EFFICIENT PARTITIONING BASED HIERARCHICAL AGGLOMERATIVE CLUSTERING USING GRAPHICS ACCELERATORS WITH CUDA

S.A. Arul Shalom[1] and Manoranjan Dash[2]

[1, 2] School of Computer Engineering, Nanyang Technological University, 50 Nanyang Avenue¸639798 Singapore
[1] sall0001@ntu.edu.sg and [2] asmdash@ntu.edu.sg

ABSTRACT

We explore the capabilities of today's high-end Graphics processing units (GPU) on desktop computers to efficiently perform hierarchical agglomerative clustering (HAC) through partitioning of gene expressions. Our focus is to significantly reduce time and memory bottlenecks of the traditional HAC algorithm by parallelization and acceleration of computations without compromising the accuracy of clusters. We use partially overlapping partitions (PoP) to parallelize the HAC algorithm using the hardware capabilities of GPU with Compute Unified Device Architecture (CUDA). We compare the computational performance of GPU over the CPU and our experiments show that the computational performance of GPU is much faster than the CPU. The traditional HAC and partitioning based HAC are up to 66 times and 442 times faster on the GPU respectively, than the time taken by a CPU for the traditional HAC computations. Moreover, the PoP HAC on GPU requires only a fraction of the memory required by the traditional algorithm on the CPU. The novelties in our research includes boosting computational speed while utilizing GPU global memory, identifying minimum distance pair in virtually a single-pass, avoiding the necessity to maintain huge data in memories and complete the entire HAC computation within the GPU.

KEYWORDS

High Performance Computing; Hierarchical Agglomerative Clustering; Efficient Partitioning; GPU for Acceleration; GPU Clustering; GPGPU

1. INTRODUCTION

Today's Graphics Processing Unit (GPU) on commodity desktop computers, gaming consoles, video processors or play stations has become the most powerful and affordable general purpose computational hardware in the computer world. The hardware architecture of these processors, which are traditionally meant for graphics applications, inherently enables massive parallel vector processing with high memory bandwidth and low memory latency. The processing stages within these GPUs are programmable. Such characteristics of the GPU make it more computationally efficient and cost effective to execute highly repetitive arithmetically intensive computational algorithms [1, 2]. Typical desk-top GPUs such as the NVIDIA GeForce 8800 GPU are extremely fast, programmable and highly powerful, precision multi-processor with 128 parallel stream processors, which are used also for general-purpose computations today[3, 4]. Over the past few years the programmable GPU has turned out into a machine whose computational power has increased tremendously [5, 6]. The application of GPU for general-purpose computations (GPGPU) is seen as a significant force that is changing the nature of performing parallel computations [7, 8]. The phenomenal growth in the computing power of GPU that can be measured as Giga floating point operations (GFLOPS) over the years is shown in Figure 1 [9].

DOI : 10.5121/ijaia.2013.4202

The internal memory bandwidth of NVIDIA GeForce 8800 GTX GPU is 86 Giga Bytes/second, whereas for a dual core 3.0 GHz Pentium IV CPU it is 8 Giga Bytes/second. GeForce 8800 GTX has peak performance of about 518 GFLOPS with 32-bit floating-point precision compared to approximately 25 GFLOPS for the CPU, showing growths and benefits from such raw computational power of the GPU [10, 11].

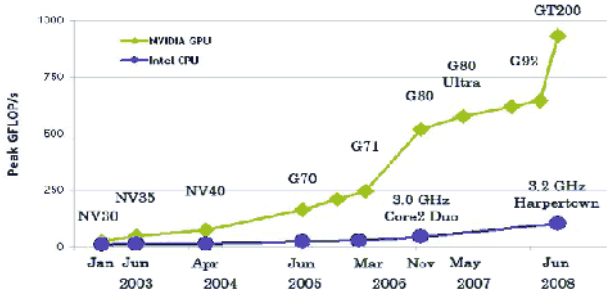


Figure 1. Comparison of Computational Growth between CPU and GPU (Courtesy: NVIDIA).

Clustering is a very important task with vast applications in the field of data-mining in various domains [12, 13]. Hierarchical Agglomerative Clustering (HAC) is an important and useful technique in data mining which produces 'natural' clusters with the flexibility for identifying the number of clusters using hierarchy. Research in microarrays, sequenced genomes and bioinformatics have focused largely on algorithmic methods for processing and manipulating vast biological data sets. Identifying meaningful clusters, interesting patterns in large data sets, such as groups of gene expression profiles is an important and active area of such research. Hierarchical clustering has been known to be effective in microarray data analysis for identifying genes with similar profiles and enables pattern extraction from microarray data sets [14]. HAC is a common but very important analysis tool for large-scale gene expressions [15], DNA microarray analysis [16, 17], revealing biological structures etc. Detecting clusters of closely related objects is an important problem in bioinformatics and data mining in general [18]. Such applications of HAC in bioinformatics will highly benefit if the processing time taken by the algorithm could be significantly accelerated. The HAC algorithm starts with considering each point as a separate cluster and iteratively agglomerates the closest pair of clusters until all points belong to a single cluster. The computations of distances and identifying the closest pair contribute to high time and memory complexities. Using a simple dissimilarity matrix of size $O(N^2)$ would require memory of size $O(N^2)$ for a data set with N data points. The HAC centroid method based on priority queue algorithm also has time complexity $O(N^2LogN)$. Efficient techniques to handle large data sets based on sampling and summarizing have been proposed in BIRCH and CURE [19, 20]. Efficient parallel algorithms on CPU have been proposed for HAC by Dash et. al. [21, 22]. Clustering techniques such as *k*-means, Fuzzy *C*-Means (FCM) and the traditional HAC, algorithms on GPU have been implemented, achieving remarkable speed gains over the respective CPU implementations by various researchers [23, 24, 25, 26, 27]. A scheme which exploits two types of fine-grain data parallelism at the different levels in the nearest neighbour search part of the data-clustering process is proposed by Takizawa et.al. [28]; efficient generation of histograms are achieved using GPU Ziegler et.al. [29]. The intense use of GPUs for multitudes of general purpose computations via parallelization of computational tasks and large data have penetrated into various domains of which the field of data mining has prominent benefits [30, 31].

In this research work we describe and implement an efficient data partitioning method based HAC on the GPU. We utilize the parallel hardware architecture of the GPU to create partially overlapping partitions (PoP) in the given data and to parallelize the data-independent computations in HAC. We compare the computational speed of this partitioning based HAC method on the GPU with the traditional HAC implementations on the GPU and CPU on typical

desk top computers. We also discuss the resultant reduction in time and memory complexities while using GPU versus the CPU. The data partitioning based HAC method on the GPU helps to minimize bottlenecks due to time and memory complexities. From our studies we find that with an understanding of the parallel architecture of the GPU, a researcher will be able to make correct implementation choices for computationally intense and data independent algorithms on the GPU. It also helps to realize that the proper use of the parallel computing features of GPU gives significantly high computational efficiency versus the structural computing capabilities of CPU without jeopardizing the accuracy of clusters.

2. HIERARCHICAL AGGLOMERATIVE CLUSTERING ALGORITHMS

In HAC each observation or vector is treated as a singleton cluster to start with. The vectors are successively merged into pairs of clusters (agglomerative) until all vectors have merged into one single large cluster. The agglomeration of clusters results in a tree-like structure called the dendrogram. The objective of HAC is to generate high level multiple partitions in a given dataset [32]. The groups of partitions of data vectors at any level will denote sets of clusters. In this bottom-up approach, the similarity between the merging clusters is highest at the lowest level of the dendrogram and it decreases as the clusters merge into the final single cluster. This structure can then be used to extract clusters by cutting the dendrogram at the required level of similarity or number of clusters expected. Parallel algorithms for single linkage HAC were proposed for arrays [20]. In [33], Olson gave optimal algorithms for HAC using PRAM, butterfly, and tree architectures. Other efficient parallel algorithms have been proposed on the CPU with improvements over previously known algorithms [34]. The common feature of the above algorithms is that the task of computing and maintaining $O(N^2)$ dissimilarities is divided among processors. These algorithms are not very efficient because they still require $O(N^2)$ total memory, and per iteration they require to update $O(N^2)$ memory [35].

