Improving the k-Nearest Neighbour Algorithm with CUDA

Graham Nolan

*This report is submitted as partial fulfilment
of the requirements for the Honours Programme of the
School of Computer Science and Software Engineering,
The University of Western Australia,
2009*

Abstract

Accurate and efficient data classification techniques are of vital importance to many problems in computing ranging from medical applications to data mining and artificial intelligence. As technology improves the need to process larger amounts of data increases, stimulating demand for more powerful and affordable techniques for doing so. The recent release of CUDA by Nvidia corporation gives developers the opportunity to perform general purpose parallel data processing using off-the-shelf hardware with relative simplicity and minimal expense.

This project explores the application of CUDA to data classification using the k-Nearest Neighbour (kNN) algorithm. The kNN algorithm was chosen due to its relative simplicity and high potential for parallelisation. A number of possible routes for optimising current parallel implementations of a kNN classifier in CUDA were explored, particularly improvements to the sorting phase of the algorithm which had previously been serialised. A number of different modifications to the algorithm were tested and compared to the previous implementation as well as each other, some being deemed to provide an acceptable performance improvement whilst others, though deemed unsuitable, highlighted the importance of efficient memory accesses on the GPU. Ultimately some improvement in the performance of a current parallel kNN classifier was demonstrated and avenues for future improvements to kNN in CUDA were identified.

Keywords: Honours, kNN, Nearest, Neighbour, CUDA, Parallel, Sort
CR Categories: A.2, I.7.2

Acknowledgements

First and foremost I'd like to thank my supervisor Associate Professor Amitava Datta for his encouragement and support throughout the year. Without his insight, ability to keep me motivated and genuine interest in the subject matter I would not have been able to complete this dissertation. I'd also like to thank my family and friends who put up with my crazy upside down schedule and helped to keep things in perspective so that I could move forward optimistically and with confidence. Despite this final year being extremely challenging and having to work towards many deadlines this has been without a doubt my most rewarding year of university both with respect to my studies and involvement in the many other facets of campus life.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

The k-Nearest Neighbour algorithm (kNN) is used to identify and classify objects in comparison to a set of known data, referred to as a training set. Unlike a primitive rote classifier the kNN algorithm is able to classify something in the absence of an exact match within the training set. It does this by calculating the 'distance' of its k nearest neighbours and then, usually based on the predominance of a certain class and their relative distances, assigns a class to the object in question. Distances are usually derived as some function of an object's attributes for example in the case of data about human body types a distance may be determined by a function of height, weight, skin tone, hair colour etc. This method of classification is relatively simple and effective but comes at considerable computational expense. In order to find an object's k nearest neighbours the algorithm must first calculate the distances from the object to all objects within the training set and then search through these distances to find the smallest k. The end result is an algorithm with an extremely high order of complexity. With a typical training set containing thousands of objects each with the potential for hundreds of attributes the performance limitations of kNN quickly become apparent.

kNN classification is used often in industry and in many scientific applications. It has been applied in areas such as medical imaging, entropy estimation, data mining, machine learning and content based image retrieval [4, 14, 5]. As our understanding in these areas grows so does the size and complexity of the data that needs to be processed. While there exists a number of strategies for improving performance of the algorithm in serial implementations such as condensing, editing and arranging reference points with kd-trees [14] these are still insufficiently scalable. Implementations that can deliver responses to large high-dimensional training sets in a small amount of time are highly sought after especially in data mining. Parallel processing represents a scalable way to perform the heavy lifting of the kNN algorithm without the complication of some serial optimisations.

Graphics Processing Units (GPUs) are high density parallel processors that specialise in producing high quality 3D graphics. Due to demand for increasingly graphically intensive applications modern GPUs now contain hundreds of parallel processor cores and are capable of handling thousands of simultaneous threads [1]. The recent release of the Compute Unified Device Architecture (CUDA) by Nvidia has made it possible for developers to harness the formidable processing power of newer model GPUs to perform general purpose parallel computing. CUDA applications can be written in the C language and compiled to run on commercial GPUs. This holds great promise for developers looking for scalable and relatively inexpensive solutions to a great number of parallel computing problems.

The ‘brute force’ implementation of the kNN algorithm is by nature easily parallelised due to the low level of interdependency between data operations in the distance calculation and sorting phases [4]. It holds that CUDA would be a relatively simple and effective way to adapt this version of kNN for parallel execution. Previous work done by Garcia et al. [4] demonstrates that implementing kNN in CUDA yields a considerable speedup over previous serial implementations. It is therefore worth considering the improvement of this implementation, particularly the sorting phase which is still performed serially for individual query points.

We will demonstrate a modified parallel implementation of the k-Nearest Neighbour algorithm in CUDA and evaluate its performance in comparison to the original. Experiments will be conducted to determine performance gains on a system running an Nvidia GTX 260 graphics card. We expect to observe a moderate speedup over the original algorithm and hope to identify some its limitations. Success here will further confirm the suitability of GPUs to process kNN queries and provide additional ideas on how to move forward in future research. Chapter 2 will examine the k-Nearest Neighbour algorithm in more detail as well as previous optimisations. Chapter 3 explains the GPU and CUDA. Chapter 4 describes how the kNN algorithm is implemented in parallel with CUDA. The methodology, results and their analysis are presented in Chapter 5. Chapter 6 concludes the paper and outlines future work in the area.

CHAPTER 2

The k-Nearest Neighbour Algorithm

2.1 Overview

The k-Nearest Neighbour (kNN) algorithm is used to classify objects in relation to a set of known objects called a training set. Unlike a rote classifier which requires an exact match within the training set the kNN algorithm looks for a subset of k objects that most closely resemble the one being classified. Once a suitable subset of k objects is found the algorithm then determines the classification usually based on the predominance of a particular class. Thus the advantage of kNN is that it can classify an object even in the absence of an exact match. An additional advantage is that it attempts to avoid incorrect or anomalous classifications by taking a variable sized sample rather than only comparing against a single object.[14]

The k-Nearest Neighbour (kNN) algorithm has a wide range of applications in modern day computing. One of its most common uses is in the field of data mining which has become increasingly valuable in recent times, particularly given the ubiquity of the internet. kNN can also be used to identify and classify objects in images, this is useful in a variety of applications most notably medical imaging where it can be used identifying anomalous data that may represent a tumour or disease. The field of Artificial Intelligence uses kNN in machine learning as a way of identifying known objects and classifying new ones.

2.2 The Algorithm

For the purposes of this paper objects being classified will be referred to as 'query points' and objects from the training set will be referred to as 'reference points'. Both query and reference points are usually represented as n-dimensional points in euclidean space. A typical training set can contain thousands of these reference points with n often exceeding one hundred.

The simplest version of the kNN algorithm is the ‘Brute Force’ implementation and consists of three stages. The first stage is to calculate all of the ‘distances’ from each query point to every reference point in the training set. The second stage is to sort these distances and select the k objects that are the closest from which the third and final stage of classification can be performed.

2.2.1 Distance Calculation

The distance referred to in kNN classification is typically derived from some function of a point’s attributes. Perhaps the most trivial example of this would be the euclidean distance between two points in n-dimensional space. However this is often not adequate as attributes may have to be scaled to prevent distance measures from being dominated by one of the attributes. For example if we were considering data that represented the physical attributes of people their height may range from 100 - 200cm and weight 30 - 150kg but their yearly income could be \$30,000 - \$150,000. If no scaling were applied then income would dominate the other attributes[14]. Scaling is best implemented by preproccessing the data rather than performing scaling during distance calculation.

