

CUDA for GPU Computing

Master 2 Calcul Scientifique - 2011 / 2012

Thé Van Luong and Nouredine Melab

University of Science and Technology (USTL/IEEEA),

Lille's Computer Science Laboratory (LIFL),

INRIA Lille Nord Europe - Villeneuve d'Ascq



This document is organized into four parts which respectively deal with the following points:

- ❶ Familiarize with the CUDA toolkit and learn to manipulate some basic concepts.
- ❷ Learn to program a small application on GPU using the CUDA threads model.
- ❸ Familiarize with some advanced concepts including the memory management.
- ❹ Learn to program some practical applications.

1 Getting to the environment

1.1 Installation

● For years, the use of graphics processors was dedicated to graphics applications. Recently, their use has been extended to other application domains (e.g. computational science) thanks to the publication of the CUDA (Compute Unified Device Architecture) development toolkit that allows GPU programming in C-like language. In some areas such as numerical computing, we are now witnessing the proliferation of software libraries such as CUBLAS for GPU. With the arrival of OpenCL as the open standard programming language on GPU and the arrival of future compilers for this language, applications on GPU will generate a growing interest.

● For using the CUDA environment on your personal machine (already installed on the university computers ...), you need to install two things:

1. The Developer Drivers (drivers for the video card)
2. The CUDA Toolkit (development tool for programming applications)

🚲 Go to the {NVIDIA_GPU_Computing_SDK}/C/src/deviceQuery directory. Compile the source code with the commands `make clean` and `make` (you have to learn these two commands ...). Then, execute the associated program with the command `../bin/linux/release/deviceQuery`. What does the program ?

1.2 Compilation

● Basically, the compilation process is separated into two phases: 1) the `g++` compiler for the CPU host code and 2) the `nvcc` compiler mixes both host and GPU device code. `nvcc` is a compiler driver that simplifies the process of compiling C and assembly codes. It provides simple command line options and executes them by invoking the collection of tools that implement the different compilation stages. More details about `nvcc` are given in the *NVIDIA CUDA C Programming Guide*. For our different implementations, we will use each time four different files:

- {program_name}.cu (main file for the host part)
- {program_name}_kernel.cu (specific part for the GPU device)
- {program_name}_cpu.h (C++ source files for a sequential version)
- {program_name}_conf.h (additional macro definitions for the programs)

🚲 Create the four different files quoted above. These are some examples of what could be such files:

```

/* test.cu */

#include <stdio.h>
#include "test_conf.h"
#include "test_cpu.h"
#include "test_kernel.cu"

int main (int argc, char* argv[]) {

    printf("program test\n");

}

```

```

/* test_kernel.cu
   it does not contain anything for the moment
*/

```

```


/* test_cpu.h
   it does not contain anything for the moment
*/


```

```

/* test_conf.h
   it does not contain anything for the moment
*/


```


||  *Now you have some code skeletons for an application. We will always use the same structure for the future programs.*

 Compile the program with `nvcc {program_name}.cu -o {program_name}` and execute it.

2 Allocations and copies

2.1 Allocations and copies in C

||  *To get in touch with the different CUDA functions, we are going to review some basic C functions. A perfect practice of these techniques is the key point to use basic CUDA functions.*


 In your `{program_name}.cu` file, allocate an array of 10 integers (with the `malloc` function). Then, initialize this array with the values from 0 to 9 (i.e. `T[0] = 0`, `T[1] = 1`, `T[2] = 2`, ...). Execute your program by displaying the different values.

```

#include <stdlib.h>
void* malloc (size_t size);
malloc() allocates size bytes and returns a pointer to the allocated memory.

void free (void* ptr);
free() frees the memory space pointed to by ptr, which must have been returned by a
previous call to malloc().

```

 Generalize this approach with an array size provided at runtime (in command line).

```

#include <stdlib.h>
int atoi (const char* nptr);
The atoi() function converts the initial portion of the string pointed to by nptr
to int.

```

- 🚲 Now we would like a copy of this array (with the `memcpy` function) where each element is increased by one (i.e. $T2[0] = T[0] + 1$, $T2[1] = T[1] + 1$, $T2[2] = T[2] + 1$, ...). Create this procedure in the `{program_name}.cpu.h` file and call it from the `{program_name}.cu` file.

```
#include <string.h>
void memcpy (void* dest, const void* src, size_t n);
The memcpy() function copies n bytes from memory area src to memory area dest.
```

⏹ *Each CPU function that is defined should be located in the `{program_name}.cpu.h` file. It is not mandatory but such decomposition should be performed to improve the readability of your code.*

2.2 Allocations and copies in CUDA

⏹ *In general-purpose computing on graphics processing units, the CPU is considered as a host and the GPU is used as a device coprocessor. This way, each GPU has its own memory and processing elements that are separate from the host computer. Data must be transferred between the memory space of the host and the memory of GPU during the execution of programs. Before manipulating any kernel on GPU, basic allocations and copies must be done in CUDA.*

⏹ *We would like an array of 10 integers on the GPU device which is a copy of a previous array where the values vary from 0 to 9 (i.e. $T[0] = T_device[0] = 0$, $T[1] = T_device[1] = 1$, $T[2] = T_device[2] = 2$, ...). For verifying that the process is done correctly, these are the following steps to achieve:*

