Performance improvement of data mining in Weka through GPU acceleration

Tiago Augusto Engel\textsuperscript{a}, Andrea Schwertner Char\~ao\textsuperscript{a,}\textsuperscript{*}, Manuele Kirsch-Pinheiro\textsuperscript{b}, Luiz-Angelo Steffenel\textsuperscript{c}

\textsuperscript{a}Universidade Federal de Santa Maria, Santa Maria, Brazil
\textsuperscript{b}Université Paris 1 Panthéon-Sorbonne, Paris, France
\textsuperscript{c}Université de Reims Champagne-Ardenne, Reims, France

Abstract

Data mining tools may be computationally demanding, so there is an increasing interest on parallel computing strategies to improve their performance. The popularization of Graphics Processing Units (GPUs) increased the computing power of current desktop computers, but desktop-based data mining tools do not usually take full advantage of these architectures. This paper exploits an approach to improve the performance of Weka, a popular data mining tool, through parallelization on GPU-accelerated machines. From the profiling of Weka object-oriented code, we chose to parallelize a matrix multiplication method using state-of-the-art tools. The implementation was merged into Weka so that we could analyze the impact of parallel execution on its performance. The results show a significant speedup on the target parallel architectures, compared to the original, sequential Weka code.

© 2014 The Authors. Published by Elsevier B.V.
Selection and peer-review under responsibility of the Program Chairs.

Keywords: data mining tools; parallel computing; GPU

1. Introduction

In the new era of data-intensive science, data mining techniques receive more and more attention. Nowadays, data analysts can rely on a broad spectrum of tools, ranging in functionality, scope and target computer architectures. In the open source scenario, we find, at one side, some recently appeared tools like Mahout and Vowpal Wabbit, which can perform big data analytics in large computer clusters. On the other hand, we have desktop-based tools like R, Weka and RapidMiner, which are usually employed in smaller yet important problems.

Considering the computational complexity of some data mining algorithms, data analytics may take hours to complete. Different alternatives can be considered in order to improve data analysis performance. While some users will rely on cloud-based solutions, heterogeneous environments based on GPU architectures appear as a valuable solution to improve performance with significant cost saving\textsuperscript{1,2}. As pointed out by Schadt et al.\textsuperscript{1}, using these environments, data can be stored and analyzed locally without requiring a specialized structure. This represents an interesting asset...
when considering access control and privacy problem that may concern analyzed data. GPU architectures represent then a complementary solution face to cloud environments. Nevertheless, several authors underline the complexity of using such architectures that often require significant expertise on GPU programming and related technologies. Such complexity may become an important limitation due to the growing popularity of data analytics for non-technical users (for Business Intelligence or for live sciences, for instance).

Currently, desktop-based tools do not usually take full advantage of parallel architectures to speed up data analysis, even though multi-core processors and GPUs are increasingly common on desktops. Most approaches to parallel data mining focus on distributed execution of multiple experiments, or specific parallel algorithms that are not integrated into popular data mining tools. Indeed, computational power offered by new GPU architectures remains almost unexplored in most popular desktop data mining tools.

In order to demonstrate the interest of GPU architectures for improving data analysis performance on desktop data mining tools, we propose, in this paper, an exploratory work in which we adapt a popular tool named Weka (Waikato Environment for Knowledge Analysis) to a GPU use. Weka is a popular open source tool for data mining, and it comprises a collection of data mining and machine learning algorithms packed together in a feature-rich toolbox, which includes preprocessing and analysis tools. Weka provides users with a Graphical User Interface and a Java-based Application Programming Interface. The Weka API reveals a number of packages containing abstract and concrete classes. It implements algorithms for regression, classification, clustering, association rule mining and attribute selection. With all these features, Weka is widely used in business, research and education.

Parallel and distributed computing was not a concern in the design of Weka, but it has been addressed by a few Weka-related projects, as for example Weka-Parallel and Grid Weka. Furthermore, the development branch of Weka (3.7) includes a package called WekaServer, which enables multiple servlet-driven process instances that can run multiple tasks in parallel. Even so, parallel processing with GPUs is still underexploited by Weka.

