Πρόβλεψη επίδοσης και κατανάλωσης ισχύος σε ετερογενή συστήματα με χρήση στατιστικών μεθόδων.

Performance and power prediction on heterogeneous systems using statistical methods

by

Spyrou Michalis

A Thesis presented for the degree of
Diploma of Electrical and Computer Engineering

Supervisors: Christos Antonopoulos, Assistant Professor
Nikolaos Bellas, Associate Professor
Department of Computer & Communication Engineering
University of Thessaly,
Volos, Greece

July 2014

Dedicated to

My family and my friends

Performance and power prediction on heterogeneous systems using statistical methods

Πρόβλεψη επίδοσης και κατανάλωσης ισχύος σε ετερογενή συστήματα με χρήση στατιστικών μεθόδων

by

Spyrou Michalis

mispyrou@inf.uth.gr

Submitted to the Department Electrical and Computer Engineering on July, 2014, for the degree of Diploma of Electrical and Computer Engineering

Περίληψη

Τα ετερογενή συστήματα προσφέρουν πολύ υψηλές αποδόσεις σε συνδιασμό με χαμηλό κόστος και χαμηλή κατανάλωση ισχύος. Τα συστήματα αυτά περιέχουν υπολογιστικά στοιχεία με διαφορετικές αρχιτεκτονικές όπως CPUs, GPUs, DSPs ή FPGAs. Είναι σημαντικό να έχουμε πλήρη γνώση των διάφορων αρχιτεκτονικών αλλά και των διαφορετικών προγραμματιστικών μοντέλων που χρησιμοποιούνται, για να μπορέσουμε να αυξήσουμε την απόδοση σε ένα τέτοιο σύστημα.

Ένας τρόπος για την αύξηση της επίδοσης είναι η πρόβλεψη του χρόνου εκτέλεσης στα διάφορα υπολογιστικά μέρη ενός ετερογενούς συστήματος χρησιμοποιώντας στατιστικά στοιχεία, τα οποία συλλέγουμε χρησιμοποιώντας διάφορους hardware counters. Στόχος αυτής της διπλωματικής εργασίας είναι η προσπάθεια αύξησης της επίδοσης ενός ετερογενούς συστήματος, χρησιμοποιώντας τα δεδομένα που συλλέξαμε εκπαιδεύοντας ένα στατιστικό μοντέλο το οποίο θα προβλέπει τον χρόνο εκτέλεσης. Απώτερος στόχος είναι η πληροφορία αυτή να χρησιμοποιηθεί σε έναν χρονοπρογραμματιστή, ο οποίος θα επανατοποθετεί την εκτελέσιμη εφαρμογή σε ένα καταλληλότερο υπολογιστικό στοιχείο ενός ετερογενούς συστήματος, με στόχο τη μείωση του συνολικού χρόνου εκτέλεσης και την αύξηση της επίδοσης.

Χρησιμοποιήσαμε διάφορα στατιστικά μοντέλα, όπως linear regression, neural networks και random forests με τα οποία προβλέψαμε τον χρόνο εκτέλεσης, με διαφορετικά ποσοστά επιτυχίας για κάθε μοντέλο σε Intel CPUs και NVIDIA GPUs.

Abstract

Heterogeneous systems provide high computing performance, combining low cost and low power consumption. These systems include various computational resources with different architectures, such as CPUs, GPUs, DSPs or FPGAs. It is crucial to have full knowledge of these architectures, but also of the programming models used in order to increase the performance on a heterogeneous system.

One way to achieve this goal, is the prediction of the execution time on the different computational resources, using statistical values which we collect with the use of hardware counters. The purpose of this thesis is to increase the performance of a heterogeneous system using the data we collected by training a statistical model which will predict the execution time. Further goal is to use this prediction model inside a run-time scheduler which will migrate the running application in order to decrease the execution time and increase the overall performance.

We used various statistical models, such as linear regression, neural networks and random forests and we predicted the execution time to Intel CPUs and NVIDIA GPUs, with different levels of success.

Copyright © 2014 by Spyrou Michalis.

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Acknowledgements

For the completion of this Thesis, I would like to thank my supervisors, Prof. Christos D. Antonopoulos and Prof. Nikolaos Bellas, for believing in me and without their assistance and dedicated involvement in every step throughout the process, this thesis would have never been accomplished. Also I would like to thank my colleague Angelina Delakoura for her support, collaboration and ideas.

Most importantly, none of this could have happened without my family, and their continious support through all the years of my studies.

Contents

Abstract iv

Acknowledgements viii

1 Introduction 1

1.1 Problem Description 1

1.2 Purpose of this thesis 2

1.3 Thesis Structure 3

2 Background Issues 4

2.1 Architectural differences 4

2.1.1 Intel CPUs 4

2.1.2 NVIDIA GPUs 5

2.2 Programming Models 7

2.2.1 OpenCL 7

2.2.2 CUDA 8

2.3 Performance Prediction 9

2.3.1 Prediction Techniques 10
2.3.2 Data Collection 10
2.3.3 Statistical Models for Data Analysis 11
2.4 Related Work 14

3 Methodology 15

3.1 Main Purpose 15
3.2 Data Collection 16
3.2.1 Intel Xeon and PAPI 16
3.2.2 CUDA and nvprof 18
3.3 Forming the Model 19
3.3.1 Data Correlation 20
3.3.2 Model for predicting OpenCL CUDA execution time from Intel OpenCL 22
3.3.3 Model for predicting Intel OpenCL execution time from NVIDIA CUDA 25

4 Evaluation 28

4.1 Intel Xeon OpenCL to GTX480 CUDA prediction results 28
4.2 GTX480 CUDA to Intel Xeon OpenCL prediction results 33

5 Conclusion and Future Work 35

Bibliography 37

List of Figures

2.1 Xeon 5600/5500 Overview 5

2.2 Kepler GF100 block diagram 6

2.3 OpenCL memory model 8

2.4 CUDA architecture model 9

2.5 Neural network overview 13

2.6 Decision Tree Overview 13

3.1 Floating point instructions measured on Nearest Neighbour kernel . . 18

3.2 Neural Network Training Performance 23

3.3 Out-of-bag error with 20 trees 24

3.4 Out-of-bag error with 60 trees 24

3.5 Out-of-bag error with 120 trees 25

3.6 Neural Network Training Performance 27

3.7 Out-of-bag error using 60 trees 27

4.1 Linear prediction results using 6 hardware counters - Intel to CUDA 29

4.2 Linear prediction results using 7 hardware counters - Intel to CUDA 30

4.3 Linear prediction results using 8 hardware counters - Intel to CUDA 31

4.4 Random Forest Prediction with 60 trees - Intel to CUDA 31

4.5 Neural Network Prediction - Intel to CUDA 32

4.6 CUDA to Intel OpenCL linear prediction results 33

4.7 CUDA to Intel OpenCL prediction results using neural networks . . . 34

4.8 CUDA to Intel OpenCL prediction results using random forests . . . 34

List of Tables

3.1 Intel to CUDA OpenCL execution time correlation values 21

3.2 CUDA to Intel OpenCL execution correlation values 21

3.3 Linear regression for CUDA prediction; RMSE values 22

3.4 Linear regression for CUDA prediction; RMSE values 26

Chapter 1

Introduction

1.1 Problem Description

During the last years, the satisfying the demand for faster and smaller processors has become an increasingly difficult challenge. For years this was not a problem as hardware vendors delivered consistently increased clock rates and instruction-level parallelism, so single threaded programs achieved very easily speedup without any modification. Nowadays we have reached a point, where the improvement of the performance on single core systems is extremely hard.

To overcome physical limitations and increase performance, multicore CPUs and massively parallel hardware accelerators such as GPUs and FPGAs were favoured by the hardware industry. For these architectures, software must be written in a multi-threaded way in order to gain performance. As a result, the problem shifted from the hardware designers to the software developers.

Each component in a heterogeneous system had its own programming model. NVIDIA developed CUDA, a parallel computing platform and programming model in order for the GPUs to be used for general purpose processing. On the other hand Intel, has various programming models such as OpemMP and Threading Building Blocks in order to program the multicoe CPUs. In order to solve this problem, Khronos

Group in collaboration with dozens of other hardware and software vendors, developed the OpenCL framework [1], so the software developers can write applications that can be executed across heterogeneous platforms such as multicore CPUs, GPUs, FPAGAs and DSPs, without the need to rewrite the application.

OpenCL promises functional portability, however does not promise performance portability. Heterogeneous systems are volatile: They can easily change in a multitude of configurations. The programmer can not know on which configuration the program will be executed. We would like to have generic programs and map the optimally on the underlying platform. This requires performance prediction.

Prediction can be made possible using statistical methods, such as linear regression, least squares, random forests or even neural networks.

1.2 Purpose of this thesis

Predicting performance on heterogeneous systems is critical for facilitating the mapping of applications on the various computational resources of a heterogeneous system. The prediction can be static at compile time or dynamic at run-time, based on analytical models or statistical models. We would like to combine a statistical model with the run-time in order to predict the execution time of an application on a different platform. A way to do that is to use an off-line statistical training of a prediction model, and then feed the model with results from the initial steps of the execution of the target application at run time. Hopefully, using information by the execution on one system component, the model should be able to predict performance on other components. Thus, if possible, we could migrate the application and increase the overall performance of the system. We created a model that predicts performance using statistical methods, like linear regression, random forests and neural networks. The model is based on the OpenCL and CUDA framework and takes into account different hardware architectures such as Intel CPUs, AMD GPUs and NVIDA GPUs, using PAPI for Intel CPUs [2] and nvprof [3] for the

CUDA architecture.

1.3 Thesis Structure

Thesis is structured in three main parts.

The first part deals with background issues and the architectural differences of each hardware platform we used to run the experiments. More specifically, in section 2.1 we start by analysing the Intel Xeon and OpenCL architecture and then CUDA architecture and explain both the programming models and the hardware architectures used in our experiments. In section 2.3 prediction techniques are discussed and in section 2.4 related work is presented.

Second part describes the methodology used in order to collect data from the applications. Section 3.2.1 analyzes the usage of PAPI with the Intel OpenCL runtime and how we collected data from the applications. Section 3.2.2 describes the method used to collect data from the CUDA platform using NVIDIA profiler.

Next, section 3.3.2 presents the prediction models we created using different statistical methods and section 4 presents our models evaluation.

Finally in section 5 we describe the conclusions that we came to and discuss possible future work.

Chapter 2

Background Issues

In order to predict performance, we first need to gain a deep understanding on the architectural characteristics of each platform, and how programs interact with hardware. In this section we describe the architectural differences of each platform we used for our experiments. First we analyse the Intel Xeon CPU hardware architecture, the NVIDIA architecture and finally we describe the OpenCL and CUDA programming models.

2.1 Architectural differences

2.1.1 Intel CPUs

During the past few years, the need for increased performance, drove the hardware vendors to find other solutions than just increasing clock rate and decreasing the transistors' size. So CPUs went from single core to multicore platforms. Software developers were able to take advantage of the cores that the CPU offered them, using the right programming models and tools, such as OpenCL. Nowadays a CPU can have up to 61 [5] cores and with the use of Hyper-Threading Technology, the CPU can increase significantly the supported number of threads.

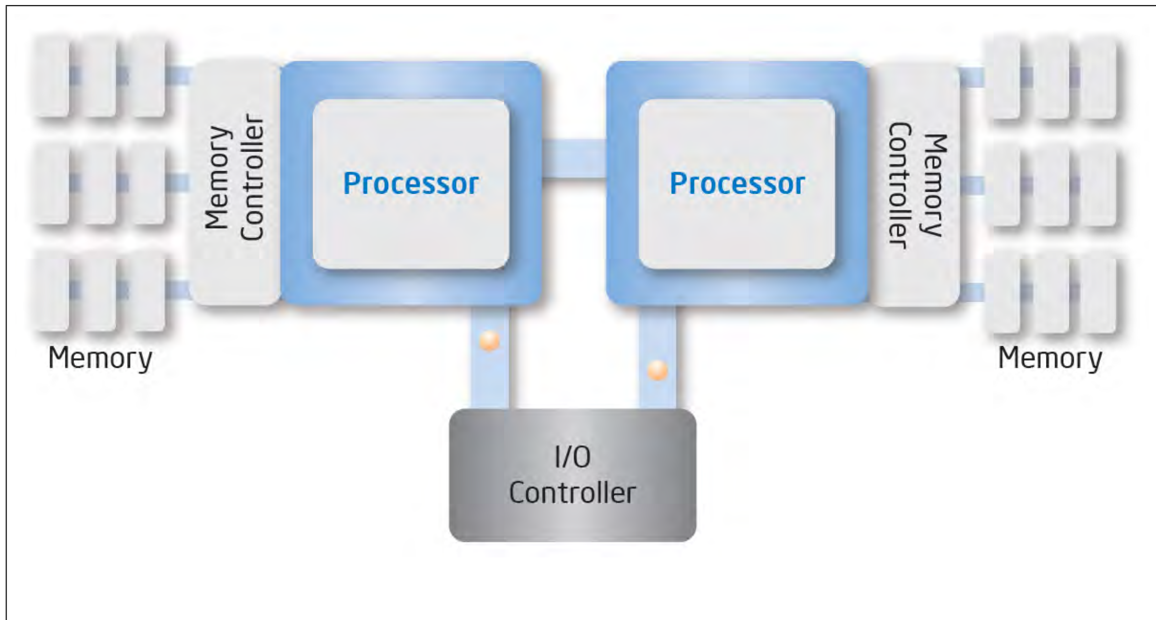

Figure 2.1: Xeon 5600/5500 Overview

The complexity of a modern CPU is hidden from the application developer in order to allow the user to focus on the algorithms and more general issues, such as the communication between different parts of the multithreaded application. This is achieved through a small set of abstractions regarding memory hierarchy, and synchronization primitives.

