Gutierrez PD, Lastra M, Bacardit J, Benitez JM, Herrera F. GPU-SME-*k*NN: Scalable and Memory Efficient *k*NN and Lazy Learning using GPUs. *Information Sciences* 2016. DOI: 10.1016/j.ins.2016.08.089

GPU-SME-kNN: Scalable and Memory Efficient kNN and Lazy Learning using GPUs

Pablo D. Gutiérrez[a,*], Miguel Lastra[b], Jaume Bacardit[c], José M. Benítez[a], Francisco Herrera[a]

[a] *Depto de Ciencias de la Computación e Inteligencia Artificial. E.T.S. Ingeniería Informática y Telecomunicación. CITIC-UGR. Universidad de Granada. Granada. Spain*
[b] *Depto. Lenguajes y Sistemas Informáticos. E.T.S. Ingeniería Informática y Telecomunicación. CITIC-UGR. Universidad de Granada. Granada. Spain*
[c] *Interdisciplinary Computing and Complex BioSystems (ICOS) Research Group, School of Computing Science , Newcastle University, Newcastle upon Tyne, Tyne and Wear, United Kingdom*

Abstract

The k nearest neighbor (kNN) rule is one of the most used techniques in data mining and pattern recognition due to its simplicity and low identification error. However, the computational effort it requires is directly related to the dataset sizes, hence delivering a poor performance on large datasets. The use of graphics processing units (GPU) has improved the run-time performance of the kNN rule but the computational requirements of current approaches limit this performance as the dataset size increases.

In this paper, we propose a new scalable and memory efficient design for a GPU-based kNN rule, called GPU-SME-kNN, that breaks the dependency between dataset size and memory footprint while delivering high performance. An experimental study of GPU-SME-kNN is presented showing a high performance, even in cases that other methods cannot address, while the computational requirements are suitable for most commercial GPU devices. Our design has also been applied to kNN-based lazy learning algorithms reducing run-times in a significant way.

Keywords: kNN, GPU, CUDA.

*Corresponding author
Email address: pdgp@decsai.ugr.es (Pablo D. Gutiérrez)

1. Introduction

The k nearest neighbor rule (kNN) [9] [28] is one of the most used data mining and pattern recognition techniques. The simplicity and low identification error of this rule makes it the reference tool to test classifiers and datasets [11]. It has been considered one of the top 10 algorithms of data mining [33]. The kNN rule is also the base of several classifiers that belong to the lazy learning family of classifiers [3].

The kNN rule is based on the idea that an unknown instance will be similar to other instances that are close to it in the space of characteristics. Although this idea is simple its computation requires a large amount of operations that increases with the dataset sizes, in terms of both attributes and instances. When addressing large problems, the time required to compute the results makes the kNN rule virtually unusable. The lazy learning algorithms that are based on the kNN rule suffer the same issues.

Currently, several real-world applications introduce scalability challenges which must be overcome by data mining techniques [26]. Given that many real world applications routinely produce massive amounts of data it is absolutely necessary to tackle the scalability challenges of the kNN rule, if this technique is going to be applied to such datasets.

Graphics processing units (GPU) have proven to be useful in managing large amounts of data efficiently in different situations like fingerprint identification [21], continuous optimization [22] bioinformatics [27] and data mining [8]. Moreover, the kNN rule has been successfully adapted to run on GPU devices to improve its run-time performance [14] [5] [19].

GPU devices provide massive parallelism that can potentially reduce the run-time of computationally intensive tasks. On the other hand, these devices have a limited amount of memory. Most approaches that tackle large datasets reduce their performance when there is not enough memory to allocate the complete datasets and kNN structures on the GPU device. Furthermore, some

of these methods just do not work if the dataset is too large. Most approaches also require sorting all the distance values or use suboptimal methods in order to locate the neighborhood, not taking advantages of all the possibilities that the GPU devices offer.

In the literature, some authors have tried to overcome these limitations. Arefin et al. [5] reduce the usage of memory dividing the computations in square-shaped portions, but the data structures required still limit the run-time performance. Komarov et al. [19] propose a quicksort-based selection method in order to improve the performance of that part of the kNN rule, but their design requires a high amount of synchronization operations that hinders the run-time performance obtained.

In this paper, we propose a design of the kNN rule, called GPU-based scalable and memory efficient kNN (GPU-SME-kNN), which addresses the aforementioned issues. To do so, we introduce two novel approaches:

- An incremental neighborhood computation scheme that eliminates the dependencies between dataset size and memory footprint. This scheme allows fully customization of its parameters and takes advantage of asynchronous memory transfers, making the required structures fit into the available memory for a broad range of GPU devices while delivering high run-time performance independently of the number of instances of the dataset. This is detailed in Sections 4.1 and 4.2.1.

- An efficient quicksort-based selection design that avoids synchronization operations and has an enhanced pivot-selection method to provide a high performance and scalable solution. This is detailed in Section 4.2.2.

GPU-SME-kNN has been tested with two large datasets from the UCI [1] repository [6]: Poker, with 1 025 009 instances and KDDCup 1999 with 4 898 431 instances. Increasing datasets sizes (up to the full size) and different values of the k parameter have been used in order to thoroughly study the behavior of

[1] http://archive.ics.uci.edu/ml

the GPU-based kNN rule in terms of scalability. The results focus on the runtime performance and the memory requirements of the method. Given that our algorithm is designed to improve the efficiency of the kNN rule, but it does not change its behavior, it is not necessary to evaluate its predictive capacity. GPU-SME-kNN is compared with well-known GPU-based kNN approaches showing a good performance.

The rest of the paper is organized as follows: Section 2 presents the kNN rule and the lazy learning family algorithms. Section 3 introduces how GPU devices work and summarizes the previous approaches of GPU-based kNN rule. Section 4 explains our design for the kNN rule. Section 5 shows the results obtained on the experiments performed. Section 6 studies design modifications that our algorithm requires to be applied to lazy learning algorithms and the results obtained. Finally, Section 7 presents the conclusions.

2. The k Nearest Neighbor rule and Lazy Learning

This section summarizes the main aspects of the k Nearest Neighbor rule and family of lazy learning algorithms that will be referred to in the design of the GPU-based method. Section 2.1 explains the kNN rule and Section 2.2 presents the lazy learning algorithms characteristics.

2.1. The k Nearest Neighbor rule

The k Nearest Neighbor rule (kNN) [9] predicts the class of a test instance as the majority class of the k training instances that have the smallest distances to that test instance [10] . This means that each test instance is compared with every training instance, measuring the distance between them. These distances are checked in order to find the k smallest values and the training instances that correspond to the selected distances are used to predict the class of the test instance.

Although different distance measures can be used [32] [24] [30] , the well-

known Euclidean distance is typically used:

$$d(x, y) = \sqrt{\sum_{i=1}^{D}(x_i - y_i)^2} \quad (1)$$

where $d(x, y)$ is the distance between instances x and y and D is the number of attributes of the problem.

The kNN rule is usually applied to a set of test instances. If M is the number of test instances and N the number of training instances, the algorithm requires $M \times N$ distance computations and M selections of k instances from an array of N elements. When training and test set sizes increase, the distance computations increase quadratically. The number of selection operations increases linearly with the value of M and the computational cost of each operation increases with the size of N in a way that depends on the specific selection method used but which is, at least, linear. This increase of the number of operations makes the application of this rule really difficult for large datasets.

