A Survey of Performance Modeling and Simulation Techniques for Accelerator-Based Computing

Unai Lopez-Novoa, Alexander Mendiburu, and Jose Miguel-Alonso, *Member, IEEE Computer Society*

Abstract—The high performance computing landscape is shifting from collections of homogeneous nodes towards heterogeneous systems, in which nodes consist of a combination of traditional out-of-order execution cores and accelerator devices. Accelerators, built around GPUs, many-core chips, FPGAs or DSPs, are used to offload compute-intensive tasks. The advent of this type of systems has brought about a wide and diverse ecosystem of development platforms, optimization tools and performance analysis frameworks. This is a review of the state-of-the-art in performance tools for heterogeneous computing, focusing on the most popular families of accelerators: GPUs and Intel's Xeon Phi. We describe current heterogeneous systems and the development frameworks and tools that can be used for developing for them. The core of this survey is a review of the performance models and tools, including simulators, proposed in the literature for these platforms.

Index Terms—Accelerator-based computing, heterogeneous systems, GPGPU, performance modeling

1 Introduction

Accelerator devices are hardware pieces designed for the efficient computation of specific tasks or subroutines. These devices are attached to a Central Processing Unit (CPU) which controls the offloading of software fragments and manages the copying and retrieval of the data manipulated in the accelerator. Most accelerators show important architectural differences with respect to CPUs: the number of computing cores, the instruction sets and the memory hierarchy are completely different, making accelerators suitable for specific classes of computation.

In the early 2000s, the High Performance Computing (HPC) community began using Graphics Processing Units (GPUs) as accelerators for general purpose computations [1], coining the term *General Purpose Computing on GPUs* (GPGPU) [2]. GPUs are designed for the efficient manipulation of computer images, and each new generation of devices arrives with significantly higher horsepower in terms of FLOP/s [3], [4]. The HPC community soon became aware of this potential, devising smart tricks to disguise scientific computations as graphics manipulations. GPU manufacturers noticed this trend and provided Application Programming Interfaces (APIs) and Software Development Kits (SDKs) to allow a more direct programming of their devices for non-graphics tasks, while simultaneously designing the newer GPUs taking into consideration the needs of this growing market. Discrete accelerators built around GPU chips but without video connectors were produced and rapidly adopted by the HPC community. Soon, other hardware manufacturers started building dedicated accelerator devices, shipping them with programming environments similar to those developed initially for GPGPU in order to ease its adoption by HPC developers.

Effectively exploiting the theoretical performance of an accelerator is a challenging task, which often requires the use of new programming paradigms and tools. Porting an application to an accelerator means extensive program rewriting, only to achieve a preliminary, not really efficient implementation. Optimization is even more complex. This complexity is exacerbated when simultaneously trying to use the aggregated performance of a processor and the accelerator(s) attached to it, not to mention massively parallel systems with thousands of hybrid nodes. Several works expound that it is hard to efficiently use (homogeneous) massively parallel computing systems [5], and even harder to deal with heterogeneous, accelerator-based supercomputers [6].

An added difficulty is that porting codes to use accelerators could be useless if the target application does not fit into a massively data parallel model. We can find highly successful porting cases [7], which may compel programmers to jump onto the accelerator bandwagon. However, metrics about the effort required for the porting are not that common. Besides, the way of measuring the degree of success can be misleading. In particular, it is a common practice to measure speedup against a serial version of the code. Works such as [8] claim that achieved speedups are not that large if the yardsticks are multi-core fine-tuned applications.

A common pitfall when developing for accelerators is to carry out code implementation and tuning using a trial-and-error methodology, without appropriate feedback and

• *The authors are with the Intelligent Systems Group, Department of Computer Architecture and Technology, University of the Basque Country UPV/EHU. P. Manuel Lardizabal 1, 20018 Donostia-San Sebastian, Spain. E-mail: {unai.lopez, alexander.mendiburu, j.miguel}@ehu.es.*

Manuscript received 20 May 2013; revised 19 Dec. 2013; accepted 30 Dec. 2013. Date of publication 24 Feb. 2014; date of current version 5 Dec. 2014. Recommended for acceptance by S. Aluru.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2308216

guidance from performance tools. Unfortunately, in the field of accelerator-based computing there is no outstanding tool or model that can be considered as the reference instrument for performance prediction and tuning. There is, though, an extensive body of literature related to this (relatively) novel area. The main objective of this survey is precisely to compile, organize and analyze this literature. We are not aware of a work similar to this in terms of breadth. We are convinced that this work will be useful for developers that want to extract the best possible performance (or performance/power tradeoff) of the widely available accelerator-based platforms.

