**Methods of fMRI segmentation**

1. Brief state of the art in methods of fMRI segmentation – o metodach segmentacji fMRI – tutaj też można wycisnąć ze 2-3 strony??
2. Description of the implemented PCA method
3. Foundations and analytical derivation of the methods ???? – tutaj podstawy teoretyczne algorytmu PCA, 1 strona
4. Block diagram of the code (general)
5. Get\_mu – center the data
6. Svd
7. Enforce a sign convention

3. Mój plan pracy po ‘Foundations and analytical derivation’

1. Wstęp do matlabowskiej wersji algorytmu – o tym, że jest to oparte na matlabie
2. I tutaj dopiero można wkleić jakiś diagram / schemat algorytmu
3. Ale też z grubsza opisać te kroki (nie robić tego razem z opisem implementacji)
4. Opis implementacji w CUDA

**FOUNDATIONS AND ANALYTICAL DERIVATION OF THE METHODS**

**Principal component analysis** (**PCA**) is a statistical procedure that uses an [orthogonal transformation](https://en.wikipedia.org/wiki/Orthogonal_transformation" \o "Orthogonal transformation) to convert a set of observations of possibly correlated variables into a set of values of [linearly uncorrelated](https://en.wikipedia.org/wiki/Correlation_and_dependence" \o "Correlation and dependence) variables called **principal components**. The number of principal components is less than or equal to the number of original variables. This transformation is defined in such a way that the first principal component has the largest possible [variance](https://en.wikipedia.org/wiki/Variance" \o "Variance) (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is [orthogonal](https://en.wikipedia.org/wiki/Orthogonal" \o "Orthogonal) to the preceding components. The resulting vectors are an uncorrelated orthogonal basis set. The principal components are orthogonal because they are the [eigenvectors](https://en.wikipedia.org/wiki/Eigenvector" \o "Eigenvector) of the[covariance matrix](https://en.wikipedia.org/wiki/Covariance_matrix), which is [symmetric](https://en.wikipedia.org/wiki/Symmetric_matrix" \l "Real_symmetric_matrices" \o "Symmetric matrix).

Principal Component Analysis (PCA) is a useful statistical technique that has found application in fields such as face recognition, image compression, and is a common technique for findings patterns in data of high dimension.

PCA is mostly used as a tool in exploratory data analysis and for making predictive models. PCA can be done by eigenvalue decomposition of a data covariance (or correlation) matrix or singular value decomposition of a data matrix, usually after centring (and normalizing or using Z-scores) the data matrix of each attribute.

PCA can be seemed as a way of identifying patterns in data, and expressing the data in such a way as to highlight their similarities and differences. Since patterns in data can be hard to find in data of high dimension, where the luxury of graphical representation is not available, PCA is a powerful tool for analysing data.

For PCA to work correctly it is necessary to subtract the mean from each of the data dimensions. The mean subtracted is the average across each dimension.

**Applications – neuroscience.**

In neuroscience, PCA is also used to discern the identity of a neuron from the shape of its action potential. [Spike sorting](https://en.wikipedia.org/wiki/Spike_sorting" \o "Spike sorting) is an important procedure because [extracellular](https://en.wikipedia.org/wiki/Electrophysiology" \l "Extracellular_recording" \o "Electrophysiology)recording techniques often pick up signals from more than one neuron. In spike sorting, one first uses PCA to reduce the dimensionality of the space of action potential waveforms, and then performs [clustering analysis](https://en.wikipedia.org/wiki/Cluster_analysis" \o "Cluster analysis) to associate specific action potentials with individual neurons.

PCA as a dimension reduction technique is particularly suited to detect coordinated activities of large neuronal ensembles. It has been used in determining collective variables, i.e. [order parameters](https://en.wikipedia.org/w/index.php?title=Order_parameters&action=edit&redlink=1), during [phase transitions](https://en.wikipedia.org/wiki/Phase_transitions" \o "Phase transitions) in the brain.

**A Sparse PCA**

A particular disadvantage of PCA is that the principal components are usually linear combinations of all input variables. Sparse PCA overcomes this disadvantage by finding linear combinations that contain just a few input variables.

**Wstęp do matlabowskiej wersji algorytmu**

Matlab pca function (standard library) is a reference implementation of this study. The implementation on CUDA bases on a code of this method. It is a default mode so singular value decomposition algorithm is used. An ‘economic’ version is implemented (**which means that the first N columns are returned**) as it is more efficient and the obtained result is enough to continue algorithm. It is worth mentioning that non-economic version would not run successfully on many GPUs as it requires a lot of memory for big datasets.

**Diagram / schemat algorytmu**

A diagram of an algorithm implemented on CUDA platform is illustrated below:

General description of the algorithm:

In general PCA methods include some optional steps and exist different versions based on singular value decomposition (SVD), eigenvalue decomposition or alternating least squares

The first step (but might be optional) in the PCA method is to center the data. Then the main part is coming which is singular value decomposition. This point is the most expensive from the computational cost standpoint. The algorithm ends with enforcing a sign convention on the coefficients (the largest element in each column will have a positive sign).

