Compute Spearman Correlation Coefficient with Matlab/CUDA

Seongho Kim
Bioinformatics & Biostatistics Department
University of Louisville
Louisville, Kentucky 40292
USA
s0kim023@louisville.edu

Ming Ouyang*
Computer Engineering &
Computer Science Department
University of Louisville
Louisville, Kentucky 40292
ming.ouyang@louisville.edu

Xiang Zhang
Chemistry Department
University of Louisville
Louisville, Kentucky 40292
USA
x0zhan17@louisville.edu

***Abstract*— *Given a data matrix where the rows are entities and the columns are features, researchers often want to compute the pairwise distances among the entities. Some common choices of distances are Euclidean distance, Manhattan distance, Chebyshev distance, and Canberra distance. Pearson and Spearman correlation coefficients, with a range from -1 to 1, can be used to define a distance: 1 minus the coefficient. Matlab is widely used in science and engineering fields for technical computing, and it provides a function in its statistics toolbox to calculate the pairwise distances, which takes a long time when the data matrix is large. Graphics processing units have become powerful co-processors to the CPUs. Nvidia GPUs can be programmed by the CUDA language. The present work studies CUDA implementation of Spearman correlation coefficient that can be called from Matlab to speed up the computation of pairwise distances. Speedups from 7.1 to 28.9 folds are achieved.*

Keywords*— *Pairwise distance, Matlab, GPU, CUDA, Spearman correlation coefficient

I. Introduction

Many fields of science and engineering collect and analyze data in a matrix form. Each row in the matrix represents an entity, and each column represents a feature. In bioinformatics, for example, the entities may be messenger RNAs, proteins, peptides, or metabolites, and the features are biological samples; the numbers in the matrix are the abundances of the entities in the samples. In text mining, as another example, the entities may be texts, and the features are words; the numbers in the matrix are the word frequencies in the texts. A basic operation in data analysis is to calculate the distance between two entities; a typical distance metric is the Euclidean distance. Very often, when given a data matrix, researchers want to obtain all the pairwise distances among the entities. Let A be the $n \times m$ data matrix. The pairwise distances of entities in A can be stored in an $n \times n$ matrix $P = [P_{ij}]$, where P_{ij} is the distance between the entities of the i-th and j-th rows of A. Henceforth P is called the distance matrix. If the distance between two entities can be calculated in $O(m)$ time, which applies to most of the commonly used distances, the (sequential) time to compute P under the random access machine model is $O(n^2m)$. The present work focuses on the parallel computation of P using graphics processing units (GPU).

⋆ Corresponding author, partially supported by US FAA Grant 11-G-010

GPUs on commodity video cards were originally designed towards the needs of the 3-D gaming industry for high-performance, real-time graphics. Software development on them used languages such as OpenGL shading language and Direct3D high-level shader language. In 2006, Nvidia Corporation released a new generation of GPUs designed for general purpose computation. These G80 GPUs provide up to 128 stream processors and support 12,288 active threads. This architecture facilitates efficient general purpose computing on GPUs (GPGPUs). In 2007, Nvidia released an extended C language for GPU programming called CUDA [1], short for Compute Unified Device Architecture. Using CUDA, innovative data-parallel algorithms can be implemented in general computing terms to solve many important, non-graphics applications, such as database searching and sorting, medical imaging, protein folding, and fluid dynamics simulation. The latest Nvidia GPU architecture is called Fermi. Released in 2010, Fermi incorporates many essential features that used to be found only in CPUs, such as L1 and L2 caches and hardware error checking and correction. A Fermi device may have up to 512 stream processors, and it brings high performance computing to a desktop environment.