For example an Intel Xeon E5600 series processor has three levels of hardware caches, 6 cores and can support up to 12 threads. There are on-chip L1 and L2 private memory caches near each core, which are shared by the "hyper"threads potentially executing on each core. Also there is a 12Mb last level cache (L3 cache) shared across all sockets.

In order for the programmers to take advantage of these architectures, different programming models have been created, such as OpenMP, OpenACC, Thread Building Blocks and OpenCL.

2.1.2 NVIDIA GPUs

Initially GPUs where used for creating graphic images and their architecture differed a lot from a CPU. The CPU is composed of few cores with lots of cache memory

Figure 2.2: Kepler GF100 block diagram

that can handle a few software threads at a time. In contrast, a GPU is composed of hundreds of cores that can handle thousands of threads simultaneously. Nowadays GPUs have moved closer and closer to being general purpose parallel computing devices.

In figure 2.2 is shown the NVIDIA Kepler architecture. The main part of the GPU is composed of computational components which are called streaming multiprocessors (SMs). In each SM, computation is performed by execution units called streaming processors (SPs). Streaming processors have access to a private set of registers allocated from a register file and also there are per-SM L1 caches, a common L2 cache and the device memory which holds all the device data including texture and constant data [6].

2.2 Programming Models

2.2.1 OpenCL

OpenCL is a portable programming model for writing programs that execute across heterogeneous platforms.

An OpenCL program consists of two parts; the host code, which runs on the CPU and interfaces to the device and the kernel code which runs on the device. The host code is responsible for device initialization, memory allocation on the device and data transfer to/from device memory and running the control-intensive part of the application. Proper usage can deliver significant performance improvements. However the OpenCL standard only specifies the access levels of different types of memory.

OpenCL defines a memory hierarchy of types as illustrated in figure 2.3. This memory hierarchy is very similar to the memory hierarchy of the GPUs. In the case of NVIDIA GPUs the device memory maps to the compute device memory, private memory maps to the local registers available to each streaming processor, constant memory is the same in both models and host and device memories are separate.

In the case of Intel CPUs there is no difference between device and host memory. Local memory is allocated directly from the L3 cache and accesses to global memory, constant memory, and private memory go through the L3 cache and LLC (last-level cache) [8].

While OpenCL is a portable programming model, the performance portability is not guaranteed. GPUs and CPUs have different hardware designs, and their differences are such that they benefit from different application optimizations.

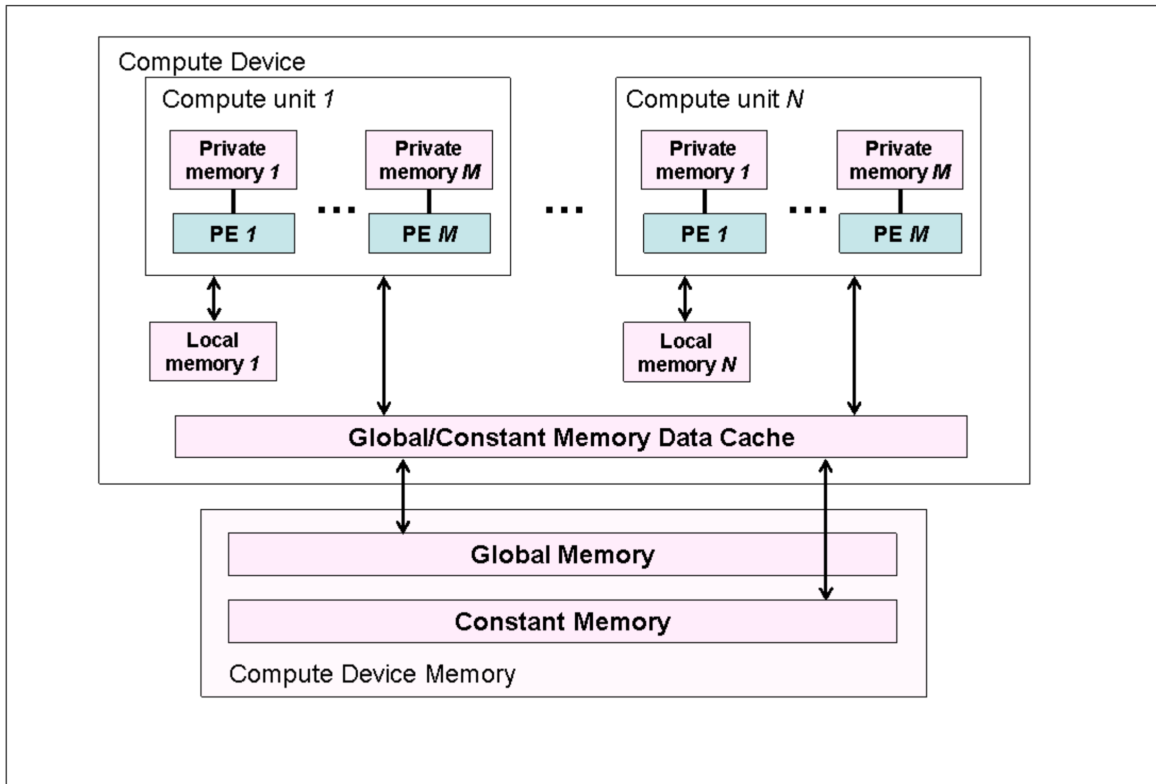

Figure 2.3: OpenCL memory model

2.2.2 CUDA

CUDA is a parallel computing platform and programming model invented by NVIDIA. CUDA gives program developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Unlike CPUs, GPUs have a parallel throughput architecture that emphasizes on executing many concurrent threads slowly, rather than executing a single thread very quickly.

CUDA architecture is much like OpenCL as we see in figure 2.4. As a parallel programming model, OpenCL is preceded by CUDA and as a result, OpenCL heavilly borrowed a lot of CUDA characteristics.

The philosophy of the CUDA programming model is to partition the code in smaller sub-problems that can be solved independently in parallel by blocks of threads. Then each of these sub-problems must be partitioned into even smaller pieces that can be solved simultaneously in parallel by the threads of a block. Therefore, the model achieves good scalability as thread blocks are independent of each other.

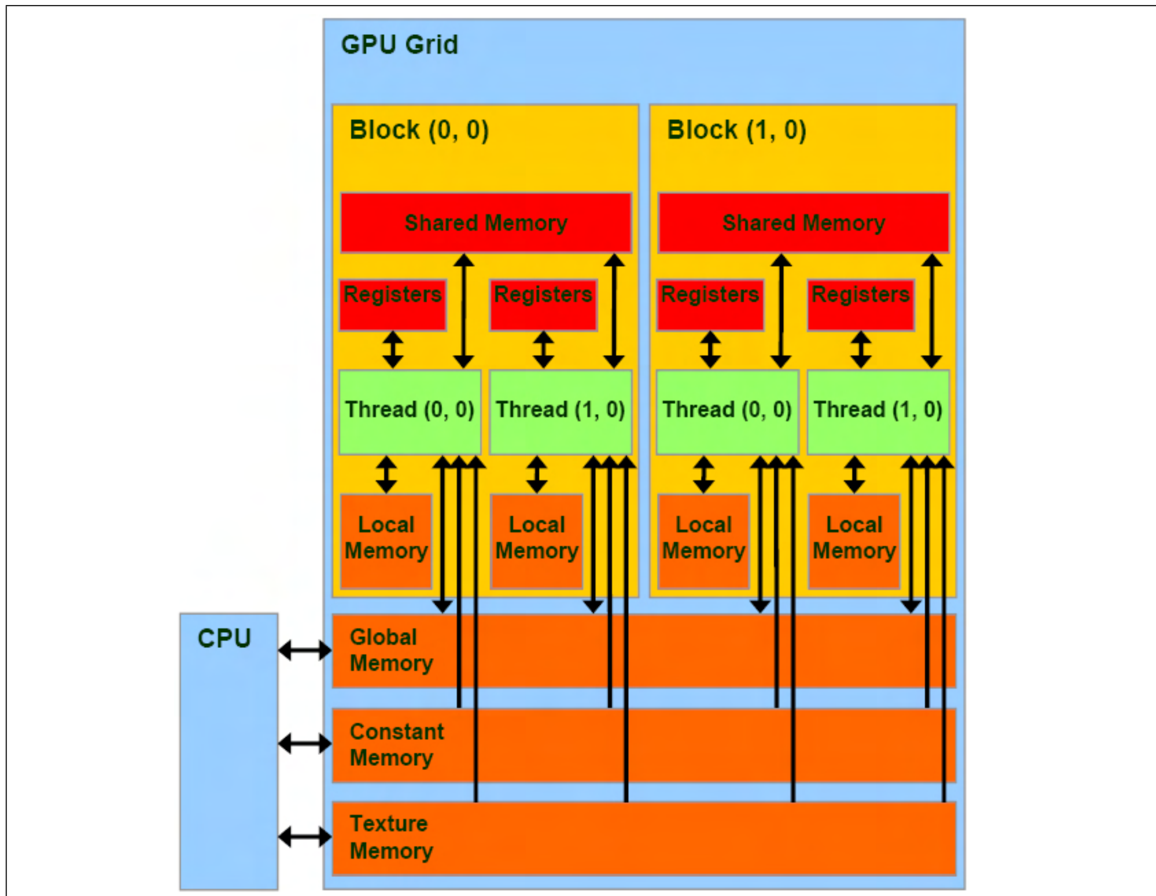

Figure 2.4: CUDA architecture model

For an in-depth treatment of the subject we refer the reader to [7].

2.3 Performance Prediction

Performance prediction has a very important role in many computer science fields; compiler developers require a way to predict the performance of the generated code on different architectures, software developers need a better understanding in order to tune their code for a particular architecture and run-time schedulers need to be able to predict performance in order to achieve a good application / architecture mapping. The main goal of most techniques that have been used, is to estimate the execution time of a program on a given architecture.

2.3.1 Prediction Techniques

We can classify the prediction techniques into the following categories:

Analytical models, where we try to create a pure mathematical model which takes into account the program input and the target architecture.

Profile based models, which are the most common technique and involve two steps. The first step is the collection of metrics, such as hardware counters, while the program executes, using various tools and/or profilers and the second step is to feed an analysis tool with these metrics. An example of modern profilers for collecting data are the NVIDIA profiler, Intel Vtune and PAPI. Models for statistical data analysis can vary, from regression techniques to neural networks and random forests.

2.3.2 Data Collection

Most modern microprocessors have hardware performance counters that are implemented as a small set of registers that count events, occurrences of specific signals related to the processor's function. Such events, for example, may be the total number of floating point operations, total L1 cache misses and number of successfully predicted branches. In order to measure the events we can either access them directly or use a higher level tool like PAPI [2].

PAPI provides a simple API which supports a wide range of modern microprocessors and accesses the hardware counters via two simple interfaces; the high level and the low level interface. The high level interface is used for the acquisition of simple measurements and the low level interface is fully programmable by the user. For our measurements we used an Intel Xeon E5645 which supports the hardware counters shown in table A.1.

Listing 2.1: Usage of PAPI's high level interface

```
int events[2] = {PAPI_BR_MSP, PAPI_FP_OPS }, ret;
```

```
long_long values[2];

if ((ret = PAPI_start_counters(events, 2)) != PAPI_OK) {
   fprintf(stderr, "PAPI failed to start counters: %s\n",
       PAPI_strerror(ret));
   exit(1);
}

/* Computation */
...

if ((ret = PAPI_read_counters(values, 2)) != PAPI_OK) {
  fprintf(stderr, "PAPI failed to read counters: %s\n",
      PAPI_strerror(ret));
  exit(1);
}

printf("Total flops: %lld\n",(float)values[1]);
printf("Conditional branch instructions mispredicted: %lld\n", values[0]);
```

PAPI also supports CUDA enabled GPUs, but we used a different approach. In order to measure hardware events on the GPU we used tools that provide more accuracy and are better supported.

NVIDIA profiler, nvprof, is a command line tool that provides a very simple mechanism to collect and view profiling data of an application. The GPU we used, GTX 480, supports a wide range of metrics that can be collected as seen in table A.2.

2.3.3 Statistical Models for Data Analysis

After collecting the values of the hardware counters of a large set of applications, called training set, the next step is to feed them to a statistical model in order to find

the correlation between the measured metrics and the execution time, a procedure called training. After analysing the data, we create a mathematical function based on the modeling approach, that describes our system. Then we can use this function to predict the execution time of a new application -not in the training set-, by giving as input the collected hardware counters.

The most widely used approach for modeling the relationship between a variable y and one or more explanatory variables is linear regression. In linear regression, data are modeled using linear predictor functions, and unknown model parameters are estimated from the data.

Given a data set: $\{y_i, x_{i1}, ..., x_{ip}\}_n^{i=1}$ a linear regression model assumes that the relationship between the dependent variable y_i and the p-vector of regressors x_i is linear. Thus the model takes the form: $y_i = \beta_1 x_{i1}, ..., \beta_p x_{ip} + \epsilon_i$. The main drawback of this method is that it only looks at linear relationships between dependent and independent variables and it assumes there is a straight-line relationship between them [9]. We can also use a non-linear model which contains an intercept, linear terms, interactions, and squared terms.

