Optimize processing time with Cuda

Objective: Learn how to use the Cuda API to port code executing on CPU to tun on GPU. Use Google Colaboratory to run code on a remote GPU. Measure processing time and GPU performance gain.

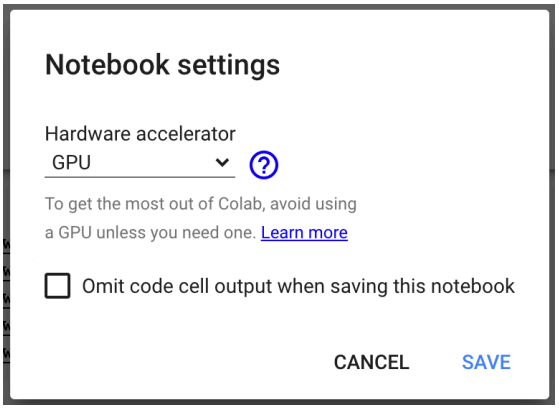
Run a Jupyter Notebook in Google Colaboratory

This lab can be done on Google Colab which is a tool to run a Jupyter Notebook on a remote machine maintained by Google. This service is entirely and only requires to have a Google account. It is even possible to configure Colab to run on a machine with a GPU.

First launch Google Colaboratory and log into your Google account: `https://colab.research.google.com/`.

Select `Import` and upload the `EdgeComputing_TP6.ipynb` Notebook file.

Click the `Runtime` menu and select `Change Runtime type`. Then choose `GPU` in the `Hardware accelerator` dropdown box, then save.



Setup the notebook by running the first two cells by clicking on the `play` icon on the left:

```
!pip install --upgrade git+git://github.com/glemercier/nvcc4jupyter.git
%reload_ext nvcc_plugin
```

```
!mkdir -p /tmp/tp6
!wget --quiet https://github.com/glemercier/nvcc4jupyter/releases/download/tp6/data1.bin -P /tmp/tp6
!wget --quiet https://github.com/glemercier/nvcc4jupyter/releases/download/tp6/data2.bin -P /tmp/tp6
!wget --quiet https://github.com/glemercier/nvcc4jupyter/releases/download/tp6/data3.bin -P /tmp/tp6
!wget --quiet https://github.com/glemercier/nvcc4jupyter/releases/download/tp6/data4.bin -P /tmp/tp6
!wget --quiet https://github.com/glemercier/nvcc4jupyter/releases/download/tp6/image-edf.jpg -P /tmp/tp6
```

The Jupyter Notebook is now ready to run C++ and Cuda code onto a GPU-enabled machine. Go have a look at the third cell and try to understand the code. It is ready to be executed, try to run it and you should see the result at the bottom of the cell.

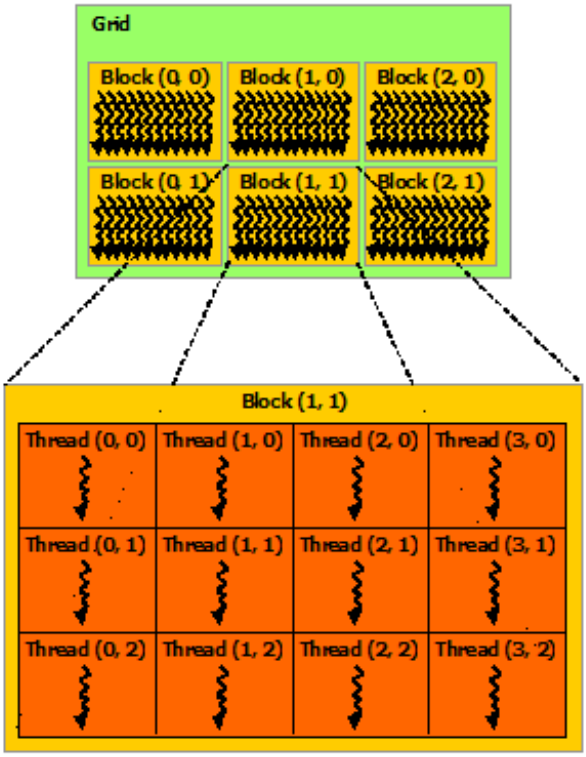
By default the program does the following:

- Run the preprocessing code on CPU and display processing time
- Run the preprocessing code on GPU and display processing time
- Verify that the result of the CPU and GPU processing match
- Run the face distance computaton code on CPU and display processing time
- Run the face distance computaton code on GPU and display processing time
- Verify that the result of the CPU and GPU computations match

Both `preprocessGpu` and `distanceGpu` functions are empty and need to be implemented. Look at the `preprocessCpu` and `distanceCpu` functions to understand the algorithms.

