View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarWorks at WMU

Western Michigan University
ScholarWorks at WMU

Parallel Computing and Data Science Lab Technical Reports

Computer Science

2017

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

Taban Eslami
Western Michigan University, taban.eslami@wmich.edu

Muaaz Gul Awan
Western Michigan University, muaazgul.awan@wmich.edu

Fahad Saeed
Western Michigan University, fahad.saeed@wmich.edu

Follow this and additional works at: http://scholarworks.wmich.edu/pcds_reports

Part of the Bioinformatics Commons, Computational Engineering Commons, Computer Sciences Commons, and the Neuroscience and Neurobiology Commons

WMU ScholarWorks Citation

Eslami, Taban; Awan, Muaaz Gul; and Saeed, Fahad, "GPU-PCC: A GPU Based Technique to Compute Pairwise Pearson's Correlation Coefficients for Big fMRI Data" (2017). *Parallel Computing and Data Science Lab Technical Reports*. 9.
http://scholarworks.wmich.edu/pcds_reports/9

This Technical Report is brought to you for free and open access by the Computer Science at ScholarWorks at WMU. It has been accepted for inclusion in Parallel Computing and Data Science Lab Technical Reports by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.

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

Taban Eslami
taban.eslami@wmich.edu
Department of Computer Science
Western Michigan University

Muaaz Gul Awan
muaazgul.awan@wmich.edu
Department of Computer Science
Western Michigan University

Fahad Saeed
fahad.saeed@wmich.edu **
Department of Computer Science
Western Michigan University

1 Abstract

Functional Magnetic Resonance Imaging (fMRI) is a non-invasive brain imaging technique for studying the brain's functional activities. Pearson's Correlation Coefficient is an important measure for capturing dynamic behaviors and functional connectivity between brain components. One bottleneck in computing Correlation Coefficients is the time it takes to process big fMRI data. In this paper, we propose GPU-PCC, a GPU based algorithm based on vector dot product, which is able to compute pairwise Pearson's Correlation Coefficients while performing computation once for each pair. Our method is able to compute Correlation Coefficients in an ordered fashion without the need to do post-processing reordering of coefficients. We evaluated GPU-PCC using synthetic and real fMRI data and compared it with sequential version of computing Correlation Coefficient on CPU and existing state-of-the-art GPU method. We show that our GPU-PCC runs 94.62× faster as compared to the CPU version and 4.28× faster than the existing GPU based technique on a real fMRI dataset of size 90k voxels. The implemented code is available as GPL license on GitHub portal of our lab at https://github.com/pcdslab/GPU-PCC.

2 Introduction

Functional Magnetic Resonance Imaging (fMRI) is a non-invasive brain imaging technique for studying the brain functional activities [1]. This technology helps researchers and physiologists to find the facts related to human behavior and psychology and is based on Blood Oxygen Level Dependent (BOLD) contrast. fMRI data includes a sequence of images taken by a scanner through time while subject performs one or more tasks. The data acquired from an fMRI experiment includes a number of cubic elements called voxels. Changes in voxel intensity across time reveals the hemodynamics change in the brain [2]. A time series data set is extracted from each voxel and used for further analysis. Pearson's Correlation Coefficient has become popular in fMRI study to analyze functional connectivity of different regions in the brain [3, 4]. This measure reveals linear dependency between pairs of elements. For a pair of variables (x, y), Pearson's Correlation

**To whom correspondence should be addressed

Coefficient is calculated using the following equation:

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

In this equation x and y are two T dimensional variables. In case of fMRI data, x and y corresponds to two individual voxels and T shows the length of time series of each voxel. Pearson's Correlation Coefficient ρ_{xy} is a real value in range -1 and 1 [5]. Absolute value of 1 indicates strong perfect linear relationship among variables while value 0 indicates no linear relationship and -1 shows perfect negative linear relationship among them. Computing pairwise correlations is computationally expensive for large number of data (like fMRI data), so multiple approaches have been proposed based on parallel computing techniques to accelerate the computations.