Another approach is the use of artificial neural networks (ANN). Neural networks are inspired by the central nervous system and are capable of machine learning and recognizing patterns. An ANN consists of a number of interconnected nodes, typically organized in layers, and each one of them has a set of inputs and an output that is forwarded to the next layer 3.2. A node is capable of performing a certain mathematical function, the activation function, which usually are the sigmoid, $S(t) = \frac{1}{(1+e^{-t})}$, the tan-sigmoid, $\sigma(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}}$, or the Heaviside step function [10].

Neural networks can be used for prediction with varying level of success. The main advantage includes automatic learning of dependencies using only the measured data without any need to add further information, like in the regression, such as type of dependency. The neural network is trained with the hope that it will discover hidden relations and that it will be able to use them for prediction. A neural network is like a black box, as seen in 3.2, that is able to learn something. This is, at the same

time its main disadvantage: the user does not have full knowledge of its internal working [11].

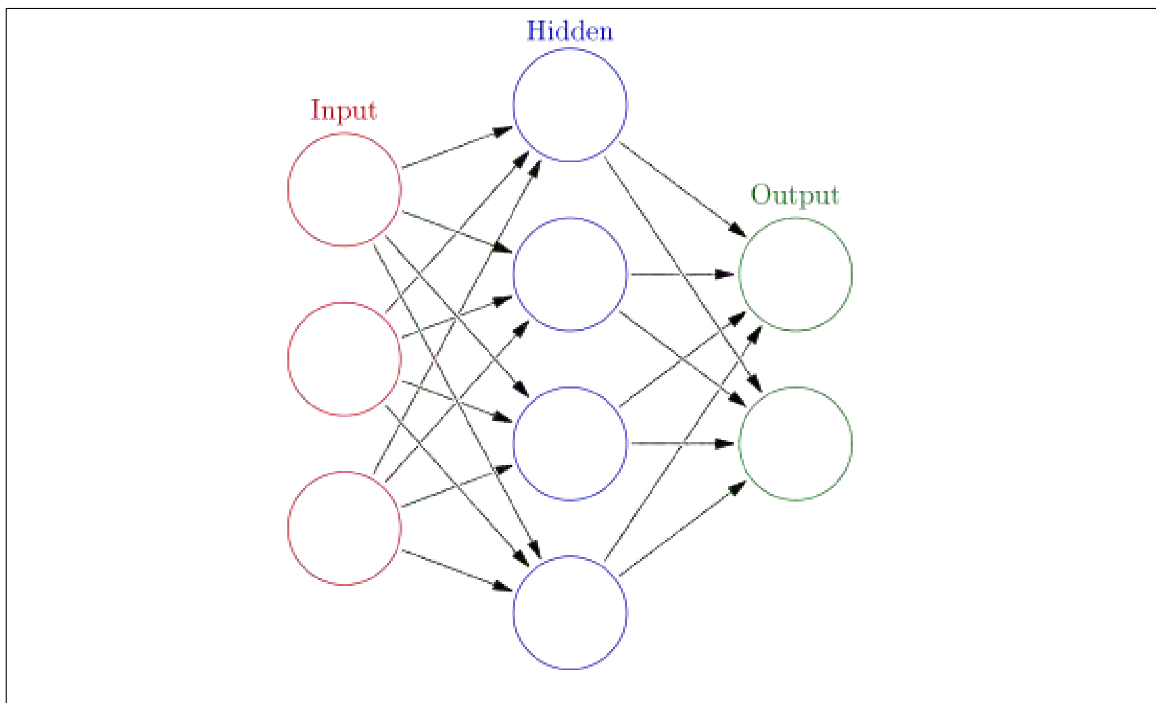

Figure 2.5: Neural network overview

Finally another statistical model used for classification and regression is this of random forests. Random forests are a combination of decision trees such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. A decision tree, as seen in 2.6, is a flowchart-like structure in which each internal node represents test on an attribute, a branch represents the result of a test and a leaf node represents the decision taken. A path from root to leaf represents classification rules [12].

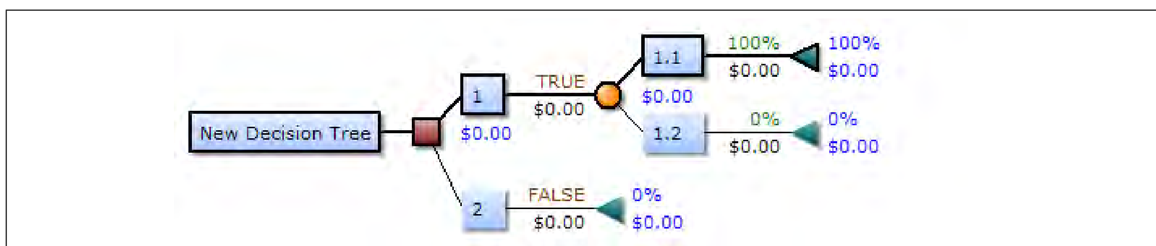

Figure 2.6: Decision Tree Overview

2.4 Related Work

Much work has been done for predicting performance and power on various systems, using linear models and performance events.

Gilberto [19] used performance monitoring unit events in order to estimate power consumption on Intel XScale Processors. Yuwen [20] presented a low-cost estimation of sub-system power with the use of performance counters. Also power has been estimated with the use of performance counters and used for thread scheduling [23]. Furthermore, many methods were used for performance prediction on various systems using hardware counters [21] and parametrized models [22] [24]. All these approaches were used to predict execution time and power on the same architecture.

In our work, the models based on one architecture were used for predicting the execution time for different architecture. We used hardware counters with a combination of linear and non-linear models, neural networks and random forests, to predict execution time.

Chapter 3

Methodology

This chapter describes the methods used for each platform in order to collect data and also explains the creation of the statistical model. As training set we used Rodinia, an open source benchmark which contains compute intensive applications written in OpenCL and CUDA and we collected data from each kernel invocation separately. In table A.3 there is the full list of all the applications we used to collect data. The total number of different kernels from all the applications is 44.

3.1 Main Purpose

in this section we present the reasons that led to the completion of this work, as well as the ways that have been followed to finally achieve it.

One way to increase performance on a heterogeneous system is the prediction of the execution time on the different computational resources, using statistical values which we collect with the use of hardware counters.After collecting the data, we train a statistical model which will predict the execution time on a different device. Having this information we can use it inside a software scheduler which will migrate the running application at run-time in order to decrease the execution time and increase the overall performance.

3.2 Data Collection

3.2.1 Intel Xeon and PAPI

Firstly, the explanation of how the Intel's OpenCL implementation works on Intel Xeon and Intel Xeon Phi coprocessors is essential. These coprocessors include many cores, each of them usually capable of supporting more than one hardware threads. Our CPU, E5645, has 6 cores and each core supports 2 hardware threads, making a total of 12. The exact number of hardware threads, can be queried with the clGetDeviceInfo() interface.

At the initialization time of the OpenCL driver, a number of software threads is created, equal to the maximum number of the hardware threads supported, and these software threads are pinned to the HW threads. Then, following a clEnqueueNDRange() call, the driver schedules the work groups (WG) of the current NDRange on the threads.

Thus, in order to collect data from an application, we have to collect the hardware events from all the software threads the OpenCL run-time creates. Using PAPI's low level API we can attach PAPI to a number of different threads, given their thread id, and collect all the hardware events at the same time. So, right after the OpenCL driver initialization, we attach PAPI to all the created software threads. The process can be seen in B.1.

Collecting data with PAPI's low level API differs from the high level example seen in 2.1. We now have to manually create user-defined groups called Event Sets, that contains the hardware events we want to measure, initialize the PAPI library, start and finally stop each event set using the low level API calls for each thread.

Listing 3.1: Attaching to the software threads

```
int *EventSet = NULL;
long long *results;
long long total_res = 0;
```

```
papi_init();
proc_threads *proccess_threads = NULL;
cl_uint compute_units;
init_papi_cpu( device, &compute_units, &proccess_threads);

clFinish(command_queue);
start_cpu_counter ( counter, &EventSet, proccess_threads, compute_units);
error = clEnqueueNDRangeKernel( command_queue, kernel, 1, NULL,
    global_work_size, local_work_size, 0, NULL, NULL);

clFinish(command_queue);
stop_cpu_counter ( EventSet, &results, compute_units);
int i;
printf("*Run #%d\n", _run);
for ( i = 0; i < compute_units; total_res += results[i], i++ )
   printf("Thread #%d: %lld\n", i, results[i]);
printf("Sum: %lld\n\n", total_res);
```

The next step is to start measuring the hardware events right before the execution of a kernel and stop after the kernel has finished executing. We want to be sure that the measurement involves only the kernel execution, so the best way to do that is by ensuring that all the other tasks inside the command queue are completed. Thus, before each clEnqueueNDRange() invocation, we call clFinish(), that blocks until all previously queued OpenCL commands in command queue are issued to the associated device and have completed. Right after, we start the the specified PAPI counters and wait until the kernel is executed, by calling again clFinish() and finally we can collect our data. An example of this process can be seen in 3.1. Note that we keep track of each thread separately.

Time profiling of the target application is done with the built in functions of OpenCL. First, at the creation of the command queue we enable the kernel profiling by pass-

ing the flag CL_QUEUE_PROFILING_ENABLE. Then, an event is passed as an argument to the clEnqueueNDRangeKernel() and after the kernel finishes execution, we can calculate the execution time with clGetEventProfilingInfo().

Each application is executed 100 times for each hardware counter and we take the average of each counter. Also we eliminate abnormal values, such as wrong measurements which some times occurred and possibly caused by frequency scaling or the number of measured registers [18]. In figure 3.1 we present the values measured from each execution of a single kernel.

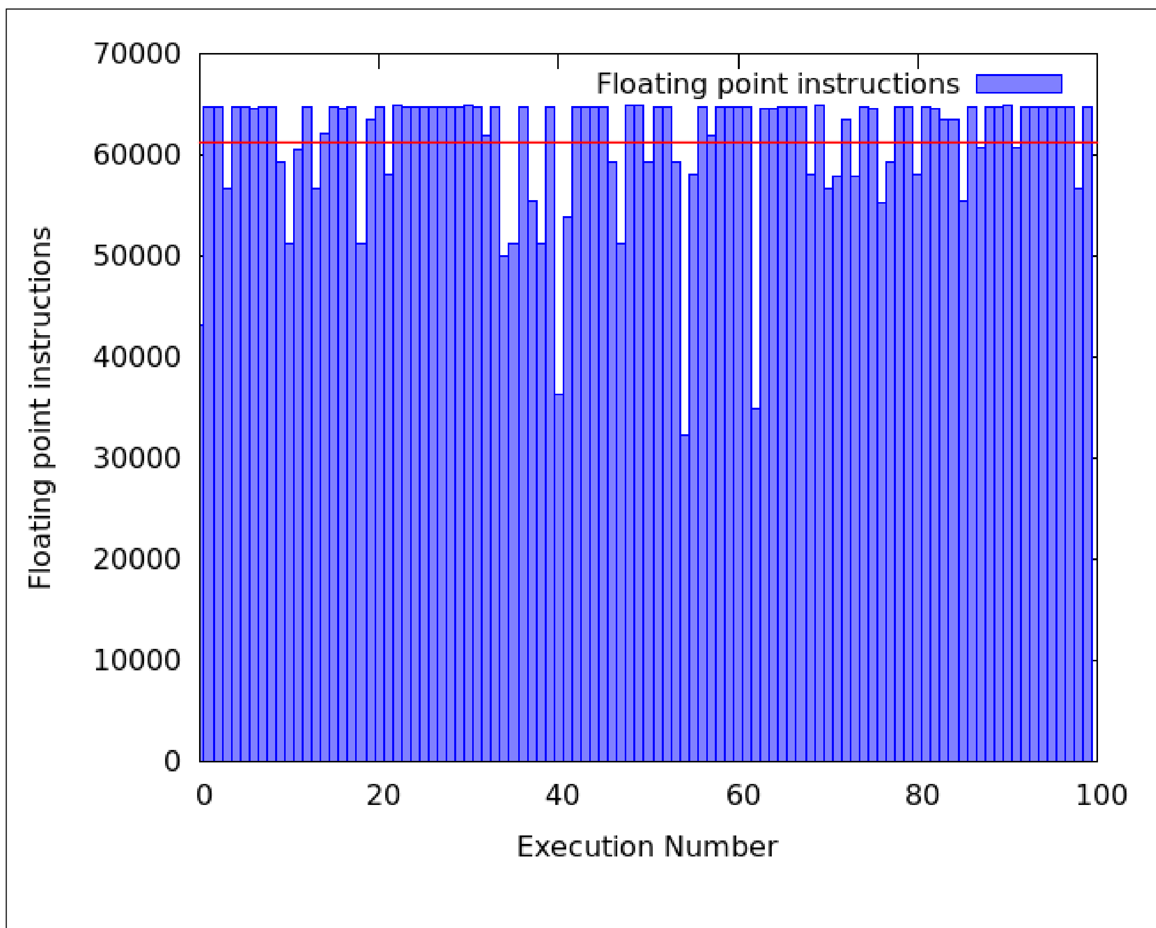

Figure 3.1: Floating point instructions measured on Nearest Neighbour kernel

3.2.2 CUDA and nvprof

The ideal way for our measurements, was to use the same applications written in OpenCL and execute them in the NVIDIA GPU. But as we figured out, there

was no way to access the hardware counters on the GPU, because nvprof does not support OpenCL kernels. Thus, the best alternative way was to execute the same applications, this time written in CUDA. We can do this because the rodinia OpenCL applications are ported from the original CUDA applications and the only difference between them is the use of OpenCL functions, such as get_global_id() which translates to blockIdx.x*blockDim.x+threadIdx.x in CUDA.

