**Methods of fMRI segmentation**

**FOUNDATIONS AND ANALYTICAL DERIVATION OF THE METHODS**

This section describes the foundations and applications of Principal Component Analysis (PCA) method. It is a statistical technique 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 graphical representation is not available, PCA is a powerful tool for analysing data.

Principal component analysis procedure uses an [orthogonal transformation](https://en.wikipedia.org/wiki/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) 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) (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) 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) of the[covariance matrix](https://en.wikipedia.org/wiki/Covariance_matrix), which is [symmetric](https://en.wikipedia.org/wiki/Symmetric_matrix#Real_symmetric_matrices).

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.

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

**Applications.**

Principal Component Analysis is a useful 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.

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) is an important procedure because [extracellular](https://en.wikipedia.org/wiki/Electrophysiology#Extracellular_recording)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) 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) in the brain.

Finally in this work we study an application of PCA to segment fMRI images, which can be used to obtain Default Mode Network (DMN).

**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.

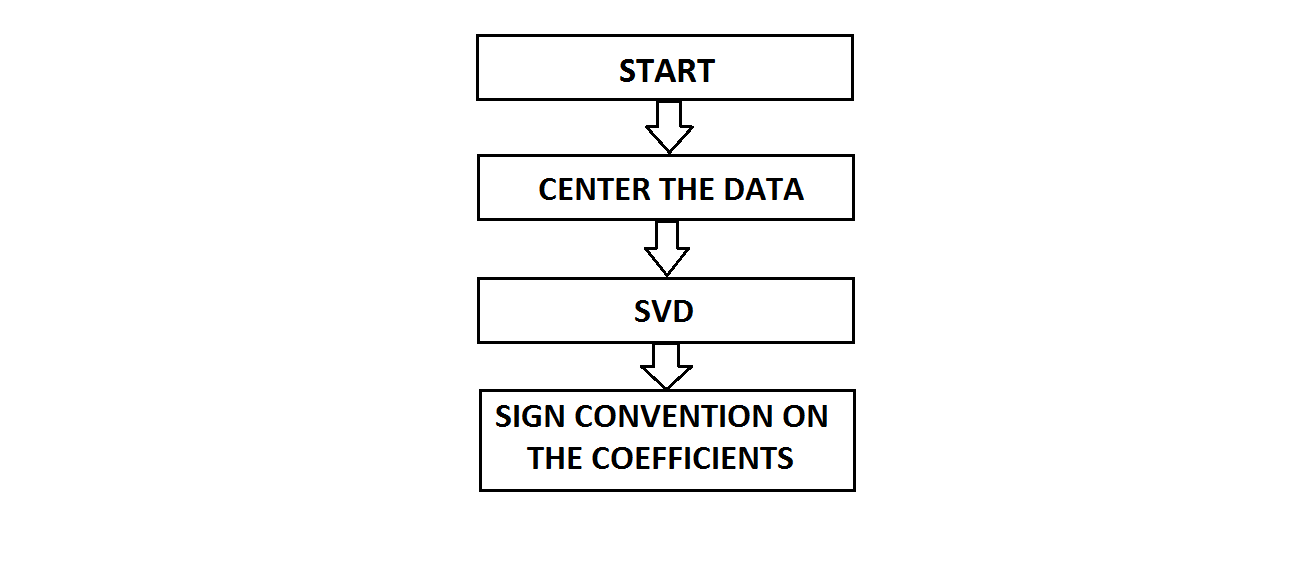
**Wstęp do matlabowskiej wersji algorytmu**

A Matlab pca function (standard library) is a reference implementation of this work. The implementation in CUDA bases on a code of this method. In a default mode singular value decomposition algorithm is used. An ‘economic’ version is implemented as it is more efficient and the obtained result is enough to continue executing the algorithm. It means that when singular value decomposition (A = U\*S\*VT) is calculated no columns of matrix U are computed and only the first min(m,n) rows of VT are computed.

It is worth mentioning that non-economic version might not run successfully on many GPUs as it requires a lot of memory for big datasets.

**Diagram / schemat algorytmu**

A diagram of the implemented algorithm in CUDA platform is illustrated below:

The first step (but might be optional) in the PCA method is centring 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).

