
   - l'url localhost:8889/lab (pour jupyter lab)

## 2 Prise en main des outils de développement CUDA

### 2.1 Environnement Unix

L'objectif du TP est de se familiariser avec le flot de compilation CUDA/C++ avec le compilateur `nvcc`.

On pourra consulter les pages de manuel en ligne et/ou utiliser l'aide en ligne de CUDA : http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

### 2.2 Know your CPU hardware

Exécuter les commandes suivantes pour explorer la machine `odette` (utiliser `ssh -X odette` pour avoir l'interface graphique) :

- `lstopo`
  - Combien de processeurs ? Combien de cœurs ? Combien d'hyperthreads ?
  - Quelle est la taille de la mémoire DRAM de l'hôte ?
  - Quelle est la bande passante mémoire crête par socket (sachant qu'il y a 8 canaux à 3200 MT/s) en Gbytes/s ?

### 2.3 Compilation du SDK CUDA/C++

Copiez les exemples du SDK CUDA/C++ (SDK = Software Development Kit), i.e. dans votre `HOME`, tapez la commande suivante [1] :

1. Ce script bash est fournit par Nvidia, et disponible après avoir installé les outils CUDA

```
module load cuda/11.6
cuda-install-samples-11.6.sh $HOME
```

Le SDK contient des exemples d'applications et de programmes en CUDA/C++. Chaque exemple peut être compilé indépendamment des autres, il suffit de se placer dans le sous-répertoire correspondant et de tapper `make`.

## 2.4 deviceQuery en CUDA/C++ - Know your GPU hardware

Tappez `cd /NVIDIA_CUDA-11.6_Samples/1_Utilities/deviceQuery` pour aller dans le répertoire source de cet exemple et ensuite `make`. Exécutez l'exemple `deviceQuery` qui permet d'avoir toutes les informations sur le matériel disponible [2].

On pourra constater ici que l'on utilise juste l'API CUDA pour interroger le driver de la carte graphique, mais qu'aucun *kernel* CUDA n'est compilé.

1. Exécuter `deviceQuery`.
2. De combien de GPU dispose-t-on sur la machine ?
3. De combien de *streaming multiprocessor* sont faits les GPU ?
4. Utiliser l'information sur le bus mémoire pour en déduire la valeur de la bande passante mémoire crête en GBytes/s.

Pour information, une liste à jour des GPU NVIDIA et leurs caractéristiques :
http://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

## 2.5 HelloWorld en CUDA/C++

### 2.5.1 Utilisation des variables intrinsèques `threadIdx` et `blockIdx`

**Activité 1 :**

— Récupérer le fichier `helloworld/1/helloworld.cu`.

— Compiler `nvcc helloworld.cu -o helloworld`

— Quelle est la sortie de ce programme ?

— Que se passe-t-il si on commente la ligne contenant l'appel à `cudaDeviceSynchronize` ? Comment interprète-t-on ce résultat ?

**Activité 2 :** Récupérez le fichier `helloworld/1/helloworld_block.cu`.
Manipulation :

1. Ouvrir le fichier. Que fait ce programme ?
2. Utiliser les informations de l'entête du fichier pour le compiler et l'exécuter.
3. Utiliser $(gridSize = 1, blockSize = 16)$. Exécuter plusieurs fois. Que constatez-vous ?
4. Utiliser $(gridSize = 16, blockSize = 1)$. Exécuter plusieurs fois. Que constatez-vous ?
5. Dans quel ordre les messages sont affichés ? Comment peut-on l'expliquer ?
6. Modifier le code de façon à traiter des **blocks 2D de taille 4 par 4**.

**Activité 3 :** Récupérez le fichier `helloworld/1/helloworld_arg.cu`.

— Compiler et exécuter. Ce code illustre le fait que l'on peut passer des variables par valeur (copie) à un noyau Cuda. Si l'on a besoin d'accéder à des tableaux, il faut avoir recours à l'allocation dynamique de mémoire (`cudaMalloc/cudaMemcpy/cudaFree`).