Gembris et al. [6] proposed a GPU based method that computes pairwise Correlation Coefficients among elements by reformulating the Pearson's correlation equation to minimize the number of necessary divisions:

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

Wang et al.[4] proposed a parallel method for computing pairwise Correlation Coefficients over multiple time windows. They used a controller worker method with Message Passing Interface (MPI). In another work, Liu et al. developed a general framework for computing all pairwise Correlation Coefficients on Intel Xeon Phi clusters [7]. Based on the symmetric property of Pearson's Correlation Coefficient ($corr(x, y) = corr(y, x)$), pairwise Correlation Coefficients among N elements can be presented by an array containing N(N-1)/2 elements which corresponds to the upper triangle above the main diagonal part of the correlation matrix. The main diagonal is disregarded since it shows the correlation of each element with itself which is always one. Fig. 1 shows an example of the desired elements of the correlation matrix and their orders in the array. In this order, 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. Liang et al [8] developed a tool using GPU for constructing gene co-expression networks based on computing N(N-1)/2 Pearson Correlation Coefficients. The order of resulting correlation coefficients in that approach is different from Figure 1; also they applied a different strategy for distributing work among threads. Based on the symmetric property, a hybrid CPU-GPU framework for computing Correlation Coefficient using General Matrix-Matrix Multiplication (GEMM) is proposed by Wang et al [9]. In this approach, they normalize the time series of each voxel v using the following equation[1]

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

and $U = (u_1, u_2, ..., u_n)$ aggregates all normalized vectors. The correlation matrix can be constructed by multiplying matrix U to its transpose ($U \times U^T$). Since the size of correlation matrix may be larger than GPU memory, they divide matrix U into smaller blocks and compute matrix multiplication of each block with others. Using this strategy, this approach can handle correlation matrices of any size. If data contains N vectors and each block contains d vectors, there will be $B = \lceil \frac{N}{d} \rceil$ blocks. Considering the symmetric property, two blocks are multiplied to each other once which results in B(B+1)/2 block multiplications. The number of computed elements is greater than N(N-1)/2 and their order is different than what we showed in Fig. 1. A post processing step is needed in their implementation to eliminate redundant elements and

[1] $u_i = \frac{v_i - \bar{v_i}}{|v_i - \bar{v_i}|}$ is the equation that is mentioned in their paper [9] but we found that equation 3 is used in their implementation

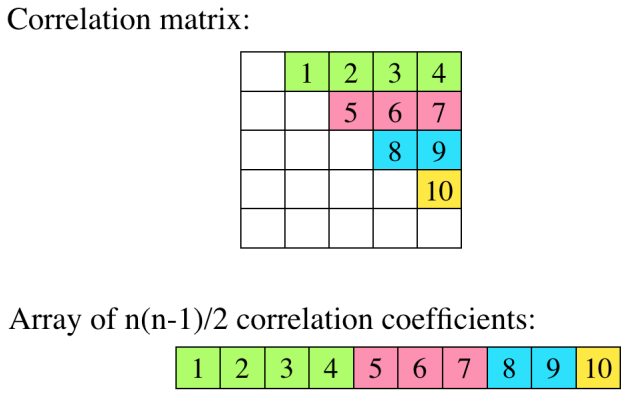

Figure 1: The order of desired Correlation Coefficients of a 5×5 correlation matrix

reorder the elements. The post processing part runs on CPU.
In this paper, we first perform a preliminary experiment for computing vector of Correlation Coefficients by performing Matrix-Vector multiplication using CUDA built-in functions. In order to exploit fine-grained thread level parallelism, we proposed an approach called GPU-PCC in which we designed our own kernel to compute Pearson's Correlation Coefficients without using built-in functions. Our proposed technique normalizes the data and computes their multiplications using massive number of GPU cores. The design of our parallel strategy allows us to compute the Correlation Coefficients in the desired order and without redundancies; eliminating the needs for post-processing strategies. This results in highly scalable parallel strategy with increasing number of elements.

2.1 GPU architecture, CUDA programming model and cuBLAS library

Graphics Processing Unit (GPU) were originally designed to satisfy the demand for higher quality graphics in video games to create more realistic 3D environment [10]. Recently GPU's have found multitude of high performance applications which can exploit high latency and high throughput of enormous number of GPU cores [11, 12]. A GPU is made up of an array of streaming multiprocessors (SMs). Each SM contains multiple streaming processors or cores. Hundreds of threads run on the same core concurrently based on SIMT (Single Instruction Multiple Threads) strategy. A warp is a group of 32 threads which execute the same instruction at the same time on the SM. Compute Unified Device Architecture (CUDA) is a programming model interface created by NVIDIA for programming graphic cards. CUDA programmers define functions called kernel which is executed on device. A kernel is defined using a number of GPU threads which execute the kernel's code in parallel. Parallel invocations of kernel are grouped into blocks which are distributed among available SMs. Each block has up to three dimensions and contains maximum 1024 threads. A grid consists of multiple blocks in one or two dimensions. The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library is the GPU version implementation of BLAS (Basic Linear Algebra Subprograms) library which performs vector and matrix operations like Matrix-Vector multiplication and Matrix-Matrix multiplication [13].

