Computing Large-scale Distance Matrices on GPU

Ahmed Shamsul Arefin, Carlos Riveros, Regina Berretta, Pablo Moscato
Centre for Bioinformatics, Biomarker Discovery & Information-Based Medicine (CIBM)
School of Electrical Engineering and Computer Science
The University of Newcastle, Australia
Callaghan, NSW 2308, Australia
Email: {Ahmed.Arefin, Carlos.Riveros, Regina.Berretta, Pablo.Moscato}@newcastle.edu.au

***Abstract*—A distance matrix is simply an $n \times n$ two-dimensional array that contains pairwise distances of a set of n points in a metric space. It has a wide range of usage in several fields of scientific research e.g., data clustering, machine learning, pattern recognition, image analysis, information retrieval, signal processing, bioinformatics etc. However, as the size of n increases, the computation of distance matrix becomes very slow or incomputable on traditional general purpose computers. In this paper, we propose an inexpensive and scalable data-parallel solution to this problem by dividing the computational tasks and data on GPUs. We demonstrate the performance of our method on a set of real-world biological networks constructed from a renowned breast cancer study.**

I. Introduction

The calculation of distance matrices is a complete data intensive operation that acts as a preliminary step to many computational methods, such as feature set analysis, gene expression data sets analysis, different time series data sets analysis etc. A feature set is typically represented as a matrix with n rows and m columns, where each row represents a feature and each column represents a sample of the feature. A feature set can also be viewed as a set F of n points in m-dimensional space, from which a distance d can be computed as follows:

$$d : F \times F \to \mathbb{R} \tag{1}$$

for any $x, y, z \in F$, the following hold

$$d(x,y) \geqslant 0 \tag{2}$$

$$d(x,y) = 0 \text{ (iff, } x = y) \tag{3}$$

$$d(x,y) = d(y,x) \tag{4}$$

$$d(x,y) \leqslant d(x,y) + d(y,z) \tag{5}$$

A distance can be computed using a number of measures, such the Euclidean, Manhattan, Mahalanobis, Minkowski distances etc. And another popular form of the distance matrix which is known as similarity matrix (widely used in bioinformatics) can be computed using different correlation measures such as, Pearson's product-moment correlation coefficient, Spearman's rank correlation coefficient etc.

The time complexity of computing a distance matrix with n points is $\mathcal{O}$ (n^2) and the task of computing a distance matrix is computationally intensive. For instance, a distance matrix from a feature set of size $n = 1,000,000$ requires a total of 500 billion distance computation. On general purpose computers this can be very slow and may take hours to complete. However, this can be performed in a data parallel fashion using parallel and distributed environments using some sort of sophisticated hardware like parallel processors, cluster-computers or in the best case with one or a set of super-computers [1], [2]. Interestingly, recent trend in solving this problem has moved to the domain of graphics processing units (GPUs). GPUs are not only becoming popular to scientific communities in different fields of research but also are commonly installed on todays home computers, workstations, consoles, and gaming devices [3]. Now, general purpose computing on GPUs (GPGPUs) provides an important solution to many complex problems [4]. However, the limitation of device memory on GPU poses a new problem on scalability. So, even if some of the methods are much faster than sequential version but in the real-world they are not widely feasible.

In this work, we adapted, enhanced and scaled the computation of distance matrix on GPUs by allowing divisions in computational task and data. Although our main concept originates from the *large-scale matrix multiplication* method on GPUs [5] but in practice we extended the method proposed in Chang et al. [6], [7], [8] using CUDA.

Rest of the paper is organized as follows: in Section II, we give an overview of the GP-GPU and its programming models and the state-of-the-art distance matrix computation algorithms on GPU. In Section III, we explain the algorithm of our proposed method on the GPU implementation. In Section IV, we show the performances of our proposed method. Finally, in Section VI, we draw a concluding remark on our work.

