MDPI

Article

Fast-GPU-PCC: A GPU Based Technique to Compute Pairwise Pearson's Correlation Coefficients for Big fMRI Data

Taban Eslami [1] and Fahad Saeed [1,*]

[1] Department of Computer Science, Western Michigan University, 49008-5466, Kalamazoo, MI, USA; taban.eslami@wmich.edu (T.E.)

* fahad.saeed@gmail.com; Tel.: +1- 269-276-3156

Academic Editor: name
Version February 18, 2018 submitted to MDPI

Abstract: Functional Magnetic Resonance Imaging (fMRI) is a non-invasive brain imaging technique which is heavily used for studying brain's functional activities in the past few years. A popular measure for capturing functional connectivities between brain regions is Pearson's correlation coefficient. fMRI data consists of huge number of small elements called voxels. Computing pairwise correlation coefficient between them using traditional CPU based technique is time consuming. In this paper, we propose a GPU-based algorithm called Fast-GPU-PCC for computing pairwise Pearson's correlation coefficient. Based on symmetric property of Pearson's correlation, this approach returns $N(N-1)/2$ correlation coefficients located at strictly upper/lower triangle part of correlation matrix. Storing correlations in an 1-dimensional array with the order as proposed in this paper is useful for further usage. Our approach is based on matrix multiplication and reordering its result which is performed on GPU. We performed some experiments on real and synthetic fMRI data for different number of voxels and varying length of time series. The proposed approach outperformed two other GPU-based techniques as well as the sequential version of computing correlation coefficient on CPU. We show that Fast-GPU-PCC runs 62.2 $\times$ faster than CPU-based version and 2.21 $\times$ and 4.05 $\times$ faster than two other GPU-based techniques. The implementation of our approach will be available as GPL license on GitHub portal of our lab (https://github.com/PCDS) after the paper is accepted for publication.

Keywords: fMRI, Pearson's correlation coefficient, GPU, CUDA, matrix multiplication

1. Introduction

Functional Magnetic Resonance Imaging (fMRI) is a non-invasive brain imaging technique which is used by researchers in order to study functional activities of the brain.[1]. Using this technology, many facts about the brain are discovered based on Blood Oxygen Level Dependent (BOLD) contrast. Analyzing fMRI data using machine learning techniques for discovering hidden patterns and early-stage detection of several brain-related diseases has gained significant attention among fMRI researchers [2,3]. During an fMRI session, a sequence of images are taken by a scanner through time while subject performs one or more tasks (task based fMRI) or the subject just rests without falling asleep (resting state fMRI). fMRI data consists of several thousands or millions of very small cubic components called voxels. Each voxel is the smallest addressable element of the brain and houses millions of neurons inside it. Hemodynamic changes inside the brain are revealed as intensity changes of the brain voxels.[4]. By keeping track of intensity of each voxel over time, a time series is extracted out of each voxel which is used for further analysis. A popular technique for analyzing

brain functional connectivity is Pearson's correlation coefficient (PCC)[5–7]. PCC computes linear association between two variables x and y using the following formula:

$$\rho_{xy} = \frac{\sum_{i=1}^{T}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{T}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{T}(y_i - \bar{y})^2}} \tag{1}$$

The value of Peasrson's Correlation Coefficient ρ_{xy} can be in range -1 and 1 [8]. Value of -1 indicates perfect negative linear relationship, 0 indicates no linear relationship and +1 shows perfect positive linear relationship among two variables. In this equation x and y correspond to two T dimensional variables. Considering fMRI data, x and y are two individual voxels each having T data points in their time series. Pairwise Pearson's correlation computation is computationally intensive for large datasets like fMRI images so using parallel computing techniques becomes necessary. Several parallel computing based approaches have been proposed in order to accelerate the PCC computation. One of these approaches is a GPU based approach proposed by Gembris et al. [9]. They reformulated the Pearson's correlation equation in order to minimize the number of necessary divisions as follows:

$$\rho_{xy} = \frac{T\sum_{i=1}^{T} x_i y_i - \sum_{i=1}^{T} x_i . \sum_{i=1}^{T} y_i}{\sqrt{T\sum_{i=1}^{T} x_i^2 - (\sum_{i=1}^{T} x_i)^2}\sqrt{T\sum_{i=1}^{T} y_i^2 - (\sum_{i=1}^{T} y_i)^2}} \tag{2}$$

Wang et al.[6] proposed a parallel technique based on a controller worker method with Message Passing Interface (MPI) to compute pairwise Pearson's correlations over multiple time windows. Another approach was proposed by Liu et al.[10] to compute all pairwise correlation coefficients on Intel Xeon Phi clusters.

Pearson's correlation has symmetric property (($corr(x, y) = corr(y, x)$)). Based on this property all pairwise correlations among N elements can be represented by an array of $N(N-1)/2$ elements instead of N^2 elements. Each element of this array is the correlation among two distinct variables i and j. The correlation array may contain all correlations in strictly upper or lower triangle part of the correlation matrix. Elements on the main diagonal are discarded since they only show the correlation of each element with itself which is always one. An example of desired elements of the correlation matrix, resulting correlation array and two possible orders of storing correlation values is shown in Fig. 1.