2.2. Lazy learning

Typically, the process of learning from data involves the generation of some kind of model, from the training data. This model is used to classify the instances of the test set. The family of lazy learners [3] skips this general model creation. When a test instance is evaluated, these algorithms compute a specific model that relates that test instance with the training set and use that model to classify it.

Lazy learning algorithms usually have greater storage requirements and high computational cost when evaluating a test instance than other algorithms. Algorithms that build a model can discard the training instances once the model is built reducing the memory requirements. Moreover, the evaluation of a model is almost inexpensive compared to the cost of building that model, reducing the computational cost for non lazy learning algorithms. The impact of these two issues increases with the dataset size.

This behavior can be observed on the kNN rule because it belongs to the

family of lazy learners [13]. There are several lazy learning algorithms that are based on the kNN rule and have the same computational issues.

3. Graphics Processing Units and NVIDIA CUDA

Graphics processing units (GPU) were originally created to offload the computations related to 3D graphics on games and design applications from the CPU device into specialized hardware. Modern GPU devices provide a specific processor with a Single Instruction Multiple Data (SIMD) architecture to handle these computations efficiently. NVIDIA CUDA [1] is a hardware/software architecture that allows the use of NVIDIA [2] GPU devices for general purpose programming.

CUDA presents GPU devices as parallel coprocessors with their own memory, caches and registers that can cooperate with one or several CPU cores. In order to take advantage of the characteristics of GPU devices, it is required to redesign the algorithms, determine which operations of the algorithm match the characteristics of each device and the amount of data required to be transferred between devices. To produce efficient GPU based programs or systems, it is mandatory to know some technical aspects of GPU devices (Section 3.1). Once these aspects have been studied we will briefly review the approaches to the kNN rule that can be found in the literature (Section 3.2).

3.1. Technical aspects of GPU devices

Functions are run on GPU devices dividing the workload into a set of threads which share the same code but operate on different data. These functions are called kernels and the set of threads of each kernel is called grid. Threads within a grid are grouped into blocks of threads.

At the hardware level, a GPU device has a set of computing cores that are grouped into stream multiprocessors (SMX). When a grid is run on the GPU, each block is assigned to one SMX, as shown in Figure 1. It is possible to synchronize all threads that belong to a block. However, there is no efficient synchronization method for threads in different blocks.

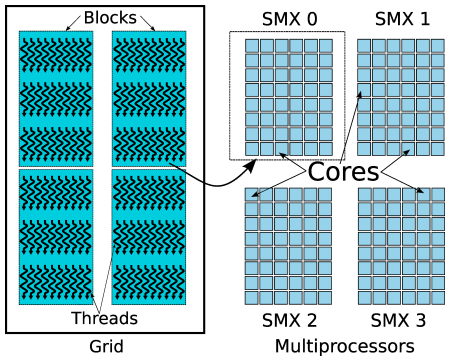

Figure 1: Threads, blocks and multiprocessors. Each block is run on the same multiprocessor.

Each block is divided into groups of 32 threads called warps, which are run synchronously on a SMX. All threads within a warp execute the same instruction (in a parallel way) at the same moment. In case of code divergence within the warp, like a conditional instruction with different results, the execution is serialized penalizing the run-time performance.

Regarding data storage and access, as in CPU devices, a GPU device has a memory hierarchy from large and slow memory banks to small and fast registers including several cache levels. The fastest cache (L1) is available to developers on demand. Each SMX has its own L1 cache but its size is limited. The L1 cache is commonly known as shared memory.

Global memory is the last level of the hierarchy, and it is the largest memory area of a GPU device but also the slowest. The most efficient way for a large number of threads to simultaneously request data from global memory is to perform these parallel requests in a coalescent way: consecutive threads in a block have to request consecutive memory positions at the same time.

In order to start the computation, the input data used by the kernels needs to be copied from the computer main memory to the GPU device global memory. This memory transfer can be done either synchronously or asynchronously. By using asynchronous copies it is possible to run a kernel while copying data that will be required in subsequent kernel calls.

When a kernel function is called, the programmer has to set the number of

threads per block, blocks per grid and shared memory required by the kernel. The maximum number of simultaneous blocks per SMX and warps per SMX is device dependent. The resources required by each block limit the number blocks that can be run in parallel.

3.2. GPU-based approaches to kNN

This section presents a brief summary of the most relevant proposed GPU-based methods that tackle the computational issues of the kNN rule. All these approaches divide the kNN rule into two parts: the computation of the distances and the identification of the nearest neighbors. The differences among them rely on how these steps are solved.

As the distances are computed in a separate stage from the selection, a distance matrix is built grouping the distance array related to each test instance. In the second step, several selections are performed in parallel on the different rows of the matrix.

Kuang et al.[20] propose to compute one distance per thread for the distance matrix calculation and sort the distances array of each test instance to get the k nearest neighbors. The distance matrix is split into blocks of a predefined number of threads that compute a distance operation. Each matrix row is sorted using a radix sort method that is computed by a block of threads.

Garcia et al.[14] use the previous distance matrix calculation scheme but they use an insertion sort method instead of radix sort. Both approaches also differ in the way the sorting is done. Garcia et al. compute one sort operation per thread instead of one per block. The code of this implementation of the kNN rule is available on-line and is used as reference for comparisons in other work [17] [29].

Kato et al.[18] propose a design that is also suitable for several GPU devices. The distance matrix is split into blocks of rows where each thread computes the distances for a matrix row. The selection method is performed with one block per test instance. The neighborhood is built in shared memory using an insertion approach.

GPU-FS-kNN, presented by Arefin et al.[5], divides the computation of the distance matrix into squared chunks in both dimensions. Each chunk is computed using a different kernel call, reusing the allocated GPU-memory. The distance computation kernel divides the chunk into smaller square subsets, one per block. Training and test data of each square subset are copied to shared memory and then each thread computes a Pearson distance. A selection step is performed after each chunk is processed with a modified version of the insertion sort technique. This process is performed using one thread per chunk row. The neighborhood computed in a previous chunk is reused for the next chunks that correspond to the same test instances. The code of this algorithm of the kNN rule is available on-line.

Jian et al.[17] use the same approach as Kuang et al. and Garcia et al. to compute the distance matrix. For the selection step, they propose a method that uses several blocks per test instance. Each block selects k distances and then the results of all blocks are combined iteratively.

Komarov et al.[19] modify the selection step with a quicksort-based selection. Each block performs a selection operation with a large number of threads per block. The matrix computation uses the Kuang et al. and Garcia et al. scheme. The authors of this method have not made its code available. In order to compare our method to it, we have re-implemented this approach. The source code is available with the rest of the code related to this work at: `http://sci2s.ugr.es/GPU-SME-kNN/`

There are other approaches in the literature related to the kNN rule issues. Some of them use FPGA devices [25] but they do not outperform GPU-based techniques in run-time. Only when power consumption is considered these devices become an interesting option. There are also kNN versions for specific problems, like text classification [16], but the optimizations performed are focused and dependent on the specific problem or the distance measure used and cannot be applied to other situations. Some pieces of work consider the resolution of only one test query at a time[7] but this approach is not suitable for cases where a large number of test queries is available, like in Lazy Learning

algorithms.

The majority of these approaches assume that the distance matrix and the rest of the data structures fit on GPU memory but this is not possible for large datasets, like the KDDCup 1999 dataset. The solution provided by Garcia et al. for this issue is to divide the test set into parts and compute these parts iteratively but without considering the use of asynchronous memory copies to avoid idle times. Furthermore, this approach affects the run-time performance of the different steps of the kNN rule and it is not suitable for the constantly increasing sizes of the datasets.