The distance calculation stage is the most computationally intensive as the distance between each query point and every reference point must be determined. Each individual distance calculation requires finding the difference between each dimension, squaring the result, applying any relevant scaling and then summing the results for each dimension. In serial implementations of kNN much of the focus on optimisation of this phase revolves around modifying the training set to reduce the total calculations that need to be performed. However there is great potential for these calculations to be done in parallel as there is little to no dependency between operations. The whole distance calculation process can be performed as a differencing of the query and reference matrices followed by an aggregation of the squared results.

2.2.2 Sorting Distances

Sorting the calculated distances is necessary in order to find the nearest k objects. This is another computationally intensive stage. In serial implementations the most commonly used algorithm is Quicksort though any sorting algorithm can be used. One important shortcut that can be made when sorting distances for kNN is that only the smallest k elements need to be found, as such it is sufficient to initially sort the first k elements properly and then only perform sorting operations on subsequent elements that are smaller than the largest of k.

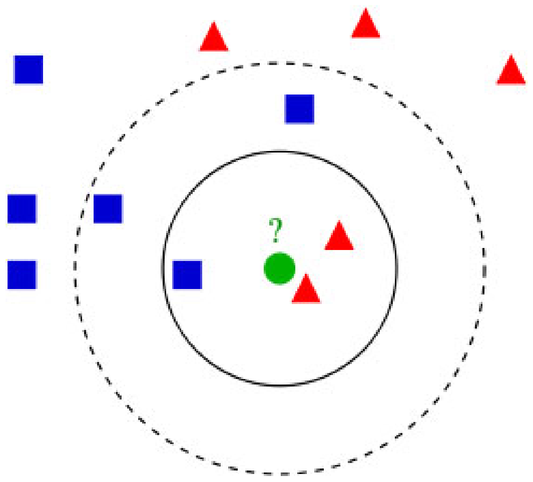

Figure 2.1: Shows the effect k can have on object classification.

2.2.3 Object Classification

Object classification is the least expensive part of the algorithm to compute but by no means the least important. Basic implementations classify query points by a majority vote between the k-nearest reference points. While this seems perfectly adequate it can be too sensitive to the value of k. For instance if k is large the algorithm may assign an inaccurate classification based on votes from reference points that are relatively far away from the query point. Conversely if k is too small accuracy can be effected by noise in the data i.e. a spot on an x-ray. This is demonstrated in Figure 2.1.

A more sophisticated approach, which is usually much less sensitive to the choice of k, weights each object's vote by its distance, where the weight factor is often taken to be the reciprocal of the squared distance [14]. This avoids the aforementioned problem by ensuring objects that are closer have a greater influence over the final classification of k. Naturally the choice of these weights is important so as to prevent localised noise from having too much influence over the final decision.

2.3 Enhancements

2.3.1 Condensing

Condensing is the process of reducing the size of the training set but in such a way that the overall classification accuracy is minimally affected. This reduction can be performed before or during execution and and can greatly speed up the classification of new objects [14]. The selection of a suitable heuristic for selecting which points to remove is imperative to ensure that this method is effective. In general the choice of heuristic is dependent on the type of data being processed for example a heuristic for condensing a training set representing types of soy bean may not be useful for a set representing car models.

2.3.2 Editing

'Editing' is the process of removing objects from the training set that may either be erroneous or representative of an outlier. If done correctly editing should improve the accuracy of object classification with the added benefit of reducing the size of the training set and facilitating quicker execution [14].

2.3.3 Data Structures

Arranging reference points in a data structure enables the kNN algorithm to find the closest objects much more quickly. Some very impressive performance gains have been demonstrated using kd-trees [15] and randomized data structures [2]. In the literature a general kd-tree implementation of kNN was shown to be three times faster than an unoptimised implementation [4].

2.4 Previous Work

2.4.1 Approximation

The Approximate Nearest Neighbour algorithm is an implementation of kNN that uses a randomized algorithm to quickly find nodes within an acceptably close proximity to the actual nearest neighbour [2]. The algorithm is an example of the application of a data structure to the training set combined with a 'randomised' heuristic to reduce the search time at the expense of some, albeit small, degree of accuracy. The authors report that the algorithm finds a fairly accurate k in

O(log3 n) time while the data structure requires O(n log n) space and can be constructed in order n squared time. ANN appears to be most suitable when applied to human speech data with a dimension up to 16 [2].

2.4.2 kd-trees in CUDA

The use of kd-trees is a well known optimisation to the kNN algorithm. Zhou et al. [15] demonstrate a parallel algorithm that generates kd-trees in realtime using CUDA. Though this is probably not directly applicable to the parallelisation of the kNN algorithm itself it is an important result for optimisation of serial implementations that make use of kd-trees as the generation of optimised data structures is notoriously slow when performed with a serial algorithm on a CPU. [15]

2.4.3 Parallelisation of kNN

Work on the parallelisation of the kNN algorithm in CUDA by Garcia et. al. serves as the starting point for the kNN enhancements demonstrated in this paper. The algorithm they put forward is a parallel implementation of the simple 'brute force' kNN algorithm. In the literature they reported a speedup of roughly 100 times faster than a serial CPU implementation and 40 times faster than serial kNN using kd-trees [4]. It was also found that the dimension of points had negligible impact on the overall computation time.

They decided that the implementation best suited to parallelisation was the brute force algorithm due to its simplicity and the low level of interdependencies between data points. The simple sequence of operations for the brute force algorithm are shown below.

Brute force algorithm:

1. Compute all the distances between the query point and reference points
2. Sort the computed distances.
3. Select the k reference points with the smallest distances
4. Classification vote by k nearest objects.
5. Repeat steps 1. to 4. for all query points.

CHAPTER 3

GPU and CUDA

Modern Graphics Processing Units (GPUs) are high density parallel processors that specialise in processing and generating high quality 3D graphics. Their most common use is in PCs where they improve overall system performance by freeing up the CPU to handle program flow control and general calculations. GPUs perform most of the ‘heavy lifting’ demanded by graphically intensive applications such as in video games or simulation software and are optimised to handle tasks in realtime that would take a CPU far too long to process.

Driven by insatiable market demand the complexity and power of Graphics Processing Units (GPUs), much like CPUs, continues to increase rapidly. Modern GPUs now contain hundreds of processor cores and are capable of handling thousands of simultaneous threads [1]. However despite their considerable power GPUs are still relatively affordable such that even the most expensive cards are within the reach of the average consumer.

In November 2006, Nvidia introduced CUDA (Compute Unified Device Architecture), a general purpose parallel computing architecture, making it possible for developers to program applications in the C language to perform parallel computing on commercially available GPUs that until recently were only capable of graphics processing. The affordability of GPUs and the relatively shallow learning curve associated with CUDA has made it extremely popular in the academic community. Research groups have reported algorithm speedups in the order of hundreds of times faster for parallel CUDA implementations when compared to their serial counterparts [15, 7, 4, 12].

