fMRI Analysis on the GPU - Possibilities and Challenges

Anders Eklund, Mats Andersson and Hans Knutsson

Original Publication:

Anders Eklund, Mats Andersson and Hans Knutsson, fMRI Analysis on the GPU - Possibilities and Challenges, 2012, Computer Methods and Programs in Biomedicine, (105), 2, 145-161.
http://dx.doi.org/10.1016/j.cmpb.2011.07.007

fMRI Analysis on the GPU
-
Possibilities and Challenges

Anders Eklund[a,b,*], Mats Andersson[a,b], Hans Knutsson[a,b]

[a] *Division of Medical Informatics, Department of Biomedical Engineering*
[b] *Center for Medical Image Science and Visualization (CMIV)*
Linköping University, Sweden

Abstract

Functional magnetic resonance imaging (fMRI) makes it possible to non-invasively measure brain activity with high spatial resolution. There are however a number of issues that have to be addressed. One is the large amount of spatio-temporal data that needs to be processed. In addition to the statistical analysis itself, several preprocessing steps, such as slice timing correction and motion compensation, are normally applied. The high computational power of modern graphic cards has already successfully been used for MRI and fMRI. Going beyond the first published demonstration of GPU-based analysis of fMRI data, all the preprocessing steps and two statistical approaches, the general linear model (GLM) and canonical correlation analysis (CCA), have been implemented on a GPU. For an fMRI dataset of typical size (80 volumes with 64 x 64 x 22 voxels), all the preprocessing takes about 0.5 s on the GPU, compared to 5 s with an optimized CPU implementation and 120 s with the commonly used statistical parametric mapping (SPM) software. A random permutation test with 10 000 permutations, with smoothing in each permutation, takes about 50 s if three GPUs are used, compared to 0.5 - 2.5 h with an optimized CPU implementation. The presented work will save time for researchers and clinicians in their daily work and enables the use of more advanced analysis, such as non-parametric statistics, both for conventional fMRI and for real-time fMRI.

Keywords: Functional magnetic resonance imaging (fMRI), Graphics processing unit (GPU), CUDA, General linear model (GLM), Canonical correlation analysis (CCA), Random permutation test

1. Introduction & Motivation

Functional magnetic resonance imaging (fMRI), introduced by Ogawa et al. [1], besides conventional MRI and diffusion MRI [2] is a modality that is becoming more and more common as a tool for planning brain surgery and understanding of the brain. One problem with fMRI is the large amount of spatio-temporal data that needs to be processed. A normal experiment results in voxels that are of the size 2-4 mm in each dimension and the collected volumes are for example of the resolution 64 x 64 x 32 voxels. For a typical acquisition speed of one volume every other second, a 5 minute long fMRI experiment will result in 150 volumes that need to be processed. Before the data can be analyzed statistically, it has to be preprocessed in order to, for example, compensate for head movement and to account for the fact that the slices in a volume are collected at slightly different time points.

Graphic processing units (GPUs) have already been applied to a variety of fields [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] to achieve significant speedups (2 - 100 times), compared to optimized CPU implementations. Applications of GPUs in the field of MRI have been image reconstruction [13], fiber mapping from diffusion tensor MRI data [12], image registration [9, 11] and visualization of brain activity [15, 16, 17]. The main advantage of GPUs compared to CPUs is the much higher degree of parallelism. As the time series are normally regarded as independent in fMRI analysis, it is perfectly suited for parallel implementations. The first work about fMRI analysis on the GPU is the work by Gembris et al. [14] that used the GPU to speedup the calculation of correlations between voxel time series, a technique that commonly is used in resting state fMRI [18] for identifying functional brain networks. Liu et al. [19] have also used the GPU to speedup correlation analysis. We recently used the GPU to create an interactive interface, with 3D visualization, for exploratory functional connectivity analysis [6]. Another example is the work by da Silva [10] that used the GPU to speedup the simulation of a Bayesian multilevel model. A corresponding fMRI CUDA package, cudaBayesreg [20] was created for the statistical program R. A final example is the Matlab - GPU interface Jacket [21] that has been used to speedup some parts of the commonly used statistical parametric mapping (SPM) software [22].

In this work the first complete fMRI GPU processing pipeline is presented, where both the preprocessing and the statistical analysis is performed on the GPU. The following sections fur-

*Corresponding author at: Division of Medical Informatics, Department of Biomedical Engineering, Linköping University, University hospital, 581 85 Linköping, Sweden, Tel: +46 (0)13 28 67 25

Email address: anders.eklund@liu.se (Anders Eklund)

ther justify the use of GPUs for fMRI analysis.

1.1. Towards Higher Temporal and Spatial Resolution

An alternative to fMRI is electroencephalography (EEG), which provides direct information about the electrical activity of neural ensambles compared to fMRI, which at present depicts activation related blood flow changes. The sampling rate can therefore be lower, which accomodates the lower speed possible with fMRI. An aim of ongoing efforts is to achieve an acquisition speed for volume data previously only possible for single slice data. Approaches investigated currently are compressed sensing [23], parallel imaging [24] and EEG-like MR imaging [25].

The advantage of fMRI compared to EEG is a much higher spatial resolution, which can be increased even further by using stronger magnetic fields. Heidemann et al. [26] have proposed how to obtain an isotropic voxel size of 0.65 mm with a 7T MR scanner. With a repetition time of 3.5 seconds they obtain volumes of the resolution 169 x 240 x 30 voxels, which are much more demanding to process than volumes of the more common resolution 64 x 64 x 32 voxels.

Increasing temporal and spatial resolution will put even more stress on standard computers and is one reason to use GPUs for future fMRI analysis.

1.2. More Advanced Real-Time Analysis

Another reason to use GPUs is to enable more advanced real-time analysis. There are several applications of real-time fMRI, an overview is given by deCharms [27]. One of the possible applications is interactive brain mapping where the fMRI analysis runs in real-time. The physician then sees the result of the analysis directly and can interactively ask the subject to perform different tasks, rather than performing a number of experiments and then look at static activity maps.

There are only three approaches to perform the statistical analysis faster, focusing the analysis on a region of interest instead of the whole brain, calculating the same thing in a smarter way or increasing the processing power. Cox et al. [28] in 1995 proposed a recursive algorithm to be able to run the fMRI analysis in real-time and a sliding window approach was proposed by Gembris et al. [29] in 2000. Parallel processing of fMRI data is not a new concept, already in 1997 Goddard et al. [30] described parallel platforms for online analysis. Bagarinao et al. [31] proposed in 2003 the use of PC clusters in order to get a higher computational performance to do the analysis in real-time, 8 PCs where used in their study.

An emerging field of real-time fMRI is the development of brain computer interfaces (BCIs) where the subject and the computer can work together to solve a given task. We have demonstrated that it is possible to control a dynamical system [32] and communicate [33] by classifying the brain activity every second; similar work has been done by LaConte et al. [34]. deCharms et al. [35] helped subjects to suppress their pain by letting them see their own brain activity in real-time.

1.3. High Quality Visualization

Visualization of brain activity is perhaps the most obvious application of GPUs for fMRI data, as demonstrated by Rößler et al. [15] and Jainek et al. [16]. In our recent work [17] the low resolution fMRI activity volume is fused with the high resolution anatomical volume by treating the fMRI signal as a light source, such that the brain activity glows from the inside of the brain. The brain activity is in real-time estimated by performing canonical correlation analysis (CCA) on a sliding window of data.

1.4. More Advanced Conventional Analysis

A higher computational power is not only needed for real-time analysis, in 2008 Cusack et al. [36] used a PC cluster with 98 processors to analyze 1400 fMRI datasets from 330 subjects. Woolrich et al. [37] proposed a fully Bayesian spatio-temporal modeling of fMRI data based on Markov chain Monte Carlo methods. The downside of their approach is the required computation time. In 2004 the calculations for a single slice needed 6 h, i.e. a total of 5 days for a typical fMRI dataset with 20 slices. Nichols and Holmes [38] describe how to apply permutation tests for multi-subject fMRI data, instead of parametric tests such as the general linear model. In 2001 it took 2 hours to perform 2000 permutations for between subject analysis.

There is thus a need of more computational power, both for conventional fMRI and for real-time fMRI. This need is not unique to the field of fMRI, it basically exists for any kind of medical image analysis, since the amount of data that is collected for each subject has increased tremendously during the last decades.

1.5. Using a GPU instead of a PC Cluster

Previous and many current approaches to high performance computing (HPC) have to a large extent been based on PC clusters, which generally cause less acquisition costs than dedicated parallel computers, but still more than single-PC solutions and also require more efforts for administration. An application of PC clusters for fMRI data processing has been described by Stef-Praun et al. [39]. The parallel computational power of a modern graphics card provides a more efficient and less expensive alternative.

A research group that works with computational methods for tomography, at the university of Antwerp, has put together a GPU supercomputer that they call Fastra II [40]. They have made a comparison, given in Table 1, between their supercomputer (GPU SC), that consists of 13 GPUs each with 240 processor cores, and a PC cluster (PC C) called CalcUA, that consists of 512 processors. This comparison should not be considered to be completely fair since, for example, the PC cluster is a few years older than the supercomputer. The purpose of the comparison is rather to give an indication of the differences in cost, computational performance and power consumption.

As can be seen in the table, the ratio between computational performance and cost for the GPU supercomputer outreaches that of the PC cluster by three orders of magnitude. In spite of

Table 1: A comparison between a GPU supercomputer [40] and a PC cluster in terms of cost, computational performance and power consumption. TFLOPS stands for tera floating point operations per second.

Property	GPU SC	PC C
Cost	8000 USD	5 million USD
Computational performance	12 TFLOPS	3 TFLOPS
Power consumption	1.2 kW	90 kW

the speed gain the GPU system has a lower power consumption leading to less operational costs. A better transportation ability is a further advantage of GPU supercomputers. Another drawback is the availability of PC clusters, that have often to be shared between users, while a PC with one or several powerful graphic cards can be used as an ordinary desktop. A further advantage is the reduction of space requirements, e.g. useful when placing such systems for real-time applications in MR control rooms, which are often small.

