Parallel Computations for Hierarchical Agglomerative Clustering using CUDA

Fast and Scalable Computations on Graphics Processors

S.A. Arul Shalom¹, Manoranjan Dash²

¹ School of Computer Engineering, Nanyang Technological University
50 Nanyang Avenue, Singapore
sall0001@ntu.edu.sg; ²asmdash@ntu.edu.sg

Abstract: Graphics Processing Units (GPU) in today’s desktops can well be thought of as a high performance parallel processor. Traditionally, parallel computing is the usage of multiple computing resources to execute computational problems simultaneously. Such computations are possible using multi-core CPUs or computers with multiple CPUs or by using a network of computers in parallel. Today’s GPUs are capable of simultaneously using multiple internal computing resources such as ‘core-processors’ or ‘multi-processors’ to compute within a fraction of the time a CPU would need. We explore the parallel architecture of GPU for cost-effective desktop parallel computing of a core data mining problem such as clustering, which could then be applied to parallelize other data mining computations. The launch of NVIDIA’s Compute Unified Device Architecture (CUDA) technology has been a catalyst to the phenomenal growth of the application of GPUs to parallelize various scientific and data mining related computations. With CUDA the skills and techniques needed in invoking the internal parallel processors of a GPU is viable to scientific researchers who might not be expert graphics programmers. We embark on the application of CUDA based programming to parallelize the traditional Hierarchical Agglomerative Clustering (HAC) algorithm and demonstrate speed gains over the CPU. Speed gains from 15 times up to about 90 times have been realized for various clustering conditions. The effects of CUDA blocks and challenges involved in invoking graphical hardware for such data mining algorithms are discussed. It is interesting to note that a block size of 8 is optimal for GPU with 128 internal processors. We further discuss the research issues that arise with parallelizing HAC on GPU with CUDA and propose the use of GPU as an efficient desktop processor. Results show that the future of extensively utilizing desktop computers for parallel computing based on GPUs is promising.

Keywords: Hierarchical clustering; High performance Computing; Parallel Computations using Graphics hardware; Single linkage; GPGPU

I. INTRODUCTION

Traditionally, in computations of data mining algorithms, the program instructions are executed in a sequence using a single processor. Some problems however, have tasks which are time independent and hence be broken down into multiple tasks that could be executed concurrently. A set of computations which are performed (as part of a system) while in a stationary state, where those computations do not instantly impact or affect the overall result can be said to be not explicitly dependent on time or simply time independent computations. Such concurrent execution of tasks is possible if and only if they are data independent. We will discuss the application of GPU for parallel computing of clustering algorithms on desktop computers. We will compare the CPU based sequential computing architectures to the GPU based parallel computing, thus proposing economic yet efficient parallel computing for clustering algorithms.

The GPU is designed to perform computations for image rendering with extreme speed in its traditional use. The raw computational power of GPU can be expressed and compared with that of the CPU in terms of ‘Giga floating-point operations per second’ and ‘Memory Bandwidth’. Fig. 1 shows the growth trend of computational power of the GPU and the CPU in terms of peak Giga Flops (GFlops). For instance, the NVIDIA 8800 GTS GPU has a peak performance of 624 GFlops and Memory Bandwidth of 62 Giga Bytes per second (GBps) whereas a 3.0GHz Dual Core Intel CPU has a peak floating point rate of 12 GFlops and Memory Bandwidth of 12.8 GBp. Such form of raw computing power of the Graphics hardware is available to be utilized for non-image processing related computations [6-8].

Traditionally, in computations of data mining algorithms, the program instructions are executed in a sequence using a single processor. Some problems however, have tasks which are time independent and hence be broken down into multiple tasks that could be executed concurrently. A set of computations which are performed (as part of a system) while in a stationary state, where those computations do not instantly impact or affect the overall result can be said to be not explicitly dependent on time or simply time independent computations. Such concurrent execution of tasks is possible if and only if they are data independent. We will discuss the application of GPU for parallel computing of clustering algorithms on desktop computers. We will compare the CPU based sequential computing architectures to the GPU based parallel computing, thus proposing economic yet efficient parallel computing for clustering algorithms.

The GPU is designed to perform computations for image rendering with extreme speed in its traditional use. The raw computational power of GPU can be expressed and compared with that of the CPU in terms of ‘Giga floating-point operations per second’ and ‘Memory Bandwidth’. Fig. 1 shows the growth trend of computational power of the GPU and the CPU in terms of peak Giga Flops (GFlops). For instance, the NVIDIA 8800 GTS GPU has a peak performance of 624 GFlops and Memory Bandwidth of 62 Giga Bytes per second (GBps) whereas a 3.0GHz Dual Core Intel CPU has a peak floating point rate of 12 GFlops and Memory Bandwidth of 12.8 GBp. Such form of raw computing power of the Graphics hardware is available to be utilized for non-image processing related computations [6-8].
A. Recent Advancements and Challenges in GPGPU