3 Proposed Methods

As stated earlier, after normalizing the data, Pearson's Correlation Coefficient among two variables can be computed by multiplying their corresponding normalized vectors. It worth mentioning that the normalization step is performed much faster than the multiplication step so we leave this step to be performed on CPU.
Assume that data is stored in an $N \times M$ matrix called U, which N corresponds to the number of data points and M corresponds to the length of each data point. In case of fMRI data, N is the number of voxels and M

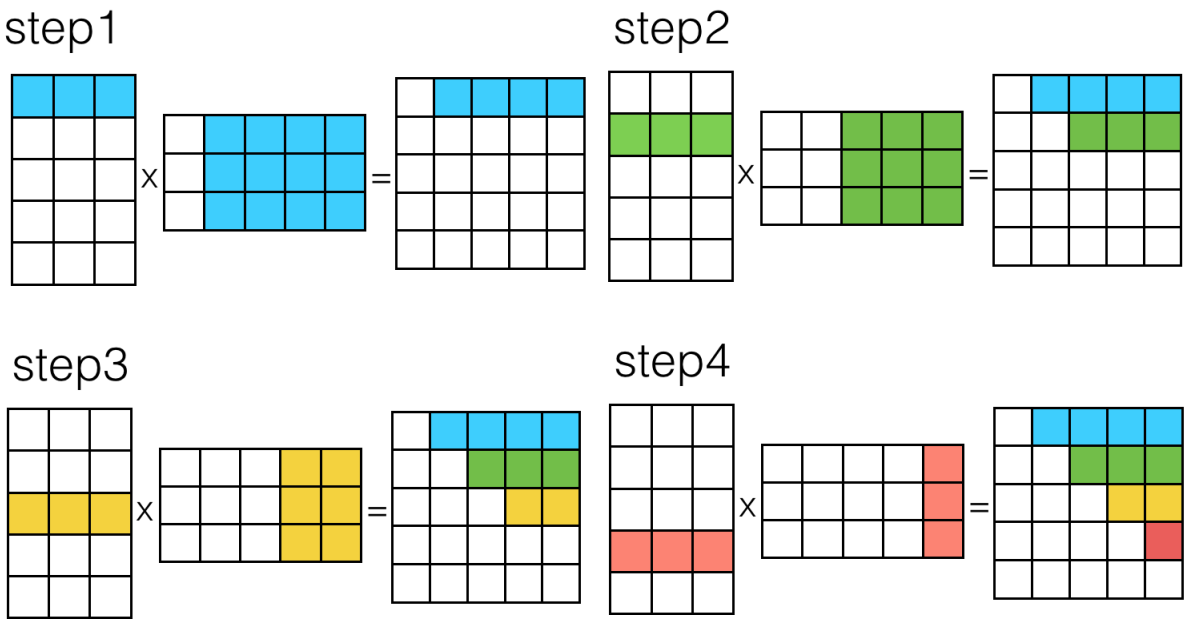

Figure 2: Example of performing Matrix-Vector multiplication, step i computes $n - i$ elements of the final result

is to the length of time series. In this section we present our preliminary experiment and proposed method to compute the ordered array of Correlation Coefficients without the need of post-processing re-ordering of elements.

3.1 Preliminary experiment of performing Matrix-Vector multiplication using CUDA built-in function

In order to obtain the desired order of Correlation Coefficients, this approach first computes the Correlation Coefficient between the first element and all other elements. This can be performed by multiplying the first row of matrix U to all other rows. We consider the transpose of all the rows except the first row as another matrix, so correlation between the first element and rest of elements can be computed by performing Matrix-Vector multiplication. Similarly, the correlation between the second element and the rest of elements can be computed in the same way, but this time we multiply the second row with a matrix containing all rows except the first two rows. We don't consider the first row because correlation between the first and second elements has been computed in previous step. In step i, row i of matrix U is multiplied to the matrix consisting of column $i+1$ to column N of matrix U^T. We continue performing this procedure until multiplying $(N-2)$th vector to $(N-1)$th vector. The procedure of this method is illustrated using an example in Fig. 2.

Algorithm 1 shows the pseudocode of this method.

