Accelerating K-Means on the Graphics Processor via CUDA

Mario Zechner
Know-Center
Inffeldgasse 21a
8010 Graz, Austria
mzechner@know-center.at

Michael Granitzer
Know-Center
Inffeldgasse 21a
8010 Graz, Austria
mgrani@know-center.at

Abstract

In this paper an optimized k-means implementation on the graphics processing unit (GPU) is presented. NVIDIA's Compute Unified Device Architecture (CUDA), available from the G80 GPU family onwards, is used as the programming environment. Emphasis is placed on optimizations directly targeted at this architecture to best exploit the computational capabilities available. Additionally drawbacks and limitations of previous related work, e.g. maximum instance, dimension and centroid count are addressed. The algorithm is realized in a hybrid manner, parallelizing distance calculations on the GPU while sequentially updating cluster centroids on the CPU based on the results from the GPU calculations. An empirical performance study on synthetic data is given, demonstrating a maximum 14x speed increase to a fully SIMD optimized CPU implementation.

1 Introduction

In the last decades the immense growth of data has become a driving force to develop scalable data mining methods. Machine learning algorithms have been adapted to better cope with the mass of data being processed. Various optimization techniques lead to improvements in performance and scalability among which parallelization is one valuable option.

One of the many data mining methods widely in use is partitional clustering which is formally defined as "the organization of a collection of patterns (usually represented as a vector of measurements, or a point in a multidimensional space) into clusters based on similarity" [1]. The application of clustering is widespread among many different fields, such as computer vision [2], computational biology [3][4] or text mining [5]. A non-optimal solution to the NP-hard problem of partitional clustering was proposed by Lloyd in [6] who's most well known variant is the k-means algorithm in [7]. The popularity of k-means is explainable by its low implementational complexity and well understood mathematical properties. However, k-means will only find non-optimal local-minima, depending on the initial configuration of centroids. This is also known as the seeding problem and was addressed in various works. Recently a new strategy yielding better clustering results was introduced in [8]. Still, the run-time performance of k-means is a concern as data is growing rapidly, especially when finding the correct parameter of k can only be done by performing several runs with different numbers of clusters and initial seedings.

With the appearance of programmable graphics hardware in 2001, using the GPU as a low-cost highly parallel streaming co-processor became a valuable option. In the following years scientific interest in this new architecture resulted in numerous publications demonstrating the advantages of GPUs over CPUs when used for data parallel tasks. Much attention was focused on transferring common parallel processing primitives to the GPU and creating frameworks to allow for more general purpose programming [9][10]. The most problematic aspect of this undertakings was transforming the problems at hand into a graphics pipeline friendly format, a task needing knowledge about graphics programming. The reader shall be referred to [11] where an in-depth discussion on mapping computational concepts to the GPU can be found. This entry barrier was recently lowered by the introduction of NVIDIA's CUDA [12] as well as ATI's Close to Metal Initiative [13]. Both were designed to enable direct exploitation of the hardware's capabilities circumnavigating the invocation of the graphics pipeline via an API such as OpenGL or DirectX. In this work CUDA was chosen due to its more favorable properties, namely the high-level approach employed by its seamless integration with C and the quality of its documentation.

In this paper a parallel implementation of k-means on the GPU via CUDA is discussed. Section 3 discusses the sequential and parallel variants of k-means leading to section 2 where related work is investigated. Section 4 gives

	Takizawa and Kobayashi [14]	Hall and Hart [15]	Cao et. al. [16]
CPU	Intel P4 3.2 Ghz	AMD Athlon 2800+	Intel P4 3.4 Ghz
Compiler	GNU C++ 3.3.5	?	Intel C++
Optimizations	SSE2	?	SSE2, Hyper-Threading
GPU	NVIDIA Geforce 6600 Ultra	NVIDIA GeforceFX 5900	NVIDIA Geforce 6800 GT
Speedup	4	2-3	4

Table 1. Summary of previous GPU-based k-means implementations. The column speedup gives the relative speedup of the GPU version to the CPU version based on total runtime

an overview of CUDA's properties and programming model followed by section 5 describing the concrete parallel implementation of k-means on the GPU. A comparison of the GPU implementation versus an optimized sequential CPU implementation is given in section 6. Finally, section 7 concludes this paper.

2 Related Work

To the best of the authors' knowledge, three different implementations of k-means on the GPU exist. All three implementations are similar to the parallel k-means implementation outlined in section 3.3 formulated as a graphics programming problem.