The programming model of the GPU is harsh, constrained and is heavily linked to computer graphics. It is vital that computational algorithms have to be carefully designed and ported, to effectively suit the graphics environment. It continues to be a challenge to scientists and researchers to harness the power of GPU for applications based on general-purpose computations. It is not straightforward to port the CPU codes to the GPU and not a simple task to realize. Researchers have to learn graphics dedicated programming platforms such as Open Graphics Language (OpenGL) or DirectX, and convert the computational problem into a graphics problem [4, 9-11]. This requires tedious efforts and is time consuming. Lately, various graphics Application Programming Interfaces (API) and development platforms made available for GPGPU developers are Open Computing Language (OpenCL) by the Khronos Group, RapidMind of Intel Corporation, ATI Stream Software Development Kit (SDK) of Advanced Micro Devices (AMD), Open Hybrid Multicore Parallel Programming (OpenHMPP) and CUDA of Nvidia Corporation, which are expected to lighten the tasks of researchers interested in GPGPU. OpenHMPP is a programming standard for heterogeneous computing, designed to handle hardware accelerators without the complexity associated with GPU programming. OpenCL is the vendor independent, heterogeneous platform parallel programming standard of modern processors found in desk top computers, handheld and embedded devices. RapidMind is another Multicore development platform which is relatively new with very less public usage. ATI Stream SDK is the stream processing architecture originally developed by ATI technologies, now being used to program the AMD stream processors. OpenHMPP is a standard for heterogeneous computing, which is very much less used too in GPU programming. Currently CUDA is the prevalent and proven platform for GPGPU based acceleration using Nvidia’s GPUs. The standard functions and libraries in CUDA allow the developers to directly access the GPU’s hardware components such as processors, memories and registers [12]. OpenCL is expected to greatly improve computational speed for a wide range of applications in heterogeneous GPGPU. OpenCL is largely based on CUDA and thus there will be relatively less efforts required in making the switch from Nvidia’s CUDA to OpenCL for GPU from Nvidia and other manufacturers [13]. Applications of other multi-core programming platforms using libraries such as Pthreads, OpenMP, etc, including implementations with CUDA to parallelize computationally intensive algorithms are demonstrated in [38, 40, 41]. CUDA is advantageous over the other APIs as it provides tremendous computational power to the programmer by fully utilizing the hardware resources of the Nvidia GPU.

B. Motivations

Recent research in computer architecture shows a trend towards the fields of streaming computations, multi-core architectures, heterogeneous architecture, distributed and grid computing, polyphonic and cloud computing. The architecture of GPU has a lot in common with these hot research topics. It is possible to offload more arithmetically intense computations from the CPU to the GPU for processing massive data sets even in desktops for applications such as scientific computing, physical simulations, image processing, computer vision, data mining etc [14, 15].

Clustering has become a common technique in data analysis, which has wide spread applications in many fields such as machine learning, data mining, text mining, pattern recognition, image processing and bioinformatics. The five major categories of clustering are k-partitioning algorithms, hierarchical algorithms, density-based, grid-based and mode-based. In the most popular k-partitioning method, the clusters are assumed to be spherical and the required number of partitions is pre-determined which may not be optimal. GPU implementations of such algorithms are available [10, 16]. In hierarchical clustering a complete hierarchical decomposition of the objects is created either by agglomeration or division and the required number of clusters can be obtained [17]. Although hierarchical agglomerative clustering (HAC) has been widely applied in various fields, it is predominantly used for document clustering and retrieval, cluster identification from micro array gene expressions and in image compression, searching and clustering where computationally intense and time consuming high throughput data processing is involved. 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 [18]. The basic pseudo code for HAC algorithm is given in Algorithm 1 along with the memory complexity analysis for the computationally intense steps. The time complexity of the HAC algorithm is at least $O(N^2)$ and the overall complexity of the algorithm is $O(N^2*\log N)$ where $N$ is the number of objects to be clustered. This makes the HAC algorithm a very good candidate for parallelization and few efficient but sequential attempts to speed up the computations have been made on the GPU [19-21].

**Algorithm 1 Basic pseudo code for Agglomerative Clustering with complexity**

<table>
<thead>
<tr>
<th>Hierarchical Agglomerative (bottom up monotonic merging) algorithms create a Dendrogram (Tree) which can be pruned to form Clusters</th>
<th>Memory Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Set of data point to be clustered</td>
<td><strong>Output:</strong> A dendrogram</td>
</tr>
<tr>
<td>1. Compute $H$: Half-distance matrix between every pair of data points</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>2. Initiate: Creation of one cluster for each individual data point</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>3. DoWhile: Number of clusters, $N &gt; 1$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>For, $i = 0$ to $N-1$</td>
<td></td>
</tr>
<tr>
<td>Find the closest pair of clusters (involves computing the distance between every pair of clusters)</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>4. Merge the data points in the closest pair into a single cluster</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>5. Update the half-distance matrix $H$ (delete the rows and columns in $H$ associated with these two nearest clusters and add to $H$ a new row and column corresponding to the new cluster)</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
C. Literature Review

It is important to discuss relevant and related work that are done in the field of Clustering using GPU in order to effectively present the novelty of our research work. In this sub section, a brief review on various existing clustering and similar data mining implementation approaches on GPU, the platforms used and the performance realized is provided.