Arefin et al.'s algorithm is the exception in terms of memory requirements assumptions, as the computation is performed chunk-wise. As the authors expose in their paper, the memory footprint is reduced because some structures are reused during the computation of the algorithm. However, this method is still limited by the memory requirements because the complete dataset (training and test sets) is copied to GPU memory. In that case, a smaller chunk size can be used but this reduces the run-time performance.

Our design (Section 4) tackles the memory related issues present in the literature, overcoming the dependence between dataset size and memory required. Nevertheless, efficient kernels have been designed for each step of the computation which improves the run-time performance.

4. A GPU-based kNN rule for large datasets

There are two main issues inherent to large datasets processing: the computational complexity, in terms of amount of operations, and the memory required to store the structures of the algorithm. Our method introduces an incremental neighborhood scheme to reduce the memory requirements. This scheme has to be combined with an efficient memory transfer design between devices. These points are addressed in Section 4.1.

The design of the kernel functions, Section 4.2, has a high impact on the run-time performance achieved. We split the distance computation into two

different kernel functions: the first one computes the square of the distance and the second one performs the square root operation, but only on the selected neighbors. For the selection step, we propose a quicksort [15] based selection that takes advantage of the iterative neighborhood computation characteristics.

Section 4.3 presents how memory requirements of GPU-SME-kNN are determined by the different algorithm parameters regardless of the dataset size.

4.1. CPU-GPU interaction model

The main steps of the kNN rule are the ones related to the computation of the distance matrix and the selection of the k nearest neighbors.

GPU devices have a small amount of memory compared to desktop and server computers. Some structures, like the distance matrix, do not fit into device memory. The solution to this problem is to split the computations in steps and take advantage of the asynchronous memory copy operations to avoid idle times.

Our method splits the computations of the distance matrix and neighborhood. The method of Arefin et al. [5] also splits these computations, however, this technique requires to compute square-shaped portions of the matrix and to copy the complete training and test sets to device memory. Depending on the GPU device and the dataset used, the size of the matrix portion may need to be reduced, leading to a loss of run-time performance. Our method overcomes these limitations providing a more customizable scheme that can be adapted to almost every GPU device. The details are discussed in the next section.

4.1.1. Incremental neighborhood computation

The distance matrix is the most memory demanding data structure needed in the kNN rule. The size of this matrix is $M \times N$, where M is the number of test instances and N the number of training instances.

The most common solution in order to make the distance matrix fit into memory is to divide it into strips. The distance matrix is split into M/m matrices of size $m \times N$, where m is a portion of the test set small enough to

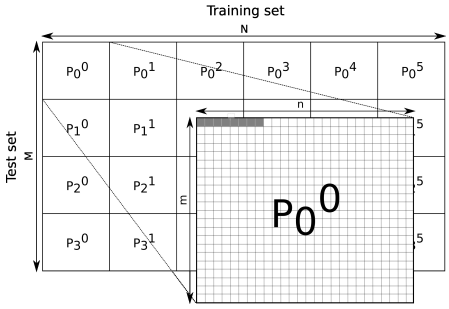

Figure 2: The distance matrix of size $M \times N$ is partitioned in pieces of $m \times n$ elements.

make the matrix fit into device memory. The algorithm iterates through the test set to complete the computation. This is the solution used by Garcia et al. [14].

However, this solution is limited by the size of the training set because the N instances of the training set and m instances of the test set need to be kept on device memory in order to perform the distance computation. In some scenarios, the GPU device could not have enough memory even when setting m to 1.

Our design is based on the algorithm of Arefin et al. [5] where the matrix is split in both dimensions. As shown in Figure 2, a portion, P_i^j, of the matrix of size $m \times n$ is computed on each step, where m and n are portions of the test and training sets, respectively. The algorithm iterates in both dimensions performing $N/n \times M/m$ iterations, covering the whole training set for each test chunk before moving to the next one. On Arefin et al's method n and m have the same value, in order to split the chunk easily into square-shaped blocks. Our distance computation model, explained in detail in Section 4.2.1, allows arbitrary values of n and m that can be tuned to offer good run-time performance and fit into the available memory for a broad range of GPU devices.

All the components in a strip of the matrix, P_i^0 to $P_i^{N/n}$, are needed to find the neighborhood of the chunk i of the test set. To reduce the amount of memory needed, a local selection of the neighbors has to be performed. A selection operation can be computed for each chunk of the matrix, as shown in Figure 3, and then a global selection is performed on the local solutions. This

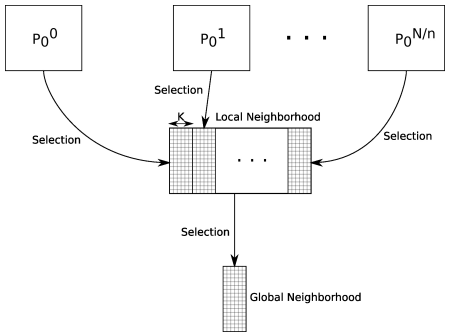

Figure 3: Local neighborhood selection scheme.

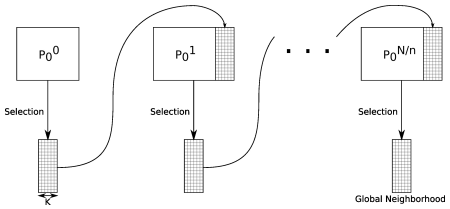

Figure 4: Incremental neighborhood selection scheme.

2-step computation scheme reduces the amount of memory needed. It keeps in memory $m \times k$ values instead of $m \times n$, taking into account that k is typically smaller than n. However, this strategy might not be enough for very large values of N.

Our algorithm computes the neighborhood in an incremental way, combining each chunk of the matrix with the resulting k neighbors of the previous chunks, as Figure 4 shows. This approach does not need a global selection step because the last local selection includes all the results obtained.

Furthermore, the amount of memory required is fixed regardless of the sizes of training and test sets, depending only on the values of m, n, k and the number of attributes of the dataset. These four values define the sizes of all data structures required for the computations and all of them are independent of the dataset size. The size of m and n can be decided in an arbitrary way in order to maximize the run-time performance and make the data fit into device memory. Section 4.3 details how to compute the exact memory footprint.

4.1.2. CPU-GPU memory transfers

CPU and GPU devices have different and exclusive memory banks. The input data and the results have to be copied between devices. The pseudocode presented on Algorithm 1 includes these interactions.

Algorithm 1: Proposed algorithm pseudocode.

input : Training and test sets, k

output: k nearest neighbors of each test instance in the training set.

```
1  copyTestPieceAsync_1
2  copyTrainPieceAsync_1
3  for i ← 1 to M/m do
4      checkTestCopy_i
5      for j ← 1 to N/n do
6          checkTrainCopy_j
7          computeDistanceMatrix_{i,j}
8          if j = N/n then
9              copyTestPieceAsync_{i+1}
10             copyTrainPieceAsync_1
11         else
12             copyTrainPieceAsync_{j+1}
13         end
14         computeSelection_j
15     end
16     copyNeighborhood_i
17     checkResultCopyAsync_i
18     copyResultPiece_i
19 end
```

$copyTestPieceAsync$ and $copyTrainPieceAsync$ represent the asynchronous copy of the instances required to compute a chunk of the matrix. When using asynchronous copies it is required to check if the copy has finished before using

the data. In Algorithm 1, *checkTestCopy* and *checkTrainCopy* represent this check. Except for the initial copy to start the algorithm, these copies (lines 8 to 13) are made in parallel during the selection process (line 14).

checkResultCopy and *copyResultPieceAsync* are, respectively, the safety check and the asynchronous copy of the neighborhood for a certain piece of the test set. The data structures used to keep the results in memory are the same for all the iterations. In order to avoid checking if the copy has finished on every iteration of the training set related loop, the final neighborhood is copied to a different structure, this operation is represented by *copyNeighborhood* in Algorithm 1.