**AN IMPLEMENTATION DESCRIPTION**

A detailed description of the implemented code goes step by step from the first point of the algorithm up to the end. Later a part about memory transfer is included

The aim of this work was to achieve efficient implementation of pca method for specific problem – processing fMRI data. Therefore the code was optimized for dimensions of matrices where number of rows is much bigger than number of columns (m >> n). It might not be efficient on a data with another ratio of dimensions.

Initially the data form a matrix, where each column contains all voxels from one slice. So the number of rows m equals x\*y\*z where x,y,z are voxel’s dimensions, and the number of columns equals the number of voxels (or simply time series). The data are stored in “column-major” order.

1.2 Centering the data

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

When reading or writing to global memory in CUDA, it is highly recommended that access to the memory is coalesced. As one block processes one row, threads from the block read successive elements from appropriate row. Because the data are stored in column-major, program transposes the matrix so that global memory access is coalesced, which is very important for efficiency in case of sum reduction. Before calculating singular value decomposition, program transposes the matrix again to restore its initial form.

For sum reduction, a function using shuffle instruction was implemented. Although this version did not result in better performance (regarding to basic function with shared memory) still it needs some work to be optimized. Shuffle instruction is a powerful tool for sharing data between threads (available in GPU with Kernel architecture) and has some advantages over shared memory. It introduces a way to directly share data between threads that are part of the same [warp](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture). On Kepler, threads of a warp can read each others’ registers by using a new instruction called SHFL, or “shuffle”. It can be very useful in the future work or optimization for already written methods.

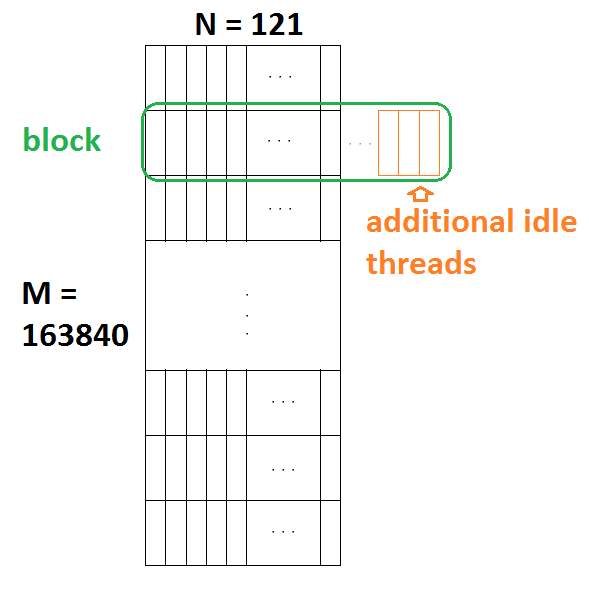
When running kernel program two configuration options need to be define by a programmer. Number of threads per block (three dimensions – x,y,z) and number of blocks in a grid (also three dimensions). Dependent on a problem size and “data shape” we can tune this options to get better performance. For example the number of threads per block should be a multiple of 32 even if the number of elements in a columns is different (smaller). It complicates a bit code (if threads instructions include reading or writing to the global memory basing on a thread index, it is necessary to check whether it exceeds an array length) but makes CUDA program being executed faster.

In case of fMRI data there is a large discrepancy between dimensions of the formed matrix. For instance in a data 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 task.

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.

Below a scheme with a matrix of data and CUDA block threads operating on a single row is illustrated:



The number of threads in the block is a multiple of 32. If it is not equal to the number of columns, then program must check index of the each thread (because they read and write to the global memory) and dependent on it continue processing or do nothing. Simultaneously 65535 blocks are working (which for many CUDA architectures is a limit). So loop is used to cover all the rows.

**1.3 SVD**

The most important (from the computational complexity and cost standpoints) 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 the ‘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. Fortunately the nifti data are stored in that way, so there is no need to transform the data.

However CULA library has not been developed since 2013, so it does not take advantage of the features new CUDA release offers. This might be a reason for writing our own implementation of singular value decomposition, but regarding available time, for this study was used the one from CULA library.

**1.4 Sign convention on the coefficients**

This task is quite similar to centring the data but in reverse. We need to span blocks of threads to process columns, not rows.

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 because the maximum number of threads per block is 1024.

The solution then is to have multiple blocks per column each one processing another part of the column. Although threads among one block can communicate each other easily, communication between blocks is very limited. In fact there are two options. One is to write results to the global memory and run as many as necessary kernel functions one after another. The second possibility is to use atomic operations. However this can be very complicated.