2. Complément d'information :
http://devblogs.nvidia.com/parallelforall/how-query-device-properties-and-handle-errors-cuda-cc/

### 2.5.2 Addition de deux vecteurs sur GPU

Le but de l'exercise est d'écrire un premier kernel CUDA et d'apprendre à

— utiliser les variables intrinsèques du modèle de programmation CUDA, i.e. `threadIdx` and `blockIdx` qui dimensionnent la grille de bloc de threads.

— utiliser l'API CUDA : `cudaMalloc`, `cudaMemcpy`, `cudaFree`

1. Éditer le code source `helloworld/2/helloworld_array.cu` et remplir les *trous* aux endroits marqués par `TODO`.
   On suppose dans un premier temps qu'un thread exécute l'addition d'un seul élement des tableaux.
2. Exécuter le programme avec les paramètres par défaut et augmenter la taille des tableaux.
3. Pour les petites valeurs de $N$ (taille des tableaux), afficher à l'écran le résultat du calcul. Vérifier que le code écrit peut être exécuté de multiple configurations, par exemple en 2 blocs de *threads* avec N/2 *threads* par bloc (en donnant évidemment les mêmes résultats).

### 2.5.3 Gestion de la mémoire unifiée (facultatif)

Dans les exemples précédents, nous avons géré explicitement les allocations et les transferts mémoires avec respectivement les appels à cudaMalloc et cudaMemcpy.

Avec l'introduction de la mémoire dite unifiée, on peut déléguer les transferts mémoires entre la carte mère et la carte graphique, en utilisant l'appel à cudaMallocManaged pour l'allocation mémoire. Avec cet appel on récupère un pointeur qui peut être utilisé à la fois sur le CPU et sur le GPU.

**Exercice :**

— Lire la page suivante https://devblogs.nvidia.com/unified-memory-cuda-beginners/

— Mettre en œuvre la mémoire unifiée sur le code précédent (addition de deux tableaux).

— Comparer votre code avec la solution proposée dans `helloworld/2/solution/helloworld_array_managed.cu`

### 2.5.4 Gestion des erreurs et profilage de code

1. — Le code `helloworld/3/helloworld.cu` contient une(des) erreurs. Pouvez-vous les corrigez en vous aidant des messages affichés à l'exécution ?
   — Quel était le problème ?
2. — On utilise à présent le code `helloworld/3/helloworld2.cu`. Compiler et visualiser la trace temporelle d'exécution à l'aide de l'outil `nvvp` (Nvidia visual profiler). Quel commentaire peut-on faire ?
   — On apprend ici à utiliser les outils de profiling : `nvvp` (en mode graphique), `nsys` (en mode ligne de commande) et la bibliothèque `nvToolsExt` pour instrumenter le code CPU.
   — Ouvrir le fichier `helloworld/3/readme.md` et suivre les instructions.

## 2.6 Bande passante mémoire : GPU-GPU et CPU-GPU

Dans un très grande classe de problèmes, le facteur limitant les performances d'un code GPU est l'utilisation de la bande passante mémoire, soit du bus Pci-Express (CPU/GPU) ou du bus mémoire entre le GPU et la mémore de la carte graphique.

On se concentre dans un premier temps sur la bande passante GPU-GPU [3] (i.e. entre le GPU et la mémoire de la carte graphique).

On pourra aussi consulter la documentation effective bandwidth calculation (CUDA best practice guide, section 8) qui explique clairement comment calculer la bande passante effective associée à un noyau CUDA donné.

3. A titre complémentaire, on pourra consulter le blog
https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/

1. Utiliser le code situé dans le répertoire `code/bandwidth`. Ce code contient plusieurs façons de copier un tableau dans un autre.
2. Ouvrir le fichier `bandwidth.cu` et remplir les TODO.
   —
   — Dans un premier temps, on adopte la stratégie naïve : 1 *thread* par élement du tableau à copier. Ecrire le code du kernel `copy`.
   — Dans un deuxième temps, comment modifier le kernel CUDA si on considère que la taille de la grille de *thread* est de taille fixe (indépendante de la taille des tableaux) ? Ecrire le code du kernel `copy2` correspondant. On définira le nombre de blocs comme un multiple du nombre de *streaming multiprocessor*.