Algorithm 1: Matrix-Vector multiplication

1: **Input:** Matrix U containing N elements each of length M
2: **output:** Array R containing $N(N-1)/2$ Correlation Coefficients
3: Normalize each vector of matrix U using equation 3
4: for $i = 1$ to $N-1$
5: Multiply row i of matrix U to the matrix containing column $i+1$ to N of U^T
6: Add the $N-i$ computed elements to array R
7: Return R

Matrix-Vector multiplication is implemented efficiently in cuBLAS library and we used that in our implementation (cublasSgemv function [13]). In order to minimize the latency of transferring data between host and device, instead of copying the $N-i$ computed elements back to the CPU in each iteration, we store them in a GPU array until there is not enough space in GPU memory. Once there is no more space all computed elements are transferred to the CPU and next iteration starts. This approach is successful in computing the Correlation Coefficients in order. To exploit fine-grained thread level parallelism, we propose an approach called GPU-PCC in which we designed our own kernel to compute Pearson's Correlation Coefficients.

3.2 GPU-PCC method

In this method normalized data is initially copied to GPU global memory in row-major order. We launch blocks of 512 threads and consider multiple groups per block. Each group of threads is responsible to perform a vector dot product in order to compute a Correlation Coefficient between two vectors. We used vectorized load by using float2 data type for reading data from global memory to increase bandwidth and decrease latency [14]. In order to access global memory efficiently and have coalesced memory access, we considered 16 consecutive threads in each block as a group. Threads of each group request consecutive values from memory which causes loading 128 bytes aligned data in one transaction. Since each block contains 512 threads and each 16 threads belong to one group, there are $\frac{512}{16} = 32$ groups per block which causes computing 32 Correlation Coefficients simultaneously. After each group finishes performing dot product of two vectors (we explain about the dot product process soon), it stores the result at index k of resulting vector R. Index k is computed based on the following equation:

$$k = blockIdx.x \times 32 + threadIdx.x/16 \tag{4}$$

Based on value of k, threads of each group can compute the index of vectors that they should multiply to each other (i and j) using the following equations:

$$i = n - 2 - \left\lfloor \frac{\sqrt{-8 \times k + 4 \times n \times (n-1) - 7}}{2} - 0.5 \right\rfloor \tag{5}$$

$$j = k + i + 1 - \frac{n \times (n-1)}{2} + \frac{(n-i) \times ((n-i)-1)}{2} \tag{6}$$

Equations 5 and 6 guarantee that for any index k of the resulting array, we pick the right vectors to compute their correlations and in this way the resulting array will have the correct order. Starting from the beginning of vectors i and j, each thread multiplies two corresponding consecutive values of vector i and j which results in 32 simultaneous multiplications per group. This process continuous to the next 32 elements of the vectors until all of their elements are multiplied to each other. A local variable in each thread stores the sum of products performed by that thread. In order to compute the result of dot product, the partial sums of 16 threads in each group should be added to each other. Since we have to add the values of 16 consecutive

threads and these values are in thread's registers, we used shuffle warp reduce technique introduced in [15] for computing global sum. This technique allows exchanging data among threads in the same warp without needing to use shared memory. After computing the global sum, the result of dot product will be stored at index k of vector R. Fig. 4 shows the process of vector dot product and Algorithm 2 shows the pseudo code of the kernel. Since we use float2 data type in our implementation and it needs $M/2$ load instructions, in cases that M is not even, we add a zero element at the end of each vector. This does not change the result of dot product (we don't consider these additional zeros in normalization step because it will change the result of correlation).

If the size of resulting array R is larger than GPU memory, we call the kernel multiple times. Each kernel call continues the computation until there is not enough space in global memory of the GPU to store the results. Otherwise, the computed elements are copied to host and new kernel call starts to continue the computation. Fig. 3 shows the work flow of GPU-PCC.

Empirical analysis of the implemented code showed that time-efficiency improved when L1 cache was enabled. Usually L1 cache can be enabled at compile time depending on the device properties.

Algorithm 2: GPU-PCC kernel

1: **Input:** Array U containing $N \times M$ normalized elements located in GPU memory
2: **output:** Array R containing $N(N-1)/2$ Correlation Coefficients
3: $thread\_groupId = threadIdx.x/16$
4: $thread\_local\_offset = threadIdx.x\%16$
5: $k = blockIdx.x \times 32 + thread\_groupId$
6: $i = n - 2 - \lfloor \frac{\sqrt{-8\times k+4\times n\times(n-1)-7}}{2} - 0.5 \rfloor$
7: $j = k + i + 1 - \frac{n\times(n-1)}{2} + \frac{(n-i)\times((n-i)-1)}{2}$
8: $iter = m/32$
9: $local\_sum\ = 0$
10: $float2\ data1, data2$
11: $for\ l = 1\ to\ iter$
12: $\quad data1 = U[i \times m/2 + l * 16 + thread\_local\_offset]$
13: $\quad data2 = U[j \times m/2 + l * 16 + thread\_local\_offset]$
14: $\quad local\_sum\ += data1.x \times data2.x + data1.y \times data2.y$
15: $if\ m\%32\ != 0$
16: $\quad$ continue the multiplication for the rest of elements
17: $sum =$ adding up $local\_sum$ of threads in a group using shuffle instruction
18: $if\ thread\_local\_offset = 0$
19: $\quad R[k] = sum$