II. Literature Review

A. GP-GPU and Its Programming Models

The GPGPU is a powerful device that is devoted to parallel data processing rather than data caching and flow control as a general purpose CPU. The massive parallel processing capability of GPU makes it more attractive for algorithmic problem solving, when the processing of data (or a large block of data) can be handled in parallel. In general, the GPUs are organized in a streaming, data-parallel model in which the processors execute the same instructions on multiple data streams simultaneously (see Figure, 1(a)). At the software level, there exist several application programming interfaces

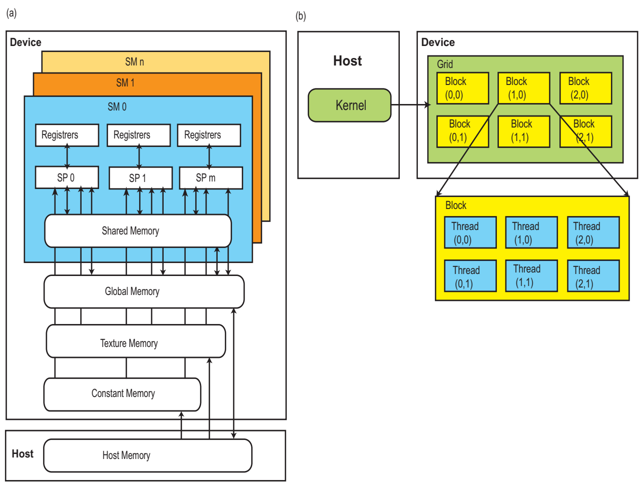

Fig. 1. (a) GP-GPU architecture and (b) 2D thread organization in CUDA

(APIs) such as CUDA, OpenCL, DirectCompute and more recent innovations like OpenACC that enable programmers to develop applications on GPU. However, NVidia's CUDA is one of the most widely adopted API that enables developing GPU-based applications using `C/C++` programming language. It exposes an abstraction to the programmers so that they can parallelize their tasks on GPU using concepts of threads, blocks and grids (see Figure, 1(b)). CUDA supports a large number of threads (up to 512 or 1024, depending on the architecture) in a block and the maximum number of blocks in a grid can be up to (2^{16} - 1), in each dimension (at most three dimensions). Both the threads and blocks can be referenced by their indices. A CUDA program typically consists of a component that runs on the CPU, or host, and a smaller, but computationally intensive component called *kernel*, that runs in parallel on the GPU. The kernel cannot access the main memory of the host directly and the input data for the kernel must be copied to the GPU's on-board memory prior to invoke the kernel and the output data also must first be written to the GPU's memory and then copied back to CPU's in-memory.

B. Existing Approaches on GPU

A number of algorithms and their implementations are available for computing distances matrices on GPUs and many of them highlight their usage on feature sets analysis. For instance, Chang et al. proposed the first distance matrix computation using CUDA on GPU employing Euclidean [6], [7], Manhattan and Pearson's correlation coefficient [8]. The main feature of their implementations is the usage of shared memory and they achieved around 20 to 44 times speed-ups for Euclidean distance, 40 to 90 times speed-ups for Manhattan distance and 28 to 38 times speed-ups for computing Pearson's correlation coefficient. However, the main limitation of their method is the dependency on device memory. If the data sets or the distance matrix doesn't fit into device memory, the method is not applicable. Additionally, they require the data size to be multiple of 16 (ie., so that the threads in *half-warp* can be executed in parallel, where a *warp* = a set of 32 threads).

Li et al. [9] proposed a chunking method for Euclidean distance Matrix computation on GPUs. They implemented a GPU algorithm that is suitable for calculating sub-matrices and then, utilized a *Map-Reduce* [10], [11] like framework to split the final distance matrix calculation into many small independent jobs of calculating partial distance matrices. Their method is scalable in terms of size of the datasets and calculations. However, the distribution of the jobs is the key issue here and certain reducers may get less jobs [9]. Thus, a system refresh is required whenever a new chunk is loaded or deleted from the system. Additionally, Srikanthan [12] porposed another variant of Euclidean distance matrix computation on GPUs for signal processing. Ying et al. [13] proposed a DNA distance matrix computation on GPUs using OpenCL which is designed to have more statistical accuracy than other methods on GPUs.