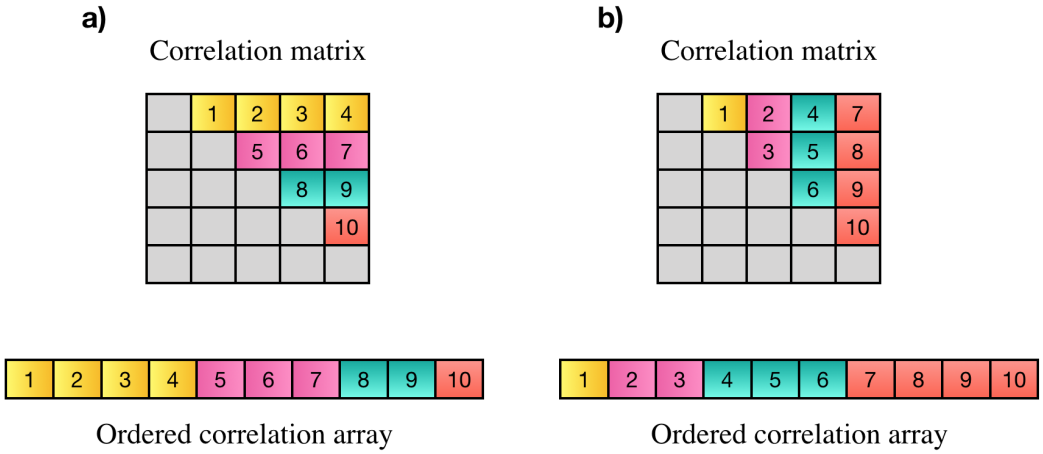

Figure 1. a and b are examples of two possible orders for Pearson's correlations in correlation matrix and their resulting correlation array. In part a, the first $N-1$ elements of the array show the Pearson's correlations between the first variable and all other variables, the next $N-2$ elements show the correlation of the second variable with all others and so on. In part b, Last N-1 elements show correlation of the last element with the rest of elements, N-2 elements before them show the correlation of the $N-1$th element to the rest of elements and so on.

In [11], a GPU based tool was developed by Liang et al for constructing gene co-expression networks based on computing N(N-1)/2 Pearson's correlation coefficient. Wang et al [12] proposed a hybrid CPU-GPU framework for computing Pearson's correlations Based on General Matrix multiplication (GEMM). Their approach is based on the fact that Pearson's correlation computation among two voxels can be reduced to vector dot product of their time series if each time series is normalized based on the following equation:

$$u_i = \frac{v_i - \bar{v}_i}{\|v_i - \bar{v}_i\|_2} \tag{3}$$

In this equation v_i is time series of voxel i and u_i is the normalized time series of v_i. All normalized voxels are then aggregated in matrix$U(u_1, u_2, ..., u_n)$. The correlation matrix can be constructed by multiplying matrix U to its transpose ($U \times U^T$). Sometimes the size of correlation matrix is larger than GPU memory, in this case, their approach divides matrix U to smaller blocks and computes the multiplication of each block to others to cover all elements in upper triangle. After performing matrix multiplication of all blocks a post processing step is needed to reorder the elements and eliminate redundant correlations. This post processing runs on CPU.

In [13] we proposed two GPU based approaches to compute $N(N-1)/2$ Pearson's correlation based on the order shown in part a of Fig. 1. In the first approach, after normalizing the data using equation 1, correlations of each voxel with the rest of voxels are computed by multiplying its time series to a matrix containing the time series of all other voxels. This process which is multiplying a vector to a matrix is continued for all voxels. The matrix that is multiplied to the time series of first voxel contains $N-1$ rows, for second voxel, matrix contains $N-2$ rows since correlation of first and second voxels have been computed by first matrix vector multiplication. Reducing the size of the matrix by one for each voxel, by performing $N-1$ matrix vector multiplication, upper triangle part of correlation matrix (part a of Fig. 1.) is computed. Second approach is called GPU-PCC and is based on performing vector dot product of normalized time series. In this technique each 16 consecutive GPU thread are considered as a group and are responsible for performing vector dot product of two normalized time series which results in computing correlation between two voxels. In order to compute correlation coefficients in desired order, threads inside each group use the following mapping equations based

on index of the group (k) to compute the index of two voxels (i and j) that their correlation should be stored at location k. Using these equations assures that correlations are computed in order.

$$i = n - 2 - \left\lfloor \frac{\sqrt{-8 \times k + 4 \times n \times (n-1) - 7}}{2} - 0.5 \right\rfloor \tag{4}$$

$$j = k + i + 1 - \frac{n \times (n-1)}{2} + \frac{(n-i) \times ((n-i)-1)}{2} \tag{5}$$

If total size of correlation matrix is larger than GPU memory, this approach computes correlations until there is no free space in GPU, transfers the results to CPU and starts computing the rest of correlations. Our experiments on synthetic and real fMRI data showed that this approach can compute correlations faster than the first other approach.

1.1. GPU architecture, CUDA programming model and cuBLAS library

Processing huge amount of data generated by fast and high-throughput instruments in the area of Bioinformatics, biomedical and health-care informatics is almost impossible using traditional and sequential CPU based techniques. Many algorithms based on parallel computing techniques have been proposed in different fields like Genomics, proteogenomics, clinical informatics, imaging informatics etc. [14–24]. Using Graphics Processing Unit (GPUs) for accelaring these type of problems has become very popular recently. The very first goal of GPUs was satisfying demands for higher quality graphics in video games and creating more relalistic 3D environment[25]. Nowadays, multitude of high-performance applications exploit high throughput of enormous number of GPU cores.[24,26]. A GPU consists of an array of streaming multiprocessors (SMs) each having multiple streaming processors or cores. On each core hundreds of threads run based on SIMT (Single Instruction Multiple Thread) strategy. A warp is a group of 32 threads that follow the same execution path and run on at the same time on the same SM. CUDA or Compute Unified Device Architecture is NVIDIA's programming model interface created for programming graphic cards. The function that executes by GPU threads on GPU device in parallel is called kernel function. Parallel invocations of kernel are grouped into blocks. A Block is a programming abstraction used by programmer to group a number of threads for running kernel. Maximum number of threads per block is 1024 which can be organized in 3 dimensions. Multiple blocks can be organized in one or two dimensions to form a grid. GPU contains different memory types such as global memory, shared memory, local memory and registers. Global memory is the main memory of GPU which is accessible by all threads. Data transferred from CPU to GPU resides on global memory. Shared memory on the other hand is on-chip memory which is shared among all threads within the same block and is not accessible by threads in other blocks. Accessing data from shared memory is much faster than global memory and is efficient in cases that threads need to access data more than once.

