Efficient Large Pearson Correlation Matrix Computing using Hybrid MPI/CUDA

Ekasit Kijsipongse and Suriya U-ruekolan
Large Scale Simulation Research Laboratory
National Electronics and Computer Technology Center
Pathumthani, Thailand

Chumpol Ngamphiw and Sissades Tongsima
Biostatistics and Informatics Laboratory, Genome Institute
National Center for Genetic Engineering and Biotechnology
Pathumthani, Thailand

Abstract— **The calculation of pairwise correlation coefficient on a dataset, known as the correlation matrix, is often used in data analysis, signal processing, pattern recognition, image processing, and bioinformatics. With the state-of-the-art Graphic Processing Units (GPUs) that consist of massive cores capable to do processing up to several Gflops, the calculation of correlation matrix can be accelerated several times over traditional CPUs. However, due to the rapid growth of the data in the digital era, the correlation matrix calculation becomes computing intensive which needs to be executed on multiple GPUs. As of now, GPUs are common components in data center at many institutions. Their GPU deployment tends towards a GPU cluster which each node is equipped with GPUs. In this paper, we propose a parallel computing based on the hybrid MPI/CUDA programming for fast and efficient Pearson correlation matrix calculation on GPU clusters. At coarse grain parallelism, the correlation matrix is partitioned into tiles which are distributed to execute concurrently on many GPUs using MPI. At fine grain level, the CUDA kernel function on each node performs massively parallel computing on a GPU. To balance load across all GPUs, we adopt the work pool model which there is a master node that manages tasks in the work pool and dynamically assign tasks to worker nodes. The result of the evaluation shows that the proposed work can ensure the load balance across different GPUs and thus gives better execution time than using a simple static data partitioning.**

Keywords- GPU, CUDA, MPI, Correlation Coefficient

I. INTRODUCTION

Correlation coefficient is a similarity measure between two objects. It is often used in data analysis, signal processing, pattern recognition, image processing, and bioinformatics. For example, in bioinformatics, it can give a measure of how two gene expressions are similar [1]. In image processing, it is used in image matching [2] and image registration [3][4]. Pearson correlation coefficient is one of the most common measure of correlation coefficient. The calculation of pairwise Pearson correlation coefficient on a dataset, known as the correlation matrix, becomes computing intensive according to the rapid growth of the data in the digital era.

Commodity Graphics Processing Units (GPUs) in today's PC evolve rapidly during recent years. The state-of-the-art GPUs consist of massive cores and are capable to do processing up to several Gflops or even Tflops. With the advent of General Purpose computing on Graphic Processing Units (GPGPU), a large number of parallel applications other than computer graphics and visualization have been migrated to GPUs. The performance of these applications can be several orders of magnitude improved than the traditional CPUs. Due to its affordable cost, GPUs have been widely adopted in high performance computing [5][6]. As of now, GPUs become common components in data center at many institutions. Their deployment tends towards GPU clusters, which each node is equipped with GPUs, to solve a particular problem. As a GPU cluster can consists of multiple different GPUs, it is critical for the developers to take load balancing as a major concern so as to achieve higher resource utilization and lower execution time in the GPU cluster environment.

In this paper, we propose a parallel computing for fast and efficient Pearson correlation matrix calculation on GPU clusters. We adopt the hybrid MPI/CUDA programming in our implementation to provide two-level parallelism in this environment. Load balancing problem on different GPUs in the cluster is also taken into consideration and it is solved using the work pool model. The rest of this paper is organized as follows. Section II provides some background information. Section III describes related work. Section IV discusses the design and implementation of an efficient hybrid MPI/CUDA programming with dynamic load balancing for Pearson correlation matrix calculation. Section V presents the evaluation result of our work on a testbed. Finally, we draw conclusions and discuss some future work in Section VI.