4.2. Kernel design

Two different steps have been defined: the computation of a chunk of the distance matrix and the incremental selection of the k nearest neighbors. However, the square root calculation of the distance measure is applied only to the selected neighbors, in order to improve the run-time performance of the algorithm.

Therefore, three different kernel functions are used: the first kernel computes a chunk of the distance matrix, line 7 on Algorithm 1, the second kernel performs the selection of the k nearest neighbors, line 14 on Algorithm 1, and the third kernel computes the square root operation on the selected neighbors while the neighborhood is copied, line 16 on Algorithm 1. The following sections explain each kernel details.

4.2.1. Distance matrix computation

The distance matrix kernel computes the distances of one piece of the matrix of size $m \times n$, as commented on Section 4.1.1. Our method introduces a completely new distribution of the kernel threads that allows the customization of the parameters while delivering a high run-time performance. The kernel grid has m blocks, one per test instance, and d threads. In Figure 2 zoomed-in area, these d threads are represented as filled cells for the first block. Each thread of

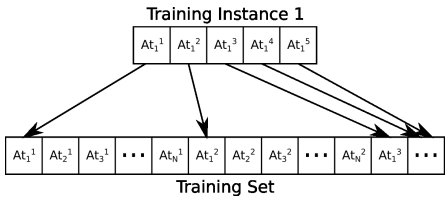

Figure 5: Dataset stored on memory. At_j^i represents attribute i of instance j.

the kernel computes several distances between 1 instance from the test set and n/d instances from the training set. The number of threads per block, d, is set to a value that delivers good performance on the GPU device regardless of the value of n that only defines the number of distance computations per thread.

None of the existing approaches uses such a thread distribution scheme. Most of them compute one distance per thread, introducing a high level of parallelism with extremely light threads. Kato et al. [18] is an exception, as all distances in a row are computed by one thread. This approach provides a higher workload per thread but reduces the degree of parallelism. Our design tries to find the right balance between both strategies in order deliver a better performance.

The input data is stored in a coalescent way to provide an efficient memory access. The distances are computed in parallel so all threads will request first the first attribute of the instance, then the second and so on. The dataset is stored in memory as an array of floating numbers as shown in Figure 5. As each block is related to one test instance, a thread computes several distances of the same test instance. Copying the values of the attributes of the test instance to shared memory provides a more efficient access rather than each thread requesting these values independently.

4.2.2. Neighborhood selection

The quicksort algorithm [15] is a well known algorithm to sort an array. The algorithm selects one element of the array (pivot) and divides the array into two parts: the left part stores the elements smaller than the pivot while the right part stores the elements greater than the pivot. Repeating the process in

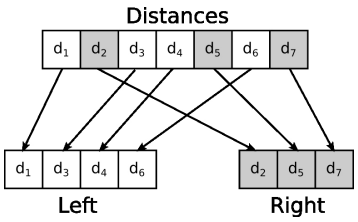

Figure 6: Non coalescent memory writings when dividing the vector.

a recursive way for each part finally gets the array sorted.

The majority of approaches in the literature use a sorting method to compute the neighborhood of the instances. This requires a high number of operations in order to completely sort the array. Our algorithm relies on a selection method to reduce the number of operations. The previous sorting technique can be adapted to select the k smallest elements of an array, repeating the process only for one of the parts of the vector: on the left part if this part has more than k elements or on the right part if the left part has less than k elements. In the second case, the left part of the array and the pivot are part of the selected elements.

This selection method is also used by Komarov et al.[19] in their method. However, this computational step has been improved in our method to avoid synchronization operations hence providing better run-time performance.

The ad-hoc design of this algorithm for GPU devices is to use a kernel to create the left and right parts of the array in a parallel way and then call the same kernel in a recursive way as it is done on CPU devices. This solution would be suitable only for the latest NVIDIA devices, which allow recursive kernel calls. However, the recursive step algorithm can be easily transformed into an iterative solution that works for most GPU devices.

Different structures are used for the distances array, the left part and the right part. At the end of each iteration, the left part or the right part, depending on their sizes, becomes the distance array and the former distance array is used as one of the parts.

Aside from this, the creation of the left and right parts of the array does not

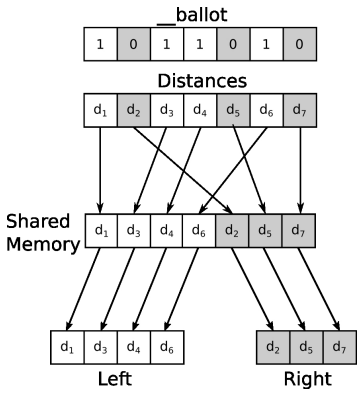

Figure 7: Local neighborhood selection

suit the characteristics of the GPU devices because it involves non coalescent memory accesses. With a grid configuration of m blocks (one per test instance), several comparisons can be done in parallel but, depending on the result of the comparison, each thread will need to write on a different part of the memory penalizing the run-time performance, see Figure 6. To solve this problem, the results are written in two steps. The first step writes the element to a shared memory array in a non coalescent way grouping the elements smaller than the pivot on one side and the elements bigger on the other side. The second step writes the element to its final position in global memory using two coalescent memory accesses, one for the left part and another for the right part of the array, as shown in Figure 7. The right part memory access can be skipped once the left part has more than k elements, improving the run-time performance.

However, to find the position where each element has to be written is a problem in itself. Each thread needs to know how many elements from the threads with a lower index are greater or smaller than the pivot. CUDA provides functions, *__ballot* and *__popc*, to solve this problem but only within a warp. The *__ballot* function creates a 32-bit integer where each bit is the result of a predicate evaluated by each thread of the warp, see Figure 7, while the *__popc* function counts the number of bits set to 1 in a 32-bit integer. Using the proper bit mask for each thread, it is possible to get the writing position within the warp.

A grid of m blocks and 32 threads per block can select the neighborhood but it might not provide enough workload for the GPU device. Generally, a higher number of threads per block provides a better performance. There are two ways of increasing the number or threads: using more than 32 threads to build the array parts or processing different arrays on each warp.

The first solution is used by Komarov et al.[19]. It requires sharing the values obtained with *__ballot* on each warp and to set synchronization points to ensure the values are correct. These requirements would penalize the run-time performance so we decided to use the second solution. The neighborhood of each test instance is computed by a single warp and each block computes several neighborhoods. However, if the number of test instances is small, Komarov's et al. algorithm obtains better GPU occupancy ratios. Our design uses fewer threads per selection step requiring a larger number of test instances to fully occupy the resources of the GPU device.

The value of the pivot used in the quicksort algorithm has an impact on the run-time performance. The pivot value is usually selected as the median of the first, last and center values of the array as an approximation of the median of the array. Using the median value of the array as pivot produces equally sized array parts, halving the size in each iteration, but we can use a more aggressive strategy thanks to the incremental neighborhood scheme. If a distance is larger than the farthest neighbor from the last chunk, that instance is not going to be a part of the neighborhood because there are already k smaller values. Setting the pivot to the farthest neighbor distance of the previous iteration of the algorithm focuses the effort on the interesting values providing a better performance. For the first matrix chunk of each test chunk, this solution cannot be applied, since we do not have a previous neighborhood, so the usual heuristic is used.