The remainder of this paper is organized as follows. In Section 2 we provide some background on hardware, development tools and modeling methodologies for accelerator-based computing. Section 3 is devoted to the review of the literature on performance and power models and simulators targeting accelerators. Finally, in Section 4 we provide some conclusions and discuss future research lines in this field.

2 Background

In this section we provide a summary of the background information required to understand the analysis of performance models carried out in Section 3, which is the core part of this survey. The reader is referred to Appendices A to E of the Supplementary File, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2308216, for more details and additional references.

2.1 Accelerators and Heterogeneous Architectures

This survey focuses on heterogeneous hardware, in the form of (1) hybrid chips integrating several cores of different characteristics, including specific-purpose accelerators, and (2) traditional computing platforms to which a discrete accelerator is attached using, for example, PCIe. Nowadays, two classes of accelerators are mainstream: GPUs and many-cores such as Intel's Xeon Phi coprocessor. Other possibilities do exist, such as Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs).

Most of the literature on performance analysis for accelerator-based computing is devoted to analyze discrete GPUs and many-cores (specifically, the Xeon Phi), and therefore we do the same in this survey. Devices targeting GPGPU have been marketed since mid-2000s, while the Phi reached the market in 2012. This fact justifies the bias of the literature towards GPUs.

A GPU is an autonomous computing system composed of a set of processing cores and a memory hierarchy. A global memory space is accessible to all the cores, which can be exclusive (this is the common case if the GPU is a discrete accelerator plugged into a PCIe slot) or shared with other processing elements in the system, including the CPU (which is the common case when using a heterogeneous, multi-core chip). The strong point of GPUs is the way they handle thousands of in-flight active threads, and make context switching among them in a lightweight way. Running threads may stall when trying to access global memory, a relatively expensive operation. GPUs hide these latencies by rapidly context-switching stalled threads (actually, groups of threads) with active ones.

The Intel Xeon Phi is an accelerator that attaches to a host device using PCIe. It includes a many-core processor and memory. Current implementations incorporate up to 61×86 cores supporting four-way Simultaneous Multithreading (SMT) and Single Instruction, Multiple Data (SIMD) capabilities.

Regarding heterogeneous chips, we pay special attention to those integrating a traditional multi-core processor and a GPU that, in addition to be useful for graphics, can be used for offloading compute-intensive program sections. The main advantage of these chips is that memory is shared by GPU and CPU, therefore avoiding the use of expensive PCIe transfers.

2.2 Development Tools for Accelerators

Applications for accelerators are generally written using software development tools provided by the device manufacturers. They can consist of manufacturer-specific tool chains, or can be implementations of standard APIs. OpenCL [9] is a standardized, vendor-neutral framework for programming all classes of accelerators, defining a hardware model and an API. CUDA [10] implements similar concepts, but it is specific for NVIDIA GPUs and uses a slightly different terminology. Both models are widely used for GPGPU.

Compared to GPUs, the Xeon Phi is a relative newcomer to the accelerator arena. However, it has inherited from Intel's vast experience with multi-core processing, and the result is the availability of a diverse collection of tools that makes the learning curve of the developer less steep. Among other APIs, Phi developers can use OpenMP, OpenCL and MPI [11].

It is important to stress that the use of standard APIs may provide code portability between different platforms, but this does not translate into performance portability: currently, device specific, performance-oriented fine-tuning of an application is required in order to fully exploit the capabilities of an accelerator.

2.3 Tools and Techniques for Performance Modeling

Tools such as profilers and debuggers, together with best practices manuals written by the manufacturers of accelerators, are of great help during the process of porting and tuning an application. However, they lack prediction abilities: they cannot estimate performance on a different platform, or for a new application. Additionally, these tools can overwhelm the programmer with excessive information, making it difficult to filter out what is actually limiting the performance.