Concerning the distance between two entities, a number of metrics are commonly used, including Euclidean distance (the L^2 norm), Manhattan distance (the L^1 norm), Chebyshev distance (the L^∞ norm), and Canberra distance [2]. Pearson correlation coefficient, ρ, and Spearman's rank correlation coefficient have a range from -1 to 1; they can be used to define a distance: $1 - \rho$. Definitions of these metrics are given in Section II. Previously we have described the GPU computation of the distance matrix using most of these metrics except Spearman rank correlation coefficient [3], [4], [5], under the assumption that n and m are multiples of sixteen. The number sixteen comes from fitting an algorithmic design to the Nvidia GPU architecture. Other researchers have attempted to extend the computation to arbitrary n and m for Euclidean distance [6]. However, the proposed extension fails to observe a general guideline of GPU algorithm design: the avoidance of divergent execution, and thus their results are sub-optimal. In [5], we proposed a framework for GPU computation of distance matrices that allows arbitrary n and m and avoids the pitfall of divergent execution. The framework

is general enough for all distance metrics considered therein.

Matlab is a platform for technical computing. It is widely used in science and engineering fields. There is a function in its *Statistics Toolbox* called `pdist` that calculates the pairwise distances. However, it may take a long time when the data matrix is large. Another Matlab toolbox called *Parallel Computing Toolbox* provides mechanisms to utilize multi-cores in a computer or multi-computers in a cluster for parallel computing. Some Matlab functions have already been rewritten to take advantage of parallel computing, but this is not the case for `pdist`. That is, regardless how much computing hardware one may have, `pdist` is still computed by a single CPU core. In addition to using multiple CPU cores, the Parallel Computing Toolbox provides mechanisms to use Nvidia GPUs for parallel computation directly from Matlab. The present work develops GPU/CUDA code and the associated Matlab script that compute the pairwise distances as defined with 1 minus the Spearman rank correlation coefficient. Three Nvidia GPUs are used: Tesla C1060, Tesla C2050, and Tesla M2090. Depending on the sizes of data matrices, using Tesla C1060 achieves 7.1 to 15.9 folds of speedup over Matlab `pdist`; Tesla C2050 achieves 8.7 to 24.4 folds; and Tesla M2090 achieves 9.5 to 28.9 folds.

Section II contains literature review, descriptions of algorithms, and CUDA and Matlab code. Section III contains computational results. Section IV contains conclusion and discussion.

II. Methods

A. Metrics

Let X and Y be two vectors in the m-dimensional space, and let p be a real number greater than or equal to 1. The L^p distance between X and Y is defined as follows.

$$L^p(X,Y) = \left(\sum_{i=1}^{m} |X_i - Y_i|^p \right)^{1/p}. \tag{1}$$

Manhattan distance is L^1, and Euclidean distance is L^2. When p approaches infinity, Equation (1) can be written as

$$L^\infty(X,Y) = \max_{i=1}^{m} |X_i - Y_i|, \tag{2}$$

which is also known as Chebyshev distance. Canberra distance [2] is defined as follows.

$$L^{\text{cad}}(X,Y) = \sum_{i=1}^{m} \frac{|X_i - Y_i|}{|X_i| + |Y_i|}. \tag{3}$$

Canberra distance is often used for data that are near the origin, and has a special consideration when both X and Y are at the origin. From the point of view of data analysis, if two coincident points are not at the origin, their Canberra distance is zero, and thus the distance from the origin to the origin is zero. However, by definition, zero divided by zero is one, and thus the distance from the origin to the origin should be m.

Let the function $\text{avg}(X)$ be the sample mean of X, and let the function $\text{std}(X)$ be the sample standard deviation of X. Pearson correlation coefficient, ρ, is defined as follows.

$$\rho(X,Y) = \frac{1}{m-1} \sum_{i=1}^{m} \left(\frac{X_i - \text{avg}(X)}{\text{std}(X)} \right) \left(\frac{Y_i - \text{avg}(Y)}{\text{std}(Y)} \right). \tag{4}$$