We propose an effective method of computing the traditional HAC on the GPU and a very efficient partitioning scheme to implement HAC on the GPU. It alleviates time and memory bottlenecks without compromising accuracy. Extensive empirical study has shown that, except for a number of top levels of the dendrogram, all lower level clusters are very small in size and close in proximity to other clusters. This characteristic is named as the '90-10 relationship' which means roughly about 90% of iterations from the beginning merge into clusters that are separated by less than about 10% of the maximum merging distance [36]. The partially overlapping partitioning structure is proposed for both two-dimensional (2-*D*) and high-dimensional data based on this 90-10 relationship. We restrict ourselves to 2-*D* implementation and leave the high-dimensional implementation for future work. In PoP, data is partitioned into p number of overlapping cells. The region of overlapping is called δ-region where, δ is the separating distance. Figure 2 shows a schematic diagram to illustrate how PoP divides the data-space uniformly into p cells. Each cell includes the core region and the adjacent δ-regions. For centroid metric (and other geometric metrics) each cluster is represented by a single representative point. If the representative point of a cluster falls in a δ-region then each affected cell holds it, otherwise (i.e., the core region) only one cell holds it.

The basic idea of PoP is that per iteration, the closest pair is found for each cell independent of all other cells, and from those the overall closest pair is found. If the overall closest pair distance is less than δ, then the pair is merged and dissimilarity matrix of only the container cell is updated. If the closest pair or the merged cluster is in a δ-region then the dissimilarity matrices of the affected cells are also updated. Initially δ is set to a very small value and p to a large value. Gradually δ is increased by say $x\%$ and p is decreased by $y\%$. Complexity analysis for priority queues algorithm has shown that the traditional algorithm requires a time complexity of the order of $O(N^2 logN)$. But PoP HAC will have an approximate time reduction in the order of $O(p)$

compared to the traditional algorithm which requires a time complexity $O(N^2)$. Memory requirement analysis of priority queues algorithm (stored matrix) is similar to that of dissimilarity matrix. The memory requirement of 'stored data' is $O(N^2)$. PoP HAC will have approximately $O(N)$ reduction in memory complexity.

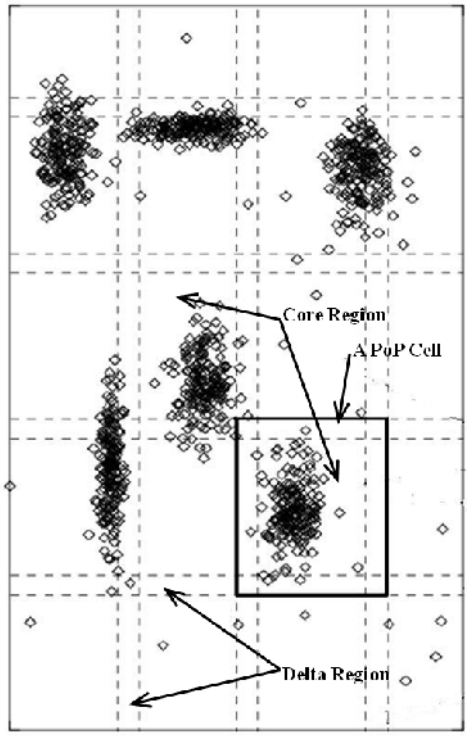


Figure 2. Partially Overlapping Partitioning Cells with Core and Delta regions

2.1. Existing HAC implementations on GPU

There are at least two Graphics Library and Shading Language (GLSL) based implementations of Hierarchical clustering on the GPU [37, 38]. In these implementations the parallelism and programmability of the graphics pipeline are exploited and employed for accelerating the HAC algorithm. The data representation is done by using one or more *2-D* textures with RGBA as the internal format. The distance matrix computations, minimum distance calculation and the cluster merging and distance updates are performed in the GPU using appropriate shaders. The proposed clustering methods on the GPU accelerate the clustering process to a maximum of four times when compared with the CPU used for clustering.

Extensive literature search for CUDA based hierarchical clustering and distance computations yielded in two other related works with significant contribution to this topic. The first work deals with the implementation of the pair-wise distance computation, which is one of the fundamental operations in HAC [24, 39]. The GPU NVIDIA 8800 GTX is employed and it uses CUDA programs to speed up the computations. Gene expression data is used and the traditional HAC algorithm is implemented using half matrix for pair-wise distance computation. The shared memory of the GPU is used and the threads are synchronized at block level. Results show that speedup of 20 to 44 times is achieved in the GPU compared to the CPU implementation. It is important to note that this speed up achieved is only for the pair-wise distance computations and not for the complete HAC algorithm. In the second research work, CUDA based HAC algorithm

on the GPU NVIDIA 8800 GTX is compared with the performance of commercial bioinformatics clustering applications running on the CPU [40]. Based on the cluster parameters and setup used, speedup of about 10 to 14 times is achieved. The effectiveness of using GPU with CUDA to cluster high dimensional vectors is shown.

2.2. Limitations and Research Issues in the existing GPU Implementations

We summarize the limitations that arise from the previous implementations of the CUDA based traditional HAC algorithm.

(1) The maximum number of observations that can be clustered is limited due to the memory size of GPU hardware.
(2) The availability of sufficient number of threads within a block to complete the computations affects the time complexity.
(3) The use of lower bandwidth global memory, focusing on the minimization of data transfers between the device memory and the global memory.

One possibility of avoiding data transfers is by simply re-computing whenever needed instead of transferring back and forth from memories to the processors provided efficiency is not at cost.

2.3. Motivations to Implement HAC and PoP HAC on GPU with CUDA

Data mining algorithms are often computationally intense and repetitive in nature which exhibit rich amounts of data parallelism. Data parallelism is a characteristic of a computational program whereby arithmetic operations can be performed on data vectors simultaneously. In the unified design as in CUDA, it is possible to execute multiple shaders by synchronizing them to the various graphics memories. This unified design provides better workload balance between the stream processors in the GPU, thus avoiding delays in completion of shading. The parallel processing architecture, the large global memory space, and the programmability of the GPU is to be exploited to efficiently implement the traditional HAC algorithm. PoP HAC is significantly more efficient than the traditional algorithms in both time and memory complexities making it attractive to be deployed in the GPU. In order to efficiently use the GPU to perform HAC clustering, the following limitations are to be addressed.

(1) Insufficient threads within blocks to perform computations and necessary synchronization.
(2) Lack of memory in the GPU for performing clustering of large data sets.
(3) Time complexities in the CUDA based traditional HAC which is same as the time complexity of HAC on CPU.
(4) Usage of the small but fast shared memory versus the large but slower global memory in programmability.

Apart from the memory and time complexity benefits that PoP brings, the following are the factors that make the implementation effective on the GPU when compared with the implementation of traditional HAC:

(1) The fact that PoP effectively reduces the effective number of distance computations, it removes or minimizes the bottleneck of having insufficient threads within a block to perform the distance matrices.
(2) Due to the reduced number of combinations of simultaneous distance computations it is possible to cluster data sets with large number of observations.
(3) It is also possible to program the GPU in a way to assemble and assign the partitioned PoP cells to the blocks in CUDA thus making it efficient and reusable.

(4) Simultaneous computation of distance matrices of the PoP cells is possible by invoking the threads within the blocks.

Memory complexity of the traditional HAC is reduced due to the use of independent cells in PoP HAC. In CUDA the queuing of threads is minimized since the distance matrices of the PoP cells are much smaller. Since parallel computation of the distances is possible on the blocks, the time complexity is further reduced by '*p/q*' where *p* is the number of partitioned cells and *q* is the number of blocks. These advantages are realized using CUDA by optimizing memory usage making most use of the large global memory on the GPU.

3. CUDA FOR GENERAL PURPOSE COMPUTATIONS

The learning and understanding of the general structure of the C language based CUDA is vital to effectively use it for GPGPU [43, 44]. The software stack of CUDA runs on the GPU hardware as an Application Programming Interface (API) to the standard C language, providing Single Instruction Multiple Data (SIMD) capabilities.

3.1. CUDA for realizing Data Parallelism on GPU

In general, the programming model of GPU remained harsh, constrained and heavily oriented towards computer graphics. Hence, general-purpose computational algorithms have to be carefully designed and ported, to be effectively executed on the graphics environment [45]. The challenge is to harness the power of GPU for non-graphic general-purpose computations such as the HAC algorithm. For this purpose, researchers had to learn graphics dedicated programming platforms such as OpenGL or DirectX, and convert the computational problem into a graphics problem until recent past [26, 27, 46]. In these legacy approaches, algorithms involved are broken down into small chunks of computations or kernels called the shaders using GLSL or C_G [37, 47]. These approaches require tedious programming efforts and are time consuming. As an upturn, CUDA, a new graphics API of NVIDIA, lightens the effort required on such tasks by researchers. CUDA gives developers access to the native instruction set and memory of the parallel computing elements in GPUs effectively making GPUs become open architectures like the CPU [44, 48]. The skills and techniques needed in invoking the internal parallel processors of a GPU are viable to data mining researchers who might not be expert graphics programmers through the numerous applications on various computations and technical reports made available to researchers to start with [49].