In [14] Takizawa and Kobayashi try to overcome the limitations imposed by the maximum texture size by splitting the data set and distributing it to several systems each housing a GPU. A solution to this problem via a multi-pass mechanism was not considered. Also the limitation on the maximum number of dimensions was not tackled. It is also not stated whether the GPU implementation produces the same results as the CPU implementation in terms of precision.

Hall and Hart propose two theoretical options for solving the problem of limited instance counts and dimensionality: multi-pass labeling and a different data layout within the texture [15]. None of the approaches have been implemented though. In addition to the naive k-means implementation the data is reordered to minimize the number of distance calculations by only calculating the metrics to the nearest centroids. This is achieved by finding those centroids by traversing a previously constructed kd-tree. The authors could not observe any problems caused by the non standard compliant floating point arithmetic implementations on the GPU, stating that the exact same clusterings have been found.

The approach of Cao et. al. in [16] differs in that the centroid indices are stored in an 8-bit stencil buffer instead of the frame buffer limiting the number of total centroids to 256. Limitations in dimensionality and instance counts due to maximum texture sizes are solved via a costly multi-pass approach. No statements concerning precision of the GPU version were made.

Summarizing the presented previous work the following can be observed:

- All implementations suffer from architectural constraints such as maximum texture size limiting the number of instances, dimensions and clusters. The limitations can only be overcome by employing more costly multi-pass approaches.
- Not all publications state the exact conditions the implementations were tested under. A direct comparison is not strictly possible. However, the given numbers indicate congruent results yielding an average speedup of a factor between 3 to 4.
- The GPU implementation's performance increases as the problem at hand grows bigger in dimensionality as well as instance and centroid count.
- Only one paper mentioned potential impact of the non standard-compliant floating point arithemtics implemented on GPU's. No effects have been observed.

Based on the previous work the main contributions of this paper are as follows:

1. A parallel implementation of standard k-means on NVIDIA's G80 GPU generation using the non-graphics oriented programming model of CUDA.
2. Removal of the limitations inherent to classical graphics-based GPGPU programming approaches for k-means, namely the number of instances, dimensions and centroids enabling large scale clustering problems to be tackled on the GPU.
3. Investigation of precision issues due to the non IEEE single precision floating point compliance of modern GPU's.
4. Performance evaluation of the presented implementation in comparison to an aggressively optimized single

core CPU implementation, using SSE3 vectorization as well as loop unrolling optimizations, showing high speedups when compared to the average speedup of previous GPU-based implementations.

5. Evaluation of on-chip memory throughput as well as floating point operation performance.

3 K-Means Clustering

In this section a definition of the k-means problem is given as well as non-optimal sequential and parallel algorithmic solutions. Additionally the computational complexity is discussed.

3.1 Problem Definition

The k-means problem can be defined as follows: a set $\mathcal{X}$ of n data points $\mathbf{x_i} \in \mathbb{R}^d, i = 1, \ldots, n$ as well as the number of clusters $k \in \mathbb{N}^+ < n$ is given. A cluster $\mathcal{C}_j \subset \mathcal{X}, j = 1, \ldots, k$ with a centroid $\mathbf{c_j} \in \mathbb{R}^d$ is composed of all points in $\mathcal{X}$ for which $\mathbf{c_j}$ is the nearest centroid using euclidean distance. The optimal set $\mathcal{C}$ of k centroids can be found by minimizing the following potential function:

$$\phi = \sum_{i=1}^{n} \min_{\mathbf{c_j} \in \mathcal{C}} \mathcal{D}(\mathbf{x_i}, \mathbf{c_j})^2 \tag{1}$$

$\mathcal{D}$ is a metric in $\mathbb{R}^d$, usually the euclidean distance. Solving equation 1 even for two clusters was proven to be NP-hard in [17]. However, a non-optimal solution for the k-means problem exists and will be described in the following section. For the rest of the discussion it is assumed that the set of data points $\mathcal{X}$ is already available in-core, that is loaded to memory.

3.2 Sequential K-Means

In [7] MacQueen describes an algorithm that locally improves some clustering $\mathcal{C}$ by iteratively refining it. An initial clustering $\mathcal{C}$ is created by choosing k random centroids from the set of data points $\mathcal{X}$. This is known as the seeding stage. Next a labeling stage is executed where each data point $\mathbf{x_i} \in \mathcal{X}$ is assigned to the cluster C_j for which $\mathcal{D}(\mathbf{x_i}, \mathbf{c_j})$ is minimal. Each centroid $\mathbf{c_j}$ is then recalculated by the mean of all data points $\mathbf{x_i} \in \mathcal{C}_j$ via $\mathbf{c_j} = \frac{1}{|\mathcal{C}_j|} \sum_{\mathbf{x_i} \in \mathcal{C}_j} \mathbf{x_i}$. The labeling and centroid update stage are executed repeatedly until $\mathcal{C}$ no longer changes. This procedure is known to converge to a local minimum subject to the initial seeding [18]. Algorithm 1 describes the procedure in algorithmic terms. The next section demonstrates how this sequential algorithm can be transformed into a parallel implementation.