Our focus in this paper is on models of parallel applications running on heterogeneous platforms, that can be used to analyze (and predict) their combined performance. These models require input data (the characteristics of the target application and platform), and may provide different sets of output data, such as performance estimations or a list of bottlenecks.

In this section we describe methods to characterize devices and applications in order to feed performance models. We also describe the different classes of models that have been proposed and used in the literature: analytical, based on machine learning (ML), and based on simulation. Once a model is proposed, it must be validated and evaluated in order to demonstrate its usefulness beyond a single application/platform pair. A common way of doing this validation is through benchmark applications.

2.3.1 Characterizing Devices

Static information about the characteristics of a particular device can be collected in several ways, including manufacturer data sheets and manuals. Often, a manufacturer-specific API is available to directly query the device. To gather dynamic information about the way an application is using hardware resources, performance counters can be used. Latest chips also provide counters to obtain data about temperature and power usage.

A complementary source of information is that obtained through micro-benchmarks, programs designed to stress a particular component of the hardware and return performance metrics about it. This technique has been used extensively for CPUs, and has been extended to accelerators.

2.3.2 Characterizing Applications

Application characterization is the task of describing a program (or kernel) using metrics that can be used to feed a performance model. The accuracy of the model will depend on the quality of the captured metrics, and the metrics to extract from a program depend on the needs of the model they will feed. Usually, metrics include memory access patterns, register utilization, number of arithmetic operations, number of branches, etc. Program metrics can be obtained using a variety of mechanisms, including:

- Analysis of the source code, directly or after some transformations.
- Analysis of an intermediate representation (IR) of the code, an assembly-like code representation produced by the compiler before the generation of the binary files. A drawback of this approach is that, as it relies on the potential optimizations that the compiler might have performed, it may not exactly represent the original source code. CUDA's PTX format is a commonly used IR, that can be analyzed using the Ocelot [12] tool set.
- Analysis (disassembling) of the final binary files. These are the ones executed by the devices, so they provide maximum information about what the hardware will do.

2.3.3 Modeling Methodologies

A performance model is designed with the aim of describing the behavior of a system. A good model has prediction abilities: providing as input descriptions of an application and a platform, it generates estimators of the performance that would be achieved. After a review of the literature, we have identified three main approaches to modeling the performance of parallel systems: analytical modeling, machine learning and simulation.

An analytical model is an abstraction of a system in the form of a set of equations [13]. These equations try to represent and comprise all (or most) of the characteristics of the system. Machine Learning [14] is a branch of artificial intelligence related to the study of relationships between empirical data in order to extract valuable information. These techniques learn and train a model, for example a classifier, based on known data. Later, this model can be used with new (unseen) data to classify it. Finally, simulators are tools designed to imitate the operation of a system [13]. They are able to mimic system behavior step by step, providing information about system dynamics. The level of accuracy of the output information of the simulator depends, as in the other approaches, on the level of detail introduced in the model, taking into account that it is not always easy to exactly identify the actual way a system behaves.

Independently of the nature of the models, most have a set of parameters that must be tuned to suit the particular scenario to which they will be applied. Parameter values may be provided by experts, or measured in previous experiments but, in general, models are trained with data obtained from scenarios similar to the target one. For example, a ML model designed to predict execution times of a particular application on a particular hardware platform with a particular configuration (number of threads, thread grouping, input data, etc.) may take as training data set the execution times of applications running in the same or similar hardware for a variety of different configurations. If the data set is large and representative enough, one could expect promising performance estimations for the target application/hardware combination. However, the model would probably provide poor results for different applications or hardware. Benchmark suites can help to fine tune models, as they usually integrate a collection of diverse applications aimed to represent actual workloads that exploit different hardware features.

Additionally, a model will not be widely accepted unless it has been previously evaluated and tested for accuracy. Benchmark suites can also be used for this purpose, providing well defined yardsticks, easier to be accepted by the community than ad-hoc applications that could be considered unrealistic or barely representative of actual workloads.

3 A Review of Performance Models

Roughly speaking, a performance model can be seen as a system representation which provides output values based on a given set of input parameters. Depending on the characteristics of the model and the goal it has been designed for, the input and output data sets can be notably different.

