Robust GPGPU plugin development for RapidMiner

Andor Kovács, Zoltán Prekopcsák
vermilion.andor@gmail.com, prekopcsak@tmit.bme.hu
Budapest University of Technology and Economics

Abstract

In recent years, significant number of papers [1][2] have been published about general-purpose graphical processing unit (*GPGPU*) programs which are able to accelerate computationally intensive applications by several times over conventional CPU programs. These papers raise an important question: With the current developer tools is it possible to integrate these *GPU* programs into a major industry application without serious performance degradation?

We chose the k-Nearest Neighbors algorithm and one of the most popular open source data mining tool, RapidMiner, to analyze the potential of *GPU* plugin development.

1 Introduction

As *GPGPU* capable graphics cards become cheaper, there is an increasing demand for such software which is able to take advantage of the *GPU*s' opportunities. However these programs are a lot awaited in some application areas, one of these is data mining. During our work, we chose the k-Nearest Neighbors algorithm (*k-NN*) which is widely used for classification, machine learning and pattern recognition by data miners, but unfortunately its high computational complexity is a serious limitation. This paper mainly focuses on the integration of the k-NN algorithm's *GPU* version into RapidMiner, while our previous paper focused on the *GPU* implementation itself [3].

The result of our work is a plugin called RapidMiner for Cuda Environment (*RACE*). It was implemented in NVIDIA's CUDA C language. Due to the fact that RapidMiner was written in *Java*, a binding for CUDA was necessary. It is called JCUDA [4].

The remainder of this paper is organised as follows. Next, we give an overview of the plugin's technical background. Section 3 analyses the requirements for robust GPGPU plugin development. In Section 4, we introduce

RapidMiner and Section 5 presents the integration of the plugin into RapidMiner. Section 6 describes the structure of the GPU k-NN operator. Finally, we analyse our plugin and demonstrate it scales well for various input sizes.

2 Technical Background

This section introduces the technologies we used for plugin development.

2.1 *GPGPU* and CUDA

Compute Unified Device Architecture (CUDA) [5] is a general parallel programming architecture. It provides a high-level programming language for the utilization of the parallel power of the *GPU*. This architecture greatly simplifies the programmers' task because there is no need for notable computer graphics experience and it also allows great scalability.

2.2 CUDA Capable *GPU* Architectures

CUDA capable cards' capabilities can be easily classified by a number called *"Compute Capability"*. This number consists of two parts: a major and a minor one. If the GPU is based on the current Nvidia architecture called Fermi the major value is 2, otherwise it is 1. The complex memory hierarchy of the Fermi architecture is shown in Figure 1. Fermi's most important new feature, compared to the previous architectures, is the global memory L1 and L2 cache hierarchy. Some optimizations of the plugin are based on this difference.

2.3 JCUDA

To be able to execute a CUDA C code from Java, a binding called JCUDA was necessary. It uses the Java Native Interface and allows the programmer to access all functions of the CUDA C Driver API from Java. Java's references are not flexible enough therefore JCUDA defines two pointer classes: one for CPU and another for *GPU* memory allocation. JCUDA uses the CUDA Driver API to execute precompiled kernels, which are CUDA functions executable on a *GPU*.

Parallel Thread Execution [5] (*PTX*) is Nvidia's intermediate language used in CUDA environment. The CUDA compiler (called NVCC) translates CUDA C source code to *PTX*. The *GPU*'s driver contains a just-in-time compiler which translates *PTX* into an executable *GPU* architecture specific binary code. We chose *PTX* because its architectural flexibility and forward compatibility.

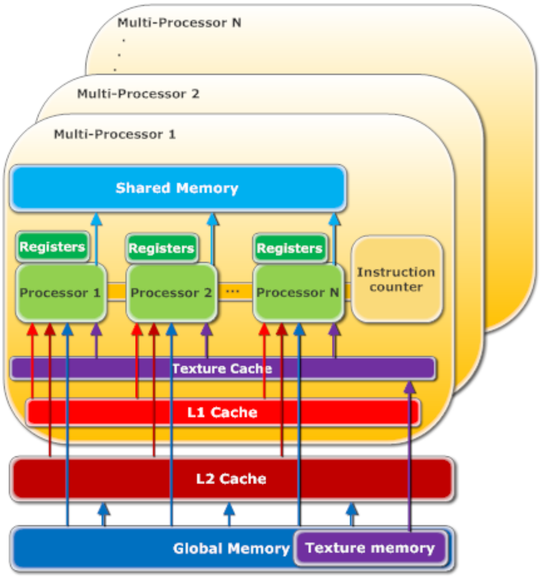

Figure 1: Nvidia Fermi GPU memory and cache hierarchy [5]

3 Requirements

After the presentation of the technologies, we analyze the essential requirements for robust *GPGPU* plugin development. The most important requirement is that the plugin has to provide the same features like its CPU equivalent and the users have to be able to use the plugin like the CPU version and easily take advantage of the plugin's speedup without any *GPU* specific knowledge.