   ```
   // utiliser cudaGetDeviceProperties
   cudaDeviceProp prop;
   cudaGetDeviceProperties(&prop, 0);  // pour le device 0
   // utiliser prop.multiProcessorCount
   ```

   — Faites en sorte que le programme prenne en argument de la ligne de commande le nombre d'éléments des tableaux à allouer [4].
3. On utilise les *timer* pour mesurer les temps d'exécution et afficher la bande passante mémoire en GBytes par seconde.

   ```
   // exemple d'utilisation
   #include "CudaTimer.h"
   ...
   CudaTimer timer;
   timer.start();
   // do something
   timer.stop();
   // use timer.elapsed_in_second() to get elapsed time
   ```

4. Exécuter le code plusieurs fois pour différentes tailles de tableau et pour les 2 versions du kernel copy. Que constatez-vous sur la valeur de la bande passante mémoire ?
5. Comparer avec la bande passante maximale possible :

   ```
   // utiliser cudaGetDeviceProperties
   cudaDeviceProp prop;
   cudaGetDeviceProperties(&prop, 0);  // pour le device 0
   printf("  Peak Memory Bandwidth (GB/s): %f\n",
   2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
   ```

On revisite l'étude de la bande passante mémoire GPU-GPU et CPU-GPU (via le bus Pci-Express).

1. Compiler l'exemple `bandwidthTest` du SDK CUDA/C et lire l'aide (`./bandwidthTest --help`)
2. Exécuter la version *release* avec l'option `--mode=quick`
3. Vérifier les ordres de grandeur discutés en cours des 3 différentes bandes passantes mémoire.
4. Exécuter l'exemple avec l'option *range* pour des tailles de transfert de données comprises entre 0 et 100kB par pas de 10kB. Que constatez-vous ?

## 2.7 SAXPY en CUDA/C++

SAXPY est une des fonctions de base que l'on trouve dans les librairies d'algèbre linéaire de type BLAS [5] qui réalise l'opération $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$, où $\mathbf{x}$ et $\mathbf{y}$ sont des vecteurs 1D de réels simple précision.

Copier le répertoire `code/saxpy/saxpy_cuda_c/saxpy.cu`. Utiliser le Makefile pour compiler l'exemple. La compilation fournit un exécutable `saxpy` qui calcule la fonction *saxpy* de 4 manières :

4. Utiliser la routine `atoi` pour convertir une chaîne de caractère en entier.
5. http://en.wikipedia.org/wiki/SAXPY

1. version séquentielle sur le CPU
2. version parallèle OpenMP sur le CPU
3. version parallèle sur le GPU avec kernel CUDA écrit *à la main*
4. version parallèle sur le GPU en utilisant les routines de la librairie cuBlas [6]

Comparer [7] les performances des 4 versions en faisant varier la taille du tableau d'entrée (paramètre `N` en début du fichier).

— Que constatez-vous lorsque la taille du vecteur est de l'ordre de quelques dizaines de milliers ? Pouvez-vous l'expliquer ? [8]

— Que constatez-vous lorsque la taille du vecteur est de l'ordre de quelques millions à dizaines de millions ? Pouvez-vous l'expliquer ?

— On pourra essayer de tracer l'allure grossière de l'évolution des performances en fonctions de la taille du vecteur pour les quatres variantes.

Complément sur les métriques de mesure de performance :
https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/

**Exercice :**

— Lire la page :
https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

— Modifier le code source du kernel `saxpy_parallel` pour implanter la variante dite *grid-stride-loop*.

— Quels sont les avantages de cette variante par rapport à la version dite monolithique ?

## 2.8 Manipulation en mémoire partagée

On utilisera le code du répertoire `code/transposition`

Des explications sur les notions d'accès coalescent à la mémoire et de conflit de banc mémoire seront données pendant le TP.

