ÆminiumGPU: An Intelligent Framework for GPU Programming

Alcides Fonseca[1] and Bruno Cabral[1]

University of Coimbra, Portugal
{amaf,bcabral}@dei.uc.pt

Abstract. As a consequence of the immense computational power available in GPUs, the usage of these platforms for running data-intensive general purpose programs has been increasing. Since memory and processor architectures of CPUs and GPUs are substantially different, programs designed for each platform are also very different and often resort to a very distinct set of algorithms and data structures. Selecting between the CPU or GPU for a given program is not easy as there are variations in the hardware of the GPU, in the amount of data, and in several other performance factors.
ÆminiumGPU is a new data-parallel framework for developing and running parallel programs on CPUs and GPUs. ÆminiumGPU programs are written in a Java using Map-Reduce primitives and are compiled into hybrid executables which can run in either platforms. Thus, the decision of which platform is going to be used for executing a program is delayed until run-time and automatically performed by the system using Machine-Learning techniques.
Our tests show that ÆminiumGPU is able to achieve speedups up to 65x and that the average accuracy of the platform selection algorithm, in choosing the best platform for executing a program, is above 92%.

Keywords: Portability, Parallel, Heterogeneous, GPGPU

1 Introduction

Since Graphics Processing Units (GPUs) have been user-programmable, scientists and engineers have been exploring new ways of using the processing power in GPUs to increase the performance of their programs. GPU manufacturers acknowledged this alternative fashion of using their hardware, and have since provided special drivers, tools and even models to address this small, but fast-growing niche.

GPUs are interesting to target because of their massive parallelism, which provides a higher throughput than what is available on current multi-core processors. But, one can argue that the difference in architectures also makes programming for the GPU more complex than for the CPU. GPU programming is not easy. Developers that do not understand the programming model and the hardware architecture of a GPU will not be able to extract all its processing power. And, even after a program has been specially designed for the GPU, its performance might still be worse than on the CPU. For instance, the usage of GPUs incurs on a penalty caused by memory copies between the main memory and the GPU-specific memory.

In many situations, it may not be feasible to know beforehand if a program will perform better in a GPU or in a CPU without actually executing it. And, for programs that are not repeatedly executed or that execute for a very long time it may not be useful to do so. Moreover, the performance of a program will commonly be influenced by the size of the input data, the actual GPU hardware and the structure of the program itself.

```java
Double integral = new Range(RESOLUTION).map(new LambdaMapper<
    Integer, Double>() {
  public Double map(Integer input) {
    double n = RESOLUTION;
    double b = Math.pow(Math.E, Math.sin(input / n));
    double B = Math.pow(Math.E, Math.sin((input+1) / n));
    return ((b+B) / 2 ) * (1/n);
  }
}).reduce(new LambdaReducer<Double>(){
  public Double combine(Double input, Double other) {
    return input + other;
  }
});
```

Listing 1.1. Example of Map-Reduce to Calculate the Integral of a Function using the trapezoid method