3.2. Data parallelism on GPU using CUDA

The CUDA Single Instruction Multiple Thread (SIMT) architecture is analogous to the Single Instruction Multiple Data (SIMD) vector architecture and works on threads rather than organized vector widths. The SIMT architecture of GPU multiprocessors enables researchers to write data parallel and independent program codes for coordinated threads. CUDA allows the structuring of the algorithm to expose as much data parallelism as possible by efficiently mapping each kernel to the device memories. For instances, in the HAC algorithm, the similarities can be computed in parallel and in PoP the identification of minimum distances in each cell can be done in parallel.
To utilize the GPU such as NVIDIA GeForce 8800 as a stream processor, the CUDA framework abstracts the graphical pipeline processors, memories and controls. This exposes the memory and instruction set as a hierarchical model so it can be effectively used to realize high-level programmability on highly intensive arithmetic computations. Chapter 2 of the CUDA programming guide, [9] explains the programming structure, memory management and the invocation of kernel functions in detail. Figure 3 shows an overview of the CUDA device

memory model for programmers to reason about the allocation, movement, and usage of the various memory types such as Shared, Local, Global, Constant and Texture.

The lowest level of computation is the thread which is analogous to shaders in OpenGL. Each thread has local data memory and access to memories at different hierarchies. Instructions are passed into the threads to perform calculations. Threads are organized into blocks, and blocks are organized in grid. Blocks form the basis for a kernel to operate on the data that resides in a dedicated, aligned memory. Threads in a block are identified by a unique *Thread ID* and blocks in a grid by a *Block ID*. The *ID* of the threads can be accessed via the corresponding blocks and grids by using these built-in variables: *blockDim (block dimension), gridDim (grid dimension), blockIdx (block index), threadIdx (thread index), and warpSize (size of a warp)*. A warp executes a number of threads on one of the processors within the GPU. When a kernel is executed, the blocks will be distributed to the many processors and to each processor a number of threads are assigned by a warp. Such an organization will allow us to control the distribution of data and instructions over the threads. At any moment, a thread is operated by only one kernel although many kernels can be queued up in a stream. All threads in the grid are executed concurrently ensuring fast parallel computation. Threads need to be synchronized at the end of kernel operation to ensure that all data have been processed completely. However, synchronization must be used sparingly as it slows down the computation.

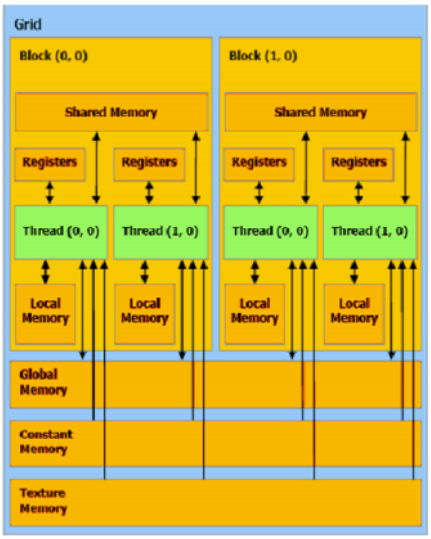


Figure 3. CUDA Memory Model (Courtesy: NVIDIA).

4. Efficient Implementation of Traditional HAC and PoP based HAC Methods

4.1. Traditional HAC Algorithm on the GPU

CUDA based HAC traditional algorithm has been implemented and we discuss our implementation in this section [41]. An iteration of the implementation which is comprised of the computational tasks of the traditional HAC algorithm and the corresponding CUDA functions used are summarized in Table 1. The process of updating the similarities matrix and merging the clusters are repeated until only one cluster exists. The clustering approach using the GPU outputs exactly the same dendrogram formed from the CPU implementations. In this section we briefly discuss the construction of the similarity matrix in the global memory of the GPU. The similarity

matrix S_{ij} is obtained by splitting the task of computing Euclidean distances between several threads as follows:
1) Each block in the grid is responsible for computing one square sub-matrix S_{sub} of S_{ij}.
2) Each thread within the block is responsible for computing one element of S_{sub}.

Table 1. An Iteration of the Centroid HAC Algorithm in GPU with CUDA

Tasks in HAC	**Functions Implemented on GPU**
Transfer input vectors from CPU to GPU	*CudaMemcpyHostToDevice();*
Populate initial similarity half matrix	*CalculateDistance();* *// k* index locates position in //the 1-D array
Identify minimum distance vectors	*cublasIsamin()*; //built in function of CUDA
Merge / Calculate new cluster vector & Update	*updateArray0()*; //calculates Centroid
Update the similarity half matrix	*UpdateArray1();* *UpdateArray2();*
Update the Minimum Distance Array	*UpdateArray3();*
Transfer Cluster data from GPU back to CPU	*CudaMemcpyDeviceToHost();*

The number of threads per block and the number of blocks within the grid should be chosen to maximize the utilization of the GPU's computing resources. This warrants that there should be as many blocks in total as the number of processors in the GPU. To be efficient we need to maximize the number of threads and allow for two or more blocks to run concurrently.

Each CUDA processing block is run on one multiprocessor. For a GPU with 128 processors, utilizing all processors with maximum number of threads will maximize the efficiency. If the dimension *blocSize* is defined as 4 x 4, then, 128/16 = 8 blocks per grid is considered optimal. When the dimension *blocSize* is 4, and if 4 blocks are used per grid, then 16 * 4 = 64 threads operate on the data per grid. While computing distances between vectors in the HAC algorithm, if there are 16 dimensions per vector, with the above block structure in one pass the distances between the first vector and four other vectors can be computed. Thus each grid with 64 threads behaves like an 'operator' on the data while the kernel is executed.

The following steps illustrate the computation of similarity matrix using the GPU.

(1) Read vectors to the *Dev_data* array on GPU
(2) Calculate index using expression in Equation (1)
(3) Store the distances in the *Dev_dist* array on GPU
(4) Compute minimum distances and Merge
(5) Update *Dev_data* array and *Dev_dist* array
(6) Repeat steps 1 to 5 till all vectors are to compute the matrix are exhausted.

We use *2-D* array for the data structures in CUDA based implementations [50]. But for minimum distance computations we find that the CUDA library function *cublasIsamin()* allows efficient reading and writing to continuous memory addresses which works only while using *1-D* linear array. But the use of *2-D* array in the global memory instead of the traditional *1-D* array showed significant improvement in the computational performance. As shown in Figure 4 even though the physical view of the *1-D* array is linear, in the logical view, it is still a *2-D* array. Figure 5 shows the access to the *1-D* memory array using *2-D* to *1-D* conversion index *k*. The expression for the conversion index *k* is shown in Equation (1).

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

The index *k* is used to get the address of the *1-D* similarity array as it is related to the index *(i,j)* of the corresponding vectors. The resultant index *k* gives the location of each individual cluster. The block and grid size are also parameters that influence the computational speed while using the GPU. The kernel that runs on the GPU to compute the similarity matrix for a 4 × 4 block is shown in Table 2. It can be seen that in order to execute the kernel on the GPU, the block and the grid size has to be declared before the computations are coded including the computation of the memory location index *k*. The *calculateDistance()* launches the kernel to compute the PoP distances in parallel. It can be seen that in order to execute the kernel on the GPU, the block and the grid size has to be declared before the computations are coded including the computation of the memory location index *k*. The function *calculateDistance()* launches the kernel to compute the PoP distances in parallel.

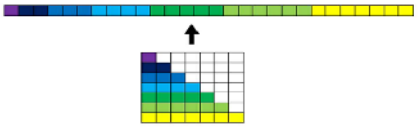


Figure 4. *2-D* to *1-D* Memory Mapping for Half Similarity Matrix.

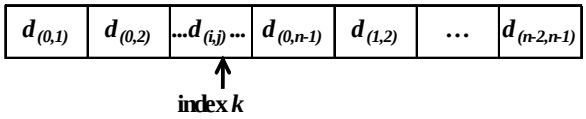


Figure 5. *2-D* to *1-D* Array Access using '*k*' index.