Nividia has provided efficient CUDA libraries such as CUDA Basic Linear Algebra Subroutines (cuBLAS). which performs vector and matrix operations like matrix multiplication and matrix vector multiplication[27]. In this study we used a built in function from this library which is very efficient for performing matrix multiplication.

2. Materials and methods

As stated in the previous section, multiplying two vectors which are normalized by equation 3 results in Pearson's correlation between them. Normalizing all time series vectors takes much less time compared to multiplying pairwise time series, so we leave the normalization part to be performed on CPU. For the rest of the paper lets assume time series of all voxels are stored in an $N \times M$ matrix called U, which N corresponds to the number of voxels and M corresponds to the number of data points of each voxel (length of time series).

After data is normalized on CPU, it is transfered to GPU global memory. Since the number of voxels

are much more than the length of time series of each voxels, size of correlation matrix is very large and sometimes cannot be fitted inside GPU memory. In this case, correlation computation must be performed in multiple rounds such that in each round part of correlation coefficients should be calculated and transferred to CPU memory to free GPU space for the rest of computation. Additionally, our approach needs some extra space in GPU for storing reordered coefficients before transferring them back to CPU. If the total space that our algorithm needs is smaller than whole GPU memory, the algorithm can be run in one round, other wise, multiple rounds are needed for completing the computations. In the next following sections, we first explain how to compute the space we need for computing correlation coefficients and reordering them inside GPU, then we go through two possible cases in which pairwise correlations can be computed in one round or several rounds.

2.1. Space storage needed for computing correlations and reordering them

Our approach is based on performing matrix multiplication and extracting the upper/lower triangle part. Multiplying matrix U ($N \times M$) to its transpose ($M \times N$) generates N^2 Coefficients. Upper/lower triangle part of the correlation matrix can be stored in an array with length $\frac{N(N-1)}{2}$. Normalized time series of voxels are transformed to GPU memory in the beginning of the algorithm and will stay there during the whole process. This will take an additional $N \times M$ space. So the total space needed for storing data, computing the correlation matrix and reordered correlation array in GPU is $N^2 + \frac{N(N-1)}{2} + NM$. If this value is smaller than GPU memory the whole computation can be done in one round, otherwise we first compute correlation of a block of data with B voxels to all other voxels, reorder and transfer them back to CPU and start a new block. Space needed for multiplying time series of B voxels to the rest of voxels is NB and extracted correlations belonging to upper triangle part of the correlation matrix corresponding to B blocks needs $NB - \frac{B(B+1)}{2}$. Fig. 2 shows an example of these elements.

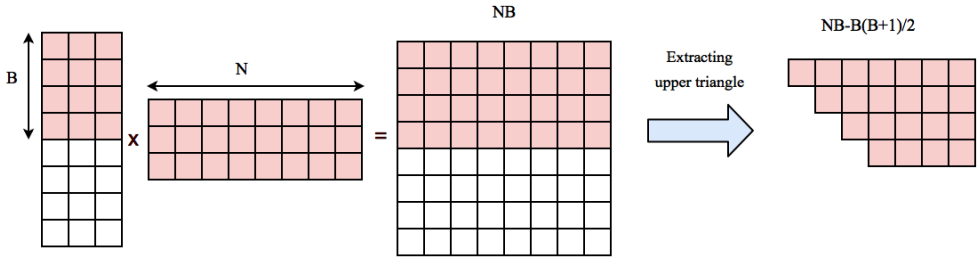

Figure 2. Space needed for computing correlation of first B voxels with the rest of voxels. Pairwise correlation is computed by multiplying a matrix containing time series of B voxels to a matrix containing time series of all voxels which results in a matrix containing $N \times B$ elements. This matrix has $NB - B(B+1)/2$ distinct correlation coefficients that need to be extracted and stored in resulting correlation array.

The total space needed for the computation is equal to $NM + NB + NB - \frac{B(B+1)}{2}$. The value of B should be chosen in such a way that the space needed for our computation is less than the free space in GPU memory at the time. Let's assume normalized time series of all voxels are already stored in GPU memory and the free space left is equal to X. Since the value of $NB - \frac{B(B+1)}{2}$ is smaller than NB, the upper bound of space we need is 2NB and value of B can be computed using the following equation

$$B = \frac{X}{2N} \tag{6}$$

We compute the value of B in the beginning of our algorithm, if this value is greater than N, it means that the computation can be done in one round otherwise several rounds are needed for computation. In the next two subsections, we go through each case in detail.

2.2. Case 1: Correlation computation can be done in one round

If GPU has enough memory to store the whole correlation matrix and ordered correlation array, by multiplying matrix U to its transpose the whole correlation matrix is computed at once and we can extract the upper/lower triangle part of the matrix. The idea that we used for extracting the upper/lower triangle part is to assign one GPU thread to each cell of correlation matrix, if the cell is located in upper/lower triangle above/below the diagonal, thread will copy its value to specific location in correlation array. Index of each thread can be computed based on its thread ID, block ID and dimension of the block as follows:

$$idx = blockDim.x * blockIdx.x + threadIDx.x \tag{7}$$

After computing this index which is unique for each thread, we compute the row and column index of the cell that the thread is assigned to it. Row index and column index of each cell can be computed as quotient and remainder of dividing idx by n , $i = idx/n$ and $j = idx\%n$ respectively. i and j are indices of voxels which their correlation is stored at index (i,j) of the correlation matrix. In order to take the elements in upper triangle part of the matrix, elements with $i < j$ are selected and in order to take the lower triangle part elements with $j < i$ are selected. Threads which are pointing to upper triangle part of the correlation matrix will save their corresponding correlations at index k of resulting correlation array which can be calculated as follows:

$$k = i \times n - \frac{i \times (i+1)}{2} + j - i \tag{8}$$

Using this equation, coefficient will be saved in correlation array based on the pattern showed in part a of Fig. 1. (row major order). In order to save correlation based on part b (column major order), the following equation is used:

$$k = \frac{i \times (i+1)}{2} + j - i \tag{9}$$

Fig. 2. shows an example of extracting upper triangle part for a 5×5 correlation matrix and algorithm 1 shows the psudocode of this process.