In [39] GPU computing is applied to implement genetic and evolutionary algorithms for density based clustering which are time consuming computations otherwise. In this research study, the design, implementation, and evaluation of genetic algorithms are realized with remarkable performance on an Nvidia GPU using CUDA. For image segmentation, the computationally demanding clustering algorithm such as k-means is implemented on GPU using CUDA [42]. In this paper, the parallelization of the k-means clustering algorithm is realized in a hybrid manner via distributing the computational load between the CPU and the GPU. Experiments have demonstrated the execution of considerably faster segmentation while comparing to a classical CPU-based approach. In [46], a fully unsupervised colour image segmentation algorithm based on k-means is proposed for complete GPU implementation, including the calculation of region features. The GPU results are compared to a popular CPU based and has faster computational time. In [44], neural network based Self Organizing Maps (SOM) are used for data clustering. SOM involves expensive training steps, while used for high dimensional applications like image clustering. In this paper, the use of Nvidia GPUs with CUDA is demonstrated to decrease computational cost of SOM. A three step implementation is proposed in order to reduce the computational complexity of the algorithm under SIMD paradigm by making good use of the GPU’s resources. Speed gains up to 44 times is achieved when compared to a C based CPU implementation. In [45] the GPU implementation of the Gustafson-Kessel (GK) clustering algorithm is discussed. Although the GK clustering algorithm brings significant benefits over the other algorithms, it becomes computationally inefficient when applied to high-dimensional data. In this GPU based parallel implementation of the GK algorithm with CUDA which uses optimized matrix multiplications with fast access to shared memory, the speed gains to a maximum of 240 times over the CPU implementation is achieved at certain clustering conditions.

On the other hand, GPU clusters have also been successfully used for high performance cloud computing on public cloud computing infrastructure, Amazon EC2 Cluster GPU Instances (CGI), utilizing the Nvidia Tesla GPUs and a 10 Gigabit Ethernet network. Results show that using GPU clusters is a viable option for cloud computing despite the significant impact on network overhead of virtualized computing resources [43].

It is also vital to explore the use of CUDA to compute algorithms with gene expression data. One such application is discussed in [47], where Nvidia GPU with CUDA is used to infer Gene Regulatory Networks (GRN) gene interactions from gene expression data, such as microarray data. In this work the authors develop a low cost parallel solution for exhaustive search with a viable cost-benefit. Speed gains up to 60 times were achieved when assuming that each target gene has two predictors. CUDA is also used to analyse gene expressions using Multi-dimensional scaling (MDS) which has high computational complexity [48]. Authors of this research work have implemented a highly-efficient algorithm, called as the CUDA-based Fast Multidimensional Scaling (CFMDS), which produces an approximate solution of the classical MDS. This GPU implementation of MDS is more than 100 times faster while tested using microarrays and compared with MDS algorithms implemented using C# and MATLAB.

In [28], we depict GPU as an effective desktop multi-processor machine which handles various computationally-intensive data-independent tasks in data mining, especially on clustering algorithms in parallel, while sequentially executing sections of the problem-program. We summarize the exploration of the parallel architecture of GPU for computing various clustering algorithms such as k-means, Fuzzy C Means (FCM) and HAC to compare speedup performances based on types of GPU hardware, programming architectures and various clustering parameters such as data size, cluster size and dimensionality. The existing implementations of HAC algorithms using GPUs are discussed in the next sub section, as it is our intended algorithm for GPU based CUDA implementation.

D. CUDA for Hierarchical Clustering and our Intension

Hierarchical clustering algorithms have been implemented on the GPU using OpenGL and CUDA in the past [4-6]. The purpose of this research work is not to reduce the memory complexity of algorithm but to look into simpler ways of using CUDA for parallelizing HAC computations, understand the effects of CUDA programming and clustering parameters on scalability and speedup performances. The first CUDA based work [6] deals with the implementation of the ‘pair-wise distance’ computation, which is one of the fundamental operations in HAC. The Nvidia 8800GTX GPU is employed and it uses CUDA programs to speed up the computations. Gene expression data is used and the standard HAC is implemented using half matrix for pair-wise distance calculation. Experimentations are done to evaluate the use of threads; ‘one thread for one row of the output matrix and one thread for entry in out’. Moreover, the GPU shared memory 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 implementations. It is important to note that this speed up is achieved only for the pair-wise distance computations and not for the complete HAC algorithm.

In the second research work [4], CUDA based HAC algorithm on the Nvidia 8800GTX GPU is compared with the performance of commercial bioinformatics clustering applications running on the CPU. Based on the cluster parameters and setup used, speedup of about 10 to 14 times is achieved. The effectiveness of using CUDA on GPU to
cluster high dimensional vectors is also shown. This is accomplished at an expense of organized memory ‘micro-management’ within the GPU.