A distance may be defined based on Pearson correlation coefficient: $1-\rho$. As noted in [4], a sequential computation of ρ may use three separate and sequentially executed `for` loops: one loop to calculate $\text{avg}(X)$ and $\text{avg}(Y)$, followed by another loop to calculate $\text{std}(X)$ and $\text{std}(Y)$, and followed by the other loop to calculate $\rho(X,Y)$. However, this straightforward approach will incur many cache misses and page faults with modern memory management systems if the data matrix is large. After simple algebraic manipulation, one can compute ρ in a different way as the followings. First, compute

$$\text{sum}(X) = \sum_{i=1}^{m} X_i, \tag{5}$$

$$\text{sum}(X^2) = \sum_{i=1}^{m} X_i^2, \tag{6}$$

$$\text{sum}(XY) = \sum_{i=1}^{m} X_i Y_i. \tag{7}$$

Similarly, $\text{sum}(Y)$ and $\text{sum}(Y^2)$ are also obtained. These five quantities $\text{sum}(X)$, $\text{sum}(Y)$, $\text{sum}(X^2)$, $\text{sum}(Y^2)$, and $\text{sum}(XY)$, can be computed by using one `for` loop, and thus a lot of cache misses and page faults can be avoided. After that, it is trivial to compute $\text{avg}(X)$, $\text{avg}(Y)$, and

$$\text{std}(X) = \sqrt{\frac{\text{sum}(X^2) - m \cdot \text{avg}(X) \cdot \text{avg}(X)}{m-1}}. \tag{8}$$

Similarly, $\text{std}(Y)$ is also obtained. Then Pearson correlation coefficient is calculated as:

$$\rho(X,Y) = \frac{\text{sum}(XY) - m \cdot \text{avg}(X) \cdot \text{avg}(Y)}{(m-1) \cdot \text{std}(X) \cdot \text{std}(Y)}. \tag{9}$$

To calculate Spearman's rank correlation coefficient of the two vectors X and Y, the values in each vector are replaced with their ranks (within the vector), from 1 to m; if two or more values are identical, their ranks are the average of what their ranks would otherwise be. That is, if there is a two-way tie, and the ranks of the tied values would be i and $i+1$, then their ranks are both $(i + (i+1))/2$. The Spearman's rank correlation coefficient is the Pearson correlation coefficient calculated from the vectors of ranks. Since ranks are needed, sorting of the values within a vector must be performed, which takes $O(m \log m)$ time. Thus it takes $O(n^2 m \log m)$ time to calculate the pairwise distance matrix when Spearman correlation is used. For all the other metrics, the computation time is $O(n^2 m)$.

For reasons that will be explained in Section II-C, a large matrix is often partitioned into 16×16 submatrices to be

processed by blocks of 16×16 CUDA threads. This tiling strategy needs to handle the cases when n and m are not multiples of 16. One straightforward solution is to use `if` statements in CUDA code to handle the special case when a block of threads hangs over the boundary of the matrix. However, the overhanging threads will follow a divergent execution path from the threads that are within the boundary of the matrix. Divergent execution will slow down the overall computation [5]. A second strategy is, when n or m are not multiples of 16, the data matrix are padded to avoid using `if` statements and divergent execution. Let n' and m' be the smallest multiples of 16 that are greater than or equal to n and m, respectively:

$$n' = \left\lfloor \frac{n+15}{16} \right\rfloor \cdot 16, \quad (10)$$

$$m' = \left\lfloor \frac{m+15}{16} \right\rfloor \cdot 16. \quad (11)$$

An $n' \times m'$ matrix may be created, where the upper left $n \times m$ submatrix is the same as the original matrix, and the extra stripes on the right and the bottom are filled with zeros. Then divergence-free CUDA code may be executed on this $n' \times m'$ matrix. This padding strategy works well for all the L^p metrics, because the extra zeros do not change the distances. Thus the distance matrix calculated from the padded data matrix is identical to the distance matrix calculated from the original data matrix. Padding also works well for Canberra distance if one adapts the convention that the distance from the origin to the origin is zero. However, if the mathematical definition of zero divided zero being one is used, care must be taken to return m instead of m' for the distance from the origin to the origin.

To use the padding strategy for Pearson correlation coefficient, a slight modification is needed. Let X' and Y' be two vectors of length m' such that X'_i is equal to X_i, for $i = 1, \ldots, m$, and X'_i is equal to zero for $i = m+1, \ldots, m'$; the same relation holds between Y' and Y. Using Equation (9), the $\rho(X', Y')$ calculated with m' substituted for m will generally be different from $\rho(X, Y)$. The required modification is as follows. To calculate sum(X'), sum(Y'), sum(X'^2), sum(Y'^2), and sum$(X'Y')$, the summations go from 1 to m' in order to avoid divergent CUDA thread execution. However, to calculate avg(X'), avg(Y'), std(X'), std(Y'), and $\rho(X', Y')$, the original value of m should still be used.