II. BACKGROUND INFORMATION

A. GPU and CUDA

Graphics Processing Unit (GPU) is a specialized hardware initially used to accelerate calculations for 3D computer graphics. It consists of a large number of small processing cores (known as Streaming Processor or SP) that work massively parallel. A group of eight SPs are organized into a Streaming Multiprocessor (SM) as depicted in Figure 1. Each GPU has multiple SMs to execute a number of threads in the data parallelism manner. Threads are grouped into thread blocks; each of which will be scheduled to execute on an SM. Every thread in thread blocks executes the same instruction on different data (SIMD). GPUs must be attached to a CPU host to operate. The communication between host and GPU is via PCI-Express bus.

Each GPU has its own memory but the GPU memory space is separated from host memory. To work with GPUs, data and instructions have to be copied back and forth between memory in hosts and GPUs through the PCI-Express bus. Each GPU consists of device memory and shared memory. The device memory can be accessible by all threads in different thread blocks. The device memory is large (GB in size) but has lower throughput than shared memory. In contrast, shared memory is small (KB) but it is very fast. Shared memory is allocated for a thread block. Only threads within the same thread block can access the same shared memory located on each SM. GPUs also have other types of memory such as registers, texture and constant cache but they are not shown here.

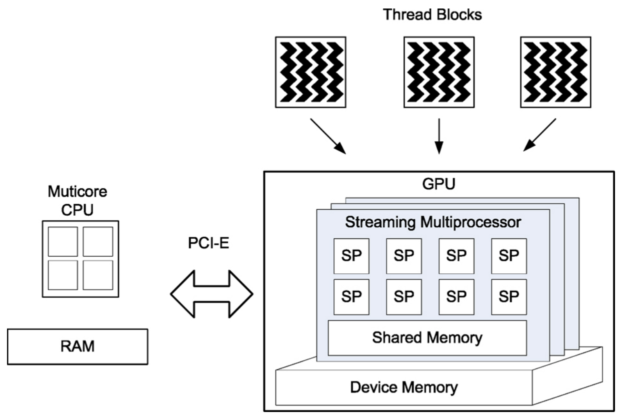

Figure 1. GPU architecture overview

To facilitate efficient general purpose computing on GPUs (GPGPUs), an extension to C language called Compute Unified Device Architecture (CUDA) [7] was introduced in 2007. Since then, programming with GPUs for general applications becomes much broader as it is easier than before. A CUDA program consists of one or more portions that are executed on either the host or the GPU. The host portions are straight C code. The GPU portions are written in C functions, enhanced with CUDA constructs and data structures, which is typically called *kernel* functions. Two CUDA structures, threadIdx (thread index) and blockIdx (block index), are used in combination to associate a thread to a different part of data for data parallel computation. A number of threads can be organized into 1D, 2D or 3D thread block. A number of thread blocks can be organized into 1D or 2D grid with a given kernel when operating on data.

B. PEARSON CORRELATION COEFFICIENT

Assume that the data are an $n \times m$ matrix where n is the number of instances and m is the number of attributes of an instance. Let X and Y be instances that contains m attributes. Mathematically, the Pearson correlation coefficient, $r_{X,Y}$, between two instances X and Y is defined as:

$$r_{X,Y} = \frac{\sum_{i=1}^{m}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{m}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{m}(Y_i - \overline{Y})^2}}$$

where $\overline{X}$ and $\overline{Y}$ are defined as:

$$\overline{X} = \frac{1}{m}\sum_{i=1}^{m} X_i$$

$$\overline{Y} = \frac{1}{m}\sum_{i=1}^{m} Y_i$$

The Pearson correlation coefficient is a measure of how two instances are linearly related. The value of $r_{X,Y}$ ranges from -1 to 1. It is closed to zero if two instances are uncorrelated. When it is positive, X and Y are correlated. The higher the value, the stronger the correlation. If the value of $r_{X,Y}$ is negative, then X and Y are negatively correlated.

The correlation between all pairs of instances can be expressed as the correlation matrix in which each element is the Pearson correlation coefficient, $r_{X,Y}$, of the different instance pairs (X,Y). The calculation of correlation matrix is highly parallelizable as each $r_{X,Y}$ can be computed independently. Figure 2 shows that the correlation matrix is partitioned into tiles for parallel computing. The output tile *A*,*B* in the correlation matrix is calculated from input tile *A* and *B* from the dataset. Note that the correlation matrix is always symmetric.

