35th Annual International Conference of the IEEE EMBS
Osaka, Japan, 3 - 7 July, 2013

GPU-Based acceleration of an automatic white matter segmentation algorithm using CUDA

Nicole Labra [1,2], Miguel Figueroa[1,2], Pamela Guevara[1]
Delphine Duclap [3], Josselin Hoeunou [3,4], Cyril Poupon [3] and Jean-François Mangin [3]

[1]Dept. of Electrical Engineering, Universidad de Concepción, Concepción, Chile
[2] Center for Optics & Photonics, Universidad de Concepción, Concepción, Chile
[3] I2BM, Neurospin, CEA, Gif-sur-Yvette, France
[4] INSERM, U955 Unit, Paris, France

Abstract— **This paper presents a parallel implementation of an algorithm for automatic segmentation of white matter fibers from tractography data. We execute the algorithm in parallel using a high-end video card with a Graphics Processing Unit (GPU) as a computation accelerator, using the CUDA language. By exploiting the parallelism and the properties of the memory hierarchy available on the GPU, we obtain a speedup in execution time of 33.6 with respect to an optimized sequential version of the algorithm written in C, and of 240 with respect to the original Python/C++ implementation. The execution time is reduced from more than two hours to only 35 seconds for a subject dataset of 800,000 fibers, thus enabling applications that use interactive segmentation and visualization of small to medium-sized tractography datasets.**

I. INTRODUCTION

The study of brain connectivity is an attractive research area for neuroscience and clinical studies. Nowadays, recent diffusion Magnetic Resonance Imaging (dMRI) techniques with high angular resolution (HARDI) combined with several accurate pre- and post-processing methods, have largely improved the quality of tractography. Resulting tractography datasets are highly accurate and contain valuable information for a better description and analysis of the human connections, but their high complexity and huge size pose a requirement for a new generation of analysis methods. Some methods for automatic segmentation of white matter (WM) bundles automatically place a set of regions of interest (ROI), defined on a brain atlas in a standard space [10]. These regions are then warped to each subject using linear or non-linear registration. White matter bundles are then extracted in function of the connected and traversed regions. In [3], the authors proposed an automatic WM segmentation method based on a multi-subject WM bundle atlas. The original method performed first intra-subject clustering [4] in order to reduce the size of the fiber dataset and then used a similarity measure for classifying the fibers and extract the known bundles. This method was compared with an accurate ROI-based automatic method [10] and was found to produce good results because fiber shape, length and position information, as well inter-subject variability, are embedded in the atlas. It was proposed for huge tractography datasets, with millions of fibers, even though it is applicable to smaller datasets.

Cluster-based methods such as the one described above can deal with datasets with a huge number of fibers, but suffer from very long processing times. For small and medium-size tractography datasets (50,000 to 2,000,000 fibers), we can use faster and more direct algorithms. Moreover, exploiting the large-scale parallelism made available by new Graphics Processing Unit (GPU) hardware, we can devise applications that perform interactive segmentation and visualization of white matter tracts. GPUs can exploit memory hierarchy and fine-grained parallelism available in hardware to achieve high performance at an affordable cost [5], [8]. For example, [6] proposes a real-time interactive fiber tracker from user-defined volume of interests. In [9], the authors implement a GPU-based Bayesian framework for probabilistic brain fiber tractography, significantly reducing the processing time. Other works explore the graphical capabilities of GPUs for the exploration of Diffusion Tensor Imaging (DTI) Fibers or the improvement of fiber rendering. Closer to our research, [7] proposes an accelerated fiber clustering/segmentation method. The authors implemented a fiber pairwise similarity measure using several GPU architectures. They obtained very good speedup with respect to a sequential processing, but they give no details about the clustering/segmentation method processing times nor the tractography dataset sizes. Nevertheless, the results show that GPUs can significantly reduce the computation times of these algorithms.

In this work we focus on the optimization of the automatic WM segmentation algorithm proposed in [3], applied to small and medium size tractography datasets. First, we modified the original algorithm written in Python/C++ to avoid the first intra-subject clustering and work with a whole-brain tractography dataset. Next, we profiled the code to identify critical sections and rewrote them to run in parallel on a GPU using the Compute Unified Device Architecture (CUDA) language. Finally, we used actual tractography data to evaluate the performance of our parallelized code.