In our work we extended the distance matrix computation approach in [6], [7], [8] which free from the device memory limitation, doesn't require any other special mechanism to distribute computational tasks and yet scalable to large-scale data sets.

III. Proposed Method

A. Overview

The basics of our proposed method is similar to Li et al. [9], however, ours is much simpler and originally designed to solve k-nearest neighbor problem on GPU [14] and data clustering [15], [16]. We deigned a distance computation algorithm on GPU (`distance kernel`) by adapting [6], [7], [8] that is suitable for calculating sub-matrices (termed as *chunks*, of size $n_{chunksize}$) of the complete distance matrix. The size of the chunk is given as a parameter. All the chunks that share the same rows in a distance matrix are considered that they are in a same *split* and they are solved one after another. Then, splits are again subdivided into several *segments*, each of which are handled in parallel by different GPUs. Whenever a chunk is solved, outcome is sent back to the host. In Figure 2, we demonstrated the basic working principle of our proposed method.

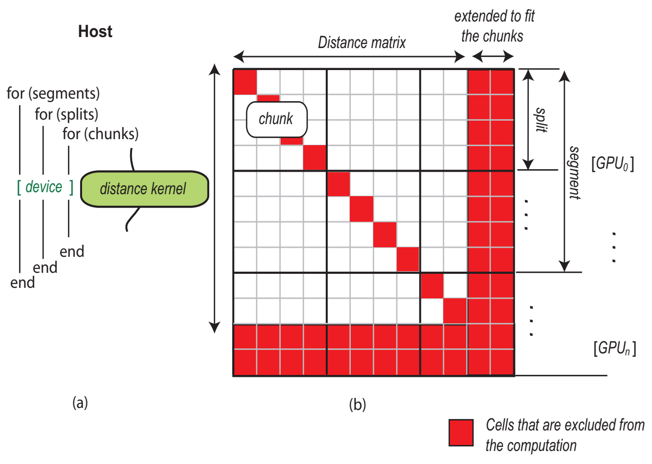

Fig. 2. Basic principle of the proposed method

B. Implementation using CUDA on GPU

Let, consider the input feature set (a matrix) contains n_{row} number of rows and n_{col} number of columns. Since, CUDA programming API does not support transferring multidimensional arrays from host to device, we store the input matrix in a single dimensional array In_a of length $(n_{row} \times n_{col})$. The complete distance matrix is stored in an array D_a and a chunk is stored in D'_a of size $(n_{chunksize} \times n_{chunksize})$, assuming D'_a fits in device memory. It is declared as a pointer to D_a. A list of all the variables/arrays used in the method implementation is given in Table I

TABLE I
VARIABLES AND ARRAYS USED IN THE METHOD

Variables	Purpose
In_a	Holds the input feature set matrix
D_a	Holds the complete distance matrix
D'_a	Holds partial distance matrix for a *chunk*
n_{row}	Original Rows
n_{col}	Original Columns
n'_{row}	Extended row
n'_{col}	Extended columns
X, Y	Holds a chunk of the input matrix
d	holds a single distance value
b	data block size

C. Scalable Distance Computation Algorithm

The proposed method is described in Algorithm 1.It receives an input matrix (or a chunk of input matrix, explained later), In_a and iteratively produces a complete distance matrix D_a as an output.

Algorithm 1 Scalable Distance Computation Algorithm (In_a: Input matrix or a portion of the input matrix)

```
1:  Initialize (segments, $n_{gpu}$)
2:  For each, segment ∈ segments do in parallel
3:      Create $D'_a$
4:      Transfer host → device ($In_a$, $D'_a$)
5:      Initialize (splits, segment)
6:      For each, split ∈ splits do
7:          Initialize (chunks, split)
8:              For each, chunk ∈ chunks do
9:                  Call distance Kernel()
10:                 Transfer device → host ($D'_a$)
11:             End for
12:     End for
13: End for
14: Return $D_a$
```

We present the `distance kernel ()` as a template function in Algorithm 3, which can be used for several types of distance measures (such as, Euclidean or Manhattan distance) or similarity measures that can be based on Pearson's or Spearman's correlation. Here, each thread is responsible for computing a distance between a pairs of feature set. The threads are organized as a set of two dimensional blocks and grids, as in [6], [7], [8], but we modified the original algorithm to compute a chunk of the distance matrix instead of the complete distance matrix. The working procedure of the algorithm is simple, during each iteration, all the threads work together to load the $b \times b$ sized data blocks of In_a into X and Y, two single dimensional shared memory array. Then the threads are synchronized and start to calculate and accumulate their own partial distances. When the distance values are finalized, each thread computes an index and store the value in the appropriate index in D'_a.