Although we could not collect hardware counters using OpenCL on CUDA, we could measure the execution time using the built in function clGetEventProfilingInfo(). This way we kept the same kernels we used on Intel platform.

The data collection from a CUDA application is a lot simpler than the case of the CPU. Using nvprof we can collect all metrics automatically just by passing as arguments the target application and the metrics we want to collect. If a kernel is invoked more than once, the total average is returned along with the highest and lowest values recorded. A simple example can be seen in 3.2. This time each application is executed only 10 times, as nvprof's results had very small to no variation.

Listing 3.2: Running nvprof and printing results

```
$ nvprof --metrics l2_l1_read_throughput,l2_l1_read_hit_rate ./nn
   filelist.txt -r 5 -lat 30 -lng 90
==11933== Profiling result:
==11933== Metric result:
Invocations Metric Name       Min        Max        Avg
Kernel: euclid(latLong*, float*, int, float, float)
1         l2_l1_read_hit_rate  70.26%     70.26%     70.26%
1         l2_l1_read_throughput 84.965GB/s 84.965GB/s 84.965GB/s
```

3.3 Forming the Model

After the collection of the raw data the next step is to use them to train a statistical model. Because of the small training set and the high number of performance

counters, mainly in the Intel CPU and the NVIDIA GPU, it is not efficient to use all of them as input because this may cause overfitting [13]. Thus in order to eliminate some performance counters from the model it, one method was to find the correlation of each counter with the target value, execution time, and keep only the counters with strong correlation. In the next sections we discuss the correlation of each measured metric with the execution time and we also present the results of the statistical models used.

3.3.1 Data Correlation

Correlation is a statistical measure that indicates the extent to which two or more variables alter one another [15] and has a range between -1 and 1. A positive correlation indicates the fact that these variables increase in parallel and also the degree of influence between them. A negative correlation indicates the fact that when one variable increases the other one decreases and vice versa and finally, a zero correlation means that the two variables are unrelated. The main correlation coefficient used was Spearman's [14], $\rho = 1 - \frac{\Sigma_i (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\Sigma_i (x_i - \overline{x})^2 \Sigma_i (y_i - \overline{y})^2}}$.

In tables 3.3.1 and 3.3.1 we present the correlations of each hardware counter with the execution time measured on the target platform. Because the training set is small and the total hardware counters are a lot more, it is needed to filter out the variables with small correlation value in order to form the model. As we observed, more than 10 hardware counters tend to overfit the model and thus, only the 10 hardware counters with the highest correlation are taken into account.

As expected the hardware counters with the biggest correlation in both cases, are those which measure the total instructions completed, total cycles and total instructions issued. In the case of Intel's Xeon architecture the variables that have high impact on the execution time are also the total branch instructions, branch instructions taken and correctly predicted, total memory operations completed and also the level 2 cache hits. In the case of CUDA architecture high impact have the level 2 cache read and write transactions, number of issued and executed control flow

instructions and the number of issued and executed load and store instructions.

Counter	Correlation
PAPI_BR_TKN	0.8874
PAPI_TOT_CYC	0.868
PAPI_BR_PRC	0.862
PAPI_TOT_INS	0.861
PAPI_LD_INS	0.8553
PAPI_BR_CN	0.8543
PAPI_LST_INS	0.8503
PAPI_BR_INS	0.8473
PAPI_L2_DCA	0.8432
PAPI_L2_TCH	0.8379

Table 3.1: Intel to CUDA OpenCL execution time correlation values

Counter	Correlation
ldst_executed	0.88
inst_issued	0.88
issue_slots	0.88
inst_executed	0.86
ldst_issued	0.86
cf_issued	0.85
cf_executed	0.85
l2_read_trans	0.83
l2_write_trans	0.72
dram_write_trans	0.70

Table 3.2: CUDA to Intel OpenCL execution correlation values

3.3.2 Model for predicting OpenCL CUDA execution time from Intel OpenCL

The first step to form the model in order to predict the execution time on the CUDA architecture using OpenCL, is to train it, using as input the hardware counters with high correlation values from the Intel Xeon architecture. Several statistical methods were used, which generated different results and are all presented below.

First method used, was linear regression using a simple linear model which contains an intercept and linear terms for each predictor. We created different models based on the number of hardware counters we used. One way to evaluate the model is the **root-mean-square error (RMSE)**, which indicates the standard deviation of the differences between predicted values and observed values. In table 3.3.3 are presented the different RMSEs from each model.

Number of Counters Used	RMSE
9	5.12
8	7.29
7	7.17
6	8.86
5	229

Table 3.3: Linear regression for CUDA prediction; RMSE values

As seen in table 3.3.3, RMSE for all the cases except the last is slightly above zero, which means our model could sometimes generate wrong results. In the case of 5 hardware counters the RMSE is too high which does not indicate a good fit for the model.

Next we used a non-linear quadratic model which contains an intercept, linear terms,

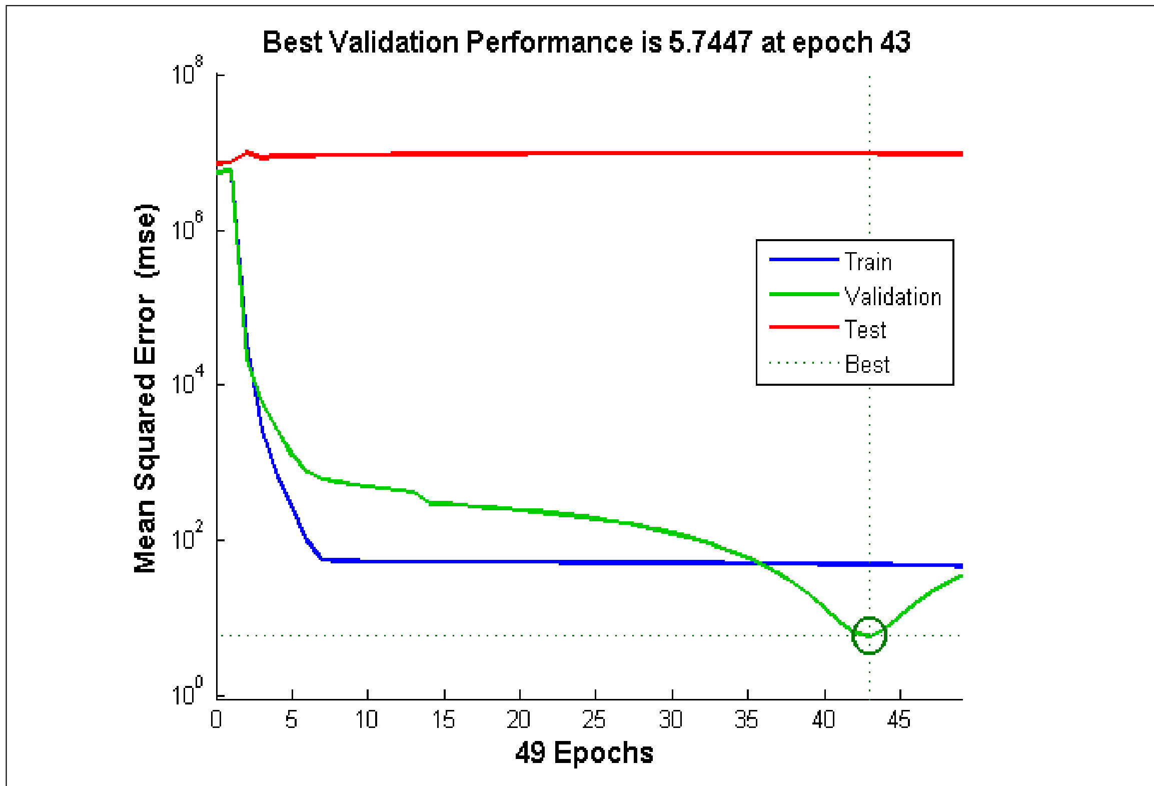

Figure 3.2: Neural Network Training Performance

interactions, and squared terms. Again we created different models based on the number of the hardware counters we used to train the model. In all cases except when we used 5 counters, the RMSE of the generated models is exactly zero, and in the case of 5 hardware counters 0.77, which indicates a good fit.

We also created models based on neural networks and random forests. As seen in figure 3.2 the MSE is quite high in the case of neural networks no matter the number of neurons used. Random forests prediction model out-of-bag error is shown in 3.5. Again the error is too high no matter the number of trees in the forest. This happens because methods like neural network fitting require a very large training set and the training set we used was too small.