3.1 Hardware

The Nvidia GeForce GTX 260 GPU is a shared memory parallel machine in that multiple processors use shared regions of memory. The GTX 260 card has 8 texture-processing clusters. Each cluster is sub-divided into three streaming

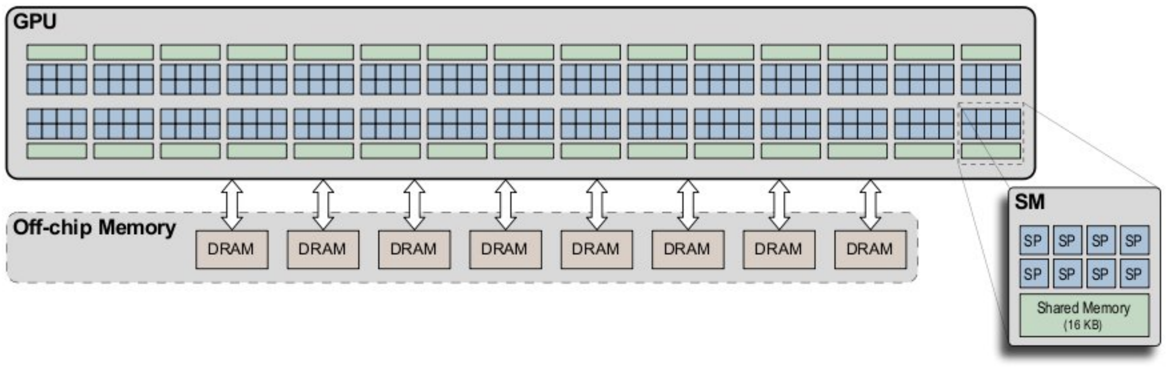

Figure 3.1: A diagram of the architecture of a 200 series card for illustrative purposes. The one shown here is a GeForce GTX 280 GPU with 240 Streaming Processors (SPs). The GTX 260 GPU is organised slightly differently and has a smaller total of 192 SPs [13].

multiprocessors (SMs) (up from two in the 8-series and 9-series GPUs), though each SM has 8 Streaming Processors (SP), just like the previous architectures. Hence each cluster has 24 SMs (three blocks of eight), making a card-wide total of 192 SPs each running at 1.242GHz [10]. Each SM is able to be assigned a different problem which allows GPU resources to be divided for optimum usage [12, 10, 11, 7]. The GTX 260 boasts a theoretical maximum memory bandwidth of 111.9GB/s [10] allowing it to make full use of its parallel processors and achieve a peak processing power of 715 GFlops while in comparison even the most expensive CPUs like the Intel QX9775 can only reach 51.2GFlops and the average CPU like the Core 2 Duo 8400 processes about 24GFlops [6]. The CUDA architecture allows for asynchronous device operation allowing the GPU to execute device code on its own without intervention from the CPU. This independence means that code executing on the GPU is not limited in any way by the speed of the CPU. The CPU is also free to perform its own tasks while waiting for kernels executing on the GPU.

3.2 The CUDA Programming Model

The CUDA architecture is designed to be a scaleable way of developing parallel applications in the C and C++ programming languages. Code that runs on both the host (CPU) and device (GPU) can be written and compiled in the same program [8]. A high level abstraction of CUDA is exposed to programmers through the CUDA API. There are three components to this abstraction:

1. A hierarchy of thread groups.
2. Shared memories.
3. Barrier Synchronization.

Programmers are able to parallelise applications through the use of these abstractions. Iterative tasks can be broken down into individual threads or collections of threads that perform operations on data in shared memory. The ability to enforce barrier synchronization ensures co-operation and introduces a level of control to the otherwise autonomous threads running on the GPU [1].

3.2.1 Kernels

A kernel is a function that can be executed in parallel on the the GPU. When invoked a kernel spawns copies of itself each running in parallel over multiple threads on the GPU. Individual threads are identifiable by a unique index or *thread ID.* Kernels can be invoked through host code with parameters that specify the dimension of the thread hierarchy and how much shared memory should be allocated to each block [1].

3.2.2 Thread Hierarchy

There are three levels of thread abstraction in CUDA, these are:

- Thread - A single process that executes an instantiation of a kernel.
- Block - A 1, 2 or 3-dimensional collection of threads. Each thread can be referenced by its (x,y,z) value within the block as seen in Figure 3.2. Threads within the same block can co-operate through the use of shared memory.
- Grid - A 1 or 2-dimensional collection of blocks. Just like threads blocks can be referenced by an (x,y) dimension.

CUDA is sometimes referred to as a SIMT (Single Instruction Multiple Thread) architecture. GPU multiprocessors execute parallel threads in groups of 32 called *warps.* A *half-warp* is, as the name suggests one half of a warp. The significance of this lies in the control flow of threads executing in a warp. If one or more threads diverge due to a conditional statement all of the other threads in the warp are put on hold while the divergent threads serially execute their branched code until they rejoin the same execution path.

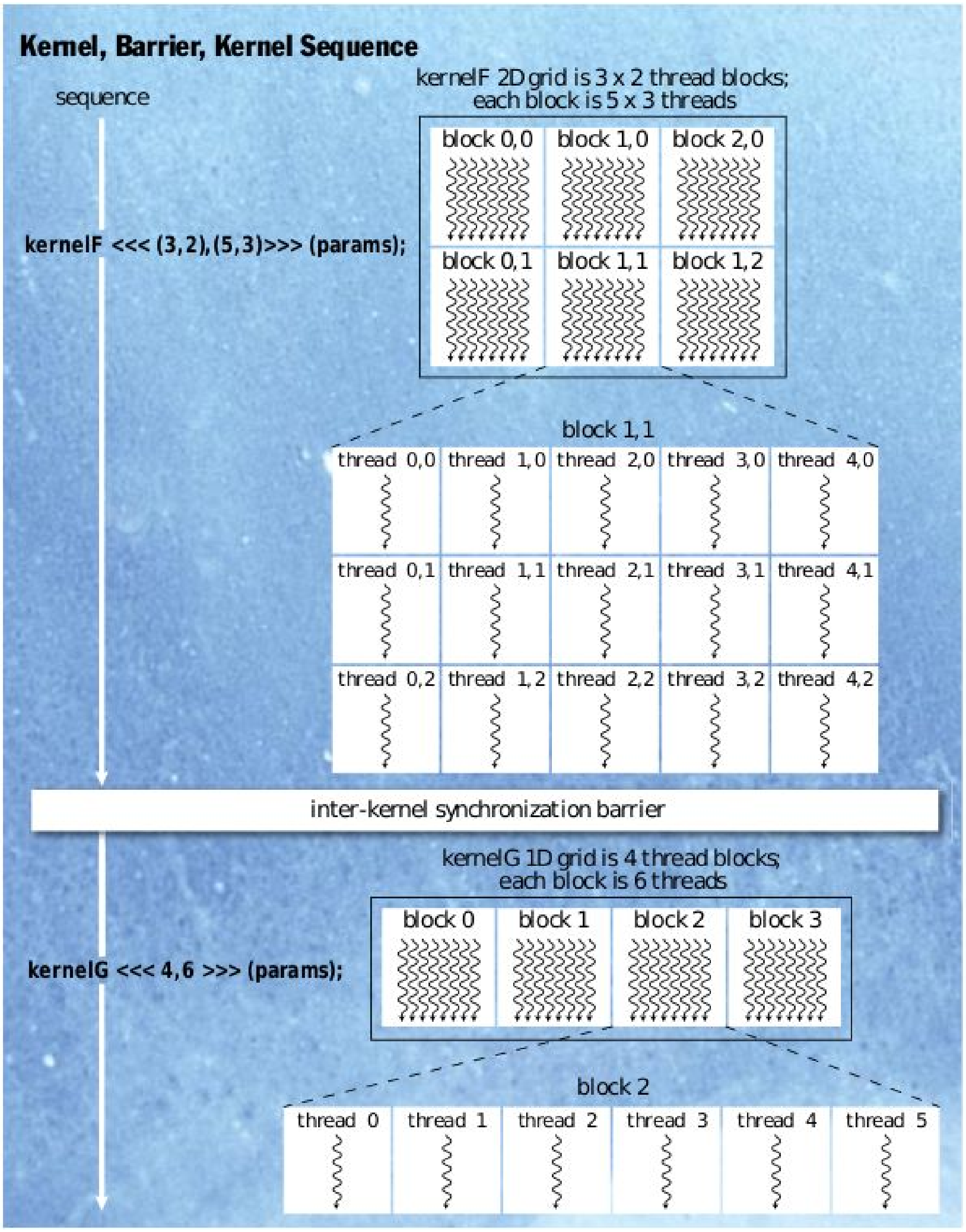

Figure 3.2: Shows an abstraction of the thread hierarchy and demonstrates how threads are assigned a unique ID.