Table 2. Pseudo code to Create Half Similarity matrices in the GPU

Sample CUDA Codes running on GPU

__global__ void calculateDistance (...):
dim3 blocSize; blocSize.x=4; blocSize.y=4;
dim3 gridSize;
for (int i=0; i<n-1; i++) // for each value of i
{ int m=n-1-i;
int x=i*(n-1)-i*(i-1)/2;
gridSize.x=m/16+(m%16!=0)*1;
for (int k=0; k<d; k++) // for each dimension
{calculateDistance<<<gridSize, blocSize>>> (dev_data|k|, dev_dist, i, n, x); }}

4.2. CUDA based PoP HAC Algorithm on GPU

In this section the CUDA based programming architecture that is employed to realize heavy data parallelism with the PoP HAC algorithm on the GPU is discussed. In the HAC algorithm, to start with, each observation is considered as a cluster and is identified by an index. The index of the individual cluster is stored in a *1-D* array *'minIndex'* in the GPU. Figure 6 shows the CUDA thread block execution architecture for the computations on the PoP cells. There are a few key differences between the programming architecture used to implement the traditional HAC algorithm on GPU and the PoP HAC.

(1) In the PoP HAC implementation, the task of computing distance matrix is accomplished by employing several blocks whereas only one block is used in the traditional HAC on GPU.
(2) In the PoP HAC using CUDA, each partitioned data cell is assigned to a block. Each block consists of several threads that compute the distances between the pairs of points within that block. During execution, each thread within a block is responsible to compute the corresponding pair's distance in the distance matrix.
(3) Data parallelism is achieved in the GPU PoP HAC through initialization of the functions in all the blocks and via merging operations.

In the traditional HAC on GPU, the threads within a block perform distance computations, which is limited to maximum number of threads possible within a block; this leads to queuing of threads and delay in processing. In the PoP HAC on GPU, threads within the multiple blocks execute this function of distance computation, identification of minimum distance pair, updating and merging of the minimum distance cluster pair. This implementation architecture gives additional performance gains which further complements the benefit of less computational overheads that is achieved via the partitioning of data in PoP HAC compared to the traditional.

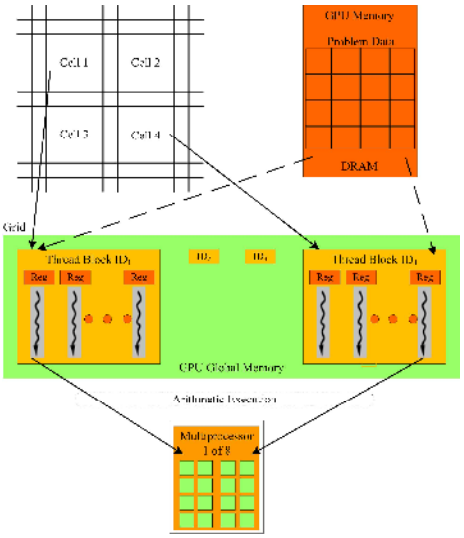


Figure 6. PoP Cells Thread Block Execution Architecture with CUDA.

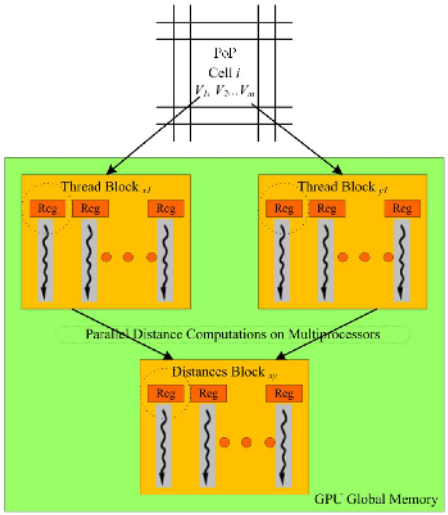


Figure 7. Mapping of PoP Cells to Thread Blocks and Distance Computation.

4.3. Parallelization of the PoP Computations on the GPU using CUDA

The PoP cells are created in the CPU. Each cell data is then transferred to the global memory of the GPU and are located via cell index and pointers to each cluster it contains. The implementation of distance matrix for each cell still uses *1-D* array but the collection of all *1-D* array forms a *2-D* array of distance matrix. The minimum distance of each cell is found and is stored in the array '*cellMin*' along with the index number of that distance in its particular distance matrix. Then the minimum distance among all the PoP cells is identified and the overall minimum distance is found and the corresponding pair of clusters is merged. For subsequent iterations, '*d*' is increased and '*p*' is decreased in the CPU and the values are transferred to and used in the GPU. The pseudo codes for the creation of PoP cells using blocks and initialization of the corresponding distance matrices are listed in Table 3. In the pseudo code, *'Dim3*' is a data structure that can contain up to 3 dimensions that is provided by CUDA. The program creates blocks '*dimBlock*' that has size *blockSizeX* * *blockSizeY*. Then it computes the number of blocks that are needed which is the '*noOfCell*' in that row divided by the '*size*' (*area*) of the block. The *gridsize* is declared as '*noOfBlock*' * 1 and the kernel is executed; each thread computes the distances for one cell. Figure 7 shows the CUDA Programming Model for mapping of PoP cells to the '*Thread blocks*' and the scheme to perform distance computation. The minimum distance of each PoP cells are computed and are stored in the *cellMin* array along with the index of each data point. The program code as shown in Equation (2) is used to retrieve the minimum distance of each PoP cell and this is executed in parallel across all the PoP cells. This code finds the overall minimum distance from the minimum distances of all cells and it returns the cell index. The minimum distance index from the '*minIndexList*' is used to find out the pair of clusters that has the corresponding minimum distance. The overall minimum distance δ of all the PoP cells is

then compared with the overlapping distance of the PoP cells. If the overall minimum distance is larger than δ, then δ will be incremented and the minimum distance computations are continued. Else the distance matrices are updated again and the computation for the next iteration continues.

$$index = cublasIsamin\ (noOfQuadrant,cellMin,1)-1 \qquad (2)$$

Table 3. Pseudo code to Create PoP cells using Blocks in the GPU.

Sample CUDA Codes running on GPU

__global__ void calculateDistance (...):
dim3 blocSize; blocSize.x=4; blocSize.y=4;
dim3 gridSize;
for (int i=0; i<n-1; i++) // for each value of i
{ int m=n-1-i;
int x=i*(n-1)-i*(i-1)/2;
gridSize.x=m/16+(m%16!=0)*1;
for (int k=0; k<d; k++) // for each dimension
{calculateDistance<<<gridSize, blocSize>>> (dev_data|k|,
dev_dist, i, n, x); }}

5. EXPERIMENTAL RESULTS AND DISCUSSIONS

Bio-informatics is our choice of domain and data for experimentation of the computational implementations. Bio-informatics is a critical field of study where clustering algorithms are applied at large for various research and medical purposes. Of which, hierarchical clustering methods are often used for studying groups in either genes or experimental samples or for both. Such clustering approaches help to identify genes that are co-regulated or that participate in similar biological processes. This can be further used for promoting prediction and gene function predictions, thus to find new potential tumour subclasses [51]. We have selected 'yeast micro array gene expression data' for testing the efficiency of our GPU implementations. There are 64000 genes each with 31 dimensions. Though gene expressions may be in the order of thousands, usually only a few thousand of the gene samples are chosen for clustering.

5.1. Experimental Setup and Planning

The implementations were run on a desktop computer with Pentium Dual Core CPU, 1.8GHz, 1.0GB RAM, MS Windows XP Professional 2002 and using MS Visual C++ 2005 for the development. Initial experiments on the traditional HAC CUDA implementation were conducted on a NVIDIA GeForce 8800 GTS with a memory of 512MB with CUDA2.0. Experiments on the PoP HAC CUDA implementation were conducted using the NVIDIA 9800 GTX with a memory of 512MB. The CUDA libraries and toolkits as in version 2.3 of CUDA were used to develop codes for the traditional HAC and PoP HAC. The implementation of CPU PoP is similar to PoP GPU but the entire algorithm is run sequentially by the CPU. The number of observations to be clustered was varied from 256 to 100k genes with higher dimensions for the traditional HAC and with two dimensions for PoP HAC. The details of the results of the experiments conducted are further discussed in this section. The computational performance of the GPU over the CPU is expressed in terms of *'Speedups'*, which is the ratio of the CPU computational time to the GPU computational time. The total computational time for the complete clustering algorithm taken by the CPU and the GPU were measured for computing the *'Speedups'* which is a measure of the computing efficiency of GPU over the CPU. The following experiments were planned, conducted and the results are analyzed.