1.6. Programming the GPU

In the past a special programming language, such as Open Graphics Library (OpenGL), DirectX, high level shading language (HLSL), OpenGL shading language (GLSL), C for graphics (Cg) etc., was the only possibility to take advantage of the computational power of GPUs. These languages are well suited for the implementation of calculations that are somehow related to computer graphics, but less for others. Today it is possible to program GPUs from Nvidia with a programming language called CUDA (Compute Unified Device Architecture). CUDA is very similar to standard C programming and allows the user to do arbitrary calculations on the GPU. The syntax of CUDA is easy to learn and it is easy to write a program for a certain task. In order to achieve optimal performance, deep knowledge about the GPU architecture is required. The CUDA programming language only works for GPUs from Nvidia. For general computations on GPUs from ATI something called "stream computing" exists. The Khronos group have created a new programming language, called Open Computing Language (OpenCL) [41], in order to be able to use the same code on any type of hardware, not only different kinds of GPUs. However, recent work by Kong et al. [42] shows that the same code implemented using CUDA or OpenCL can give very different performance currently.

Our implementations are based on the Matlab software (Mathworks, Natick, Massachusetts), which is used by many scientists for signal and image processing, and in particular for fMRI data analysis in SPM, as this widely employed software is programmed in Matlab. Matlab is very slow for certain calculations, such as for-loops, and to remedy this defiency it is possible to rewrite parts of the code into mex-files, which allow the embedding of C code in Matlab programs. It is now also possible to include CUDA code, which is executed by the GPU, in the mex-files functions.

1.7. Structure of the Paper

The rest of this paper is divided into six parts. The first part describes the main architecture of the Nvidia GPU, the different memory types and the principles of parallel calculations in general.

The second part describes the different kinds of preprocessing that is normally applied to the fMRI data, i.e. slice timing correction, motion compensation, detrending, etc. and how they were implemented on the GPU.

In the third part the GPU implementation of two statistical approaches, the general linear model (GLM) and canonical correlation analysis (CCA), for analysis of fMRI data is described.

Performance comparisons are presented for our implementations in SPM, Matlab, Matlab OpenMP and Matlab CUDA. The fourth part describes the general concepts of these implementations.

In the fifth part the processing times for the different implementations are given.

The paper ends with a discussion about the different implementations and the possibilities and challenges of GPU-based fMRI analysis.

2. Parallel Calculations and the Nvidia GPU Architecture

This section gives an introduction to parallel calculations in general and describes the hardware of the Nvidia GPU and that of the graphics card Nvidia GTX 480, which has been used for the implementations and testing. The interested reader is referred to the Nvidia programming guide [43] and the book about parallel calculations with CUDA by Kirk and Hwu [44] for more details about CUDA programming.

A general concept that is valid for any kind of processing on the GPU is that the data transfers between the GPU and the CPU should be minimized. Certain calculations, like the computation of the correlation between each voxel time series and the stimulus paradigm, might not result in a significant speedup, since it takes time to transfer the fMRI volumes to the GPU and the activity map back from the GPU. The fMRI data should therefore both be preprocessed and statistically analyzed on the GPU. As it is easy to visualize something that is already on the GPU, it is possible to visualize the activity map without copying it back to the CPU.

2.1. Parallel Calculations

While the central processing unit (CPU) of any computer is good at serial calculations and branching, the GPU is very good at parallel calculations. The reason for this is the evolution of more and more complex computer games with more advanced computer graphics, simulated physics and better artificial intelligence. A modern CPU has 4-6 processor cores and can run 8 - 12 threads at the same time. The current state of the art GPU from Nvidia has 480 processor cores and can run 23 040 threads at the same time.

To be able to do the calculations in parallel, the processing of each element has to be independent of the processing of other elements. The suitability of a certain application for a GPU

implementation thus greatly depends on the possible degree of parallelization. For fMRI analysis this means that the estimation of activity in a voxel should not depend on the activity result from other voxels, fortunately this is seldom the case.

When a task is performed on the GPU, a large number of threads is used. Each thread performs a small part of the calculations, e.g. only those for a single pixel or voxel. The threads are divided into a number of blocks, the grid, with a certain number of threads per block. Each block and each thread has an index that is used in order for each thread to know what data to operate on. Functions that run on the GPU are called kernels.

2.2. The Hardware Model of the Nvidia GPU

The Nvidia GTX 480 has 480 processor cores that run at 1.4 GHz. The processor cores are divided into 15 multiprocessors (MP) with 32 processor cores each. Each multiprocessor has its own cache for texture memory and constant memory, L1 cache, shared memory and a set of registers. The architecture of a multiprocessor on the GTX 480 is shown in Fig. 1. In order to achieve high performance, each multiprocessor should manage a large number of threads, 1536 is the maximum for the GTX 480. It is also important that the threads have a balanced workload, otherwise some threads will be inactive when they wait for active threads to finish.

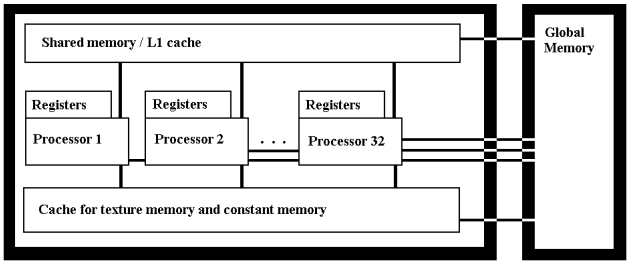

Figure 1: *The hardware model for a multiprocessor on the Nvidia GTX 480. The multiprocessor consists of 32 processor cores, which share registers, shared memory / L1 cache and a cache for constant memory and texture memory.*

One main difference between the CPU and the GPU is that the GPU has a lot of different memory types with different characteristics, while the CPU only has a big global memory and a cache memory. In order to achieve optimal performance, it is necessary to know when and how to use the different memory types. Another difference is that the memory bandwidth is much higher for the GPU, due to the fact that a lot of concurrent threads have to read from and write to the memory at the same time. In Table 2 a comparison between three different consumer graphic cards from Nvidia and ATI, from three different generations, is given.

- Global memory

  The global memory on the GPU is currently of the size 256 MB to 4 GB and the GTX 480 has a memory bandwidth of 177.4 GB/s. Despite this high bandwidth the global memory is the slowest memory type and should only be used to store variables between the execution of different kernels. To make sure that memory read and write accesses to global memory are minimized, it is important to make sure that they are *coalesced*, i.e. that as many transactions as possible are done in the same clock cycle, by taking advantage of the wide data bus.

- Constant memory

  For constants that are read by all or many of the threads, like a filter kernel in image or volume convolution, the constant memory should be used since it is cached and thereby much faster than the global memory.

- Texture memory

  The texture memory is a special way of using the global memory, such that read accesses that are local are cached, providing a good alternative when coalesced reads are hard to achieve.

  The texture memory is predestined for image registration applications, as it has hardware support for linear interpolation. This means that it does not take more time to get the trilinearly interpolated value at the floating point position (1.1, 2.5, 3.3) than the original value at the integer position (1, 2, 3). Operations such as rotation and translation of images and volumes are therefore extremely fast on the GPU.

- Shared memory

  The shared memory can be read from and written to by all the threads in the same block. Applications can be made faster by copying data from the global memory to the shared memory first. An application well suited for this type of memory is convolution: a block of pixels is first read into the shared memory and then the filter responses are calculated in parallel by letting many threads read from the shared memory at the same time. As for example a 9 x 9 filter uses 81 pixels to calculate the filter response for each pixel, and the fact that the filter response at a neighbouring position is calculated by using mainly the same pixels, the shared memory is a very efficient way to let the threads share the data.

  Due to the small size of the shared memory it is sufficient for 2D convolution, but for 3D and 4D convolution it is not as efficient. The reason for this is that the proportion of valid filter responses that fit into the shared memory decreases rapidly with every increase in the number of dimensions. One possible remedy to this is to use an optimized kernel for 2D convolution, and then incrementally calculate the filter response by calling the kernel once for each sample of the remaining dimensions.

- Registers

  The registers are specific for each thread and are used to store thread specific variables. If the number of registers per multiprocessor is divided by the number of active threads per multiprocessor, an approximate number of

possible registers per thread is obtained. Full occupancy on the GTX 480 means that 1536 threads run on each multiprocessor. At full occupancy each thread can use 21 registers at most, since the GTX 480 has 32768 registers per multiprocessor.

- L1 and L2 cache

  Each multiprocessor on the GTX 480 has a L1 cache that is used to speedup transactions to and from the global memory. The GTX 480 also has one L2 cache that helps different multiprocessors to share data in an efficient way.

3. Data

To test our implementations, a typical fMRI dataset that consists of 80 volumes of the resolution 64 x 64 x 22 voxels was used. Each voxel has the physical size of 3.75 x 3.75 x 3.75 mm. The data was collected with a Philips Achieva 1.5 T MR scanner. The data contains four time periods, where the subject alternatively performed an activity and stayed passive, in periods of 20 seconds each. The repetition time (TR) was 2 s, the echo time (TE) was 40 ms and the flip angle was 90 degrees. The sequence used for the data collection was EPI (echo planar imaging).

In single precision floating point format the fMRI dataset requires 29 MB storage size, and it is thereby no problem to fit it into the global memory on the GPU. The limited size of the global memory is otherwise one disadvantage of using GPUs.

4. Preprocessing of fMRI Data on the GPU

Before the fMRI data is statistically analyzed, several preprocessing steps are normally applied. A comprehensive review of the different preprocessing steps is given by Strother [45]. A description of the most common preprocessing steps and their implementation on the GPU are given in this section.

4.1. Slice Timing Correction

The fMRI volumes are normally acquired by sampling one slice at a time. This means that the slices in a volume are taken at slightly different timepoints. If a volume consists of 20 slices and it takes 2 seconds to acquire the volume, there is a 0.1 s difference between each slice. In order to compensate for this, slice timing correction is applied such that all the slices in a volume correspond to the same timepoint. Slice timing correction is critical for event related fMRI experiments used when higher time resolutions in the order of 100 ms are needed, which was shown e.g. by Henson et al. [46], but not for block design experiments like the one from which our data stems with an activity-rest period of 40 seconds.

The standard way to correct for the time difference between the slices is to use sinc interpolation, which can be applied by alternating the phase in the frequency domain. For each voxel time series, a forward 1D fast Fourier transform (FFT), a pointwise multiplication and an inverse 1D FFT is required to do perform sinc interpolation of the fMRI data. This is well suited for the GPU since many small FFTs can run in parallel. In our implementation a batch of forward 1D FFTs are first executed, then a kernel performs all the multiplications in the frequency domain and finally a batch of inverse 1D FFTs is applied. The CUFFT library by Nvidia includes support for launching a batch of FFTs.