We have implemented the complete linkage method based HAC method in [22] and demonstrated speed gains of about 30 to 65 times than the CPU. As a novelty, we found it easier to conceive a one-dimensional grid to accomplish the highly intensive computation of pair-wise comparison for identifying the clusters with minimum distance than two-dimensional arrays. The complete linkage clustering avoids a drawback of the single linkage method, where clusters formed may be forced together (chaining-phenomenon) due to single data points being close to each other, even though most data points in each cluster might be at fairly large distances to each other. Given the fact that single linkage method groups cluster pairs with minimum distance, in this research we intend to explore the possibility achieving higher speed gains in spite of the impact of the above ‘chaining-phenomenon’. The idea is to boost the performance of GPU implementation using the ‘cublastsamin()’, which is a built in function of CUDA.

We exploit the parallel processing architecture, the large global memory space, and the programmability of the GPU to efficiently implement the traditional HAC algorithm completely. We use a relatively cheaper graphics card on a desktop computer. Nvidia 8800GTS GPU which has lower specification than the GTX version. The cost of 8800GTX ranges from $500 to $700, whereas the 8800GTS ranges from $100 to $400. We use the latest graphics programming API; Nvidia’s CUDA 2.0 to parallelize the computations of HAC in the GPU. We utilize the global memory more than the shared memory in the GPU, which is relatively simple to be programmed, though the access is slower. We present the novelties of our research and recommend certain criteria for the selection of programming parameters of CUDA. We explore the relations between the CUDA parameters such as block size, number of blocks versus size of data and dimensions, intending to find the optima where a given GPU configuration would peak its performance. We have implemented the single linkage method of HAC and tested using micro array gene expression data of yeast. We achieved speed gains up to 90 times faster than the CPU, for various gene expression sizes and CUDA parameters.

The rest of the paper is organized as follows: chapter 2 briefly explains the traditional HAC algorithm and its typical implementation in the CPU, chapter 3 we discuss how CUDA is used to parallelize the computations in HAC. In chapter 4, we present the experiments and discussions on the results, chapter we close with the conclusions and future directions for related research work.

II. TRADITIONAL HIERARCHICAL AGGLOMERATIVE CLUSTERING ALGORITHM

HAC is a common and important algorithm used in data mining in the domains of micro array analysis, genome clustering, image processing and web page clustering. Hierarchical clustering seeks to build up a hierarchy of clusters from which a desired number of clusters could be identified. Hierarchical clustering does not require us to pre-specify the number of clusters. This advantage of hierarchical clustering comes at the cost of lower efficiency due high time and memory complexity.

A. Understanding the HAC Algorithm

The objective of HAC is to generate high level multiple partitions in a given dataset. The groups of partitions of data vectors will denote the sets of clusters. In this bottom-up approach, each vector is treated as a singleton cluster to start with and then they are successively merged into pairs of clusters (aggglomerate) until all vectors have merged into one single large cluster. The agglomeration of clusters results in a tree-like structure called the dendrogram [22]. A measure of similarity between the clusters is required to decide which clusters should be merged. We use the Euclidean distance as the metric to determine the similarities between pair of clusters. [6, 17] where the vector norms $a_i$ and $b_j$ can be calculated using Equation (1), where $n$ is the number of cluster vectors to be merged. Selection of a pair of cluster to merge depends on the linkage criteria. There are several methods of selecting the pair of clusters to be merged; resulting in different algorithms [18]. The pair of cluster selected based on any of the linkage criteria are merged and a cluster vector is updated. In single linkage agglomerative clustering (single linkage method), the distance between two clusters to be merged is the distance between the nearest pair of data points, where each data point belongs to a different cluster. Let $A$ and $B$ be two clusters. The $distance(A, B)$ is the minimum distance $a, b$ for all pairs of data objects $a, b$ such that $a$ belongs to ‘A’ and $b$ belongs to ‘B’. This criteria used by the single linkage clustering is given in Equation (2).

$$\|a - b\|_2 = \sqrt{\sum_{i=1}^{n} (a_i - b_j)^2}$$ (1)

$$Minimum\{dist(a,b): a \in A, b \in B\}$$ (2)

The combination 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 partitions of the data set by cutting the dendrogram at the required level of combination similarity or number of clusters expected. Fig. 2 shows such a dendrogram illustrating the HAC algorithm, where the singleton clusters $p, q, r, s$ and $t$ are progressively merged into one single large cluster.

B. HAC Implementation on the CPU

We implemented the half similarity matrix based single linkage HAC in the CPU and used its computational time is measured as reference to compare with the computational time taken by the GPU. Assuming that the vectors of size $n$ and dimension $d$ is stored in an array, the code listed in Algorithm II will compute the half-matrix of the pairwise Euclidean distances and store it in array $dist$.