The original distance kernel algorithm in Chang et al. [6], [7], [8], work only with the data sets where the number of rows and columns are multiple of 16. This limitation is imposed so that all threads in any half-warp (a *warp* = 32 threads) can access the data in a sequence. This is modified to make the distance kernel compatible with any number of rows and columns in the input matrix, respectively. The number of columns (n_{col}) is extended to n'_{col} (see Algorithm 2) and the number of rows (n_{row}) to n'_{row} (see Algorithm 4). In the final computation, the extra rows and columns are excluded.

Algorithm 2 Extend Columns (n_{col}: Columns in the data set)

```
1: $n'_{col} \leftarrow n_{col}$
2: While ($n'_{col}$ mod $b$)≠0 do
3: $n'_{col} \leftarrow n'_{col}$ + 1 /* data block size, b */
4: Return $n'_{col}$
```

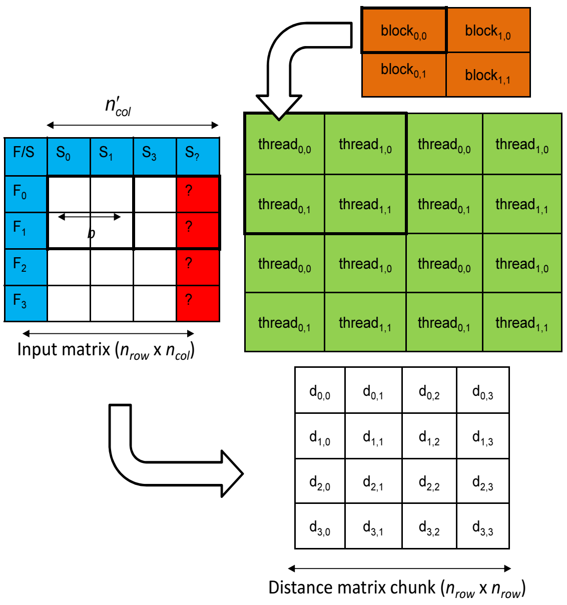

Fig. 3. An Illustration of threads and blocks organization for `distance kernel`

In Figure 3, we illustrate a similar scenario with a dataset containing four rows (n_{row} = 4) and three columns (n_{col} =

Algorithm 3 The `distance kernel` Algorithm (In_a: Input matrix or a portion of the input matrix)

```
1: Create two arrays, $X$ and $Y$ in shared memory to load $b \times b$ blocks of data from $In_a$

2: /* Initialize the thread and block indices */
3:    $bx \leftarrow$ blockIdx.x, $by \leftarrow$ blockIdx.y
4:    $tx \leftarrow$ threadIdx.x, $ty \leftarrow$ threadIdx.y

5: /* Identify the data blocks to process */
6:    $Xbegin \leftarrow (bx + n_{chunksize}/b \times chunk) \times b \times n_{col}$
7:    $Ybegin \leftarrow (by + n_{chunksize}/b \times split) \times b \times n_{col}$
8:    $Yend \leftarrow Yend + n_{col} - 1$

9: For $y \leftarrow Ybegin$, $x \leftarrow Xbegin$ to $y \leq Yend$ do by $b$   /*load data*/
10:       $Y[ty][tx] \leftarrow In_a[y + ty \times n_{col} + tx]$
11:       $X[tx][ty] \leftarrow In_a[x + ty \times n_{col} + tx]$
12:       synchronize threads()
13:           For each, column $i \in n_{col}$ to $b$ do
14:               If $(n'_{col} - n_{col}) \neq 0$ and $y \geqslant (Ybegin + n'_{col} - b)$ and $i \geqslant (n_{col} - n'_{col} + b)$
15:                   continue /* exclude extra columns */
16:               End If
17:           $d \leftarrow$ distance (or similarity) between partial data in $X$ and $Y$
18:         End for
19:    synchronize threads()
20: End for

21: /*Identify the location of the computed value in the chunk $D_a$ */
22: $index \leftarrow by \times b \times n_{chunksize} + ty \times b + bx \times b + tx$
23: $D'_a[index] \leftarrow d$
```