4.2. Motion Compensation

Participants of fMRI studies are usually instructed not to move their head during the measurement, but residual motion can occur during the individual experiments, which are several minutes long. Therefore it is necessary to perform motion compensation of the acquired fMRI volumes, such that they are aligned to one reference volume. There exists a number of implementations for registration of fMRI volumes. One example is the work by Cox et al. [47] who in 1999 were able to perform motion compensation in real-time. Cox et al. are also the developers of the the widely used AFNI software for fMRI processing [48]. Motion compensation of fMRI volumes is also implemented in the SPM software. A good comparison of fMRI motion compensation algorithms has been presented by Oakes et al. [49].

Image registration on the GPU has been done by several groups, a recent overview is given by Shams et al. [9], and all have achieved significant speedups compared to optimized CPU implementations. The main approach that has been used for these implementations is to find the rotation and translation that maximize the mutual information between the volumes, by searching for the best parameters with an optimization algorithm. The mutual information approach to image registration is very common and was first proposed by Viola et al. [50].

An algorithm has been implemented by us that is based on optical flow of the local phase from quadrature filters [51], rather than using the image intensity as in previous GPU implementations. Phase based registration was demonstrated earlier by, for example, Hemmendorff et al. [52] and Mellor and Brady [53]. The advantage of the local phase is that it is invariant to the image intensity. Another difference to our implementation is that no optimization algorithm is needed to find the best solution. Instead an equation system is solved in each iteration and the registration parameters are incrementally updated. The equation system is given by globally minimizing the L_2 norm of the phase based optical flow equation, with respect to the parameter vector. To handle stronger movements one can use several scales, i.e. start on a low resolution scale to find the larger differences and continue on finer scales to improve the registration. For fMRI volumes, it is sufficient to use one scale and 3 iterations per volume, while mutual information based algorithms normally require 10-50 iterations per volume.

In each iteration of the algorithm, three quadrature filters that are oriented along x, y and z are applied. The quadrature filters have a spatial size of 7 x 7 x 7 voxels; their elements are complex valued in the spatial domain and are not Cartesian separable. From the filter responses, phase differences, phase gradients and certainties are calculated for each voxel and each filter. An equation system is created by summing over all the voxels and filters and is then solved to get the optimal parameter vec-

Table 2: A comparison between three Nvidia GPUs and three ATI GPUs, from three different generations, in terms of processor cores, memory bandwidth, size of shared memory, cache memory and number of registers, MP stands for multiprocessor and GB/s stands for gigabytes per second. For the Nvidia GTX 480, the user has the choice of using 48 KB of shared memory and 16 KB of L1 cache or vice versa.

Property / GPU	Nvidia 9800 GT	Nvidia GTX 285	Nvidia GTX 480	ATI Radeon HD 4870	ATI Radeon HD 5870	ATI Radeon HD 6970
Number of processor cores	112	240	480	160	320	384
Normal size of global memory	512 MB	1024 MB	1536 MB	512 MB	1024 MB	2048 MB
Global memory bandwidth	58 GB/s	159 GB/s	177 GB/s	115 GB/s	154 GB/s	176 GB/s
Constant memory	64 KB	64 KB	64 KB	48 KB	48 KB	48 KB
Shared memory per MP	16 KB	16 KB	48 / 16 KB	None	32 KB	32 KB
Floating point registers per MP	8192	16384	32768	16384	32768	32768
L1 cache per MP	None	None	16 / 48 KB	None	8 KB	8 KB
L2 cache	None	None	768 KB	None	512 KB	512 KB

tor. From the parameter vector a motion vector is calculated in each voxel and trilinear interpolation is then applied, using a 3D texture, to rotate and translate the volume. The interested reader is referred to our recent work [11] for details about the algorithm and the CUDA implementation.

Image registration is also needed in order to overlay the resulting activity map onto a high resolution anatomical T_1 weighted volume. This is not a preprocessing step, but since it is done for each fMRI experiment, and is related to motion compensation, it is mentioned here. Our registration algorithm can be used for this step as well. Prior to the registration, the activity map needs to be interpolated to have the same resolution as the T_1 volume. This is a perfect task for the GPU due to the hardware support for linear interpolation.

4.3. Smoothing

Some prefer to smooth each fMRI volume with a Gaussian lowpass filter before the statistical analysis. Convolution is an operation that is well supported by the GPU, as it is performed the same way in each voxel, except at the borders of the data. Gaussian lowpass filters have the property that they are Cartesian separable, such that the convolution in 2D or 3D can be divided into the corresponding number of 1D convolutions. For a GPU implementation it is important that the filter is separable, since this results in a significant reduction of memory reads and multiplications. For each of the 1D convolutions the fMRI data is first loaded into the shared memory and then the filter kernel, which is stored in constant memory, is applied. Our implementation supports filters up to the size of 9 x 9 x 9 voxels, which is sufficient for smoothing of fMRI volumes.

4.4. Detrending

Due to scanner imperfections and physiological noise, there are drifts and trends in the fMRI data that have to be removed prior to the statistical analysis. This is called detrending, for which different approaches exist, see for example the work by Friman et al. [54]. The detrending is normally done for each voxel time series separately, and is thereby well suited for the GPU. In our case polynomial detrending has been implemented, such that the best linear fit between the time series and a polynomial of a certain degree is removed. Basically any kind of detrending that works independently for each voxel time series is suitable for the GPU.

As the best linear fit between a time series and a number of temporal basis functions can be found by using linear regression, almost the same code is used for the GLM analysis and for the detrending. Our GPU implementation of the GLM is described in the next section.

5. Statistical Analysis of fMRI Data on the GPU

In this section the GPU implementation of two statistical approaches for fMRI data analysis will be described. The most common approach for fMRI analysis, the general linear model (GLM), and the approach preferred by us, canonical correlation analysis (CCA), have been implemented.

One possible explanation to why there is so little work about fMRI analysis on the GPU is that most of the statistical approaches use different kind of matrix operations, such as matrix inverses and eigenvalue decompositions, and there has not until recently been any official support for these kind of operations on the GPU. The Basic Linear Algebra Subprograms (BLAS) library has been implemented by Nvidia and is known as CUBLAS in CUDA. The Linear Algebra Package (LAPACK) that provides higher order matrix operations, that are e.g. used by Matlab, is not yet implemented by Nvidia, however. A package called CULA [55] supports some parts of LAPACK, one version is freely available. MAGMA [56] is another freely available package with similar functions.

5.1. Parametric Tests

5.1.1. The General Linear Model

The most common approach for statistical analysis of fMRI data is to apply the general linear model independently to each voxel time series [57]. The main idea is to find the weights of a set of temporal basis functions, that gives the best agreement with the measured signal, and then see if there is a statistical significant difference between the voxel values at rest and activity. The GLM can in matrix form be written as

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{1}$$

where $\boldsymbol{Y}$ are the observations, i.e. all the samples in the voxel time series, $\boldsymbol{\beta}$ are the parameters to optimize, $\boldsymbol{X}$ is the design matrix that is given by the stimulus paradigm and $\boldsymbol{\epsilon}$ are the errors that not can be explained by the model. By minimizing the

squared error $\|\boldsymbol{\epsilon}\|^2$, it is easy to show that the best parameters are given by

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y}. \quad (2)$$

A useful property of this expression is that the term $C = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T$ does not depend on the voxel time series and can thus be precalculated and stored in constant memory. The only thing that needs to be calculated for each voxel in order to get $\hat{\boldsymbol{\beta}}$ is thus a product between the constant term C and the current voxel time series $\boldsymbol{Y}$. If the contrast $\boldsymbol{c}$ is defined as a column vector (e.g. $[1\ 0]^T$), the t-test value is given by

$$t = \frac{\boldsymbol{c}^T\hat{\boldsymbol{\beta}}}{\sqrt{\mathrm{var}(\hat{\boldsymbol{\epsilon}})\boldsymbol{c}^{\mathrm{T}}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{c}}}. \quad (3)$$

The shared memory is used for fast calculation of the matrix product, the error and its variance. Due to the small size of the shared memory it is not possible to store intermediate results in it but only data from the voxel time series. The voxel time series $\boldsymbol{Y}$ are first loaded into the shared memory to calculate the best parameters from the matrix product between C and $\boldsymbol{Y}$. The parameters are stored in registers in each thread. The voxel time series are then loaded into shared memory a second time, to calculate the mean of the error term $\boldsymbol{\epsilon}$ from the expression

$$\bar{\epsilon} = \frac{1}{N}\sum_{t=1}^{N}(Y(t) - \boldsymbol{X}(t)\hat{\boldsymbol{\beta}}), \quad (4)$$

where N is the number of time points and $\boldsymbol{X(t)}$ are the values in the design matrix that correspond to time point t. Once the mean has been calculated the voxel time series is loaded into shared memory a third time to calculate the variance of $\boldsymbol{\epsilon}$ and then it is easy to calculate the t-test value. The term $\boldsymbol{c}^T(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{c}$ is a scalar that is precalculated and sent to the kernel. It might not sound optimal to load the same voxel time series into shared memory three times, but there is no space to, for example, store the error term in the shared memory. If the error term is stored in the global memory, it is necessary to write to *and* read from global memory, while a read is sufficient if the voxel time series is loaded into shared memory again. The shared memory is actually not necessary for the GLM, there is no data sharing between the threads, but it is used for better coding practice and it makes it easier to get coalesced reads from the global memory.

The GLM generally needs pre-whitening, to make the errors temporally uncorrelated. The whitening is normally done by first estimating an auto regressive (AR) model for each voxel time series and then removing this model from the time series. As the estimation of AR parameters is done independently for each time series, it suits perfectly for a parallel implementation. Whitening is also applied prior to a random permutation test, which is described in the section about non-parametric tests.