Performance models can be classified according to different criteria. We already provided a classification in analytical models, machine-learning models and simulators. However, in this section we use a different classification criterion, based on the output generated by the model, that is, the information it provides about a particular hardware/software combination. We have identified models designed to:

1) predict the execution time of a target application on a target hardware platform
2) identify performance bottlenecks and propose code modifications to avoid them
3) provide estimations of power consumption
4) provide detailed, step-by-step information of resource usage, based on simulation

In the forthcoming sections we make a review of model proposals found in the literature, arranged using this classification. In Appendix F of the Supplementary File, available online, we have included a table for each model class, summarizing the main features of each model.

3.1 Execution Time Estimation

Estimating the execution time of parallel applications can be useful in several situations, such as making a decision about acquiring new hardware or testing an application for scalability. After a literature review of performance models aiming to predict the execution time of an accelerator-based parallel application, we have organized the different works taking into account whether they propose general models applicable to any program/kernel, or they are specific for a particular application or pattern. We have also taken into account the input information required by the model: actual kernel codes, or pseudocodes/patterns derived from the program structure.

3.1.1 General-Purpose Models

Models using kernel codes. As previously explained, GPUs hide latencies by handling thousands of in-flight active threads, rapidly switching stalled warps (due to memory transactions) with ready-to-run warps. A *warp* is the CUDA name of a group of 32 threads, the minimum scheduling unit in NVIDIA GPUs. Hong and Kim [15] model this behavior on NVIDIA GPUs using two different metrics: MWP and CWP. The first one, memory warp parallelism, measures the maximum number of warps that can overlap memory accesses. The second, computation warp parallelism, measures the number of warps that execute instructions during one memory access period. Expressions are provided to compute these values for a given device and application and, from them, the number of cycles required to run the kernel. The basic idea behind the model is that, when $MWP \leq CWP$, performance is dominated by the time spent accessing memory but, when $MWP > CWP$, memory accesses are mostly hidden and overall performance is dominated by the computation cycles. A variety of kernels and GPUs were used to validate the model. Authors report a geometric mean error of 13.3 percent between estimated and actual cycles per instruction (CPI). Due to its simplicity and accuracy reflecting the way GPUs work, this model has been used and extended by many other authors. For instance, Ganestam and Doggett use MWP-CWP in [16] as part of an auto-tuning tool for the optimized execution of a ray-tracing algorithm. We will see some additional examples throughout this paper.

Based on similar concepts, Kothapalli et al. [17] propose two kernel behaviors, also in a CUDA context: MAX model (maximum value between compute and memory cycles), and SUM model (sum of compute and memory cycles). Authors claim that for most of the tested cases, predictions provided by these SUM and MAX models are close to the real measurements but, unfortunately, they do not provide clues as to how to choose the right one for a given kernel. An extension of this work, presented by Gonzalez [18], includes more complex hardware features (memory copy overheads, branch-divergence latencies, etc.), and improves modeling accuracy, choosing in most of the cases the SUM model. As in the original work, no reason is given for this decision.

In addition to these analytical models, there are also others which rely on machine learning techniques identifying first a set of code and hardware features, and later using feature selection, clustering and regression techniques to estimate execution times. Kerr et al. [19] present four different models: application, GPU, CPU, and combined $\mathrm{CPU} + \mathrm{GPU}$. The paper states that the GPU model (estimation of application execution times on an unknown GPU) provides fairly accurate results, with a maximum deviation of 16 percent in the worst case. Unfortunately, the remaining models are not that accurate, producing poorer estimates and high variability. The proposal by Che and Skadron [20] has accuracy levels close to those of the MWP-CWP model [15] when used to predict the execution time of different applications on the GPUs used to build the model. However, accuracy is not that good, although still reasonable (between 15.8 and 27.3 percent), when estimating execution time on an unknown, new GPU. Finally, the paper presented by Sato et al. [21] discusses different machine learning models, reporting for the best one error rates around 1 percent. However, the process used to calibrate the model is not detailed sufficiently.