Fig. 2 A Dendrogram of Hierarchical Agglomerative Clustering
### III. CUDA FOR PARALLELIZING HAC COMPUTATIONS ON GPU

In the legacy approach, algorithms involved are broken down into small chunks of computations or kernels called the shaders in Graphics Library and Shading Language (GLSL) or C for Graphics (Cg). In the unified design as in CUDA, it is possible to execute multiple shaders by synchronizing them to the various memories of the GPU. This unified design provides better workload balance between the stream processors in the GPU, thus avoiding delays in completion of shading. In this section we will discuss the general structure of CUDA for effective use in a preferably simple method to parallelize the time and data independent computations in HAC algorithm.

#### A. Programming Structure of CUDA for Parallel Computations

The software stack of CUDA runs on the GPU hardware as an API to the standard C language, providing Single Instruction Multiple Data (SIMD) capabilities. 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. GPU has many more Arithmetic Logic Units (ALU) than the CPU which provides its intensive computational power, while lacking the complex system of logical control. This concentration of ALUs on a single GPU chip promotes the distribution of massive data and instructions over those units simultaneously, enabling parallel computation.

Chapter 2 of [14] explains the programming structure, memory management and the invocation of kernel functions in detail. Fig. 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 available on the GPU. The lowest level of computation is the thread which is analogous to shaders in OpenGL [23, 24]. Each thread has access to local data memory and to memories at different hierarchies. Instructions are passed into the threads to perform computations. 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 [25, 26]. Threads in a block are identified by a unique Thread ID and blocks in a grid by a Block ID. The

#### 3.1. Algorithm II Code to Compute the Upper Half Matrix of Pair-wise Euclidean Distances on GPU

```c
double** dist;
double start, end;
start=clock();
dist=new double*[n-1];
for (int i=0; i<n-1; i++)
    dist[i]=new double[n-1-i];
for (int i=0; i<n-1; i++)
{
    for (int j=i+1; j<n; j++)
    {
        int v=j-i-1; dist[i][v]=0;
        for (int k=0; k<d; k++)
        {
            double r=a[i][k]-a[j][k]; dist[i][v]+=r*r;
        }
    }
}
end=clock();
double time=(double)(end-start)/CLOCKS_PER_SEC;
```

#### B. Parallelization of HAC on the GPU

The computational steps that can be made parallel are implemented on the Graphics processor. Table III summarizes the functions used in the implementation of such computations in the HAC single linkage method using CUDA on GPU. This implementation architecture is common for both the complete linkage and centroids methods too. Understanding the CUDA architecture is vital in effectively implementing computational algorithms on the GPU and is well explained in [2, 26, 27]. It can be said that the CUDA based GPU for computing architecture is heterogeneous. The program consists of sequential codes and parallel kernel codes as shown in Fig. 4. The serial codes are executed on the CPU host while the kernel codes are executed on the device which is the GPU. The data and time independent tasks that could be parallelized are identified to be computed on the GPU in Table III. The parallel program would then contain multiple data and time independent tasks being simultaneously solved. The ‘divide and conquer’ technique is used on a given parallel task which is divided into multiple sub-tasks for achieving parallelism [28]. These sub-tasks could be then made available to multiple processors in the GPU in an efficient manner for computations.

---

**Fig. 3 CUDA Device Memory Model of GPU. [Courtesy: Nvidia]**
C. Parallelization Half Similarity Computations on the GPU

Computation of distances between cluster pairs is the most time and memory consuming task in HAC. In this section we intend to discuss the construction of the similarity matrix in the global memory of the GPU in detail. The half similarity matrix $S_{ij}$ is obtained by splitting the task of computing Euclidean distances between several CUDA 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 $S_{sub}$.

The number of threads per block or the equivalent 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.

The computational performance of the GPU is considered optimal when the number of threads per block is considered as 128 processors, utilizing all processors. While accessing the 2D global memory, if the blockSize is 4 by 4 ($4^2=16$), then, $128/16 = 8$ blocks per grid is considered optimal. When the dimension blockSize is 4, and if 8 blocks are used per grid, then 16 $(128/8=16)$ threads operate on the data per grid. While computing distances between vectors in the HAC algorithm, if there are 4 dimensions per vector, with the above block structure in one pass the distances between the first vector and 4 other vectors can be computed $(4*4=16)$. Thus each grid with 16 threads behaves like an ‘operator’ on the data in parallel while the kernel is executed. The following steps illustrate the parallel computation of similarity matrix using the GPU.

1. Read vectors from the Dev_data array on GPU
2. For each row of vectors in the Dev_data array, calculate the index ‘k’ using expression in Equation (3)
3. Store the distances in the Dev_dist array on GPU
4. Update Dev_data array and Dev_dist array
5. Repeat steps 1 to 4 till all the vectors are exhausted.