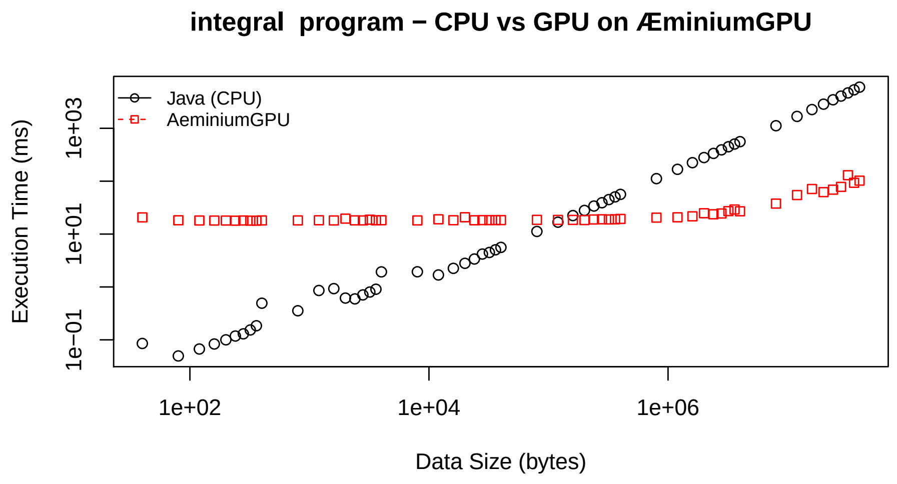

Fig. 1. Performance of the Integral program on CPU and GPU

Listing 1.1 is an example of programs that can execute on the GPU and calculates the integral of $f(x) = e^{sin(x)}$. This is an embarrassingly parallel problem, which is expressed using a data-parallel approach by means of map and reduce operations. Figure 1 shows the execution time of the program in both CPU and GPU for different data sizes. The GPU version is faster after a certain data size and it is able to achieve up to 64 times of speedup. But, note that the threshold from which the GPU performance starts to gain on the CPU is not always the same. The actual threshold value depends of the program logic and even with the hardware being used. Thus the decision whether to run a program on the GPU or CPU is not an easy one.

The goal of this work is to present a new framework which simplifies the task of writing data-parallel programs for transparently executing in GPUs, with improved performance. Our approach drives inspiration from Functional Programming and puts the power of GPUs in the hands of developers without forcing them to understand the particularities of GPU programming. While programs are written in a mainstream programming language using a Map-Reduce approach for now, specific parts of their code are compiled to OpenCL and executed on the GPU. In order to minimize the impact of well known bottlenecks in GPU programming and maximize the speedup obtained by the usage of GPUs, the framework performs several optimizations on the generated code. Such optimizations include the generation of data on the GPU in opposition to its copy from main memory, among others. Furthermore, in ÆminiumGPU, programs are compiled into "hybrid" executables that can run in either GPU and CPU platforms. Since the final purpose of ÆminiumGPU is to execute programs as fast as possible, independently of the platform being used, hybrid executables allow us to delay the decision of which platform is best for executing a specific program until run-time, when much more information is available to fundament a choice. ÆminiumGPU, by means of Machine-Learning techniques, is able to make this decision autonomously with high accuracy.

The contributions of this work are:

- A new and state-of-the-art framework for GPGPU programming, which hides GPU architectural details and execution model from developers;
- A translation library that is able to automatically generate optimized OpenCL code from code written in a mainstream general purpose programming language;
- A machine-learning solution for predicting the efficiency of programs in both CPUs and GPUs;
- And, to the best of our knowledge, the first runtime system using machine-learning techniques for autonomously deciding either to execute programs in the CPU or GPU.

2 Approach

In this section we will depict the architecture and design of the ÆminiumGPU framework. We will use the Map-Reduce algorithm as an example in this section since it is a suitable representative of data-parallel algorithms in general.

2.1 Architecture

The ÆminiumGPU framework was designed for supporting Æminium[1] and Java programming languages. Since Æminium compiles to Java, this paper will present the architecture from the point of view of Java. The Java language is not supported by GPUs. Thus it is necessary to translate Java into OpenCL functions. Translation is performed at compile-time by the ÆminiumGPU Compiler. The OpenCL functions are then executed by the ÆminiumGPU Runtime during execution. The general architecture can be seen in Figure 2.

ÆminiumGPU Compiler The ÆminiumGPU Compiler is a source-to-source compiler from Java-to-Java, in which the final Java code has some extra OpenCL code. The OpenCL code is based on lambda functions present in the source code. For each lambda in the original code, the compiler creates an OpenCL version. This version is later used to generate a kernel which will execute on the GPU.

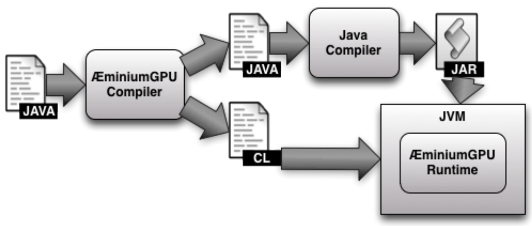

Fig. 2. Architecture of ÆminiumGPU

The compiler was implemented using Spoon, a Java-to-Java compiler framework[2]. Spoon parses and generates the AST and generates the Java code from the AST. The ÆminiumGPU compiler introduces new phases that produce the OpenCL version of existent lambdas. The compiler looks for methods with a special signature, such as map or reduce. The AST of lambdas passed as arguments are then analyzed and a visitor tries to compile Java code to OpenCL as soon as it descends the AST.