1. *An allocation of the GPU device array with a size of 10 (`cudaMalloc` function).*
2. *A copy of the CPU initial array to the GPU device array (`cudaMemcpy` function).*
3. *HERE is the kernel (but we will manage it later for other programs).*
4. *A copy the GPU device array to a new auxiliary array (`cudaMemcpy` function).*
5. *Print the values of the new auxiliary array.*
6. *A deallocation the GPU device array (`cudaFree` function).*

⏹ *As you may guess, no `printf` mechanism can be performed on the GPU device for many graphic cards. That is the reason why, in general, an auxiliary array is used to display the different values performed on the GPU (here a basic copy).*

- 🚲 Use the `cudaMalloc` function to allocate an array declared on the GPU device with a size of 10. Look at carefully to the function header and try to compile your program.

```
void cudaMalloc (void** ptr, size_t size)
cudaMalloc() allocates size bytes on the GPU device memory.
```

⏹ *Unless the standard `malloc` function, the `cudaMalloc` function does not return a pointer to the allocated memory. Therefore, since such mechanism is necessary, an appropriate cast and a reference (denoted by `&`) should be used instead.*

- 🚲 Perform a copy from the initialized array (whose values vary from 0 to 9) to the GPU device array (using the `cudaMemcpy` function). Try to compile your program.

```
void cudaMemcpy (void* dest, void* src, size_t n, enum cudaMemcpyKind direction);  
The cudaMemcpy() function copies n bytes from memory area src to memory area dest.
```

The copy direction is defined according to the following enumeration :

```
enum cudaMemcpyKind {  
    cudaMemcpyHostToDevice,  
    cudaMemcpyDeviceToHost,  
    cudaMemcpyDeviceToDevice  
}
```

🚲 Declare and allocate a new auxiliary array of 10 integers defined on CPU (using malloc). Remember that this array must not be initialized ! Then perform a copy from the GPU device array to this new array (cudaMemcpy function). Try to compile your program.

🚲 Display the values of this new array and deallocate all your arrays. Then, execute your program.

```
void cudaFree (void* ptr);  
cudaFree() frees the device memory space pointed to by ptr.
```

⏹ *If the work is correctly done, you should find the same values as before. Otherwise, you should revise your code from the cudaMalloc step.*

3 Kernel execution

⏹ *Each processor device on GPU supports the single program multiple data (SPMD) model, i.e. multiple autonomous processors simultaneously execute the same program on different data. For achieving this, the concept of kernel is defined. The kernel is a function callable from the host and executed on the specified device simultaneously by several processors in parallel.*


⏹ *The CUDA toolkit introduces a model of threads which provides an easy abstraction for SIMD architecture. The concept of a GPU thread does not have exactly the same meaning as a CPU thread. A thread on GPU can be seen as an element of the data to be processed. Compared to CPU threads, GPU threads are lightweight. That means that changing the context between two threads is not a costly operation.*

⏹ *Regarding their spatial organization, threads are organized within so called thread blocks. A kernel is executed by multiple equally threaded blocks. Blocks can be organized into a one-dimensional or two-dimensional grid of thread blocks, and threads inside a block are grouped in a similar way. All the threads belonging to the same thread block will be assigned as a group to a single multiprocessor, while different thread blocks can be assigned to different multiprocessors. Thus, a unique id can be deduced for each thread to perform computation on different data.*

⏹ *In the following, we would like to increment each value of our device array by one. For doing this, we are going to proceed step by step.*


🚲 In the {program_name}_kernel.cu file, implement the kernel which adds one to the first value of the array (i.e. T_device[0] += 1). Then, try to compile it. The skeleton of the kernel definition is given in the following:


```
__global__ void kernel_name (parameters)
{
    some code
}
```

||  The `__global__` qualifier defines the kernel which is callable from the CPU host.

- 🚲 In the `{program_name}.cu` file, call the associated kernel using only **one** block with **one** thread. Display the obtained array (using the auxiliary array) and execute your program. To call a kernel:


```
kernel_name<<<dim3 dG, dim3 dB>>>(parameters)
```

||  We will come back later on the grids (`dim3 dG`) and blocks structures (`dim3 dB`). In our case, these numbers will represent respectively the number of blocks and the number of threads per block in one dimension.


 As you may guess, we would like to generalize this approach for an arbitrary size on GPU. Before doing that, we would like to perform the sequential version.

- 🚲 In the `{program_name}_cpu.h` file, implement the `void inc_cpu(int* a, int n)` function which increases each value of the array by one (unless a previous version, this one does not copy anything).
- 🚲 In the `{program_name}.kernel.cu` file, implement the `__global__ void inc_gpu(int* a, int n)` which does the same task in parallel. Perform the execution for a number of 64 threads (using one block). Notice that `__global__` functions have access to some automatically defined variables:

```
dim3 threadIdx;
Thread index within the block.
In one dimension, threadIdx.x returns the x coordinates (int value).
```

||  Since `n` is a constant, passing such parameter in the kernel call does not require an explicit transfer from the CPU to the GPU. Indeed, it is automatically performed by the `nvcc` compiler.

- 🚲 Try to vary the number of threads (256, 512, 1024, 2048). What observations can you make ?