Models using program skeletons. Given the (serial) source code of an application, it would be highly desirable to know its potential performance when running it in an accelerator. There are some performance-prediction frameworks that aim to do precisely this, although they do not take the code directly as an input: the user is required to provide a skeleton (abstract view) of the application using a set of constructors that identify parallelization opportunities. Grophecy, presented by Meng et al. [22], takes an application skeleton and generates several proposals for implementing the accelerated version of the code, with different types of optimizations, such as tiling and loop unrolling. The framework makes use of the above-described MWP-CWP model to estimate the execution time of each proposal and recommends the most promising one.

Nugteren and Corporaal propose in [23] the boat hull model, inspired by the well-known roofline model [24]. They claim that the roofline model is not designed to predict performance, but to help a programmer detect bottlenecks and improve performance (see Section 3.2.1). In contrast, the boat hull model does provide performance estimations using a few architectural parameters and a description of the code. The target program must be split into several sections, and each section must be characterized as belonging to a certain class. Then, a modified roofline model (performance graph) specific to each class is applied to the different sections, using in the X axis a "complexity" metric (instead of operations per byte), and in the Y axis an "execution time" metric (instead of FLOP/s). All together,

the model can be used to predict the performance of an application on different target architectures, including GPUs and other accelerators. The model is applied to an image processing application on two GPUs, and the difference between predicted and measured execution times is 3 percent in one device and 8 percent in the other.

The two proposals discussed in this section can be used to predict application performance without carrying out accelerator-specific implementations. However, users are required to describe application behavior in an abstract way, learning a new syntax and carefully splitting the code into sections that must be well-defined and properly associated to a collection of skeletons or classes. These are not trivial tasks, especially when handling large, complex pieces of code.

3.1.2 Models Designed for Particular Applications

The works discussed above can be considered general-purpose, that is, suitable to any application. We can find in the literature other proposals which focus on modeling specific programs or program classes (patterns). For example, Meng and Skadron propose in [25] a framework that automatically selects the best parameters for an Iterative Stencil Loops (ISL) application, given some parameters of the domain and some features of the target GPU. ISL is a technique applied in many domains, including molecular dynamics simulation and image processing, that distribute computations into overlapping regions (tiles), with neighboring regions interacting via halo zones. Similarly, Choi et al. present in [26] an auto-tuned implementation of the Sparse Matrix-Vector (SpMV) multiplication. The difficulty in computing with sparse matrices is related to using compression algorithms (such as BCSR or ELLPACK [27]) that represent matrix data as lists. Authors designed their own implementation of the Blocked ELLPACK (BELLPACK) algorithm and built it with a model that, given architectural features, finds the application parameters that minimize execution time.

3.2 Bottleneck Highlighting and Code Optimization

While a program is being ported and fine-tuned, it is necessary to carefully analyze its behavior at run time on the target platform, looking for potential resource bottlenecks and, if possible, finding alternatives to eliminate or mitigate them. Toolkits for programming accelerators commonly include a profiler (e.g., NVIDIA Visual Profiler for the CUDA platform, CodeXL for the AMD OpenCL platform or VTune for the Intel Xeon Phi platform), which should be the first tools to use when optimizing a kernel. They analyze code execution, spotting bottlenecks, and can even make recommendations to the programmer about code changes or compiler flags. However, they cannot predict the performance benefits associated to these changes. To this end, several models have been proposed in the literature with the aim of helping developers in code optimization tasks.

3.2.1 The Roofline Model

The work by Williams et al. [24] has provided a well-accepted mechanism to visually describe those characteristics of a multi-core platform that may limit the performance of a particular application. Using a diagram with two axes, representing an "Operational Intensity (FLOP/Byte)" metric in the X-axis, and an "Attainable GFLOP/s" metric in the Y-axis, the compute platform is represented by a line or *roof* that first grows linearly with the operational intensity, until it reaches an inflection point and then flattens. An application (or kernel) fits somewhere in the X-axis (has a certain operational intensity) and, consequently, can reach a maximum performance (limited by the line representing the platform). The programmer can try code optimizations aimed at increasing operational intensity, moving the kernel to the right side of the graph, until it reaches the flat section where further improvements are not fruitful.