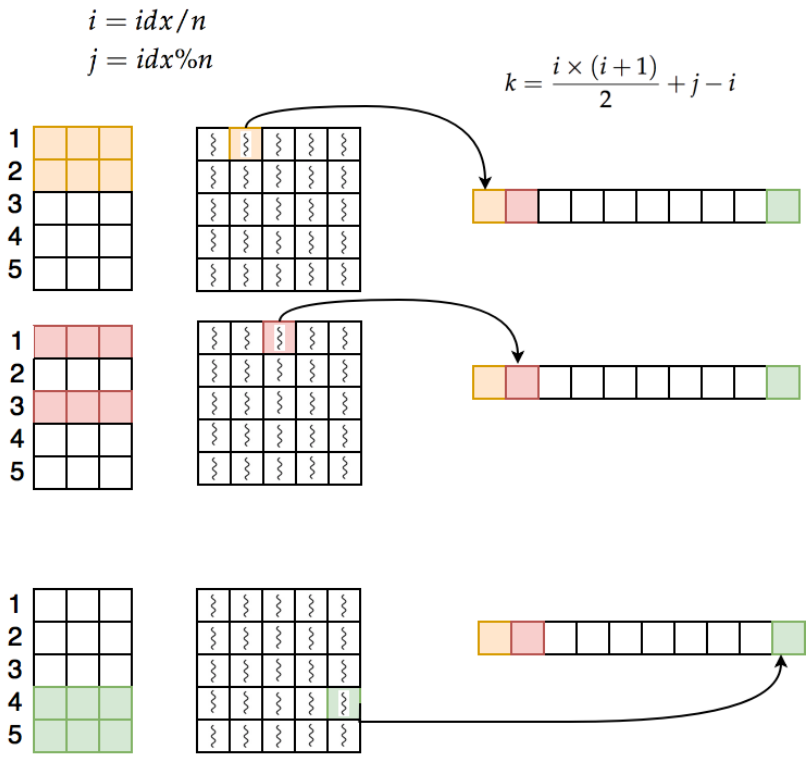

Figure 3. Process of extracting upper triangle part of correlation matrix based on algorithm 1

The psudocode of reordering kernel is shown in Algorithm 1. After computing correlations and

Algorithm 1 Extracting ordered upper triangle part of correlation matrix

Input: $N \times N$ correlation matrix S
Output: Ordered correlation array C of size $N(N-1)/2$

1: $idx = blockDim.x * blockIdx.x + threadIdx.x$
2: $i = idx/n$
3: $j = idx\%n$
4: **if** $i < j$ and $i < N$ and $j < N$ **then**
5: $\quad k_1 = i \times n - \frac{i \times (i+1)}{2} + j - i$
6: $\quad k_2 = j \times n + i$
7: $\quad C[k_1 - 1] = S[k_2]$
8: **end if**

storing distinct pairs in correlation array, it will be copied to CPU memory.

2.3. case2: Correlation computation needs to performed in multiple rounds

In cases that both correlation matrix and resulting array cannot be fitted inside GPU memory, the correlation of the first B voxels (B is computed using equation 6) to the rest of voxels are computed and reordered using algorithm 1. Results are transferred back to the CPU. A new block number should be calculated for computing the rest of coefficients. Since the correlation of the first B voxels with the rest of voxels are computed, new block number can be calculated using equation 6 but this time using N-B instead of N in denominator. By doing this process all correlation coefficients can be computed in multiple rounds.

2.4. Overall algorithm

Considering both cases, algorithm 3 shows the overall scheme of our proposed method. Data is

Algorithm 2 Fast-GPU-PCC

Input: $N \times M$ matrix U of time series data **Output:** Correlation array C of size $N(N-1)/2$

1: Preprocess the fMRI data using equation 3
2: Copy normalized data to GPU global memory
3: $B = X/2N$
4: **if** $B > N$ **then**
5: Multiply matrix U to its transpose U^T
6: Extract upper/lower triangle part of the matrix using algorithm 1
7: Transfer the correlation array to CPU
8: **else**
9: $flag = 1, i = 0, N' = N$
10: **while** $i < N$ **do**
11: Multiply rows i to $i + B$ of matrix A to columns i to N of U^T
12: Extract the upper/lower triangle using algorithm 1
13: Transfer the extracted correlations to CPU
14: $i = i + B$
15: $N' = N' - B$
16: $B = X/2N'$
17: **if** $B > N'$ **then**
18: $N' = B$
19: **end if**
20: **end while**
21: **end if**

preprocessed and copied to GPU memory (lines 1,2). Lines 4 to 7 runs when the total computation can be done in one round as explained in section 2.2. Lines 9 to 20 runs when computation cannot be done in one round (section 2.3). In this case correlation of B voxels (B is computed in line 3) to the rest of voxels are computed, reordered and copied back to CPU. In line 16, new size of B is computed using equation 7 this time ignoring the first B voxels. A new variable called *N_prime* stores the number of remaining voxels that their pairwise correlations to the rest of voxels should be computed. If block size B is greater than N', shows the case that pairwise correlation of the rest of elements can be done in one round, otherwise this process should be continued for more rounds. The overall process of this algorithm is shown in in Fig. 4.