 According to the CUDA programming guide, the maximal number of threads per block vary between 512 and 1024 depending on the architecture. Exceeding this number may not guarantee the validity of the obtained results. To deal with this constraint, we will try to execute many thread blocks.

- 🚲 Adapt your code to launch many thread blocks. Notice that `__global__` functions have access to some automatically defined variables:

```
dim3 blockIdx;
Block index within the grid.
In one dimension, blockIdx.x returns the x coordinates (int value).

dim3 blockDim;
Dimensions of the block in threads.
In one dimension, blockDim.x returns the number of threads per block x.
```

⊛ *From a hardware point of view, the number of blocks should be at least twice the number of multiprocessors to keep the GPU busy. In addition to this, since threads of thread block are executed on one multiprocessor concurrently in warps of 32 threads (active threads), the number of threads per block should be at least a multiple of 32. We will study later the influence of such parameters in terms of performance. However, we can notice that by fixing the number of threads per block with a multiple of 32, the total number of threads may be greater than the number of tasks to achieve (e.g. n values of an array). As a consequence, some threads may access to incoherent values leading to a segmentation fault.*

🚲 Adapt your code (kernel code and kernel call) to launch an arbitrary number of threads.

✌ *In this first TP, we have been familiarized with the GPU environment in particular by using the CUDA toolkit.*

✎ *The GPU device memory is clearly separated from the CPU host memory. Therefore, allocations and copies must be performed on the GPU.*

✎ *The GPU threads model follows the SPMD model where the same program is executed on different data.*

PART 2: CUDA threads model.

⌚ The goal of this TP is to familiarize with the threads and block structures. For doing that, we will implement some small applications.

4 Basic computations and performance measurements

4.1 Basic calculation involving logarithm

⌚ To go on with the concepts learned in the previous TP, we would like to implement a small computation. Given an array T of n elements and a certain number of iterations nb_iters , we would like to implement the following calculation:

```
for (iter = 0; iter < nb_iters, iter++){
    T[id] = logarithm_2(T[id]*T[id]+T[id]+iter);
}

/* id varies between 0 and n-1 */
```

🚲 Create a new repertory with the different skeletons of your program (i.e. `{program_name}_cpu.h`, `{program_name}.cu` and `{program_name}_kernel.cu`). Adapt your main program to provide two parameters in command line (the array size n and the number of iterations nb_iters).

🚲 In your main program, create an array where the values are taken randomly in $[1; nb_iters[$.

```
#include <stdlib.h>
int rand (void);
```

The `rand()` function returns a value between 0 and `RAND_MAX`.

```
#include <stdlib.h>
void srand (unsigned int seed);
```

The `srand()` function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by `rand()`.

```
#include <time.h>
time_t time (time_t *tloc);
```

The `time()` function returns the value of time in seconds since January 1 1970.

🚲 Implement the `logarithm_2` method in the `{program_name}_cpu.h` (this function returns the number of times that a number can be divided by 2 until reaching the value 1 ...).

🚲 With the `logarithm_2` function, implement the `compute_cpu` method in the `{program_name}_cpu.h` which performs the calculation mentioned above.

🚲 Implement the `compute_gpu` method which performs the calculation in parallel on GPU. The following example shows an use of a `__device__` function:


```

__device__ int _logarithm_2 (int number) {

    ...

}

__global__ void kernel (parameters) {

    ... here a call to the _logarithm_2() function

}

```

⊛ The `__device__` qualifier represents an auxiliary function. Unless the `__global__` qualifier which represents the kernel, a `__device__` function cannot be called from the host CPU. Furthermore, since no stack structures exist on GPU, there is no possibility to perform any recursivity.

- 🚲 Verify the conformity of your results between the different versions by varying the two parameters in command line (n and nb_iters).

4.2 Performance measurements

⊛ Now we would like to measure the performance of the different implementations i.e. the GPU acceleration in comparison with a single-core CPU. To perform this, we will use some libraries to measure precisely the elapsed time of a given task. The following example gives an illustration on how to do this:

```

#include <sys/time.h>
#include <unistd.h>

timeval t1, t2;
double elapsedTime;

gettimeofday(&t1, NULL);

... TASK TO MEASURE

gettimeofday(&t2, NULL);

// compute and print the elapsed time in sec
elapsedTime = t2.tv_sec - t1.tv_sec;      // sec
elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000000.0;  // us to sec

printf("Processing time : %1.11f (s) \n", elapsedTime);

```

- 🚲 Implement the different timers to measure the GPU acceleration for the previous program.
- 🚲 Try to vary the parameters in command line with some high values (e.g. $n = 10000$ and $nb_iters = 10000$).
- 🚲 Try to vary the number of threads per block and observe its influence in terms of performance.

⊛ When performing your experiments, it is possible that you do not obtain very good speed-ups with your graphic card. You should also test your program on high-performance graphic cards based on GT200, GT400 or Tesla series. Some of these graphic cards are available on grid'5000 in the Grenoble site if you look at the adonis cluster: `oarsub -I -p "cluster='adonis'"`. Since they are valuable machines, these resources might be already occupied by some other users.

