# CUDA TP2
# Organisation des threads
# Application à du traitement d'images

## 1 Organisation des threads

Dans le TP précédent, la taille des données est fixée dans le programme ce qui permet également de fixer le nombre de blocs et de threads par bloc. On souhaite maintenant définir des codes plus génériques qui s'adaptent à des données de tailles variables.

### 1.1 Récupération des informations sur le GPU

Á l'aide de la documentation sur la fonction cudaDeviceGetAttribute créer un code pour récupérer :

- le nombre maximum de threads par bloc,
- les dimensions maximales pour les blocs et les grilles,
- le nombre de multiprocesseurs.

### 1.2 Problème 1D

Reprendre les exemples du TP1 et les adapter pour que le nombre de blocs et de threads par bloc soient déterminés automatiquement à partir de la taille des données. Essayer d'optimiser en fonction de la taille des données ainsi que des caractéristiques de la carte.

### 1.3 Problème 2D

Créer 2 matrices de taille $n \times n$ et définir un kernel permettant de les additionner à partir de blocs 2D de threads. Essayer d'optimiser en fonction de la taille des données ainsi que des caractéristiques de la carte.

## 2 Traitement d'images

Code OpenCV pour lire une image au format jpeg :

```
1  #include <opencv2/opencv.hpp>
2
3  #include <iostream>
4  using namespace std;
5
6
7  int main()
8  {
9     // lecture de l'image test.jpg.
10    cv::Mat m = cv::imread("test.jpg");
11
12    // récupération du pointeur vers les données
13    unsigned char* data = m.data;
14
15    // dimensions de l'image
16    cout << "width=" << m.cols << endl;
17    cout << "height=" << m.rows << endl;
```

```
18
19    // toutes les composantes de pixels à 0 (pixels noirs)
20    for( int i = 0 ; i < m.cols * m.rows*3 ; ++i ) {
21      data[ i ] = 0;
22    }
23
24    // écriture de l'image de sortie
25    imwrite( "out.jpg", m );
26
27    return 0;
28  }
```

Pour compiler : `g++ -o test_opencv test_opencv.cpp $(pkg-config --libs --cflags opencv) -O3`

Convertir ce code en CUDA, le kernel CUDA remplissant tous les pixels avec des 0 également.

## 2.1 Niveaux de gris

Pour convertir une image en niveaux de gris, un algorithme simple consiste à appliquer la formule suivante pour chaque pixel :

$$Gris = \frac{307 \times Rouge + 604 \times Vert + 113 \times Bleu}{1024} \tag{1}$$

Créer un code CUDA pour effectuer cette conversion. Attention l'image de sortie est 3 fois plus petite que celle d'entrée. L'image de sortie doit être déclarée de la manière suivante :

```
1  // Allocation de la mémoire pour l'image de sortie.
2  std::vector< unsigned char > out( m.cols * m.rows );
3  // Création de l'image OpenCV en niveaux de gris à partir de la mémoire allouée précédemment.
4  cv::Mat m_out( m.rows, m.cols, CV_8UC1, out );
5  ...
6  // Écriture de l'image en niveaux de gris.
7  imwrite( "out.jpg", m_out );
```

## 2.2 Convolutions

Choisir 2 des convolutions présentées sur la page suivante : convolutions en Gimp et les implanter en CUDA.