In this paper, we explore further opportunities for parallel execution within Weka. Our main goal is to improve the performance and reduce the end-user response time of Weka on a single machine, taking advantage of GPU acceleration. To do so, we carried out an in depth study of Weka source code and analyzed its performance using Java profilers. After that, we focused on the parallelization of one of its hotspots – matrix multiplication – using state-of-the-art tools for GPU programming in Java.

The rest of the paper is organized as follows: Section 2 presents related work on parallel execution of Weka algorithms. Section 3 discusses the profiling of Weka and presents some hotspots found on Weka classifiers. Section 4 describes our solution for replacing the sequential Weka matrix multiplication code by a parallel, GPU-accelerated code. Section 5 presents our experiments and results with this parallel implementation. Finally, in Section 6 we draw some conclusions and describe our next steps of research.

2. Related Work

Several initiatives to improve performance of data mining software can be found on the literature. Often such works focus on distributed architectures, such as cluster and grids. Unfortunately, such architectures involve significant equipment and maintenance costs. They may also be concerned by data transfer problems and issues related to data access. GPU architectures appear then as a complementary solution offering improved cost performance ratio without requiring any specialized infrastructure. Multiple works in the literature propose improving performance of data analysis thanks to GPU architectures. Among them, several works propose programming interface, modules or frameworks for programming dedicated application for data mining. Kumar et al., for instance, propose a framework based on R for building data mining tools over heterogeneous architectures. Ma and Agrawal propose a framework for translating data intensive application to GPU cluster. Similarly, Jiang and Agrawal proposes a framework focusing particularly on building map reduce application over GPU architectures. On those works, data analysis is performed by dedicated code particularly designed for these frameworks.

Indeed, only a few initiatives involve improving the performance of final users data mining tools by considering GPU architectures. Weka is a good example of this kind of tools and of this issue. Most of works improving
Weka performance have focused on distributed architectures. GridWeka\textsuperscript{7} leverages the power of grid computing to accelerate data mining tasks. The two main components of this system are Weka server and Weka client. The server is based on the original Weka. Each machine participating in a Weka grid runs the server component. The Weka client is responsible for accepting a learning task and input data from a user and distributing the work on the grid. In this system, a set of data mining tasks can be distributed across several machines in an ad-hoc environment. Tasks that can be executed using GridWeka include: building a classifier on a remote machine, labeling a dataset using a previously built classifier, testing a classifier on a dataset, and cross-validation.

Weka4WS\textsuperscript{13} is an application that extends Weka to perform data mining tasks on WSRF enabled grids. The rst prototype of Weka4WS has been developed using the Java Web Services Resource Framework (WSRF) library provided by GT4. The goal of Weka4WS is to support remote execution of data mining algorithms in such a way that distributed data mining tasks can be concurrently executed on decentralized nodes on the grid, exploiting data distribution and improving performance. Each task is managed by a single thread and therefore a user can start multiple tasks in parallel, taking full advantage of the grid environment.

WekaG\textsuperscript{8} is another application that performs data mining tasks on a grid. It extends the data mining toolkit Weka. WekaG implements a vertical architecture called Data Mining Grid Architecture (DMGA), which is based on the data mining phases: preprocessing, data mining and post-processing. The application implements a client/server architecture. The server side is responsible for a set of grid services that implement the different data mining algorithms and data mining phases. The client side interacts with the server and provides a user interface which is integrated in the Weka interface (without modifying Weka itself).

Using another approach, Senger\textsuperscript{9} presents and evaluate Inhambu, a distributed object-oriented system that supports the execution of data mining applications on clusters of PCs and workstations. This system provides a resource management layer, built on the top of Java/RMI, which supports the execution of Weka. The results are compared with those achieved by another initiative called Weka-Parallel\textsuperscript{6}. Inhambu provides additional advantages such as application checkpointing, support for dynamic aggregation of hosts to the cluster, automatic restarting of failed tasks, and a more effective usage of the cluster.