Run Cuda code on the GPU

A GPU is composed of thousands of computing units, organized as a grid of blocks, with each block running X threads:



It is possible to run algorithms in a highly paralllelized fashion. The principle is to write a Cuda `kernel` which is a C function that will be executed on a single computation unit (called a `thread`) of the GPU. When launching a kernel you can specify the number of blocks and threads in each block onto which to run the kernel. The kernel is then executed in parallel on X different threads at the same time.

A Cuda kernel is implemented as a C function as follows:

```
__global__ void simpleCudaKernel(const float *in, float *out, int size) {
  // Get the index corresponding to the computation unit  where this kernel is running
  int idx = threadIdx.x + (blockIdx.x * blockDim.x);

  // Perform computation
}
```

It can be launched from a C/C++ function like this:

```
simpleCudaKernel<<<NUMBER_OF_BLOCKS, THREADS_PER_BLOCK>>>(in, out, size);
```

Note: the pointers passed as parameters to the kernel must point to GPU memory (called `device` memory) otherwise the kernel will crash when trying to access them. It is therefore needed to copy any data residing in CPU memory (called `host` memory) to a buffer in GPU memory before launching the kernel. It can be done with the following Cuda APIs:

```
// Allocate GPU memory of 'size' bytes and store its pointer into 'devPtr'
cudaError_t cudaMalloc ( void** devPtr, size_t size )

// Release GPU buffer previously allocated ato pointer 'devPtr'
cudaError_t cudaFree ( void* devPtr )

// Copy 'size' bytes to (kind: cudaMemcpyHostToDevice) or
// from (kind: cudaMemcpyDeviceToHost) GPU memory
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind )
```

For more details, feel free to refer to the full Cuda documentation available here: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Following is a simple example of code that takes an input buffer of size 1024 and increment each value into an output buffer:

```
#include <cassert>
#include <cuda_runtime_api.h>

#define CUDA_ASSERT(x)    assert(x == cudaSuccess)
#define BUFFER_SIZE       1024
#define THREADS_PER_BLOCK 100
#define NUM_BLOCKS        11 // NUM_BLOCKS * THREADS_PER_BLOCK must be higher than BUFFER_SIZE

__global__ void incrementCudaKernel(const int *in, int *out) {
  // Get the index in the array based on the thread and block IDs where this kernel is running
  const int idx = threadIdx.x + (blockIdx.x * blockDim.x);

  // Make sure we are not running on a thread ID > to the actual buffer size
  if (idx < BUFFER_SIZE) {
    // Read the input buffer at this index and stored the incremented value
    // at the same index in the output buffer
    out[idx] = in[idx] + 1;
  }
}

int main(void) {
  int inputArray[BUFFER_SIZE];

  // Fill up inputArray with some data ...

  // Allocate buffers in device memory and copy the input data to it
  int *inputGpu, *outputGpu;
  CUDA_ASSERT(cudaMalloc(&inputGpu, BUFFER_SIZE * sizeof(int)));
  CUDA_ASSERT(cudaMalloc(&outputGpu, BUFFER_SIZE * sizeof(int)));
  CUDA_ASSERT(cudaMemcpy(inputGpu, inputArray, BUFFER_SIZE * sizeof(int),
      cudaMemcpyHostToDevice));

  // Launch the kernel
  incrementCudaKernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(inputGpu, outputGpu);

  // Copy the output buffer to host memory
  int outputArray[BUFFER_SIZE];
  CUDA_ASSERT(cudaMemcpy(outputArray, outputGpu, BUFFER_SIZE * sizeof(int),
      cudaMemcpyDeviceToHost));

  // Check the output buffer is an incremented version of the input buffer
  for (int i = 0; i < BUFFER_SIZE; i++) {
    assert(outputArray[i] == inputArray[i] + 1);
  }
}
```

It is also possible to have shared memory across all threads in a cuda kernel:

```
__shared__  int shared_array[256];
```