It is important to notice that not all Java code can be translated to OpenCL. The ÆminiumGPU compiler does not support all method calls, non-local variables, for-each loops, object instantiation and exceptions. It does support a common subset between Java and C99 with some extra features like static accesses, calls to methods and references to fields of the Math object.

ÆminiumGPU Runtime The ÆminiumGPU Runtime is a Java library responsible for providing Æminium Programs with parallel-ready lists that implement the GPU methods, such as `map` and `reduce` methods. Each list can be associated with a GPU, thus supporting several GPUs on the same machine. Whenever a GPU operation is summoned, the following phases occur: firstly the compiler-generated OpenCL function is inserted in a predefined template (specific for each operation, such as reduce) and the resulting kernel is compiled to the GPU; afterwards, the input data, if required, is copied to the GPU memory; next the kernel execution is scheduled with a suitable `workgroups` and `workitems` arrangement for the data size and operation in question; finally the output data is copied back to the host device and every GPU resource is manually released.

The templates used for Map and Reduce, since we are focusing in these operations for this work, are really straightforward. The map kernel only applies a function to an element of the input array and writes it to the output array. The reduce kernel is a generic version of NVIDIA's implementation[3], allowing for more data-types than the four originally supported.

For these operations in particular, one optimization already implemented is the fusion of maps with maps, and maps with reduces. This optimization is done by considering the Map operation a lazy operation that is only actually performed when the results are needed. This laziness allows for merging together several operations, saving time in unnecessary memory copies and kernel calls. Because of this optimization, the final kernel is only known and compiled at runtime.

All operations, even the ones that cannot be translated to the GPU, have a sequential version written in Java. For the original purposes of this framework, it was not important to parallelize on the CPU, but it will be considered in future work, and the same technique can be used.

2.2 GPU-CPU Decision

ÆminiumGPU uses Machine Learning techniques to automatically decide if a given operation should be executed on either the GPU or CPU. The problem can be described as two-classed because each program execution can be classified as either *Best on GPU* or *Best on CPU*. Supervised learning will be used, since it is important to associate certain features of programs to the two platforms.

Since decisions are hardware dependent (CPU and GPU combination), we considered two ways for tackling the problem: training the classifier in each machine; or considering CPU and GPU specifications as features in a general classifier. The former was selected for this work, although it can be extended to the later in the future. Using a large number of features would increase classification time and it would be a very hard to train a general classifier with a large set of CPU and GPUs. This means that when installing ÆminiumGPU, it is necessary to execute a standard benchmark for collecting training data.

The critical aspect for having a good classification is choosing the right features to represent programs. For instance, it is not feasible to consider the full program in ASCII, since the length would be variable and the abstraction level ill-suited for classification techniques. Table 1 lists all the features used in the classification process.

```
a(); // Level 1
for (int i=0; i<10; i++) {
  b(); // Level 2
  while ( j < 20)
    c(); // Level 3
}
```

Listing 1.2. Examples of Level categorization

Features can be extracted either during compilation or during runtime. This means that a given program will always hold the same values for the first features, while the last three features may be different, depending on the conditions of execution. Features marked with a size of 3 have three values, one for each depth of loop scopes. Listing 1.2 shows an example in which three functions are considered in 3 different loop levels. This distinction is important since operations in inner levels are executed more times than ones in the outer levels.

The choice of some selected features was inspired by other applications of Machine Learning in this area ([4], [5] and [6]). Memory accesses were considered a feature as they are one of the main reasons why GPU programs are not as fast as one would expect. As such, there are features for all three main kinds of memories in GPUs (global and slow, local and fast, global read-only and fast). Note that some GPU models may not have one of them, but it is still required for other models.

Name	Size	Collected during	Description
OuterAccess	3	Compilation	Global GPU memory read.
InnerAccess	3	Compilation	Local (thread-group) memory read. This area of the memory is faster than the global one.
ConstantAccess	3	Compilation	Constant (read-only) memory read. This memory is faster on some GPU models.
OuterWrite	3	Compilation	Write in global memory.
InnerWrite	3	Compilation	Write in local memory, which is also faster than in global.
BasicOps	3	Compilation	Simplest and fastest instructions. Include arithmetic, logical and binary operators.
TrigFuns	3	Compilation	Trigonometric functions, including *sin*, *cos*, *tan*, *asin*, *acos* and *atan*.
PowFuns	3	Compilation	*pow*, *log* and *sqrt* functions
CmpFuns	3	Compilation	*max* and *min* functions
Branches	3	Compilation	Number of possible branching instructions such as *for*, *if* and *whiles*
DataTo	1	Runtime	Size of input data transferred to the GPU in bytes.
DataFrom	1	Runtime	Size of output data transferred from the GPU in bytes.
ProgType	1	Runtime	One of the following values: Map, Reduce, PartialReduce or MapReduce, which are the different types of operations supported by ÆminiumGPU.