When the left part of the array is smaller than k, those distances and the pivot belong to the final neighborhood of that step. However, as the selection algorithms continues and needs to reuse that memory area, these values are copied to a different array of $m \times k$ elements.

4.2.3. Square root calculation

The square root operation is a costly operation and it is not required to select the neighborhood of an instance. In our classification scenario, this operation can be skipped. However, it is performed for two reasons: to provide the accurate distance information in case any future application needs it and to be able to establish a fair comparison with other approaches that perform this operation.

Other algorithms, like Garcia et al. [14], also use a specific kernel for the square root computation, although it is only briefly commented in the corresponding papers, as a minor optimization. In our design, this kernel has been combined with one of the required memory transfer operations, as commented in Section 4.1.2. This provides better run-time performance than performing two separate operations.

All the selected distances are located on coalescent memory, it can be considered as a long array. This array has a size of $m \times k$ elements. In order to get the highest possible occupancy of the GPU device, we split the array to create $\frac{m}{128}$ blocks of 128 threads. Each thread performs a square root operation on k distances and copies their respective indexes in the training set to the separate structures.

4.3. Total memory required

As mentioned before, the distance matrix size can be defined in an arbitrary way depending on the values of the parameters k, m and n. The amount of memory required for the rest of the operations also depends on these parameters and on the number of attributes of the dataset, D.

This way, it is possible to define an expression that represents the amount of elements required to keep on memory for each configuration of the parameters:

$$D \times m + D \times n + 6(m \times (n + k)) + 4(m \times k) \tag{2}$$

The exact amount of memory can be obtained weighing each part of the equation with the size, in bytes, of the type of elements (floating point numbers, integers)

that are stored. The memory requirements of the experiments performed are shown in Section 5.3.

5. Experimental results of GPU-SME-kNN

Different experiments have been carried out and the results obtained are presented here. Section 5.2 presents the hardware and datasets used and the experiments performed, Section 5.3 shows the results obtained and Section 5.4 analyses the results.

5.1. Experiments

Although the design of the algorithm for GPU and CPU devices changes significantly, the same computations are performed in both devices. This means that the same algorithm obtains the same results regardless of the device. Taking this into account, the experiments have been designed to highlight the efficiency differences between GPU-SME-kNN and the reference implementations.

Two large datasets have been selected for the experiments. These datasets have been subsampled at different sizes to show how the behavior of the algorithm changes as the size of the dataset increases. A 5-fold cross validation scheme has been used with all sizes of the dataset. This scheme reduces the impact that the relative order of the instances within the dataset can have on the performance. Different values of k have also been used. The results shown in Section5.3 are computed as the average times of the values obtained.

The performance of GPU-SME-kNN has been compared with several existing techniques:

1. The technique of Garcia et al. [14], denoted as GPU-Garcia-kNN, available on Github.
2. The technique of Arefin et al. [5], GPU-FS-kNN, is also available on-line, but modified to use the Euclidean distance instead of the Pearson distance in order to make a fair comparison.

3. The technique of Komarov et al. [19], denoted as GPU-Komarov-kNN, is not available on-line, but we have implemented their design of the quicksort selection with our incremental neighborhood calculation scheme. Using the same scheme for the distance calculation the differences rely on the design of the selection algorithm. In addition, our scheme allows this method to scale to problem sizes that the original one cannot address. The parameters n and m have been set to $65\dot{5}36$ and 2048, respectively, in order to provide a matrix chunk of a size similar to the one used in [19] .

A website associated to this paper has been created that includes the datasets, results and code of GPU-SME-kNN and GPU-Komarov-kNN used in this work. The URL for this website is: `http://sci2s.ugr.es/GPU-SME-kNN/`

5.2. Hardware and datasets

The experiments have been performed on a server-class computer equipped with a high-end GPU device. This computer has an Intel Xeon E5-2630 processor at 2.30 GHz. The GPU device is a NVIDIA Tesla K20m with 5GB of RAM memory and 2496 CUDA cores. Nevertheless, the proposed design can be run on a computer with lower specifications.

Two large datasets from the UCI repository [6] have been used in the experiments:

- The poker dataset has 1 025 009 instances, 10 attributes and 10 classes. The dataset has been subsampled at sizes ranging from 50 000 to 1 000 000 instances in steps of 50000 instances.

- The KDDCup 1999 dataset has 4 898 431 instances, 41 attributes and 5 classes. This dataset has been subsampled in steps of 250 000 instances from 250 000 to 1 500 000 instances and in steps of 500 000 instances for larger sizes.

Different experiments have been performed with these datasets using a 5-fold cross validation scheme for all sizes. The experiments use k values of 5, 100

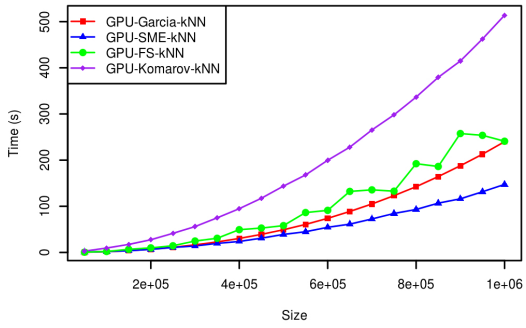

Figure 8: Poker dataset results with $k = 5$.

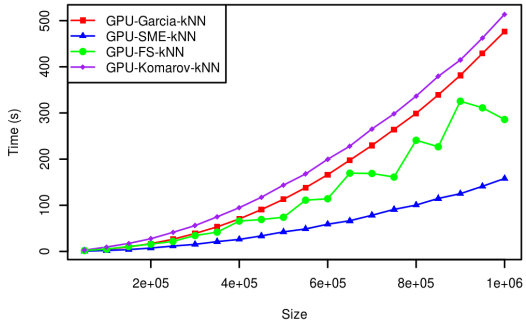

Figure 9: Poker dataset results with $k = 100$.

and 1000 for both datasets. Some k values might be too large to offer accurate classification ranges but the objective is to assess the scalability of the evaluated methods in relation to k.

5.3. Empirical results

This section presents the experiments and the results obtained on the previously mentioned hardware and datasets. For all the experiments the value of m was set to 16384 and the value of n to 2048, the number of threads per block for the distance matrix kernel, d is 256.

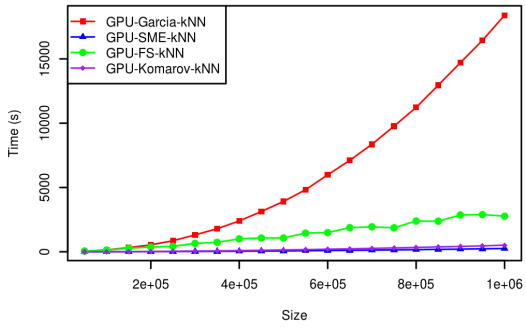

Figure 10: Poker dataset results with $k = 1000$.

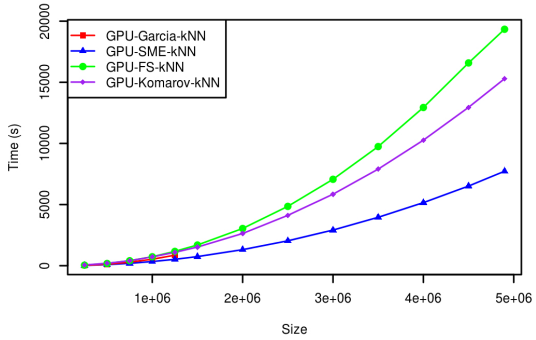

Figure 11: KDDCup 1999 dataset results with $k = 5$.