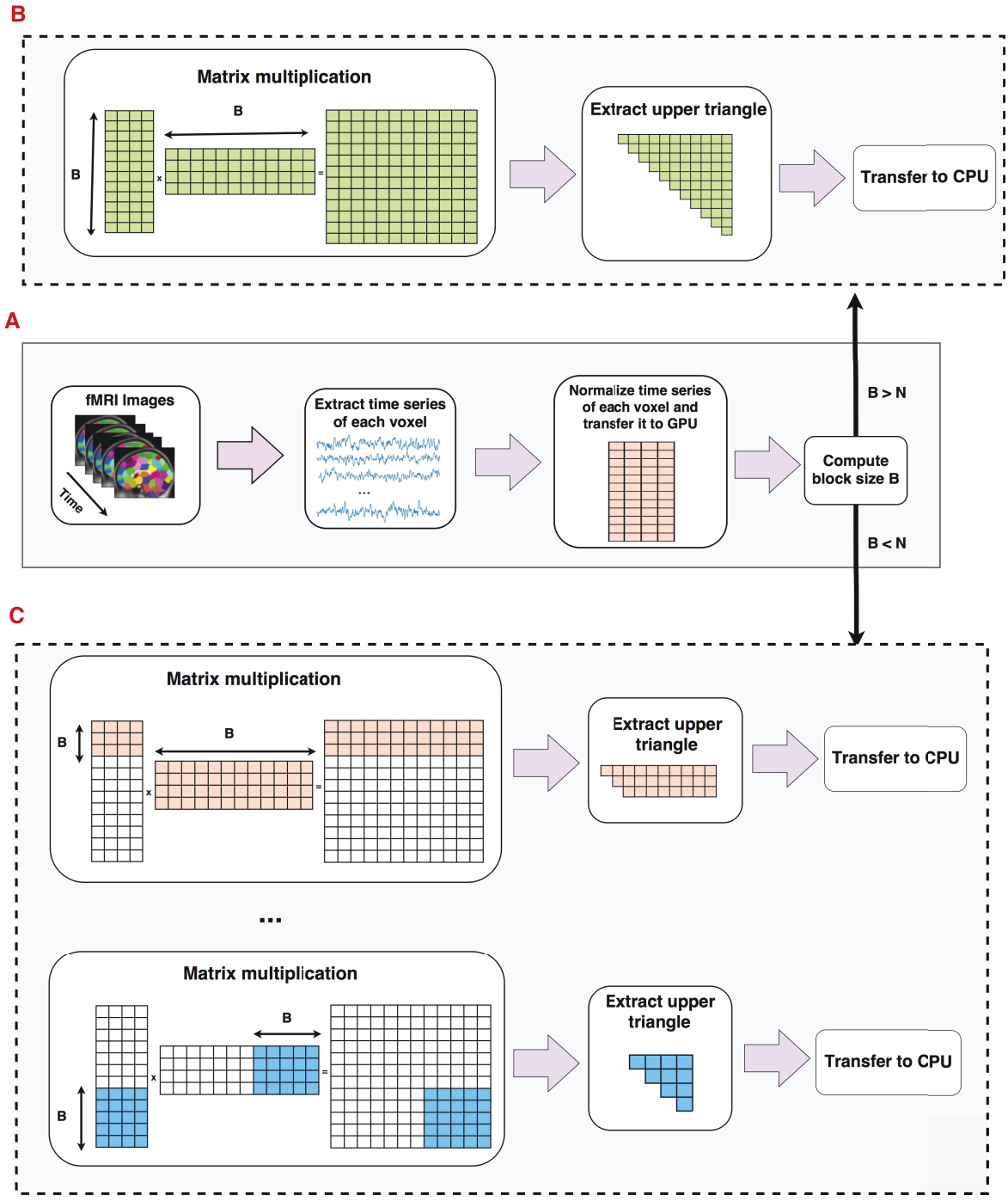

Figure 4. Overall process of Fast-GPU-PCC. In part A, fMRI time series is normalized in CPU and transferred to GPU memory. Block size B is computed using equation 6. If B is larger than N means that the whole computation can be performed in one round which is shown in part B. In part B the whole normalized matrix is multiplied to its transpose and upper triangle is extracted and transferred back to CPU. If block size is computed in part A is smaller than N means that only pairwise correlation of B voxels with the rest of voxels can be computed which is shown in part C. In part C after correlation of the first B voxels with the rest of voxels is computed and transferred back to CPU, new block size is computed and this process is repeated multiple time until all pairwise correlations are computed.

3. Experiments and results

All the experiments reported in this section are performed on a Linux server with Ubuntu Operating System version 14.01. The server consists of two Intel Xeon E5 2620 processors with clock speed 2.4 GHz, 48 GBs RAM and NVIDIA Tesla K40c Graphic Processing Unit. This GPU contains 15 Streaming Multiprocessors each consists of 192 CUDA cores and 11520 MBytes global memory. We have compared our method with three other methods. The first method is sequential version of computing Pairwise Pearson's correlation coefficient. The second method is GPU-PCC [13] algorithm which is a GPU based technique able to compute Pearson correlations in order and the third method is proposed by Wang et al. [12]. In Wang's method they compute pairwise correlations by performing matrix multiplications on GPU multiple times and in order to reorder the correlation coefficients and eliminate redundant ones, the results is post processed on CPU. We considered the time of both matrix multiplication and post-processing steps. All the experiments for each dataset are repeated multiple times and the minimum running time is reported. Optimization level O2 was used for compiling codes that run on CPU. We compared the scalability of our method with other methods by increasing the number of voxels and increasing the length of time series. The following sections explain the experiments in more details.

3.1. Increasing number of voxels

Today fMRI scanners are able to provide high resolution images in which we are dealing a huge amount of voxels. To assure our method is able to handle large number of voxels we performed an experiment considering different number of voxels from 20000 to 100000 each having a time series of length 100. We used synthetic dataset for this experiment. For each voxel, we generated a vector of 100 uniformly random floating point numbers in range -2 and 2 as intensity of each voxel. Table 2 shows the running time of each method based on different number of voxels in seconds. We also plotted the running times of all GPU-based techniques in Fig. 5 and compared the running time of Fast-GPU-PCC and sequential version in Fig. 6 (we used a different figure for this comparison since having sequential version with other techniques in the same figure makes comparison of GPU based techniques difficult). As we see in figures, Fast-GPU-PCC runs faster than other techniques for all values of N. Speedup of Fast-GPU-PCC compared to other techniques are shown in table 2. The speed up over CPU version, GPU-PCC and Wang's technique is about 30, 2 and 3 times respectively.