II. MATERIAL AND METHODS

A. *Diffusion and Tractography Datasets*

For this analysis, we used healthy subjects from a High Angular Resolution Diffusion Imaging (HARDI) database. Acquisitions were obtained using a MRI Siemens Magnetom TrioTim 3T, 12-channel head-coil. The protocol

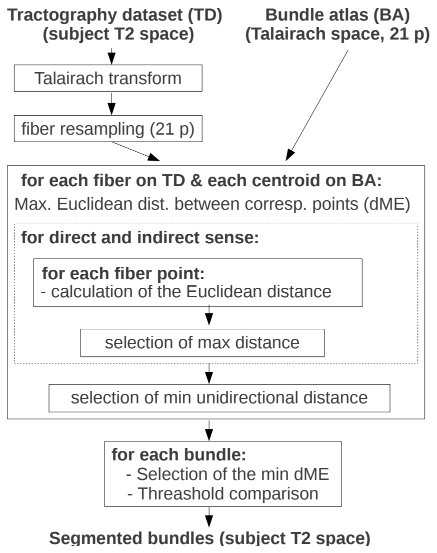
Tractography dataset (TD) (subject T2 space)
Bundle atlas (BA) (Talairach space, 21 p)
Talairach transform
fiber resampling (21 p)
for each fiber on TD & each centroid on BA:
Max. Euclidean dist. between corresp. points (dME)
for direct and indirect sense:
for each fiber point:
- calculation of the Euclidean distance
selection of max distance
selection of min unidirectional distance
for each bundle:
- Selection of the min dME
- Threashold comparison
Segmented bundles (subject T2 space)

Fig. 1. Algorithm for segmentation of white matter bundles

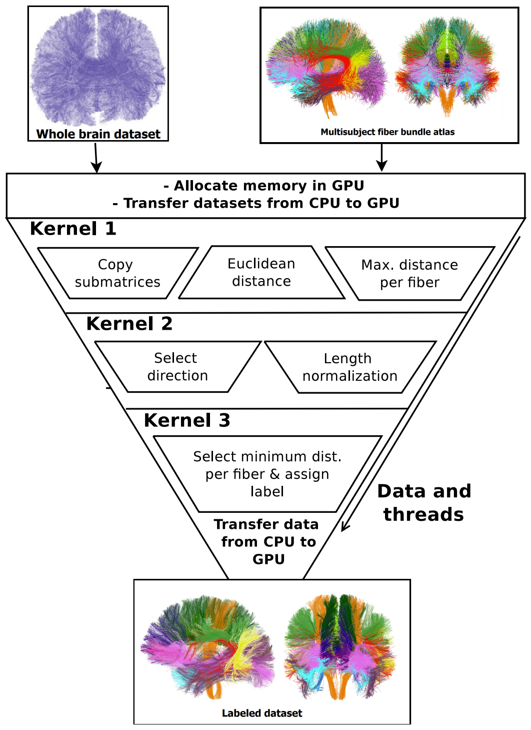

Fig. 2. General schematics of the algorithm implementationon for a GPU

included a high-resolution T1-weighted acquisition (echo time = 2.98ms, repetition time=2300ms, 160 sagittal slices, 1.0x1.0x1.1mm) and a DW-EPI sequence along 41 directions (2.0x2.0x2.0mm, b=1000s/mm^2, plus one b=0 image, echo time 87ms, repetition time 14000ms, 60 axial slices).

B. Preprocessing

The data were processed using BrainVISA/Connectomist-2.0 software [2]. They were preliminary corrected for all the sources of artifacts and outliers were also removed. Then, the analytical Q-ball model [1] was computed to obtain ODF fields in each voxel. A streamline deterministic tractography was performed on the entire T1-based brain mask using a forward step of 0.5mm. This lead to a dataset with an average of one million of fibers, that was then subsampled in order to get smaller testing datasets.

C. White matter bundle segmentation

Fig. 1 shows our automatic WM segmentation algorithm [3], written in a combination of Python and C++, and without the application of the intra-subject clustering. First, the algorithm was adapted for receiving a new dataset from the whole brain (not divided into left/right hemispheres and inter-hemispheric bundles). The algorithm normalizes the new tractography dataset to the bundle atlas space (Talairach space). Then, each fiber is resampled using 21 equally-spaced points. Next, a distance metric is computed between each fiber of the tractography dataset and each centroid of the multi-subject atlas, with a total of 9,085 centroids for left/right hemispheres and inter-hemispheric bundles. The distance metric used is the maximum of the Euclidean distances between corresponding points [3]. Finally, each individual fiber is labeled by the closest atlas bundle, provided that the distance to this bundle, namely the smallest pairwise distance to the centroids representing this bundle, is lower than a given threshold for each bundle.