Algorithm 1 Sequential K-Means Algorithm

$\mathbf{c_j} \leftarrow$ random $\mathbf{x_i} \in \mathcal{X}, j = 1, \ldots, k$, s.t. $\mathbf{c_j} \neq \mathbf{c_i} \forall i \neq j$
repeat
  $\mathcal{C}_j \leftarrow \emptyset, j = 1, \ldots, k$
  for all $\mathbf{x_i} \in \mathcal{X}$ **do**
    $j \leftarrow \arg\min \mathcal{D}(c_j, x_i)$
    $\mathcal{C}_j \leftarrow \mathcal{C}_j \cup x_i$
  end for
  for all $\mathbf{c_j} \in \mathcal{C}$ **do**
    $\mathbf{c_j} \leftarrow \frac{1}{|\mathcal{C}_j|} \sum_{\mathbf{x_i} \in \mathcal{C}_j} \mathbf{x_i}$
  end for
until convergence

3.3 Parallel K-Means

In [19] Dhillon presents a parallel implementation of k-means on distributed memory multiprocessors. The labeling stage is identified as being inherently data parallel. The set of data points $\mathcal{X}$ is split up equally among p processors, each calculating the labels of all data points of their subset of $\mathcal{X}$. In a reduction step the centroids are then updated accordingly. It has been shown that the relative speedup compared to a sequential implementation of k-means increases nearly linearly with the number of processors. Performance penalties introduced by communication cost between the processors in the reduction step can be neglected for large n.

Algorithm 2 Parallel K-Means Algorithm

if threadId = 0 **then**
  $\mathbf{c_j} \leftarrow$ random $\mathbf{x_i} \in \mathcal{X}, j = 1, \ldots, k$, s.t. $\mathbf{c_j} \neq \mathbf{c_i} \forall i \neq j$
end if
synchronize threads
repeat
  for all $\mathbf{x_i} \in \mathcal{X}_{threadId}$ **do**
    $l_i \leftarrow \arg\min \mathcal{D}(c_j, x_i)$
  end for
  synchronize threads
  if threadId=0 **then**
    for all $\mathbf{x_i} \in \mathcal{X}$ **do**
      $\mathbf{c_{l_i}} \leftarrow \mathbf{c_{l_i}} + \mathbf{x_i}$
      $m_{l_i} \leftarrow m_{l_i} + 1$
    end for
    for all $\mathbf{c_j} \in \mathcal{C}$ **do**
      $\mathbf{c_j} \leftarrow \frac{1}{m_j} \mathbf{c_i}$
    end for
    if convergence **then**
      signal threads to terminate
    end if
  end if
until convergence

Since the GPU is a shared memory multiprocessor architecture this section briefly outlines a parallel implementation on such a machine. It only slightly diverges from the approach proposed by Dhillon. Processors are now called threads and a master-slave model is employed. Each thread is assigned an identifier between 0 and $t-1$ where t denotes the number of threads. Thread 0 is considered the master thread, all other threads are slaves. Threads share some memory within which the set of data points $\mathcal{X}$, the set of current centroids $\mathcal{C}$ as well as the clusters $\mathcal{C}_j$ reside. Each thread additionally owns local memory for miscellaneous data. It is further assumed that locking mechanisms for concurrent memory access are available. Given this setup the sequential algorithm can be mapped to this programming model as follows.

The master thread initializes the centroids as it is done in the sequential version of k-means. Next $\mathcal{X}$ is partitioned into subsets $\mathcal{X}_i, i = 0, \dots t$. This is merely an offset and range calculation each thread executes giving those $\mathbf{x_i}$ each thread processes in the labeling stage. All threads execute the labeling stage for their partition of X. The label of each data point $\mathbf{x_i}$ is stored in a component l_i of an n-dimensional vector. This eliminates concurrent writes when updating clusters and simplifies bookkeeping. After the labeling stage the threads are synchronized to ensure that all data for the centroid update stage is available. The centroid update stage could then be executed by a reduction operation. However, for the sake of simplicity it is assumed that the master thread executes this stage sequentially. Instead of iterating over all centroids the master thread iterates over all labels partially calculating the new centroids. A k-dimensional vector $\mathbf{m}$ is updated in each iteration where each component m_j holds the number of data points assigned to cluster $\mathcal{C}_j$. Next another loop over all centroids is performed scaling each centroid $\mathbf{c_j}$ by $\frac{1}{m_j}$ giving the final centroids. Convergence is also determined by the master thread by checking whether the last labeling stage introduced any changes in the clustering. Slave threads are signaled to stop execution by the master thread as soon as convergence is achieved. Algorithm 2 describes the procedure executed by each thread.