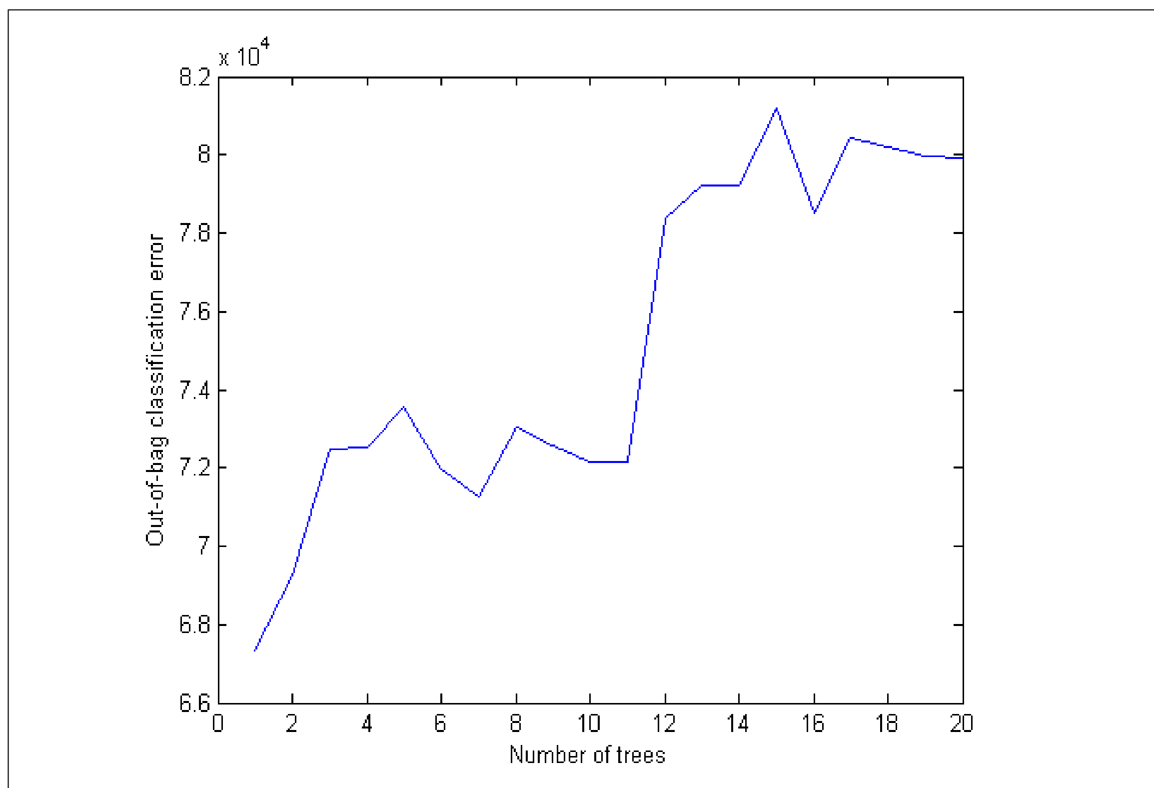

Figure 3.3: Out-of-bag error with 20 trees

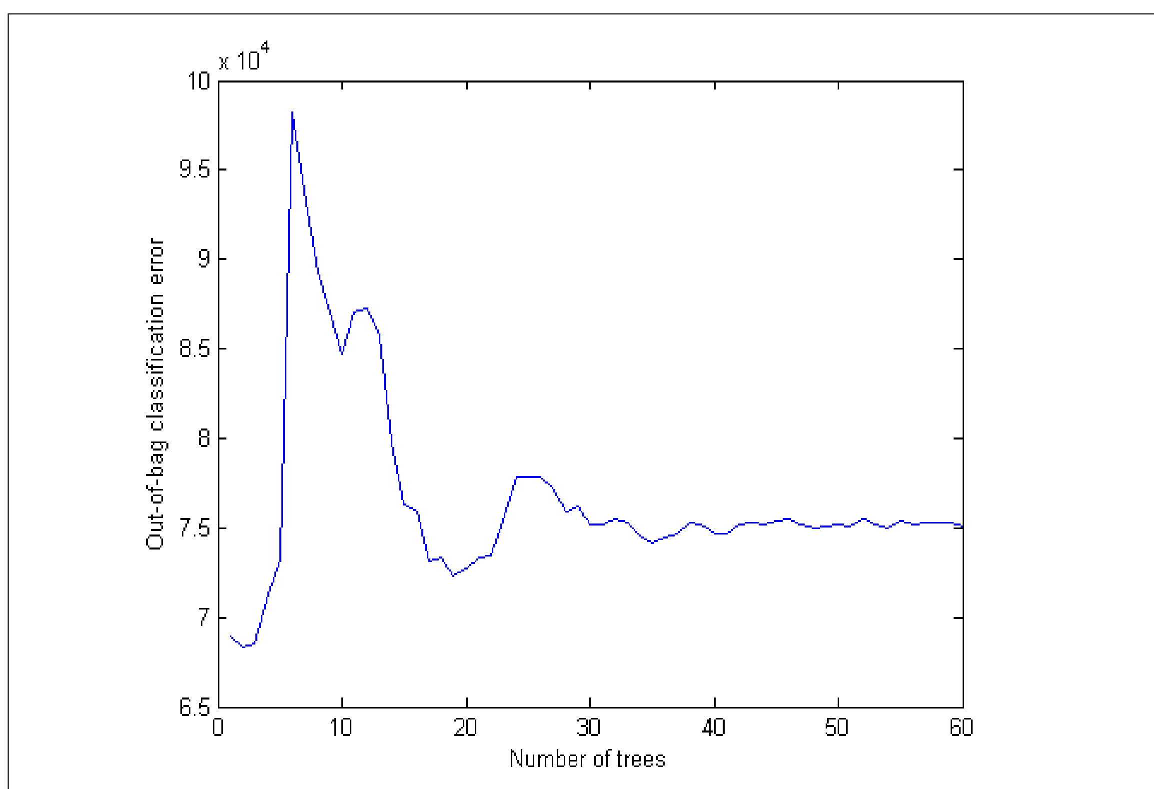

Figure 3.4: Out-of-bag error with 60 trees

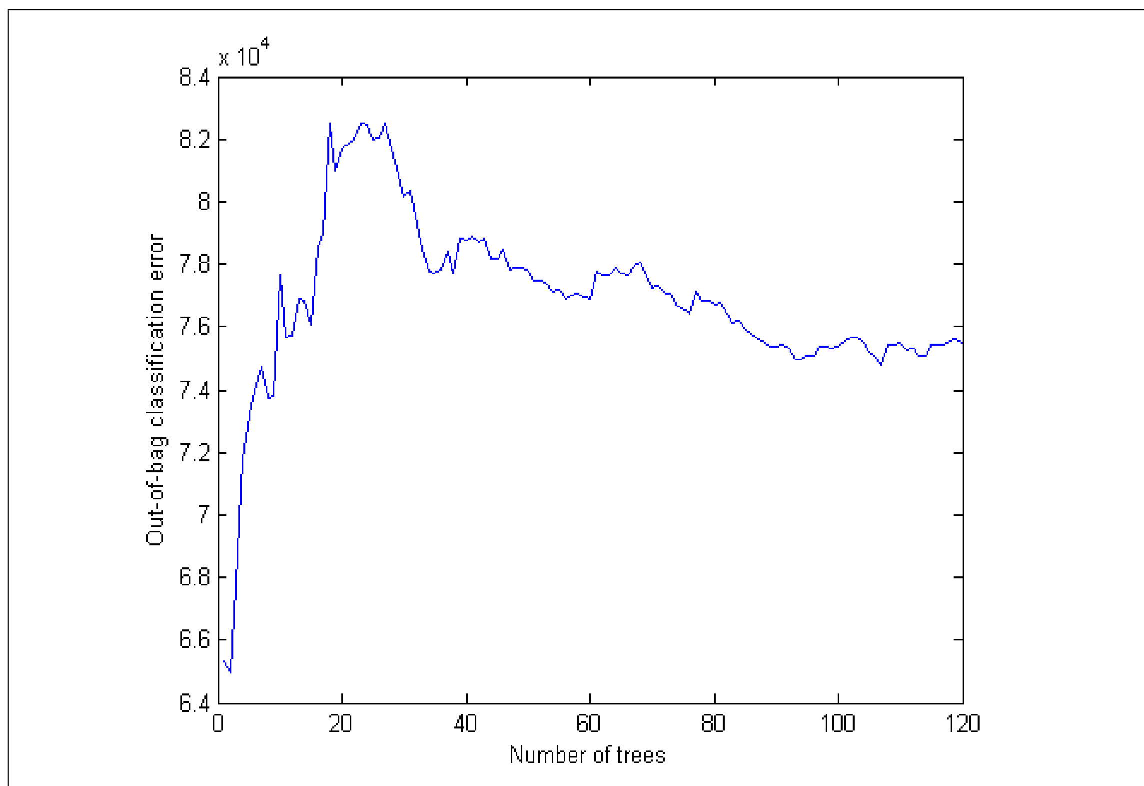

Figure 3.5: Out-of-bag error with 120 trees

3.3.3 Model for predicting Intel OpenCL execution time from NVIDIA CUDA

The same approach was used to form the model for prediction Intel OpenCL execution time using CUDA. We again trained the models using the hardware counters with the highest correlation values.

In the case of linear regression the RMSE error, as seen in **??**, was higher compared to the case of Intel OpenCL to CUDA prediction. The smallest error achieved when we used 10 hardware counters to our model.

Number of Counters Used	RMSE
10	92.6
9	99.1
8	110
7	110
6	109
5	115

Table 3.4: Linear regression for CUDA prediction; RMSE values

The non-linear quadratic model returned an RMSE of 3.52 for 10 and 9 hardware counters, 16 for 8 hardware counters and above 100 for less hardware counters used. Again the non-linear model indicates a better fit than the linear.

Again, in the case of neural networks, the MSE was too big as seen in 3.6, making the model unable to predict the correct values. High out-of-bag error had again random forests, no matter the number of trees used, as seen in 3.7.

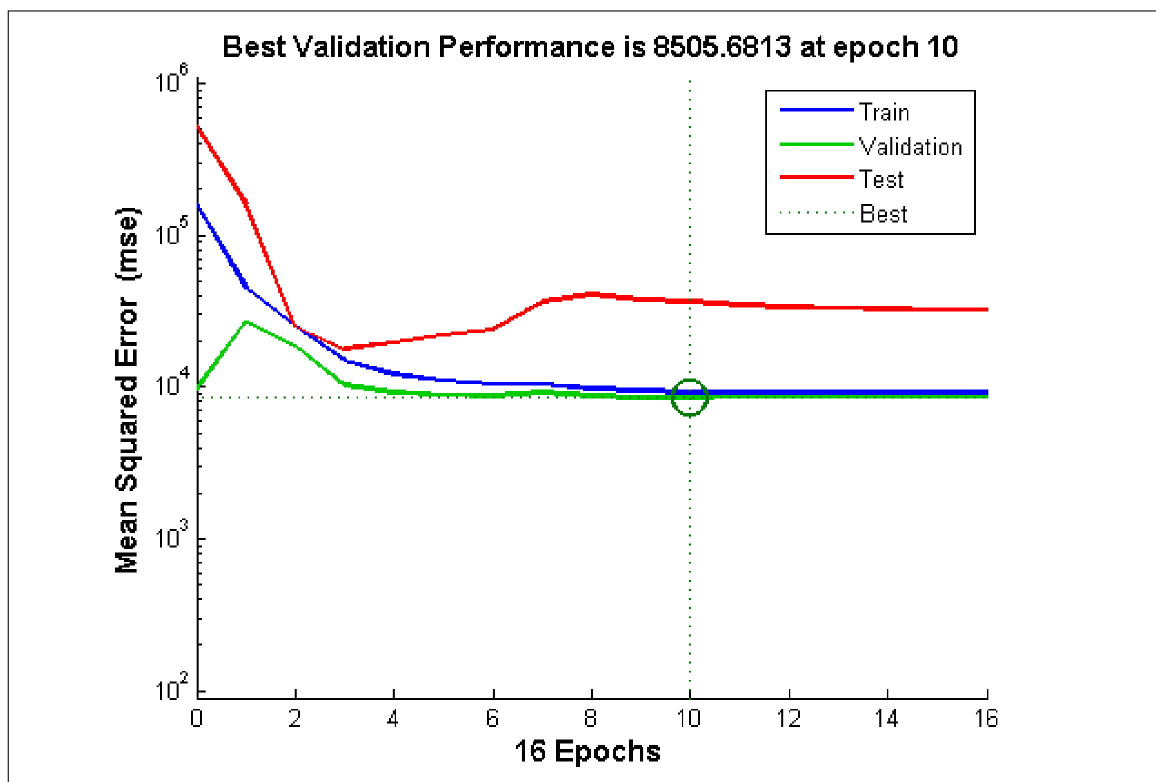

Figure 3.6: Neural Network Training Performance

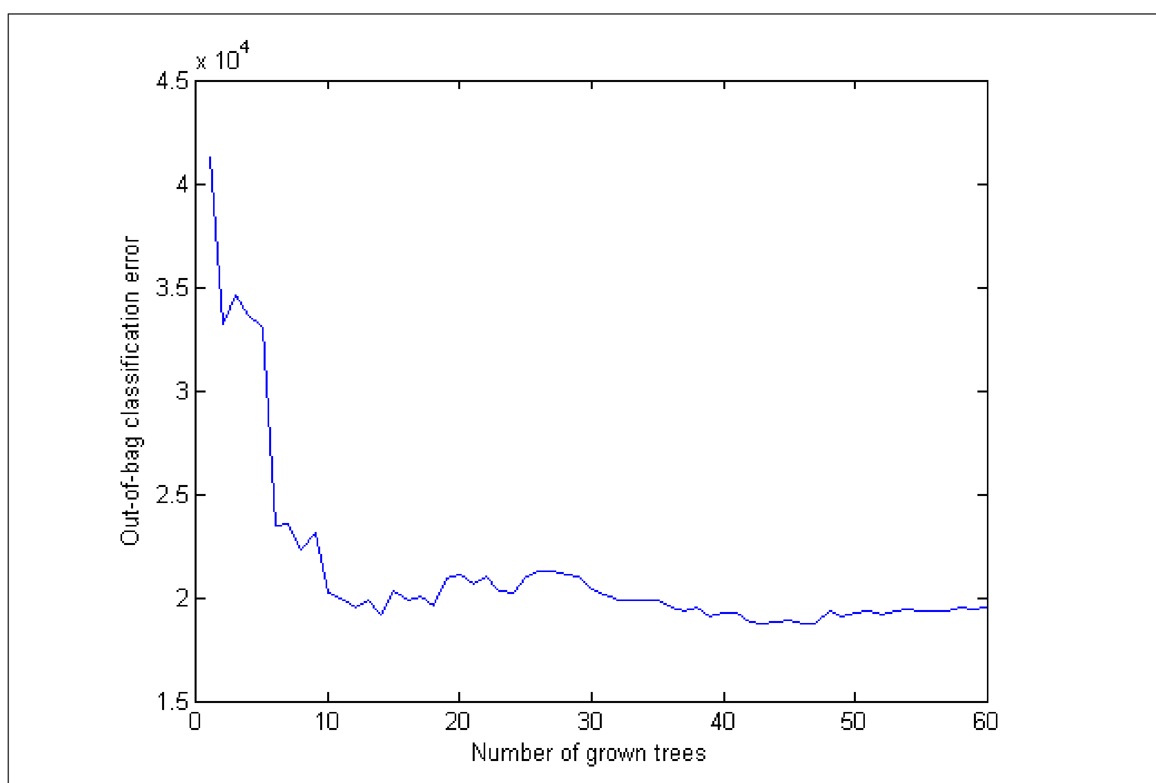

Figure 3.7: Out-of-bag error using 60 trees

Chapter 4

Evaluation

In this chapter we present how well our models predicted the execution time in each platform, compared with the actual measured times.

As target applications we used the Scalable HeterOgeneous Computing (SHOC) Benchmark Suite [16]. Hardware configuration was the same, using an NVIDIA GTX480 GPU and an Intel Xeon E5645 CPU.

There were 20 kernels available, that were used as our target in the case of Intel to OpenCL prediction. In the case of CUDA to Intel OpenCL prediction we had only 10 kernels available, because not all were written in the same way between OpenCL and CUDA making the prediction impossible.

4.1 Intel Xeon OpenCL to GTX480 CUDA prediction results

In figure 4.1 we present the predicted times vs the measured times. We got our best results using only the first 6 hardware counters with the highest correlation values. Also because the execution times on each kernel was really small (0.008 - 0.3 ms) and we observed a high divergence between the predicted and

real times, we increased the number of iterations of each kernel in order to increase the execution times. The formula for linear prediction with 6 hardware counters is: $Execution_time = -2.9 * 10^{-8} * Branch_instructions_taken - 1.24 * 10^{-9} * Total_cycles - 1.12 * 10^{-6} * Branch_instructions_correctly_predicted + 7.865 * 10^{-9} * Instructions_completed - 4.81 * 10^{-9} * Load_Instructions + 1.08 * 10^{-6} * Branch_instructions + 2.0829$

In figures 4.2 and 4.3 we present predictions of the linear models using more hardware counters, 7 and 8 respectively, but as the reader can see, the prediction times were not close to the real execution times.