Type	**Location**	**Hit Latency**	**Size**	**Scope**	**On-chip Cache**
Global	Off-chip	100s of cycles	768MB	Program	No
Shared	On-chip	100s of cycles	16KB per SM	Block	No
Local	Off-chip	1 cycle	Up to global	Block	No
Texture	Off-chip	100s of cycles	Up to global	Program	8KB per SM
Constant	Off-chip	1 cycle	64KB	Program	8KB per SM
Register	On-chip	1 cycle	32KB per SM	Thread	No

Table 3.1: Properties of the CUDA memory types. [7]

3.2.3 CUDA Memory

There are a number of distinct memory types in CUDA each with their own unique characteristics. A summary of these types and their specifications can be seen in Table 3.2.3.

Global memory is the largest memory type as shown in Figure 3.2.3 and is available to all threads irrespective of where they are being executed. The downside is that accesses to global memory are very expensive and should be minimised in order to optimise kernel performance. Shared memory is located on-chip and is accessible to threads within the same block at much less expense. Although it is limited in size its a good idea to store frequently accessed values in shared memory wherever possible.

Reading data stored in texture memory is referred to as texture *fetching* [1]. Texture fetches are cached and not subject to the same performance constraints on memory access patterns as global memory reads meaning they can exhibit higher memory bandwidth if there is locality in the texture fetches. Data can also be broadcast from a texture to multiple variables in a single operation further reducing read latencies. These attributes make textures best suited for use as read only memory and should be used for this purpose instead of global memory wherever possible.

3.3 Optimisation Strategies

Ensuring maximum memory bandwidth is essential for the peak performance of data intensive algorithms like the k-Nearest Neighbour Algorithm. One of the most obvious optimisations is to reduce latency by storing values as close to the processor as possible. However memory capacity decreases as the proximity to the processor increases. Identifying which values are accessed the most and storing them in the right memory is a key strategy for performance optimisation. [12, 5]

Coalescing memory reads is another important way of optimising performance. Memory reads by consecutive threads to consecutive memory locations can be batch executed as a single instruction. Ensuring that most global memory reads are coalesced can greatly increase memory bandwidth.

Reducing the number of bank conflicts in shared memory also optimises memory bandwidth. If two or more threads from the same half-warp access the same bank of shared memory at the same time it causes a bank conflict and serialises all the subsequent memory reads for the access. It is important to reduce the number of bank conflicts by ensuring that all threads in a half-warp access different banks concurrently.

CHAPTER 4

CUDA Implementation of kNN

In this section we will detail the implementation of the kNN algorithm in parallel using CUDA and outline our expectations for its performance. Firstly the original implementation provided by Garcia et. al. [4] will be explained and then we will move on to how our own modifications and kernels were incorporated into the overall system.

4.1 General

All code was written in the C programming language and compiled with the `nvcc` compiler for CUDA v1.0. Functions are implemented using the CUDA Runtime API as opposed to the Driver API. Tested kernels were invoked with 256 threads per block, this number was chosen due to the number of local variables per thread generally being too large too accommodate 512 threads per block (which is the maximum allowable). A few simple tests confirmed that invoking kernels with 512 threads per block resulted in consistently poor performance by comparison.

4.2 Data Representation on the Device

CUDA arrays are texture referenced arrays that are padded and aligned to conform to CUDA memory access patterns, allowing for optimal memory accesses[1]. They can be 1,2 or 3 dimensional and are only readable by kernels through texture fetching. In this kNN implementation data loaded onto the device is often stored as 2D CUDA arrays, also called 2D ‘pitched’ arrays. Values in these arrays cannot be referenced directly as they are offset by a ‘pitch’ or stride that is unique for each array and determined at the time it is allocated. For instance if we wanted to iterate over every element of the 2D pitched `dist` matrix we would do something like this:

```
for(int int i=0; i < COLS; i++){
  for(int j=0; j < ROWS*pitch; j+= pitch){
     element = *(dist + i + j);
  }
}
```

Memory for query points on the device is allocated as a 2D CUDA pitched array. Each 'column' of the array represents an individual query point. Each query point is a linear array of floats long enough to hold the query point itself plus the distances to each point of the reference array i.e. length = dimension + number of reference points. For the Bubble Sort implementation this was extended to accommodate the first k sorted distance values. Reference points, though stored as a texture for small training sets (discussed later), are most often stored as a 2D CUDA array in global memory.

4.3 Distance Calculations

The reference array is only ever read from during kNN execution and as such it's feasible to store the reference points as a read-only texture. Because texture fetches are cached and not subject to the same performance constraints on memory access patterns as global memory reads are it stands to reason that storing the reference array in a texture is advantageous. An added benefit is that texture co-ordinates are normalised due to the fact that the reference array is stored as a CUDA array thus making it easier to reference array elements [1]. At runtime the program checks whether or not there is enough space to hold the reference array in texture memory and if there is loads it as a texture, otherwise reference points are stored as a 2D CUDA array in global memory. As such there are two separate kernels for performing distance calculations depending on where the reference array is stored.

4.3.1 Reference Points in Global Memory

The distance calculation kernel loads sub matrices of the reference array into shared memory and performs pairwise differencing of elements with each query point then squares the result. These results are then added to the subtotal in shared memory. Once `__synchthreads()` has been called the final subtotal, which now represents the squared distance between points, is written to the distance array. Finding the square root of the subtotal to determine actual distances is not required before the sorting phase so it is left for later and only performed on the k smallest values. The kernel is invoked with a thread hierarchy mirroring the dimensions of the reference matrix such that each block can store a sub-matrix in shared memory and each thread can use its (x,y) index to easily determine which element to compute.