3.4 Computational Complexity

In this section the number of operations executed by k-means in each iteration is investigated. This number is equal for both implementations. It therefore serves as the basis for comparing runtime behavior in section 6.

For the computational complexity analysis each floating point operation is counted as one computational unit. Additions, multiplications and comparisons are considered to be floating point operations. Also, the seeding stage is ignored in this analysis.

The labeling stage consists of evaluating the distance from each data point $\mathbf{x_i}$ to each centroid $\mathbf{c_j}$. Given an euclidean distance metric each distance calculation consists of one subtraction, one multiplication and one addition per dimension totaling in $3d$ operations. Additionally a square root is calculated adding another operation per distance calculation. Finding the centroid nearest to a data point $\mathbf{x_i}$ is an iterative process where in each iteration a comparison between the last minimal distance and the current distance is performed. This adds another operation to the total number of operations per labeling step. There is a total of nk labeling resulting in the total numbers of operations of

$$O_{labeling} = 3nkd + 2nk = nk(3d + 2) \tag{2}$$

operations for the labeling stage in each iteration.

In each iteration of the centroid update stage the mean for each cluster $\mathcal{C}_j$ is calculated consisting of adding $|C_j|$ d-dimensional vectors as well as dividing each component of the resulting vector by $|C_j|$. In total n d-dimensional vectors are added yielding nd operations plus kd operations for the scaling of each centroid $\mathbf{c_j}$ by $\frac{1}{|C_j|}$. For the labeling stage there are thus

$$O_{update} = nd + kd = d(n + k) \tag{3}$$

operations executed per k-means iteration. The total number of operations per k-means iteration is given by

$$O_{iteration} = O_{labeling} + O_{update} = nk(3d+2) + d(n+k) \tag{4}$$

From equations 2 and 3 it can be observed that the labeling stage is clearly the most costly stage per iteration. If $d \ll n$ and $k \ll n$ the labeling stage contributes insignificantly to the total number of operations making the labeling stage the dominant factor.

4 CUDA

With the advent of the unified shader model the separation of vertex and fragment shader processors in hardware has vanished. Shader processors can now be configured to perform both tasks depending on the requirements of the application. Also, a third kind of shader, the geometry shader was introduced that allows generation of geometry in hardware on the fly [20]. Starting from the G80 family of GPUs NVIDIA supports this new shader model resulting in a departure from previous GPU designs. The GPU is now composed of so called multiprocessors that house a number of streaming processors ideally suited for massively data-parallel computations.

NVIDIA's CUDA is build on top of this new architecture eliminating the need to reformulate computations to the

graphics pipeline. The GPU is viewed as a set of multiprocessors executing concurrent threads in parallel. Threads are grouped into thread blocks and execute the same instruction on different data in parallel. One or more thread blocks are directly mapped to a hardware multiprocessor where time sharing governs the execution order. Within one block threads can be synchronized at any execution point. A certain execution order of threads within a block is not guaranteed. Blocks are further grouped into a grid, communication and synchronize among blocks is not possible,execution order of blocks within a grid is undefined. Threads and blocks can be organized in three and two dimensional manners respectively. A thread is assigned an id depending on its position in the block, a block is also given an id depending on its position within a grid. Thread and block id of a thread is accessible at runtime allowing for specific memory access patterns based on the chosen layouts. Each thread on the GPU executes the same procedure known as a kernel [12].

Threads have access to various kinds of memory. Each thread has very fast thread local registers and local memory assigned to it. Within one block all threads have access to block local shared memory that can be accessed as fast as registers depending on the access patterns. Registers, local memory and shared memory are limited resources. Portions of device memory can be used as texture or constant memory which benefit from on-chip caching. Constant memory is optimized for read-only operations, texture memory for specific access patterns. Threads also have access to uncached general purpose device memory or global memory [12].