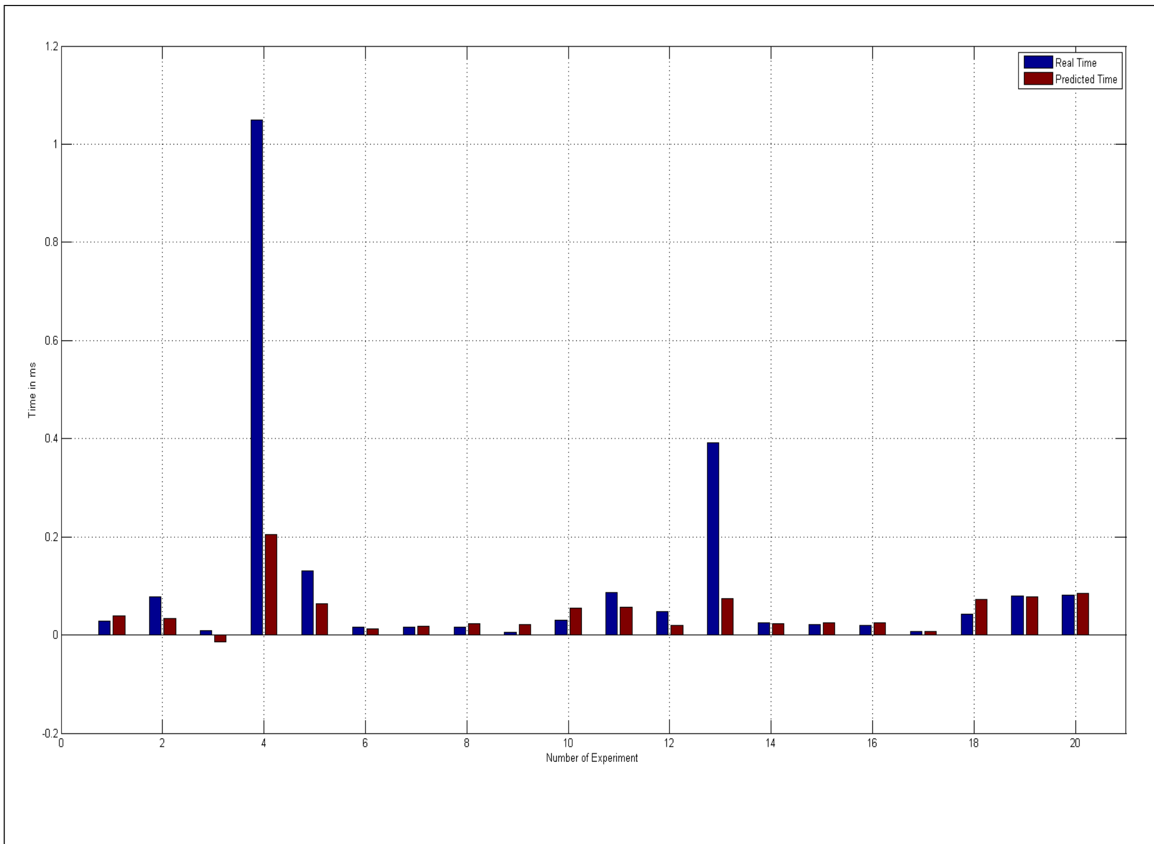

Figure 4.1: Linear prediction results using 6 hardware counters - Intel to CUDA

The full list of each experiment number shown in the graph and the application name can be seen in table A.4. 6 out of 20 kernels tested were correctly predicted with a small deviation, between 1% and 9% and an average of 5%. 4 out of 20 kernels had a average deviation of 20% and the rest 50%. 2 kernels out of 20, number 4 and 13 in figure 4.1, presented a huge deviation. This was caused by a high number of resource stalls, above 50% of total cycles measured for kernel #4 and 17% for

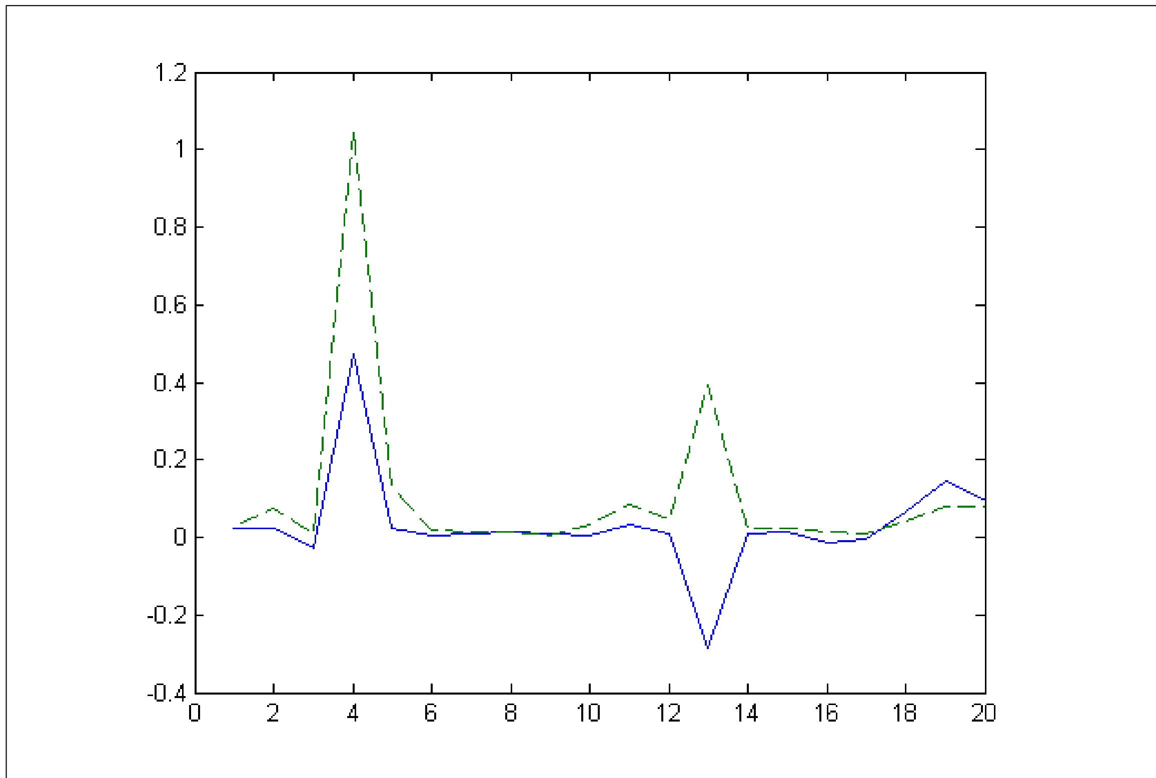

Figure 4.2: Linear prediction results using 7 hardware counters - Intel to CUDA

kernel #13, making the model to fail. When we tried to put the resource stalls as a parameter to the model, the prediction failed in all cases.

Non linear regression, although having zero RMSE, the values predicted where much higher than the real ones, with the difference being close to 1000ms. Same results where generated using the neural network model, making the two models unusable.

Random forest prediction was closer to the real execution times, relatively to neural networks and non-linear regression, but again the model was much worse than the linear regression model we presented earlier. We got the best results, shown in figure 4.5, using 60 trees, but the average deviation was above 100%.

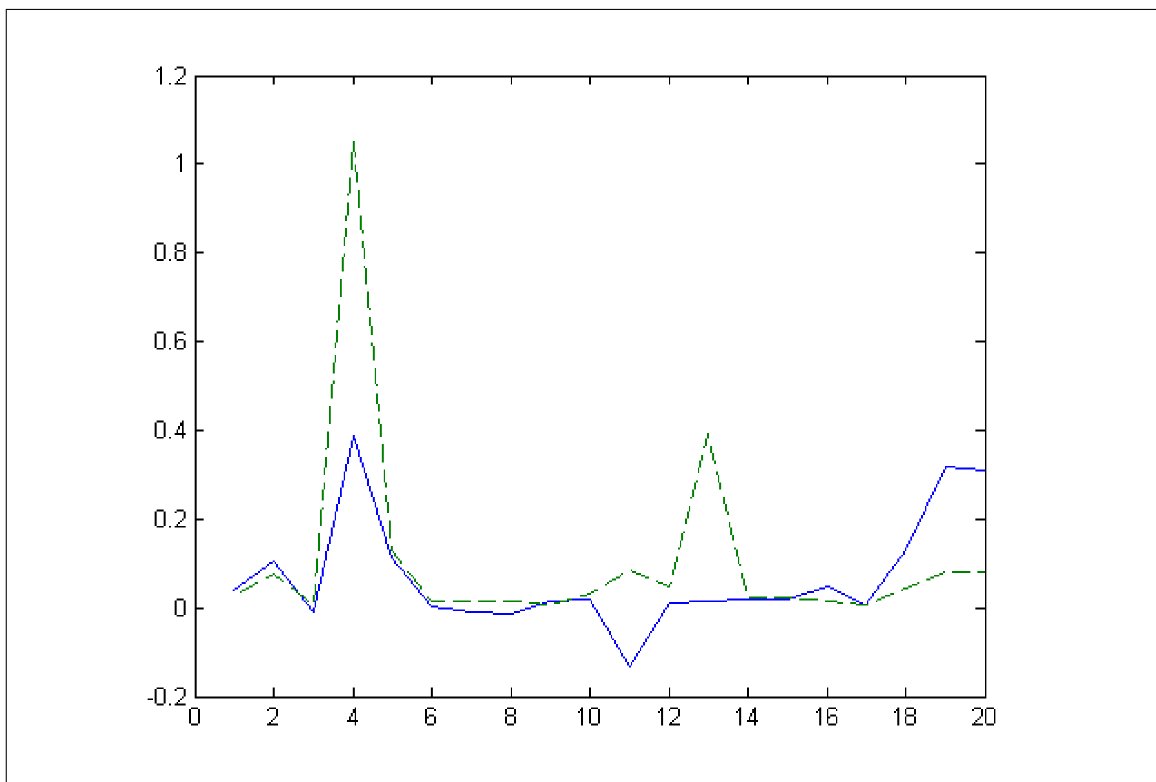

Figure 4.3: Linear prediction results using 8 hardware counters - Intel to CUDA

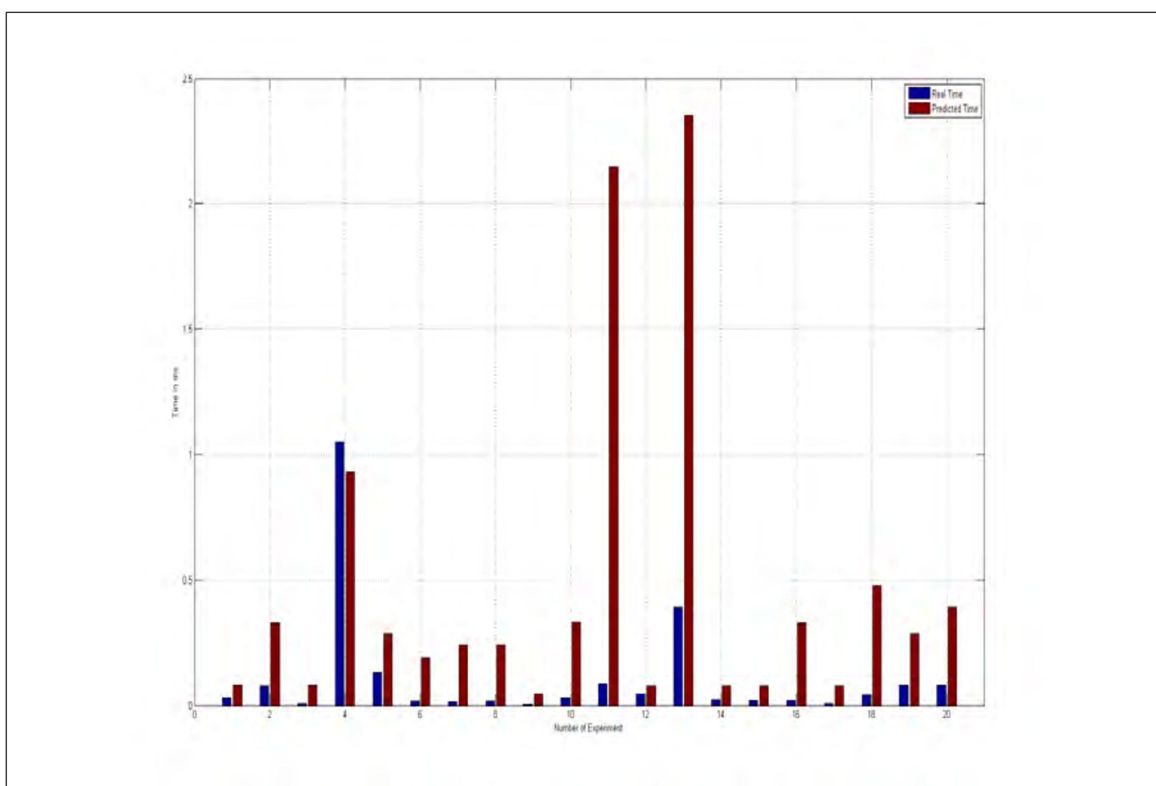

Figure 4.4: Random Forest Prediction with 60 trees - Intel to CUDA

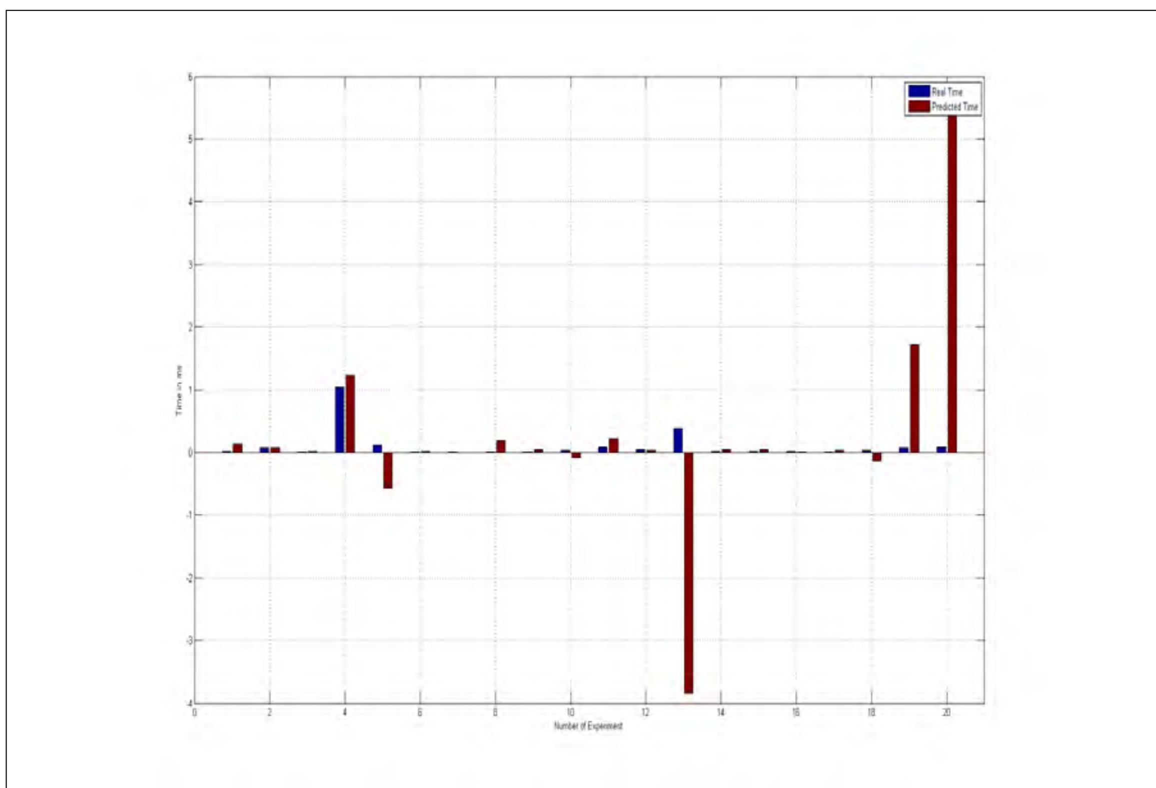

Figure 4.5: Neural Network Prediction - Intel to CUDA

4.2 GTX480 CUDA to Intel Xeon OpenCL prediction results

In the case of CUDA to Intel OpenCL prediction, we had a smaller target set, containing only 10 kernels. Below we present the results for each statistical method used.