Figures 8 to 10 present the results for the poker dataset. Three different values of k have been used with this dataset: 5, 100 and 1000.

Figures 11 to 13 present the results for the KDDCup 1999 dataset. The algorithm of Garcia et al. was able to complete the experiments successfully for this dataset only up to 1 250 000 instances. For bigger sizes the experiments failed due to memory requirement problems. Tables 1 to 3 compare the results of all approaches for these values. GPU-FS-kNN also presents some issues with this dataset and the largest value of k(1000). In this case, the software provided by the authors behaves in an abnormal way for sizes larger than 1.5 million of

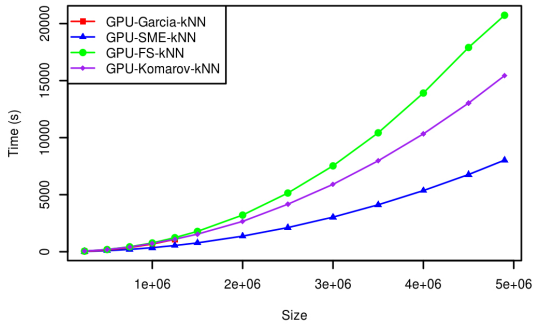

Figure 12: KDDCup 1999 dataset results with $k = 100$.

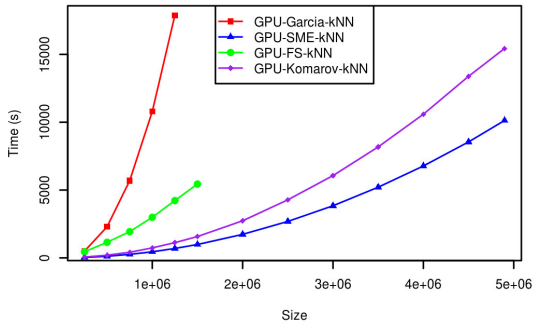

Figure 13: KDDCup 1999 dataset results with $k = 1000$.

instances and it does not provide accurate results.

5.4. Analysis of the results

As the results of the previous section presents, GPU-SME-kNN outperforms the results of the other approaches. The strategy for dealing with large distance matrices causes high run-times performance differences. GPU-Garcia-kNN stores full distance matrix strips on GPU memory to compute the neighborhood. The matrix strip width becomes smaller as the size of the training set increases and this reduces the performance of the design. GPU-SME-kNN keeps the width of the matrix, through the incremental neighborhood computation,

Table 1: KDDCup 1999 dataset time results, in seconds, with $k = 5$

Size	GPU-Garcia-kNN	GPU-SME-kNN	GPU-FS-kNN	GPU-Komarov-kNN
250 000	29.052	**25.192**	45.127	60.837
500 000	121.281	**90.243**	180.955	196.098
750 000	284.577	**194.467**	409.278	407.561
1 000 000	528.100	**338.979**	732.452	719.672
1 250 000	862.306	**522.745**	1170.310	1093.420

Table 2: KDDCup 1999 dataset time results, in seconds, with $k = 100$

Size	GPU-Garcia-kNN	GPU-SME-kNN	GPU-FS-kNN	GPU-Komarov-kNN
250 000	37.931	**26.182**	50.388	59.856
500 000	158.213	**93.964**	193.651	194.744
750 000	366.198	**202.579**	432.981	405.127
1 000 000	667.465	**352.820**	772.101	725.162
1 250 000	1083.842	**544.529**	1233.264	1097.123

providing the desired performance regardless the training set size.

The run-time performance differences with GPU-Garcia-kNN also rise when value of k is increased. A higher value of k requires more space to store the neighborhood, reducing the amount of memory available for the distance matrix which produces the same effect that happens when the training set size increases. However, the difference is also introduced by the selection method. GPU-Garcia-kNN uses an insertion method to select the neighborhood of a test instance on each thread of the kernel. By increasing the value of k the probability of code divergence also increases. The divergence happens when some threads in a warp find a neighbor candidate at some position but not all threads find it. The threads that did not find a neighbor have to wait until the threads that find one make the required computations.

The code divergence problem also affects GPU-FS-kNN. However, the modifications of the insertion scheme introduced in its selection method lower the impact of the divergence on the run-time performance. Increasing the value of k

Table 3: KDDCup 1999 dataset time results, in seconds, with $k = 1000$

Size	GPU-Garcia-kNN	GPU-SME-kNN	GPU-FS-kNN	GPU-Komarov-kNN
250 000	496.624	**33.890**	449.668	61.665
500 000	2297.120	**119.163**	1150.744	199.355
750 000	5691.280	**259.552**	1932.540	415.234
1 000 000	10 788.800	**448.624**	2992.348	736.441
1 250 000	17 885.280	**694.892**	4221.566	1125.497

Table 4: GPU-SME-kNN GPU memory usage in MB

k	Poker dataset	KDDCup 1999 dataset
5	1158	1162
100	1247	1251
1000	2091	2159

also reduces the performance because this method does not uses asynchronous memory copies. This introduces computation idle times while the results are copied from GPU to CPU memory. The distance computation scheme of GPU-FS-kNN, which is similar to our incremental neighborhood computation, allows this method to complete almost all the experiments but the performance is affected by the issues mentioned above.

GPU-SME-kNN does not suffer from this kind of code divergence problems, as a warp collaborates to select the neighborhood. Furthermore, this selection method reduces the impact of the value of k: each iteration the same steps are followed, the value of k only changes the number of iterations made. However, the most important factor on the run-time performance of the selection step is the incremental neighborhood computation.

Our selection method requires a large number of memory accesses to read the array and store the left and right parts. Although these accesses are programmed on a coalescent way and are efficient, a large number of iterations could be required depending on the quality of the pivots provided by the heuristic, especially when k is set to a small value. The incremental neighborhood tech-

nique provides the maximum distance to be considered a neighbor candidate, reducing the number of iterations.

Our implementation of GPU-Komarov-kNN also uses this idea. However, we cannot see the effect in the results because GPU-Komarov-kNN uses 512 threads per selection step [19] whereas GPU-SME-kNN uses 32. As we commented in Section 4.2.2, by using 32 threads it is possible to avoid synchronization operations, improving the run-time performance of GPU-SME-kNN. In addition, the use of a small number of threads improves the use of the resources of the GPU. If the array size is lower than the number of threads per selection some threads do not perform any computation, but their resources cannot be released until the selection process is finished. This situation is reached in a faster way when a large number of threads per selection is used, taking into account that the size of the array is approximately halved in each iteration of the algorithm. Small values of k highlight this fact. On the other hand, when using large values of k GPU-Komarov-kNN outperforms other approaches. However, as we can see in Figure 13, the differences against GPU-SME-kNN are significant. The situation is similar in Figure 10 but, in this case, the results of the GPU-Garcia-kNN technique affect the scale of the plot and it cannot be observed.

For scenarios with a small number of test instances the run-time performance differences between GPU-SME-kNN and GPU-Komarov-kNN would decrease. The higher number of threads used in the selection step makes GPU-Komarov-kNN reach the maximum occupancy of the GPU device faster than GPU-SME-kNN. Therefore, under some circumstances, GPU-Komarov-kNN could outperform GPU-SME-kNN. However, in the presented 5-fold cross validation results, GPU-SME-kNN exhibits a better performance, especially as the number of test instances increases.