5 Matrix multiplication

5.1 Two-dimensional thread structures

Ⓢ Now we would like to manipulate the thread hierarchy through a matrix multiplication. For convenience, the variable `threadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional, two dimensional, or three-dimensional thread index. As a consequence, this provides a natural way to invoke computation across the elements for vector, matrix and volume structures.

🚲 Create a new repertory with the different skeletons of your program (i.e. `{program_name}_cpu.h`, `{program_name}.cu`, `{program_name}_kernel.cu` and `{program_name}_conf.h`).

Ⓢ Since there are a lot of parameters implying many arguments during the kernel call, those parameters will be defined as macros in the `{program_name}_conf.h`. This way, those macros can be called from any file. In a general way, this mechanism slightly improves the implementation performance but such macros need to be known at the compilation time (no command line at runtime). The following example gives an example of such macros:

```
#ifndef _MATRIXMUL_H_
#define _MATRIXMUL_H_

// Thread block size
#define BLOCK_SIZE 16

// Matrix dimensions
#define WA (3 * BLOCK_SIZE) // Matrix A width
#define HA (5 * BLOCK_SIZE) // Matrix A height
#define WB (8 * BLOCK_SIZE) // Matrix B width
#define HB WA // Matrix B height
#define WC WB // Matrix C width
#define HC HA // Matrix C height

#endif // _MATRIXMUL_H_
```

🚲 Create the two matrices with a random initialization of float values.

🚲 Implement the matrix multiplication on CPU in the `{program_name}_cpu.h` file.

Ⓢ Whilst standard array notation of `A[row][col]` can be used in C, the declaration of such arrays does not allow us to test if there is enough memory in the DRAM stack to initialise it. This becomes important when dealing with very large array sizes. Also, CUDA requires that arrays use a single contiguous block of linear memory. So, turns out 2 dimensional arrays are not advised on kernels since allocating a 2D array on the GPU is still in a linear block of memory. The row-major format is employed to reference the linear array in a 2D abstraction. Each row of the 2D array is set out end-to-end, and referenced using: `Linear_address = row * 2D_width + column`.

🚲 Implement the multiplication for square matrices on GPU using only **one** block of two-dimensional threads.

```
/* kernel variables */

dim3 threadIdx;
Thread index within the block.
threadIdx.x returns the x coordinates (int value).
threadIdx.y returns the y coordinates (int value).
```

```
typedef struct dim3 { unsigned int x,y,z; }

/* constructor example callable from the host*/
dim3 mystruct (2,3)
-> mystruct.x = 2
-> mystruct.y = 3
-> mystruct.z = 1
```

⏹ *For problems with two-dimensional domains such as matrix math or image processing, it is often convenient to use two-dimensional thread indexing to avoid annoying translations from linear to rectangular indices.*

- 🚲 To test the validity of your results, implement a function which compares each element of each resulting matrices (the one obtained on CPU and the other one on GPU). This function must return false if two elements located on a same column and row are different.

⏹ *The floating point precision might not be the same according to the different graphic cards. To deal with this, your program can accept an error of 0.0001f.*

- 🚲 Try to test your program with different matrix sizes. Try to test your program with more than 512 elements (i.e. threads). What can you conclude ?

⏹ *As previously said, according to the CUDA programming guide, the maximal number of threads per block vary between 512 and 1024 depending on the architecture. Exceeding this number does not guarantee the validity of the obtained results.*

5.2 Two-dimensional block structures

- 🚲 Try to generalize your program for non-square matrices for an arbitrary size using thread blocks. For doing that, use the macro `TILE_WIDTH` which represents the width of a two-dimensional threads block. For the sake of simplicity, the dimensions of matrices will be chosen as multiples of the block width.

```
/* kernel variables */

dim3 blockIdx;
Block index within the grid.
blockIdx.x returns the x coordinates (int value).
blockIdx.y returns the y coordinates (int value).
```

- 🚲 Try to vary different matrix sizes to measure the performance of your algorithm.
- 🚲 Try to vary the `TILE_WIDTH` value and see its impact in terms of performance. Try to test your program with a `TILE_WIDTH` value greater than 26. What can you conclude ?



In this TP, we have been familiarized with the CUDA threads model in particular by manipulating block and threads structures.

- 📝 *Blocks can be organized into a one-dimensional or two-dimensional grid of thread blocks, and threads inside a block are grouped in a similar way. This abstraction allows to perform computations on the different data whatever the used architecture.*
- 📝 *Varying the number of threads per block and the number of blocks being processed by the multiprocessors has an influence in terms of performance.*

PART 3: Memory management

● From a hardware point of view, graphics cards consist of streaming multiprocessors, each with processing units, registers and on-chip memory. Since multiprocessors are used according to the SPMD model, threads share the same code and have access to different memory areas. Communication between the CPU host and its device is done through the global memory. Since this memory is not cached and its access is slow, one needs to minimize accesses to global memory (read/write operations) and reuse data within the local multiprocessor memories.

● Graphics cards provide also read-only texture memory to accelerate operations such as 2D or 3D mapping. Texture memory units are provided to allow faster graphic operations. Constant memory is read only from kernels and is hardware optimized for the case where all threads read the same location. Shared memory is a fast memory located on the multiprocessors and shared by threads of each thread block. This memory area provides a way for threads to communicate within the same block. Registers among streaming processors are private to an individual thread, they constitute fast access memory. In the kernel code, each declared variable is automatically put into registers. Local memory is a memory abstraction and is not an actual hardware component. In fact, local memory resides in the global memory allocated by the compiler. Complex structures such as declared array will reside in local memory.