In general, padding with zeros would not work for Spearman rank correlation coefficient because it changes the ranks of the values originally in the vector, unless all values are less than zero. The most suitable value for padding for Spearman correlation is the maximum floating point number representable by the system. In Matlab, the function `realmax` returns such a value. In C and thus CUDA [1], the value is the constant `FLT_MAX` defined in the header file `float.h`. Thus, after sorting, the padded maximum floating point values will all stay on the high (right) end of the vector, and they receive the highest (averaged) rank. Still, the values of their ranks will change the subsequent calculation of Pearson correlation coefficient. Thus, the ranks of the padded values must be reset to zeros. In short, to calculate Spearman correlation using CUDA, the data matrix must be padded with the maximum floating point value; after sorting and ranking the values in the padded vector, the ranks of the padded values must be reset to zeros.

	C1060	C2050	M2090
Compute capability	1.3	2.0	2.0
# of multiprocessors	30	14	16
# of cores per multiprocessor	8	32	32
total # of cores	240	448	512
Core clock, MHz	602	575	650
Peak single precision GFLOPs	933	1,288	1,331
Peak double precision GFLOPs	78	515	665
RAM, GB	4	3	6
RAM bandwidth, GB/s	102	144	177
Power consumption, watts	188	238	225

Table 1. Key features of three Nvidia Tesla cards.

B. GPU and CUDA

Nvidia has released several generations of CUDA-capable GPU devices, which are characterized by their Compute Capability. The Matlab Parallel Computing Toolbox can work with GPU cards of Compute Capability 1.3 or higher. There are three such GPU cards in our labs: Tesla C1060, Tesla C2050, and Tesla M2090. Some of their key features are listed in Table 1. The computation in the present work is conducted with single precision floating point operations.

Nvidia has a programming guide to CUDA [1], which is an extension of C. Briefly, the Single Program Multiple Data (SPMD) code is written in a GPU *kernel* function, which contains the code that will be executed by the GPU processor cores. CUDA supports a large number of threads. The threads are organized into *blocks*, and the blocks are further organized into a *grid*. A block can be one-, two-, and three-dimensional, and it may contain 512 threads for Compute Capability 1.3 and lower, and 1,024 threads for Compute Capability 2.0 and higher. A grid can be one-, two-, and three-dimensional with up to $(2^{16} - 1)$ blocks in each dimension. Thus a kernel may be invoked with up to $(2^{16} - 1) \times (2^{16} - 1) \times (2^{16} - 1) \times 1024$ threads in one execution configuration. Each block of threads is executed on a multiprocessor. The threads within a block are dispatched to the processors in groups of 32, called a *warp*. A limitation is that all cores in one multiprocessor must execute the same instruction or a "No Operation." When there is an `if` statement, and when some threads within a warp follow the `if` branch while the other threads in the warp follow the `else` branch, the execution of the two branches are serialized. This divergent execution slows down the computation. When developing parallel algorithms for the GPU/CUDA platform, divergent execution should be avoided or reduced. This is the reason why the padding strategy was

developed in [5]. Additionally, high performance is achieved by having many blocks of threads so that the utilization of all the multiprocessors is kept high.

The GPU device provides *registers* and *local memory* for each thread, a *shared memory* for each block, and a *global memory* for the entire grid of blocks of threads. Although all threads execute the same GPU kernel function, a thread is aware of its own identity through its block and thread indices, and thus a thread can be assigned a specific portion of the data on which it can perform computation. The shared memory for a block of threads is fast, yet it is limited in size. One strategy to attain high performance is for the threads in the same block to collaborate on loading data that they all need from the global memory to the shared memory. The shared memory is further partitioned into banks. The threads in the same block may access different banks simultaneously, yet a memory bank conflict will serialize the threads involved in the conflict. Thus another strategy for high performance is to avoid shared memory bank conflicts as much as possible. For Compute Capability 1.3 and lower, there is 16KB of shared memory. For Compute Capability 2.0 and higher, there is a total of 64KB of RAM for shared memory and L1 cache, which can be configured either as 16KB shared memory plus 48KB L1 cache, or as 48KB shared memory plus 16KB L1 cache. There is also 768KB L2 cache.

