GPU Architecture: the Fermi's example

Each GPU device is a scalable processor array consisting of a set of SIMT (Single Instruction Multiple Threads) Streaming Multiprocessors (SM), each containing several Stream Processors (SPs) (see Figure 1(a)). The GPU has a global scheduler (Giga Thread) for distributing the work to the SMs and a host interface. Different memory spaces are also available within a GPU, having different latencies, storage capacity and access methods. These memory spaces, ordered from low to high latency are: the register file (32768 32-bit registers per SM in NVIDIA compute capability devices 2.X), the shared memory/L1 cache (64 KB per SM), the L2 cache (768 KB) and the global memory (DRAM, 1 - 6 GB). The CUDA programming model utilizes this architecture and is based on a hierarchy of abstraction layers (see Figure 1(b)). The thread is the basic execution unit that is mapped to a single SP. A thread-block or simply block is a batch of threads assigned to the same SM, and therefore share all the resources included in that multiprocessor, such as the register file and shared memory.
The threads within a block can communicate through the shared memory. Finally, a grid is composed of several blocks which are equally distributed and scheduled across all SMs in a nondeterministic manner.
Threads included within a block are divided into batches of 32 threads called warps. The warp is the scheduled unit, so the threads of the same block are executed in a given multiprocessor warp-by-warp in a SIMD (single instruction, multiple data) fashion. The programmer arranges parallelism by declaring the number of blocks and the number of threads per block to use in a specific kernel. To avoid wasting SP resources, the number of threads per block should be a multiple of 32 (i.e. a warp). The maximum number of threads per block since NVIDIA 2.0 compute capability is 1024.