4 Performance Evaluation

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 evaluated the performance of GPU-PCC by comparing it with three other approaches. The first approach is the experiment we performed using Matrix-Vector multiplication; second one is the sequential version of Pearson's Correlation Coefficient computation on CPU. We used the code that is implemented by Wang et

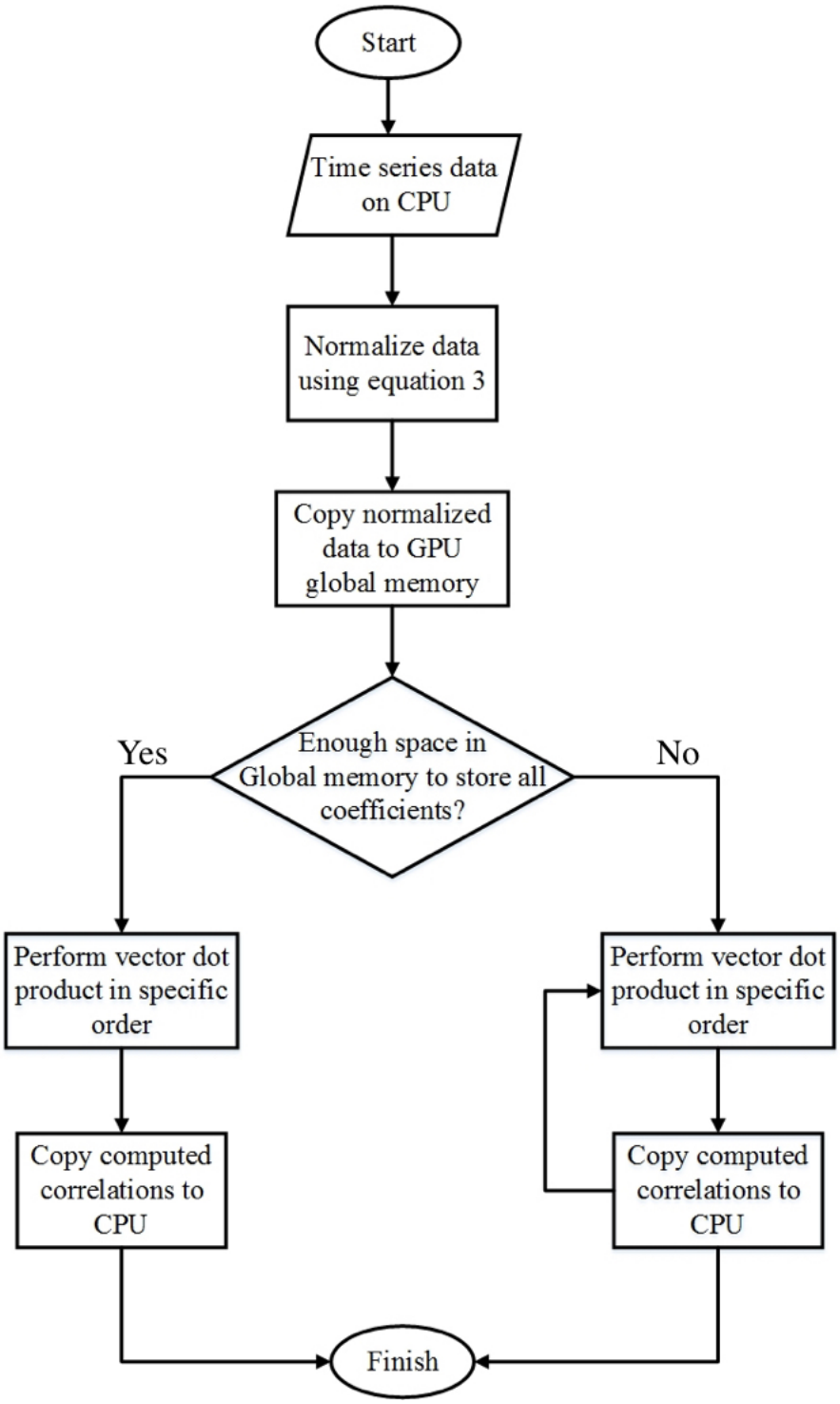

Figure 3: The work flow of GPU-PCC algorithm

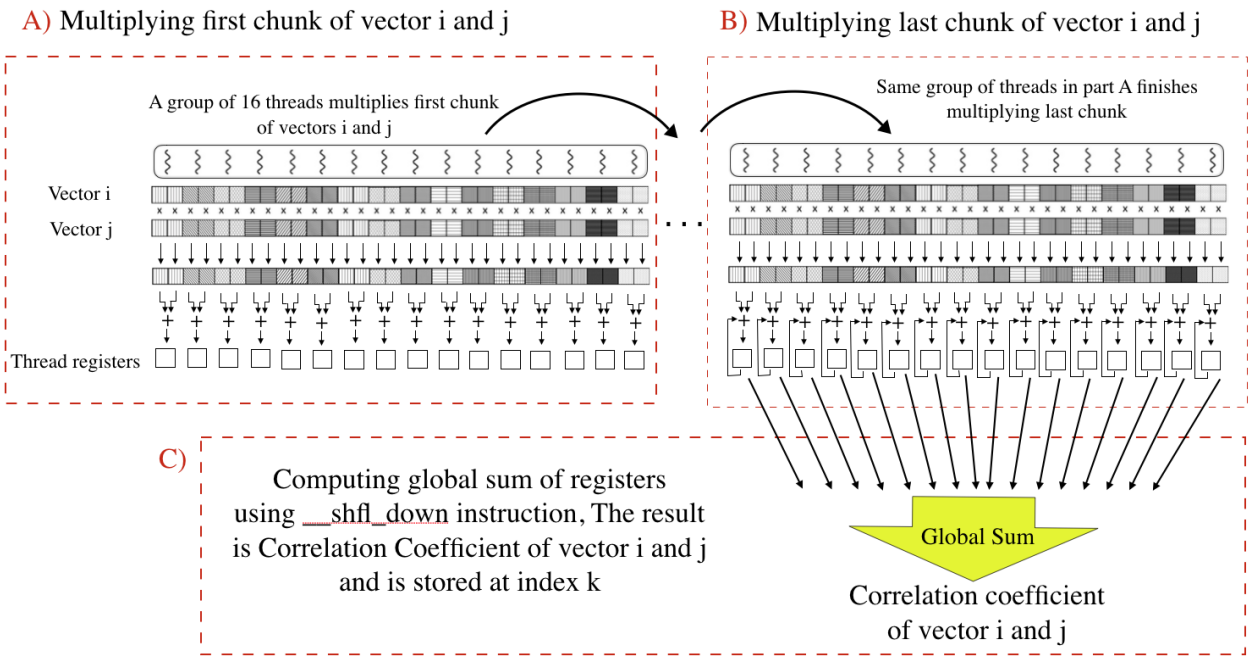

Figure 4: Example of performing vector dot product of two normalized rows (i and j) of matrix U. Multiplication of i and j is performed by a group of 16 threads. Each thread multiplies two consecutive corresponding elements of i and j, adds the results and stores it in its register. Elements that are processed with the same thread are shown using the same pattern in figure. Part A shows the multiplication of first chunk of two vectors each chunk containing 32 elements (since there are 16 threads in group each working on two elements). Part B shows the same process for the last chunk of the vectors. After multiplying the first chunk, each thread needs to update its register value by adding the new result to it. In part C, all the elements in thread registers are summed using warp shuffling technique and stored at index k (equation 4).

al [9][2]. In this implementation after normalizing data using equation 3, N(N-1)/2 vector dot products are computed on CPU. No math library is used in this implementation. We compiled the sequential code using g++ compiler version 4.8.4. The third approach that we compared