In the multithreading scenario, Fang\textsuperscript{14} has introduced GPUMiner, a parallel data mining system that utilizes new-generation graphics processing units (GPUs). The software was designed focusing on three main components: a GPU-CPU buffer manager, a co-processing parallel mining module and a GPU-based viewing module. GPUMiner implemented a k-means clustering and the Apriori frequent pattern mining algorithms. The preliminary results have shown significant speedups over state-of-the-art CPU implementations.

In spite of these efforts to distribute Weka across a cluster or a grid, no real advance was made to effectively improve the usage accelerators on a single machine. Indeed, the current version of Weka only includes the possibility to launch a different task for each core, through the use of a client-server model (WekaServer package). While this "batch" mode allows the deployment of tasks over a multi-core machine or even a cluster of machines, each single task remains bounded to a single computing core. In our work, we try to improve the performance of Weka tasks by providing GPU acceleration on key elements of Weka. We believe that this effort contributed to the overall performance of Weka, as both single tasks and multiple experiments in parallel shall benefit from this acceleration.

3. Profiling Weka

Profilers are tools that allow the gathering and analysis of execution characteristics of a program. One of the main uses of profilers is to evaluate the different software elements and to detect which parts of the code are more computing intensive or introduce slowdowns. This can be helpful as a starting point to software parallelization.

To identify the most computing intensive procedures on Weka – "hotspots" – we focused on a subset of algorithms that are common to several Weka applications. We selected the MSP\textsuperscript{15} algorithm, which is responsible for the creation of decision trees that will be used in regression models. This algorithm was suggested by experts in data mining, which had used the MSP in a previous job\textsuperscript{16}.

Several profilers for Java are available, with different strengths and strategies to collect information about a running program. For this reason, the results of different profilers may vary\textsuperscript{17}. It also worth note that the use of profilers is an intrusive approach that slows-down considerably the execution of a program, diluting the impact of external factors like I/O and network accesses. Having that in mind, this work combined the analysis from different profiling tools,
merging the results and focusing on the methods that most appeared. The tools we used are VisualVM\textsuperscript{18}, JProfiler\textsuperscript{19}, JProbe\textsuperscript{20} as well as Java own integrated profiler (jprof).

As expected, the performance and results from different profilers varied a lot. Indeed, VisualVM was exceptionally intrusive and slow, preventing the collection of any useful data. JProfiler had a better behavior than VisualVM, showing a lower overhead and allowing the detection of some hotspots as illustrated in Fig. 1. JProbe had the best behavior concerning the profiling overhead, and it highlighted several hotspots pointed by JProfiler. Finally, the Java profiler showed useful, even though it does not show a lot of detailed information.

<table>
<thead>
<tr>
<th>Hotspot</th>
<th>Inherent time</th>
<th>Average Time</th>
<th>Invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>weka.classifiers.functions.LinearRegression</td>
<td>5 159 s (21 %)</td>
<td>150 ms</td>
<td>34 240</td>
</tr>
<tr>
<td>weka.core.DenseInstance</td>
<td>4 883 s (20 %)</td>
<td>2 µs</td>
<td>2 147 483 647</td>
</tr>
<tr>
<td>weka.core.matrix.Matrix</td>
<td>4 366 s (18 %)</td>
<td>4 µs</td>
<td>1 058 482 902</td>
</tr>
<tr>
<td>weka.core.matrix.LinearRegression</td>
<td>3 860 s (16 %)</td>
<td>72 773 µs</td>
<td>53 052</td>
</tr>
<tr>
<td>weka.core.AbstractInstance</td>
<td>2 916 s (12 %)</td>
<td>2 µs</td>
<td>1 201 282 738</td>
</tr>
<tr>
<td>weka.core.Matrix</td>
<td>2 164 s (9 %)</td>
<td>1 µs</td>
<td>1 337 513 053</td>
</tr>
<tr>
<td>weka.core.matrix.LUDecomposition</td>
<td>1 154 s (0 %)</td>
<td>2 914 µs</td>
<td>53 052</td>
</tr>
</tbody>
</table>