5.2. Experimental Results and Discussions on Traditional HAC Implementations

5.2.1. Performance Evaluation - Number of Blocks in CUDA versus Number of Genes

The objective of this experiment is to evaluate the effect of *'Number of CUDA blocks'* used in the GPU, *'Number of Genes'* and *'Gene Dimension size'* to be clustered on the GPU computational efficiency metric *'Speedups'*. CUDA blocks of size 4, 6, 8 and 16 per grid are used with increasing number of the Yeast genes from 10k to 15k each and with varying dimensions 6 and 31 [52]. The experimental computations of the traditional HAC algorithm are conducted both on the CPU and the GPU. The total computational time in the CPU and GPU to complete clustering are measured.

The computational efficiency of GPU in terms of *'Speedups'* are computed. The experimental results are plotted as shown in Figure 8. It can be observed that the speedup efficiency of the GPU while using 8 CUDA blocks is slightly better than using 4, 6 or 16 blocks per grid, though the difference is not significant. The impact of increasing number of genes on the computational efficiency of the GPU is evident when the dimension size is 6. For genes with 31 dimensions, there is hardly any difference in performance while using different number of CUDA blocks. The 8800GTS hardware with 512MB has 128 core internal processors [53]. Our study indicates that using 8 blocks the performance level is better at certain instances else almost the same. It can be noticed that the overall computational efficiency of the GPU drops as the gene sizes are increased. The drop in performance with increase in number of genes and its dimensions is attributed to the fact that we use a simple method of memory management in CUDA; i.e. the use of large but slower global memory instead of the shared or local memory which is faster and specifically meant for the threads within a block. Hence as gene size and gene dimension increases the efficiency of GPU drops; yet faster than the CPU by about 33 times.

5.2.2. Speedup Profile determination

The objective of this experiment is to evaluate the effect of *'Number of CUDA blocks'* and the *'Number of Genes'* to be clustered on the peak performance of the CUDA implementation of HAC [41]. The speedup efficiency profiles of the GPU with varying number of CUDA blocks while increasing the number of genes with 31 dimensions are shown in Figure 9. It can be seen from the speedup profiles that the performance peaks when there are 8k to 11k genes to be clustered and that is the region where the speedup is significantly higher while having 8 blocks per grid. In an 8800 GTS GPU there are 16 multiprocessors with 8 internal processors each that makes the total number of core internal processors 128. Thus while using a block size of 8, up to 8 * 8 = 64 threads can be used in the computations. Each thread independently operates on a vector and thus exploits the parallel computing power of the GPU. Internal processors which do not belong to a block cannot be accessed by that block. So there will be a maximum of 8 execution cycles to process 64 threads while the block size is 8. If 8 such blocks can be used simultaneously, then all the 128 internal processors can be used simultaneously, thus fully harnessing the power of the GPU. This also explains why the speedups with block size of 8 are higher [52]. The optimal number of blocks to be used can be 8 irrespective of the number of genes to be clustered and their dimensions, which is the same as the theoretically calculated optimal number of blocks per grid on the 8800GTS with 512MB GPU. A peak performance of 46 times speedup is achieved with 31 dimensional 10k genes.

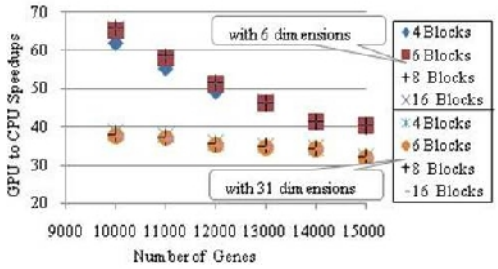


Figure 8. Performance of GPU HAC based on Blocks per Grid versus number of Genes and Dimensions.

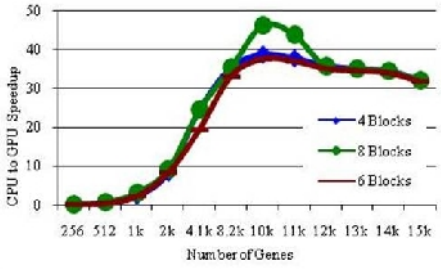


Figure 9. Speedup Profiles of GPU versus Blocks per Grid and number of Genes.

5.2.3. Effect of using Global memory on Scalability

The objective of this experiment is to understand the effect of using Global memory with varying gene sizes and dimensions on the Speedup performance of the GPU [41]. In previous GPU based CUDA HAC research works [38, 39] shared memories of CUDA are used instead of the global memory. The shared memory can be accessed 150 to 200 cycles faster than accessing the global memory of the GPU, but it comes at an expense. The shared memory is much smaller than the global memory, so the data and the distance matrix have to be split into multiple chunks. Thread alignment becomes critical while sharing common data by the threads and coding is tedious. Though the global memory gives relatively slower performance, it is the largest memory in the GPU, it is easier to code, all threads are able to access the data in common thus little memory management is required. Figure 10 shows that while using our global memory management the speedup drops tremendously when the dimensionality is over 100. The performance can be reversed if shared and local memories of the blocks are used. We propose that scalability trade-offs can be made between choice of memory, ease of programming and memory management for 100 or less dimensions.

5.2.4. Speedup Performance of Single linkage versus complete linkage HAC methods

The fundamental assumption in HAC is that the merge is monotonic and descending, which means that the combination similarities s_1, s_2, s_3,…, s_{n-1} of successive merges of the vectors are only in descending order. Cluster pairs are usually merged based on minimum distances and that method of HAC is called as the 'single linkage'. Cluster pairs can also be merged based on maximum distances which is called as HAC 'complete linkage'. The objective of this experiment is to compare the speed-up performance of both the single linkage and complete linkage HAC methods on the GPU compared to their CPU implementations [52]. Varying Gene sizes from 256 to 15k with 31 dimensions were used. Figure 11 shows the speed-up comparison between the single link and complete link methods of HAC. It is found that the single linkage method can be about 35 times faster than the CPU implementation when there are 10000 Genes to be clustered, whereas the complete linkage method reaches only about 20 times the speed of the CPU implementation. In single link HAC on GPU the minimum distance pair is identified virtually in one pass using the built in '*cublasIsamin*' function in CUDA whereas the identification of maximum distance pair in complete linkage needs a custom developed function; thus the single linkage implementation is more efficient on the GPU.

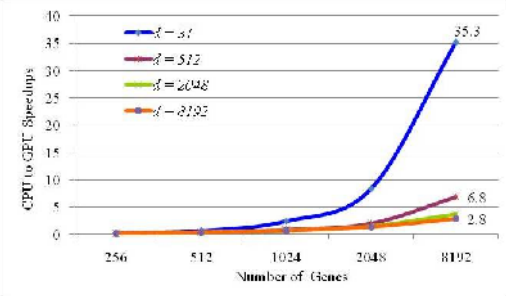


Figure 10. Traditional HAC Algorithm: Speedup Profiles of CPU to GPU while varying Gene dimension size.

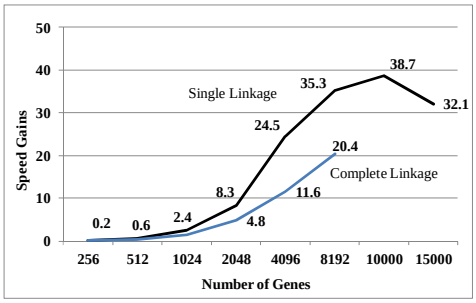


Fig. 11. Performance profile of HAC methods on GPU using CUDA: Single and Complete linkage vs. number of Genes.

5.3. Experimental Results and Discussions on PoP HAC Implementations

In this section we discuss the experiments conducted to evaluate the speedup performance of the PoP HAC algorithm on the GPU. We use the NVIDIA 9800 GTX GPU with 512MB memory. Here we intend to evaluate the computational performance of the PoP HAC on GPU and compare it with the performances of the traditional HAC implementations on GPU and CPU with various gene sizes. We study and compare the memory requirements of the PoP HAC and the traditional HAC on the GPU while using global memory. Moreover, in the PoP HAC method one of the variables in the algorithm is *p*, which is the number of partitions created in the dataset. We intend to compare the performance of the GPU with the CPU while varying *p* else it is set at 25. We use the 'yeast micro array gene expression data' for this purpose. The following experiments were planned, conducted and the results are analyzed.