Below is pseudocode showing how distances are calculated when reference points are stored as a 2D CUDA array in global memory. The shared submatrices `shared_A` and `shared_B` correspond to the reference matrix and query matrix respectively. The parameters pA and pB refer to the matrix pitch, wA and wB refer with matrix width and dim is the dimension of the points.

```
__global__ void cuComputeDistanceGlobal( float* A, int wA,
             int pA, float* B, int wB, int pB, int dim,  float* AB){

// Obtain the 2D thread index
int tx = threadIdx.x;
int ty = threadIdx.y;

// Loop parameters
begin_A = BLOCK_DIM * blockIdx.y;
begin_B = BLOCK_DIM * blockIdx.x;
step_A  = BLOCK_DIM * pA;
step_B  = BLOCK_DIM * pB;
end_A   = begin_A + (dim-1) * pA;

// Conditions
int cond0 = (begin_A + tx < wA);
int cond1 = (begin_B + tx < wB);
int cond2 = (begin_A + ty < wA);

//Loop over all the sub-matrices of A and B
//required to compute the block sub-matrix
for (int a = begin_A, b = begin_B; a <= end_A;
             a += step_A, b += step_B) {

//Load the matrices from device memory
//to shared memory
if (a/pA + ty < dim){
 shared_A[ty][tx] = (cond0)? A[a + pA * ty + tx] : 0;
 shared_B[ty][tx] = (cond1)? B[b + pB * ty + tx] : 0;
}
else{
    shared_A[ty][tx] = 0;
    shared_B[ty][tx] = 0;
}

__syncthreads();

//Compute the difference between the two matrices
if (cond2 && cond1){
  for (int k = 0; k < BLOCK_DIM; ++k){
      tmp = shared_A[k][ty] - shared_B[k][tx];
      ssd += tmp*tmp;
  }
}

//Synchronize before next iteration
__syncthreads();

}

//Write the result
if (cond2 && cond1)
 AB[ (begin_A + ty) * pB + begin_B + tx ] = ssd;
}
```

4.3.2 Reference Points in Texture Memory

When reference points are loaded onto the device as a texture calculating the distances is trivial as nothing needs to be loaded to shared memory, threads don't need to be synchronized and elements of the reference array can easily be referenced by thread ID. Each thread calculates the difference between the dimensions of the query point and one of the reference points and the result is squared and added to the subtotal which in this instance can be stored as a local thread variable instead of having to be in shared memory.

Pseudocode for texture distance calculation:

```
__global__ void cuComputeDistanceTexture(int wA, float * B,
int wB, int pB, int dim, float* AB){
unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
unsigned int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
```

```
  if ( xIndex<wB && yIndex<wA ){
    float ssd = 0;
    for (int i=0; i<dim; i++){
      float tmp  = tex2D(texA, (float)yIndex, (float)i)
                            - B[ i * pB + xIndex ];
      ssd += tmp * tmp;
    }
    AB[yIndex * pB + xIndex] = ssd;
  }
}
```

4.4 Sorting

The distance computation kernels of the original implementation appeared to be already well optimised but for some reason the authors decided on using a serial Insertion Sort for distance sorting. As such the primary focus of this project was on implementing a truly parallel distance sorting algorithm suited for kNN with the expectation that the overall performance of the algorithm could be noticeably improved.

4.4.1 Original Insertion Sort

The original sorting function was implemented as a kernel, but the algorithm itself was a partially optimised serial insertion sort. In the implementation each thread invoked by the kernel handles sorting the distances corresponding to one query point in the query matrix. Naturally this is going to be quite efficient when there are many queries to be processed but for large training sets with few queries the performance of this part of the algorithm should degrade rather quickly. The sorting algorithm is modified to take advantage of the value of k to avoid having to perform sorting on every element of the array.

Pseudocode for the algorithm:

```
for (l=1;l<k;l++){
   v = *(p+l);
   if (v<max_value){
    i=0; while (i<l && *(p+i)<=v) i++;
    for (j=l;j>i;j--)
      *(p+j) = *(p+j-1);
    *(p+i) = v;
   }
   max_value = *(p+l);
}

//Sort only if value < max_value
for (l=k;l<height;l++){
   v = *(p+l);
   if (v<max_value){
     i=0; while (i<k && *(p+i)<=v) i++;
     for (j=(k-1);j>i;j--)
       *(p+j) = *(p+j-1);
     *(p+i) = v;
     max_value  = *(p+(k-1));
   }
}
```

The algorithm breaks the sorting down into two distinct parts. The first part properly sorts the first k elements of the distance array and the second part iterates over the rest of the distance array but only performs an insertion if it comes across a value that is less than the largest of the first k sorted elements. On average the amount of actual sorting that needs to be done is substantially reduced compared to an unoptimised implementation.

4.4.2 Parallel 'Bubble Sort'

This is our first implementation of a parallel sorting algorithm for kNN and is inspired by the simplicity of its namesake, Bubble sort, but is not a direct adaptation of the algorithm. The algorithm is implemented as a kernel dubbed `cuBubbleSort` and invoked as a 1-dimensional grid where each thread handles one element of the distance array. Threads iterate over the distance array to determine the 'rank' of their assigned element and then write the value to the index equaling the rank in the result array. Seeing as we only need the smallest k elements, and in the interest of reducing memory use, values are only written to the result array if their rank is less than k.

Determining the rank of an element is a simple process as can be seen in the pseudocode:

```
int rank = 0;
```

```
float v = *(dist + xIndex);
for(int i=0; i < dist_width; i++){
  elem = *(dist+i);
  if(elem < v || (elem == v && i < xIndex)){
++rank;
  }
}

if(rank < k)
  *(result+rank) = v;
```

As each thread iterates over the distance array it simply increments the rank if it comes across a value that is less than the element being compared. It takes into account duplicate values by incrementing rank only if the index of the duplicate is less than the thread index. This ensures duplicates keep their position relative to each other and prevents accidentally overwriting them.

It should be noted that `cuBubbleSort` makes no uses of shared memory, instead reading distance values directly from the array in global memory. This approach could therefore be considered somewhat naive as very little focus is placed on optimising memory reads.

4.4.3 Bitonic Sort

The original implementation of Bitonic sort is a recursive algorithm and as such couldn't be implemented in CUDA as it doesn't allow recursion. However further examining how the elements interact with each other allows the algorithm to be implemented in an iterative form. Elements can be paired and sorted into ascending and descending subsequences by pairing elements according to the binary value of their array index. The iterative form of Bitonic sort is perfect for parallelisation. The recursive calls of merge can be done in parallel. The loops in the merges, comparing and conditionally exchanging of elements can also be run in parallel [3].

Below is the pseudocode for the iterative form of Bitonic sort:

```
int i,j,k;
for (k=2;k<=N;k=2*k) {
  for (j=k>>1;j>0;j=j>>1) {
     for (i=0;i<N;i++) {
        int ixj=i^j;
```

```
          if ((ixj)>i) {
            if ((i&k)==0 && get(i)>get(ixj)) exchange(i,ixj);
            if ((i&k)!=0 && get(i)<get(ixj)) exchange(i,ixj);
          }
      }
   }
}
```

The CUDA implementation of Bitonic sort (distributed as an example project with the CUDA SDK) is inspired and made possible by this iterative implementation. The CUDA implementation is more or less identical to the iterative algorithm above with the only notable difference being that the innermost `for` loop is no longer needed as it is instead performed by N number threads in parallel. Array elements are loaded into shared memory and because there are no inter-thread data dependencies the distance array can be sorted in place without the need for a separate result array as is the case for `cuBubbleSort`. The kernel is invoked as a 1-dimensional grid where each block in the grid contains 256 threads and is allocated enough memory to store 256 floating point variables as is required by the algorithm.

4.4.4 Proposed CUDA Bubble Sort 2.0

Admittedly the original `cuBubbleSort` is a sub-optimal implementation and as such an enhanced implementation that makes effective use of shared memory is proposed. If values are loaded into shared memory they can only be accessed by threads within the same block, as such it would no longer be possible to iterate over the entire array. One possible solution to this would be to split the array into block sized sections and sort each section in parallel. Because we are only interested in obtaining the smallest k values we would then write the lowest k number of ranked values to an index in a temporary array offset by the block index.

In the case of k = 1 it would be trivial to perform another bubble sort on the temporary array to find the smallest k elements. For larger values of k it may be necessary to perform additional sorts in this way a number of times over. However as the value of k and the the number of reference points increases the performance of this method of 'distilling' the shortest k distances would degenerate quickly. A viable alternative would be to perform a merge sort on the first k elements of each sorted subsection. Though this may be slow for large values of k it is highly unlikely that this would cause problems in any practical implementation because choosing large values for k can cause erroneous object classifications and as such is undesirable.