6 The shared memory

6.1 Reverse array

● Given a one-dimensional array A containing the integer values $[0, 1, \dots, \text{sizeA} - 1]$, we would like to reverse the values in order to obtain a new array containing the values $[\text{sizeA} - 1, \dots, 1, 0]$.

🚲 Implement the reverse array program on CPU (`{program_name}_cpu.h`).

🚲 Implement the GPU version for an arbitrary size. Verify the validity of the obtained results.

```
/* kernel variables */  
  
dim3 gridDim;  
Dimensions of the grid in blocks.  
In one dimension, gridDim.x returns the number of blocks per grid.
```

● From a hardware point of view, graphics cards consist of multiprocessors, each with processing units, registers and on-chip memory. Accessing global memory incurs an additional 400 to 600 clock cycles of memory latency. As a consequence, since this memory is not cached and its access is slow, one needs to minimize accesses to global memory (read/write operations) and reuse data within the local multiprocessor memories. To accomplish this, the shared memory is a fast on-chip memory located on the multiprocessors and shared by threads of each thread block. This memory can be considered as a user-managed cache which can provide great speedups by conserving bandwidth to main memory. Furthermore, since the shared memory is local to each threads block, it provides a way for threads to communicate within the same block.

🚲 Implement a new version which uses the shared memory. These are the key points to follow:

1. Load one element per thread from global memory and store it in reversed order into temporary shared memory.
2. Perform synchronization barriers until all threads in the block have written their data to the shared memory.
3. Write the data in forward order from the shared memory to the global memory.
4. Remember that the shared memory is only local to each threads block.

The `__shared__` qualifier, declares a variable that:

- Resides in the shared memory space of a thread block,
- Has the lifetime of the block,
- Is only accessible from all the threads within the block.

When declaring a variable in shared memory as an external array such as
`extern __shared__ float shared[];` (kernel code)
the size of the array is determined at launch time
(third argument of the kernel call `<<<arg1, arg2, arg3>>>` in the host code
where `arg3=NbThreads*NbBlock*sizeof(float)`).

Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the `__syncthreads()` intrinsic function; `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed.

- 🚲 Verify the validity of your results and measure the performance in comparison with the previous versions.

6.2 Reduction operation

|| 🎯 *Given a one-dimensional array containing numbers, we would like to calculate the sum of all the elements.*

- 🚲 Implement the function on CPU which returns the array sum (`{program_name}.cpu.h`). For doing that, we will initialize the initial array with small integer values (e.g. values varying between -10 and 10).
- 🚲 Implement the associated version using the shared memory for dealing with array sizes which are a power of two. For doing that, we will use a reduction operation on the different elements of the array. An illustration of such mechanism can be found in <http://www.lifl.fr/~luong/arbre.pdf>. These are the key points to achieve this:

- Most of the writing/reading operations are performed on the shared memory.
- As illustrated in the previous figure, synchronization must be performed between each level of the tree.
- The sum of two values of the array may be written in the address memory corresponding to the first value.
- Remember that the shared memory is local to each threads block.

- 🚲 Test and verify the conformity of your results. Measure the performance of your implementation in comparison with the CPU one for a large number of elements (e.g. 131072 or any power of two).
- 🚲 Adapt your code for dealing with an array size which is a non power of two.
- 🧠 The parallel reduction can be also applied for commutative operators (e.g. +, ×, min, max).
- 🚲 Try to adapt your code to find the minimum of the array (instead of the sum).

7 The texture memory

7.1 Matrix vector multiplication

- 🚲 Implement a new program on CPU which performs a matrix vector multiplication for an arbitrary size.
- 🚲 Implement the associated version on GPU using one-dimensional threads.
- 🚲 Verify the validity of your results and compare the performance results with the CPU version.

🧠 In GPGPU paradigm, each block of threads is split into SIMD groups of threads called warps. At any clock cycle, each processor of the multiprocessor selects a half-warp (16 threads) that is ready to execute the same instruction on different data.

🧠 For more efficiency, global memory accesses must be coalesced, which means that a memory request performed by consecutive threads in a half-warp is associated with precisely one segment. The requirement is that threads of the same warp must read global memory in an ordered pattern. If per-thread memory accesses for a single half-warp constitute a contiguous range of addresses, accesses will be coalesced into a single memory transaction. Otherwise, accessing scattered locations results in memory divergence and requires the processor to perform one memory transaction per thread. The performance penalty for non-coalesced memory accesses varies according to the size of the data structure.

🧠 Optimizing the performance of GPU applications often involves optimizing data accesses which includes the appropriate use of the various GPU memory spaces. The use of texture memory is a solution for reducing memory transactions due to non-coalesced accesses. Texture memory provides a surprising aggregation of capabilities including the ability to cache global memory (separate from register, global, and shared memory). Indeed, texture memory provides an alternative memory access path that can be bound to regions of the global memory. Each texture unit has some internal memory that buffers data from global memory. Therefore, texture memory can be seen as a relaxed mechanism for the threads to access the global memory because the coalescing requirements do not apply to texture memory accesses.