5.1.2. Canonical Correlation Analysis

The main problem with the GLM is that it tests each voxel time series separately and does thereby not take advantage of the spatial information. To increase the signal-to-noise ratio, and to use the spatial information to some extent, it is therefore common to apply an isotropic Gaussian lowpass filter to each fMRI volume prior to the analysis. The problem is, however, that the activity map will be blurred, which can prevent the detection of small activity areas. A better approach is to adaptively combine neighbouring pixels, by using the technique for fMRI analysis of our choice, canonical correlation analysis (CCA). Friman et al. [58] were the first to use CCA for fMRI analysis and the idea has also been used by Nandy and Cordes [59]. Canonical correlation analysis [60] maximizes the correlation between the projection of *two* multidimensional variables $\boldsymbol{x}$ and $\boldsymbol{y}$, GLM is a special case of CCA as it only handles *one* multidimensional variable. The canonical correlation ρ is defined as

$$\rho = \mathrm{Corr}(\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{x}, \boldsymbol{\gamma}^{\mathrm{T}}\boldsymbol{y}) = \frac{\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{C}_{\mathrm{xy}}\boldsymbol{\gamma}}{\sqrt{\boldsymbol{\beta}^T\boldsymbol{C}_{xx}\boldsymbol{\beta}\,\boldsymbol{\gamma}^T\boldsymbol{C}_{yy}\boldsymbol{\gamma}}}, \quad (5)$$

where $\boldsymbol{C}_{xy}$ is the covariance matrix between $\boldsymbol{x}$ and $\boldsymbol{y}$, $\boldsymbol{C}_{xx}$ is the covariance matrix for $\boldsymbol{x}$ and $\boldsymbol{C}_{yy}$ is the covariance matrix for $\boldsymbol{y}$, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are the weight vectors with unit length that determine the linear combinations of $\boldsymbol{x}$ and $\boldsymbol{y}$. To calculate the canonical correlation, the covariance matrices first need to be estimated. This can be done using the following expressions

$$\boldsymbol{C}_{xx} = \frac{1}{N-1}\sum_{t=1}^{N}\boldsymbol{x}(t)\boldsymbol{x}(t)^T, \quad (6)$$

$$\boldsymbol{C}_{yy} = \frac{1}{N-1}\sum_{t=1}^{N}\boldsymbol{y}(t)\boldsymbol{y}(t)^T, \quad (7)$$

$$\boldsymbol{C}_{xy} = \frac{1}{N-1}\sum_{t=1}^{N}\boldsymbol{x}(t)\boldsymbol{y}(t)^T, \quad (8)$$

where $\boldsymbol{x}$ denotes the temporal basis functions (and $\boldsymbol{x(t)}$ denotes the function values at time point t), $\boldsymbol{y}$ denotes the spatial basis functions (i.e. the filter responses (time courses) for the current voxel) and N is the number of time points. This means that the covariance matrices have to be estimated in each voxel, by summing over all the time points. Since the summation can be performed independently in each voxel, this operation is well suited for parallelization. Note that $\boldsymbol{C}_{xx}$ is constant and only has to be calculated once. To find the temporal and spatial weight vectors, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$, that give the highest correlation it is possible to show that the solution is given by two eigenvalue problems, such that the weight vectors are the eigenvectors, and the correlation is the square root of the corresponding eigenvalue. The eigenvalue problems can be written as

$$\boldsymbol{C}_{xx}^{-1/2}\,\boldsymbol{C}_{xy}\,\boldsymbol{C}_{yy}^{-1}\,\boldsymbol{C}_{yx}\,\boldsymbol{C}_{xx}^{-1/2}\,\boldsymbol{a} = \lambda^2\boldsymbol{a}, \quad (9)$$

$$\boldsymbol{C}_{yy}^{-1/2}\,\boldsymbol{C}_{yx}\,\boldsymbol{C}_{xx}^{-1}\,\boldsymbol{C}_{xy}\,\boldsymbol{C}_{yy}^{-1/2}\,\boldsymbol{b} = \lambda^2\boldsymbol{b}. \quad (10)$$

To get the weight vectors from $\boldsymbol{a}$ and $\boldsymbol{b}$ it is necessary to make a change of base, according to

$$\boldsymbol{\beta} = \boldsymbol{C}_{xx}^{-1/2}\,\boldsymbol{a}, \quad (11)$$

$$\boldsymbol{\gamma} = \boldsymbol{C}_{yy}^{-1/2}\, \boldsymbol{b}. \tag{12}$$

It is sufficient to solve one of the problems, as there are direct relationships between the weight vectors. The relationships can be written as

$$\boldsymbol{\beta} = \boldsymbol{C_{xx}}^{-1}\, \boldsymbol{C_{xy}}\, \boldsymbol{\gamma} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{Y}\, \boldsymbol{\gamma}, \tag{13}$$

$$\boldsymbol{\gamma} = \boldsymbol{C_{yy}}^{-1}\, \boldsymbol{C_{yx}}\, \boldsymbol{\beta}. \tag{14}$$

From the first relationship it is clear that the GLM is a special case of CCA.

GPU Implementation Considerations

It is thus necessary to calculate one matrix inverse, a number of matrix products and an eigenvalue decomposition for each voxel, i.e. in each GPU thread. The LAPACK routines that have been implemented in CULA [55] and MAGMA [56], however, do not support a matrix inversion or an eigenvalue decomposition in each thread. The standard approach to matrix inversion is to use Gauss-Jordan elimination. One though problem is that it is necessary to store several copies of the current matrix, which uses a lot of registers. Another problem is that it is an iterative method, meaning that different threads may need different execution times for the matrix inversion, which is far from optimal in CUDA programming. Fortunately there are direct solutions (i.e. not algorithms such as Gauss-Jordan elimination) that can be used to invert small matrices. Direct solutions for eigenvalue decomposition of matrices with dimensions up to 3 x 3 exists. For larger matrices it would be possible to, for example, use the Power iteration method which is very easy to implement. In our implementation the number of dimensions is kept as low as possible, allowing us to apply a direct solution for the eigenvalue decomposition.

Two temporal basis functions are used, one sine wave and one cosine wave that are of the same frequency as the stimulus paradigm. A sine and a cosine wave can be linearly combined to any phase, in order to compensate for the unknown BOLD (blood oxygenation level dependent) delay.

Guaranteeing Plausible Combinations of Spatial Basis Functions

To use neighbouring *pixels* as spatial basis functions works reasonably well as there are not that many possible spatial basis functions in 2D if a 3 x 3 neighbourhood is used. If neighbouring *voxels* from the surrounding 3 x 3 x 3 volume are used directly, there will be too many spatial basis functions such that high correlations will be found everywhere in the brain. Friman et al. [61] therefore extended their work to adaptive filtering, such that CCA instead combines the filter response from a number of anisotropic lowpass filters. The filter responses can linearly be combined to a lowpass filter with arbitrary orientation, in order to prevent unnecessary smoothing. All the possible linear combinations of the filter responses are, however, not allowed. For example, filters are excluded that are positive in one direction and negative in another direction. Such a case would imply that the voxels along the first direction are positively correlated with the stimulus paradigm, while the voxels in the other direction are negatively correlated. This form of brain activity is not very likely to occur in the brain.

To guarantee that the resulting filters are plausible, restricted CCA [62] (RCCA) was used, since it then can be guaranteed that all the filter weights are positive. There are however several problems with this approach. The first problem is that RCCA is much more computationally demanding than CCA, as the algorithm iterates the weights until they are all positive. As the number of necessary iterations can differ between the voxels, it is not well suited for a GPU implementation. Another problem is that the analysis will not be rotation invariant, as the original filters (instead of combinations of them) will be favourized by RCCA. This problem was solved by Rydell et al. [63] who also used RCCA, but added a structure tensor to estimate the best orientation of the lowpass filter.

A new solution to guarantee plausible filters and rotation invariant analysis is therefore proposed. This solution does not need RCCA and it is thereby much better suited for a GPU implementation. The filters, that are applied to the fMRI slices, are first redesigned, such that they only contain positive coefficients. To create the Cartesian non-separable anisotropic lowpass filters, $\boldsymbol{ALP}$, a small, $\boldsymbol{ILP_S}$, and a large, $\boldsymbol{ILP_L}$, isotropic Gaussian lowpass filter is first created, shown in Fig. 2. The $\boldsymbol{ALP}$, shown in Fig. 3, are then calculated as

$$\boldsymbol{ALP} = \boldsymbol{ILP_L} \cdot (1 - \boldsymbol{ILP_S}) \cdot \cos(\phi - \phi_0)^2, \tag{15}$$

where ϕ_0 was set to 0^o, 60^o and 120^o.

Four 2D filters are applied to each fMRI slice in each volume, the small lowpass filter and the three anisotropic filters. Note that this makes it possible for CCA to also create filters with different size, and not only filters with different orientation. A small lowpass filter is achieved if the weights for all the anisotropic filters are set to zero and the weight for the small lowpass filter is set to one. If the weights for the anisotropic filters and the small lowpass filter are the same, the result is instead a large lowpass filter. If the brain activity is located in a single voxel, the small lowpass filter is selected by CCA and small activity areas are thus not lost.

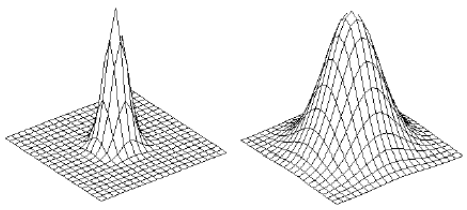

Figure 2: *The small and the large lowpass filter. For visualization purposes, these filters have a higher resolution than the filters that are actually used to smooth the fMRI slices.*

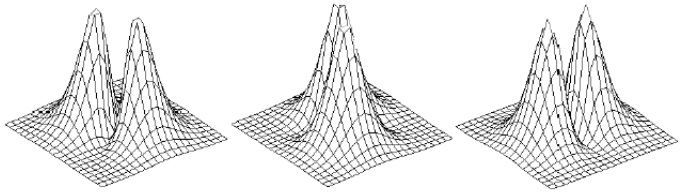

Figure 3: *The three anisotropic filters that can be linearly combined to a filter with arbitrary orientation. Note that these filters are Cartesian non-separable. For visualization purposes, these filters have a higher resolution than the filters that actually are used to smooth the fMRI slices.*

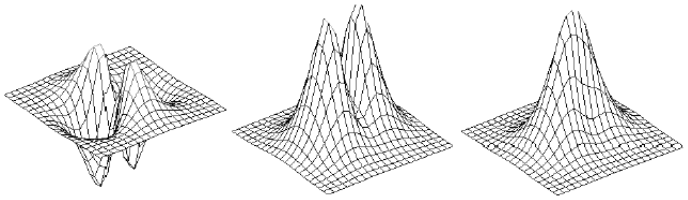

Figure 4: *The figure shows how an implausible filter (left) is adjusted to a plausible filter in two steps. First, the anisotropic filter weights are adjusted such that they all are positive (middle), and then the small lowpass filter is added (right). If a resulting filter has two directions, the direction corresponding to the largest absolute filter coefficients is preserved.*

Non-separable 2D Convolution on the GPU

As three of the four CCA filters are Cartesian non-separable, non-separable convolution has to be used, in contrast to the GLM for which separable convolution can be used. Our GPU implementation of non-separable 2D convolution uses the shared memory, such that a 64 x 64 block of pixels is first loaded into it (requiring 16 KB of shared memory), the four filter kernels are stored in the constant memory. The four filter kernels are then applied, at the same time, to the data in the shared memory and the results are then written to the global memory. In this case, as well as for the separable 3D convolution described earlier, the shared memory is very useful, since there is a lot of data sharing between the threads. Each thread block uses 512 threads and for each block 48 x 48 valid filter responses are calculated. Each thread uses 19 registers and thereby 3 blocks can run on each multiprocessor, resulting in full occupancy. For optimal performance, the convolution loops were completely unrolled manually, by generating the code with a Matlab script.