Using 1D array instead of the traditional 2D array in the global memory showed significant improvement in the computational performance. The 1D array as shown in Fig. 5 would allow efficient reading and writing to continuous memory addresses using the index ‘k’. The expression shown in Equation (3) is used to get the index k of the similarity array as it is related to the index (i, j) of the corresponding vectors. The kernel that runs on the GPU to compute the similarity matrix for blockSize of 4 is shown in Algorithm IV. 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 and the distances are computed in parallel.

D. Characteristics of Data Mining Algorithms for Parallelism

Clustering algorithms are one of the most important and prevailing data mining methods in various fields such as customer relationship management, financial engineering, bio-medical sciences etc. The observations made are based on evaluations done on the possible computations in clustering algorithms such as k-means, FCM and HAC in [29-32].

<table>
<thead>
<tr>
<th>Computational Steps in HAC</th>
<th>GPU CUDA functions</th>
<th>Kernel runs in GPU?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transfer input vectors from CPU to GPU</td>
<td>CudaMemcpyHostToDevice;</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>Populate initial similarity half matrix</td>
<td>CalculateDistance(); k index locates position in the 1D array</td>
<td>Yes</td>
</tr>
<tr>
<td>Identify minimum distance vectors</td>
<td>cublasamin(); built in function of CUDA</td>
<td>Yes</td>
</tr>
<tr>
<td>Merge Clusters</td>
<td>updateArray0();</td>
<td>Yes</td>
</tr>
<tr>
<td>Update the similarity half matrix</td>
<td>UpdateArray1(); updateArray2();</td>
<td>Yes</td>
</tr>
<tr>
<td>Update the Minimum Distance Array</td>
<td>UpdateArray3();</td>
<td>Yes</td>
</tr>
<tr>
<td>Transfer Cluster data from GPU back to CPU</td>
<td>CudaMemcpyDeviceToHost();</td>
<td>Not Applicable</td>
</tr>
</tbody>
</table>

For a GPU with 128 processors, utilizing all processors with maximum number of threads will maximize the efficiency. While accessing the 2D global memory, if the blockSize is 4 by 4 ($4^2=16$), then, $128/16 = 8$ blocks per grid is considered optimal. When the dimension blockSize is 4, and if 8 blocks are used per grid, then 16 $(128/8=16)$ threads operate on the data per grid. While computing distances between vectors in the HAC algorithm, if there are 4 dimensions per vector, with the above block structure in one pass the distances between the first vector and 4 other vectors can be computed $(4*4=16)$. Thus each grid with 16 threads behaves like an ‘operator’ on the data in parallel while the kernel is executed. The following steps illustrate the parallel computation of similarity matrix using the GPU.
function predictions, thus to find new potential tumour subclasses [35]. Gene expressions of Human Mammary Epithelial Cells (HMEC) with 60000 gene expressions and 31 features each were used to evaluate the performance of the parallel implementation of HAC compared to the CPU implementation. This microarray dataset has been obtained from experiments conducted on HMEC. Though gene expressions may be in the order of thousands or hundreds of thousands, usually only a few thousand of the gene samples or a group of genes are chosen for clustering.

B. Experimental Setup, Results Analysis and Discussions

The CUDA based parallelization of the single linkage HAC algorithm is executed and tested on a Nvidia GeForce 8800 GTS GPU with memory of size 512MB. The corresponding CPU implementation is run on a desktop computer with Pentium Dual Core CPU, 1.8 GHz., 1.0GB RAM on MS Windows Professional operating system. The following studies are planned, conducted and the results are analyzed.

1) Effect of Number of Genes and CUDA block sizes on Speed Gain:

The CUDA implementation uses block of sizes of 4, 8 and 16. For this experiment with single linkage HAC algorithm, the number of genes with 31 dimensions each is increased and the computational time taken for clustering the gene expressions is measured. The experiment is repeated for block sizes of 4, 8 and 16 per grid. Fig. 6 shows the effect of increase in number of genes on computational speed gains. It can be noticed that maximum speed gain over the CPU is about 46 times for 10000 genes while using a block size of 8 per grid. The speed gain drops as the number of genes increase to about 32 times for 15000 genes. Further, it can be deduced that the speedup performance while using 8 blocks is significantly better than using 4 or 16 blocks per grid though the difference is less evident for higher number of genes. Each CUDA processing block is run on one multiprocessor (MP). In an 8800 GTS GPU there are 16 such MPs with 8 internal processors each; that makes the total number of internal processors to 128. Thus while using a block size of 8, the use of 8 * 8 = 64 threads is referred to. Each thread is associated with an internal processor during runtime. 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. This also explains why the speed gains with block size of 8 are higher due to optimal use of GPU resources such as the internal processors.