The limits imposed by the platform may vary depending on the use of different features. For example, by using SIMD instructions or transcendental functional units, if available, peak performance (the flat portion) can be improved. Using memory accesses that make better use of memory channels (such as FastPath in AMD GPUs or data coalescing in NVIDIA GPUs) raises the growing section. Therefore, the platform is not characterized by a single line, but by a collection of lines. The particular roof limiting a given application depends on the use the programmer makes of the device features. In [28] Jia et al. make a roofline characterization of two GPUs: AMD HD5850 and NVIDIA C2050, showing how the set of features affecting performance is different for each GPU and, therefore, the code modifications programmers may apply in each case are also different. The roofline model has also been used by Cramer et al. to characterize the Xeon Phi running OpenMP codes [29].

3.2.2 Program Dissection

Models by Lai and Seznec [30], Zhang and Owens [31] and Baghsorkhi et al. [32] provide a breakdown of a CUDA kernel execution into several stages, such as compute phases or memory stalls, although they differ in the way they retrieve application-related features. In all cases, platform information is gathered via micro-benchmarks, while the application is characterized by code analysis. Both [30] and [31] require running the target kernel within a simulator to gather performance counters. The former also requires an analysis of the disassembled kernel binary file. In contrast, Baghsorkhi et al. [32] work with the source code of the kernel, building from it a program dependence graph (PDG) [33] that represents the workflow of the application.

Once the information characterizing device and application has been gathered, the models are constructed and used to generate different sorts of output (or performance predictions): Lai and Seznec [30] present statistics of some hardware features such as the level of contention for the scalar cores; Zhang and Owens [31] present a detailed breakdown of the memory instructions, showing the number of cycles spent in global or shared memory instructions, along with the number of bank conflicts for the shared memory; finally, Baghsorkhi et al. [32] derive from the PDG a breakdown of the instructions into compute instructions and memory accesses, pointing out the number of cycles spent in stages such as synchronization or branch divergence, as well as in the different memory stages.

Regarding accuracy values, Lai and Seznec [30] report an average error of 4.64 percent when predicting the number of cycles used by the kernels chosen for the experiments. Zhang and Owens [31] report a relative prediction error within 5-15 percent. The model by Baghsorkhi et al. [32] could be expected to be the least accurate one, because program information is obtained from the source code instead of from lower-level measurements. Authors do not report numerical values, but their experiments show that the model is quite accurate, highlighting the most time-consuming stage in each execution.

3.2.3 Detection of Specific Bottlenecks

When a profiler or some of the previously discussed models point to a particular issue hindering performance when a kernel is running on a device, hints are required as to how to overcome it. Several models have been proposed to characterize specific bottlenecks and, thus, could be of use in this context.

The work by Ramos and Hoefler [34] is focused on creating a performance model for the cache coherence protocols used in many-cores, assessing the ways they impact communication and synchronization. Authors show how their analytical model helps in defining highly-optimized parallel data exchanges. Using as target platform the Xeon Phi, their methodology is tested against vendor-provided primitives (OpenMP and MPI libraries by Intel), achieving up to $4.3\times$ faster communications.

Baghsorkhi et al. [35] present a memory hierarchy model for GPUs through a framework that predicts the efficiency of the memory subsystem. They predict the latency of main memory accesses, and the hits and misses in the L1 and L2 caches. This memory model is based on the hardware memory hierarchy of the Tesla C2050 card, and validations with this particular hardware provide good results: authors compare the predictions of their model against the hardware counters, obtaining average absolute errors of 3.4 percent for L1 read hit ratios, 1.9 percent for L2 read hit ratios and 0.8 percent for L2 write hit ratios.

Cui et al. [36] present a performance model focused on reducing the effects of control-flow divergence in GPU kernels. They propose a framework that retrieves a representation of a kernel workflow, from which different groupings and reorderings of threads are derived, choosing those minimizing divergence. Authors report speedups up to $3.19\times$ in their experiments in GPUs, but the method is tested with kernels that do not have strong inter-thread dependences or synchronization.

3.2.4 Selecting Code Optimizations

As stated before, manufacturers provide best practices manuals recommending different optimizations that can be applied to accelerate the execution times of kernels. Some of these optimizations can even be implemented automatically by compilers. For example, in Chapter 10 of [37] Rahman describes a methodology and some heuristics to find and optimize parallel code in the Xeon Phi. He provides a taxonomy of potential optimizations, relates the metrics that indicate the presence of bottlenecks and describes some good practices to remove them. However, choosing the right combination of optimizations is not trivial, because of possible negative interactions among them.