Fig. 1. Example of hotspot detection provided by JProfiler

In addition to the overhead caused by the profiling applications, another important factor that must be considered relates to the choice of MSP parameters. Indeed, this algorithm allows different parameter combinations when analyzing the dataset, and we noticed differences between the profiling results when these parameters were changed. Therefore, the methods that caused most impact in the performance were chosen by crossing the results of the different parameters with the output from different profilers, resulting in the list illustrated in Table 1.

Table 1. Hotspots identified in Weka

<table>
<thead>
<tr>
<th>Method</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>weka.core.matrix.Matrix.times</td>
<td>Matrix Multiplication</td>
</tr>
<tr>
<td>weka.core.matrix.LinearRegression.calculate</td>
<td>Correlation</td>
</tr>
<tr>
<td>weka.core.Instances.quickSort</td>
<td>Sort</td>
</tr>
<tr>
<td>weka.classifiers.functions.LinearRegression.findBestModel</td>
<td>Regression model search</td>
</tr>
<tr>
<td>weka.core.matrix.LUDecomposition.solve</td>
<td>Linear system resolution</td>
</tr>
<tr>
<td>weka.classifiers.Evaluation.evaluateModel</td>
<td>Model evaluation</td>
</tr>
</tbody>
</table>

Most of these methods are well-known to the parallel computing community and can be ported to GPUs. In this work we decided to evaluate the impact of GPUs through the parallelization of a single method, the Matrix Multiplication (\texttt{weka.core.matrix.Matrix.times}), as this method was presents a high number of calls with a non-negligible impact on the time they consume. Also, this matrix multiplication method is not only used by MSP but also by other classifier algorithms within Weka, so its parallelization may improve the overall Weka’s performance.

4. GPU implementation

Since Weka is written entirely in Java, and since Java does not support GPU devices, we had to find ways to do so. The Java policy of "Write once, run everywhere" makes difficult to implement such feature due to several difficulties such as hardware detection and bytecode adaptation at runtime. Therefore, the simplest way is to use Java’s native interface to call CUDA (Compute Unified Device Architecture) functions in C. There are several initiatives enabling Java access to GPU devices. We can cite Aparapi\textsuperscript{21}, JCudaMP\textsuperscript{22}, JaBEE\textsuperscript{23} and JCuda\textsuperscript{24}.

The toolkit we chose was JCuda\textsuperscript{24}. JCuda uses the Java Native Interface (JNI), creating an interface for Java programmers to access the CUDA programming toolkit. The syntax is very similar to the original CUDA\textsuperscript{25} and, as in CUDA, the programmer writes a serial program that calls parallel kernels, which may be simple functions or full programs. Java native bindings are used to launch the kernel, which executes in parallel across a set of parallel threads.

JCuda is a stable software suite that provides an extension to CUDA Basic Linear Algebra Subroutines (CUBLAS), called JCublas\textsuperscript{20}, which we found useful to adapt the matrix multiplication (sequential) to a GPU-enabled one. Indeed,
JCu blas provides a GPUs method cuBlasDgemm, which calculates matrix multiplications using double precision according to the framework presented on Algorithm 1.

**Algorithm 1:** The basic framework of DGEMM routines

- **Input:** Size of thread block: $vl \times vly$
- **Input:** Input matrix: $A[mk], B[kn]$
- **Data:** Registers: $rA[rx], rB[ry]$
- **Data:** $acc[0...rx][0...ry]$ // $rx \times ry$ is a factor of register blocking
- **Data:** Shared mem: $smA[bk][bm], smB[bk][bn]$
- **Result:** $C[m \times n]$