Table 1. List of features

In terms of operations, we performed micro-benchmarks to assess their execution cost. For instance, 4 or 5 *plus* operator calls execute much faster than one single *sin* call. As such, OpenCL functions were grouped according to the relative cost they have on execution time.

Besides these features, each benchmark also collected the execution time in both CPU and GPU, and the class to each execution belongs to. This is used for training and also evaluation.

3 Evaluation and Classifier Selection

In this section we will describe the experiments performed for verifying and validating our approach and to select a classifier to use in the implementation.

3.1 Dataset

Our workload for generating the training and testing dataset is composed by the following 8 programs:

1. A map operation that adds 1 to each element of the input array;
2. A map operation that applies the *sin* function to each element of the input array;
3. A map operation that applies the *sin* and *cosine* functions to each element of the input array and sums the values;
4. A map operation that calculates the factorial for each element of the input array;
5. A map-reduce operation that calculates the integral from 0 to the size of the array for $f(x) = e^{sin(x)}$;
6. A map-reduce operation that calculates the minimum value from 0 to the size of the array for $f(x) = 10x^6 + x^5 + 2x^4 + 3x^3 + \frac{2}{5}x^2 + \pi x$;
7. A map-reduce operation that calculates the sum of all natural numbers up to a given value that are divisible by 7;
8. A map-reduce operation that calculates the sum of all elements of the input array that are divisible by 7.

Each one of these programs was executed several times with varying amounts of input data. The size of input data varies from 10 to 10^7 elements, executing with 10 values for each power of 10, and in each level multiplied by all natural numbers until 9. Thus, the first sizes would be 10,20,30,40,50,... and the last sizes would be $50^6, 60^6, 70^6, 80^6, 90^6, 10^7$. Overall, the dataset has 440 instances of different program executions, from 8 individual programs, each executed with the 55 different data sizes.

3.2 Experimental Setup

These are the specifications of the hardware and software used for the experiments: Intel Core2 Duo E8200 at 2.66GHz; 4GB of RAM memory; NVIDIA GeForce GTX 285, with 240 CUDA cores and 1GB of memory; OS Ubuntu Linux 64bits with the NVIDIA CUDA SDK 5.0 preview 2 with OpenCL 1.1 and OpenJDK 1.7. The results presented here are specific to this particular hardware and software and can not represent all possible combinations.

3.3 Feature analysis

To evaluate features we used two feature ranking techniques: Information Gain and Gain Ratio. Both techniques were applied to the whole dataset. The ranking obtained was different for each method, but both returned 3 groups of features: A first group of high-ranked features, a group of low-ranked features and a third group of unused or unrepresentative features. This later group exists because the dataset programs do not cover all possibilities. But, this does not mean that such features should be ignored, on the contrary, they should be studied for particular examples which are out of the scope of this work. Table 2 shows the two other groups ranked using the Information Gain method.

Rank	Feature	Rank	Feature
0.2606	DataTo	0.172	OutterAccess1
0.2517	DataFrom	0.0637	Branches1
0.1988	BasicOps2	0.0516	InnerAccess1
0.1978	BasicOps1	0.0425	TrigFuns1
0.1978	ProgType	0.0397	InnerWrite2
0.1978	OutterWrite1	0.0397	InnerAccess2

Table 2. Features rank using Information gain

Notice that features related with data sizes are high ranked, which is supported by the high penalty caused by memory transfers. Basic Operations are also very representative, since they are very common, specially in loop conditions (*BasicOps2*). The program type is also important because maps and reduces have a different internal structure. Maps happen in parallel, while parallel reduces are executed with much more synchronization in each reduction level.