C. Implementation

The input data is an $n \times m$ matrix, where n is the number of entities and m is the number of features. There are three steps to calculate the pairwise distance matrix where the distance is based on Spearman rank correlation coefficient. The first step is to sort each row of the matrix. The second step is to assign ranks to the features in each row, and to calculate the average ranks if there are ties. The third step is to calculate the Pearson correlation coefficient based on the ranks. The first and second steps are combined in one CUDA kernel function, and the third step is in another kernel.

C.1. Sorting

A number of classical sorting algorithms have been adapted to the GPU platform, such as odd-even merge sort [7], bitonic sort [7], quicksort [8], radix sort [9], and merge sort [9]. Most of these studies focused on sorting a very long array. In the present work, the number of features, m, is expected to be from a few dozens to a few hundreds in data analysis, whereas the number of vectors to be sorted, n, is expected to be from thousands to tens of thousands. Under these assumptions, the most suitable sorting algorithm for the present work is odd-even merge sort [7], [9] as implemented in the CUDA SDK. Specifically, the kernel function `oddEvenMergeSortShared` in the file `oddEvenMergeSort.cu` is slightly modified so that ranks can be assigned after sorting. The input matrix is stored in the global memory of the GPU. The numbers in each vector are read into the shared memory, and the sorting is performed with the data in the shared memory.

Each row of the data matrix is sorted by a block of threads, and thus the number of blocks in the grid of the execution configuration is n. If m is not a power of 2, the matrix is padded with extra columns so that, after padding, the number of columns, m', is the smallest power of 2 that is equal to or greater than m. The number for padding is the largest floating point number representable in the system. The number of threads in a block is $m'/2$. For Nvidia GPU Compute Capability 1.3 and lower, a block may have up to 512 threads, and thus m can be up to 1,024, which is adequate for most of the tasks in data analysis. For Compute Capability 2.0 and higher, a block may have up to 1,024 threads, and thus m can be up to 2,048.

C.2. Assigning ranks

Immediately after sorting, in the same CUDA kernel function, the ranks are assigned according to the sorted vector in the shared memory, and they are written to the global memory. A number receives the rank $i+1$ if its position is i (between 0 and $m'-1$) in the sorted list. This is straightforward, except that when there are ties, the ranks of the tied numbers must be replaced with the average of their ranks.

There does not seem to be an efficient way to detect the left and right boundaries of a sequence of tied values. The parallel scan algorithm [10] requires the operation of the scan to be associative, which is not true for the transitions between non-tied and tied values. Thus a simple strategy is implemented as follows. Each thread examines a number in the sorted list. If the number is less than its right (larger) number and equal to its left (smaller) number, it is looking at the right boundary of a sequence of tied values. Then this thread will move sequentially to the left till it reaches the left boundary of the tied values, and it calculates the average rank and assigns it to all the tied values. The CUDA code has several `if` statements, and a lot of divergent execution is resulted. Obviously this simple implementation is very slow if there are many tied values. However, if the underlying data distribution is continuous, the probability of ties is very small, and thus the computation penalty seldom arises.

If m is not a power of 2, the matrix is padded before the CUDA kernel function is executed. Then the elements in positions from m to $m'-1$ are the largest floating point number representable in the system. These values stay in their own places after sorting, and they receive the (same, averaged) largest rank. However, their high rank values will affect the subsequent calculation of Pearson correlation coefficient. Thus, their ranks are all reset to zero. Finally, the values of ranks are written to the GPU global memory to be used by the next CUDA kernel execution.