On pourra utiliser cet exemple pour s'initier aux outils développeur `nsight-sys`. On adaptera les *vieilles planches* suivantes :
https://www.olcf.ornl.gov/wp-content/uploads/2013/01/Hands-On-CUDA-Optimization1.pdf

## 2.9 Algorithmes de reduction

Les algorithmes de reduction (e.g. somme des éléments d'un tableau) sont au cœur de la plupart des applications de calcul scientifique. La mise en œuvre d'une implémentation parallèle sur GPU n'est du tout triviale.

On va commencer par revisiter les planches de Mark Harris. Voir le répertoire `code/reduction`.

Ensuite, placez-vous dans le répertoire des sources de l'exemple `reduction` du SDK CUDA/C++ (`6_Advanced/reduction`).

1. Éditez le code source pour comprendre comment appeller les différentes versions de `kernel`.
2. Ouvrez le document PDF (sous répertoire `doc` dans les sources) pour avoir des explications claires sur les différents `kernel`.
3. Le document PDF nous informe que la bande passante mémoire maximale entre le GPU et sa SDRAM externe est de 86.4 GBytes/s pour le GPU FeForce GTX 8800 (toute première version de l'architecture CUDA, fin 2006). Quelle la valeur correspondante pour notre GPU (sur AWS) ? Utiliser les informations de l'exemple `deviceQuery` (celui du SDK) pour connaître la bande passante théorique maximale.
4. Executer l'exemple réduction pour les différents *kernels*, retrouve-t-on les chiffres indiqués dans le document ?

6. cuBlas est fournie par NVIDIA en installant le toolkit CUDA. Cf /usr/local/cuda/doc/pdf/CUDA_CUBLAS_Users_Guide.pdf.
7. Attention cette étude dépendant TRÈS fortement de la plateforme matérielle utilisée (laptop, desktop or supercalculateur).
8. Utiliser les informations de la commande lstopo pour vous aider à interpréter les résultats sur CPU.

On pourra compléter cet exercice par la lecture du chapitre 12 du livre de Nicholas Wilt : http ://www.cudahandbook.com/.

# 3 Calcul de type stencil / Équation de la chaleur

On se propose de résoudre l'équation de la chaleur (cf annexe D) par la méthode des différences finies sur une grille cartésienne en 2D puis 3D en explorant plusieurs variantes d'implantations avec CUDA.

L'équation de la chaleur représente l'archétype du problème parallélisable [9] par décomposition de domaine. Le travail proposé permet d'explorer les diverses façons d'implanter cette décomposition dans le modèle de programmation CUDA.

1. Utiliser le code situé dans le répertoire `code/heat/heat2d3d_cmake`
2. Tout au long de l'exercice, il faudra éditer le fichier `CMakeLists.txt` pour permettre la compilation des différentes variantes.
3. Une version de référence en C++ est fournie. Elle est constituée de 4 fichiers :
   (a) `heat_solver_cpu.cpp` : contient le `main`, les allocations mémoire et les sorties dans des fichiers pour visualisation,
   (b) `heat_kernel_cpu.cpp` : les routines de résolution du schéma numérique (à l'ordre 2 et à l'ordre 4),
   (c) `param.cpp` : definition des structures de données pour paramètres du problème (taille des tableaux, nombre de pas de temps, sortie graphique, etc...),
   (d) `heatEqSolver.par` : exemple de fichier de paramètres (utilisant le format GetPot)
   (e) `misc.cpp` : les routines utiles annexes (initialisation des tableaux).
4. Compiler, exécuter la version de référence avec les valeurs par défaut des paramètres et de la condition initiale.
5. Reprendre la question précédente en modifiant la condition initiale/condition de bord (fichier `misc.cpp`, routine `initCondition2D`, mettre par exemple tous les bords à 0 sauf un bord à 1) et en calculant 1000 pas de temps avec une sortie graphique tous les 100 pas de temps sur un domaine 2D de taille $256 \times 256$. Visualiser les résultats (images PNG ou fichiers VTK) en vous aidant des informaions contenues dans le sous-répertoire `visu`.

On se propose de porter cet algorithme sur GPU graduellement en terme d'optimisation.

**version naïve 2D** Dans un premier temps, on n'utilisera pas la mémoire partagée du GPU, tous les accès mémoire se feront à partir des tableaux situés en mémoire globale.