our method was implemented by Wang et al [9][2]. This approach computes Correlation Coefficients by performing Matrix-Matrix multiplication on GPU (refer to section I for more details of this approach). To reorder computed elements and eliminate redundant ones, the results are post processed on CPU. So for comparing the running time of other methods with this approach, we considered both matrix multiplication and post processing steps. All reported running times in this section measure the execution time of pairwise Pearson Correlation Coefficients in desired order. The execution starts from normalizing data to performing last multiplication (For our proposed approaches to finish copying the last chunk from GPU to CPU and for Wang's approach after finishing the postprocessing to reorder computed coefficients). All the experiments for each dataset are repeated multiple times and the minimum running time is reported. We performed our experiments using synthetic and real fRMI data sets. Synthetic data sets were created with $N = 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000$ and $M = 300$. For each vector, we generated uniformly random floating point numbers in range -2 and 2 as intensity of each voxel. For real data set, we used Orangeburg dataset (www.nitrc.org/projects/fcon_1000/). The dataset contains resting state fMRI data of 20 healthy subjects, 15 female and 5 male with age range 20-55. The data were acquired using 1.5 Tesla scanner. Subject were asked to close their eyes during acquisition. We picked one of the subjects randomly for our experiments. Table 1, Fig. 5 and Fig. 6 compare the running