The second most important feature is about improving the scalability of the previous implementations. That is very crucial because data mining on the *GPU* is not a specific area where programmers can easily define the minimum and maximum size of the input data set. The plugin should perform as fast as possible even if the size of the input varies by several orders of magnitude [3].
Variables which should be considered:

- Number of elements in the Train set
- Number of elements in the Test set
- Value of k
- Number of attributes

Based on the previous papers, there is no doubt if the size of these parameters varies by several orders of magnitude, then a single kernel is unable to provide that high level of performance scalability and flexibility which is achievable by the memory hierarchy of the GPU.

Learning from the previous solutions' mistakes, the plugin should be able to cut the problem into sub-problems because of the parallel execution of the program. A relatively small input data can cause running out of the *GPU*'s global memory. Therefore, the plugin should be able to solve this efficiently.

4 Introduction to RapidMiner

RapidMiner is an open source software which offers high variety of data mining functions for the users. Its developers wanted to create an easy-to-use tool for data mining. They have achieved this by a user-friendly graphical interface. The users do not need to be able to handle a complex command-line interface nor do they need to learn any scripting languages.

4.1 RapidMiner's Data Storage Strategy

RapidMiner's strategy is based on a class called *"Example Table"*. It was designed for storing actual raw data. In this representation, the data does not have any meaning yet because most of the data is saved as a single or a double precision floating point number. RapidMiner does not duplicate raw data, in contrast of that it just creates views -called *"Example Set"*- for the *"Example Table"*. These views can contain any number of columns and any rows of the *"Example Table"* in any order [6].

In the next subsection, we explain a really important thing, how the users can interact with the CPU implementation of the k-NN algorithm.

4.2 RapidMiner's k-NN Operator

Functions in RapidMiner are called operators. These are simple rectangles on the user interface. Nearly all of them have input and output ports. The users can place any number of these operators on the interface of the RapidMiner and they can easily wire them together by connecting one operator's output to the other's input. As a result, when the first operator finishes its execution it sends the result object through the wire to the input of the next operator. Therefore, the user can create workflows from these operators. The usual workflow of the CPU based k-NN is shown in Figure 2.

In this example, there are three types of operators. The first one is called *"Read CSV"*. It reads a file into the memory, which consists of comma-

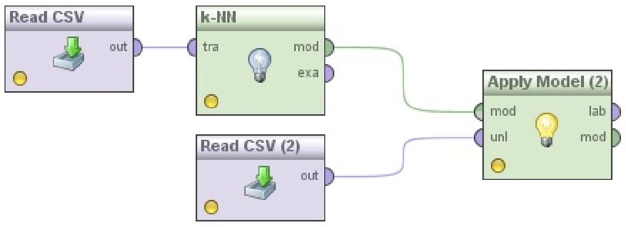

Figure 2: The CPU version of the k-NN operator on the graphical user interface of RapidMiner

separated values, and stores them in an Example Table object. After this, it sends this object to the input of the *"k-NN"* operator. It modifies this object and creates a model from it, which makes the k-NN calculation easier, then sends this model to its output port called *"mod"*. As you can see, it is connected to the *"mod"* input port of the operator called *"Apply model"*. It has another input port. In this case the user wired the second *"Read CSV (2)"* to this. This is responsible for the reading of the test set, while the other one was responsible for the train set, the two important input files of the k-NN algorithm. *"Apply model"* executes the whole algorithm and sends the results to its *"lab"* output. The result is an *"Example Set* which includes a *"prediction"* attribute.

5 Integration of the Plugin

In this section we present the integration of RACE into RapidMiner.

5.1 Overview of the Integration

As you can see in Figure 3, RACE connects RapidMiner to the Nvidia CUDA Driver API through JCUDA and Java Native Interface. This makes the communication with an Nvidia *GPU* possible.