Looking at the lower ranked features, it is important to consider that memory accesses also impact the decision. It is also expected that branching conditions would have an impact on the performance of programs. Finally, trigonometric functions do not have such an high impact as basic operations, but they are still relevant for the decision.

3.4 Classifier Comparison

In order to achieve the best accuracy, it is important to choose an adequate classifier. For this task, several off-the-shelf classifiers from Weka[7] were evaluated, and some custom classifiers were also developed. The used classifiers include: a **Random** classifier that randomly assigns either class to a particular instance; **AlwaysCPU** and **AlwaysGPU** that classifies all instances as *Best on CPU* and *Best on GPU*; a **NaiveBayes** Classifier; a Support Vector Machine (**SVM**) obtained from a Sequential Minimal Optimization algorithm with $c = 1$, $\epsilon = 10^{-12}$ and a Polynomial Kernel; a Multi-Layer Perceptron (**MLP**); a **DecisionTable** classifier; and a Cost-Sensitive version of the DecisionTable(**CSDT**) that uses 0.4 as the cost for 0.4 for misclassified *Best on CPU* programs and 0.6 for *Best on GPU* programs.

Besides these classifiers, we also experimented with a regression-based approach using additional metrics such as: *CPUTime* and *GPUTime*. The main idea was to use regression techniques to predict values of *CPUTime* and *GPUTime* for each instance and then select the smallest value. However, regressions have shown to have a poor quality with correlation coefficients between 70 and 80%. The final classifier behaved very similarly with the Random classifier. Thus, we decided to not pursue this line of research further.

Classifiers were evaluated using both 7 and 8 fold cross validation. Data was not randomized and was ordered by program. Since the number of folds is lower than or equal to the number of programs, some programs are not present in all the training sets. This simulates the real-world scenario of classifying programs that were not previously seen. The results with 7 and 8 folds were very similar, as well as the results with randomized data. The results presented from here on are with 7 folds and without randomization.

Figure 3 shows the accuracy distribution of the evaluated classifiers. AlwaysCPU and AlwaysGPU do not have 0.5 of accuracy because programs that are faster on the GPU are larger number on the dataset. This was not balanced on purpose, to reflect the actual distribution of CPU and GPU execution times for the tested programs. The DecisionTable classifier achieved a very high accuracy, only second to its Cost-Sensitive version which had a slightly higher accuracy with a more condensed distribution.

In this problem, the distinction between False Positives and Negatives is not relevant. This may seem to contradict the usage of a Cost-Sensitive Classifier, but the cost of misclassification does not only depend on the class, but also on the size of the data in that execution, according to Figure 1. In order to represent the impact of taking the wrong decision, a measure of cost was introduced to replace the traditional confusion matrix. The cost of a misclassification is the absolute difference between the real GPU and GPU execution times previously measured during the feature extraction.

Figure 4 shows the distribution of the total cost of the classification for each cross-validation execution with a logarithmic scale on the Cost (yy axis). The lowest the cost is, the better. A perfect classifier would have a cost of 0. The random classifier has an average cost of 9.8×10^9, which can be considered as a ceiling for this dataset.

The measure of cost is important because we can see that some classifiers such as NaiveBayes and SVM have a better accuracy but have an higher penalty on performance than the classifier that executes everything on the GPU. The two versions of the DecisionTable classifier were also the ones with the lowest cost. Another evaluation metric was classification time, since it could not be representative in execution time. Except for the NaiveBayes classifiers, all others classified instances in less than 20 microseconds, which is acceptable for this task. The classifier training time was not considered for this study as it is not relevant since it is only performed once per machine.

Looking at all the metrics, the Cost Sensitive version of the DecisionTable classifier was the best, achieving 92% of average accuracy and the lowest misclassification cost.