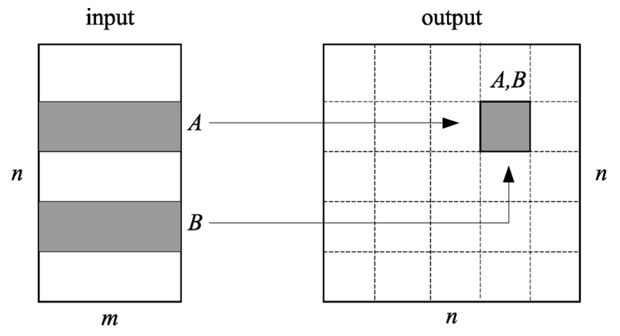

Figure 2. Partitioning correlation coffcient matrix

III. RELATED WORK

Chang et al. [1] developed an algorithm to compute pairwise Pearson correlation coefficient on a single GPU using CUDA. The speedup of the algorithm comparing to traditional CPUs is over 30 times. However, the size of the data that can be processed by their algorithm is limited by the amount of memory on the GPU. We have observed the same speedup and extended their work for the calculation on large datasets using a cluster of different GPUs.

Karunadasa and Ranasinghe [8] used hybrid CUDA and MPI to accelerate the performance of matrix multiplication and conjugate gradient method. MPI is used as the data distribution mechanism between cluster nodes and CUDA for computing engine. They could achieve a good performance enhancement. The hybrid CUDA and MPI programming has also been studied by Noaje et al. [9]; however, the MPI has been used for multiple GPUs in the same machine rather than distributed in a cluster system. Yang et al. [10] discussed the performance gain via hybrid CUDA and MPI programming through several applications including matrix multiplication, MD5 and Bubble sort on GPU clusters. Although the hybrid CUDA and MPI demonstrates a good combination, the load balancing problem among GPUs has not been concerned in the aforementioned works.

Load balancing is a critical concern in parallel computing and it receives a lot of attentions from researchers over time. Dynamic load balancing on multi-GPU systems discussed in [11] is based on task queue model which the host enqueues tasks into a queue and the GPUs dequeue tasks for processing. Though their task queue model is conceptually similar to the work pool model used in our work, it can only handle load balancing of multiple GPUs within a single node since the implementation does not support communication between nodes. Acosta et al. [12] have developed a library that allows an application with iterative calculation to adjust work amount on different GPUs based on the performance observed while the iteration proceeds.

IV. Hybrid MPI/CUDA architecture

We develop an efficient hybrid MPI/CUDA parallel program for correlation matrix calculation on GPU clusters. At coarse grain parallelism, the matrix in partitioned into tiles which are assigned to execute concurrently on many nodes. MPI is used to exchange data between nodes at this level. At fine grain level, the CUDA kernel function on each node performs massively parallel computing of each tile on a GPU.

To balance load across all GPUs, we adopt the work pool model which there is a master node that manages tasks in the work pool and dynamically assign tasks to worker nodes. Under the work pool model, the output matrix is divided into tiles of equal size as illustrated in Figure 3. Each tile corresponds to a task in the work pool. Grey tiles represent tasks that have been processed and white tiles are being processed by a GPU. The master node waits for a request of tasks and decides which task to assign to which GPU node. Each worker node receives the task, loads the relevant input data into GPU, and performs the calculation on GPU. As each GPU has different performance, it can finish a task in different time. After a GPU finish computing a task, it becomes ready to process a new task again. The master node can assign the next task to it as long as there remain tasks in the work pool. Thus, load on all GPUs should be closely balanced.

The implementation consists of two parts: MPI and CUDA. The MPI part is used for managing the work pool model among nodes while CUDA for the calculation in GPU. The interaction between MPI and CUDA is shown in Figure 4 and is also delineated below.