Do note, that RACE has several CUDA C kernels. We compiled these kernels to *PTX* with Nvidia's official CUDA compiler called *NVCC*. When RACE calls these kernels, the Driver API compiles just in time the *PTX* intermediate representation to an executable binary and executes it on the *GPU*.

Java objects would need more memory space on the *GPU* and would rise serious optimization questions. It would not be an efficient solution for the memory management of the plugin. In conclusion, we used float or double arrays, because they fit much better for the *GPU* architecture.

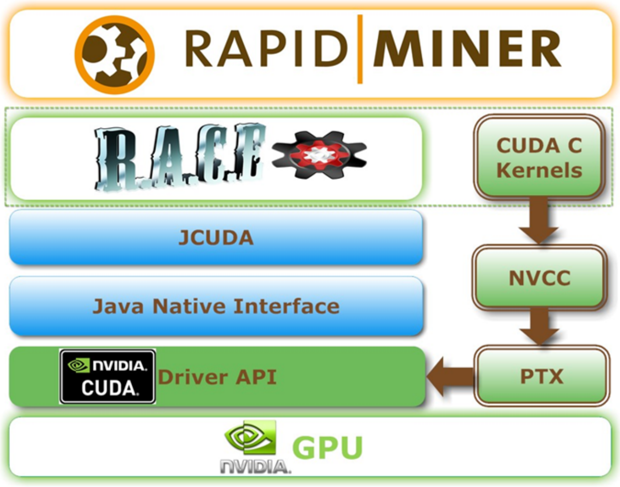

Figure 3: Overview of the Integration

Page-locked memory is a very important part of the optimization of RACE's memory model. With using it, the bandwidth between CPU and *GPU* memory is much higher. Due to this, it drastically decreases the time of memory copies. But unfortunately it reduces the overall amount of physical memory available to the operating system for paging, so too much page-locked memory consumption reduces overall performance [5].

5.2 Asynchronous Operation of the Plugin

Figure 4 illustrates how the plugin operates after the slicing of an example input problem into 3 nearly equal sized slices. The number of slices depends on the size of the *GPU*'s global memory and the type of the algorithm. It should be mentioned, that slicing is only possible for algorithms like k-NN, which are naturally parallelizable. As we mentioned in the previous section *"Example Table"* stores the Test and Train set of the algorithm, and it also stores the result of the computation. Due to this, RapidMiner will be able to visualize the results on its user interface.

Figure 4 shows the parallel operations executed on the 3 parts of the input. While RACE copies the third part of the input file from the Example Table Java object to the page-locked C double array, meanwhile the *GPU* does the computation on the second part. And also at the same time the plugin uses a special feature of the *GPU*, which allows computation and memory copy simultaneously. With this feature, RACE can copy back the previous input slice's results to the CPU memory, while the *GPU* works on the current part. After this and also parallel with the *GPU* computation, RACE copies back the results into RapidMiner's internal representation.

As you can see, with this solution, the *GPU* and the CPU can operate asynchronously and we minimized the time of the memory copies with the use of page-locked memory. Due to the slicing of the input, there is no need for high amount of page-locked memory. This is also a very important part of the optimization. We also used a special feature for the input array to gain performance. It is called write-combining page-locked memory. It frees up the CPU's L1 and L2 cache resources, making more cache available to the rest of the application. As a result it can improve the plugin's performance [5].

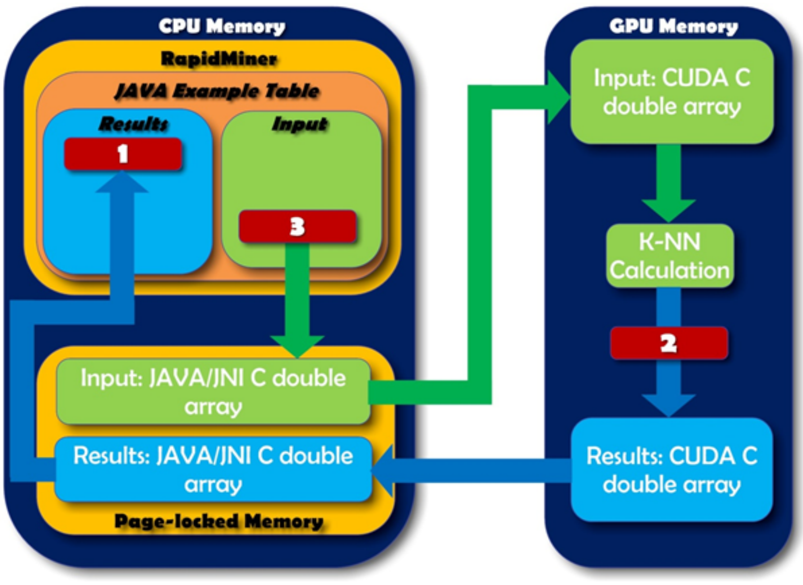
CPU Memory
RapidMiner
JAVA Example Table
Results
Input
1
3
Input: JAVA/JNI C double array
Results: JAVA/JNI C double array
Page-locked Memory
GPU Memory
Input: CUDA C double array
K-NN Calculation
2
Results: CUDA C double array