The Complete CCA GPU Algorithm

Ordinary CCA (i.e. not RCCA) is first applied to get the temporal weight vector from the 2 x 2 eigenvalue problem given by Eq. 9 (the total matrix product is a 2 x 2 matrix as two temporal basis functions are used). Instead of solving the second eigenvalue problem to get the filter weights, Eq. 14 is used to calculate the filter weights from the temporal weights. Note that by only solving Eq. 9, the calculation of a matrix square root (i.e. not element wise square root) is avoided. This could otherwise be problematic to do in each GPU thread.

The anisotropic filter weights are then adjusted such that the resulting filter is plausible. This is done by adding the absolute value of the most negative anisotropic weight to all the anisotropic filter weights. If the number of negative coefficients in the original resulting filter (without the isotropic lowpass filter) is larger than the number of positive coefficients, the sign of all the anisotropic filter weights are first flipped and then adjusted. If all the filter coefficients are negative the weights are only flipped. If all the anisotropic filter weights are positive from the beginning, no adjustment is done.

The weight of the small lowpass filter is always set to 1.2 times the largest anisotropic filter weight, to guarantee that the center pixel has the highest weight. This is important, since there otherwise can be a so-called bleeding effect in the activity map, i.e. that the activity at voxels that are close to an active area is overestimated. This is for example discussed by Cordes et al. [64], instead of using RCCA they add linear constraints to prevent CCA from creating certain voxel combinations. The total filter weight vector is then normalized such that it has unit length. An example of a filter adjustment is shown in Fig. 4. If the filter weights are adjusted, the canonical correlation has to be calculated again, using Eq. 5. For each voxel time series, i.e. in each GPU thread, it is necessary to

- Estimate the covariance matrices $\boldsymbol{C}_{yy}$ and $\boldsymbol{C}_{xy}$, $\boldsymbol{C}_{yx}$ is the transpose of $\boldsymbol{C}_{xy}$.
- Calculate the inverse of $\boldsymbol{C}_{yy}$, by applying a direct solution.
- Calculate the matrix product $\boldsymbol{C}_{xx}^{-1/2}\,\boldsymbol{C}_{xy}\,\boldsymbol{C}_{yy}^{-1}\,\boldsymbol{C}_{yx}\,\boldsymbol{C}_{xx}^{-1/2}$ in Eq. 9, which results in a 2 x 2 matrix if two temporal basis functions are used, $\boldsymbol{C}_{xx}^{-1/2}$ is precalculated since it is the same for all voxel time series.
- Calculate the eigenvector, $\boldsymbol{a}$, corresponding to the largest eigenvalue of the 2 x 2 matrix and then make a change of base, Eq. 11, to get the temporal weight vector $\boldsymbol{\beta}$. Normalize the temporal weight vector to have unit length.
- Calculate the filter weights $\boldsymbol{\gamma}$ from the temporal weights $\boldsymbol{\beta}$, by using Eq. 14. Normalize the filter weight vector to have unit length.
- If necessary, adjust the filter weights to guarantee a plausible filter.
- Calculate the canonical correlation once again, using Eq. 5, with the new filter weights.

GPU Implementation Details

The estimation of the covariance matrices requires an outer product for each time point and summing over all the time points. Similarly to the GLM implementation, the shared memory is used for these calculations. However, it is now necessary to to read 4 voxel time series into shared memory for each voxel, since there are 4 filter responses, but it is sufficient to read them from global memory once. It is necessary to store $\boldsymbol{C}_{yy}$, $\boldsymbol{C}_{yy}^{-1}$, $\boldsymbol{C}_{xy}$, the total matrix product, the temporal weight

vector $\boldsymbol{\beta}$ and the filter weight vector $\boldsymbol{\gamma}$ in each thread. $\boldsymbol{C}_{yx}$ does not need to be stored since it is the transpose of $\boldsymbol{C}_{xy}$. Due to the symmetry of $\boldsymbol{C}_{yy}$ and its inverse $\boldsymbol{C}_{yy}^{-1}$, it is sufficient to store 10 values instead of all the 16. In total this requires a lot of registers and for this reason the CCA kernel does not achieve full occupancy. The temporal basis functions $\boldsymbol{x}$, its covariance matrix $\boldsymbol{C}_{xx}$ and the matrix square root of its inverse $\boldsymbol{C}_{xx}^{-\frac{1}{2}}$ are the same for all voxel time series and are therefore precalculated and stored in the constant memory.

5.2. Non-Parametric Tests

One problem with the GLM is that it is necessary to assume that the fMRI data is normally distributed and independent. Another problem in fMRI analysis is the choice of the significance threshold, that determines which voxels are regarded as activated or not activated. Even small deviations from normality can have a large impact on the *maximum* null distribution of the test statistics [65], that is required to calculate p-values that are *corrected* for the massive multiple testing in fMRI. Non-parametric tests, such as random permutation tests, do not have these problems, as they do not require an apriori distribution function, instead empirically estimated distributions are obtained. The major drawback of non-parametric tests is that they are very computationally expensive. One subset of non-parametric tests is that of permutation tests, where the statistical analysis is done for all the possible permutations of the data, this is, however, not feasible if the number of possible permutations is very large. For a voxel time series with 80 samples, there exist $7.16 \cdot 10^{118}$ possible permutations. It is therefore common to instead do *random* permutation tests, also called Monte Carlo permutation tests, where the statistical analysis is made for a sufficiently large number of random permutations, for example 10 000, of the data.

Brammer et al. [66] was one of the first to apply non-parametric tests to fMRI data. If the subject in the MR scanner follows a repeated paradigm, the voxel time series will contain auto correlations, which first should be removed as first described in [67]. This is further discussed by Friman et al. [68] who also state that non-parametric methods will likely play an increasingly important role for fMRI analysis when more computational power is available. Belmonte et al. [69] describe an optimized algorithm for permutation testing of fMRI data. Nichols and Holmes [38] also applied permutation tests to fMRI data but only for multi subject testing, since they claim that permutations of the fMRI time series should not be done if they contain auto correlation.

Random permutation tests for CCA are especially needed, as the asymptotic distribution of the canonical correlation coefficients is rather complicated. Restricted CCA has, to our knowledge, no parametric distribution. Permutation testing for CCA has, for example, been done by Yamada et al. [70] but no processing times are stated.

Permutation testing on GPU has recently been done by Shterev et al. [5], who in one case, were able to decrease the processing time from 21 minutes on the CPU to 16 seconds on the GPU. Another example is the work by Hemert and Dickerson [8]. We recently extended our work to random permutation tests of *single subject* fMRI data on the GPU [7], in order to be able to compare activity maps from GLM and CCA at the same significance level. Before the time series are permuted, an auto regressive (AR) model is estimated for each voxel time series and each time series is then whitened. In each permutation a new time series is then created by applying an inverse whitening transform, with the permuted whitened time series as innovations. The smoothing of the fMRI volumes has to be applied *in each permutation*. If the data is smoothed prior to the whitening transform, the estimated AR parameters will change with the amount of smoothing applied, since the temporal correlations are altered by the smoothing. For our implementation of 2D CCA, 4 different smoothing filters are applied. If the smoothing is done prior to the permutations, 4 time series have to be permuted for each voxel and these time series will have different AR parameters. The smoothing will also change the null distribution of each voxel. This is incorrect since the surrogate null data that is created always should have the same properties, regardless of the amount of smoothing that is used for the analysis. If the data is smoothed after the whitening transform, but before the permutation and the inverse whitening transform, the time series that are given by simulating the AR model are incorrect since the properties of the noise are altered. The only solution to this is to apply the smoothing *after* the permutation and the inverse whitening transform, i.e. in each permutation. This is also more natural in the sense that the surrogate data is first created and then analysed. In order to get significance thresholds and p-values that are *corrected* for the multiple testing, only the maximum of all the statistical test values in the brain is saved in each permutation. This is done in order to estimate the null distribution of the *maximum* test statistics.

6. The Different Implementations

In this section the general concepts of the different implementations will be described.

6.1. SPM

The SPM software [22], which is not very optimized in terms of speed, was included in our comparison, as it is widely used for fMRI analysis. Parts of SPM are implemented as mex-files, but the computational performance is limited due to the time spent on reading from and writing to files. It might seem unfair to include this time in the comparison, but this is the time experienced by the user. As mentioned previously, the motion compensation is in our case performed in another way than SPM does. The resulting execution times for this step can therefore not be compared directly. The other preprocessing steps, and the GLM analysis, are done as in SPM. For all the preprocessing and the statistical analysis the default settings were used. For the motion compensation, the interpolation was changed to trilinear interpolation, the same interpolation used by the other implementations. All the volumes were registered to the first volume.

As SPM supports GLM, but not CCA, it is not possible to give any execution time for CCA. The detrending is done together with the model estimation, it is therefore not possible to state its execution time separately.

6.2. Matlab

Matlab eases the development of an algorithm, as it provides easy programming and there are a lot of functions that can be used directly. One issue, though, is that Matlab is slow at certain operations, such as for-loops, while it is very fast as long as the calculations are vectorized. To speedup some functions that require for-loops, mex-files were therefore used for some of the calculations, as described below.

- The motion compensation; the spatial 3D convolution, the setup of the equation system and the 3D interpolation.
- The 3D smoothing; the 2D smoothing is based on the built-in function conv2.
- The detrending and the GLM, which can be written as matrix operations, such that no for-loops are used. If one however only wants to perform the calculations for the voxels that are inside the brain, one has to use if-statements and then it gets more complicated. Therefore mex-files were used for the detrending as well as for the GLM.
- The CCA algorithm, as it requires an adjustment of the filter weights which are hard to do without for-loops.

To summarize, this implementation is very optimized for being a Matlab implementation and it can be seen as an optimized version of SPM.

6.3. Matlab OpenMP

To get an optimized CPU implementation to compare our CUDA implementation with, the Open Multi Processing (OpenMP) [71] library was used, as it lets the user take advantage of all the processor cores of the CPU. OpenMP is very easy to use, as can be seen in this example which runs a for-loop in parallel with 4 threads.

```
omp_set_num_threads(4);
#pragma omp parallel for
shared(shared variables)
private(private variables)
```

Shared variables are the variables that are shared by all the threads, like the pointers to the input and output data, while the private variables are the variables that differ between the threads, like those for the convolution result.