1. Editer le fichier `heat2d_solver_gpu_naive.cu`, et remplir de façon appropriée les endroits signalés par `TODO`. Consulter la documentation en ligne de CUDA pour savoir comment utiliser les routines `cudaMalloc` et `cudaMemcpy` [10].
2. Editer de même le fichier `heat2d_kernel_gpu_naive.cu` qui contient le code du *kernel* exécuté sur le GPU
3. Editer le `Makefile` et décommenter les lignes correspondantes à la compilation de cette version.
4. Vérifier que le code est fonctionnel (en comparant aux résultats de la version de référence).
5. Comparer les performances de cette première version sur des tableaux qui ont des tailles en puissance de 2 et non-puissance de 2.

**version simple avec mémoire partagée 2D** Reprendre les questions précédentes en utilisant les fichiers `heat2d_solver_gpu_shmem1.cu` et

9. Voir le site `http://www.cs.uiuc.edu/homes/snir/PPP/` pour avoir plus d'informations sur les différents archétypes de problèmes parallèles
10. Voir la doc en ligne CUDA : `/usr/local/cuda-11.0/doc/html/index.html`

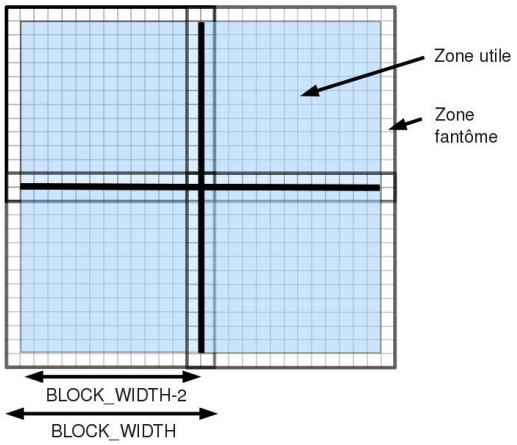


FIGURE 1 – Schéma représentant la grille de blocs de threads utilisée dans le kernel défini dans le fichier `heat2d_kernel_gpu_shmem1.cu` ; les blocs de thread adjacent se recouvre sur une zone de largeur 2.

`heat2d_kernel_gpu_shmem1.cu`. La mémoire coté GPU est à présent allouée avec les routines `cudaMallocPitch` (voir la documentation de l'API CUDA : `/usr/local/cuda/doc/html/group__CUDART__MEMORY.html`) pour respecter les alignements mémoire. On se propose dans cette version d'utiliser la mémoire partagée (meilleurs temps d'accès) ; en revanche la mémoire partagée étant privée à un bloc de *threads*, il est nécessaire d'utiliser un découpage du domaine en blocs qui se chevauchent (voir la figure 1) pour assurer la continuité de l'accès aux données (voir les explications données pendant le TP). On pourra s'aider de ce schéma pour déterminer quel *thread* accède à quelle case mémoire.

1. Après avoir testé cette version, que se passe-t-il en terme de performance si on échange les rôles des indices de *thread* `tx` et `ty` ? Expliquez.

**version simple avec mémoire partagée 2D - variante** Il s'agit d'une légère variante de la version précédente. A présent, on alloue un tableau en mémoire partagée plus grand que la taille des blocs de threads pour tenir compte des cellules *fantômes* qui permettent aux blocs de threads de travailler sur des blocs complètement indépendants. Ne pas hésiter à demander des explications pendant le TP.

**version optimisée avec mémoire partagée 2D (facultatif)** Afin de profiter au maximun de la copie en mémoire partagée, on demande à chaque *threads* de calculer plusieurs cellules du tableau de sortie. Le *kernel* est divisé en 2 *sous-kernels* travaillant respectivement sur les lignes puis les colonnes.