Various pitfalls exist that can degrade performance of the GPU. Shared memory access by multiple threads in parallel can produce so called bank conflicts serializing execution of those threads and therefore reducing parallelism. Second, when accessing global memory addresses have to be a multiple of 4, 8 or 16, otherwise an access might be compiled to multiple instructions and therefore accesses. Also, addresses accessed simultaneously by multiple threads in global memory should be arranged so that memory access can be coalesced into a single continuous aligned memory access. This is often referred to as memory coalescing. Another factor is so called occupancy. Occupancy defines how many blocks and therefore threads are actually running in parallel. As shared memory and registers are limited resources the GPU can only run a specific number of blocks in parallel. It is therefore mandatory to optimize the usage of shared memory and registers to allow to run as many blocks and threads in parallel as possible [12].

The CUDA SDK gives the developer easy to use tools that fully integrate with various C++ compilers. Code for the GPU is written in a subset of C with some extensions and can coexist with CPU (host) code in the same source file. The host code is responsible for setting up the layout of blocks and threads as well as uploading data to the GPU. Kernel execution is performed asynchronously, primitives to synchronize between CPU and GPU code are available. Debugging of device code is possible but only in an emulation environment that runs the kernel on the CPU in heavyweight threads which does not simulate all peculiarities of the GPU. For more in depth information on CUDA the reader is referred to [12].

5 Parallel K-Means via CUDA

This section describes the CUDA based implementation of the algorithm outlined in section 3.3. In the first sub section the overall program flow is described. The next subsection presents the labeling stage on the GPU followed by section 5.3 outlining the data layout used and CUDA specific optimizations employed to further speed up the implementation.

5.1 Program Flow

The CPU takes the role of the master thread as described in section 3.3. As a first step it prepares the data points and uploads them to the GPU. As the data points do not change over the course of the algorithm they are only transfered once. The CPU then enters the iterative process of labeling the data points as well as updating the centroids. Each iteration starts by uploading the current centroids to the GPU. Next the GPU performs the labeling as described in section 5.2. The results from the labeling stage, namely the membership of each data point to a cluster in form of an index, are transfered back to the CPU. Finally the CPU calculates the new centroid of each cluster based on these labels and performs a convergence check. Convergence is achieved in case no label has changed compared to the last iteration. Optionally a thresholded difference check of the overall movement of the centroids can be performed to avoid iterating infinitely for some special cluster configurations.

5.2 Labeling Stage

The goal of the labeling stage is to calculate the nearest centroid for each data point and store the index of this centroid for further processing by the centroid update stage on the CPU. Therefore each thread has to calculate which data points it should process, label it with the index of the closest centroid and repeat this for any of its remaining data points. The task for each thread is thus divided into two parts: calculate and iterate over all data points belong to the thread according to a partitioning schema and performing the actual labeling for the current data point. The following paragraphs will thus first discuss the partitioning schema

and the first part of this task followed by a description of the actual labeling step.

As discussed in section 4 the GPU slightly differs from the architecture assumed in section 3.3. Threads are additionally grouped into blocks that share local memory. Instead of assigning each thread a chunk of data points, each block of threads is responsible for one or more chunks. One such chunk contains t data points where t is the number of threads per block. As the amount of threads per block as well as blocks is limited by various factors, such as used registers, each block processes not only one but several chunks depending on the total amount of data points. Denoting the amount of data points by n then

$$n_{chunks} = \lceil n/t \rceil \tag{5}$$

gives the number of chunks to be processed. Note that the last chunk does not have to be fully filled as n does not have to be a multiple of t. This chunks have to be partitioned among the number of blocks b. Two situations can arise:

1. $n_{chunks} \mod b = 0$, no block is idle
2. $n_{chunks} \mod b \neq 0$, $b - n_{chunks}$ blocks are idle

Therefore each block processes at least $\lfloor n_{chunks}/b \rfloor$ chunks. The first $n_{chunks} \mod b$ blocks process the remaining chunks. For each chunk one thread within a block labels exactly one data point. For chunks that have less data points than there are threads within a block some threads will be idle and not process a data point. Based on the partitioning schema described each thread processes at most n_{chunks} data points. For each data point a thread therefore has to calculate the index of the data point based on it's block and thread id. This is done iteratively in a loop. The thread starts by calculating the index of its data point of the first chunk to be processed by the thread's block expressed by $block.id + thread.id$. In each iteration the next data point's index is calculating by adding tb to the last data points index. In case the calculated index is bigger than $n-1$ the thread has processed all it's data points. No thread can terminate before the other threads within the same block so any thread that is done processing all its data points has to wait for the other threads to finish processing their remaining data points. Therefore each thread iterates $\lceil n/tb \rceil$ times and simply does not execute the labeling code in case its current data point index is bigger than $n - 1$. To minimize the number of idling threads it is therefore mandatory to adjust the number of blocks to the number of data points minimizing $n \mod tb$.