In the program the first option was implemented. There are multiple blocks per column each one processing its own part of the column, and writing a result to an intermediate array. This is functionality done by first kernel function. Second kernel function takes the intermediate array as an argument, 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.

This time we configure kernel’s grid to be two dimensional (see figure below describing the method). In the process of centring the data, the length of a block is large enough to cover the whole row. So we have only “a column” of blocks. In this case we have “a matrix” of blocks (which in turn covers all rows). Likewise in centring the data, a loop is necessary to process the whole length of columns.

Again when searching for a maximum element (first kernel) the global access memory is coalesced.

In order to do it in two kernel functions, the length of the intermediate array must be equal or smaller than 1024\*number of the columns. 1024 is the maximum number of threads per block and finally intermediate results from one column must be processed by one block (threads among a block can communicate each other).

Another recommendation when implementing in CUDA, is using multiplication in place of conditional branches (if else instructions). If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path [1].

For example when defining the sign of the largest element, instead of the following conditional if instruction:

int sign = 1;

if (max\_element < 0)

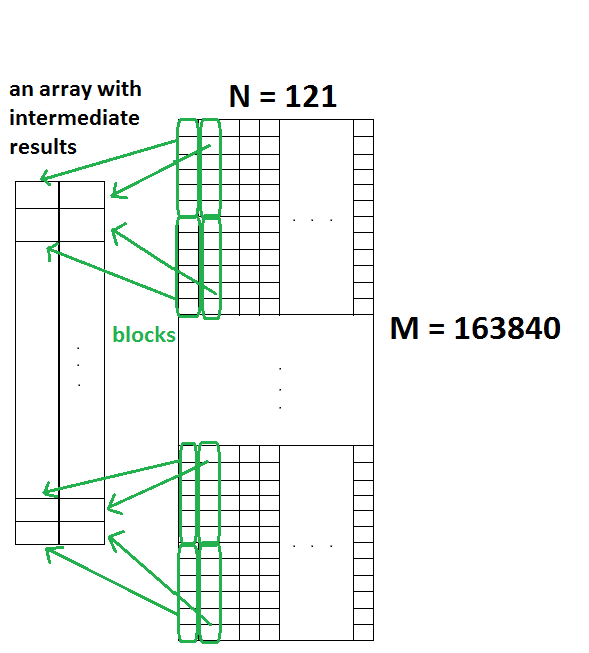
sign = -1;

we can write:

int result = max\_element >= 0;

int sign = (result == 0)\*(-1) + (result > 0);

Below a scheme with a matrix of data and CUDA block threads operating on a part of a column (first kernel program) is illustrated:



Blocks of threads go throughout the column in a loop writing to the array (or matrix) with intermediate results. In the second dimension there are more blocks than N (again it is a multiple of 32) so only one loop is needed.

1.1 Memory transfer

Memory transfer on CUDA platform is a time-consuming bottleneck. To mitigate this problem one can generate data directly on GPU memory but it is not always possible. Another possibility is a pinned memory. By 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, transfering them by 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).

**Performance comparison between CUDA and Matlab**

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 executed on Intel Core i3-2120 CPU 3.3 GHz processor with 8 GB RAM.

Data samples were obtain from www.openfmri.org/dataset/ds000105/.

Both programs were executed ten times for a sample of fMRI data and an average execution time was calculated. The table below shows results of execution time and speed-up. CUDA execution time is a total execution time of function called on a host runPCA() – it is a CPU function, which performs memory transfer from the host to the device and calls appropriate CUDA kernel functions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data sample | Sample 1 | Sample 2 | Sample 3 | Sample 4 | Sample 5 |
| CUDA execution time [ms] | 2004 | 2241 | 2173 | 2112 | 2095 |
| Matlab execution time [ms] | 7680 | 8550 | 8102 | 7953 | 7891 |
| Speed-upb | 3.83 | 3.81 | 3.72 | 3.76 | 3.76 |

Generally CUDA version program was executed 3.7-3.8 times much faster than Matlab script. In CUDA program data was transferring by pageable memory, so it could be even faster including pinned memory. Nevertheless speed-up on CUDA platform is usually much bigger, but more work need to be done to optimized the code (or implement new methods).

JAKI UDZIAŁ CZASOWY MA KAŻDY Z PUNKTÓW ALGORYTMU