Sim et al. present in [38] two related contributions: an elaborated analytical model for GPU architectures, and a framework that suggests the most profitable combination of optimizations for a CUDA kernel in terms of performance. The model is an extension of MWP-CWP that takes into account many hardware features of recent GPUs that were not present in the original model. These improvements include considering the effect of the cache hierarchy, the presence of special function units (which, in NVIDIA GPUs, implement transcendental functions) and a more detailed effect of the instruction level/thread level parallelism of the kernels. The performance projection framework is a tool that points out the most promising optimizations for a given kernel. Authors claim that their model closely estimates the speedup of potential optimizations, although no numerical values are given. For a particular kernel they tested 44 combinations of different optimizations. The best one runs three times faster, compared to the baseline implementation.

3.3 Power Consumption Estimation

A trending topic in the HPC field is improving the power efficiency of computers. To that extent, significant effort is being devoted to modeling the power characteristics of systems. Application power modeling aims to estimate the energy required to run a particular code on a particular device. It has to consider not only code and device properties, but also program input and some other runtime characteristics. In this section we review a collection of proposals aiming to model power efficiency.

3.3.1 Standalone Models

Wang and Ranganathan [39] and Hong and Kim [40] propose analytical power models. Both require the execution of a set of micro-benchmarks and external power consumption meters to characterize the target devices (in both cases, NVIDIA GPUs). Both use PTX intermediate representation ([40] processes it through Ocelot) to obtain information about the target kernel. The models estimate the total power consumed by the kernel execution, and are aimed to identify the right number of multiprocessors and blocks that would provide the best performance/power ratio. It is worth mentioning that the proposal by Hong and Kim is an extension of their MWP-CWP performance model (discussed in paragraph "Models using kernel codes" in Section 3.1.1), and that model validations carried out by them report a geometric mean error of 8.94 percent between estimated and actual power consumption.

In a similar fashion, the work by Shao and Brooks [41] presents an energy model of the Xeon Phi based on information gathered from performance counters and on measurements by an external meter. Through microbenchmarking they characterize the energy consumed by each instruction type (scalar, vector, prefetch) with operands in different locations (registers, L1 cache, global memory, etc.), compiling an energy-per-instruction table. The model is validated through a collection of benchmarks, reporting 1-5 percent discrepancies between predicted and actual power use. This work was carried out

with a pre-release of the Xeon Phi. The current marketed product implements power-related counters that can be used to avoid instrumenting the accelerator to perform power metering.

Other proposals gather performance and power measurements and use this data to build machine learning models, see for example [42], [43], [44], [45]. Samples of performance counters and energy measurements are taken at a given frequency while the kernel runs on the device, characterizing power use along time. This allows identifying power and performance bottlenecks, as well as efficient power/performance configurations. A main drawback of these approaches is that performance counters are collected per warp/GPU multiprocessor, and therefore the models assume that the applications use all the computing resources in a GPU. The most recent of these works [45] uses the NVIDIA Fermi architecture, which implements power-related hardware counters. This particular model is based on artificial neural networks (ANNs) and reports quite accurate estimations (2.1 percent error rate when predicting power consumption and 6.7 perecnt when predicting execution time).

3.3.2 Models Tied to Simulators

System simulators, such as those that will be discussed in the forthcoming section, can also be enhanced to provide not only performance-related metrics, but also power-related estimations. To that extent, a power model has to be integrated in the simulator, to allow it to compute cycle-by-cycle use of resources, together with the power implications of that use.

We have identified two analytical models that work with GPGPU-Sim (see Section 3.4): GPUWattch by Leng et al. [46], and GPUSimPow by Lucas et al. [47]. Both are adaptations to GPUs of the McPat power modeling framework for multi-core architectures [48], but differ in the way GPU architecture and power use is modeled. In terms of accuracy, comparing model predictions with power use in actual GPUs, both are similar (using the data provided by the authors): for GPUWattch the error is 9.9-13.4 percent, and for GPUSimPow it is 11.7-10.8 percent. An interesting feature of GPUWattch is a module that can be used by GPGPU-Sim to simulate power-related runtime optimizations, such as emulating dynamic voltage and frequency scaling (DVFS). Authors claim that using DVFS, GPU energy consumption could be reduced up to 14.4 percent, with less than 3 percent of performance loss.