The actual labeling stage is again composed of two distinct parts. A thread has to calculate the distance of its current data point to each centroid. In the implementation presented here all threads within a block calculate the distance to the same centroid at any one time. This allows loading the current centroid to the block's local shared memory accessible by all threads within the block. For each centroid the threads within the block therefore each load a component of the current centroid to shared memory. Each thread then calculates the distance from their data point to the centroid in shared memory fetching the data point's components from global memory in a coalesced manner. See section 5.3 on the data layout used for coalescing reads and writes. Loading the complete centroid to memory limits the amount of dimensions as shared memory is restricted to some value, on the hardware used it's 16 kilobytes. Given that components are encoded as 32-bit floating point values this roughly equals a maximum dimension count of 4000. To allow for unlimited dimensions the process of loading and calculating the distance from a data point to a centroid is done in portions. In each iteration t components of the centroid are loaded to shared memory. For each component the partial euclidean distance is calculated. Depending on d not all threads have to take part in loading the current components to memory, so some threads might idle. When all threads have evaluated the nearest centroid the resulting label, being the index of the centroid a data point is nearest to, is written back to global memory. The labels for each data point are stored in an additional vector component.

After all blocks have finished processing their chunks the CPU is taking over control again, downloading the labels calculated for constructing the new centroids and checking for convergence. The next section describes the data layout as well as other optimizations.

5.3 Data Layouts & Optimizations

A GPU-based implementation of an algorithm that is memory bound, as is the case with k-means, can yield very poor performance when the GPU's specifics are not taken into account. For memory throughput these specifics depend on the memory type used for storing and accessing data on the GPU as described in section 4. For the k-means implementation presented in this paper global memory was chosen as the storage area for the data points and centroids. As data points are only read during the labeling stage on the GPU, storage in constant or texture memory might have increased memory throughput to some degree. However, texture and constant memory restrict the maximum amount of data and therefore processable data points and centroids, a drawback earlier GPU-based k-means implementations suffered from as described in section 2. Global memory on the other hand allows gather and scatter operations and permits to use almost all of the memory available on the GPU. For global memory coalescing reads and writes are mandatory to achieve the best memory throughput. All vectors are assumed to be of dimensionality d and stored in dense form.

As described in the last section centroids are loaded from

global memory to shared memory in portions, each portion being made up of at most t components. As t threads read in subsequent components at once the centroids are stored as rows in a matrix, achieving memory coalescing.

Data points are stored differently due to the order in which components are accessed. Here, each thread accesses one component of its current data point simultaneously to the other threads. Therefore data points are stored column wise again providing memory coalescing. Additionally a component is added as the first component of each vector where each threads writes the label of the closes centroid to for further processing by the CPU. This layout also allows downloading this labels in a bulk operation.

For both centroids and data points special CUDA API methods where used that allocate memory at address being a multiple of 4 yielding the best performance.

As the implementation of k-means using an euclidean distance metric is clearly memory bound further optimizations have been made by increasing occupancy. This was achieved by decreasing the amount of registers each thread uses. Specifically counter variables for the outer loops are stored in shared memory. This optimization increased performance by around 25%. The program executed by each thread uses 10 registers. The optimal number of threads is therefore 128 according to the NVIDIA CUDA Occupancy Calculator included in the CUDA SDK.

As descibed in section 5.2 partial or entire thread blocks can be idle depending on the ratio between the number of blocks and threads within a block to the number of data points. To reduce the effect of idle blocks on performance the block count is adapted to the number of data points to be processed, minimizing $n_{chunks} \mod b$.

The next section discusses experiments and their results for the k-means implementation presented in this section.

6 Experiments & Results

Experimental results where obtained on artificial data sets. As the performance is not dependant on the actual data distribution the synthetic data sets were composed of randomly placed data points. To observe the influence of the number of data points on the performance data sets with 500, 5000, 50.000 and 500.000 instances were created. For each instance count 3 data sets were created with 2, 20 and 200 dimensions.

The sequential k-means implementation and the centroid update phase for the gpu-base k-means was coded in C using the Visual C++ 2005 compiler as well as the Intel C++ compiler 10.1. For both compilers full optimizations were enabled, favoring speed over size as well as using processor specific extensions like SSE3. In the case of the Intel C++ compiler all vector related operations such as distance measurements, additions and scaling were vectorized using SSE3. The CUDA portions of the code were compiled using the CUDA Toolkit 2.0.