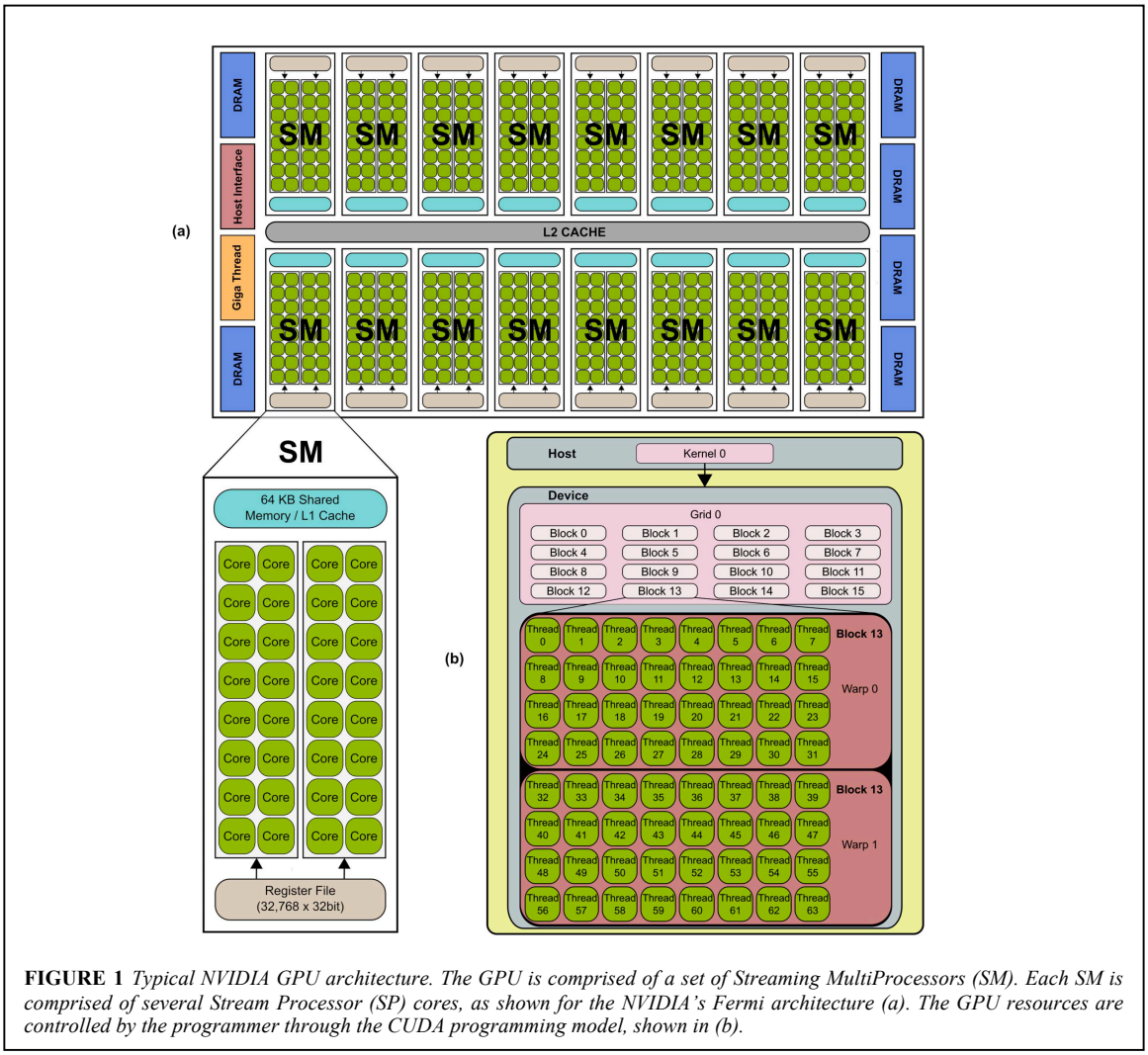

FIGURE 1 *Typical NVIDIA GPU architecture. The GPU is comprised of a set of Streaming MultiProcessors (SM). Each SM is comprised of several Stream Processor (SP) cores, as shown for the NVIDIA's Fermi architecture (a). The GPU resources are controlled by the programmer through the CUDA programming model, shown in (b).*

Non-Gaussian diffusion model based DW-MRI reconstruction by GPU

We propose to implement a parallel GPU-based design of non-Gaussian diffusion methods used for the analysis of brain magnetic resonance imaging (MRI): the Diffusional Kurtosis Imaging (DKI) and the Stretched Exponential Model (SEM). More specifically, we are concerned with a model-based approach for extracting tissue structural information from diffusion-weighted (DW) MRI data. Specifically, a complex non-linear relation describing the diffusion-weighted MRI (DW-MRI) signal attenuation has to be fitted to experimental dataset voxel-by-voxel by using the Levenberg-Marquardt algorithm.

Certain features of the proposed implementation make it a good candidate for a GPU-based design. These can be summarised into the following: a) Independence between voxels across the three-dimensional brain volume allows voxel-based parallelisation, b) Within each voxel, certain computation steps of data analysis are intrinsically iterative and independent, allowing further parallelization (i.e. fitting parameters estimation), c) Relatively simple mathematical operations are needed and these can be handled effectively by the GPU instruction set and d) Memory requirements are moderate during each step of the algorithm.

The Levenberg-Marquardt algorithm is based on an iterative numerical optimization procedure that minimizes the sum of squared model residuals. The CUDA kernel performs the Levenberg-Marquardt algorithm. This kernel maps each CUDA thread to a voxel (see Figure 2), and it launches as many threads as voxels contained in a particular slice. Because processing of different voxels is totally independent, the threads do not need to synchronize. It is noteworthy that each thread must compute all steps of the Levenberg-Marquardt algorithm using large intermediate structures (the size of the structures depends on the number of parameters to fit and the number of diffusion-sensitising gradient directions K of the input dataset). That involves managing many hardware resources and on-chip memories, specifically the registers, which are limited to a maximum number of 64 per thread, at least up to NVIDIA's Fermi. The more recent Kepler architecture supports up to 255 registers per thread and will potentially improve performance. To achieve a high occupancy of the GPU hardware, while also accounting for the fact that different slices in the brain may have very different number of voxels of interest, we optimised the number of threads per block Q (which needs to be a multiple of 32 to avoid wasting resources with under-populated warps). The target is to have as many threads as possible per SM (organised in warps) to ''hide'' latencies that may be induced by Global memory access (while a warp is accessing Global memory, the SM can process another warp). The available number of registers per SM is 32768. If we use the maximum number of registers per thread (minimizing that way the number of Global memory accesses), the maximum number of threads running per SM simultaneously is 512 (32768 registers/64 registers per thread = 512 threads per SM or 16 warps per SM). Choosing how to distribute these threads in blocks of size Q affects performance.
In general, smaller Q provides greater flexibility to the scheduler to distribute threads better. For instance, let's imagine the case where 2 SMs are free and there are only 128 voxels (threads) to be processed. If we choose Q~128 (4 warps) we will use only one of the two SMs (all threads belong to the same block and therefore must be executed by the same SM). But if Q~64 (2 warps) we can use both SMs (2 warps in each SM), therefore achieving greater parallelisation. The minimum number for Q is 32 (to avoid underpopulated warps). However, a limitation imposed in NVIDIA 2.X compute capability devices is that any SM can only handle 8 different blocks simultaneously. Therefore, if we set Q~32 the maximum number of warps simultaneously in a SM are 8 (32 threads * 8 blocks/32 threads of a warp), i.e. half of the maximum warps an SM can handle in our case. Therefore, we set the number of threads to the next minimum number Q~64 to get the best balance of threads between the different SMs and the best performance for this algorithm in our application.