C.3. Calculating correlation

The input matrix `in` is $n \times m$ of n entities and m features. The values of the features are the ranks as described in the previous section. Let us assume that the matrix has been padded so that n is a multiple of 16, and m is a power

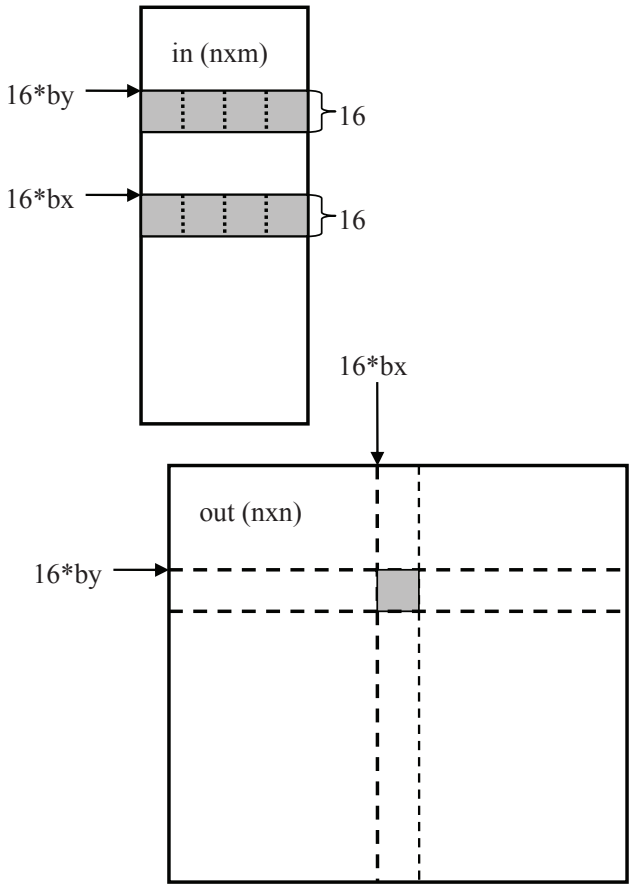

Fig. 1. Each block (of 16 by 16 threads) computes one sub-matrix of the distance matrix `out`, and the threads work on one pair of aligned sub-matrices of the data matrix `in` at a time.

of 2. The output matrix `out` is $n \times n$ of pairwise Pearson correlation coefficients. Figure 1 illustrates the idea of the CUDA algorithm. In CUDA, each thread has its block and thread indices. A two-dimensional grid and two-dimensional blocks are used, and thus each thread has four indices:

```
int bx = blockIdx.x, by = blockIdx.y;
int tx = threadIdx.x, ty = threadIdx.y;
```

Each thread will calculate one entry of `out`. Let us assume that a thread needs to calculate the entry indexed by (i, j) in the matrix `out` where i is `16*by` and j is `16*bx`; that is, both `tx` and `ty` are zero. Row `16*by` of the matrix `in` is first copied to shared memory for fast access. While the copy of row `16*by` is in shared memory, those threads that are responsible for entries $(i, j+1), (i, j+2), \ldots, (i, j+15)$ of the matrix `out` can all use the same copy for fast memory access. Similarly, after row j of the matrix `in` is copied to shared memory, the threads that are responsible for entries $(i, j), (i+1, j), \ldots, (i+15, j)$ can use the same copy. Thus it is advantageous to have 16×16 square blocks of threads that share the data in the shared memory. The blocks are organized into a grid.

```
__global__ void gpuPearson(float *in, int n,
  int mPrime, int m, float *out){
  __shared__ float Xs[BlockSize][BlockSize];
  __shared__ float Ys[BlockSize][BlockSize];
  int bx = blockIdx.x, by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;
  int xBegin = bx * BlockSize * mPrime;
  int yBegin = by * BlockSize * mPrime;
  int yEnd = yBegin + mPrime - 1;
  int x, y, k, outIdx;
  float sumX, sumY, sumX2, sumY2, sumXY;
  float avgX, avgY, varX, varY, cov, rho;

  sumX = sumY = sumX2 = sumY2 = sumXY = 0.0;
  for(y=yBegin,x=xBegin; y<=yEnd;
      y+=BlockSize,x+=BlockSize){
    Ys[ty][tx] = in[y + ty*mPrime + tx];
    Xs[tx][ty] = in[x + ty*mPrime + tx];
    __syncthreads();
#pragma unroll
    for(k=0;k<BlockSize;k++){
      sumX += Xs[k][tx];
      sumY += Ys[ty][k];
      sumX2 += Xs[k][tx] * Xs[k][tx];
      sumY2 += Ys[ty][k] * Ys[ty][k];
      sumXY += Xs[k][tx] * Ys[ty][k];
    }
    __syncthreads();
  }
  avgX = sumX/m;
  avgY = sumY/m;
  varX = (sumX2-avgX*avgX*m)/(m-1);
  varY = (sumY2-avgY*avgY*m)/(m-1);
  cov = (sumXY-avgX*avgY*m)/(m-1);
  rho = cov/sqrtf(varX*varY);
  outIdx = by*BlockSize*n + ty*n +
           bx*BlockSize + tx;
  out[outIdx] = rho;
}
```