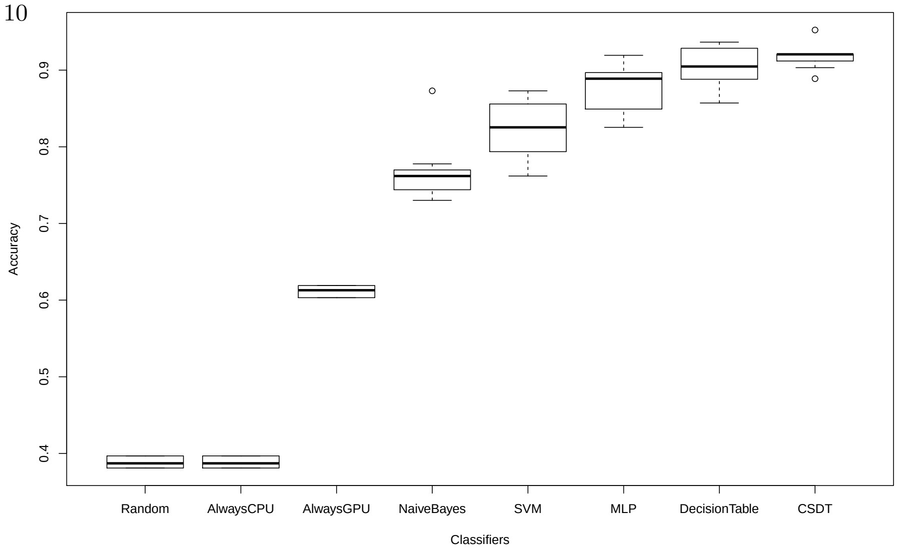

Fig. 3. Box plot of the distributions of accuracy of several classifiers

4 Related Work

There have been several works which can be compared with the ÆminiumGPU framework. There are also approaches that allow to write the kernel code in higher-level languages such as Aparapi[8] (for Java), Copperhead[9] (for Python) and ScalaCL[10] (for Scala) and in X10[11]. ÆminiumGPU is different from these approaches since it provides an interface at an even higher level, as it does not require programmers to write kernels, or know about which code can execute in the GPU or CPU.

Accelerate[12] has a more similar approach in which it also executes higher-level functions over arrays on the GPU. The purity of Haskell makes this somehow easier than in Java. Due to the monadic approach, programmers must type annotate all the code that can execute on the GPU, making GPU Programming less transparent than in ÆminiumGPU.

Both the second version of ScalaCL and JikesVM[13] can convert for loops to OpenCL code and execute it on the GPU. The former uses reflection while the later uses bytecode instrumentation. ÆminimumGPU uses a different approach, using a Source-to-Source compiler to generate the OpenCL code.

MARS[14] and MapCG[15] are two map-reduce frameworks for the GPU and, in the case of the latter, CPU as well with a low-level C API. Both these platforms follow the distributed key-value approach to map-reduce. The overhead of copying both keys and values is significant on the GPU, where every memory transfer counts, in cases where the values are not aggregated by key.

Qilin[16] is a C++ framework that has adaptive mapping in which it tries to record executes of the same program to build a cost model for future executions of the program with different data sizes. ÆminiumGPU also uses previous program executions to build information for future decisions, but it does not require executions of the same program. For programs that only execute a few

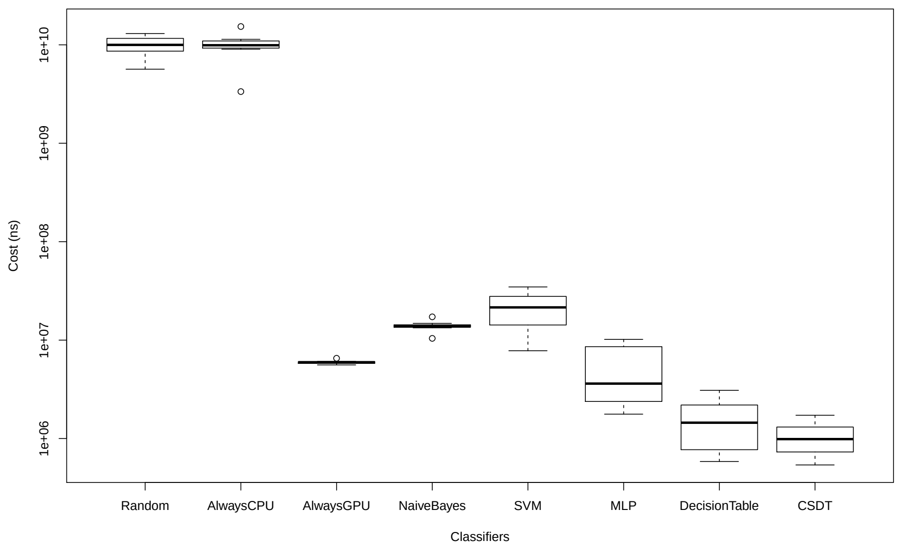

Fig. 4. Box plot of the distributions of costs of several classifiers

times in each machine, the approach used in Qilin is not feasible. There are also approaches for real-time systems[17]. However this work is limited to operations inside a ever-running loop, in which each iteration is schedule to the CPU or GPU according to estimated time, based on previous runs.