2) Effect of Gene size on Speed Gain:

The CUDA implementation uses block size of 8 per grid as recommended from the previous study. Fig. 7 contrasts the performance with 6 and 31 dimensions with increasing gene dimensions. Speed gains from about 54 to 90 times have been achieved for low dimensional genes and about 30 to 35 times for the 31 dimensional gene data which is remarkable. The drop in speed gain with increase in number of dimensions or increase in number of genes can be attributed to the grid configuration used in this
experimentation and the available GPU memory. The use of 1 grid in the GPU to invoke the multi-processors defined as per the block size via the threads, limits the number of processors that can used simultaneously for computing. Multiple grids and blocks should be used, thus maximizing the use of the internal processors to minimize the impact of higher dimensional data computing on performance.

![Fig. 6 HAC Single linkage on GPU with CUDA: Speed Gains vs. Number of Genes with 31 dimensions each and CUDA Block sizes](image)

![Fig. 7 HAC Single linkage Speed Gains: Number of Genes vs. 6 and 31 dimensions of Genes](image)

### C. Research Issues with Clustering Algorithms on 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. The inherent parallelism in the graphics hardware is invoked by CUDA features to realize the computations in parallel on the GPU. In this section we discuss the research issues in doing so.

1) **GPU resource analysis for Half Distance Matrix Computation:**

Fig. 3 shows the device memory model of CUDA and the hardware resources which are used in the parallelization of HAC computations on the GPU. Half distance matrix calculation has a time complexity of the $O(N^2)$ which will need the utilization of significant resources of the GPU. Fig. 8 shows the effect of increase in dimensions of genes on computational speed gains and the % of time taken to compute the half similarity matrices. It can be noticed that speed up to about 90 times is gained at low dimensions and it drops as the dimensions increase to about 13 to 15 times, while the corresponding % of time taken to compute the half distance matrix is about 6 for low dimensional genes and about 12 to 13 for higher dimensional gene clustering. The data transfer time is about 1% or less. This shows that about 85% of the time is used to perform computations such as minimum distance comparisons etc, but excluding the time for half distance matrix computations. Table V shows the computational times both on the GPU and CPU along with the speed gains and % calculated. The reduction in time complexity due to parallelization is explained as follows.

There is no direct control possible at block-level for the programmer and the block allocation to internal processors cannot be determined. When a grid is launched, CUDA automatically allocates blocks to the processors. There will be $n*n/(2*k)$ threads created per block, where $n$ is the number of observations and $k$ is the possible number of threads per block. In hierarchical clustering, $n*n/2$ is the size of the half-similarity matrix, which is also the number of threads needed to simultaneously operate the entire matrix. In this HAC parallelization only one block is used per grid. For block size 8, the number of threads invoked per block is 64. For the total number of threads generated to be invoked in the block, only $k*number of blocks$ will be executed simultaneously. Though the total number of blocks required is $n*n/2$, there will be queuing while only one block is used. Within a grid, a number of blocks used will be queued up with threads and allocated to processors in a MP. To be effective we need to maximize the use of blocks within the grid. When a grid is launched, the number of blocks processing is equal to 'number of MP used*8', the number of blocks is designated as twice as the number of MP because computations in some of the blocks may finish earlier than the others. When a computational queue is complete, the processor will be idle and that is a waste. To overcome this issue multiple blocks may be used thus managing and utilizing the hardware resources of the GPU more effectively [27].

2) **Analysis of Threads in CUDA for Data Parallelism:**

The number of threads invoked via a program is dependent on the algorithmic design. For example, for computing the vector distance of array $A$ and array $B$ of size $n$, there would be at least $m$ threads operating on a pair of element $(a_i, b_i)$, where $1 \leq i \leq m$. This design theoretically would provide maximum parallelization. In the distance computations, $m$ threads are arranged in a way to naturally group into blocks of similar size, satisfying the relation: $m = number of threads = (number of blocks) * (number of threads per blocks)^2$.

![Fig. 8 HAC Single linkage method: Speed Gains with 10000 Genes vs. Dimensions and % of Time taken to compute half similarity matrices](image)
TABLE V TYPICAL COMPUTATIONAL TIME IN SECONDS, SPLIT OF GPU TIMING AND SPEED GAINS IN GPU FOR 10000 GENE EXPRESSIONS WITH INCREASING DIMENSIONS FOR SINGLE LINKAGE HAC METHOD

<table>
<thead>
<tr>
<th>d</th>
<th>Data Transfer time in GPU</th>
<th>Computation time in GPU</th>
<th>Similarity half-matrix time in GPU</th>
<th>Total time on GPU</th>
<th>CPU time</th>
<th>Speed Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0.38</td>
<td>33.35</td>
<td>2.00</td>
<td>33.73</td>
<td>3032.40</td>
<td>89.9</td>
</tr>
<tr>
<td>31</td>
<td>0.38</td>
<td>67.20</td>
<td>8.42</td>
<td>75.62</td>
<td>3108.30</td>
<td>46.0</td>
</tr>
<tr>
<td>50</td>
<td>0.98</td>
<td>127.90</td>
<td>13.60</td>
<td>141.50</td>
<td>3160.40</td>
<td>24.5</td>
</tr>
<tr>
<td>80</td>
<td>0.38</td>
<td>174.60</td>
<td>21.73</td>
<td>196.33</td>
<td>3243.33</td>
<td>18.5</td>
</tr>
<tr>
<td>100</td>
<td>0.39</td>
<td>209.11</td>
<td>27.17</td>
<td>236.28</td>
<td>3299.75</td>
<td>15.8</td>
</tr>
<tr>
<td>120</td>
<td>0.39</td>
<td>242.90</td>
<td>32.63</td>
<td>275.53</td>
<td>3358.90</td>
<td>13.8</td>
</tr>
</tbody>
</table>