The test system was composed of an Intel Core 2 Duo E8400 CPU, 4 GB RAM running Windows XP Professional with Service Pack 3. The GPU was an NVIDIA GeForce 9600 GT hosting 512 MB of RAM, the driver used was the NVIDIA driver for Windows XP with CUDA support version 178.08.

Figures 1 and 2 present the speedups gained by using the GPU for the labeling stage. While full optimizations were turned on for the Visual C++ the GPU-based implementation outperformed it by a factor of 4 to 43 for all but the smallest data set. A clear increase in performance can be observed the higher the number of instances dimensions and clusters.

For the fully optimized Intel C++ version the speedups are obviously smaller as this version makes use of the SIMD instruction-set of the CPU. A speedup by a factor of 1.5 to 14 can be observed for all but the smallest data set. Interestingly this version performs better for lower dimensionality for high instance counts. This is due to the fact that as the centroid update time decreases due to optimization the transfer time starts to play a bigger role. Nevertheless there is still a considerable speedup observable.

The diagrams in figure 3 also explain why the GPU-based implementation does not match the CPU implementation for very small data sets. From the plot it can be seen that for 500 data points nearly all the time is spent on the GPU. This time span also includes the calling overhead for invoking the GPU labeling stage. This invocation time actually takes longer than labeling the data items.

The GPU-based implementation is clearly memory bound as there are more memory accesses than floating point operations. Therefore the approximate data throughput rate for the labeling stage was also computed. The values ranged from 23GB/s to 44GB/s depending on the instance and cluster count as well as dimensionality. For the used hardware the peak performance is given as 57.6 GB/s. Therefore we are highly confident that the implementation is nearly optimal. Due to being memory bound the GFLOP counts do of course not reach the hardwares peak values. 26GFLOP/s to 36GFLOP/s could be achieved approximately.

For some test runs slight variations in the resulting centroids were observed. These variations are due to the use of combined multiplication and addition operations (MADD) that introduce rounding errors. Quantifying these errors was out of the scope of this work, especially as no information from the vendor on the matter was available.

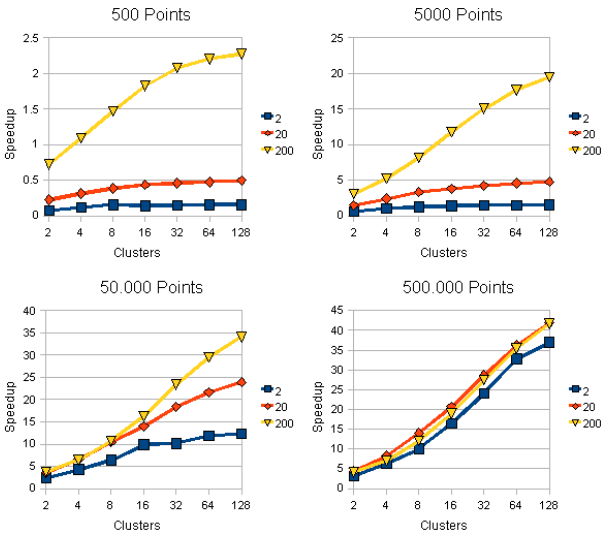

Figure 1. Speedup measured against the Visual C++ compiler for various instance counts and dimensions

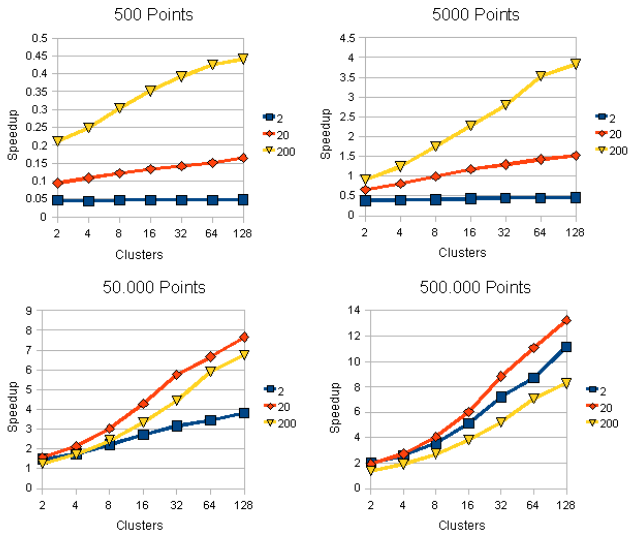

Figure 2. Speedup measured against Intel C++ compiler for various instance counts and dimensions