D. Algorithm optimization

We started the optimization of the algorithm by rewriting the original implementation entirely in C, using dynamic RAM assignment and local registers in the classification algorithm. We restructured the code, removing redundant sections and computations that do not significantly affect the classification results. We compiled the code using gcc with third-level optimizations for shortest execution time on a single CPU core. We used a profiler (gprof) to analyze the C code and identify computationally-intensive blocks, which became candidates for a parallel implementation.

From the profiling analysis it became clear that the execution time is largely dominated by the computation of the Euclidean distance between the sampling points in each subject fiber and the corresponding point in each centroid of the atlas, which is $O(NM)$, where N is the number of fibers in the test subject and M is the number of total centroids in the atlas. Other computationally-intensive parts of the algorithm include computing the distance metric (selecting the maximum distance among the points for each subject fiber), recomputing the previous distances for the inverse direction of the subject fiber and picking the minimum metric, normalizing the distance metric by the length of the fiber, and selecting the closest centroid to each fiber and labeling the fiber with its bundle if the distance is smaller than the bundle's threshold.

We then wrote a parallel version of the algorithm for an nVidia GPU using the CUDA language. Fig. 2 depicts the architecture of the program. We wrote three kernels that execute on the GPU and concentrate most of the execution

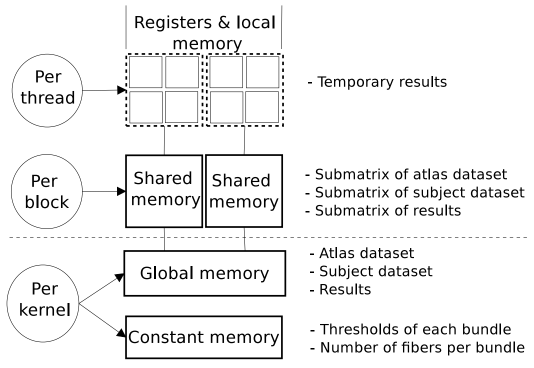

Fig. 3. Distribution of data in GPU memory

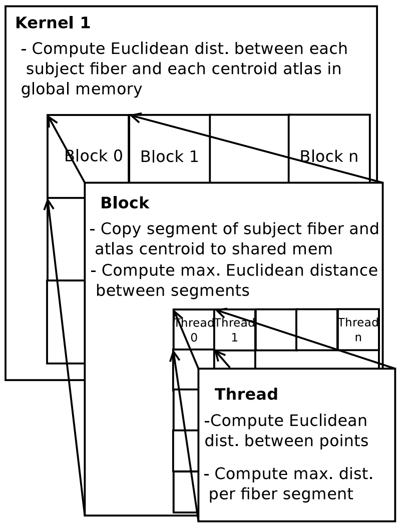
Kernel 1
- Compute Euclidean dist. between each subject fiber and each centroid atlas in global memory
Block 0
Block 1
Block n
Block
- Copy segment of subject fiber and atlas centroid to shared mem
- Compute max. Euclidean distance between segments
Thread 0
Thread 1
Thread n
Thread
-Compute Euclidean dist. between points
- Compute max. dist. per fiber segment

Fig. 4. Distribution of computation in Kernel 1

time. The program first reserves global memory on the video RAM, leaving some extra memory for temporary data and final results. The CPU then transfers the entire atlas dataset to the global memory. Because the subject dataset is too large to fit in the global memory (between 12,500 and 2,000,000 fibers), the CPU partitions the dataset and transfers it to the memory in segments of up to 80,000 fibers. Because the atlas contains only 9,085 centroids, the whole set fits in the global memory. The CPU then iteratively invokes Kernel 1 on the GPU for each segment of data that it transfers.

Fig. 3 shows how we distribute the data in the GPU memory hierarchy in order to optimize the operation of the algorithm. There are three levels of hierarchy: registers and local memory that can be privately accessed by each thread executing on the GPU, shared memory which is accessed by all threads belonging to the same thread block, and global memory available to all threads on the GPU. The constant memory is also globally accessible, but it is of small size (only 64KB), cannot be modified within the program, has very low latency and reduces the bandwidth requirements when multiple threads need to access the same memory location simultaneously. We use the constant memory to store parameters such as the threshold values, and the number of centroids for each bundle in the atlas. On the other hand, the global memory is more flexible and larger (6GB in our video card), but it has a much higher latency and its bandwidth is more limited. We use the global memory as an intermediate stage to transfer data between the CPU memory and the GPU, in particular the values of the atlas fiber datasets, as well as the computation results.