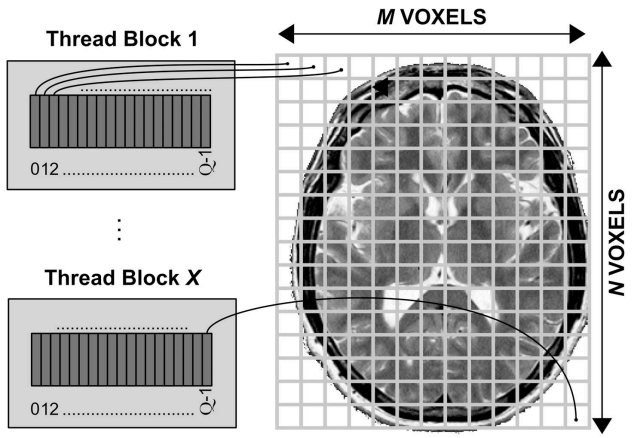

FIGURE 2 *Distribution of resources for the CUDA kernel that performs the Levenberg-Marquardt algorithm. Voxels are assigned to threads of CUDA blocks. Each CUDA block is comprised of Q threads and processes Q voxels (Q~64 was used in this study).*

Steps to achieve our goals

(i) Fast-implementation: using of available Matlab custom script.

1. Implementation of a parallelized version of the Levenberg-Marquadt algorithm for usage on GPU by means of the Matlab Parallel Toolbox;
2. Optimization of available Matlab custom script for DKI and SEM analysis for usage on GPU by means of the Matlab Parallel Toolbox.

(ii) Slow-implementation: translating available Matlab custom script in CUDA language.

1. Implementation of a parallelized version of the Levenberg-Marquadt algorithm for usage on GPU by means of CUDA platform;
2. Translation of available Matlab custom script for DKI and SEM analysis for usage on GPU by CUDA platform.

(iii) Implementation of numerical simulations supporting experimental data analysis: Monte Carlo and Finite Elements Method (FEM) on GPU

1. Implementation of a parallelized version of available Monte Carlo and FEM algorithms to numerically simulate DW-MRI signal attenuation in complex restricting/hindering geometries for usage on GPU by means of the Matlab Parallel Toolbox;
2. Translation of parallelized version of Monte Carlo and FEM algorithms to numerically simulate DW-MRI signal attenuation for usage on GPU by CUDA platform.

Points *1s* of steps *(i)* and *(ii)* will be organized and developed as described in "Non-Gaussian diffusion model based DW-MRI reconstruction by GPU" section. In particular, for point *1* of step *(i)* it is necessary the Matlab Parallel Toolbox on GAP01 server; point *1* of step *(ii)* will require some months to be developed (more than 3 months) if some source codes are not available and freeware online. Some suggestions about this?

Point *2* of steps *(i)* may require some weeks (less than 4 weeks), while point *2* of step *(ii)* may require some months (more than 3 months) because it will be necessary some practice with CUDA platform and the GPU architecture currently mounted on GAP01 server. Maybe Matteo Bauce's expertise about this point could be a precious help for us.

Point *1* of step *(iii)* is partially completed, as you were able to read in our last report about NMR Group (Rome) activity. Point *2* of step *(iii)* will require some months (more than 1 month).