PROGRAMACIÓN PARALELA
CUDA Tutorial

In the accompanying archive, there is a compressed directory called `Lab07`. Once the archive is uncompressed change to that directory. Inside it there are three directories: `00-CheckCapabilities`; `01-HelloWorld`; `02-VectorAdd`.

The `00-CheckCapabilities` directory has a program called `checkDeviceInfo.cu`. Its purpose is to display information on the first CUDA device in this system, including driver version, runtime version, compute capability, clock rate, amount of global memory, cache size, etc. There is also a directory (`bandwidthTest`). Inside it there is a nvidia-provided simple test program (`bandwidthTest.cu`) that allows us to measure the memcopy bandwidth of the GPU and memcpy bandwidth across PCI-e.

The `01-HelloWorld` directory has three different versions of the same program: `hello0.cu`; `hello1.cu`; `hello2.cu`. The first program (`hello0.cu`) uses just the CPU to print a message. The second program (`hello1.cu`) launches a kernel with one block of five threads. The last program (`hello2.cu`) launches a kernel with two blocks; each block has five threads.

The `02-VectorAdd` directory has several programs.

The first to try is `vectorAddition.c`; it adds two vectors (`augend`, `addend`) of length `VEC_LEN`. The resulting vector is stored in the `result` array. The memory required to store the vectors is dynamically allocated. Since we use a sequential approach, the addition is done via a `for` loop inside the `add` function. Before finishing the execution, the allocated memory is returned to the memory heap.

The second program is `integerAddition.cu`. It illustrates the general layout of a CUDA program. The CUDA device just adds two integers using the `add` kernel.

The third program (`vectorAddition_blocks.cu`) adds two vectors of length `VEC_LEN`. Since we use `VEC_LEN` blocks and one single thread per block on the CUDA device for that purpose (i.e., block-level parallelisation), the `add` function (CUDA kernel) just needs to indicate the operation that will execute each single thread on every block by using `blockIdx.x` as index. In principle, that is the way for writing kernels; they must be written to do the work of a single iteration of the loop.

The fourth program (`vectorAddition_threads.cu`) adds two vectors of length `VEC_LEN`. Since we use a single block with `VEC_LEN` threads on the CUDA device for that purpose (i.e., thread-level parallelisation), the `add` function (CUDA kernel) just needs to indicate the operation that will execute each single thread by using `threadIdx.x` as index. In principle, that is the way for writing kernels; they must be written to do the work of a single iteration of the loop. We may have up to 1024 threads in a block, so this approach is better than the previous one.

The fifth program (`vectorAddition_blocks_threads.cu`) adds two vectors of length `VEC_LEN`. Now on the CUDA device we use BLOCKS blocks and each block has THREADS_PER_BLOCK threads inside each block (i.e., multiple blocks of threads parallelisation). The `add` function (CUDA kernel) thus needs to calculate the index by means of the `blockIdx.x`, `blockDim.x`, and `threadIdx.x` indices. In principle, that is the way for writing kernels; they must be written to do the work of a single iteration of the loop. Now we are exploiting the GPU resources in a better manner due to using a potentially massive number of threads that can scale up better than the two previous approaches.