CHAPTER 5

Results and Analysis

To evaluate the performance of our parallel sort implementations of the kNN algorithm a number of experiments were conducted on each implementation. To gain a point of reference the behaviour of the original implementation was evaluated first after which the CUDA Bubble Sort version was tested. Despite poor initial performance the results justified further optimisation and experimentation on the algorithm. Finally, an implementation that used Bitonic Sort was tested. These results will demonstrate the suitability of parallel sorting algorithms for use with this implementation of the kNN algorithm in CUDA and indicate what further optimisations may be made in future work.

The experiments were all run on a computer with the following specifications:

- Linux operating system (kernel 2.6.19.2).
- Intel 8400 Core 2 Duo CPU running at 3.0GHz.
- 3gb of addressable RAM
- 2x Nvidia GTX 260 GPU running in SLi each with 896MB of memory.

Each experiment was executed 10 times, and the results averaged. For clarity all relevant information will be presented here in graph form. Unless otherwise stated the following values are assumed to be fixed for each experiment:

- Number of queries = 128
- Number of reference points = 65536
- Dimension = 64
- Value of k = 128

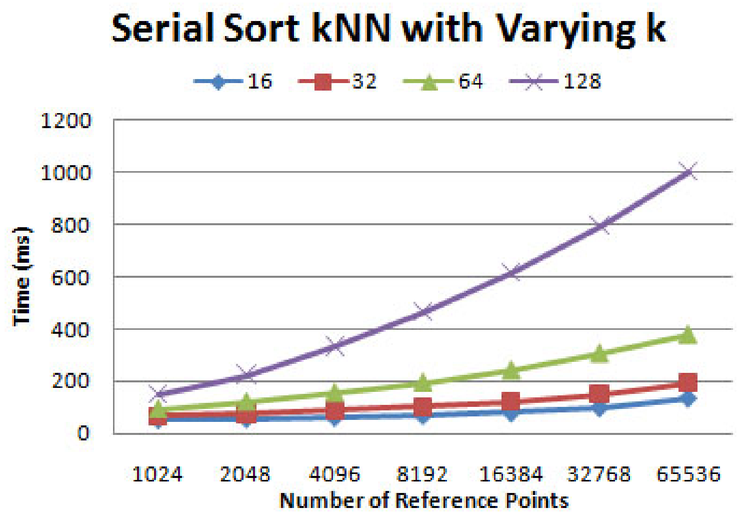

Figure 5.1: Time the original implementation takes to process queries for different values of k.

5.1 Original Implementation

Tests were run on the original kNN algorithm not just for comparison with our modified kNN implementation but also to determine any particular strengths or weaknesses that it had despite its use of a serial insertion sort algorithm. As expected the performance was unaffected by the number of query points being processed as the insertion sort kernel is written to sort the distances for each query point in its own dedicated thread. For large numbers of query points this compensated somewhat for the use of a serial sorting algorithm. As demonstrated in Figure 5.3 the performance of the serial insertion sort kNN slows linearly as the number of queries processed increases but not as much as the Bitonic kNN implementation indicating that for very large numbers of queries the serial algorithm may in fact perform better.

Another result worth noting was the original implementation's sensitivity to the value of k. As can be seen in Figure 5.4 the performance degrades rapidly as k increases, this is demonstrated in more detail in Figure 5.1. The reason for this is because the serial insertion sort kernel relies heavily on a small value of k to reduce the number of overall sorting operations performed. We would expect the performance to converge to a constant value as k approaches the size of the training set though this is of little interest because the classification of the kNN algorithm would be too inaccurate by this point.

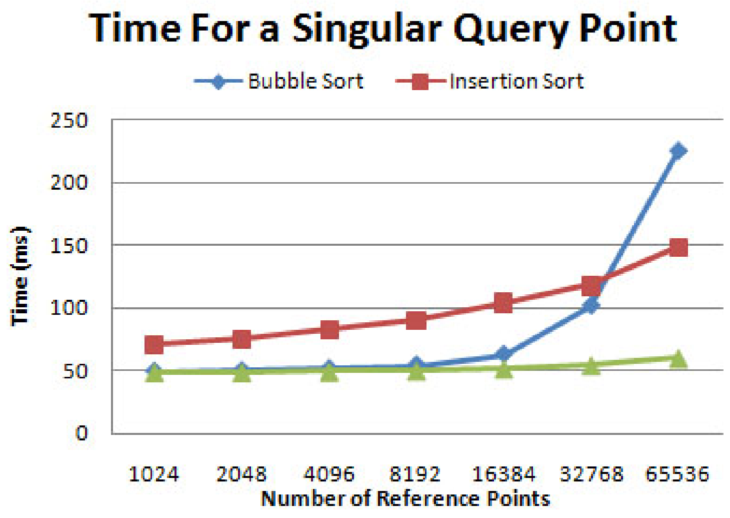

Figure 5.2: Time taken to process a single query point with a variable sized training set. The green line represents Bitonic Sort.

5.2 First Approach: CUDA Bubble Sort

Testing the performance of the parallel Bubble Sort implementation of kNN was a critical milestone in this project. The performance results of this implementation determined whether or not it was worth making further enhancements to the algorithm. Initial expectations were high for the success of this algorithm, though as will be shown, this enthusiasm was perhaps a little misplaced. Lessons learnt from the observed behaviour of `cuBubbleSort` provided valuable information on the effect of inefficient memory accesses on kernel performance.

Although implemented later, the kNN with Bitonic Sort performance is included in Figure 5.2 to act as a point of reference for what sort of performance should be expected from a properly optimised parallel sorting algorithm in CUDA.

5.2.1 Discussion

The performance of CUDA Bubble Sort is comparable to Bitonic sort for small training sets but as the number of reference points increases it appears to degrade exponentially. This phenomenon was the cause of some initial confusion. Code isolation brought about the first theory for this behaviour which was that it occurred when writing the sorted values from each thread to their designated positions in the result array. However this didn't seem to make sense as the number of writes being done by threads was the same as the value of k which was too low to explain such a dramatic performance hit.

A test kernel was written that simply read a value from an array in global memory and then wrote it to its corresponding index in the result array. Tests run on this kernel confirmed that performing this kind of simple read and write operation in parallel didn't have any adverse affect on performance and revealed that the problem was being caused by the part of the algorithm reading values from global memory. The reason this had been passed over initially is due to the fact that removing the write operation in the `cuBubbleSort` kernel seemed to fix the issue. What was later learned is that this was most likely an optimisation by the `nvcc` compiler. Because the kernel wasn't writing any of its output it was being allowed to continue running independently on the GPU without delaying the rest of the execution. The down side to this was that repeated execution of the algorithm resulted in the device on the test machine being flooded with kernel invocations, filling the memory and in the end causing the system to hang and require a restart.

After further research and an extensive process of elimination it was eventually determined that the poor performance was being caused by excessive non-coalesced memory reads. Seen in Figure 5.2 performance is reasonable and then suddenly rises, this apparent exponential increase can be interpreted as a response to excessive unoptimised and unaligned global memory accesses. Every thread was simultaneously reading values from the same unsorted array in global memory with the consequence being that `cuBubbleSort` was being effectively serialised for large training sets. For the sake of curiosity `cuBubbleSort` was tested on a machine running an Nvidia 8800GTX graphics card to see if there was any difference in behaviour. What was observed was a disproportionately large drop in performance whereby the algorithm took sometimes hundreds of times longer overall than the serial insertion sort implementation. After taking into account the difference in compute capability and memory between the Nvidia 260GTX and 8800GTX it is unclear how this behaviour should be interpreted. One reason may be that it demonstrates a difference in the way the GPU architectures handle memory accesses.