5.3.1. Performance Evaluation of PoP HAC on GPU with traditional HAC on GPU and CPU versus Number of Genes

The objective of this experiment is to evaluate the speedup performance of the PoP HAC on the GPU and compare it with the traditional HAC on GPU [42]. The traditional HAC on GPU versus on CPU is also evaluated. The computational performances of the GPU vs. the corresponding CPU implementation are shown in Figure 12 along with the table of computational time in the GPU and CPU. It can be seen that PoP GPU is up to 6.6 times faster compared to the traditional HAC in GPU and about 443 times faster compared to the CPU for 15k data points. The traditional HAC on GPU is about 67 times faster than the traditional HAC on the CPU. For data size of 20k, the PoP GPU can run 2331 times faster compared to the CPU. It can also be noted that the PoP HAC computations take about seven times lesser than when using the GPU to compute the traditional HAC. Certain speedup calculations are not done because the corresponding experiments on the CPU took too long to complete and those traditional HAC on GPU could not be completed due to memory requirements. Computations beyond 15k points are very time consuming on the CPU and the traditional HAC on GPU has memory access limitation to hold the half distance matrices; which is 512MB in the 9800 GTX.

5.3.2. Evaluation of Memory Requirements

One of the motivations to develop partitioning based HAC method on the GPU is due to the lack of memory in the GPU for performing clustering of large data sets using traditional HAC as discussed in section 2.3. Computing the half distance matrix is the arithmetically intense computational step which in the traditional HAC on GPU still has high memory complexity [52]. The objective of this experiment is to evaluate the memory requirements of both the traditional

HAC and the PoP HAC methods on the GPU and to compare the same while varying the number of genes [42]. For this purpose, the memory required by the GPU for the computations of both the HAC methods was measured. Figure 13 shows the comparisons of memory requirements while varying the gene sizes. It can be noticed that the PoP HAC method on GPU uses far less memory compared to traditional HAC. For 100k data points PoP HAC on GPU uses only around 67MB while traditional HAC on GPU would use up to 28GB which is about 418 times less memory. It can be concluded that the PoP HAC on GPU uses less memory in the *O(logN)* where *N* is the number of observations clustered.

Data Points	Computational Time (Seconds)			CPU to GPU Speedups		
	PoP HAC GPU	HAC GPU	HAC CPU	HAC GPU vs. PoP GPU	HAC CPU vs. PoP GPU	HAC CPU vs. HAC GPU
1000	1.59	2.92	1	1.84	0.63	0.34
5000	8.46	12.16	350	1.44	41.37	28.78
10000	14.62	70.94	3817	4.85	261.08	53.81
15000	23.5	155.8	10404	6.6	442.7	66.8
20000	34.37		80158		2332.2	
50000	68.66	-	-	-	-	-
100000	196.1					

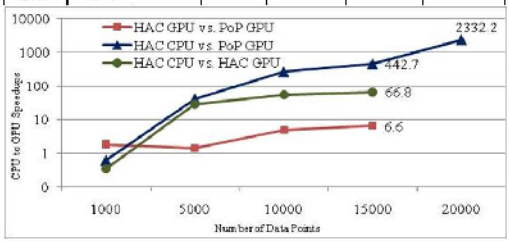
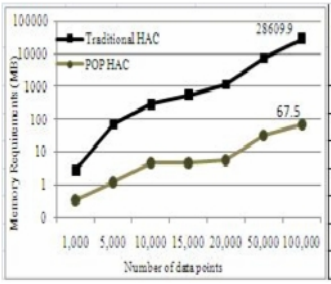


Figure 12. HAC Algorithm CPU to GPU Speedups and PoP Algorithm CPU to GPU Speedups versus Number of data points.



No of Data points	Memory Requirement on GPU (MegaByte)	
	Traditional HAC	POP HAC
1,000	2.9	0.3
5,000	71.5	1.2
10,000	286.1	4.6
15,000	550.0	4.9
20,000	1144.4	5.6
50,000	7152.4	32.7
100,000	28609.9	67.5

Figure 13. Memory Requirements of Traditional HAC and PoP HAC on GPU versus Number of data points

5.3.3. Effect on number of Partitions: Computational Time Comparison between PoP HAC on GPU versus CPU

The number of initial data partitions p is a variable in the PoP HAC algorithm, which effectively reduces the number of observations within each cell in a given data set for clustering. The objective of this experiment is to evaluate the computational time taken by PoP HAC on GPU and CPU for various numbers of data partition cells *p*. The experiment was conducted using *2-D* 100k data points and the initial cell size was varied from 25 to 1600 [42]. The computational time for PoP HAC on NVIDIA 9800 GTX GPU and a dual core CPU were measured and the comparison is shown in Figure 14. It is noted that the GPU is more efficient than the CPU when the number of PoP cells *p* is about 100 or below. When *p* is above 100 the CPU computes faster than the GPU. This phenomenon is due to the fact that as *p* increases, the number of observations per PoP cell decreases. This in turn reduces the effective data size per computation on the GPU. Moreover, the number of data transfer between CPU and GPU increases with increase in *p* and this shadows the speed benefit of using the GPU. This study again confirms that the efficiency of using GPU for PoP HAC would be masked if too little observations per cell are computed with relatively large number of partitions made in the data set. Overall, the results of the experiments signify that while data size is sufficiently large, efficient partitioning like PoP can maximize the advantage of using GPU, by further alleviating time and memory bottlenecks without compromising accuracy. Thus it is important to maximize the utilization of the GPU hardware for high efficiency.

5.3.4. Research Issues and Limitations addressed using Data Partitioning based GPU HAC

The motivation of this research was to efficiently use the GPU to perform HAC clustering. At large, this is achieved by the partitioning of data in the GPU. The insufficiency of threads within a block to perform the traditional HAC computations and necessary synchronization are overcome by using multiple blocks and assigning partitions of data to each block in the PoP implementation. The PoP HAC implementation on GPU uses lesser memory than the traditional HAC requires on the GPU in *O(LogN)*. Computations are simultaneously executed across the threads of the multiple blocks which reduced time complexity. Moreover the effective numbers of distance computations are reduced by using the partitioning of data. This has tremendous impact on making the task of entire iterations of distance computations efficient. Thus it is possible to perform HAC for up to 100,000 observations with remarkably less time. The efficiency of using the PoP HAC on the GPU increases with the number of observations. We do not foresee any limitation in the GPU implementation of PoP HAC to handle even larger data sizes. The GPU is more efficient than the CPU for smaller number of PoP cells and larger number of observations within each cell. This again proves that GPU gets more efficient while being used for computations with large size of data and lesser number of data transfers between the CPU and GPU. Above all, the simplicity in programmability is achieved by using the global memory and avoiding the shared memory of the GPU. This has eliminated the performance issues that would arise due to 'synchronization' while using the shared memory

5.4. General Model for Parallelization of Data Mining Computations on GPU

We identify the following as the characteristic features of the GPU architecture that makes parallel processing of clustering algorithms possible within the single hardware:

(1) Identify and Compute data independent tasks using multiple 'Core-processors'.
(2) Solve discrete parts of the problem (tasks) via kernels.
(3) Break down each task into series of instructions to the 'Core-processors'.
(4) Execute instructions for each task simultaneously on the multiple 'Core-processors'.

These features are identified based on the extensive research done on using both OpenGL and CUDA and carefully exploring data mining algorithms such as *k*-means, FCM, HAC and PoP HAC. Though the algorithms are different, we find commonality across the methods used for implementing these computations on the GPU. Further, we propose a model for parallelization of data mining computations on the GPU. Figure 15 depicts the model we propose for translating a highly repetitive arithmetically intense (HRAI) data mining algorithm into GPU based parallel computational tasks. The steps in the Model for parallelization of data mining computations on the GPU can be briefly explained as follows:

(1) *Identify:* For any data mining algorithm of interest to be parallelized on the GPU, the first step is to identify the tasks that can be parallelized. This will require that the algorithm is broken down into simpler tasks and those tasks which are highly repetitive and arithmetically intense are identified. The execution time taken by the CPU implementation of these tasks should be calculated and the bottlenecks should be assessed. With this knowledge, the researcher could enlist the tasks that should be parallelized using the GPU. The performance gain would depend on the repetitiveness and arithmetic intensity of tasks identified for parallelization.
(2) *Parallelize:* After identifying the bottleneck tasks, the next step is to parallelize the codes. The tasks are broken down into executable multi-steps which are similar to inner loops. The inner loops which are data independent are parallelized by coding as 'kernels'. Kernels should be coded so as to access multiple memory locations simultaneously via threads

enabling simultaneous operations. There are arithmetic computations which exist as standard libraries of CUDA such as *cuBLAS*, *cuFFT* etc. which can be used for parallelizing tasks.