In cases with similar number of training and test instances the run-time performance differences between GPU-SME-kNN and GPU-Komarov-kNN would be even larger. This situation can be found in different scenarios, for instance, some steps of lazy learning algorithms require computing the neighborhood of the training test. Taking into account that the differences are already significant

in the results presented in this section, only GPU-SME-kNN can compute these steps in a reasonable time.

Table 4 shows that the amount of memory used by GPU-SME-kNN is similar for both datasets regardless of the difference of more than 3 500 000 instances. The memory requirements difference is actually caused by the different number of attributes of both datasets. In all cases, the amount of memory required is relatively small making GPU-SME-kNN suitable for most current GPU devices.

6. GPU-based Lazy learning

The design patterns used on the GPU-based method for the kNN rule should also be suitable for lazy learning algorithms. The following sections detail the algorithms that have been adapted (Section 6.1), the design modification that they required (Section 6.2) and the results obtained (Section 6.3).

6.1. Algorithms

We have selected three different algorithms from the ones available on the KEEL software tool [4] in order to test our design. These algorithms are described in this section.

6.1.1. Center$k$NN

The Center$k$NN technique [12] is a lazy learning algorithm based on the kNN rule that modifies the distance computation. The algorithm computes the center of each class, as the average of the instances that belong to that class and uses this value to modify the reference point used to measure the distance.

To compute the distance between a test instance, y, and a training instance, x, the algorithm computes first the line that passes through x and the center of its class, c_x, then, projects y onto that line and uses the resulting point p_y to measure the distance. Figure 14 shows these values graphically.

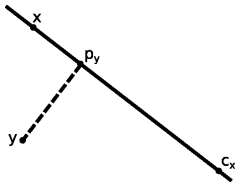

Figure 14: Center$k$NN distance measure, the dashed line is the distance measured

6.1.2. kNN adaptive

The kNN adaptive algorithm [31] weighs the measured distance of the kNN rule. The weight used is specific for each training instance. As a previous step before using the kNN rule, each training instance computes the distance to the closest instance that does not belong to its own class within the training set.

That distance is used as weight when the kNN rule is applied. The distance between a training and a test instance is divided by this weight. This weighting scheme considers training instances that are close to the frontier of two classes less reliable as neighbors, than the ones in the center of clusters of the same class.

6.1.3. Symmetric kNN

The Symmetric kNN algorithm [23] computes the kNN rule in both directions. In particular, the algorithm computes the kNN rule of each training instance compared to the training set, as a first step. When the distance between a training instance and a test instance is computed, it is compared to the distance of the farthest neighbor of the training instance. The test instance would be part of the neighborhood of the training instance if the distance between them is smaller than the distance from the training instance to the farthest neighbor of the training instance. A training instance is considered part of the symmetric neighborhood when this happens.

This symmetric neighborhood is joined with the regular neighborhood, obtained with the usual application of the kNN rule, to obtain the final neighborhood that classifies the test instance. The join operation is made in a way that avoids double voting if there is an instance in both symmetric and regular

neighborhoods.

6.2. GPU design modifications

The lazy learning algorithms can use the same distance matrix and incremental neighborhood calculation scheme as the kNN rule. However, these algorithms require a previous step where part of the information they need is calculated. The specific modifications for each method are presented in this section.

6.2.1. Center$k$NN

The Center$k$NN algorithm requires the computation of the centers of each class as a previous step to the kNN rule. The center of a class is computed as the average value for each attribute on a training instance that belongs to that class. This step can be performed on the CPU device when the dataset is loaded into RAM memory. These center values are copied to device memory only once, before the first piece of the matrix is computed.

The projection of the test instance is computed in the distance kernel. The kernel keeps the same structure but it performs more operations to compute the projected instance.

6.2.2. kNN adaptive

The weight values of the kNN adaptive technique can be computed as the kNN rule with $k = 1$ ignoring the training instances that belong to the same class of the instance whose weight is being computed. A modified version of the kNN rule that introduces a void value in the matrix when both instances have the same class is used to compute the weight for each instance as a previous step.

The original kNN rule is also modified in order to include the weighting of the distance. In this algorithm, the square root of the distance needs to be computed for every training instance because the weight is different for each one and can modify the selected instances.

The weights of the training instances are copied chunk-wise to a dedicated array alongside the copy of the training chunk they correspond to.

6.2.3. Symmetric kNN

To apply the kNN rule in both directions, it is required to know the farthest distance of the neighborhood of each training instance. A modified version of the kNN rule that introduces a void value in the matrix when the training and test instances have the same index is used to compute the neighborhood of the training set as a previous step.

The distance matrix kernel of the original kNN rule is modified to compare the distance obtained with the farthest distance of the neighborhood. The symmetric part of the rule is satisfied when the distance computed by the kernel is smaller than the one stored before. When that happens, the training instances vote, increasing the value of its class in a voting structure specific to this algorithm.

The voting structure has a size of $m \times C$ where C is the number of classes. This structure is set to 0 at the beginning of each strip of the matrix and it is copied with the final neighborhood. The process that assigns the class to the test instance uses these votes as base and adds the votes of the final neighborhood, checking if these instances have already voted to avoid double voting.

6.3. Empirical Results

The lazy learning algorithms have been tested against the CPU implementation available on the KEEL software[4]. These algorithms have been tested with the poker dataset up to 650 000 instances using a 5-fold cross validation scheme. Figure 15 shows the results for the Center$k$NN. The implementation available on KEEL does not allow changing the k value for this algorithm which is set to 1.

Figures 16 and 17 present the results for the kNN adaptive and the symmetric kNN algorithms. In both algorithms, k can be set to different values, and for these experiments, the value selected was 5.

The results of the lazy learning algorithms show a similar behavior on the three cases: our approaches reduce the time from hours to minutes. Center$k$NN shows the highest differences, as the computation of the projection introduces

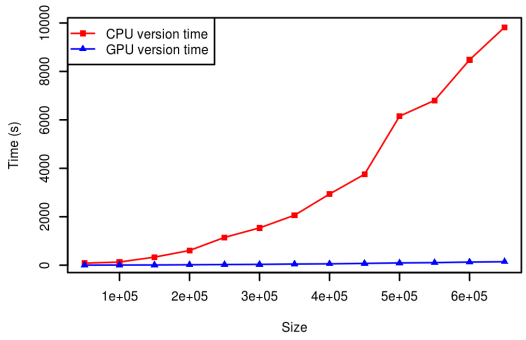

Figure 15: Center$k$NN results on poker dataset.

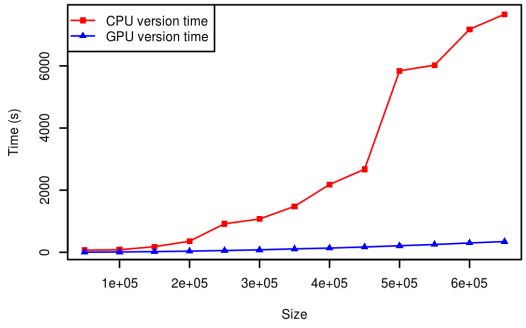

Figure 16: kNN adaptive results on poker dataset.

more floating point computations that can be addressed efficiently by GPU devices. On the other hand, Symmetric kNN requires more comparisons and data accesses, which are addressed less efficiently than floating point operations on GPU devices, making the differences smaller. The performance of kNN adaptive is similar to the symmetric kNN.