1. $acc[0...rx][0...ry] = 0$
2. Load one $bm \times bk$ block of $A$ into $smA[bk][bm]$
3. Load one $bk \times bn$ block of $B$ into $smB[bk][bn]$
4. Synch
5. While $(k/bk) > 0$
   6. For $ki = 0$ to $bk$
      7. Load one column of $A$ in $smA[0...rx]$
      8. Load one row of $B$ in $smB[0...ry]$
      9. $acc[0...rx][0...ry] += rA[0...rx] \times rB[0...ry]$
   10. End
   11. Load one $bm \times bk$ block of $A$ into $smA[bk][bm]$
   12. Load one $bk \times bn$ block of $B$ into $smB[bk][bn]$
   13. Synch
14. End
15. Merge $acc[0...rx][0...ry]$ with $bm \times bn$ block of $C$.

Please note that not all matrix multiplications are fit for GPU acceleration. Hence, we found that M5P uses mostly rectangular matrices of different sizes, and sometimes these matrices are too small to benefit from GPUs due to the overhead of data transfer to and from the GPU. Having that in mind, we created a mechanism to identify when the matrix size would benefit from GPU execution, i.e., when there is enough work to be done in parallel without suffering from the data transfer overhead. This mechanism identifies the size of the matrix and, according to its size, calls the parallel matrix multiplication or the sequential multiplication.

5. Experiments

To conduct these experiments, we executed the sequential and the GPU-improved version of Weka on three different machines. These machines contain different processors and NVIDIA GPUs, which allow us to infer on the contribution of both CPUs and GPUs to the overall performance of Weka. In all experiments and developments, we used Weka 3-7-7 developer version with Java(TM) SE Runtime Environment (build 1.7.0-09), and the machines run Debian 2.6.32. Please note that the ROMEO node is part of the ROMEO cluster†, a new platform that entered at the 151st position of the last TOP500‡ ranking (and 5th position at the Green500§ ranking). All the characteristics are presented in Table 2.

The dataset used in the experiments is entitled Relative location of CT slices on axial axis, and can be obtained at UCI Data Mining Repository. In this dataset, we used to predict the relative location of CT (Computed Tomography) slices on the axial axis using k-nearest neighbor search. Also used the data to apply weighted combinations of image features for the localization of small sub volumes in CT scans. The dataset has 386 continuous attributes and 53500 instances. It is a relatively large dataset for desktop-based data mining tools.

In order to evaluate the impact of GPUs on Weka, our experiments considered three different approaches:

† https://romeo.univ-reims.fr
‡ http://www.top500.org
§ http://www.green500.org
Table 2. Characteristics of the nodes used in our experiments

<table>
<thead>
<tr>
<th>GPU Configuration</th>
<th>LSC4</th>
<th>DoubleKITT</th>
<th>ROMEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Model</td>
<td>Tesla M2050</td>
<td>2 x Quadro K2000</td>
<td>2 x Tesla K20Xm</td>
</tr>
<tr>
<td>CUDA Driver Version</td>
<td>4.2</td>
<td>5.5</td>
<td>5.5</td>
</tr>
<tr>
<td>CUDA Capability</td>
<td>2.0</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>GPU Clock rate</td>
<td>1.15 GHz</td>
<td>0.95 GHz</td>
<td>0.73 GHz</td>
</tr>
<tr>
<td>CUDA Cores (thread processors)</td>
<td>448</td>
<td>384</td>
<td>2688</td>
</tr>
<tr>
<td>Maximum number of threads per multiprocessor</td>
<td>1536</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Memory Clock rate</td>
<td>1546 Mhz</td>
<td>2000 Mhz</td>
<td>2600 Mhz</td>
</tr>
<tr>
<td>Memory Bus Width</td>
<td>384-bit</td>
<td>128-bit</td>
<td>384-bit</td>
</tr>
<tr>
<td>Total amount of global memory</td>
<td>3072 MBytes</td>
<td>2048 MBytes</td>
<td>6144 MBytes</td>
</tr>
</tbody>
</table>

CPU Configuration

<table>
<thead>
<tr>
<th>Processor Model</th>
<th>LSC4</th>
<th>DoubleKITT</th>
<th>ROMEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>4</td>
<td>2x6</td>
<td>2x8</td>
</tr>
<tr>
<td>CPU Clock Rate</td>
<td>2.4 GHz</td>
<td>2.1 GHz</td>
<td>2.6 GHz</td>
</tr>
</tbody>
</table>

1. Sequential execution (Weka original behavior)
2. GPU-only (using JCublas)
3. A mixed code that selects the best approach (GPU or sequential) according to the matrices sizes.