A. Managing work pool with MPI

The work pool implemented using MPI for load balancing between the master and GPU worker nodes are outlined in the host section of Figure 4. The master process manages the work pool. It iterates through the collection of output tiles and keeps waiting for a request from workers. If a request arrives, the master loads the corresponding input tiles, A and B, and sends them to the requesting worker. After all tiles have been completed, the master sends the termination messages to all workers.

The worker process allocates device memory and sends a request to the master for a new tile. Once the new tile is received, the worker copies relevant data to GPU's device memory and start executing the kernel on this tile. Each tile is processed with 2D grid of thread blocks, each of which is 2D thread block of size BLOCKSIZE×BLOCKSIZE. After the calculation is completed, the result gC is copied from GPU and then saved into a separate file on a shared disk. One could possibly merge all output files into a single file later. At the termination, the worker deallocates memory in GPU.

B. CUDA kernel for Pearson Correlation Coefficient

The calculation of pairwise Pearson correlation coefficient of each tile is performed by the CUDA kernel which is shown in the GPU section of Figure 4. Initially, the kernel allocates shared memory for storing the entire input tiles, gA and gB, from the device memory. Utilizing shared memory in this calculation can give a significant speedup since each element in the input tiles has to be read multiple times by different threads in a thread block. Due to the limited size of shared memory, only partial input tiles can be loaded into shared memory. Thus, the kernel needs to iterate several times through the input tiles. Each time, the kernel performs calculation on the partial inputs and then accumulates them into the final result which is eventually stored into the device memory.

Each thread in a thread block has the responsibility to load input elements into shared memory, and calculate an output element based on its threadIdx and blockIdx. The indices *i* and *j* refer to the position of an element in the output tile. The __syncthreads() provides the barrier synchronization for all threads in the same thread block at the point where the data consistency is required. Without loss of generality, it is assumed that TILESIZE is a multiple of BLOCKSIZE. For greater details, readers may refer to [1].

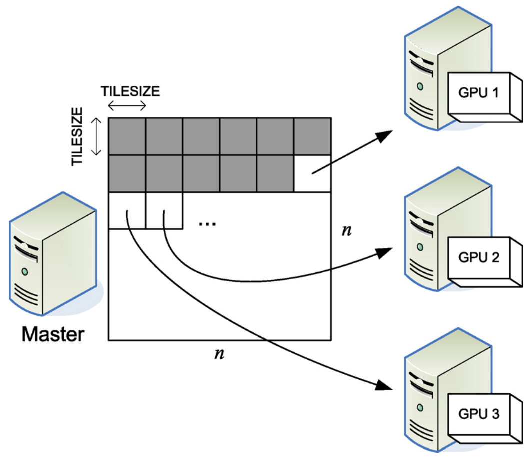

Figure 3. Dynamic load balancing using work pool model.

V. Evaluations

We have carried out an experiment on our GPU cluster testbed which consists of 3 nodes; each of which has a different GPU ranging from an entry-level to a so-called *personal supercomputer* level as shown in Table I. The host machines are Intel Quad Core 2.33 GHz with 4GB DDR3 RAM running

```
if (master process) then
      for each output tile (A,B) from tiles do
          MPI_Recv(&request, MPI_ANY_SOURCE, &status);
          task.A = load_input_tile(A);
          task.B = load_input_tile(B);
          MPI_Send(&task, status.MPI_SOURCE);
      end for
      task = no_more_task;
      for each worker process do
          MPI_Send(&task, worker process);
      end for
 else  /* worker process */
      cudaMalloc(gA); cudaMalloc(gB); cudaMalloc(gC);
      dim3 blockDim(BLOCKSIZE,BLOCKSIZE);
      dim3 gridDIM(TILESIZE/BLOCKSIZE,TILESIZE/BLOCKSIZE);
      task = empty_task;
      while (task != no_more_task) {
          MPI_Send(&request, master process);
          MPI_Recv(&task, master process, &status);
          if (task != no_more_task) {
              cudaMemcpy(gA,task.A); cudaMemcpy(gB,task.B);
              corrcoef_kernel<<<gridDim,blockDim>>>(gC,
                                   gA,gB,TILESIZE,m);
              cudaMemcpy(C,gC);
              save_output_tile(C);
          end if
      end while
      cudaFree(gA); cudaFree(gB); cudaFree(gC);
end if
```