**version 3D naïve** Les limitations de CUDA font que l'on ne peut pas créer des grilles de bloc de threads de n'importe quelle dimension dans la direction `z`. On se propose de reprendre la version 2D naïve et de l'étendre en 3D, chaque *thread* s'occupant de toute une colonne suivant `z`

1. Ecrire le code du *kernel* `heat3d_ftcs_naive_kernel` dans le fichier `heat3d_kernel_gpu_naive.cu`
2. Vérifier qu'il est fonctionnel en comparant les résultats avec ceux de la version de référence.

**version 3D optimisée** On reprend la version 2D qui utilise la mémoire partagée. La partagée ayant une taille maximale qui ne permet pas d'allouer dans la direction `z` la même taille que dans les directions `x` et `y`, on se contente d'allouer le tableau suivant :

```
__shared__ float shmem[3][BLOCK_HEIGHT][BLOCK_WIDTH];
```

qui contient les données de 3 plans en `z` consécutifs. On peut ainsi accéder à toutes les cases mémoires voisines nécessaires à la mise à jour de $\phi_{i,j,k}^{n+1}$ (cf Eq. (4)). A chaque fois qu'un plan est calculé complètement (dans un bloc), on avance en permuttant les indexes des plans (`shmem[z]`) et en chargeant les données du plan suivant dans `shmem[3]`.

1. Remplir les trous laissés dans le *kernel* `heat3d_ftcs_sharedmem_kernel` dans le fichier `heat3d_kernel_gpu_shmem1.cu` ; vérifier la fonctionalité du programme et tester ses performances.
2. On pourra également développer une version où seul le plan médian est stocké en mémoire partagée, les données des plans $z-1$ et $z+1$ étant mise dans des registres (variables locales du thread courant). Cette version présente l'avantage de mieux utiliser les resources matérielles (**équilibrer registres et mémoire partagée**).

### Pour aller plus loin...

Etude de l'influence de certains paramètres sur les performances CPU/GPU.

1. impact de l'ordre du schéma numérique (ordre 2 ou 4).
2. impact de la double précision (ajouter le symbol `USE_DOUBLE` aux flags de compilation, voir le Makefile en tête de fichier)
3. impact de la dimension des tableaux de simulation
   - Essayer la version 3D naïve avec des tailles de tableaux en puissance de 2 et non-puissance de 2.
   - Faîtes la même chose avec la version 3D en mémoire partagée. Que constatez-vous ?
4. impact de la dimension des blocs de *threads* et de leur forme (bloc carré ou alongé suivant $x$)

# 4 Tutoriel OpenACC

Utilisation du matériel pédagogique :

— https://github.com/eth-cscs/SummerSchool2020/tree/master/topics/openacc

— https://github.com/OpenACC/openacc-training-materials

# 5 Activités complémentaires

## 5.1 Initiation à cmake pour un projet CUDA

— Voir le project template https://github.com/pkestene/cuda-proj-tmpl

— Voir le code source du projet LBM / C++

## 5.2 Initiation à python/cuda

Deux situations pratiques :

— si on a une application existante écrite en python, que l'on souhaite accélérer en portant quelques noyaux de calcul en CUDA, on préfèrera utiliser numba, CuPy ou pycuda

— si on a une grande quantité de code existante, écrite en CUDA/C++ et que l'on veut les utiliser depuis python, on préfèrera utiliser une bibliothèque de bindings comme pybind11, cython ou SWIG

### 5.2.1 Tutoriel Numba

Cf https://github.com/ContinuumIO/gtc2020-numba
**Sur votre laptop, assurez-vous d'avoir les prérequis (cf. section 1.3)**

```
# Download notebooks
git clone https://github.com/ContinuumIO/gtc2020-numba.git
cd gtc2020-numba
# Start jupyter lab
jupyter lab
```

#### 5.2.2 CMake / Cuda/C++ / pybind11

Exemple de projet *template* : https://github.com/pkestene/pybind11-cuda

### 5.3 Cuda/C++ / cython / swig

Exemple de projet *template* : https://github.com/pkestene/npcuda-example

## 6 Compléments

### 6.1 Programmation multi GPU

On pourra utiliser le code suivant https://github.com/NVIDIA/multi-gpu-programming-models

## A Accès aux machines

### A.1 Autres resources en ligne pour le calcul sur GPU

— Kaggle (https://www.kaggle.com/) : vous pouvez créer des notebooks jupyter/python et accéder à des GPU de type P100, 30 heures de calcul gratuites par semaine.