The FFT part of the slice timing correction was done in Matlab, since the implementation is multi threaded, while a mex-file was used to do the complex valued multiplications to perform the phase shift.

It should be noted that OpenMP is quite easy to use with Matlab mex-files under a Linux operating system, in contrast to Windows. The interested reader is referred to the book by Chapman et al. [72] for details about programming with OpenMP.

6.4. Matlab CUDA

The Matlab CUDA implementations uses ordinary CUDA programming, together with the mex interface such that the functions can still be called from Matlab. One difference between Matlab and CUDA is that Matlab uses column major order (i.e. y first) to store matrices, while CUDA assumes that the data is stored in row major order (i.e. x first). In this paper it is assumed that the data is stored in single precision in time major order before it is sent to the mex-file (since slice timing correction is the first preprocessing step), such that no conversion is necessary. This conversion will otherwise take additional time, but since it is not part of the computational time it is not included in this paper.

Before the GPU can do anything with the data, it is necessary to allocate memory and copy the data from CPU (host) memory to GPU (device) memory, this is done as follows,

`DATA_W`

is the number of columns in the data and

`DATA_H`

is the number of rows. The function

`cudaMalloc`

allocates memory on the GPU (i.e. very similar to the C-function malloc) and the

`cudaMemcpy`

function copies the data between the host and the device.

```
float *h_Image, *d_Image, *d_Result;
cudaMalloc((void **)d_Image, DATA_W * DATA_H * sizeof(float) );
cudaMalloc((void **)d_Result, DATA_W * DATA_H * sizeof(float) );
cudaMemcpy(d_Image, h_Image, DATA_W * DATA_H * sizeof(float), ...
cudaMemcpyHostToDevice);
```

After the calculations the data is copied back from the GPU to the CPU and the memory is deallocated by using the

`cudaFree`

function.

```
cudaMemcpy(h_Result, d_Result, DATA_W * DATA_H * sizeof(float), ...
cudaMemcpyDeviceToHost);
cudaFree(d_Image);
cudaFree(d_Result);
```

The CUDA programming language can easily generate 2D indices for each thread, for example by using the following code which results in 512 threads per block. The indices are required for each thread to know which part of the data to operate on.

```
// Code that is executed before the kernel is launched
int threadsInX = 32;
int threadsInY = 16;

int blocksInX = DATA_W / threadsInX;
int blocksInX = DATA_H / threadsInY;

dimGrid = dim3(blocksInX, blocksInY);
dimBlock = dim3(threadsInX, threadsInY, 1);

// Code that is executed inside the kernel
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

To generate 3D indices is harder, as each thread block can be *three* dimensional but the grid can only be *two* dimensional. One approach to generate 3D indices is given below,

```
DATA_D
```

is the number of slices in the data.

```
// Code that is executed before the kernel is launched
int threadsInX = 32;
int threadsInY = 16;
int threadsInZ = 1;

int blocksInX = (DATA_W+threadsInX-1)/threadsInX;
int blocksInY = (DATA_H+threadsInY-1)/threadsInY;
int blocksInZ = (DATA_D+threadsInZ-1)/threadsInZ;
dim3 dimGrid = dim3(blocksInX, blocksInY*blocksInZ);
dim3 dimBlock = dim3(threadsInX, threadsInY, threadsInZ);

// Code that is executed inside the kernel
int blockIdxz = __float2uint_rd(blockIdx.y * invBlocksInY);
int blockIdxy = blockIdx.y - blockIdxz * blocksInY;
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdxy  * blockDim.y + threadIdx.y;
int z = blockIdxz  * blockDim.z + threadIdx.z;
```

As fMRI data is 4D it is normally necessary to generate 4D indices and this is even more difficult. To solve this, a 3D index (x,y,z) is first created and then each thread loops over the time dimension inside the kernel. Another possibility is to call the kernel once for each sample of the remaining dimension.

To do an element wise multiplication between two volumes, $A = B \cdot C$, with floating point values, the following code can be used. Each GPU thread performs the calculation for one voxel. The (x,y,z) indices are created as previously described.

```
// The code for the kernel that runs on the GPU,
each thread performs one multiplication

__global__ void Multiply(float* A, float* B, float* C,
                         int DATA_W, int DATA_H, int DATA_D)
{
    // Do not read/write outside the allocated memory
    if (x >= DATA_W || y >= DATA_H || z >= DATA_D)
        return;

    // Calculate the linear index, row major order
    int idx = x + y * DATA_W + z * DATA_W * DATA_H;

    // Each thread performs one multiplication, A = B * C
    A[idx] = B[idx] * C[idx];
}