While these results may appear to be 'the final nail in the coffin' for `cuBubbleSort` it may yet be possible to salvage some of the algorithm. One proposal is to split the distance array into block sized sub arrays, copy them to shared memory, `cuBubbleSort` sort each of these sub arrays and then perform a merge sort on the first k elements of each sub array. Only having to deal with the first k elements in this way would considerably reduce the amount of sorting that needed

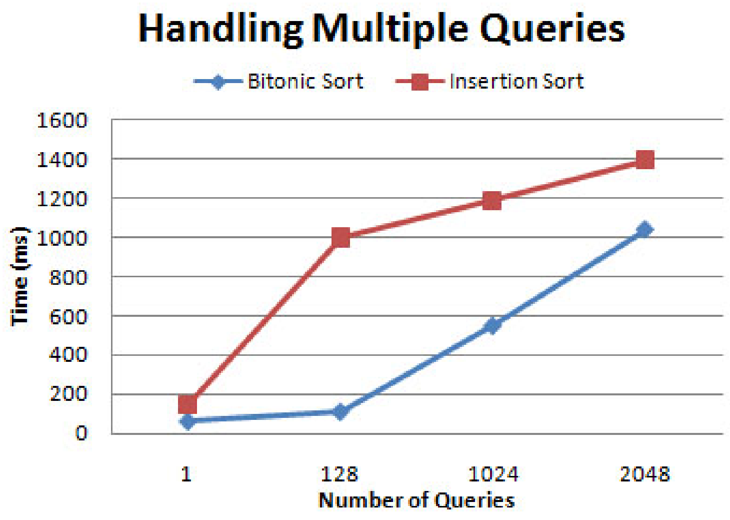

Figure 5.3: Time taken to process multiple queries.

to be done. Assuming the algorithm didn't suffer from a similar problem to before and there were minimal bank conflicts the performance could even be better than Bitonic Sort for small values of k making it well suited for use with the kNN algorithm.

5.3 Second Approach: CUDA Bitonic Sort

After the disappointment of our first approach it became all the more important to demonstrate an improvement of the kNN algorithm using a parallel sort. This spurred the adaptation of CUDA Bitonic Sort for use with kNN. Although Bitonic Sort is not able to exploit the value of k to its advantage like the serial insertion sort and the proposed `cuBubbleSort2` these results do however show that respectable performance gains can be made by using a truly parallel sorting kernel.

5.3.1 Discussion

The times taken to process a single query point shown in Figure 5.2 above demonstrate that kNN with Bitonic Sort has superior performance despite not exploiting the value of k to reduce the amount of distance sorting required. However not making use of k can be advantageous as its value increases; as shown in Figure

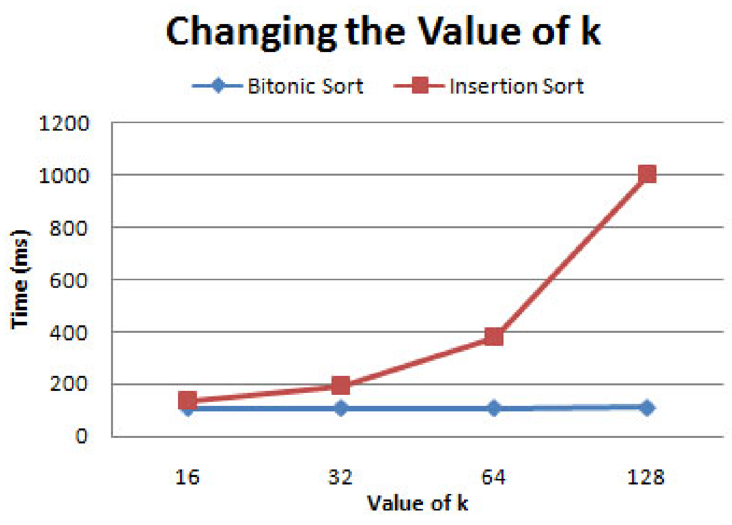

Figure 5.4: Time taken to process queries for varying values of k.

5.4 the performance of the insertion sort implementation quickly degrades while Bitonic Sort is unaffected.

Increasing the number of simultaneous queries being processed exposes one possible weakness of using a parallel sorting kernel in this kNN implementation. Seen in Figure 5.3 the performance degrades in a linear fashion as the number of queries increases just as it does with insertion sort but at a greater rate. The reason for this behaviour is most likely due to sub-optimal GPU utilisation. Queries are fed to the Bitonic Sorting kernel one at a time unlike the insertion sort kernel which handles all of the queries at once. It can be extrapolated that the Bitonic Sort implementation of kNN may not perform as well as the insertion sort implementation when processing a large number of queries in proportion to the size of the training set. This issue may be overcome by optimising the kernel invocation to make better use of available resources or by invoking multiple kernels for different queries when there are sufficient resources.

5.4 Miscellaneous

For the sake of confirmation experiments were run to determine what effect the dimension of points had on each implementation. Due to the fact that the sorting is done to pre-calculated distances with no reliance on the dimension of points it was no surprise to see that dimensionality had no bearing on the performance of

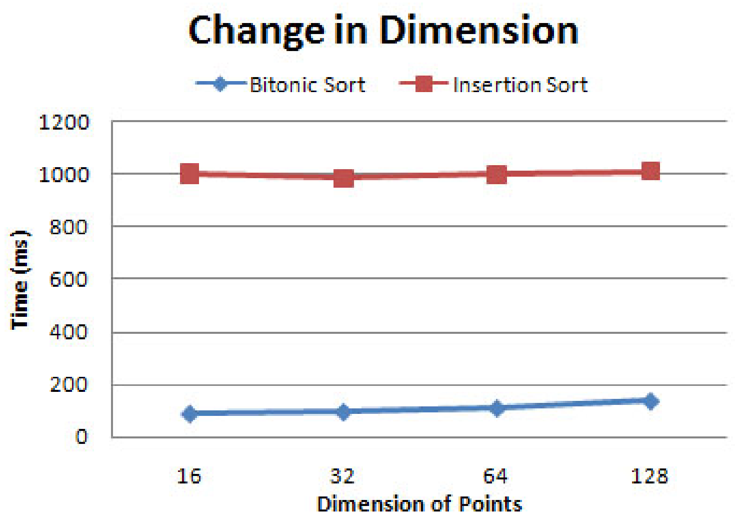

Figure 5.5: Time taken to process points of varying dimension.

the sorting algorithms themselves. What did become apparent was the effect of increases in dimension on the overall performance of each implementation. Aside from the extra calculations required with higher dimensions it also has the effect of limiting performance due to memory constraints just as the number of reference points does. When the size of the 2D reference array (training set) becomes too large to reside in texture memory we notice a slight drop in performance possibly due to extra global memory reads as seen in Figure 5.2 and 5.5.

CHAPTER 6

Conclusion

We have been able to demonstrate a respectable performance improvement to a parallel implementation of the k-Nearest Neighbour algorithm in CUDA. This was achieved by implementing a fully parallelised sorting algorithm as a substitute for the previously used serial sorting algorithm. Additionally this project has highlighted some of the difficulties of implementing certain algorithms while also taking into account the constraints imposed by the necessity of efficient memory accesses. The consequences of not fully satisfying these requirements can have a devastating impact on kernel performance, in the worst case resulting in the serialisation of thread execution as was observed with the original 'Bubble sort' kernel.