(3) *Translate:* After completely parallelizing each task, the researcher should integrate it with the main control program which can be based on either C language or C++. After integration of each task, the researcher tests the code and ensures that the tasks are executed faster than CPU and the speed gain targets are achieved. There may be parts of code which requires rework, fine-tuning or further optimization. In this step, the time taken for data transfers from the CPU to the GPU and the results transfer from the GPU memory to the CPU should be monitored. Any bottlenecks foreseen should be addressed by using suitable memory selection and data transfer schemes. The researcher may also optimize the codes by experimenting on the use of different functions, libraries or customized codes. Considering the above, the best possible translation of the sequential operations into the GPU should be evaluated and selected for implementation.

(4) *Implement:* The last step in this iterative model is to test the parallelized and translated tasks for speed gains. The researcher should confirm if the parallelization of the initially identified bottlenecks have yielded significant reduction in computational time. This can be done by comparing with the sequential implementation or with a previous GPU implementation. For quick improvements in speed gains, implementation may be done partially, as and when the tasks get parallelized and translated. The iterative cycle as shown in Figure 15 can be continued until the desired speed gain is achieved.

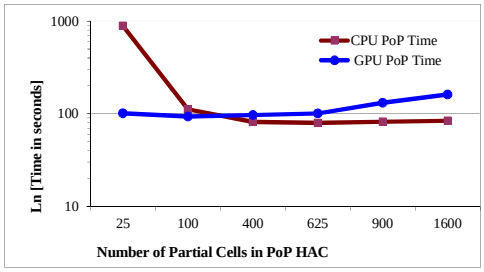


Figure 14. Effect of PoP GPU HAC versus CPU with various Partial Cells and 100K *2-D* data points on Computational time.

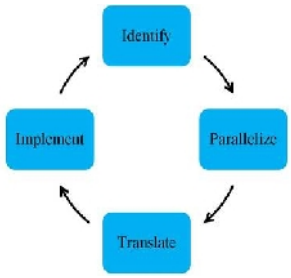


Figure 15. General Model for parallelization of Data Mining and General-purpose computations on GPU.

6. CONCLUSIONS AND FUTURE WORK

Often HAC methods are used for analysis in data mining which is large enough to justify cost efficient parallel computing and hence exhibit exploitability of the GPU hardware and CUDA programming capabilities on desktop computers. In this paper, we have presented a cost-effective, simple yet complete implementation of the traditional and the PoP HAC algorithms using CUDA for clustering genes on desk top computers. We have shown that the traditional HAC implementation on GPU has surpassed the CPU implementation by 30 to 65 times. For clustering 15000 genes with 31 feature expressions each, the CPU takes almost 3 hours to converge whereas our GPU based clustering machine processes it within six minutes. By efficient use of threads in the processors and the large global memory, a trade-off has been proposed between scalability speedups and ease of memory management and hence programming. The novelties in our implementations include the following:

(1) Speedup is obtained via accessing data in the global memory using a *1-D* half similarity array.
(2) The built in *'cublasIsamin'* function is used to identify the minimum distance pairs in virtually one pass.

(3) Completing all the HAC tasks within the GPU itself without necessity to move data back and forth the CPU during the iterations.
(4) Moreover, the need to maintain huge data, align threads to the local and shared memories are avoided; thus minimizing overheads due to 'synchronization of threads'. We propose to utilize the large global memory of the GPU compared to the small but faster shared memory so that performance issues due to overhead can be avoided.

Apart from the GPU hardware capabilities, the *speedup* depends on the number of vectors used and its dimensions for a given GPU configuration and the CUDA parameters such as block size and number of threads per blocks. The results on the HAC PoP experiments show that our implementation of PoP HAC on GPU is 442 times faster than the traditional HAC on CPU. It can also be noted that memory requirement for the PoP HAC on GPU is less by *O(LogN)* than that required by the traditional HAC on GPU. These results demonstrate the exploitation made on the GPU hardware using efficient data partitioning and the speedup gained on desktop computers. The use of GPU for the PoP algorithm has resulted in significant reduction in the time and memory complexity of the HAC algorithm. Our CUDA based implementations are fully portable to any latest Nvidia GPU which is CUDA compatible. The benefits is accelerating computations will be even higher with newer and improved hardware architectures from Nvidia. We have proposed a general model for parallelization of data mining computations on the GPU.

6.1. Research Issues and Limitations addressed

The insufficiency of threads within a single block to perform the traditional HAC computations and necessary synchronization are overcome by using multiple blocks and assigning partitions of data to each block in the PoP implementation. The PoP HAC implementation on GPU uses up to about 400 times lesser memory than that the traditional HAC requires on the GPU. Computations are simultaneously executed across the threads of multiple blocks which reduces time complexity. Thus it is possible to perform HAC for more than 50,000 observations with remarkably less time. As shown in Figure 12, the efficiency of using the PoP HAC on the GPU increases with the number of observations. This again proves that GPU gets more efficient while being used for computations with large size of data. Moreover the effective numbers of distance computations are reduced by using the partitioning of data. This has tremendous impact on making the task of entire iterations of distance computations efficient. Above all, the simplicity in programmability is achieved by using the global memory and avoiding the shared memory of the GPU. Thus the performance issues that would arise due to 'synchronization' while using the shared memory.

6.2. Performance Bottlenecks Identified while using CUDA for Computations

Performance optimization while using the GPU with CUDA for data mining computations often becomes the burden of the programmer. It is vital to note and understand the effect due to interactions between the instructions used and the memories accessed. Computationally less intense tasks should not be run on GPU as it will cause overhead and the performance is not worth the effort to parallelize. Operations involving large data sizes will benefit from utilizing maximum number of processors within the GPU. Memory access such as fetch and store to the texture memory or the global memory require more clock cycles and thus are slower than the shared memory. On the other hand the shared memory is too small in size to handle vast data matrices and might require multiple global synchronizations. Often it is better to re-compute the results to reduce memory traffic rather than storing and retrieving them from the memories. It is better to use GPU for algorithms which can be heterogeneous; thus it can be decided which part of the algorithm should be run on GPU or on the CPU depending on computational intensity.

6.3. Future Research Direction

Our future direction is to extend the *2-D* PoP to *d*-dimensional PoP. Such an implementation will increase the required number of PoP cells to the *O(d)*. The challenge arising could be overcome by using a regional algorithm to efficiently divide data with higher dimensionality. Tessellation based computing geometric structures such as Voronoi diagrams are currently studied to realize high dimensional PoP HAC on GPU. While using Voronoi for tessellation, several distinguished split points can be chosen and the space should be broken up into domains based on those points. The remaining points can be clustered based on which space it fall into and each cluster is then structured recursively. We propose to utilize the large global memory of the GPU compared to the small but faster shared memory so that performance and overhead issues can be avoided. We will also explore the constraint of maximum vector length that the GPU could support vs. speed gain performance.

Intel Corporation has recently released its new range of processors such as Core i3, i5 and i7. The maximum clock speed remains limited by the processors; for the Core i3 and i5 it is limited to 1100 MHz, while the Core i7 K can reach 1350 MHz [54].This sounds much faster than the CPU used in this research work, and it would give a substantial boost to the threaded performance which is yet to be studied. It is important to bear in mind that most features of such 3rd generation processors are unlikely to appeal to an average user, widely unexplained, but are instead targeted towards enterprise users. As applications of GPGPU, particularly in bioinformatics increases, GPUs will be designed and developed not just for their astonishing graphic performances but also for their extreme computational excellence, thus reducing cost of parallel data processing and enabling highly efficient computations on desk top computers.

REFERENCES

[1] M. Hopf, M. Luttenberger, and T. Ertl, Hierarchical splatting of scattered 4D data. IEEE Computer Graphics and Applications 2004, 64-72.

[2] J. Lu, B.M. Zhang, W. Huang, and E.S. Li, IHS Transform Algorithm of Remote Sensing Image Data Fusion Based on GPU. Computer Engineering 35 2009, 261-263.

[3] H. Takizawa, and H. Kobayashi, Multi-grain parallel processing of data-clustering on programmable graphics hardware. Parallel and Distributed Processing and Applications 2005, 16-27.

[4] C. DeCoro, and N. Tatarchuk, Real-time mesh simplification using the GPU, Symposium on Interactive 3D graphics and games, ACM, Seattle, 2007, pp. 161-166.

[5] J.D. Hall, N.A. Carr, and J.C. Hart, Cache and bandwidth aware matrix multiplication on the GPU. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware 2003,.

[6] 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, 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, IEEE, Daegu, Korea 2009, pp. 501-506.