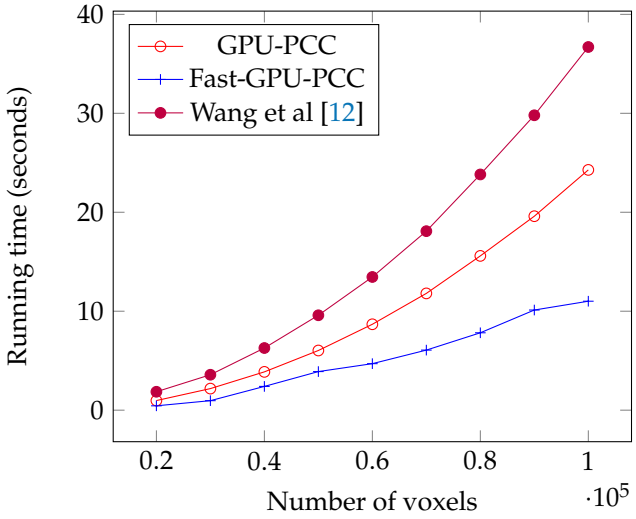

Figure 5. Running time comparison of Fast-GPU-PCC with other GPU-based techniques

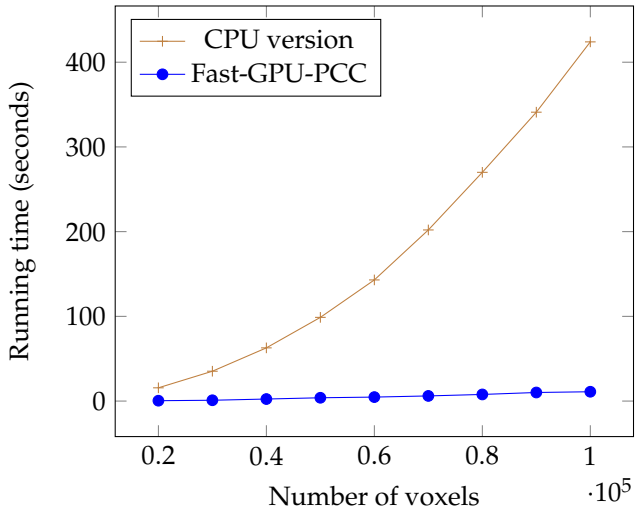

Figure 6. Running time comparison of Fast-GPU-PCC and CPU version

Table 1. Comparing running time (Seconds) of different approaches on synthetic fMRI data

Number of voxels (N)	GPU-PCC	Fast-GPU-PCC	Wang et al [12]	CPU-version
20000	0.97	0.44	1.86	15.65
30000	2.18	0.96	3.58	35.23
40000	3.87	2.40	6.28	62.81
50000	6.03	3.2	9.59	98.8
60000	8.68	4.7	13.46	143
70000	11.8	6.07	18.09	202
80000	15.59	7.82	23.82	270
90000	19.6	10.12	29.8	341
100000	24.28	11.01	36.7	424

Table 2. Speed up gained by Fast-GPU-PCC over other methods by increasing the number of voxels

Number of voxels (N)	GPU-PCC	Wang et al [12]	CPU-version
20000	2.22	4.22	35.56
30000	2.27	3.7	36.69
40000	1.5	2.61	26.17
50000	1.88	2.99	30.8
60000	1.84	2.86	30.42
70000	1.94	2.98	33.27
80000	1.99	3.04	34.52
90000	1.99	2.94	33.69
100000	2.2	3.33	38.54

3.2. Increasing the length of time series

We performed another experiment to measure the running time of our approach by increasing the length of time series. The data that we used in this section is also synthetic data. To observe

how increasing the length of time series affect the running time, we performed our experiment by considering fixed 60000 voxels and each time changed the length of time series. We measured the running time for 50, 100, 200, 300, 400 and 500 time point in each time series. Similar to last section, uniformly random floating point number in range -2 and 2 is used as intensity of each voxel.

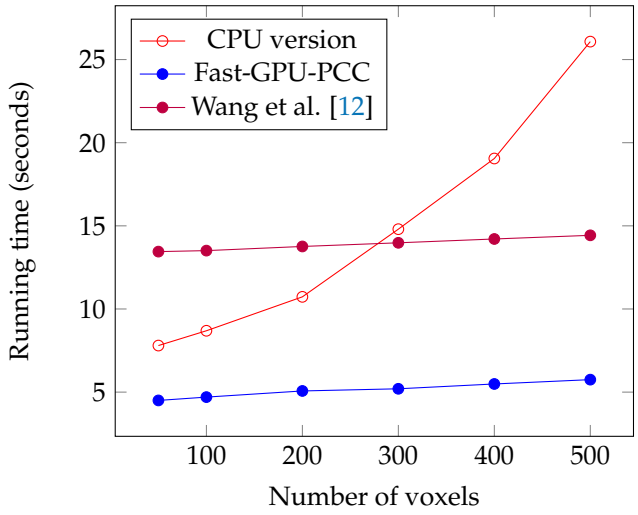

Figure 7. Running time comparison of Fast-GPU-PCC with other GPU-based techniques