- 🚲 Implement a new version which uses the texture memory.

```
/* declaration: this global variable must be visible on host and device */
texture <int, 1, cudaReadModeElementType> texRefObj;
```

```
/* host code: bind texture memory of global memory */
cudaBindTexture(NULL, texRefObj, mystructdeclaredasglobalmemory);
```

```
/* host code: unbind texture memory */
cudaUnBindTexture(texRefObj);
```

```
/* kernel code: accessing the ith element in the texture */
tex1Dfetch(texRefObj,i);
```

- 🚲 Verify the validity of your results and measure the acceleration in comparison with the previous versions.



In this TP, we have been familiarized with the CUDA memory management in particular by manipulating the shared and the texture memory.

- 🖋️ *The shared memory is a fast memory located on the multiprocessors and shared by the threads of each block.*
- 🖋️ *The read-only texture memory allows to accelerate operations such as 2D or 3D mapping. Binding texture on global memory allows to improves random access patterns.*

PART 4: Applications on GPU

📌 The goal of this TP is to provide an insight of different possible GPU applications. We will mainly focus on the performance of GPU programs in terms of efficiency. As a consequence, for the two following programs, your final implementations will be tested on a GTX 480 card and a comparison between the different student results (i.e. the execution time) will be made.

8 Matrix multiplication

📌 The following example gives a sequential version of the matrix multiplication:

```
////////////////////////////////////
//! Compute reference data set
//! C = A * B
//! @param C          reference data, computed but preallocated
//! @param A          matrix A as provided to device
//! @param B          matrix B as provided to device
//! @param hA         height of matrix A
//! @param wB         width of matrix B
////////////////////////////////////
void computeGold(float* C, const float* A, const float* B, unsigned int hA,
unsigned int wA, unsigned int wB)
{
    for (unsigned int i = 0; i < hA; ++i)
        for (unsigned int j = 0; j < wB; ++j) {
            float sum = 0;
            for (unsigned int k = 0; k < wA; ++k) {
                float a = A[i * wA + k];
                float b = B[k * wB + j];
                sum += a * b;
            }
            C[i * wB + j] = sum;
        }
}
```

🚲 Implement an optimized version of the matrix multiplication on GPU. Your program must meet the following constraints:

- Matrices values are float numbers taken between [0; 1].
- Matrices must be non-square matrices.
- Your implementation must deal with large matrices size.
- The kernel memory management must be applied (e.g. the use of shared or texture memory).
- Your parameters must be defined in a configuration file ({program_name}.conf.h): the number of threads per block, the matrices widths, the matrices heights, etc.
- To compare the results between a CPU and a GPU implementation, you must use the following test to compare the results of the two arrays:

```
#include <cutil_inline.h>
...
// check if the device result is equivalent to the expected solution
CUTBoolean res = cutComparefe(reference, array_calculated,
size_elements, 0.0001f);
printf("Test %s\n", (1 == res) ? "PASSED" : "FAILED");
```


⊙ *Extra advanced optimization: A common global memory access pattern is when each thread of index (tx, ty) uses the following address to access one element of a 2D array of width **width**, located at address **BaseAddress** of type **type*** (where **type** might be **int**, **float**, ...). For these accesses to be fully coalesced, both the width of the thread block and the width of the array must be a multiple of the warp size (i.e. 32 threads):*

$$\text{BaseAddress} + \text{width} * ty + tx$$

9 Local search optimization algorithms

9.1 Principles of Local Search Algorithms

⊙ Local search (LS) methods are search techniques that have been successfully applied for solving many real and complex problems. LS methods could be viewed as “walks through neighborhoods” meaning search trajectories through the solutions domains of the problems at hand. The walks are performed by iterative procedures that allow to move from a solution to another one in the solution space.

A local search starts with a randomly generated solution. At each iteration of the algorithm, the current solution is replaced by another one selected from the set of its neighboring candidates, and so on. An evaluation function associates a fitness value to each solution indicating its suitability to the problem (selection criterion). Many strategies related to the considered LS can be applied in the selection of a move: best improvement, first improvement, random selection, etc.

Figure 1: Local search pseudo-code

```

1: Generate( $s_0$ );
2: Specific LS pre-treatment
3:  $t := 0$ ;
4: repeat
5:    $m(t) := \text{SelectMove}(s(t))$ ;
6:    $s_{t+1} := \text{ApplyMove}(m(t), s(t))$ ;
7:   Specific LS post-treatment
8:    $t := t + 1$ ;
9: until Termination_criterion( $s(t)$ )

```

⊙ The simplest LS method is the hill climbing algorithm. It starts with at a given solution. At each iteration, the heuristic replaces the current solution by the best neighbor that improves the objective function. The search stops when all candidate neighbors are worse than the current solution, meaning a local optimum is reached.

