GPUs and CUDA programming

In this section, we give a very short introduction of the GPU programming model that we used to parallelise the WPM algorithm.

GPUs (Graphics Processing Units) are hardware acceleration devices built with massively parallel architectures specialised in "Single Instruction Multiple Data" (SIMD) type of tasks. Originally designed for graphics and display purposes, it is possible to use them for general purpose processing using CUDA[1].

GPUs are used to speed up specific parts of a program that present opportunities for a parallel execution. Therefore, a CUDA program is always executed on a CPU (called "host"), which will interact with the GPU (called "device") by calls to CUDA functions and written device functions (which are called "kernels"). The host and the device have separate memory and address spaces, which means that generally the host can't access device memory and vice versa[2]. Consequently the data on which the device works must always be copied from the host to the device before execution. This is done using built-in CUDA functions.

To write a working and efficient kernel (the function that will be executed on the device), one must understand the architecture of a GPU and the abstractions that are the thread[3] and memory hierarchies, illustrated figure 1.

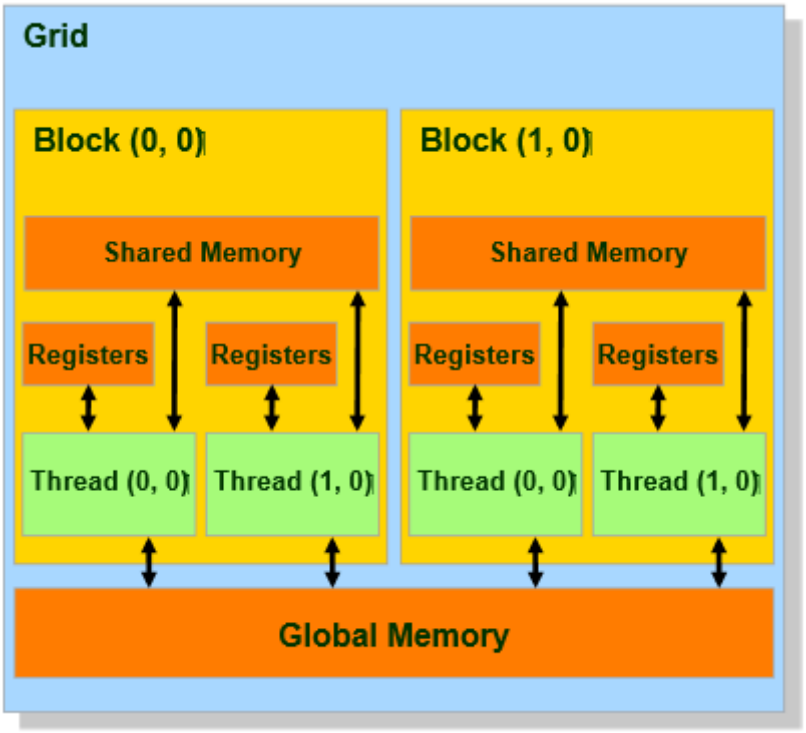

Figure 1: GPU memory and thread hierarchy - source: courses from H.Fröning

Basically, multiple threads form a thread block in which the threads can be synchronised and communicate inside the kernel. Together those blocks form a grid, however two threads that aren't in the same blocks cannot be synchronised or communicate together. The grid and the thread blocks can have up to three dimensions that must be specified at every kernel call, and cannot exceed the available resources on the GPU. In the kernel, it is possible for the threads to access the block dimensions and their respective block number and thread number. This allows all the threads to execute the same instruction on different entries of an input array. It is also be possible to assign different tasks to the threads this way with an `if` condition on their block and thread numbers, however we should keep in mind that GPUs are designed

[1] `http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html`
[2] it is possible to save some memory in a common space but it must be used carefully as its size is restricted
[3] `https://en.wikipedia.org/wiki/Thread_(computing)`

to work best when every thread executes the same instruction, branch divergence can result in significant performance loss.

With the thread hierarchy comes a corresponding memory hierarchy. Each thread has registers, its own local address space, from which loads are the cheapest. Within a thread block there is shared memory, that is accessible by all threads within the block, although it is a bit slower than the local memory. Note that to use shared memory, it must be initialised by the threads inside the kernel. Finally, there is global memory, accessible by every thread, and to which the data from the host is generally copied. However loads and stores from and to global memory are the most expensive. On top of all this, there is also a two level cache system : the L1 cache at the shared memory level and the L2 cache at the global memory level. Optimising the memory accesses to make the most out of the caches or take advantage of data re-use among the threads using shared memory can result in very significant performance improvements.