[2]https://github.com/BNAplatform-organization/PAGANI-toolkit/tree/master/src/BNAPlatform-win64-cuda7.0-20151118/src/CorMat

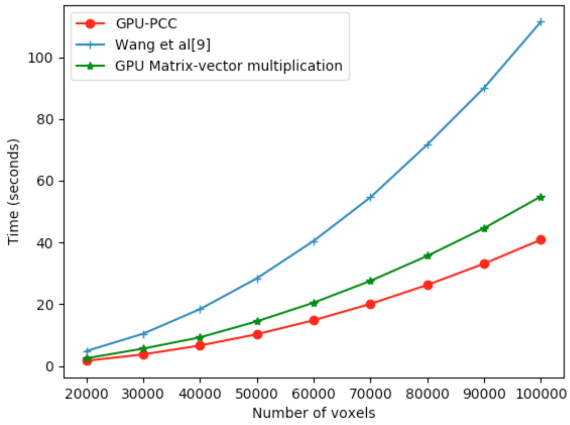

Figure 5: Running time comparison of our proposed methods and Wang's method

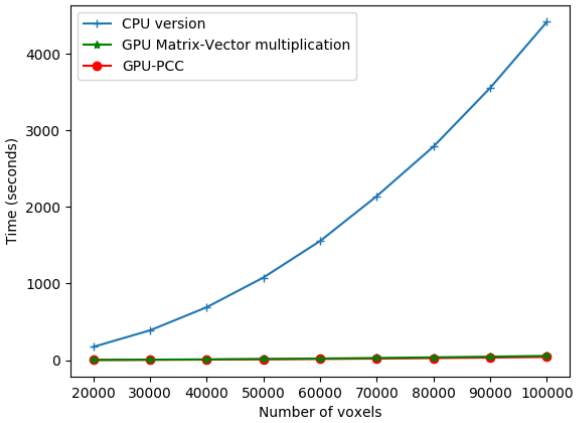

Figure 6: Running time comparison of our proposed methods and CPU version

time of different approaches. Based on the results shown in Table 1 and Fig. 5, GPU-PCC and Matrix-Vector multiplication using CUDA built-in functions show better performance than the other two approaches and GPU-PCC shows superior results. Fig. 7 shows the achieved speedups of these two methods on synthetic data. Based on the results, the achieved speedup over Wang's method for the Matrix-Vector multiplication using CUDA built-in functions is around 2 and for GPU-PCC is around 2.7. The speedups are around 70 and 100 over CPU version respectively. Table 2 shows the running time comparisons on real data. On real dataset, Matrix-Vector Multiplication using CUDA built-in functions runs 65.12× faster than CPU version and 2.95× faster than Wang's technique. Our GPU-PCC runs 94.62× faster as compared to the CPU version and 4.28× faster than the Wang's method. It worth mentioning that we didn't find specific optimization in the sequential and the postprocessing codes and we used the default CPU optimizations without using any optimization flags. Optimization of those codes could result in better running time.

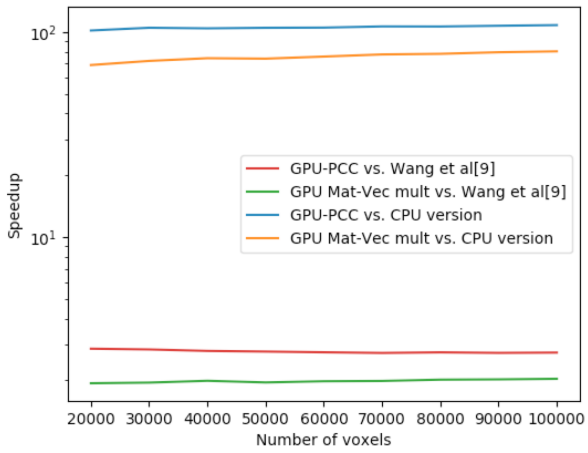

Figure 7: Speedup gained by our proposed methods over Wang's method and CPU version

Table 2: Comparing running time (Seconds) of different approaches on real fMRI data