Figure 2: Hill climbing pseudo-code

```

1: Generate( $s_0$ );
2:  $t := 0$ ;
3: repeat
4:    $m(t) := \text{SelectBestMove}(s(t))$ ;
5:    $s_{t+1} := \text{ApplyMove}(m(t), s(t))$ ;
6:    $t := t + 1$ ;
7: until Local_optimum( $s(t)$ )

```

⊙ The following template gives a more refined view of the hill climbing algorithm:

Figure 3: Hill climbing details

- 1: Choose an initial solution
- 2: Evaluate the solution
- 3: **repeat**
- 4: **for** each neighbor of the current solution **do**
- 5: Complete/Incremental evaluation of the neighbor
- 6: **end for**
- 7: Replace the current solution with the best neighbor
- 8: **until** a local optimum is reached

9.2 The quadratic assignment problem

⊙ The quadratic assignment problem (QAP) arises in many applications such as facility location or data analysis. Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times n$ matrices of positive integers. Finding a solution of the QAP is equivalent to finding a permutation $\pi = (1, 2, \dots, n)$ that minimizes the objective function:

$$z(\pi) = \sum_{i=1}^n \sum_{j=1}^n a_{ij} b_{\pi(i)\pi(j)}$$

⊙ Problem instances: The considered instances are the Taillard instances proposed in the QAPLIB ¹. We will consider the instances **Taixxa** which are uniformly generated and well-known for their difficulty. This following code gives an example to read values from a **Taixxa** file and store them into arrays.

¹<http://www.seas.upenn.edu/qaplib/inst.html>

```

#include <iostream>
#include <fstream>
using namespace std;

...

void loadInstances(char* filename, int& n, int* & a, int* & b) {

    ifstream data_file;
    int i, j;
    data_file.open(filename);
    if (! data_file.is_open())
    {
        cout << "\n Error while reading the file " << filename
        << ". Please check if it exists !" << endl;
        exit (1);
    }

    data_file >> n;

    // ***** dynamic memory allocation ***** /
    a = (int*) malloc (sizeof(int)*n*n);
    b = (int*) malloc (sizeof(int)*n*n);

    // ***** read flows and distance matrices ***** /
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            data_file >> a[i*n+j];

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            data_file >> b[i*n+j];

    data_file.close();
}

...

int main(int argc, char** argv) {

    int* a;
    int* b;
    int n;

    if (argc < 2) {
        cout << "Please give a dat file" << endl;
        exit(1);
    }

    loadInstances(argv[1], n, a, b);

    ...

    free(a);
    free(b);

}

```

🔗Solution representation: Designing any iterative local search needs an encoding of a solution. The encoding plays a major role in the efficiency and effectiveness of any local search, so it constitutes an essential step in designing a heuristic. The encoding must be suitable and relevant to the tackled optimization problem. A solution of the QAP can be represented as a permutation of positive integers where the values are taken in $[0, 1, 2, \dots, n - 1]$ (with n the instance size). For example, for an array where n is equal to six, 4 2 1 0 3 5 and 3 2 0 1 5 4 represent a permutation. It is straightforward to see that the search space for this kind of optimization problems (i.e the total number of permutations) is equal to $n!$.

🔗Solution initialization: In our case, we will consider a random initialization of the initial candidate solution. The following code shows an example on how to this:

```
void create(int* solution, int n) { // create and initialize a solution

    int random, temp;
    for (int i=0; i< n; i++)
        solution[i]=i;

    // we want a random permutation so we shuffle
    for (int i = 0; i < n; i++){
        random = rand()%(n-i) + i;
        temp = solution[i];
        solution[i] = solution[random];
        solution[random] = temp;
    }
}
```

🔗Solution evaluation: To compare the different provided solutions, a score must be assigned to each one of them. For the QAP, the following code gives an example of the evaluation function:

```
int evaluation (int* a, int* b, int* solution, int n) {
    int cost=0;
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            cost += a[i*n+j] * b[solution[i]*n+solution[j]];

    return cost;
}
```

🔗Neighborhood: A neighbor is considered as a slight variation of the candidate solution which generates the neighborhood. Building a neighborhood by pair-wise exchange operations (known as the 2-exchange neighborhood) is a standard way for permutation problems. For instance, given the permutation 2 1 3 0, the associated neighborhood is the following:

- 1 2 3 0
- 3 1 2 0
- 0 1 3 2
- 2 3 1 0
- 2 0 3 1
- 2 1 0 3

For a permutation of size n , the size of the neighborhood is $\frac{n \times (n-1)}{2}$. Therefore, the code associated with the generation of the 2-exchange neighborhood for **one iteration** can be easily done:

```

for (int i = 0; i < n-1 ; i++){
    for (int j = i+1; j < n; j++){
        ...
    }
}

```

🕒 Neighbor evaluation: A more efficient way to evaluate the set of neighbors is to consider the incremental evaluation (or partial evaluation). It consists in evaluating only the move transformation applied to a solution rather than the complete evaluation of the objective function. Indeed, you have to remind that a neighbor is a slight variation from the candidate solution which generates the neighborhood. The following code gives an example of the incremental evaluation for the pair-wise exchange operator for the QAP:

```

/* i and j are the two indices to exchange (i.e. a neighbor) */

int compute_delta (int* a, int* b, int* p, int i, int j, int n) {

    int d; int k;

    d = (a[i*n+i]-a[j*n+j])*(b[p[j]*n+p[j]]-b[p[i]*n+p[i]]) +
        (a[i*n+j]-a[j*n+i])*(b[p[j]*n+p[i]]-b[p[i]*n+p[j]]);

    for (k = 0; k < n; k++)
        if (k!=i && k!=j)
            d = d + (a[k*n+i]-a[k*n+j])*(b[p[k]*n+p[j]]-b[p[k]*n+p[i]]) +
                (a[i*n+k]-a[j*n+k])*(b[p[j]*n+p[k]]-b[p[i]*n+p[k]]);

    return d;

}

/* in this case, a neighbor evaluation does no make any copy or modification
of the candidate_solution */
int incremental_evaluation (int* a, int* b, int* candidate_solution,
                           int i, int j, int n, int score) {

    return score + compute_delta(a,b,candidate_solution,i,j,n);

}