Using linear regression 5 out of 10 kernels where predicted successfully with an average deviation of 5.8% and the other 5 where not predicted successfully, as seen in 4.6. The final formula for predicting the execution time using linear regression is: $Execution_time = -2.12 * 10^{-5} * lsd_executed + 65.06 * 10^{-5} * inst_issued - 65.9 * 10^{-5} * issue_slots + 1.07 * 10^{-5} * inst_executed + 0.4 * 10^{-5} * ldst_issued + 7.96 * 10^{-5} * cf_issued - 7.89^{-5} * cf_executed + 0.37^{-5} * l2_read_transactions - 35.8l^{-5} * l2_write_transactions + 43.4^{-5} * dram_write_transactions + 14.791$

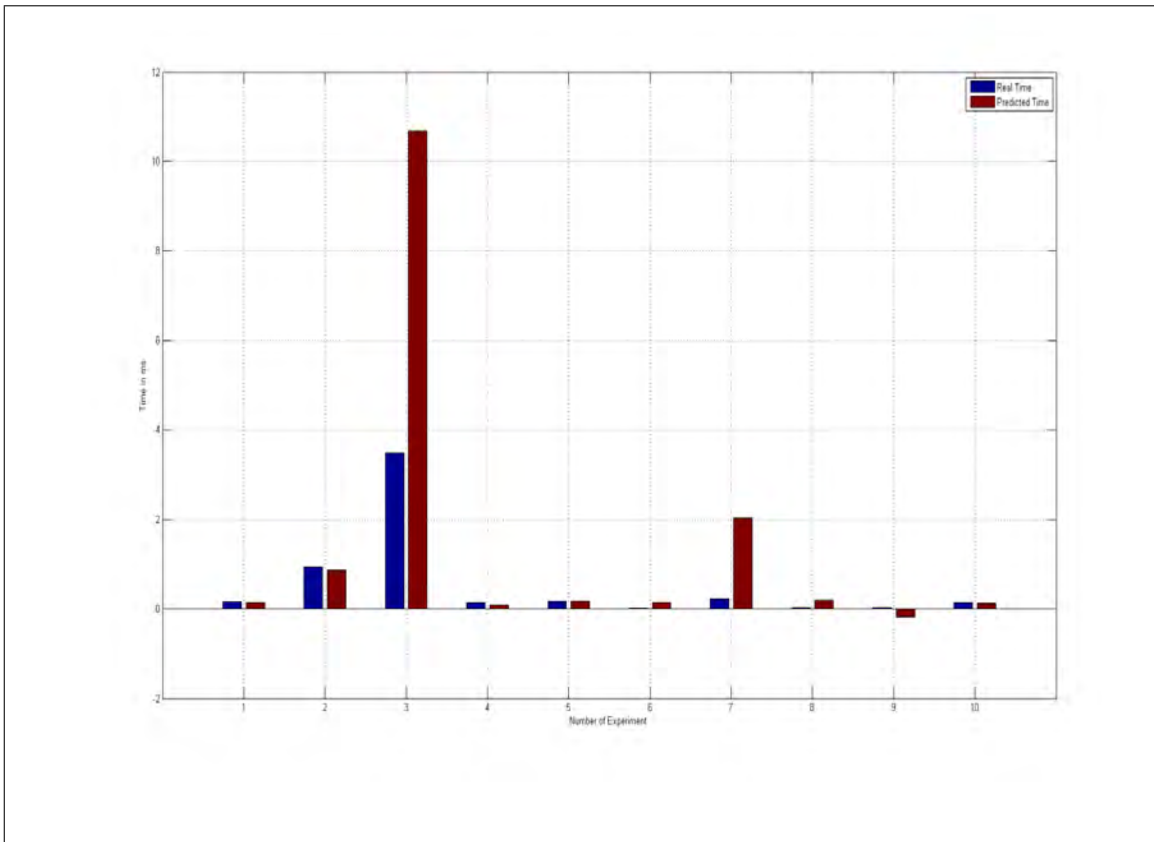

Figure 4.6: CUDA to Intel OpenCL linear prediction results

As in the case of Intel OpenCL to CUDA prediction, non-linear regression and neural

networks 4.7, generated poor results. Again random forests prediction results were much closer to the real times measured compared to neural networks, but the error was high as seen in figure 4.8. The main reason for this difference was that we used CUDA applications in order to predict the execution time on Intel architecture. Even though the applications were the same, we do not have full knowledge of the under-laying tools used to form and execute the binary to the target GPU device, such as the compiler and the runtime. Thus, event the smallest difference would affect the prediction result.

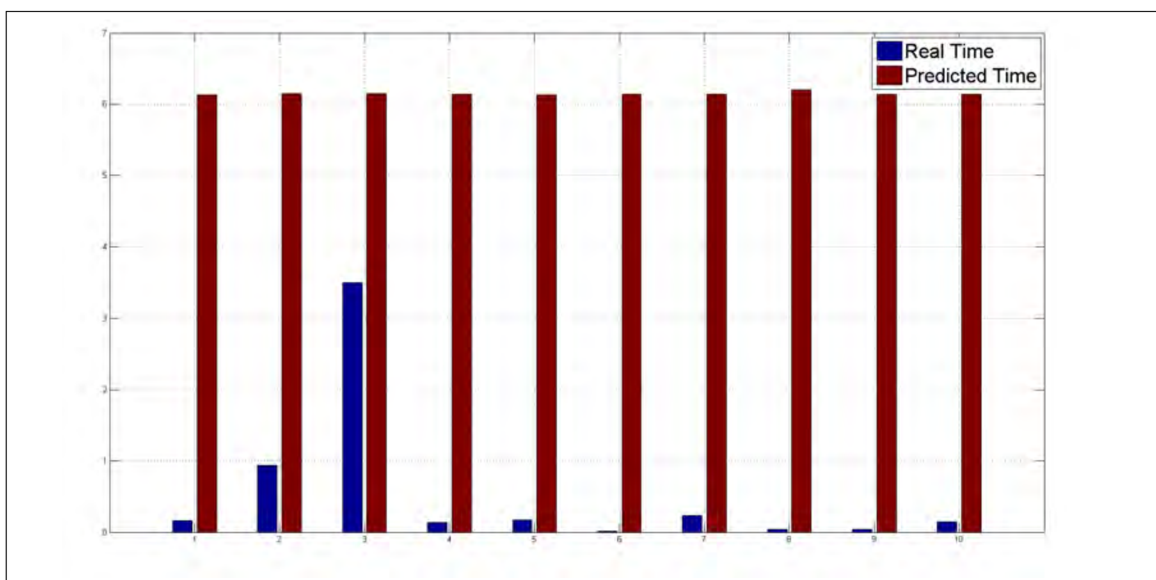

Figure 4.7: CUDA to Intel OpenCL prediction results using neural networks

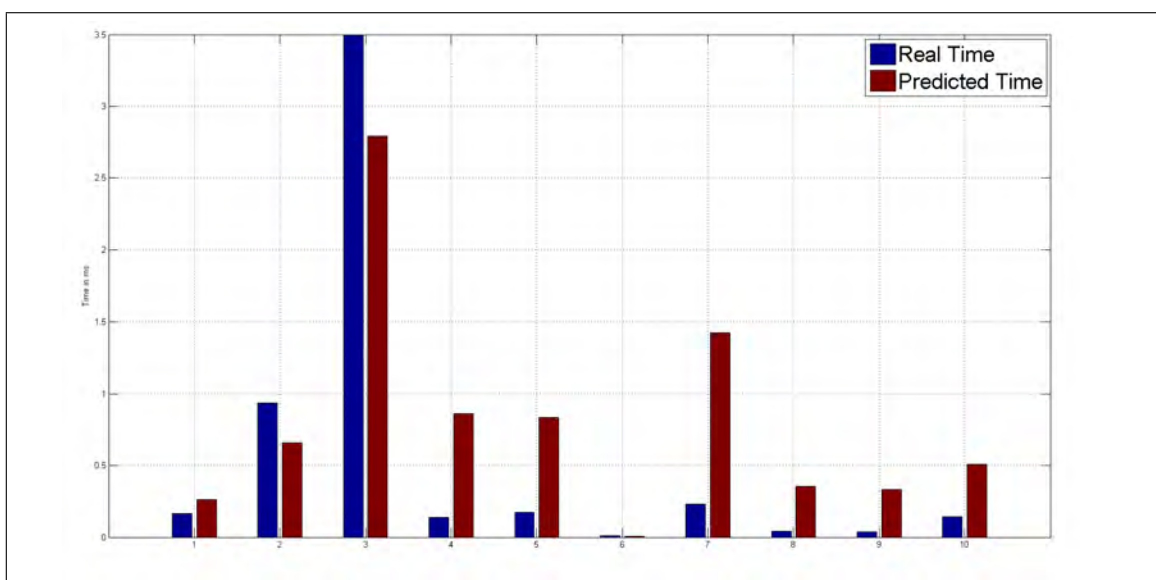

Figure 4.8: CUDA to Intel OpenCL prediction results using random forests

Chapter 5

Conclusion and Future Work

Today programmers are not able to exploit all the available computational power on a heterogeneous system. Due to the different architectures, they have to manually optimize their programs depending the target device and run benchmarks to find where the application will be executed better.

We addressed this problem by creating statistical models that predicts the execution time on different hardware architectures and different programming models. This way we can predict the execution time on a different device on a heterogeneous system at run-time and using this information we can migrate the executing application to a more suited device, increasing the overall performance.

This thesis is the first step towards performance prediction for heterogeneous systems. Our work has several limitations which we hope to address in the future.

First, our models are not always correct making them unstable. The main reason was the small training set we used and we hope to increase it by using more applications. Also we would like to include to our model ATI GPUs which support OpenCL and provide a simple API, GPUPerfAPI [4], that can measure hardware events on ATI GPUs.

Future work also includes testing our model with a run-time system such as GOpenCL [25] in order to measure the overall performance gain on a heterogeneous system.

Finally we want to include power prediction to our models using hardware events. This way we can not only increase the performance of a heterogeneous system but also decrease the power consumption.

Bibliography

[1] https://www.khronos.org/opencl/ *The open standard for parallel programming of heterogeneous systems*

[2] http://icl.cs.utk.edu/papi/ *Performance Application Programming Interface*

[3] http://docs.nvidia.com/cuda/profiler-users-guide/ *CUDA Profiler*

[4] http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/GPUPerfAPI-UserGuide.pdf *GPUPerfAPI*

[5] http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-phi-coprocessor-datasheet.pdf *Intel® Xeon Phi^{TM} Coprocessor*

[6] http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf *Kepler Compute Architecture White Paper*

[7] http://docs.nvidia.com/cuda/cuda-c-programming-guide/ *CUDA C Programming Guide*

[8] https://software.intel.com/sites/products/documentation/ioclsdk/2013/OG/Intel_R__SDK_for_OpenCL_Applications_2013_-_Optimization_Guide.pdf *Intel SDK for OpenCL Applications 2013*

[9] http://en.wikipedia.org/wiki/Linear_regression *Linear Regression*

[10] http://en.wikipedia.org/wiki/Heaviside_step_function *Heaviside step function*

[11] http://en.wikipedia.org/wiki/Artificial_neural_network *Artificial neural network*

[12] http://en.wikipedia.org/wiki/Random_forest *Random forest*

[13] http://en.wikipedia.org/wiki/Overfitting *Overfitting*

[14] http://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient *Spearman's rank correlation coefficient*

[15] http://en.wikipedia.org/wiki/Correlation_and_dependence *Correlation and dependence*

[16] https://github.com/vetter/shoc/wiki *The Scalable HeterOgeneous Computing Benchmark Suite*

[17] Ying Zhang, Lu Peng, Bin Li, Jih-Kwon Peir, Jianmin Chen (2011), *Architecture Comparisons between Nvidia and ATI GPUs: Computation Parallelism and Data Communications* , Louisiana State University.