Algorithm 4 Extend Rows (n_{row}: Rows in the data set)

```
1: $n'_{row} \leftarrow n_{row}$
2: While $(n'_{row} \bmod n_{chunksize}) \neq 0$ do
3:     $n'_{row} \leftarrow n'_{row} + 1$
4: Return $n'_{row}$
```

3), where the data block size is b=2. The algorithm starts with (2×2) block of threads and (2×2) grid of blocks, where each thread is responsible for computing a single distance in the distance matrix. The number of columns in the input is not a multiple of the data block size (b), thus we extend the number of columns to n= 4, so that all data blocks can fit properly and subsequently, we exclude the extra column from the computation. For example, when the block (0,0) operates in the second iteration (line 14 in Algorithm 3), threads in this block will exclude column four of the input matrix from the computation. This modification allows us to compute the distance matrix for uneven chunk sizes.

IV. Performance

The performance of the proposed approach is tested against the Pearson's correlation and Euclidean distance computation algorithms in [6], [7], [8], on different variants of a renown breast cancer study dataset by Vijver et al. [17]. Along with the original dataset of 24,158 probe sets, we created nine other datasets having 4096, 8192, 12288, 32,768, 65,536, 131,072, 262,144, 524,288 and 1,048,576 elements using the concept of metafeatures in Rocha de Paula [18]. The number of samples is kept as 64 only. The computational tests were performed on a Xenon Nitro T5 Super-micro server that has Dual Xeon 5620 2.4GHz processors, 32GB of 1066 MHz DDR3 RAM and Four NVIDIA Tesla C2050 GPU cards.

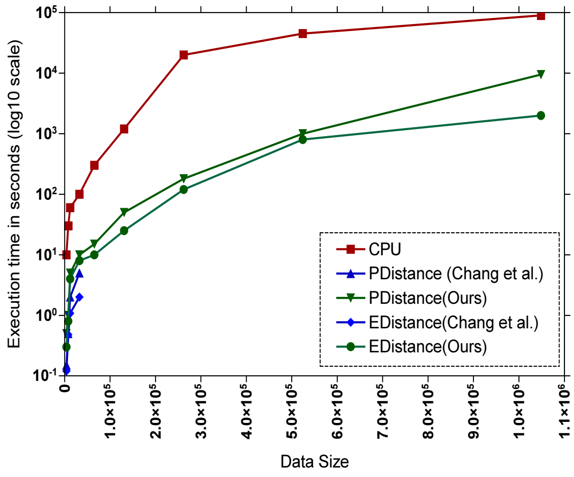

Fig. 4. Performance comparison on expanded breast cancer datasets [17].

From Figure 4, we can see that only the CPU based method and the proposed method scaled upto the largest available data sets. It should be noted here that for the data sets over 65,536 we have not attempted to keep the complete distance matrix (only computation of chunks and transfer took place).

V. Complete Scalability

In addition to the chunking of the distance matrix, we also propose to perform the chunking of the input data set, when it does not fit into GPU's onboard in-memory and in the worst case, host'ss in-memory. We create chunking of the data set using external memory programming environment [19]. The basic idea is to split the input matrix horizontally into *n* number of data chunks and then sending a set of data chunks to the device for partial distance computation. Although it is slightly slower than the original approach but it scales the method to large-scale instances.

VI. Conclusion

In this paper, we extended, enhanced a known distance matrix computation method proposed in [6], [7], [8]. Although our proposed method is comparatively slower (overall 20-30 times speed-ups observed) than the original method but in terms of scalability it is much more suitable on real world data sets. We expect that our proposed method will serve as a useful tool to a number of practical domains of scientific research.

