Comparative study of methods of Principal Component Analysis of automatic segmentation of functional magnetic resonance imaging (fMRI).

1. Introduction

1.1 Motivation

In the beginning of the last decade an increase of CPU (Central Processing Unit) clock speed was generally stopped. The main reason for that is because of the thermal losses. In order to maintain continuous increase of the performance, nowadays processors comprise many cores (multicores processor). This implies that a paradigm of sequentially written programs has become unable to fully utilize this architecture. To achieve that it is necesarry to develop parallel applications i.e. applications which exploit all available cores efficiently.

In practice there are two main approaches to develop parallel applications. The first one is about processors containing several cores (2,4,6,8,…), each one (processor) processing several „heavy” threads. Another type of processors are those which contain many cores (hundreds, thousands) being able to process many „light” threads. This is how GPU (Graphic Processor Unit) works. Nowadays numerical applications with big computational complexity are implemented mainly on GPUs. They are specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control [1]. A low price and availability are others advantages of GPUs.

1.2 Objectives

The aim of this work was to implement a parallel version of PCA-based methods of segmentation of functional magnetic resonance imaging (fMRI) on CUDA (Compute Unified Device Architecture) platform in order to obtain better performance (speed-up) regarding to Matlab method’s version. The study comprises a complete documentation of the code of the implemented algorithm in CUDA C, explaining some tricks characteristc of CUDA, and other possible solutions. Later a comparison of the execution time of the methods in Matlab and CUDA are shown.

* 1. Explanation of CUDA platform and differences between CPU

There are some important differences between GPU and CPU architecture to consider when optimizing code. CPU cores are designed to execute instructions sequentially, so they are optimized for flow control. They have bigger cache than GPUs to minimize the memory access latency (memory bandwitdh in CPUs is generally low).

On the other hand GPUs architecture was optimized for computer games, so they contain many simple floating-point ALU executing in groups millions of instructions. The flow control is simplified. Many „light” threads are executed simultaneously, so that the memory access latency can be hidden with calculations instead of big data caches. The essential advantages of CUDA technology over CPU processors are memory bandwidth (Byte/s) and computional throughput (FLoating point Operations Per Second).

CUDA platform consists of a host (CPU) and one or more devices (NVIDIA GPU) under host’s control. The NVIDIA GPU architecture is built around a scalable array of multithreaded Streaming Processors (SMs). A parallel application is divided into blocks of threads which are executed independently from each other. Every block is executed by one Streaming Processor, so that threads can communicate each other around the block they belong to. The partition into blocks of threads makes it possible for a scheduler to transparently scale application’s parallelism when it is run on a GPUs with a bigger number of multiprocessors.

NVIDIA developers named CUDA parallelism model „SIMT” (Single Instruction, Multiple Threads) which is similar to SIMD model (Single Instruction, Multiple Data). In fact threads grouped in a warp (a group of 32 threads around one block) work as SIMD models describes, but at the same time other threads from another block can execute another instruction from the same kernel program.

One of the drawback of CUDA is memory transfer between a host and a device. It is higly recommended then to minimize this, and in order to obtain speed-up a program must have enough work to do to cover memory transfer time cost.

1. References

[1] CUDA C Programming Guide, <http://docs.nvidia.com/cuda/cuda-c-programming-guide>, September, 1, 2015