“to conform CULA alignment requirements”

**AN IMPLEMENTATION DESCRIPTION**

Plan opisu implementacji w CUDA:

1. Wczytywanie danych nifti
2. Transfer danych, pinned transfer, zysk czasowy
3. Implementacja wymuszona rozmiarem danych (m>n)
4. Transpozycje macierzy danych
5. Obliczanie mu, shuffle version, wątki per block, flexible grid-strip loop approach
6. SVD – CULA, svd cuda, wersja ekonomiczna algorytmu
7. Colsign2
8. Get\_colsign – predykaty zastąpione mnożeniem (szybsze)

I generalnie o takich technikach jak:

* Grupowanie dostępu do pamięci
* Czytanie kolumnami
* Są różne wersje algorytmu redukcji sum, różniące się również wydajnością między sobą
* Liczba wątków na wieloprocesorze powinna być wieloktronością 32

A detailed description of the implemented code goes step by step from the beginning of the program (initial memory transfer) up to the last part of the algorithm. The code was optimized for the specific dimensions of matrices (m >> n) of tested fMRI data. It might not be efficient on a data with another ratio of dimensions.

1.1 Memory transfer – a może o tym memory transfer później, bo to w końcu nie jest najważniejsze?

Using pinned memory instead of pageable memory to transfer the data from a host to a device we can obtain some speed-up. On a sample of fMRI data of size 24 838 kB, to copy it using pinned memory takes on average 122 ms, whereas the same operation performed by pageable memory lasts on average 170 ms. However in some cases memory transfer ~~over~~ pinned memory may fail especially where the data size is too big (it depends on an operating system).

1.2 Centering the data – get\_mu – obliczanie MU

Centring the data in this case is done by calculating the average of each column, and subtract this average from each element of the column. The simplest scheme to calculate the average is to do a sum reduction. Each column is processed by one block, so that shared memory is used for sum reduction.

Because the data are stored in a column-major order, global memory access is coalesced, which is very important for efficiency in case of sum reduction. A version using shuffle instruction (to compute sum reduction) was tested but did not result in better performance (there was no speed-up).

The number of threads per block should be a multiple of 32 even if the number of elements in a columns is different. It complicates a bit the code (conditional branches) but makes CUDA program run faster.

To process fMRI data in a matrix, three voxel’s dimension (x,y,z) form one dimension (a row in matrix) and the second one is maintained by time course. This causes that the dimensions of the formed matrix vary a lot. For example in a dataset used in this study, the size of the matrix was 163840 x 121. Processing such data in CUDA in an efficient way is not trivial.

As one block processes one row, we can configure kernel to have as many blocks in grid as rows. This style of kernel is known as a *monolithic kernel*, because it assumes a single large grid of threads to process the entire array in one pass (which for the GPUs used in this study this is still possible – the maximum number of blocks per grid (x dimension) is 2147483648). But instead of completely eliminating the loop when parallelizing the computation, it is more flexible approach to use a grid-stride loop. Kernel is configured to have 63536 blocks in grid, so it processes rows in a loop (number of rows in the case of using dataset is 163840).

There are some benefits to using a grid-stride loop. The first one is scalability and thread reuse. By using a loop a program is able to support any problem size even if it exceeds the largest grid size CUDA device supports. Moreover we can limit the number of block to tune performance.

1.3 SVD

The most important (computational complexity and cost) part of the algorithm is singular value decomposition. A quick research of already implemented SVD method has shown that there are not many libraries for CUDA offering it. In fact CUDA API includes cuSOLVER library with SVD methods but they do not support “economic” version of the algorithm, so they are impractical for the large datasets. In this work an implementation of SVD from CULA library was used. This is the library of linear algebra methods basing on LAPACK library implementation. It supports ‘economic’ version and works faster when m > n (the number of rows of the matrix is bigger than the number of columns). CULA routines expect that any data provided will be stored in “column-major” order.

However CULA library has not been developed since 2013, so it does not take advantage of the features new CUDA release offers.

1.4 Sign convention on the coefficients

To find the largest element (absolute value) in each column of very large length (for example 163840) we cannot use ‘one block per column’ approach. The maximum number of threads per block is 1024. The solution is to have multiple blocks per column each one processing another part of a column, and writing a result to an intermediate array. It is then using by the second program kernel, which collects obtained results, finds the largest element in each column, and if its sign is negative it multiplies every element in the column by -1.

Searching for the largest element implementation does not include ‘if’ instruction (conditional branches) but use multiplication instead. It is recommended to avoid conditional branches as much as possible.

An implementation of this part is divided on two kernel programs. The first one searches (multiplication instead of predicates

GPU technical specification:

CUDA program version was tested on a GPU GeForce GTX TITAN X with compute capability 5.2. It has 24 multiprocessors each one containing 128 cores which in total is 3072 CUDA cores. GPU max clock rate is 1076 MHz.

Matlab’s script were tested on Intel Core i3-2120 CPU 3.3 GHz processor with 8 GB RAM.