Figure 4: Asynchronous Operation of the Plugin I

Figure 5 illustrates these parallel processes. You can see the computation of the 3 slices of an input file. We marked the operations related to the same slice with the same color.
If you examine the two operations executed on the *GPU*, you will recognize that the utilization of the *GPU* is very high. We have to mention that the copies form CPU to the GPU are synchronous, it is caused by JCUDA, because it is very difficult to execute asynchronous operations through Java Native Interface.

In contrast, when the *GPU* starts the calculation, the CPU starts the preparation of the next slice immediately and after that, it copies back the result of the previous slice and put it back to RapidMiner. With this method we minimized the overhead of the communication between RapidMiner and the *GPU*.

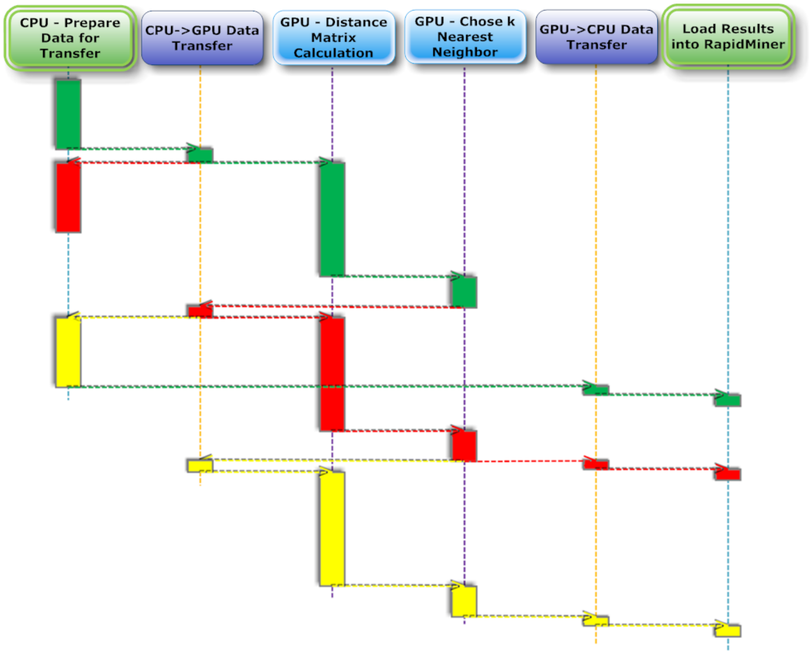
CPU - Prepare Data for Transfer
CPU->GPU Data Transfer
GPU - Distance Matrix Calculation
GPU - Chose k Nearest Neighbor
GPU->CPU Data Transfer
Load Results into RapidMiner

Figure 5: Asynchronous Operation of the Plugin II

5.3 Scalability of the Plugin

As we mentioned in Section 3, we needed multiple kernels for every distance metric because with these RACE can provide high level of performance scalability and flexibility. We created two decision trees for the current two major GPU architecture families. These trees help RACE to select the fastest kernel for the concrete distance metric and size of input. This selection is based on the size of the training set and number of the attributes. The Fermi architectures decision tree can be observed in Figure 6.