Fig. 2. The CUDA kernel function for Pearson correlation coefficient that assumes the matrices are padded.

```
dim3 block(16,16);
dim3 grid((n+15)/16,(n+15)/16);
```

Figure 2 contains the CUDA kernel function for Pearson correlation coefficient. Notice that the CUDA kernel function `gpuPearson` in Figure 2 needs two parameters for the number of columns of the input matrix: the parameter `mPrime` for the dimension m' after padding, and the parameter `m` for the original dimension m.

C.4. Interface between Matlab and CUDA

The input data matrix is generated in Matlab using a pseudorandom number generator. The padding of the matrix for GPU/CUDA processing is carried out with the Matlab function `padarray`. Then the Parallel Computing Toolbox is used to establish the connection between Matlab and the GPU device. The function `gpuArray` is used to copy the data matrix from the CPU RAM to the GPU global memory. Of particular interests is that Matlab uses column-major storage of

a matrix while the CUDA kernels in the present work use row-major. Thus the Matlab function `transpose` is applied to the data matrix before it is passed to the CUDA kernel. After the CUDA kernel execution, the Matlab function `gather` is used to copy the pairwise distance matrix from the GPU global memory to the CPU RAM. For the purpose of comparison, the Matlab function `pdist` in the Statistics Toolbox is used to calculate Spearman correlation of the same data matrix. The computation time is measured by using the Matlab functions `tic` and `toc`.

III. Results

The computational experiments are conducted on a Linux machine with a 3.4 GHz AMD Phenom II X4 965 processor, 16 GB RAM, and a solid state boot drive. Four GPU cards, C870, C1060, C2050, and M2090, are installed directly on the motherboard, although C870 is not used in the present work because its Compute Capability 1.0 is below what is required by Matlab. For the dimensions of the data matrices, the values of n are 3,000, 6,000, 9,000, 12,000, 15,000, 18,000, and 21,000, and the values of m are 200, 600, and 1,000. The data matrices are filled with pseudorandom numbers between zero and one. Table 2 contains the experimental results. The time unit is seconds. Each computation is repeated three times, and the average time is reported. The GPU time includes the data transfer time between CPU and GPU RAM. When using C1060 and for matrices of sizes 18000×600 or larger, Matlab gave an error message that the GPU device was out of memory. This is bizarre because the memory usage is well below the 3GB of C1060; perhaps Matlab takes away some GPU RAM in an undocumented manner. Overall, Tesla C1060 achieves 7.1 to 15.9 folds of speedup over Matlab `pdist`; Tesla C2050 achieves 8.7 to 24.4 folds; and Tesla M2090 achieves 9.5 to 28.9 folds.

IV. Conclusion and Discussion

Previously we described the GPU computation of the distance matrix [3], [4], under the assumption that n and m are multiples of sixteen. In [5], we relaxed the requirement so that n and m can take arbitrary values. In the present work, we extend the CUDA computation to Spearman rank correlation coefficient, for which we need to sort the values in each of the vectors, and calculate the ranks of the values. If there are tied values, their average ranks must be calculated. Furthermore, the CUDA computation is integrated to Matlab, which is a popular platform in science and engineering fields for technical computing. Depending on the GPU devices used and sizes of the data matrices, the CUDA computation is 7.1 to 28.9 times faster than the Matlab function `pdist`. This provides a lot of speedup for routine computations by scientists and engineers.