This memory zone is shared between all threads and you should therefore be very careful when accessing it to avoid race conditions, illustrated as below with a kernel that increments a shred value:

```
__shared__ int shared;

// thread 0 initializes the shared variable
shared = 0;

// thead 1 executes
val = shared;   // val = 0
val += 1;       // val = 1
shared = val;   // shared = 1

// thread 2 executes         ||  // thread 3 executes
val = shared;  // val = 1    ||  val = shared;  // val = 1
val += 1;      // val = 2    ||  val += 1;      // val = 2
shared = val;  // shared = 2 ||  shared = val;  // shared = 2

// Since thread 2 and 3 executed in parallel and accessed the
// shared variable at the same time, resulting in a wrong
// result 2 where it should have been 3
```

To avoid such issues, we need to use ”atomic” operations that guarantee to be performed in a non-preemptible way. Cuda offers a lot of atomic operations we can use for this kind of purpose:

```
int atomicAdd(int* address, int val);
int atomicSub(int* address, int val);
int atomicExch(int* address, int val);
int atomicMin(int* address, int val);
int atomicMax(int* address, int val);
unsigned int atomicInc(unsigned int* address, unsigned int val);
unsigned int atomicDec(unsigned int* address, unsigned int val);
int atomicCAS(int* address, int compare, int val);
int atomicAnd(int* address, int val);
int atomicOr(int* address, int val);
int atomicXor(int* address, int val);
```

Note: These APIs exist with a lot of types other than ’int’, refer to the documentation for complete details: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Sometimes we also need to synchronize the threads before performing a specific operation that requires all the threads to reach a certain point in the execution of the kernel. This can be done by calling the `__syncthreads()` API that generates synchronization points between all threads. See the example below, illustrating how to use atomic operations and synchronization points to properly handle a shared value to be incremented by all threads:

```
__global__ void incrementCudaKernel(unisgned int *out) {
  const int idx = threadIdx.x;

  // Declare shared integer, it cannot be statically initialized
  __shared__ unsigned int incremented;

  // Since it has to be dynamically initialized, have thread 0 do it
  if (idx == 0) {
    incremented = 0;
  }

  // Have all the threads synchronize to this point to make sure they all start
  // incrementing after the shared variable is initialized by thread 0
  __syncthreads();

  // Increment the shared value atomically, overflows to 0 if > 1000000
  atomicInc(&incremented, 1000000);

  // Synchronize all the threads to make sure they all had the chance to increment
  // the shared value
  __syncthreads();

  // Store the final value in the output buffer, have thread 0 do it
  if (idx == 0) {
    *out = incremented;
  }
}
```

Implement the GPU processing functions

You should now know everything you need in order to fill the following functions in the Jupyter Notebook:

```
// Cast the pixels contained in 'inputImage' from uint8_t to float and normalize the R,G,B components
// of each pixel around 0.5 as per the following formula: a = ((a / 255)  - 0.5) / 0.5
// Return the resulting image as a pointer to a GPU buffer allocated by the function.
// 'size' is the number of pixels * 3 channels of the image, or the number of total elements stored
// in the array. See 'preprocessCpu' for CPU implementation.
float* preprocessGpu(const uint8_t *inputImage, int size);

// Compute the distances between inputs and store them ino the 'distance' array. 'size' is the number
// of elements in the arrays (here it's 512):
//
// distance[0] = distance(a, b)
// distance[1] = distance(a, c)
// distance[2] = distance(a, d)
//
// The distance between two arrays a and b  is computed as per the formula:
//   distance = square_root(sum(pow2(a - b)))
//
// See 'distanceCpu' for CPU implementation
void distanceGpu(const float *a, const float *b, const float *c, float *d, float *distance, int size);
```

Feel free to look up online for Cuda examples or ask your instructor for pointers if you are lost. When you run the code cell in Colab, you should see the result and see if your Cuda implementation is working well. To ensure your code is good, there should be no error and hopefully the processing time of your Cuda implementation is faster than the original CPU code.

If you did it, congratulations! You may go further and work on the following tasks:

- Reuse your Cuda implementations in the code of lab 5 to accelerate your face recognition code running on the Jetson Nano
- See if you could haved used the Thrust library to implement some of it more easily
- Try to optimize the GPU occupancy even more by playing around with the `block count` and `threads per block` launch parameters