Table 3 presents execution times for the three scenarios. To perform the tests, the number of folds was fixed in two and we compared different values for the number of attributes per leaf (-M). This parameter orients how the model tree (a regression tree) is built, by setting the minimum number of instances to allow at a leaf node. A small value of M produces a larger tree, favoring parallelization, while a large value of M produces a more compact tree, better adapted to sequential computation. By default, Weka sets the value of M to 4.

Table 3. Execution times of M5P on LSC4 with different approaches and values of M

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Sequential</th>
<th>GPU</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>-M 4</td>
<td>734.82s</td>
<td>289.01s</td>
<td>276.15s</td>
</tr>
<tr>
<td>-M 8</td>
<td>728.31s</td>
<td>286.40s</td>
<td>274.82s</td>
</tr>
<tr>
<td>-M 16</td>
<td>651.18s</td>
<td>274.52s</td>
<td>262.74s</td>
</tr>
<tr>
<td>-M 32</td>
<td>603.17s</td>
<td>266.84s</td>
<td>256.61s</td>
</tr>
<tr>
<td>-M 64</td>
<td>451.29s</td>
<td>240.47s</td>
<td>233.26s</td>
</tr>
<tr>
<td>-M 128</td>
<td>338.54s</td>
<td>215.46s</td>
<td>210.69s</td>
</tr>
</tbody>
</table>

The GPU implementation improved considerably the M5P’s performance on LSC4, as can be seen in Fig. 2. As less the number of attributes per leaf (-M), more of them are needed, augmenting the height of the tree. When the tree is higher, the matrix multiplication algorithm is called more times and the speedup is more significant. On the other hand, when the tree is shorter, the accelerated matrix multiplication method was not called so many times, and its impact in the overall performance was less significant but yet considerable. The mixed approach showed itself the fastest approach, as the use of GPU-only brings an unnecessary overhead when the matrices are too small.

While the performance of DoubleKITT can be compared to that of LSC4, we observe that in ROMEO the improvement is much more important (Fig. 3). Although this node presents a better sequential performance than the other machines as it uses a new generation of CPUs, the GPU accelerated Weka presents a speedup of at least 89%. This improvement (compared to 49% on DoubleKITT or 60% on LSC4) indicates that the next generations of GPUs shall keep improving their efficiency, and therefore bringing a massive computing power to the user.

6. Conclusion

The popularization of GPUs represents new opportunities for software parallelization and performance improvement accessible to the final user on its own desktop. Data mining is one of the application fields that can benefit from
In this paper, we presented the results of a first experience to improve the performance of Weka data mining software, a well-known data mining tool. Thanks to the use of Java profilers, we identified a set of operations that are time-consuming and that can easily be adapted to GPUs. In this first experiment we parallelized the Matrix Multiplication method, adapting the work method to take advantage of CPUs or GPUs according to the size of the matrices. As a result, we observed speedup levels of at least 49%, drastically decreasing the time the algorithm consumes to handle a dataset.

Future works shall continue the parallelization of other Weka algorithms on GPUs, but also improve the efficiency of CPUs on Weka, which for the moment is not able to take all the advantages of multi-core architectures.

The last version of our GPU-based Weka implementation can be found at https://github.com/tiago11/MGPWeka.

Acknowledgments

The authors would like to thank the LSC Laboratory and the ROMEO Computing Center for the access to their resources. This project is partially financed by the STIC-AmSud PER-MARE project (project number 13STIC07), an international collaboration program supported by CAPES/MAEE/ANII agencies.

http://cosy.univ-reims.fr/PER-MARE
References