References

[1] A. Sarje, J. Zola, and S. Aluru, "Accelerating pairwise computations on cell processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 69–77, Jan. 2011.

[2] S. Mimaroglu, M. Yagci, and D. Simovici, "Approximative distance computation by random hashing," *The Journal of Supercomputing*, pp. 1–18, 2011, 10.1007/s11227-011-0618-0.

[3] D. Kirk, "Nvidia cuda software and gpu parallel computing architecture," in *Proceedings of the 6th international symposium on Memory management*, ser. ISMM '07. New York, NY, USA: ACM, 2007, pp. 103–104.

[4] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," in *Eurographics 2005, State of the Art Reports*, Aug. 2005, pp. 21–51.

[5] NVidia, "Matrix multiplication (cuda runtime api version, sdk samples),."

[6] D. Chang, N. A. Jones, D. Li, M. Ouyang, and R. K. Ragade, "Compute pairwise euclidean distances of data points with gpus," in *Proc. of the IASTED International Symposium on Computational Biology and Bioinformatics*. IASTED, 2008, pp. 278–283.

[7] D. Chang, A. H. Desoky, M. Ouyang, and R. E. C, "Compute pairwise euclidean distances of data points with gpus," in *Proc. of the 10th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. ACIS, 2009, pp. 501–506.

[8] D.-J. Chang, A. H. Desoky, M. Ouyang, and E. C. Rouchka, "Compute pairwise manhattan distance and pearson correlation coefficient of data points with gpu," in *SNPD*, H.-K. Kim and R. Y. Lee, Eds. IEEE Computer Society, 2009, pp. 501–506.

[9] Q. Li, V. Kecman, and R. Salman, "A chunking method for euclidean distance matrix calculation on large dataset using multi-gpu," in *Proceedings of the 2010 Ninth International Conference on Machine Learning and Applications*, ser. ICMLA '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 208–213.

[10] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, 2005.

[11] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[12] S. S., A. Kumar, and V. Krishnan, "Accelarating the euclidean distance matrix computation using gpus," *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, vol. 4, pp. 422–426, 2011.

[13] Z. Ying, X. Lin, S. C.-W. See, and M. Li, "Gpu-accelerated dna distance matrix computation," *ChinaGrid, Annual Conference*, vol. 0, pp. 42–47, 2011.

[14] A. S. Arefin, C. Riveros, R. Berretta, and P. Moscato, "Gpu-fs-knn: A fast and scalable *k*nn computation technique using gpu," Faculty Postgrad Research Poster, The University of Newcastle, Australia, Sept 2011.

[15] A. S. Arefin, M. Inostroza-Ponta, L. Mathieson, R. Berretta, and P. Moscato, "Clustering nodes in large-scale biological networks using external memory algorithms," in *ICA3PP (2)*, ser. Lecture Notes in Computer Science, Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou, Eds., vol. 7017. Springer, 2011, pp. 375–386.

[16] A. S. Arefin, C. Riveros, R. Berretta, and P. Moscato, "kNN-MST-Agglomerative: A fast and scalable graph-based data clustering approach on gpu," *IEEE 7th International Conference on Computer Science & Education (ICCSE'12)*, 2012, (Accepted).

[17] M. J. van de Vijver, Y. D. He, and et al., "A gene-expression signature as a predictor of survival in breast cancer." *The New England journal of medicine*, vol. 347, no. 25, pp. 1999–2009, Dec. 2002.

[18] M. Rocha de Paula, M. G. Ravetti, O. A. Rosso, R. Berretta, and P. Moscato, "Differences in abundances of cell-signalling proteins in blood reveal novel biomarkers for early detection of clinical alzheimers disease," *PLoS ONE*, vol. 6, no. 3, p. e17481, 2011.

[19] R. Dementiev, L. Kettner, and P. Sanders, "Stxxl: standard template library for xxl data sets," *Softw. Pract. Exper.*, vol. 38, no. 6, pp. 589–637, 2008.