Each thread in a block is given a unique thread-index, in order to identify threads in different blocks. Therefore, to differentiate any two threads a thread ID can be conceived as follows: \( \text{thread ID} = (\text{block-index}, \text{thread-index}) \) where block-index is unique among any block. Blocks are organized into a grid. Thread-index and block-index may be formed of one, two or three dimensions. For the computations in HAC methods, it is found easier to conceive a one-dimensional grid [22].

D. General Lessons Learnt from Parallelizing the HAC computations using CUDA on GPU

1. With no previous experience in graphics programming or using CUDA for parallel computing, the CUDA based programming framework is learned step-by-step from fundamentals to implementation, putting the language theory into practical coding through design and development.

2. Necessary understanding of CUDA tools, techniques and structure building from simple codes for minimum distance computations using matrix reduction to more the complex computations of distances is gained.

3. The foundational issues in using CUDA for GPGPU are: efficient practice of multi-threaded programming and the different memory hierarchy, which is understood and handled.

4. Simple implementation using less complex memory management still yields remarkable speed gains when compared to a CPU performance on desk top computer.

5. Use of standard CUDA libraries renders significant parallel performance: identification of minimum distance pairs in virtually one pass for any data size.

V. CONCLUSIONS

We implemented the computations of the single linkage HAC method in parallel on the GPU using CUDA. Speed gains about 15 to 90 times than the CPU have been achieved for various combinations of cluster parameters. The computational speed gain on HAC single linkage method is almost twice as obtained for the complete linkage method [22]. This is due to the fact that identification maximum distance pair needs a custom developed function whereas the identification of the minimum distance pair uses \( \text{cublasIsamin} \) which is a built in CUDA library function. The novelities in our implementation include: (1) Speedup of accessing data in the global memory using a 1D half similarity array, (2) Identifying the minimum distance pairs in virtually one pass using the \( \text{cublasIsamin} \) function, and (3) Completing all the HAC tasks within the GPU itself without the necessity to transfer intermediate results to the CPU. The issues rising from the implementation of HAC methods using CUDA have been discussed and generalized. The optimal block size for CUDA processing on GPU with 128 internal processors should be 8. Maximum number of observations that can be currently clustered is limited by the size of distance matrix as it was confined to one grid. The effect of number of gene expressions and their dimensions on speed performance is studied. By efficient use of threads in the processors and the choice of using the large global memory, a trade-off has been made between scalability speedups and ease of memory management and hence programming. Moreover, the need to maintain huge data, align threads to the local and shared memories are avoided; thus minimizing ‘synchronization of threads’. We propose that the speedup depends more on the number of vectors used and its dimensions for a given GPU configuration and the CUDA parameters such as block size etc. are to be selected accordingly to maximize performance. Moreover, the CUDA codes used in parallelizing the HAC algorithm on the GPU can be ported to various other programming platforms with little conscious efforts; while the performance would depend on the hardware capabilities of the GPU on which the program is invoked. It would be possible to achieve similar results as demonstrated in our research, on any GPU hardware with comparable performance while utilizing other programming platforms. The future plan is to include the use of multiple blocks per grid. The number of blocks per grid should be determined based on the dimensional size of the data, the split that could be made in the data set and the number of multiprocessors in the GPU. CUDA permits the use of a large range of possibilities, but the choice should be based on what makes the kernel run most efficiently. Several aspects of improving the performance of software performance for applications coded in CUDA are addressed in [37]. The parallel computational capabilities of GPU lead into the new era where higher-level programming models are required for heterogeneous parallel computing. It is vital to understand the economic parallel computing architectures of the GPU and the use of such architectures is essential for implementing low-cost and efficient data mining algorithms without compromising the accuracy of the results. The latest family of GPUs such as GeForce GTX 400/500 is based on NVIDIA’s Fermi architecture which is optimized for next generation graphic applications and general purpose computations [36]. For instance, the 3300GTX hosts a GPGPU, with Fermi architecture, which is optimized for floating-point operations and memory bandwidth to provide the performance required for processing very large volumes of data. The Fermi architecture is poised to deliver the ever increasing performance needs at lower power consumption requirements of today’s signal and image data processing.
applications, thus rendering an economic solution for high throughput, low latency, and scalable computing capabilities.

ACKNOWLEDGMENT

We acknowledge all those who were in some way or other supportive of this research work at Centre for Advanced Information Systems (CAIS), Nanyang Technological University, Singapore.

REFERENCES