5 Conclusions and Future Work

The Æminium framework tries, as much as possible, to optimize the generated code and to schedule operations to the GPU. In many situations, performance increases as soon as the size of the input data goes above a certain threshold. But, since this value is program-dependent, ÆminiumGPU uses a Machine-Learning approach to decide which platform offers more guarantees of providing the best performance. Our tests show that ÆminiumGPU is able to achieve a 92% average accuracy with a low misclassification penalty.

The approach presented is language independent and can be applied to typical HPC languages like C and Fortran, even without using the Map-Reduce pattern. The approach can also work with other accelerators like FPGAs, and improved with specific features for those processors.

Concluding, ÆminiumGPU allows programmers to write data-parallel programs whose performance can, if possible, be improved automatically by using the GPU.

Acknowledgments

This work was partially supported by the Portuguese Research Agency FCT, through CISUC (R&D Unit 326/97) and the `CMU|Portugal` program (R&D Project Aeminium CMU-PT/SE/0038/2008).

References

1. Stork, S., Marques, P., Aldrich, J.: Concurrency by default: using permissions to express dataflow in stateful programs. In: OOPSLA Companion. (2009) 933–940
2. Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program analysis and transformation in java. (2006)
3. Harris, M.: Optimizing parallel reduction in cuda (2010)
4. Russell, T., Malik, A.M., Chase, M., van Beek, P.: Learning basic block scheduling heuristics from optimal data. In: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research. CASCON '05, IBM Press (2005)
5. Cavazos, J., Moss, J.E.B.: Inducing heuristics to decide whether to schedule. SIGPLAN Not. **39**(6) (June 2004) 183–194
6. Wang, Z., O'Boyle, M.F.: Mapping parallelism to multi-cores: a machine learning based approach. In: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. PPoPP '09, New York, NY, USA, ACM (2009) 75–84
7. Holmes, G., Donkin, A., Witten, I.: Weka: A machine learning workbench. In: Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on, Ieee (1994) 357–361
8. Frost, G.: Aparapi. URL: http://code.google.com/p/aparapi/ (2011)
9. Catanzaro, B., Garland, M., Keutzer, K.: Copperhead: Compiling an embedded data parallel language. Principles and Practices of Parallel Programming (PPoPP) (2011) 47–56
10. Chafik, O.: Scalacl. URL: http://code.google.com/p/scalacl/ (2011)
11. Cunningham, D., Bordawekar, R., Saraswat, V.: Gpu programming in a high level language: compiling x10 to cuda. In: Proceedings of the 2011 ACM SIGPLAN X10 Workshop. X10 '11, New York, NY, USA, ACM (2011) 8:1–8:10
12. Chakravarty, M., Keller, G., Lee, S., McDonell, T., Grover, V.: Accelerating haskell array codes with multicore gpus. In: Proceedings of the sixth workshop on Declarative aspects of multicore programming, ACM (2011) 3–14
13. Leung, A., Lhoták, O., Lashari, G.: Automatic parallelization for graphics processing units. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, ACM (2009) 91–100
14. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a mapreduce framework on graphics processors. In: Proceedings of the 17th international conference on Parallel architectures and compilation techniques. PACT '08, New York, NY, USA, ACM (2008) 260–269
15. Hong, C., Chen, D., Chen, W., Zheng, W., Lin, H.: Mapcg: writing parallel program portable between cpu and gpu. In: Proceedings of the 19th international conference on Parallel architectures and compilation techniques. PACT '10, New York, NY, USA, ACM (2010) 217–226
16. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 42, New York, NY, USA, ACM (2009) 45–55
17. Joselli, M., Zamith, M., Clua, E., Montenegro, A., Conci, A., Leal-Toledo, R., Valente, L., Feijó, B., d'Ornellas, M., Pozzer, C.: Automatic dynamic task distribution between cpu and gpu for real-time systems. In: Computational Science and Engineering, 2008. CSE'08. 11th IEEE International Conference on, Ieee (2008)