Because the global memory is slow, the GPU first transfers the data to internal shared memory. The CUDA architecture partitions the computation of a kernel in thread blocks, which share access to local memory. These shared memories are much faster than global RAM, and the data stored in them can be accessed and modified by multiples threads in parallel, thus boosting the performance of the program. The data exists in the memory for the duration of the thread block (the lifetime of the kernel). Finally, each thread can access local memory and registers used to store temporary results that require quick access from within the same thread. We use registers to store partial results in the computation of a single Euclidean distance.

As Fig. 2 shows, the program runs three kernels on the GPU. Kernel 1 exploits the parallelism available in the computation of the Euclidean distance between corresponding points for each pair (subject fiber, atlas centroid). This kernel represents the largest computational load of the algorithm, and stores the data in bidimensional matrices of width 21 (the number of sampling points for each fiber) and length NM, where N is the length of the segment of the subject dataset transferred to the global memory, and M is the size of the atlas dataset (9,085). Fig. 4 shows how we distribute the computation of Kernel 1 in the GPU. The kernel executes three main procedures. The first is to copy the input data from global to shared memory, and then compute the Euclidean distance. In order to compute the distance, each thread within a block accesses one 3D point in space (coordinates x, y, and z) of the subject fiber and atlas centroid. The computation is performed with the subject fiber in the order it is stored in the dataset, and also inverting the order of the sampling points. The distance between the fiber and the centroid is computed as the maximum value of the distance between corresponding points. The GPU also computes the maximum value in parallel, using a tree reduction on the locally-computed distances. The maximum is computed first within a block in shared memory and then among all blocks in global memory. Because several threads compete for access to the memory to store their results, we use an atomic maximum function available in CUDA to avoid memory conflicts.

Kernel 2 in Fig. 2 uses considerably less data and computation than Kernel 1, and therefore we use mainly global memory and some registers for temporary results. The first step compares the distance metric computed for the direct and inverse versions of each subject fiber, and chooses the minimum as the distance between that fiber and centroid. The second step normalizes this distance by adding to it a correction value that approaches zero when the fiber and the centroid have similar length and increases with the

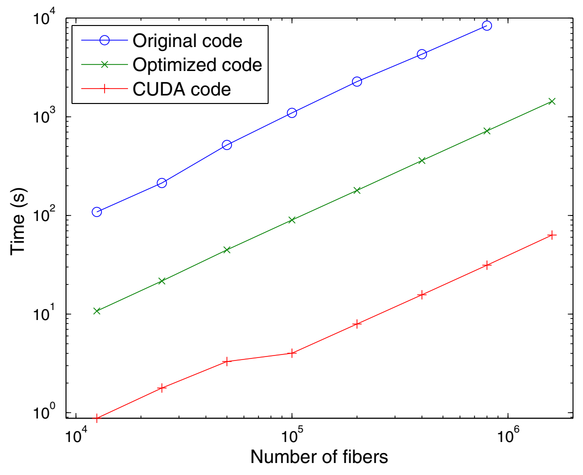

Fig. 5. Execution time of the three implementations (log-log)

difference in their lengths. The correction value requires the computation of the Euclidean distance between the first two points of each fiber and centroid.

Finally, Kernel 3 also works directly on global memory. The kernel performs the final step in the classification: First, it uses a tree reduction to compute in parallel the minimum distance between each subject fiber and each fiber in the atlas. For each bundle, it compares this minimum distance with the bundle threshold stored in constant memory. If the value is smaller than the threshold, the fiber is labeled with the name of the bundle. Finally, the results are transferred back to the CPU memory.

III. RESULTS

We implemented the algorithm described above on an nVidia Quadro 6000 video board with 6GB of GDDR5 global memory and a Fermi GPU with 448 CUDA cores and a 574MHz core clock. The host that runs the sequential C implementation of the algorithm uses an Intel i7-3820 CPU with a 3.6GHz clock and 8GB of DDR3 RAM.