— Google Colab (ttps://colab.research.google.com) est un service de Cloud gratuit avec des ressources GPU ; voir par exemple https://colab.research.google.com/notebooks/gpu.ipynb ; vous aurez accès sans doute à des GPU de type K80.

## B API CUDA

On pourra se servir de la documentation officielle de l'API CUDA :
/usr/local/cuda/doc/html/index.html

## C Editeur de texte

Pour avoir la coloration syntaxique du c++ dans emacs sur les fichiers d'extension .cu, tapper : `Echap-x c++-mode` et `entrée`.

## D Schéma numérique pour l'équation de la chaleur

Dans la section 3, on utilise la méthode des différences finies et plus précisément un schéma explicite de type FTCS [11] pour résoudre l'équation de la chaleur

$$\partial_t \phi = D \left[ \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \right], \; 0 \leq x \leq L_x, \; 0 \leq y \leq L_y, \; t \geq 0 \tag{1}$$

sur un domaine rectangulaire muni de conditions de bord et d'une condition initiale.

On étudie deux versions du schéma, qui correspond à deux approximations de la dérivée seconde :

— différence centrée à 3 points, ordre 2 (erreur en $h^2$)

$$\begin{aligned} D^2\phi(x_0) &= \frac{1}{h^2}\left[\phi(x_0-h) - 2\phi(x_0) + \phi(x_0+h)\right] \\ &= \phi''(x_0) + \frac{1}{12}h^2\phi^{(4)}(x_0) + O(h^4) \end{aligned}$$

11. Forward Time Centered Space

— différence centrée à 5 points, ordre 4 (erreur en $h^4$)

$$\begin{aligned} D^2\phi(x_0) &= \frac{1}{12h^2}\left[-\phi(x_0-2h)+16\phi(x_0-h)-30\phi(x_0)+16\phi(x_0+h)-\phi(x_0+2h)\right] \\ &= \phi''(x_0)+\frac{1}{90}h^4\phi^6(x_0)+O(h^6) \end{aligned}$$

Le schéma FTCS (à 3 points) met à jour le tableau 2D $\phi_{i,j}$ (ou 3D $\phi_{i,j,k}$) au temps $t = t_{n+1} = (n+1)\Delta t$ par la relation :

$$\phi_{i,j}^{n+1} = \phi_{i,j}^n + D\frac{\Delta t}{\Delta x^2}\left[(\phi_{i+1,j}^n - 2\phi_{i,j}^n + \phi_{i-1,j}^n) + (\phi_{i,j+1}^n - 2\phi_{i,j}^n + \phi_{i,j-1}^n)\right] \quad (2)$$

$$= R_2\phi_{i,j}^n + R\left[\phi_{i+1,j}^n + \phi_{i-1,j}^n + \phi_{i,j+1}^n + \phi_{i,j-1}^n\right] \quad (3)$$

où $R = D\Delta t/\Delta x^2$ et $R_2 = 1-4R$ (en 2D). De manière similaire en 3D, on trouve

$$\phi_{i,j,k}^{n+1} = R_3\phi_{i,j,k}^n + R\left[\phi_{i+1,j,k}^n + \phi_{i-1,j,k}^n + \phi_{i,j+1,k}^n + \phi_{i,j-1,k}^n + \phi_{i,j,k+1}^n + \phi_{i,j,k-1}^n\right] \quad (4)$$

avec $R_3 = 1-6R$ en 3D.

N.B. : En pratique, pour résoudre l'équation de la chaleur, on préfére utiliser un schéma implicite comme celui de Crank-Nicolson, qui conduit à l'inversion d'un système linéaire tridiagonal et qui possède l'avantage d'être inconditionnellement stable.

On pourra consulter la réference suivante :

— *Finite Difference Methods for Ordinary and Partial Differential Equations*, R. J. LeVeque, SIAM, 2007.
http://www.amath.washington.edu/~rjl/booksnotes.html