Size of N	*GPU Mat-vec mult*	*GPU-PCC*	*Wang et al[9]*	*CPU version*
90112	30.31	20.86	89.45	1973.85

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

Size of N	*Mat-Vec mult*	*GPU-PCC*	*Wang et al[9]*	*CPU version*
20000	2.5	1.69	4.83	172.412
30000	5.37	3.76	10.45	387.89
40000	9.26	6.62	18.39	689.49
50000	14.45	10.28	28.35	1077.28
60000	20.49	14.8	40.49	1554.23
70000	27.55	20.11	54.61	2140.32
80000	35.64	26.22	71.68	2787.92
90000	44.63	33.14	90.06	3553.47
100000	54.89	40.88	111.47	4415.22

5 Conclusion

Pearson's Correlation Coefficient is a well-known technique that measures the functional connectivity between brain voxels. Since there are thousands of voxels in one fMRI experiment, using traditional CPU based methods are very time consuming. Parallel computing techniques will be essential for processing data- and compute-intensive operations for big brain research especially in the context of precision and personal medicine. Exploiting the symmetric property of the Pearson's Correlation, we can reduce the number of coefficients that need computation from N^2 to N(N-1)/2. Thereafter, we first did an experiment by performing Matrix-Vector multiplication using CUDA built-in function and then proposed a method called GPU-PCC which can compute ordered correlations and do not require further post-processing. We compared our implemented methods with a sequential C++ implementation and also with a GPU based technique based on General Matrix-Matrix Multiplication (GEMM). Both real and synthetic fMRI data sets were used

in evaluation. We show that our proposed HPC method outperforms existing state-of-art methods and is around 94× faster than the CPU versions and 4.28× faster than the GPU based techniques for similar GPU devices and data set.

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] R. C. Craddock, R. L. Tungaraza, and M. P. Milham, "Connectomics and new approaches for analyzing human brain functional connectivity," *GigaScience*, vol. 4, no. 1, p. 1, 2015.

[2] M. A. Lindquist *et al.*, "The statistical analysis of fmri data," *Statistical Science*, vol. 23, no. 4, pp. 439–464, 2008.

[3] H. Zhang, J. Tian, and Z. Zhen, "Direct measure of local region functional connectivity by multivariate correlation technique," in *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. IEEE, 2007, pp. 5231–5234.

[4] Y. Wang, J. D. Cohen, K. Li, and N. B. Turk-Browne, "Full correlation matrix analysis of fmri data," Technical report, Princeton Neuroscience Institute, Tech. Rep., 2014.

[5] J. Lee Rodgers and W. A. Nicewander, "Thirteen ways to look at the correlation coefficient," *The American Statistician*, vol. 42, no. 1, pp. 59–66, 1988.

[6] D. Gembris, M. Neeb, M. Gipp, A. Kugel, and R. Männer, "Correlation analysis on gpu systems using nvidia's cuda," *Journal of real-time image processing*, vol. 6, no. 4, pp. 275–280, 2011.

[7] Y. Liu, T. Pan, and S. Aluru, "Parallel pairwise correlation computation on intel xeon phi clusters," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2016 28th International Symposium on*. IEEE, 2016, pp. 141–149.

[8] M. Liang, F. Zhang, G. Jin, and J. Zhu, "Fastgcn: a gpu accelerated tool for fast gene co-expression networks," *PloS one*, vol. 10, no. 1, p. e0116776, 2015.

[9] Y. Wang, H. Du, M. Xia, L. Ren, M. Xu, T. Xie, G. Gong, N. Xu, H. Yang, and Y. He, "A hybrid cpu-gpu accelerated framework for fast mapping of high-resolution human brain connectome," *PloS one*, vol. 8, no. 5, p. e62789, 2013.

[10] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[11] M. G. Awan and F. Saeed, "Gpu-arraysort: A parallel, in-place algorithm for sorting large number of arrays," in *Parallel Processing Workshops (ICPPW), 2016 45th International Conference on*. IEEE, 2016, pp. 78–87.

[12] A. Eklund, M. Andersson, and H. Knutsson, "fmri analysis on the gpu-possibilities and challenges," *Computer methods and programs in biomedicine*, vol. 105, no. 2, pp. 145–161, 2012.

[13] NVIDIA. (2017, Mar.) cublas. [Online]. Available: http://docs.nvidia.com/cuda/cublas/index.html#axzz4VJn7wpRs

[14] J. Luitjens. (2013, Dec.) Udaf pro tip: Increase performance with vectorized memory access. [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/

[15] ——. (2014, Feb.) Faster parallel reductions on kepler. [Online]. Available: https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/