Host

```
__global__ void corrcoef_kernel(float *gC,
                     float *gA, float *gB,
                     int nrow, int ncol) {
  __shared__ float sA[BLOCKSIZE][BLOCKSIZE];
  __shared__ float sB[BLOCKSIZE][BLOCKSIZE];

  int i,j,k;
  int offset;
  float a,b;
  float sum_a, sum_b, sum_a2, sum_b2, sum_ab, corrcoef;

  i = blockIdx.y*blockDim.y + threadIdx.y;
  j = blockIdx.x*blockDim.x + threadIdx.x;

  sum_a = sum_a2 = sum_b = sum_b2 = sum_ab = 0;
  for (offset=0; offset < ncol; offset += blockDim.x) {
     sA[threadIdx.y][threadIdx.x] = gA[(blockIdx.y*blockDim.y +
                     threadIdx.y)*ncol+offset+threadIdx.x];
     sB[threadIdx.y][threadIdx.x] = gB[(blockIdx.x*blockDim.x +
                     threadIdx.y)*ncol+offset+threadIdx.x];
     __syncthreads();

     for (k=0; k < blockDim.x; k++) {
        a = sA[threadIdx.y][k];
        b = sB[threadIdx.x][k];
        sum_a += a;
        sum_a2 += a*a;
        sum_b += b;
        sum_b2 += b*b;
        sum_ab += a*b;
     }
     __syncthreads();
  }
  corrcoef = (ncol*sum_ab - sum_a*sum_b)/
      sqrtf((ncol*sum_a2-sum_a*sum_a)*(ncol*sum_b2-sum_b*sum_b));
  gC[i*nrow+j] = corrcoef;
}
```

GPU

Figure 4. MPI and CUDA code

TABLE I. HARDWARE SPECIFICATIONS

	GPU 1	GPU 2	GPU 3
Model	Nvidia Geforce 8400GS	Nvidia GTS 250	Nvidia Tesla 1060C
Number of Cores	8	128	240
Core Clock (MHz)	1400	1620	1300
Memory Amount (MB)	512	1024	4096
Memory Bandwidth (GB/s)	8.0	70.4	102
Single Precision Peak Performance (GFlops)	67	705	933

Linux OS. All machines are connected via a Gigabit switch. MPICH 1.2.7 and CUDA SDK 3.2 are used in our implementation.

The input dataset of 200,000 instances having 4,000 attributes is simulated. All input elements are floating points. The dataset is partitioned into tile with TILESIZE set to 2,000 instances. Thus, each output tile has 2,000×2,000 output elements to calculate. The total number of output tiles (tasks) in the work pool is 10,000 for the entire correlation matrix. To process each tile, a GPU requires 64MB in device memory to store two input tiles and 16MB for each output tile. Note that it is currently impossible to store entire input and output matrices in the device memory of a single GPU as the total memory requirement is larger than 100GB. Each thread block has 256 threads organized into 16×16 threads (BLOCKSIZE is set to 16). The performance of each GPU as well as the CPU host to process a single tile is shown in Table II along with their market prices at the beginning of 2011. Note that the GPU prices listed in the table do not include the host price. Since each GPU has different performance, it can finish processing a tile at different time. We have observed that significant speedup can be achieved by using GPUs, particularly, almost 70 on GPU 3.

TABLE II. CPU AND GPU PRICE/PERFORMANCE COMPARISON

	CPU		GPU 1	GPU 2	GPU 3
	1 Core	4 Core			
Execution Time (second/tile)	276	71	119.72	6.47	3.96
Speedup	1	3.88	2.31	42.66	69.70
Price[1] ($)	750		35	135	1,600

1. Source: http://www.amazon.com

To see the effect of dynamic load balancing, we measure the execution time spent on each GPU as it directly depends on the assigned load. Figure 5 presents the execution time of each GPU when the work pool dynamic load balancing is used comparing to the simple static data partitioning which divides the work equally on all GPUs. In case of the work pool model, each GPU processes the different number of tiles according to its performance. As a result, the loads on all GPUs are mostly equal and so are their execution times. For simple static data partitioning, the load on all GPUs are imbalanced. Particularly, the load on GPU 1 is much higher than others due to its performance is several times lower. As the overall execution time depends on the longest execution time among all GPUs, the work pool model gives better overall execution time than the static data partitioning.