6.1 Future Work

Proper implementation of the aforementioned CUDA Bubble sort 2.0 as a combination bubble and merge sort may be worth exploring in future work. It is doubtful that even a well optimised implementation would outperform Bitonic sort in general but the fact that the merge phase theoretically performs better for smaller values of k means that it may scale well and prove to be a more suitable way of sorting distances for the kNN algorithm.

Bibliography

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[2] Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., and Wu, A. Y. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM 45*, 6 (1998), 891–923.

[3] Christopher, T. Bitonic sort. http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm. [Online; accessed 30-September-2009].

[4] Garcia, V., Debreuve, E., and Barlaud, M. Fast k nearest neighbor search using gpu. *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on* (June 2008), 1–6.

[5] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V. Parallel computing experiences with cuda. *Micro, IEEE 28*, 4 (2008), 13–27.

[6] Metrics, I. M. E. C. Nvidia geforce gtx 200 gpu datasheet. http://www.intel.com/support/processors/sb/cs-023143.htm. [Online; accessed 30-September-2009].

[7] Neumann, A. Parallel reduction of multidimensional arrays for supporting online analytical processing (olap) on a graphics processing unit (gpu).

[8] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda. *Queue 6*, 2 (2008), 40–53.

[9] Nvidia. Getting started with CUDA. http://developer.download.nvidia.com/ compute/cuda/2_1/toolkit/docs/CUDA_Getting_Started_2.1_Linux.pdf. [Online; accessed 10-March-2009].

[10] Nvidia. Nvidia corporation. http://www.nvidia.com. [Online; accessed 30-September-2009].

[11] Nvidia. Nvidia geforce gtx 200 gpu datasheet. http://www.nvidia.com/docs/IO/55506/GPU_Datasheet.pdf. [Online; accessed 30-September-2009].

[12] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), ACM, pp. 73–82.

[13] Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), pp. 1–10.

[14] Wu, X., Kumar, V., Ross Quinlan, J., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G. J., Ng, A., Liu, B., Yu, P. S., Zhou, Z.-H., Steinbach, M., Hand, D. J., and Steinberg, D. Top 10 algorithms in data mining. *Knowl. Inf. Syst. 14*, 1 (2007), 1–37.

[15] Zhou, K., Hou, Q., Wang, R., and Guo, B. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5 (2008), 1–11.

APPENDIX A

Original Honours Proposal

Title:	Improving the k-Nearest Neighbour Algorithm with CUDA
Author:	Graham Nolan
Supervisor:	Associate Professor Amitava Datta

Background

Data classification is an ever expanding field of research in computer science. Accurate classification techniques are of vital importance to areas such as biological and market research, medical imaging, machine learning and various data mining applications. As technology in these areas improves the amount of data that needs to be processed grows with it, hence creating the demand for faster and more efficient (not to mention affordable) ways to classify information.

There are many different algorithms available for classifying objects in data, the most simple of these try to compare objects to those stored within a pre-classified set of data or training set. While these algorithms are easy to implement they run into trouble when processing highly variable and multi-dimensional data as they can only classify objects that have an exact match within the training set [14].

The k-Nearest Neighbour algorithm (KNN) is an enhancement to this method that does not require an exact match to classify an object. The algorithm first identifies a group of k objects (neighbours) from the training set that are closest to the object in question and then assigns it a classification based on the predominance of a certain class within the neighbourhood and their relative distance from the object [14].

In order to determine which objects are neighbours and their significance a distance metric is required. The distance between two objects can be determined from any or all of an objects attributes, for instance if classifying a set of data representing people we could determine distance based on a function of height, weight and income. It may be necessary to scale one or more variables to avoid one becoming too dominant for example the height of a person varies from 1.5 to 1.8 m, the weight of a person varies from 45 to 150kg, and the income of a person varies from $10,000 to $1,000,000. Clearly without scaling income would have too much influence over the distance measurement [14].

While building a training set for a KNN classifier is inexpensive the classification of an object is relatively expensive because it requires the computation of its k-nearest neighbours. This in turn requires computing the distance between all the objects in the training set and the unclassified object, which can become very expensive particularly for large training sets.

There exists a number of methods that can improve the efficiency of the kNN algorithm such as eliminating objects from the training set without that won't effect the classification accuracy (this is known as 'condensing') [14]. While reducing the number of calculations in this way can improve the performance of kNN it can still be very computationally intensive and is less effective on high-dimensional data. Recently there has been some promising research by *Garcia et al.* [4] into improving the efficiency of kNN with parallel processing that has demonstrated potentially significant speedups. Based on these results is appears that modifying kNN to performs some computation in parallel is worth exploring.

The recent release of CUDA by Nvidia corporation presents the opportunity to perform parallel computing with relative simplicity and minimal expense. The CUDA architecture gives software developers the capability to perform general purpose parallel computation on commercially available GPU's that until recently were only capable of graphics processing. CUDA-capable GPUs have hundreds of cores that can collectively run thousands of computing threads. Each core has shared resources, including registers and memory [9, 8, 12]. The on-chip shared memory allows parallel tasks running on the GPU to share information without having to communicate with main memory. The CUDA architecture also supports heterogeneous computation where applications use both the CPU and GPU [9], this makes it highly suitable for adapting algorithms such as kNN as only those parts that would benefit from parallel computing can be run on the GPU whilst the less intensive parts can be run concurrently on the CPU without contention for memory resources.

Aim

The aim of this project will be to modify the k-Nearest Neighbour algorithm to run using CUDA. It is hypothesized that by enabling parts of the algorithm to run in parallel on a GPU that a significant speedup can be achieved when compared with the serial implementation, particularly for computationally intensive stages such as object distance calculation and result sorting.

Initially only the 'brute force' algorithm will be implemented however there may be scope to implement some of the known enhancements to the kNN algorithm such as condensing to determine any additional benefit that can be obtained when used in conjunction with CUDA.

Method

Initial research will involve implementing a basic serial version of the kNN algorithm to run on a CPU. An application for generating different sets of n-dimensional data for use as training sets will also need to be implemented along with sets of unclassified data. Additionally a suitable metric for measuring the distance between objects for each set of data will need to be determined.

The next step will be to modify the algorithm to compute object distances in parallel on the GPU using CUDA. Following this, other tasks performed by the algorithm will be modified to use CUDA such as the sorting of object distances. The performance of the parallel algorithm will then be measured and compared to that of the serial one for multiple sets of data of varying dimension.

Below is an approximate schedule for the project:

Task(s)	To be completed
Perform research and learn CUDA	Mid Mar. – End Apr. '09
Present proposal talk to research group	23 – 27 Apr. '09
Begin implementation	May '09
Submit revised proposal	21 May '09
Performance testing and code modification	June '09
Begin dissertation	July '09
Submit draft dissertation	10 Sept. '09
Correct dissertation, prepare talk and poster	Mid Sept. – Oct. '09
Submit final dissertation	15 Oct. '09
Present talk and submit poster	19 – 23 Oct. '09
Correct and resubmit dissertation	23 Oct. – 16 Nov '09

Software and Hardware Requirements

Code for the project will be written in the C programming language and compiled, run and tested on a CUDA enabled Linux system with an nVidia 8800GTX GPU provided by the UWA Computer Science department. Documentation will be written and compiled using LaTeX.