[18] Dmitrijs Zaparanuks, Milan Jovic, JMatthias Hauswirth (2008), *Accuracy of Performance Counter Measurements* , University of Lugano.

[19] Gilberto Contreras, Margaret Martonosi (2005), *Power Prediction for Intel XScale Processors Using Performance Monitoring Unit Events* , Princeton University.

[20] Yuwen Sun, Lucas Wanner, Mani Srivastava (2012), *Low-cost Estimation of Sub-system Power* , University of California Los Angeles.

[21] Leo T. Yang, Xiaosong Ma, Frank Muelle (2005), *Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution* , North Carolina State University, Oak Ridge National Laboratory.

[22] Gabriel Marin, John Mellor-Crummey (2004) *Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models* , Department of Computer Science Rice University.

[23] Karan Singh, Major Bhadauria, Sally A. McKee (2008) *Real Time Power Estimation and Thread Scheduling via Performance Counters* , Cornell University, Chalmers University of Thechnology.

[24] Andreas Resios (2011) *GPU performance prediction using parametrized models* , Utrecht University.

[25] Konstantis Daloukas, Christos D. Antonopoulos, Nikolaos Bellas (2010) *GLOpenCL: Compiler and Run-Time Support for OpenCL on Hardware- and Software-Managed Cache Multicores* , University of Thessaly.

Appendix A

Tables

A.1 PAPI counters

Counter Name	**Description**
PAPI_BR_CN	Conditional branch instructions
PAPI_BR_INS	Branch instructions
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_BR_NTK	Conditional branch instructions not taken
PAPI_BR_PRC	Conditional branch instructions correctly predicted
PAPI_BR_TKN	Conditional branch instructions taken
PAPI_BR_UCN	Unconditional branch instructions
PAPI_FP_INS	Floating point instructions
PAPI_FP_OPS	Floating point operations
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L1_ICA	Level 1 instruction cache accesses
PAPI_L1_ICH	Level 1 instruction cache hits
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L1_ICR	Level 1 instruction cache reads
PAPI_L1_LDM	Level 1 load misses

PAPI_L1_STM	Level 1 store misses
PAPI_L1_TCM	Level 1 cache misses
PAPI_L2_DCA	Level 2 data cache accesses
PAPI_L2_DCH	Level 2 data cache hits
PAPI_L2_DCM	Level 2 data cache misses
PAPI_L2_DCR	Level 2 data cache reads
PAPI_L2_DCW	Level 2 data cache writes
PAPI_L2_ICA	Level 2 instruction cache accesses
PAPI_L2_ICH	Level 2 instruction cache hits
PAPI_L2_ICM	Level 2 instruction cache misses
PAPI_L2_ICR	Level 2 instruction cache reads
PAPI_L2_LDM	Level 2 load misses
PAPI_L2_STM	Level 2 store misses
PAPI_L2_TCA	Level 2 total cache accesses
PAPI_L2_TCH	Level 2 total cache hits
PAPI_L2_TCM	Level 2 cache misses
PAPI_L2_TCR	Level 2 total cache reads
PAPI_L2_TCW	Level 2 total cache writes
PAPI_L3_DCA	Level 3 data cache accesses
PAPI_L3_DCR	Level 3 data cache reads
PAPI_L3_DCW	Level 3 data cache writes
PAPI_L3_ICA	Level 3 instruction cache accesses
PAPI_L3_ICR	Level 3 instruction cache reads
PAPI_L3_LDM	Level 3 load misses
PAPI_L3_TCA	Level 3 total cache accesses
PAPI_L3_TCM	Level 3 cache misses
PAPI_L3_TCR	Level 3 total cache reads
PAPI_L3_TCW	Level 3 total cache writes
PAPI_LD_INS	Load instructions
PAPI_LST_INS	Load/store instructions completed

PAPI_RES_STL	Cycles stalled on any resource
PAPI_SR_INS	Store instructions
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_TLB_TL	Total translation lookaside buffer misses
PAPI_TOT_CYC	Total cycles
PAPI_TOT_IIS	Instructions issued
PAPI_TOT_INS	Instructions completed

A.2 CUDA metrics

Counter Name	**Description**
l1_cache_global_hit_rate	Hit rate in L1 cache for global loads
branch_efficiency	Ratio of non-divergent branches to total branches expressed as percentage
l1_cache_local_hit_rate	Hit rate in L1 cache for local loads and stores
sm_efficiency	The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU
achieved_occupancy	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
gld_requested_throughput	Requested global memory load throughput

gst_requested_throughput	Requested global memory store throughput
executed_ipc	Instructions executed per cycle
sm_efficiency_instance	The percentage of time at least one warp is active on a specific multiprocessor
inst_per_warp	Average number of instructions executed by each warp
gld_transactions	Number of global memory load transactions
gst_transactions	Number of global memory store transactions
local_load_transactions	Number of local memory load transactions
local_store_transactions	Number of local memory store transactions
shared_load_transactions	Number of shared memory load transactions
shared_store_transactions	Number of shared memory store transactions
gld_transactions_per_request	Average number of global memory load transactions performed for each global memory load
gst_transactions_per_request	Average number of global memory store transactions performed for each global memory load
local_load_transactions_per_request	Average number of local memory load transactions performed for each local memory load

local_store_transactions_per_request	Average number of local memory store transactions performed for each local memory store
shared_load_transactions_per_request	Average number of shared memory load transactions performed for each shared memory load
shared_store_transactions_per_request	Average number of shared memory store transactions performed for each shared memory store
local_load_throughput	Local memory load throughput
local_store_throughput	Local memory store throughput
shared_load_throughput	Shared memory load throughput
shared_store_throughput	Shared memory store throughput
shared_efficiency	Ratio of requested shared memory throughput to required shared memory throughput expressed as percentage
flops_sp	Single-precision floating point operations executed
flops_sp_add	Single-precision floating point add operations executed
flops_sp_mul	Single-precision floating point multiply operations executed
flops_sp_fma	Single-precision floating point multiply-accumulate operations executed
flops_dp	Double-precision floating point operations executed
flops_dp_add	Double-precision floating point add operations executed

flops_ dp_ mul	Double-precision floating point multiply operations executed
flops_ dp_ fma	Double-precision floating point multiply-accumulate operations executed
flops_ sp_ special	Single-precision floating point special operations executed
stall_ inst_ fetch	Percentage of stalls occurring because the next assembly instruction has not yet been fetched
stall_ exec_ dependency	Percentage of stalls occurring because an input required by the instruction is not yet available
stall_ data_ request	Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding
stall_ sync	Percentage of stalls occurring because the warp is blocked at a _ _syncthreads() call
inst_ executed	The number of instructions executed
stall_ texture	Percentage of stalls occurring because the texture sub-system is fully utilized or has too many outstanding requests
stall_ othe	Percentage of stalls occurring due to miscellaneous reasons
inst_ replay_ overhead	Average number of replays for each instruction executed

shared_replay_overhead	Average number of replays due to shared memory conflicts for each instruction executed
global_cache_replay_overhead	Average number of replays due to global memory cache misses for each instruction executed
tex_cache_hit_rate	Texture cache hit rate
tex_cache_throughput	Texture cache throughput
dram_read_throughput	Device memory read transactions
dram_write_throughput	Device memory write transactions
gst_throughput	Global memory store throughput
gld_throughput	Global memory load throughput
warp_execution_efficiency	Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage
gld_efficiency	Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage
gst_efficiency	Ratio of requested global memory store throughput to required global memory store throughput expressed as percentage
local_replay_overhead	Average number of replays due to local memory accesses for each instruction executed
l2_l1_read_hit_rate	Hit rate at L2 cache for all read requests from L1 cache.

l2_texture_read_hit_rate	Hit rate at L2 cache for all read requests from texture cache
l2_l1_read_throughput	Memory read throughput seen at L2 cache for read requests from L1 cache
local_memory_overhead	Ratio of local memory traffic to total memory traffic between the L1 and L2 caches expressed as percentage
issued_ipc	Instructions issued per cycle
issue_slot_utilization	Percentage of issue slots that issued at least one instruction, averaged across all cycles
sysmem_read_transactions	System memory read transactions
sysmem_write_transactions	System memory write transactions
l2_read_transactions	Memory read transactions seen at L2 cache for all read requests
l2_write_transactions	Memory write transactions seen at L2 cache for all write requests
tex_cache_transactions	Texture cache read transactions
dram_read_transactions	Device memory read transactions
dram_write_transactions	Device memory write transactions
l2_read_throughput	Memory read throughput seen at L2 cache for all read requests
l2_write_throughput	Memory write throughput seen at L2 cache for all write requests
sysmem_read_throughput	System memory read throughput
sysmem_write_throughput	System memory write throughput
cf_issued	Number of issued control-flow instructions
cf_executed	Number of executed control-flow instructions

ldst_issued	Number of issued load and store instructions
ldst_executed	Number of executed load and store instructions
l1_shared_utilization	The utilization level of the L1/shared memory relative to peak utilization on a scale of 0 to 10
l2_utilization	The utilization level of the L2 cache relative to the peak utilization on a scale of 0 to 10
tex_utilization	The utilization level of the texture cache relative to the peak utilization on a scale of 0 to 10
dram_utilization	The utilization level of the device memory relative to the peak utilization on a scale of 0 to 10
sysmem_utilization	The utilization level of the system memory relative to the peak utilization on a scale of 0 to 10
ldst_fu_utilization	The utilization level of the multiprocessor function units that execute global, local and shared memory instructions on a scale of 0 to 10
cf_fu_utilization	The utilization level of the multiprocessor function units that execute control-flow instructions on a scale of 0 to 10
tex_fu_utilization	The utilization level of the multiprocessor function units that execute texture instructions on a scale of 0 to 10

inst_issued	The number of instructions issued
issue_slots	The number of issue slots used

A.3 Rodinia benchmark application list

Application name	**Description**
backprop	Machine-learning algorithm that trains the weights of connecting nodes on a layered neural network
bfs	Breadth-first search
b+tree	Traversal algorithm
cfd	Unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow
gaussian	Gaussian Elimination
heartwall	Tracks the movement of a mouse heart over a sequence of 104 609x590 ultrasound images
hotspot	Estimate processor temperature based on an architectural floorplan and simulated power measurements
kmeans	Clustering algorithm used extensively in data-mining
lavaMD	Calculates particle potential and relocation due to mutual forces between particles within a large 3D space
leukocyte	Detects and tracks rolling leukocytes (white blood cells) in in vivo video microscopy of blood vessels
lud	LU Decomposition

myocyte	Models cardiac myocyte (heart muscle cell) and simulates its behavior according to the work by Saucerman and Bers
nn	Finds the k-nearest neighbors from an unstructured data set
nw	Nonlinear global optimization method for DNA sequence alignments
particlefilter	Statistical estimator of the location of a target object given noisy measurements of that target's location and an idea of the object's path in a Bayesian framework
pathfinder	Dynamic programming to find a path on a 2-D grid
srad	Diffusion method for ultrasonic and radar imaging applications based on partial differential equations
streamcluster	For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center

A.4 SHOC Applications list

Application name	**Graph number**
Breadth-first search	1
Triad	2-3
Molecular Dynamics	4
FFT	5-6
Sum Reduction	7
Parallel prefix sum	8-10

Radix Sort	11-13
Sparse matrix-vector multiplication	14-16
Stencil Calculation	17-18
Single precision general matrix multiply	19-20

Appendix B

Source Code Listing

Listing B.1: Attaching to the software threads

```
int init_papi_cpu ( cl_device_id device , cl_uint *compute_units,
    proc_threads **proccess_threads ) {
  int ret, number_of_threads;
  DIR *fd;
  struct dirent *in_file;
  proc_threads *next = NULL;

  /* Get number of compute units for the CPU */
  ret = clGetDeviceInfo( device, CL_DEVICE_MAX_COMPUTE_UNITS,
      sizeof(cl_uint), compute_units, NULL);
  if ( ret != CL_SUCCESS ) {
    printf("Error %d in clGetDeviceInfo\n", ret);
    return 0;
  }

  /* Get my pid and find all my threads so we *
   * can attach PAPI to theses threads     */
  int pid = getpid();
  char buffer[20];
  snprintf(buffer, 10,"%d",pid);
```

```
char *path = strdup ("/proc/" );
path = strcat ( path, buffer );
path = strcat( path, "/task/");

fd = opendir(path);
if ( fd == NULL ) {
  perror("opendir");
  exit(1);
}

#ifdef _DEBUG_
  printf("Proccess path: %s\n", buffer);
#endif

*proccess_threads = ( struct _proc_threads *) malloc ( sizeof( struct
    _proc_threads ) );
(*proccess_threads)->tid = pid;
next = (*proccess_threads);

number_of_threads = 0;
while ( (in_file = readdir(fd)) ) {

  /* Do not count parents and self */
  if (!strncmp (in_file->d_name, ".", sizeof(char)))
    continue;
  if (!strncmp (in_file->d_name, "..", sizeof(char) *2 ))
    continue;
  if (!strncmp (in_file->d_name, buffer, sizeof(in_file->d_name) ) )
    continue;

  #ifdef _DEBUG_
    printf("New thread id added: %d\n", atoi(in_file->d_name));
  #endif
```

```
    next->next = ( struct _proc_threads *) malloc ( sizeof( struct
        _proc_threads ) );
    next = next->next;
    next->tid = atoi(in_file->d_name);
    next->next = NULL;

    number_of_threads++;
  }
  return number_of_threads;
}
```