```

10 Hill climbing implementation

10.1 CPU implementation

- 🚲 Implement the hill-climbing procedure for the QAP on CPU. For doing that, implement and test the different components step by step (i.e. read the matrices values, generate a solution with a random seed, evaluate this solution, ...).

🕒 *Keep in mind that the solution replacement at each local iteration is done by the best improving neighbor (not the first improving neighbor). The following example gives an output of the hill climbing algorithm for the tai20a instance:*

```

iteration: 1 ; score = 853532
iteration: 2 ; score = 826328
iteration: 3 ; score = 806290
iteration: 4 ; score = 790374
iteration: 5 ; score = 784074
iteration: 6 ; score = 780342
iteration: 7 ; score = 763870
iteration: 8 ; score = 757198
iteration: 9 ; score = 751510

```

```

19 6 16 7 9 1 18 0 14 13 10 17 4 11 8 12 15 5 2 3

```

⊙ In comparison with the best known solutions provided in the *QAPLIB*, the quality of the solutions found by the hill climbing is rather low. Indeed, since this algorithm is really simple, the goal of this application is not to provide competitive algorithms. However, the best known solutions provided in the *QAPLIB* may give you a reference point to ensure the validity of your solutions.

⊙ As you can notice, the hill climbing convergence is pretty fast whatever the instance size. We would like therefore to perform multiple independant executions of the algorithm (also known as multi-start). The best solution returned will be the solution found by the best hill climbing.

🚲 Adapt your code to implement the multi-start for the hill climbing algorithm on CPU.

10.2 GPU implementation

⊙ Even if LS heuristics allow to significantly reduce the computational time of the solution search space exploration, this latter cost remains exorbitant when very large problem instances are to be solved. Therefore, the use of GPU-based parallel computing is required as a complementary way to speed up the search. Basically, to parallelize the multi-start local search on GPU, there are two main ways to do this:

- Generation and evaluation of the neighborhood on GPU.
- Full parallelization of the algorithm on GPU.

10.2.1 Generation and evaluation on GPU

⊙ Regarding this scheme, since the evaluation of neighboring candidates is often the most time-consuming part of local searches, it might be done in parallel. Therefore, according to the Master-Worker paradigm, this step can be executed in parallel on GPU: a kernel is associated with the generation and evaluation of the neighborhood and the CPU (host) controls the whole sequential part of the local search process. In other words, the resource-consuming part i.e. the incremental evaluation kernel is calculated by the GPU and the rest is handled by the CPU.

10.2.2 Full algorithm on GPU

⊙ A second scheme consists in parallelizing the whole algorithm on GPU. This way, it allows to reduce the data transfers between the CPU and the GPU (which may lead to a loss of performance). However, some operations such as the minimum selection (i.e. selecting the best neighbor) are not straightforward to achieve in parallel.

10.2.3 Implementations

- 🚲 Choose and implement on GPU one of the two parallelization methods described above for the multi-start local search. For doing that, you have to proceed step by step i.e. to begin with the neighborhood evaluation for one iteration, then do this for the hill climbing and finally generalize the previous concepts for the multi-start local search. For a same seed, the quality of the final solution on GPU must be identical to the one found on the CPU version.

⊛ *For the first scheme, a mapping between a CUDA thread and a neighbor might be necessary. Indeed, for instance, in a case of one-dimensional threads and neighbors described by two indexes (an exchange), finding the right mapping for each neighbor is not really straightforward. One way to handle this is to construct some mapping tables on CPU and transfer it once on the GPU device memory. Thereby, each CUDA thread can access via the global memory to the right neighbor (i.e. the two indexes).*

⊛ *In the second scheme, the main difficulty is to perform the best neighbor selection on GPU. For doing that, a parallel reduction might be done to find the neighbor which has the minimal score. This way, each threads block will contain its own best neighbor. After that, a loop inside or outside the kernel might be done to select the best neighbor with the minimal score. In any case, remember that the kernel execution is done in an asynchronous manner regarding the CPU. In other words, all the host instructions on CPU will be immediately performed right after a kernel even if the GPU execution is not finished. In some cases (e.g. two successive kernels), such asynchronous behaviour might not be desirable. As a consequence, instructions such as `cudaMemcpy()` or `cudaDeviceSynchronize()` allows to avoid such situations by synchronizing both the CPU and the GPU.*

- 🚲 Optimize your version by using some memory management (e.g. the use of texture memory).
- 🚲 Provide a script which runs your program by displaying only the CPU and GPU execution times for the different Taixxa instances (from tai20a to tai100a). The number of global iterations (i.e. the number of hill climbing) in the multi-start local search must be set to 10000.