We tested our implementations of the algorithm using an atlas containing 9,085 centroids (fibers), and varying the size of the dataset (number of fibers of the subject on which we perform segmentation) between 12,500 and 1,600,000 subject fibers. Fig. 5 shows the execution time of the algorithm in each of the implementations (original Python/C++, optimized sequential C, and GPU), which includes both data processing and file I/O, albeit the execution time is vastly dominated by data processing. To make it easier to visualize the remakably different execution times, we plotted the graph on a log-log scale. In all cases, the execution time is linear with respect to the size of the dataset. The execution time of the Python/C++ implementation varies between 108 and 8,403 seconds for 800,000 fibers. The Python/C++ program supports only up to one million fibers because the fiber number was first reduced by clustering.

The optimized sequential C implementation reduces the execution time to between 10.7 and 719.8 seconds (1,435 seconds for 1.6 million fibers), which represents a speedup between 10.1 and 12.7 with respect to the Python/C++ program (speedup is defined as the ratio between the execution time of an unoptimized program and an optimized version). Finally, the execution time of the CUDA implementation is between 0.9 and 31.4 seconds for up to 800,000 fibers (63.3 seconds for 1.6 million fibers), with a speedup between 12.2 and 23 compared to the optimized C version, and between 119.8 and 286.7 compared to the Python/C++ version.

IV. DISCUSSION AND CONCLUSION

We have presented a GPU-based parallel implementation of an automatic segmentation algorithm for classification of white matter fibers based on a labeled atlas. Our work shows how placing special care in the distribution of the computation and exploitation of the memory hierarchy available in the GPU can lead to important reductions in execution time, going from more than two hours in the original implementation to only 31 seconds for a dataset of 800,000 fibers. Such a speedup opens the road for interactive analysis of small and medium-sized tractography datasets using affordable hardware. The acceleration of the proposed implementation with more optimized algorithms, like a multi-scale approach for distance calculation, enables us to envision interactive segmentation and visualization of white matter tracts which can be used for neuroscience and clinical studies.

V. ACKNOLEDGMENTS

This work was partially funded by the Chilean government via Fondecyt grants 112101 and 11121644, and PIA-CONICYT grant PFB 0824. The authors would like to thank Marion Leboyer for providing the testing HARDI database.

REFERENCES

[1] M. Descoteaux, E. Angelino, S. Fitzgibbons, and R. Deriche. Regularized, fast and robust analytical q-ball imaging. *Magn. Reson. Med.*, 58:497–510, 2007.

[2] D. Duclap, A. Lebois, B. Schmitt, O. Riff, P. Guevara, L. Marrakchi-Kacem, V. Brion, F. Poupon, J.-F-Mangin, and C. Poupon. Connectomist-2.0: a novel diffusion analysis toolbox for BrainVISA. In *ESMRMB 2012*, 2012.

[3] P. Guevara, D. Duclap, C. Poupon, L. Marrakchi-Kacem, P. Fillard, D. Lebihan, M. Leboyer, J. Houenou, and J-F. Mangin. Automatic fiber bundle segmentation in massive tractography datasets using a multi-subject bundle atlas. *Neuroimage*, 61(4):1083–1099, Jul 2012.

[4] P. Guevara, C. Poupon, D. Riviére, Y. Cointepas, M. Descoteaux, B. Thirion, and J.-F. Mangin. Robust clustering of massive tractography datasets. *NeuroImage*, 54(3):1975–1993, Feb 2011.

[5] D. B. Kirk and W.-M. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. M. Kaufmann, 2010.

[6] A. Mittmann, T. H. C. Nobrega, E. Comunello, J. P. O. Pinto, P. R. Dellani, P. Stoeter, and A. von Wangenheim. Performing real-time interactive fiber tracking. *J Digit Imaging*, 24(2):339–351, Apr 2011.

[7] C. Ros, R. Tandetzky, D. Güllmar, and J.R. Reichenbach. GPGPU computing for the cluster analysis of fiber tracts: Replacing a $15000 high end PC with a $500 graphics card. In *ISMRM 2011*, 2011.

[8] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2012.

[9] M. Xu, X. Zhang, Y. Wang, L. Ren, Z. Wen, Y. Xu, G. Gong, N.i Xu, and H. Yang. Probabilistic brain fiber tractography on gpus. In *IPDPSW 2012*, pages 742–751, Washington, DC, USA, 2012. IEEE Computer Society.

[10] Y. Zhang, J. Zhang, K. Oishi, and et al. Atlas-guided tract reconstruction for automated and comprehensive examination of the white matter anatomy. *NeuroImage*, 52(4):1289 – 1301, 2010.