// The GPU kernel is launched from the CPU by the following command
Multiply<<<dimGrid, dimBlock>>>(A, B, C, DATA_W, DATA_H, DATA_D);
```

7. Results

7.1. Rotation Invariant Analysis

First, it is proven that our new approach to guarantee plausible filters results in a rotation invariant analysis. The dominant orientation of the resulting filter in each voxel was calculated for our test dataset, in which each direction is equally likely. A tensor $\boldsymbol{T}$ is first calculated according to

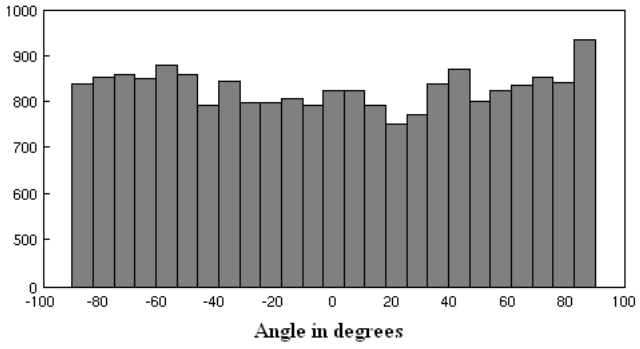

Figure 5: *The figure shows the distribution of the orientation of the resulting anisotropic lowpass filter. The orientations are approximately uniformly distributed, which means that the analysis is rotation invariant.*

$$\boldsymbol{T} = \sum_{k=1}^{3} \gamma_k \, \hat{\boldsymbol{n}}_k \, \hat{\boldsymbol{n}}_k^T, \tag{16}$$

where $\hat{\boldsymbol{n}}_k$ is the orientation vector for anisotropic filter k and γ_k is the filter weight. The eigenvector that corresponds to the largest eigenvalue of the tensor is then calculated; the dominant orientation of the filter is the angle of this eigenvector. The resulting distribution of the orientation is given in Fig. 5. The orientations are approximately uniformly distributed, meaning that the analysis is rotation invariant.

7.2. Hardware

All of our implementations use single precision format, i.e. 32 bit floating point numbers, as they provide sufficient accuracy. Our computer system, running under Linux Fedora 12, was equipped with three Nvidia GTX 480 GPUs, each equipped with 480 processor cores and 1.5 GB of memory. The CPU of our system was an Intel Xeon 2.4 GHz with 12 MB of L3 cache and 4 processor cores, 12 GB of memory was used. The CPU supports hyper threading, such that 8 threads can run in parallel. Running 8 threads, instead of 4, did, however, not improve the performance, it rather degraded it.

7.3. Processing Times

The resulting activity maps from all the implementations were inspected in order to guarantee agreement of the results. For each processing step, and for each implementation, 1000 runs were performed and the average processing time was calculated. The processing times for the all processing steps are given in Table 3, only one GPU was used for the CUDA implementation. To save additional time, the detrending and the statistical analysis is only performed for the voxels that are inside the brain. A simple thresholding technique was used for the segmentation, about 20 000 of the 90 000 voxels were classified as brain voxels.

For the motion compensation step, the three quadrature filters have to be applied in each iteration. This can either be done as spatial convolution or as a multiplication in the frequency domain (FFT based convolution). The processing times for both

spatial and FFT based convolution were therefore included. The FFT routine in Matlab is multi-threaded and very fast, even for dimensions that not are a power of two. The FFT in the CUFFT library seems well optimized for dimensions that are a power of 2, but not necessarily for other dimensions.

The processing times for different number of permutations are given in Table 4 for the GLM and in Table 5 for CCA. As our computer contains three GPUs, a multi-GPU implementation was also made, such that each GPU performs one third of the permutations.

The conversion of data from double precision column major format to single precision time major format takes about 0.4 s (this is not necessary if the data is stored in the right way from the beginning). To copy the fMRI data to the GPU takes about 20 ms and then it takes about 0.2 ms to copy the activity map back to the CPU. After the slice timing correction, the data is flipped from time major format to row major format, this takes about 2 ms.

7.4. GLM vs CCA

With the random permutation test it is possible to calculate corrected p-values for fMRI analysis by CCA, and thereby activity maps from GLM and CCA can finally be compared at the same significance level. To do this, a random permutation test with 10 000 permutations was used and then the activity maps were thresholded at the same significance level, corrected p = 0.05. With 8 mm of 2D smoothing applied to our test dataset, GLM detects 302 significantly active voxels while CCA detects 344 significantly active voxels. The aim of this small comparison is not to prove that CCA has a superior detection performance, but to show that objective evaluation of different methods for single subject fMRI analysis becomes practically possible by using fast random permutation tests.

8. Discussion

8.1. Processing Times

As can be seen in Table 3, the CUDA implementation provides significant speedups compared to SPM, Matlab and OpenMP, both for the preprocessing steps and for the statistical analysis.

In general, the motion compensation with FFT based convolution runs faster than the motion compensation based on spatial convolution, as shown in Table 3. The computation times for the CUDA implementation are much shorter compared to the other implementations, but the speed advance of the FFT approach turns into the opposite. However, if all the dimensions of the volumes are a power of 2, motion compensation with FFT based convolution is about 40% faster than with spatial convolution. The performance of the 3D FFT in the CUFFT library has been investigated and enhanced by Nukada et al. [3]. The reason for the slower speed of the spatial convolution approach for the Matlab implementation is that the 3D convolution is called 6 times, since it only can handle one real valued filter at a time, while in the OpenMP implementation all three complex valued filter responses are calculated simultaneously.

The largest speedup is achieved for the non-separable 2D smoothing; this is due to the fact that this kernel is not bounded by the global memory bandwidth as much as the other kernels. Once the data has been loaded into the shared memory, all the processor cores can access the data extremely fast and thereby achieve close to optimal processing performance. The difference compared to the other kernels that also use shared memory, is that the convolution requires a much larger number of calculations. There is also a larger speedup for CCA, than for the GLM, as CCA requires more calculations and does not need to read the data from global memory several times.

Table 4 and Table 5 clearly show that the GPU makes it practically possible to apply random permutation tests to single subject fMRI data. This enables the use of more advanced detection statistics, like RCCA, which do not have any parametric null distribution.

While it is obvious that GPUs should be used for permutation tests, it might not be as obvious that the GPU should be used for the preprocessing as well. For the GLM the preprocessing takes 15.5 s with Matlab, 5 s with OpenMP and 0.4 s with the CUDA implementation. If we look into the future a couple of years, we believe that it will be rather common with fMRI datasets that have 1 mm isotropic resolution. As mentioned earlier, fMRI data with 0.65 mm isotropic resolution has already been presented by Heidemann et al. [26]. Using a field of view of 256 mm would result in volumes of the resolution 256 x 256 x 128 voxels to cover the brain. If compressed sensing or parallel imaging is used at the same time, the temporal resolution might be increased to 2 Hz, compared to the more common 0.5 Hz. An fMRI experiment that consists of four periods of rest and activity, where each period is 40 seconds, would then result in 320 volumes of the resolution 256 x 256 x 128 voxels to process. For fMRI datasets of this size, it becomes much more important with an efficient implementation in order to process the data. If the processing time scales linearly with the size of the data, the Matlab implementation would require 96 minutes to preprocess the data, OpenMP 31 minutes and the CUDA implementation 1.7 minutes (FFT based convolution used in all three cases).

8.2. Implementing GLM & CCA

As the GLM does not require any higher order matrix functions, like CCA does, the implementation efforts were low, requiring only the programming of a matrix multiplication and a variance calculation.

To be able to do CCA the filters first had to be redesigned, such that RCCA is not needed. The implementation of CCA is more demanding as it requires a matrix inverse and an eigenvalue decomposition for each voxel time series - fortunately there are direct solutions for small matrices. Our current implementation only works for adaptive 2D filtering. For adaptive 3D filtering the same approach to guarantee plausible filters can be used, but a total of seven 3D filters must be used and thereby a 7 x 7 matrix must be inverted in each thread, which requires a lot more registers. fMRI analysis by CCA is a good example for the situation that an existing algorithm might have to be altered in order to fit the architecture of the GPU. An alternative

Table 3: Processing times of the different processing steps for the different implementations. The size of the fMRI dataset is 80 volumes with the resolution 64 x 64 x 22 voxels.

Processing step	SPM	Matlab	Matlab OpenMP	Matlab CUDA
Slice timing correction	32 s	280 ms	235 ms	7.8 ms
Motion compensation, spatial convolution	28 s	126 s	17.5 s	415 ms
Motion compensation, FFT convolution	-	13.7 s	4.6 s	650 ms
Smoothing, one separable 3D filter of size 9 x 9 x 9 voxels for GLM	32 s	1.5 s	195 ms	10.4 ms
Smoothing, four non separable 2D filters of size 9 x 9 pixels for CCA	-	5.9 s	850 ms	9.9 ms
Detrending, for GLM	-	8.6 ms	4.6 ms	0.37 ms
Detrending, for CCA	-	33.6 ms	17.1 ms	1.44 ms
GLM	33 s	16.6 ms	5.8 ms	0.38 ms
CCA	-	31.5 ms	15.2 ms	0.47 ms
Total time for GLM	125 s	15.51 s	5.04 s	0.43 s
Total time for CCA	-	19.93 s	5.72 s	0.43 s

Table 4: Processing times for random permutation tests with the GLM for the different implementations. Note that smoothing of the fMRI volumes is done in each permutation.

Number of permutations with GLM	Matlab	Matlab OpenMP	Matlab CUDA, 1 x Nvidia GTX 480	Matlab CUDA, 3 x Nvidia GTX 480
1000	25 min	3.5 min	13.5 s	4.5 s
10 000	4 h 10 min	35 min	2 min 15 s	45 s
100 000	1 day 17 h 40 min	5 h 50 min	22.5 min	7.5 min

approach to guarantee plausible filters could be to include linear constraints, as proposed by Cordes et al. [64]. In our future work we want to investigate how other statistical approaches suit for a GPU implementation.

8.3. Training of Classifiers

A growing field in fMRI is the development of BCIs where the brain activity is classified in real-time, allowing control of various systems. In order to classify the brain activity, a classifier first has to be trained. Catanzaro et al. [4] have implemented support vector machine (SVM) training and classification on the GPU and achieved a speedup of 9-35 for training and 81-138 for classification. Classifiers are not only used to classify brain activity in real-time, but also to improve conventional fMRI analysis by incorporating spatial information in adaptive ways. Åberg et al. [73] have proven that fMRI analysis by finding important voxels for SVM performs better than the GLM. To find the most important voxels for the classifier, an evolutionary algorithm is used in which different voxel clusters are used to train and evaluate the classifier. According to the paper this takes about 20 minutes, but it is also stated that evolutionary algorithms are fairly trivial to run in parallel, and thereby they would be well suited for the GPU.

9. Conclusions

To summarize, doing the fMRI analysis on the GPU opens up a lot of possibilities. All the preprocessing steps and both the statistical approaches that we used suit the GPU well and significant speedups were obtained. The obvious advantage is that this saves time for researchers and clinicians, but it also makes it easier to play around with the different parameters, to for example see the effects of different amounts of smoothing. One of the most important results is that the GPU makes it practically possible to apply non-parametric statistics. We believe that the GPU will result in a more common usage of permutation tests for fMRI data. One of the challenges is to make GPU programming easy, such that as many as possible can take advantage of the computational power of GPUs. The main challenge, as we see it, is to make fMRI analysis on the GPU available to the large community that does fMRI analysis, i.e. both researchers and clinicians.

10. Software

The authors are currently working on a software package that will be freely available, the name of the package is WABAACUDA (Wanderines Accelerated Brain Activity Analyzer using CUDA). The software package will be available under the GNU general public license (GPL) at

`http://www.wanderineconsulting.com/wabaacuda`

Acknowledgement