[7] H. Nguyen, GPU Gems 3, Addison-Wesley, 2008.

[8] L. Gelisio, C.L. Azanza Ricardo, M. Leoni, and P. Scardi, Real-space calculation of powder diffraction patterns on graphics processing units. Journal of Applied Crystallography 43 2010, 647-653.

[9] N. Corportation, Cuda programming guide 2.0, December 12, 2008.

[10] W. Michael, Compilers and More:Programming GPUs Today, HPCwire, 2009.

[11] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell, A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum 26 2007, 80-113.

[12] A.K. Jain, and R.C. Dubes, Algorithms for clustering data, Prentice-Hall, Inc., 1988.

[13] J. Han, M. Kamber, and J. Pei, Data mining: concepts and techniques, Morgan Kaufmann Pub, 2011.

[14] J. Seo, and B. Shneiderman, Interactively exploring hierarchical clustering results [gene identification]. Computer 35 2002, 80-86.

[15] A. Sturn, J. Quackenbush, and Z. Trajanoski, Genesis: cluster analysis of microarray data. Bioinformatics 18 2002, 207-208.
[16] M.J.L. de Hoon, S. Imoto, J. Nolan, and S. Miyano, Open source clustering software. Bioinformatics 20 2004, 1453-1454.
[17] R. Suzuki, and H. Shimodaira, Pvclust: an R package for assessing the uncertainty in hierarchical clustering. Bioinformatics 22 2006, 1540-1542.
[18] P. Langfelder, B. Zhang, and S. Horvath, Defining clusters from a hierarchical cluster tree: the Dynamic Tree Cut package for R. Bioinformatics 24 2008, 719-720.
[19] T. Zhang, R. Ramakrishnan, and M. Livny, BIRCH: an efficient data clustering method for very large databases, ACM SIGMOD international conference on Management of data, ACM, Montreal, Quebec, Canada, 1996, pp. 103-114.
[20] S. Guha, R. Rastogi, and K. Shim, Cure: an efficient clustering algorithm for large databases. Information Systems 26 2001, 35-58.
[21] X. Li, Parallel algorithms for hierarchical clustering and cluster validity. Pattern Analysis and Machine Intelligence, IEEE Transactions on 12 1990, 1088-1092.
[22] X. Li, and Z. Fang, Parallel clustering algorithms. Parallel Computing 11 1989, 275-290.
[23] F. Cao, and A.Y. Zhou, Fast Clustering of Data Streams Using Graphics Processors [J]. Journal of Software 2 2007,.
[24] D. Chang, M. Kantardzic, and M. Ouyang, Hierarchical clustering with CUDA/GPU, Symposium on Computer Animation New Orleans, USA 2009, pp. 7-12.
[25] F. Cao, A. Tung, and A. Zhou, Scalable clustering using graphics processors. Advances in Web-Age Information Management 2006, 372-384.
[26] S.A. Shalom, M. Dash, and M. Tue, Efficient K-Means Clustering Using Accelerated Graphics Processors, International conference on Data Warehousing and Knowledge Discovery, Springer-Verlag, Turin, Italy, 2008, pp. 166-175.
[27] S.A.A. Shalom, M. Dash, and M. Tue, Graphics Hardware based Efficient and Scalable Fuzzy C-Means Clustering. in: J.F. Roddick, J. Li, P. Christen, and P.J. Kennedy, (Eds.), Seventh Australasian Data Mining Conference (AusDM) 2008, ACS, Glenelg, South Australia, 2008, pp. 179-186.
[28] H. Takizawa, and H. Kobayashi, Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing. The Journal of Supercomputing 36 2006, 219-234.
[29] G. Ziegler, A. Tevs, C. Theobalt, and H.P. Seidel, GPU point list generation through histogram pyramids, MPI Informatik, Bibliothek & Dokumentation, 2006.
[30] R. Wu, B. Zhang, and M. Hsu, Clustering billions of data points using GPUs, Combined Workshops on UnConventional high performance computing workshop plus memory access workshop ACM, Ischia, Italy 2009, pp. 1-6.
[31] W.B. Langdon, and A.P. Harrison, GP on SPMD parallel graphics hardware for mega bioinformatics data mining. Soft Computing-A Fusion of Foundations, Methodologies and Applications 12 2008, 1169-1183.
[32] L. Kaufman, P.J. Rousseeuw, and E. Corporation, Finding groups in data: an introduction to cluster analysis, Wiley Online Library, 1990.
[33] C.F. Olson, Parallel algorithms for hierarchical clustering. Parallel Computing 21 1995, 1313-1325.
[34] C.-H. Wu, S.-J. Horng, and H.-R. Tsai, Efficient Parallel Algorithms for Hierarchical Clustering on Arrays with Reconfigurable Optical Buses. Journal of Parallel and Distributed Computing 60 2000, 1137-1153.
[35] M. Dash, H. Liu, P. Scheuermann, and K.L. Tan, Fast hierarchical clustering and its validation. Data & Knowledge Engineering 44 2003, 109-138.
[36] M. Dash, and H. Liu, Efficient hierarchical clustering algorithms using partially overlapping partitions. Advances in Knowledge Discovery and Data Mining 2001, 495-506.
[37] J.C.H. Jesse D. Hall, GPU Acceleration of Iterative Clustering, ACM Workshop on General Purpose Computing on Graphics Processors, and SIGGRAPH 2004 poster, 2004.
[38] J. Wilson, M. Dai, E. Jakupovic, S. Watson, and F. Meng, Supercomputing with toys: harnessing the power of NVIDIA 8800GTX and playstation 3 for bioinformatics problem, Computational Systems Bioinformatics Imperial College Pr, San Diego, 2007, pp. 387-390.
[39] D. Chang, N.A. Jones, D. Li, M. Ouyang, and R.K. Ragade, Compute pairwise euclidean distances of data points with GPUs, IASTED International Symposium on Computational Biology and Bioinformatics, CBB 2008, November 16, 2008 - November 18, 2008, Acta Press, Orlando, FL, United states, 2008, pp. 278-283.

[40] Q. Zhang, and Y. Zhang, Hierarchical clustering of gene expression profiles with graphics hardware acceleration. Pattern recognition letters 27 2006, 676-681.
[41] S.A.A. Shalom, M. Dash, M. Tue, and N. Wilson, Hierarchical Agglomerative Clustering Using Graphics Processor with Compute Unified Device Architecture, International Conference on Signal Processing Systems, IEEE, Singapore, 2009, pp. 556-561.
[42] S.A.A. Shalom, and M. Dash, Efficient Hierarchical Agglomerative Clustering Algorithms on GPU Using Data Partitioning, International Conference on Parallel and Distributed Computing, Applications and Technologies, IEEE Computer Society, Gwangju, South Korea, 2011.
[43] A.J. Rueda, and L. Ortega, Geometric algorithms on CUDA, International Conference on Computer Graphics Theory and Applications, Citeseer, Funchal, Madeira - Portugal 2008, pp. 107-112.
[44] T.R. Halfhill, Parallel Processing with CUDA Nvidia's High-Performance Computing Platform Uses Massive Multithreading. Microprocessor Report 22 2008, 1-8.
[45] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming ACM, Salt Lake City, Utah 2008, pp. 73-82.
[46] N.K. Govindaraju, N. Raghuvanshi, and D. Manocha, Fast and approximate stream mining of quantiles and frequencies using graphics processors, ACM SIGMOD international conference on Management of data, ACM, Baltimore, Maryland, 2005, pp. 611-622.
[47] P. Trancoso, and M. Charalambous, Exploring graphics processor performance for general purpose applications, Euromicro Conference on Digital System Design, Porto, Portugal, 2005, pp. 155-162.
[48] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing 68 2008, 1370-1380.
[49] N. Corporation, Data Mining, Analytics, and Databases, 2010.
[50] R. Farber, Parallel Programming Tutorial Series - Part 8 - CUDA, UBM TechWeb Reader Services, 2008.
[51] H.C. Causton, J. Quackenbush, and A. Brazma, Microarray gene expression data analysis: a beginner's guide, Wiley-Blackwell, Malden, 2003.
[52] S. Shalom, M. Dash, and M. Tue, An Approach for Fast Hierarchical Agglomerative Clustering Using Graphics Processors with CUDA. Advances in Knowledge Discovery and Data Mining 2010, 35-42.
[53] N. Corportation, GeForce Specifications / Performance, December 10, 2010.
[54] M. S. Smith, Core i3 vs i5 vs i7: A Summary of Intel's Processors, Bright Hub Inc., 2012.