In Table 2, for matrices with the same number of rows but 600 or 1,000 columns, the GPU computation times have similar values. This is because both matrices are padded to have 1,024 columns before passing to the CUDA kernels. Sometimes, the matrix with 600 columns may need more time than that of 1,000 columns because the former has more padded values, which create divergent computation in calculating the ranks.

		`pdist`	M2090		C2050		C1060	
n	m	time	time	ratio	time	ratio	time	ratio
3000	200	1.39	0.15	9.5	0.16	8.8	0.19	7.4
3000	600	3.84	0.27	14.2	0.31	12.4	0.46	8.4
3000	1000	6.72	0.27	24.6	0.31	21.8	0.46	14.6
6000	200	4.99	0.53	9.5	0.58	8.7	0.70	7.1
6000	600	15.98	0.97	16.4	1.12	14.3	1.71	9.3
6000	1000	26.50	0.96	27.6	1.11	23.8	1.71	15.5
9000	200	11.16	1.15	9.7	1.24	9.0	1.54	7.3
9000	600	35.52	2.09	17.0	2.45	14.5	3.79	9.4
9000	1000	59.08	2.09	28.3	2.44	24.3	3.77	15.7
12000	200	20.50	1.95	10.5	2.19	9.4	2.72	7.6
12000	600	63.01	3.82	16.5	4.26	14.8	6.64	9.5
12000	1000	104.51	3.66	28.6	4.27	24.5	6.65	15.7
15000	200	33.13	3.04	10.9	3.39	9.8	4.21	7.9
15000	600	98.12	5.81	16.9	6.63	14.8	10.33	9.5
15000	1000	162.77	5.73	28.4	6.67	24.4	10.31	15.8
18000	200	47.30	4.53	10.5	4.88	9.7	6.06	7.8
18000	600	141.00	8.13	17.4			14.80	9.5
18000	1000	234.67	8.18	28.7			14.81	15.8
21000	200	65.32	5.99	10.9			8.21	8.0
21000	600	192.29	11.11	17.3			20.08	9.6
21000	1000	319.71	11.06	28.9			20.09	15.9

Table 2. Performance comparisons of Matlab `pdist` and GPU CUDA code computing the pairwise distance matrices of Spearman correlation coefficient. The time unit is seconds. The ratios are the speedup folds of GPU over CPU.

References

[1] Nvidia, *NVIDIA CUDA C Programming Guide, Version 4.0*. Nvidia Corporation, 2011.

[2] G. N. Lance and W. T. Williams, "Computer programs for hierarchical polythetic classification (similarity analyses)," *The Computer Journal*, vol. 9, no. 1, pp. 60–64, 1966.

[3] D. Chang, N. Jones, D. Li, M. Ouyang, and R. Ragade, "Compute pairwise Euclidean distances of data points with GPUs," in *Proceedings of the IASTED International Symposium on Computational Biology and Bioinformatics*, 2008, pp. 278–283.

[4] D. Chang, A. Desoky, M. Ouyang, and E. Rouchka, "Compute pairwise Manhattan distance and Pearson correlation coefficient of data points with GPU," in *Proceedings of the 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, 2009, pp. 501–506.

[5] S. Kim and M. Ouyang, "Compute distance matrices with GPU," in *Proceedings of the 3rd Annual International Conference on Advances in Distributed & Parallel Computing*, 2012, p. in press.

[6] 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 9th International Conference on Machine Learning and Applications*, 2010, pp. 208 –213.

[7] P. Kipfer and R. Westermann, "Improved GPU sorting," in *GPU Gems 2*, M. Pharr, Ed. Addison-Wesley, 2005, pp. 733–746.

[8] D. Cederman and P. Tsigas, "GPU-Quicksort: A practical quicksort algorithm for graphics processors," *J. Exp. Algorithmics*, vol. 14, pp. 4:1.4–4:1.24, 2010.

[9] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–10.

[10] G. E. Blelloch, "Prefix sums and their applications," 1993.