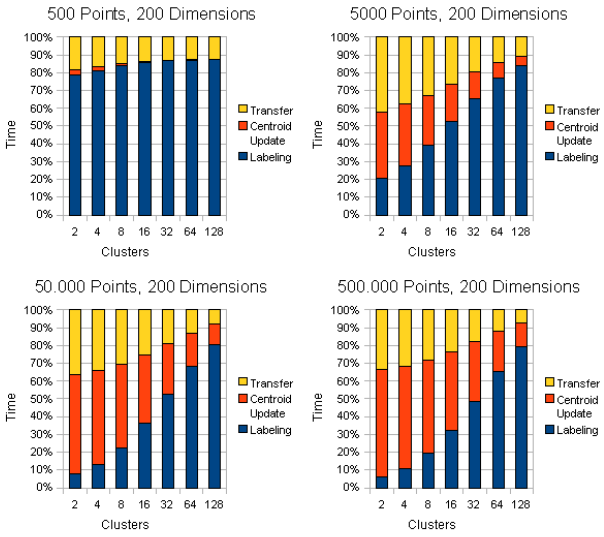

Figure 3. Percentage of time used for the different stages on the GPU

7 Conclusion & Future Work

Exploiting the GPU for the labeling stage of k-means proved to be beneficial especially for large data sets and high cluster counts. The presented implementation is only limited in the available memory on the GPU and therefore scales well. However, some drawbacks are still present. Many real-life data sets like document collections operate in very high dimensional spaces where document vectors are sparse. The implementation of linear algebra operations on sparse data on the GPU has yet to be solved optimally. Necessary access patterns such as memory coalescing make this a very hard undertaking. Also, the implementation presented is memory bound meaning that not all of the GPUs computational power is harvested. Finally, due to rounding errors the results might not equal the results obtained by a pure CPU implementation. However, our experimental experience showed that the error is negligible.

Future work will involve experimenting with other k-means variations such as spherical or kernel k-means that promise to increase the computational load and therefore better suit the GPU paradigm. Also, an efficient implementation of the centroid update stage on the GPU will be investigated.

References

[1] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323,

September 1999.

[2] Jian Yi, Yuxin Peng, and Jianguo Xiao. Color-based clustering for text detection and extraction in image. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 847–850, New York, NY, USA, 2007. ACM.

[3] Dannie Durand and David Sankoff. Tests for gene clustering. In *RECOMB '02: Proceedings of the sixth annual international conference on Computational biology*, pages 144–154, New York, NY, USA, 2002. ACM.

[4] Adil M. Bagirov and Karim Mardaneh. Modified global k-means algorithm for clustering in gene expression data sets. In *WISB '06: Proceedings of the 2006 workshop on Intelligent systems for bioinformatics*, pages 23–28, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[5] Shi Zhong. Efficient streaming text clustering. *Neural Netw.*, 18(5-6):790–798, 2005.

[6] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–136, 1982.

[7] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[8] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *SODA*, pages 1027–1035. SIAM, 2007.

[9] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 908–916, New York, NY, USA, 2003. ACM.

[10] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[11] Mark Harris. Mapping computational concepts to gpus. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 50, New York, NY, USA, 2005. ACM.

[12] Nvidia cuda site, 2007.

[13] Ati close to metal guide, 2007.

[14] Hiroyuki Takizawa and Hiroaki Kobayashi. Hierarchical parallel processing of large scale data clustering on a pc cluster with gpu co-processing. *J. Supercomput.*, 36(3):219–234, 2006.

[15] Jesse D. Hall and John C. Hart. Gpu acceleration of iterative clustering. Manuscript accompanying poster at GP2: The ACM Workshop on General Purpose Computing on Graphics Processors, and SIGGRAPH 2004 poster (2004).

[16] Feng Cao, Anthony K. H. Tung, and Aoying Zhou. Scalable clustering using graphics processors. In *WAIM*, pages 372–384, 2006.

[17] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay. Clustering large graphs via the singular value decomposition. *Mach. Learn.*, 56(1-3):9–33.

[18] Leon Bottou and Yoshua Bengio. Convergence properties of the K-means algorithms. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 585–592. The MIT Press, 1995.

[19] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Parallel Data Mining, Lecture Notes in Artificial Intelligence*, pages 245–260, 2000.

[20] David Luebke and Greg Humphreys. How gpus work. *Computer*, 40(2):96–100, 2007.