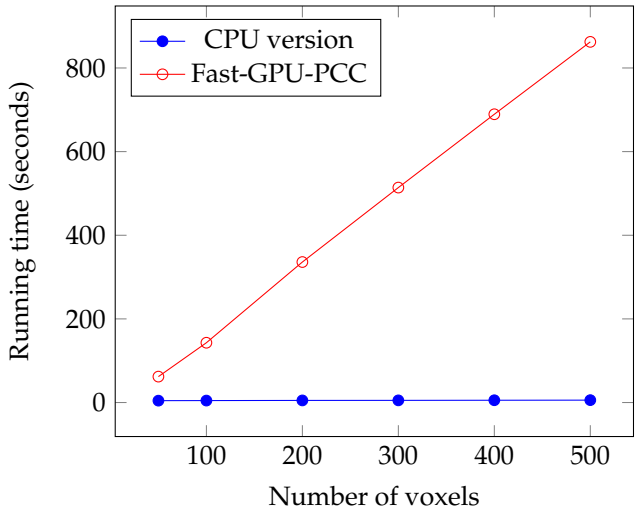

Figure 8. Running time comparison of Fast-GPU-PCC with other GPU-based techniques

Table 3. Running time comparison of Fast-GPU-PCC with other GPU-based techniques

Length of time series (M)	GPU-PCC	Fast-GPU-PCC	Wang et al. [12]	CPU-version
50	7.81	4.06	13.45	62
100	8.68	4.7	13.43	143
200	10.73	5.07	13.76	335.97
300	14.8	5.4	13.98	514
400	19.05	5.67	14.21	689.329
500	26.8	5.89	14.43	862.417

Table 4. Speed up gained by Fast-GPU-PCC over other methods by increasing length of time series

Length of time series (M)	GPU-PCC	Wang et al. [12]	CPU-version
50	1.92	3.31	15.27
100	1.84	2.85	30.42
200	2.11	2.41	66.26
300	2.74	2.58	95.18
400	3.35	2.5	121.57
500	4.55	2.44	146.34

Like increasing the number of voxels, Fast-GPU-PCC runs faster than other techniques by increasing the length of time series. Speed up over Wang's technique is about 2.5 times for all values of M. Speed up over GPU-PCC and CPU-version increases as we increase the length of time series. It starts from 1.92× and 15.27× for M = 50 and reaches to 4.55× and 146.34× for M = 500.

3.3. Experiment on real data

We performed another experiment on real fMRI data and measured the running time of all techniques. The dataset we used is called Orangeburg dataset [1]. It consists of resting state fMRI data of 20 healthy subjects, 5 male and 15 female with age range 20-55. We picked a random subject from this dataset for our experiment. Number of voxels in this dataset is equal to 90112 and length of time series is equal to 165. Table 5 and 6 show the running time comparison and speed up achieved by Fast-GPU-PCC over other methods.

Table 5. Running time comparison of Fast-GPU-PCC with other techniques on real fMRI data

Fast-GPU-PCC	GPU-PCC	Wang et al [12]	CPU-version
9.41	20.83	38.15	585.33

Table 6. Speed up gained by Fast-GPU-PCC over other methods on real fMRI data

GPU-PCC	Wang et al [12]	CPU-version
2.21	4.05	62.2

Similar to synthetic data, FAST-GPU-PCC runs faster than other techniques on real data. It runs 4.05× faster than Wang's technique, 2.21× faster than GPU-PCC and 62.2× faster than CPU-version.

4. Conclusion and future direction

Pearson's correlation coefficient is a very well used technique in fMRI data analysis for studying functional connectivities of the brain. fMRI images contain thousands of voxels and using traditional techniques for computing pairwise Pearson's correlation is very time consuming and not efficient. Therefore, using parallel computing techniques is essential for processing data- and compute-intensive operations like computing correlation for big brain research. Based on symmetric property of Pearson's correlation, the entire correlation matrix can be stored in an N(N-1)/2 array which stores the element in strictly upper or lower triangle of correlation. Storing correlations in this array with a meaningful order makes it easier for future applications. In this paper, we proposed a GPU-based technique called Fast-GPU-PCC which computes correlation coefficients and reorders them in two possible ways. Both

[1] www.nitrc.org/projects/fcon_1000/

correlation computation and reordering steps are performed on GPU. We used an efficient CUDA built-in function for performing matrix multiplication. The size of the correlation matrix usually exceeds the GPU memory specially for large datasets. Therefore, we performed the multiplication in multiple steps, where in each step, we multiply time series of a block of B voxels to the remaining voxels. The post processing step is performed right after each matrix multiplication, then we reorder the computed correlations and store them in the resulting correlation array. In order to compute the block size B, we considered both sizes of resulting matrix multiplication and the correlation array into account. We performed several experiments on synthetic and real fMRI data and compared it with two other GPU based technique and CPU-version of computing Pearson's correlation coefficient. During our experiments on synthetic data, we investigated the effects of increasing the number of voxels and length of time series on scalability of Fast-GPU-PCC. To see how scalable Fast-GPU-PCC is in terms of number of voxels, we began by using 20000 voxels and continued our process to 100000 voxels. Fast-GPU-PCC outperformed all other techniques for all sizes and achieved about 2× and 3× speed up compared to other GPU-based techniques and more than 30× compared to CPU-version. In another experiment, we checked the effect of increasing the length of time series on our approach by increasing it from 50 to 200. Fast-GPU-PCC out performed other techniques such that its speed up increased over CPU-version and one of GPU-based techniques and ran about 2.5× faster compared to another GPU-based technique when increasing the length of time series. Experiments on real data containing about 90000 voxels also showed promising result for Fast-GPU-PCC such that it ran 2× and 4× faster than other GPU-based techniques and 62.2× faster than CPU-version. Scalability of Fast-GPU-PCC shows that it can can be used in many fMRI-based applications. This technique has the flexibility to work with correlation coefficients inside the GPU memory, meaning that after reordering the elements and before transferring data to CPU any computation can be performed on coefficients using GPU threads, for example comparing it with a predefined threshold to detect activated voxels, performing compression techniques, etc.

For future direction of this study we focus our attention on dynamic functional networks which are becoming popular in fMRI studies. Constructing dynamic functional networks are very time consuming since pairwise correlations for multiple time windows should be computed so an efficient GPU based algorithm can accelerate this process significantly. We also focus on storage space of correlation array which is becoming challenging for large datasets specifically for the cases in which several correlation arrays need to be computed.