The theoretical basis of these decisions is very complex. For example, notice that, the Euclidean and the Mixed Euclidean distance (Euclidean distance for

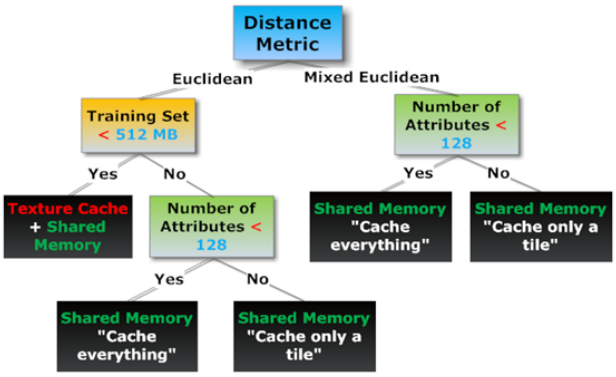
Distance Metric
Euclidean
Mixed Euclidean
Training Set < 512 MB
Number of Attributes < 128
Yes
No
Yes
No
Texture Cache + Shared Memory
Number of Attributes < 128
Shared Memory "Cache everything"
Shared Memory "Cache only a tile"
Yes
No
Shared Memory "Cache everything"
Shared Memory "Cache only a tile"

Figure 6: Decision-tree of the Fermi GPU architecture

numerical and nominal values) branches are different because Mixed Euclidean distance has higher computational complexity and inside the kernels, it has different data dependency. Our previous paper gives detailed explanation for these decisions [3].

6 Structure of the *GPU* k-NN operator

In this field, our goal was to allow the user to use the *GPU* operator like the CPU version. Due to this goal the user does not have to learn any new mechanism. And there is another beneficial feature which is related to the correct use of the Example Table. The *GPU* and CPU operators can work together heterogeneously. For example the user can preprocess the data on the CPU with a built-in RapidMiner operator, use the *GPU* operator and after that they can easily visualize the results on the CPU with a built-in tool.

Figure 7 shows the *kNNLearner* Java class, which realizes the CPU version of the *k-NN* operator. Its most important function is called *learn()*. It creates an object which implements the *Model* interface, from its *Example Set* parameter and sends it to the output of the operator.

Figure 8 shows that *PredictionModel* defines the *Model* interface's *apply()* method. It calls an abstract function called *performPrediction()*, which was defined by *KNNClassificationModel*.

The previously shown *Apply Model* operator calls this *apply()* method and pass the Test set from the operators input as a parameter. Based on this knowledge we realized that we just had to create a new *GPU* specific model class, which also implements the *Model* interface, so it has an *apply()* method,

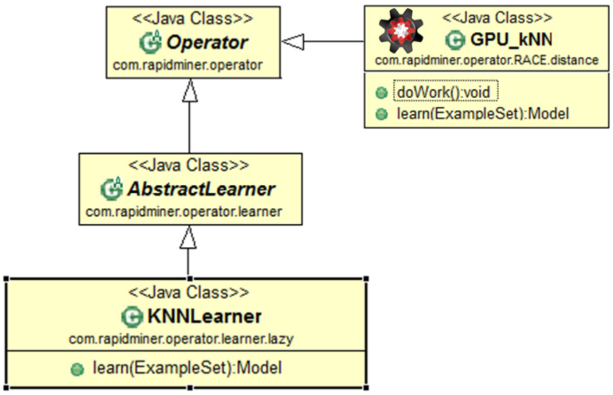

Figure 7: CPU and GPU versions of the k-NN Operator

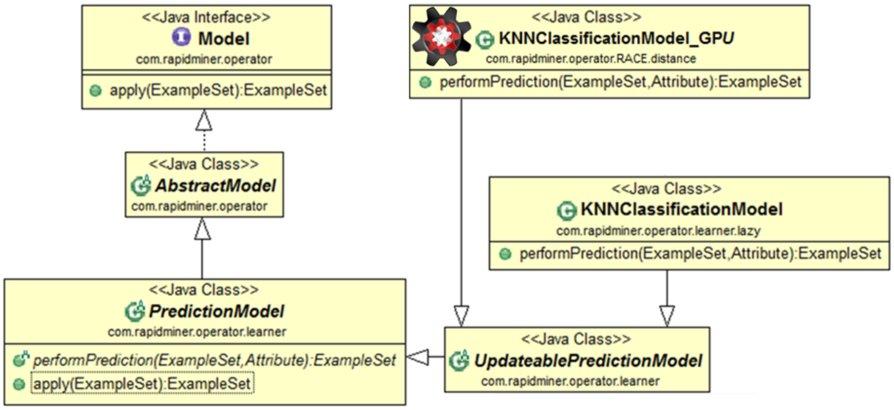
<<Java Interface>>
Model
com.rapidminer.operator
apply(ExampleSet):ExampleSet
<<Java Class>>
KNNClassificationModel_GPU
com.rapidminer.operator.RACE.distance
performPrediction(ExampleSet,Attribute):ExampleSet
<<Java Class>>
AbstractModel
com.rapidminer.operator
<<Java Class>>
KNNClassificationModel
com.rapidminer.operator.learner.lazy
performPrediction(ExampleSet,Attribute):ExampleSet
<<Java Class>>
PredictionModel
com.rapidminer.operator.learner
performPrediction(ExampleSet,Attribute):ExampleSet
apply(ExampleSet):ExampleSet
<<Java Class>>
UpdateablePredictionModel
com.rapidminer.operator.learner