7. Conclusions

We have presented a new GPU-based approach for the kNN rule that outperforms the approaches in the literature and provides a high scalability in terms

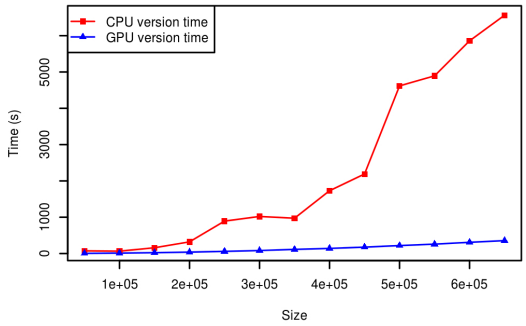

Figure 17: Symmetric kNN results on poker dataset.

of dataset size and k value. GPU-SME-kNN keeps a stable level of memory usage that allows to address any dataset regardless of its size, which was not possible by any of the previous GPU kNN methods. Furthermore, given that (1) the memory footprint of the method can be totally controlled by user-defined parameters and that (2) we do not use capabilities only present in the most recent GPU cards, our method can be efficiently used across a very broad range of GPU devices with varying amount of card memory and CUDA capabilities.

We have also proven that our design is suitable for lazy learning algorithms based on the kNN rule. The run-time performance of the three algorithms presented, Center$k$NN, kNN adaptive and Symmetric kNN, has been improved in a significant way reducing the run-time from hours to minutes.

All own code is available as open source, along with the datasets and results, on the website: `http://sci2s.ugr.es/GPU-SME-kNN/`

Acknowledgements

This work was supported by the research Projects TIN2014-57251-P, TIN2013-4720-P and P12-TIC-2958. P.D. Gutirrez holds an FPI scholarship from the Spanish Ministry of Economy and Competitiveness (BES-2012-060450) and a short stay in foreign institutions scholarship (EEBB-I-14-08977).

References

[1] CUDA, http://www.nvidia.com/object/cuda_home_new.html.

[2] NVIDIA, http://www.nvidia.com/.

[3] D. W. Aha, Lazy Learning, Springer, 1997.

[4] J. Alcalá-Fdez, L. Sánchez, S. García, M. del Jesus, S. Ventura, J. Garrell, J. Otero, C. Romero, J. Bacardit, V. Rivas, J. Fernández, F. Herrera, KEEL: A software tool to assess evolutionary algorithms for data mining problems, Soft Computing 13 (3) (2009) 307–318.

[5] A. S. Arefin, C. Riveros, R. Berretta, P. Moscato, GPU-FS-kNN: A Software Tool for Fast and Scalable kNN Computation Using GPUs, PLoS ONE 7 (8) (2012) e44000.
URL http://dx.doi.org/10.1371%2Fjournal.pone.0044000

[6] K. Bache, M. Lichman, UCI machine learning repository (2013).
URL http://archive.ics.uci.edu/ml

[7] G. Beliakov, G. Li, Improving the speed and stability of the k-nearest neighbors method, Pattern Recognition Letters 33 (10) (2012) 1296 – 1301.

[8] B. Catanzaro, N. Sundaram, K. Keutzer, Fast support vector machine training and classification on graphics processors, 2008, pp. 104–111.

[9] T. Cover, P. Hart, Nearest neighbor pattern classification, Information Theory, IEEE Transactions on 13 (1) (1967) 21–27.

[10] A. Dhurandhar, A. Dobra, Probabilistic characterization of nearest neighbor classifier, International Journal of Machine Learning and Cybernetics 4 (4) (2013) 259–272.

[11] R. O. Duda, P. E. Hart, D. G. Stork, Pattern Classification (2Nd Edition), Wiley-Interscience, 2000.

[12] Q.-B. Gao, Z.-Z. Wang, Center-based nearest neighbor classifier, Pattern Recognition 40 (1) (2007) 346–349.

[13] E. Garcia, S. Feldman, M. Gupta, S. Srivastava, Completely Lazy Learning, Knowledge and Data Engineering, IEEE Transactions on 22 (9) (2010) 1274–1285.

[14] V. Garcia, E. Debreuve, F. Nielsen, M. Barlaud, K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching, in: Proceedings - International Conference on Image Processing, ICIP, 2010, pp. 3757–3760.

[15] C. A. R. Hoare, Algorithm 64: Quicksort, Commun. ACM 4 (7) (1961) 321–.

[16] L. Huang, Z. Li, A novel method of parallel gpu implementation of knn used in text classification, in: Networking and Distributed Computing (ICNDC), 2013 Fourth International Conference on, 2013, pp. 6–8.

[17] L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi, Y. Shi, Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA), Journal of Supercomputing 64 (3) (2013) 942–967.

[18] K. Kato, T. Hosino, Multi-GPU algorithm for k-nearest neighbor problem, Concurrency and Computation: Practice and Experience 24 (1) (2012) 45–53.

[19] I. Komarov, A. Dashti, R. M. D'Souza, Fast k-NNG Construction with GPU-Based Quick Multi-Select, PLoS ONE 9 (5) (2014) e92409.
URL `http://dx.doi.org/10.1371%2Fjournal.pone.0092409`

[20] Q. Kuang, L. Zhao, A practical GPU based KNN algorithm, in: In Proceedings of the Second Symposium on International Computer Science and Computational Technology (ISCSCT 09), 2009.

[21] M. Lastra, J. Carabao, P. D. Gutiérrez, J. M. Benítez, F. Herrera, Fast fingerprint identification using GPUs, Information Sciences 301 (0) (2015) 195 – 214.

[22] L. Mussi, F. Daolio, S. Cagnoni, Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture, Information Sciences 181 (20) (2011) 4642 – 4657, special Issue on Interpretable Fuzzy Systems.

[23] R. Nock, M. Sebban, D. Bernard, A simple locally adaptive nearest neighbor rule with application to pollution forecasting, International Journal of Pattern Recognition and Artificial Intelligence 17 (8) (2003) 1369–1382.

[24] J. R. Prasad, U. Kulkarni, Gujrati character recognition using weighted k-nn and mean χ 2 distance measure, International Journal of Machine Learning and Cybernetics 6 (1) (2015) 69–82.

[25] Y. Pu, J. Peng, L. Huang, J. Chen, An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl, in: Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on, 2015, pp. 167–170.

[26] A. Rajaraman, J. Ullman, Mining of Massive Datasets, Cambridge University Press, 2011.

[27] M. Schatz, C. Trapnell, A. Delcher, A. Varshney, High-throughput sequence alignment using Graphics Processing Units, BMC Bioinformatics.

[28] G. Shakhnarovich, T. Darrell, P. Indyk, Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Neural Information Processing), MIT Press, 2006.

[29] I. Stamoulias, E. Manolakos, Parallel architectures for the kNN classifier - Design of soft IP cores and FPGA implementations, Transactions on Embedded Computing Systems 13 (2).

[30] N. Tomašev, M. Radovanović, D. Mladenić, M. Ivanović, Hubness-based fuzzy measures for high-dimensional k-nearest neighbor classification, International Journal of Machine Learning and Cybernetics 5 (3) (2014) 445–458.

[31] J. Wang, P. Neskovic, L. Cooper, Improving nearest neighbor rule with a simple adaptive distance measure, Pattern Recognition Letters 28 (2) (2007) 207–213.

[32] D. Wilson, T. Martinez, Improved heterogeneous distance functions, Journal of Artificial Intelligence Research 6 (1997) 1–34.

[33] X. Wu, V. Kumar, The top ten algorithms in data mining, CRC Press, 2010.