Acknowledgment

This material is based in part upon work supported by the National Science Foundation under Grant Numbers NSF CRII CCF-1464268 and NSF CAREER ACI-1651724. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We would also like to acknowledge the donation of a K-40c Tesla GPU from NVIDIA which was used for all GPU based experiments performed in this paper.

References

1. Craddock, R.C.; Tungaraza, R.L.; Milham, M.P. Connectomics and new approaches for analyzing human brain functional connectivity. *GigaScience* **2015**, *4*, 1.
2. Raschka, S.; Mirjalili, V. *Python Machine Learning*; Packt Publishing Ltd, 2017.
3. Hosseini-Asl, E.; Gimel'farb, G.; El-Baz, A. Alzheimer's Disease Diagnostics by a Deeply Supervised Adaptable 3D Convolutional Network. *arXiv preprint arXiv:1607.00556* **2016**.
4. Lindquist, M.A.; others. The statistical analysis of fMRI data. *Statistical Science* **2008**, *23*, 439–464.
5. Zhang, H.; Tian, J.; Zhen, Z. Direct measure of local region functional connectivity by multivariate correlation technique. 2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE, 2007, pp. 5231–5234.

6. Wang, Y.; Cohen, J.D.; Li, K.; Turk-Browne, N.B. Full correlation matrix analysis of fmri data. Technical report, Technical report, Princeton Neuroscience Institute, 2014.
7. Chang, C.; Glover, G.H. Time–frequency dynamics of resting-state brain connectivity measured with fMRI. *Neuroimage* **2010**, *50*, 81–98.
8. Lee Rodgers, J.; Nicewander, W.A. Thirteen ways to look at the correlation coefficient. *The American Statistician* **1988**, *42*, 59–66.
9. Gembris, D.; Neeb, M.; Gipp, M.; Kugel, A.; Männer, R. Correlation analysis on GPU systems using NVIDIA's CUDA. *Journal of real-time image processing* **2011**, *6*, 275–280.
10. Liu, Y.; Pan, T.; Aluru, S. Parallel pairwise correlation computation on intel xeon phi clusters. Computer Architecture and High Performance Computing (SBAC-PAD), 2016 28th International Symposium on. IEEE, 2016, pp. 141–149.
11. Liang, M.; Zhang, F.; Jin, G.; Zhu, J. FastGCN: a GPU accelerated tool for fast gene co-expression networks. *PloS one* **2015**, *10*, e0116776.
12. Wang, Y.; Du, H.; Xia, M.; Ren, L.; Xu, M.; Xie, T.; Gong, G.; Xu, N.; Yang, H.; He, Y. A hybrid CPU-GPU accelerated framework for fast mapping of high-resolution human brain connectome. *PloS one* **2013**, *8*, e62789.
13. Eslami, T.; Awan, M.G.; Saeed, F. GPU-PCC: A GPU Based Technique to Compute Pairwise Pearson's Correlation Coefficients for Big fMRI Data. Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics. ACM, 2017, pp. 723–728.
14. Luo, J.; Wu, M.; Gopukumar, D.; Zhao, Y. Big data application in biomedical research and health care: a literature review. *Biomedical informatics insights* **2016**, *8*, BII–S31559.
15. Vargas-Perez, S.; Saeed, F. A Hybrid MPI-OpenMP Strategy to Speedup the Compression of Big Next-Generation Sequencing Datasets. *IEEE Transactions on Parallel and Distributed Systems* **2017**.
16. Saeed, F.; Perez-Rathke, A.; Gwarnicki, J.; Berger-Wolf, T.; Khokhar, A. A high performance multiple sequence alignment system for pyrosequencing reads from multiple reference genomes. *Journal of parallel and distributed computing* **2012**, *72*, 83–93.
17. Awan, M.G.; Saeed, F. An Out-of-Core GPU based dimensionality reduction algorithm for Big Mass Spectrometry Data and its application in bottom-up Proteomics. Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics. ACM, 2017, pp. 550–555.
18. Saeed, F.; Hoffert, J.D.; Knepper, M.A. A high performance algorithm for clustering of large-scale protein mass spectrometry data using multi-core architectures. Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining. ACM, 2013, pp. 923–930.
19. Schatz, M.C. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics* **2009**, *25*, 1363–1369.
20. Pandey, R.V.; Schlötterer, C. DistMap: a toolkit for distributed short read mapping on a Hadoop cluster. *PLoS One* **2013**, *8*, e72614.
21. Lewis, S.; Csordas, A.; Killcoyne, S.; Hermjakob, H.; Hoopmann, M.R.; Moritz, R.L.; Deutsch, E.W.; Boyle, J. Hydra: a scalable proteomic search engine which utilizes the Hadoop distributed computing framework. *BMC bioinformatics* **2012**, *13*, 324.
22. Wang, S.; Kim, J.; Jiang, X.; Brunner, S.F.; Ohno-Machado, L. GAMUT: GPU accelerated microRNA analysis to uncover target genes through CUDA-miRanda. *BMC medical genomics* **2014**, *7*, S9.
23. Liu, Y.; Wirawan, A.; Schmidt, B. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC bioinformatics* **2013**, *14*, 117.
24. Eklund, A.; Andersson, M.; Knutsson, H. fMRI analysis on the GPU-possibilities and challenges. *Computer methods and programs in biomedicine* **2012**, *105*, 145–161.
25. Sanders, J.; Kandrot, E. *CUDA by example: an introduction to general-purpose GPU programming*; Addison-Wesley Professional, 2010.
26. Awan, M.G.; Saeed, F. GPU-ArraySort: A parallel, in-place algorithm for sorting large number of arrays. Parallel Processing Workshops (ICPPW), 2016 45th International Conference on. IEEE, 2016, pp. 78–87.
27. NVIDIA. cuBLAS. http://docs.nvidia.com/cuda/cublas/index.html#axzz4VJn7wpRs, 2017.