Figure 8: CPU and GPU version of the k-NN Model

which starts the execution on the *GPU*. It is also visible in Figure 8. After we created this model, we just needed to create an operator. It builds up this *GPU*-specific model and gives it to the classic *Apply Model*. Its name is *"GPU k-NN"* and it is shown in Figure 7.

Figure 9 illustrates the final result. As you can see the users can interact with it absolutely like the CPU version, but the next section will present the performance difference between the CPU and the *GPU* operator.

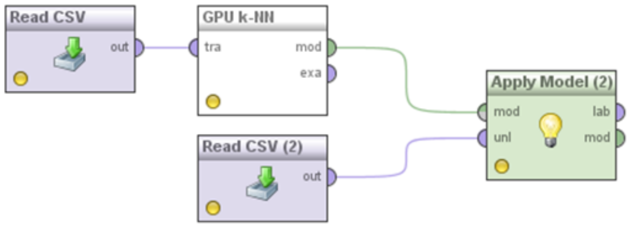

Figure 9: The *GPU* version of the k-NN operator on the graphical user interface of the RapidMiner

7 Experimental Results

Finally, we present the summarized execution times of the operators in Figure 2 and 9. We used a Geforce 480 GTX Nvidia *GPU* for the measurements, which is a member of the Fermi architecture family. We compared its performance to RapidMiner's built-in k-NN operator, running on an Intel Quad Core q8400 CPU. The distance metric was *"Mixed Euclidean"*, the number of test and train elements were equal and the number of attributes was 50 for all of the measurements. On the *GPU* we used the fastest kernel of the decision tree. The results are visible in Table 1.

Table 1: Comparison of an Nvidia *GPU* and a Quad-Core CPU

Number of elements	5000	10000	50000	100000
GPU k-NN Computation time (s):	0.822	1.673	8.1	20.65
CPU k-NN Computation time (s):	11	62	1334	3533
Speedup (x):	**13.38**	**37.05**	**164.69**	**171.09**

8 Conclusion

As we can see in the previous section, the plugin can be up to ***170 times faster*** than the CPU version depending on the type of the *GPU* and CPU. We successfully created a very robust and flexible solution which scales very well when the size of the input varies by several orders of magnitude and perform well compared with the possibilities.

In addition, it should also be mentioned that we did not change anything on the user interface. So the users can use the GPU version exactly like the CPU version without any *GPU*-specific knowledge.

9 Future Plans

We would like to create a *GPU* extension platform from RACE. It will offer a very wide variety of functions which will make the integration of an existing *GPGPU* application a lot easier. Like the current solution, it will help the developers to create standard *GPU* operators, which are able to cooperate with CPU operators and other *GPU* operators.

It will be very useful for the developers, who want to create an easy-to-use *GPU* software, but they do not want to write a totally new user interface for it, and due to the standard interfaces and data handling of their software will be able to cooperate with other *GPU* programs.

We intend to release the first version of this platform in the third quarter of 2012, with the k-NN implementation presented in this paper.

References

[1] GARCIA, V., Debreuve, E., BARLAUD, M. Fast k Nearest Neighbor Search using GPU, Universite de Nice-Sophia Antipolis, Sophia Antipolis, France, 2008.

[2] KUANG, Q., ZHAO, L., A Practical GPU Based KNN Algorithm, Proceedings of the Second Symposium International Computer Science and Computational Technology(ISCSCT '09), Huangshan, P. R. China, 26-28,Dec. 2009, pp. 151-155

[3] KOVÁCS, A., Robust GPGPU plugin development for data mining applications, POSTER 2012, 16th International Student Conference on Electrical Engineering, Prague May 17

[4] YAN, Y.,GROSSMAN, M., SARKAR, V. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA, Euro-Par 2009 Parallel Processing, Volume 5704. ISBN 978-3-642-03868-6.

[5] Nvidia Cuda C Programming Guide, Version 4.1, 11/18/2011

[6] Approaching Vega: The final descent: How to extend RapidMiner 5.0