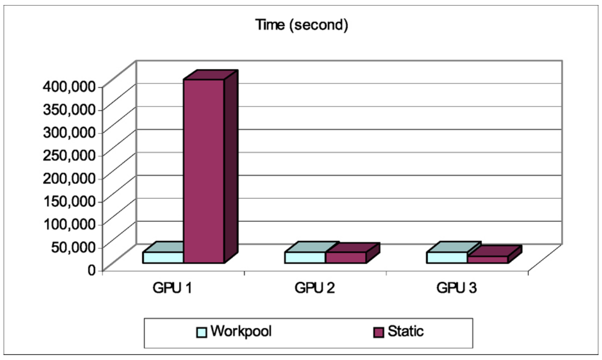

Figure 5. Static VS. dynamic load balancing

VI. Conclusions

We have developed an efficient hybrid MPI/CUDA parallel program for Pearson correlation matrix calculation on GPU clusters. At coarse grain parallelism level, the matrix is partitioned into tiles which are assigned to execute concurrently on many nodes. MPI is used to exchange data between nodes at this level. At fine grain level, the CUDA kernel function on each node performs massively parallel computing of each tile on a GPU. However, as GPUs become more common components in data center at many institutions, their deployment tends towards utilizing multiple different GPUs in a large cluster. To balance load across all GPUs, we use the work pool model which there is a master node that manages tasks in the work pool and dynamically assigns tasks to worker nodes. We have carried out an experiment on our GPU cluster testbed of 3 nodes, each of which is equipped with a different GPU. The result shows that the work pool model gives better overall execution time than a simple static data partitioning since each GPU processes the different number of tiles according to its performance. For future work, we would extend our work for multi-GPU clusters which each node may have more than one GPU installed. We also plan to develop an auto-tuning kernel for performance improvement.

VII. References

[1] 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 Proceedings of the 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing (SNPD), 2009.

[2] R. C. Gonzalez and R. E. Woods, Digital Image Processing (2nd ed.), Prentice Hall, 2002, pp. 701-703.

[3] W. Pratt, “Correlation techniques of image registration,” IEEE Trans Aerospace and Electronic Systems, AES-10(3), 1974, pp. 353-358.

[4] L. G. Brown, “A survey of image registration techniques,” ACM Comput. Surv. 24(4), 1992, pp. 325-376.

[5] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, “GPU Cluster for High Performance Computing,”. In Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC '04), 2004.

[6] V. V. Kindratenko, J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. W. Hwu, “GPU clusters for high-performance computing,” In Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER), 2009.

[7] CUDA Zone, http://www.nvidia.com/object/cuda_home.html, 2011.

[8] N. P. Karunadasa, and D. N. Ranasinghe, “Accelerating high performance applications with CUDA and MPI,” in Proceedings of the 2009 international Conference on Industrial and Information Systems (ICIIS), 2009.

[9] G. Noaje, M. Krajecki, and C. Jaillet, "MultiGPU computing using MPI or OpenMP," in Proceedings of the 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing (ICPP), 2010.

[10] C. T. Yang, C. L. Huang, C. F. Lin, and T. C. Chang, "Hybrid Parallel Programming on GPU Clusters," in Proceedings of the International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2010.

[11] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, “Dynamic Load Balancing on Single- and Multi-GPU Systems,” in Proc. of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2010.

[12] A. Acosta, R. Corujo, V. B. Pérez, and F. Almeida, “Dynamic load balancing on heterogeneous multicore/multiGPU systems,” in Proceedings of the 2010 International Conference on High Performance Computing & Simulation (HPCS), 2010.