This work was supported by the strategic research center MOVIII, funded by the Swedish foundation for strategic research (SSF), and the Linnaeus center CADICS, funded by the Swedish research council. The authors would like to thank the NovaMedTech project at Linköping university for financial support of our GPU hardware and Johan Wiklund for support with the CUDA installations.

References

[1] S. Ogawa, D. Tank, R. Menon, J. Ellermann, S. Kim, H. Merkle, K. Ugurbil, Intrinsic signal changes accompanying sensory stimulation: Functional brain mapping with magnetic resonance imaging, Proc. Natl. Acad. Sci. USA 89 (1992) 5951–5955.

Table 5: Processing times for random permutation tests with 2D CCA for the different implementations. Note that smoothing of the fMRI volumes is done in each permutation.

Number of permutations with 2D CCA	Matlab	Matlab OpenMP	Matlab CUDA, 1 x Nvidia GTX 480	Matlab CUDA, 3 x Nvidia GTX 480
1000	1 h 40 min	14 min 50 s	15 s	5 s
10 000	16 h 37 min	2 h 28 min	2 min 30 s	50 s
100 000	6 days 22 h	24 h 43 min	25 min s	8 min 20 s

[2] P. Basser, J. Mattiello, D. LeBihan, MR diffusion tensor spectroscopy and imaging, Biophysical Journal 66 (1994) 259–267.

[3] A. Nukada, Y. Ogata, T. Endo, S. Matsuoka, Bandwidth intensive 3-D FFT kernel for GPUs using CUDA, in: International Conference for High Performance Computing, Networking, Storage and Analysis, 2008, pp. 1–11.

[4] B. Cataranzo, N. Sundaram, K. Keutzer, Fast support vector machine training and classification on graphics processors, in: Proceedings of the 25th International conference on Machine learning, 2008, pp. 104–111.

[5] I. Shterev, S.-H. Jung, S. George, K. Owzar, permGPU: Using graphics processing units in RNA microarray association studies, BMC Bioinformatics 11 (2010) 329.

[6] A. Eklund, O. Friman, M. Andersson, H. Knutsson, A GPU accelerated interactive interface for exploratory functional connectivity analysis of fMRI data, in: IEEE International Conference on Image Processing (ICIP), 2011.

[7] A. Eklund, M. Andersson, H. Knutsson, Fast random permutation tests enable objective evaluation of methods for single subject fMRI analysis, International Journal of Biomedical Imaging, Article ID 627947 (2011).

[8] John L. Van Hemert, Julie A. Dickerson, Monte Carlo randomization tests for large-scale abundance datasets on the GPU, Computer Methods and Programs in Biomedicine 101 (2011) 80–86.

[9] R. Shams, P. Sadeghi, R. A. Kennedy, R. I. Hartley, A survey of medical image registration on multicore and the GPU, IEEE Signal Processing Mag. 27 (2010) 50–60.

[10] A.R. Ferreira da Silva, A Bayesian multilevel model for fMRI data analysis, Computer Methods and Programs in Biomedicine 102 (2010) 238–252.

[11] A. Eklund, M. Andersson, H. Knutsson, Phase based volume registration using CUDA, in: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2010, pp. 658 – 661.

[12] T. McGraw, M. Nadar, Stochastic DT-MRI connectivity mapping on the GPU, IEEE Transactions on Visualization and computer graphics 13 (2007) 1504–1511.

[13] S. Stone, J. Haldar, S. Tsao, W. Hwu, B. Sutton, Z. Liang, Accelerating advanced MRI reconstructions on GPUs, Journal of Parallel and Distributed Computing 68 (2008) 1307–1318.

[14] D. Gembris, M. Neeb, M. Gipp, A. Kugel, R. Männer, Correlation analysis on GPU systems using NVIDIA's CUDA, Journal of real-time image processing (2010) 1–6.

[15] F. Rößler, E. Tejada, T. Fangmeier, T. Ertl, M. Knauff, GPU-based multivolume rendering for the visualization of functional brain images, Proceedings of SimVis (2006) 305–318.

[16] W. M. Jainek, S. Born, D. Bartz, W. Straßer, J. Fischer, Illustrative hybrid visualization and exploration of anatomical and functional brain data, Computer Graphics Forum 27 (2008) 855–862.

[17] T. K. Nguyen, A. Eklund, H. Ohlsson, F. Hernell, P. Ljung, C. Forsell, M. Andersson, H. Knutsson, A. Ynnerman, Concurrent volume visualization of real-time fMRI, in: Proceedings of the 8th IEEE/EG International Symposium on Volume Graphics, 2010, pp. 53–60.

[18] B. Biswal, F. Yetkin, V. Haughton, J. Hyde, Functional connectivity in the motor cortex of resting state human brain using echo-planar MRI, Magnetic Resonance in Medicine 34 (1995) 537–541.

[19] W. Liu, P. Zhu, J. Anderson, D. Yurgelun-Todd, P. Fletcher, Spatial regularization of functional connectivity using high-dimensional markov random fields, Proceedings of the 13th International Conference on Medical Image Computing and Computer Assisted Intervention (MICCAI), Lecture notes in computer science 6362 (2010) 363–370.

[20] A.R. Ferreira da Silva, cudaBayesreg: Bayesian Computation in CUDA, The R Journal 2/2 (2010) 48–55.

[21] Jacket - The GPU Engine for Matlab, 2010. http://www.accelereyes.com/.

[22] Statistical Parametric Mapping (SPM) software for fMRI analysis, 2010. http://www.fil.ion.ucl.ac.uk/spm/.

[23] M. Lustig, D. Donoho, J. Pauly, Sparse MRI: The application of compressed sensing for rapid MR imaging, Magnetic Resonance In Medicine 58 (2007) 1182–1195.

[24] X. Golay, J. A. de Zwart, Y.-C. L. Ho, Y.-Y. Sitoh, Parallel imaging techniques in functional MRI, Topics in Magnetic Resonance Imaging 15 (2004) 255–265.

[25] B. Zahneisen, T. Grotz, K. Lee, S. Ohlendorf, M. Reisert, M. Zaitsev, J. Hennig, Three-dimensional MR-encephalography: Fast volumetric brain imaging using rosette trajectories, Magnetic Resonance in Medicine 65 (2011) 1260 – 1268.

[26] R. M. Heidemann, D. Ivanov, R. Trampel, J. Lepsien, F. Fasano, J. Pfeuffer, R. Turner, Isotropic sub-millimeter fMRI in humans at 7T, in: Proceedings of the annual meeting of the International society for magnetic resonance in medicine (ISMRM), 2010, p. 1083.

[27] R. C. deCharms, Applications of real-time fMRI, Nature Reviews Neuroscience 9 (2008) 720–729.

[28] R. W. Cox, A. Jesmanowicz, J. S. Hyde, Real-time functional magnetic resonance imaging, Magnetic Resonance in Medicine 33 (1995) 230–236.

[29] D. Gembris, J. G. Taylor, S. Schor, W. Frings, D. Suter, S. Posse, Functional magnetic resonance imaging in real time (FIRE): Sliding-window correlation analysis and reference-vector optimization, Magnetic Resonance in Medicine 43 (2000) 259 – 268.

[30] N. Goddard, G. Hood, J. Cohen, W. Eddy, C. Genovese, D. Noll, L. Nystrom, Online analysis of functional MRI datasets on parallel platforms, Journal of Supercomputing 11 (1997) 295–318.

[31] E. Bagarinao, K. Matsuo, T. Nakai, Real-time functional MRI using a PC cluster, Concepts in magnetic resonance 19B (2003) 14–25.

[32] A. Eklund, H. Ohlsson, M. Andersson, J. Rydell, A. Ynnerman, H. Knutsson, Using real-time fMRI to control a dynamical system by brain activity classification, Proceedings of the 12th International Conference on Medical Image Computing and Computer Assisted Intervention (MICCAI), Lecture notes in computer science 5761 (2009) 1000–1008.

[33] A. Eklund, M. Andersson, H. Ohlsson, A. Ynnerman, H. Knutsson, A brain computer interface for communication using real-time fMRI, in: Proceedings of International Conference on Pattern Recognition (ICPR), 2010, pp. 3665–3669.

[34] S. M. Laconte, S. J. Peltier, X. P. Hu, Real-time fMRI using brain-state classification, Human Brain Mapping 28 (2007) 1033–1044.

[35] R. C. deCharms, F. Maeda, G. H. Glover, D. Ludlow, J. M. Pauly, D. Soneji, J. D. Gabrieli, S. C. Mackey, Control over brain activation and pain learned by using real-time functional MRI, PNAS 102 (2005) 18626–18631.

[36] R. Cusack, A. M. Owen, Distinct networks of connectivity for parietal but not frontal regions identified with a novel alternative to the "resting state" method, in: Fifteenth Annual Meeting of the Cognitive Neuroscience Society, 2008.

[37] M. W. Woolrich, M. Jenkinson, M. Brady, S. M. Smith, Fully bayesian spatio-temporal modeling of fMRI data, IEEE Transactions on Medical Imaging 23 (2004) 213–231.

[38] T. E. Nichols, A. P. Holmes, Nonparametric permutation tests for functional neuroimaging: A primer with examples, Human Brain Mapping 15 (2001) 1–25.

[39] T. Stef-Praun, B. Clifford, I. Foster, U. Hasson, M. Hategan, S. Small, M. Wilde, Y. Zhao, Accelerating medical research using the swift workflow system, Studies in Health Technology and Informatics 126 (2007) 207–216.

[40] FASTRA II, 2010. http://fastra2.ua.ac.be/.

[41] The Khronos Group & OpenCL, 2010. http://www.khronos.org/opencl/.

[42] J. Kong, M. Dimitrov, Y. Yang, J. Liynage, L. Cao, J. Staples, M. Man-

tor, H. Zhou, Accelerating Matlab image processing toolbox functions on GPUs, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010, pp. 75–85.
[43] Nvidia, 2010. CUDA Programming Guide, Version 3.0.
[44] D. Kirk, W. Hwu, Programming Massively Parallel Processors, A Hands-on Approach, Morgan Kaufmann, 2010. ISBN 978-0-12-381472-2.
[45] S. Strother, Evaluating fMRI preprocessing pipelines, IEEE Engineering in Medicine and Biology Magazine 25 (2006) 27–41.
[46] R. Henson, C. Buchel, O. Josephs, K. Friston, The slice-timing problem in event-related fMRI, Neuroimage 9 (1999) 125.
[47] R. W. Cox, A. Jesmanowicz, Real-time 3D image registration for functional MRI, Magnetic Resonance in Medicine 42 (1999) 1014–1018.
[48] R. Cox, AFNI: Software for analysis and visualization of functional magnetic resonance neuroimages, Computers and Biomedical Research 29 (1996) 162–173.
[49] T. Oakes, T. Johnstone, K. Ores Walsh, L. Greischar, A. Alexander, A. Fox, R. Davidson, Comparison of fMRI motion correction software tools, Neuroimage 28 (2005) 529–543.
[50] P. Viola, W. Wells, Alignment by maximization of mutual information, International Journal of Computer Vision 24 (1997) 137–154.
[51] G. Granlund, H. Knutsson, Signal Processing for Computer Vision, Kluwer Academic Publishers, 1995. ISBN 0-7923-9530-1.
[52] M. Hemmendorff, M. Andersson, T. Kronander, H. Knutsson, Phase-based multidimensional volume registration, IEEE Transactions on Medical Imaging 21 (2002) 1536–1543.
[53] M. Mellor, M. Brady, Phase mutual information as similarity measure for registration, Medical Image Analysis 9 (2005) 330–343.
[54] O. Friman, M. Borga, P. Lundberg, H. Knutsson, Detection and detrending in fMRI data analysis, NeuroImage 22 (2004) 645–655.
[55] CULA - GPU-accelerated LAPACK, 2010. http://www.culatools.com/.
[56] MAGMA - Matrix Algebra on GPU and Multicore Architectures, 2010. http://icl.cs.utk.edu/magma/.
[57] K. Friston, A. Holmes, K. Worsley, J. Poline, C. Frith, R. Frackowiak, Statistical parametric maps in functional imaging: A general linear approach, Human Brain Mapping 2 (1995) 189–210.
[58] O. Friman, J. Carlsson, P. Lundberg, M. Borga, H. Knutsson, Detection of neural activity in functional MRI using canonical correlation analysis, Magnetic Resonance in Medicine 45 (2001) 323–330.
[59] R. Nandy, D. Cordes, A novel nonparametric approach to canonical correlation analysis with applications to low CNR functional MRI data, Magnetic Resonance in Medicine 49 (2003) 1152–1162.
[60] H. Hotelling, Relation between two sets of variates, Biometrika 28 (1936) 322–377.
[61] O. Friman, M. Borga, P. Lundberg, H. Knutsson, Adaptive analysis of fMRI data, NeuroImage 19 (2003) 837–845.
[62] S. Das, P. Sen, Restricted canonical correlations, Linear Algebra and its Applications 210 (1994) 29–47.
[63] J. Rydell, H. Knutsson, M. Borga, On rotational invariance in adaptive spatial filtering of fMRI data, NeuroImage 30 (2006) 144–150.
[64] D. Cordes, R. Nandy, M. Jin, Constrained CCA with different novel linear constraints and a nonlinear constraint in fMRI, in: Proceedings of the annual meeting of the International society for magnetic resonance in medicine (ISMRM), 2010, p. 1151.
[65] R. Viviani, P. Beschoner, K. Ehrhard, B. Schmitz, J. Thöne, Non-normality and transformations of random fields, with an application to voxel-based morphometry, NeuroImage 35 (2007) 121–130.
[66] M. J. Brammer, E. T. Bullmore, A. Simmons, S. C. R. Williams, P. M. Grasby, R. J. Howard, P. R.Woodruff, S. Rabe-Hesketh, Generic brain activation mapping in functional magnetic resonance imaging: A non-parametric approach, Magnetic Resonance Imaging 15 (1997) 763–770.
[67] J. J. Locascio, P. J. Jennings, C. I. Moore, S. Corkin, Time series analysis in the time domain and resampling methods for studies of functional magnetic resonance brain imaging, Human Brain Mapping 5 (1997) 168–193.
[68] O. Friman, C.-F. Westin, Resampling fMRI time series, NeuroImage 25 (2005) 859–867.
[69] M. Belmonte, D. Yurgelun-Todd, Permutation testing made practical for functional magnetic resonance image analysis, IEEE Transactions on Medical Imaging 20 (2001) 243–248.
[70] T. Yamada, T. Sugiyama, On the permutation test in canonical correlation analysis, Computational Statistics & Data Analysis 50 (2006) 2111–2123.
[71] OpenMP - The OpenMP API Specification for Parallel Programming, 2010. http://www.openmp.org/.
[72] B. Chapman, G. Jost, R. van der Pas, Using OpenMP, Portable Shared Memory Parallel Programming, MIT Press, 2007. ISBN 978-0-262-53302-7.
[73] M. B. Åberg, J. Wessberg, An evolutionary approach to the identification of informative voxel clusters for brain state discrimination, IEEE Journal of selected topics in signal processing 2 (2008) 919–928.