3.4 Simulation

A simulator is a system representation (model) able to mimic, step-by-step, the behavior of the target (real) system. Simulators are widely used to carry out performance studies of existing hardware and software platforms, and also to analyze platforms that either do not exist, or are not available. The accuracy of the output information provided by a simulator depends on many factors, among them being the level of detail with which the system has been modeled and the quality and detail of the workloads provided to feed the model.

3.4.1 Simulators of GPU-Based Accelerators

Two popular simulators for GPU-based accelerators are GPGPU-Sim [49], [50] and Barra [51], [52]. Both are functional simulators of NVIDIA GPUs, capable of running CUDA codes (GPGPU-Sim works also with OpenCL). A collection of user-configurable parameters is required in order to fine-tune the way the target device is modeled: number and features of multiprocessors, interconnection network and its topology, memory size and organization, etc. As output, the simulators generate detailed step-by-step information about performance counters, memory accesses, resource utilization, etc.

Regarding simulation accuracy, authors state that GPGPU-Sim v3.1.0 reaches an accuracy of 98.3 percent for the instructions per cycle metric when simulating a NVIDIA GT-200 card and 97.3 percent for a Fermi card. For Barra, authors carry out some experiments comparing simulator-predicted values with hardware counters and, in most cases, discrepancies range between 0 and 23 percent. Only for one benchmark and a particular counter (measuring memory transactions), the discrepancy reaches 81.58 percent.

One of the weakest points of simulators is their execution speed: depending on the level of simulation detail, slowdown (comparison against execution on actual hardware) can be severe. GPGPU-Sim provides two ways of working: a fast functional way, which only executes the application and generates its output, and a detailed way (5 to 10 times slower), which collects performance statistics. However, the latter is the really useful mode: the fast one should be used only to verify that the code being analyzed runs in the GPGPU-Sim environment. Barra has been designed as a multi-core enabled program and, therefore, if its execution is slow, it can be accelerated by adding more cores: reported experiments show excellent scalability for up to four cores.

The flexibility of simulators, and the rich and detailed information they provide, have made them a tool of choice to feed other models. They can be used also as a complement to profilers, in order to drive performance optimizations (see for example [53], [54]).

3.4.2 Simulators of Hybrid Architectures

The simulators described in the previous section focus on kernels running on an accelerator (GPU) device. Other simulators go a step further, modeling hybrid CPU+GPU architectures that run heterogeneous applications, either as separate devices (CPU connected to GPU via PCIe, with separate RAMs) or as a fused chip (GPU+CPU, sharing the RAM). Three simulators in this class are MacSim [55], FusionSim [56] and Multi2Sim [57]. In order to narrow down the description, we will focus on the last one, which is under active development and whose features increase with each release. Currently, Multi2Sim supports simulations of systems integrating x86, MIPS-32 and ARM CPUs; AMD and NVIDIA GPUs; a memory hierarchy (per device and/or shared) and an interconnection network. It is possible to run a detailed, architectural simulation that mimics the use of hardware resources, but also a faster emulator that just replicates the behavior of program instructions. Tools are provided to pre-process the target applications in

order to run them in the simulated environment, with support for CUDA and OpenCL. A visual tool is provided to interactively follow (and pause/analyze/resume) a simulation. When assessing Muti2Sim accuracy, authors report errors between 7 and 30 percent estimating the execution cycles of OpenCL kernels on an AMD GPU. The large discrepancies are due to the lack of fidelity in the way the memory subsystem is simulated.

4 Conclusions

The appearance of new computing devices and the design of new algorithms in different fields of science and technology is forcing a fast evolution of HPC. Designing and developing programs that use currently available computing resources efficiently is not an easy task. As stated in [58], present-day parallel applications differ from traditional ones, as they have lower instruction-level parallelism, more challenging branches for the branch-predictors and more irregular data access patterns. Simultaneously, we are observing a processor evolution towards heterogeneous many-cores. The goal of these architectures is twofold: providing an unified address space that eliminates the need to interchange data with an external accelerator using a system interconnect and improving power efficiency reducing the total transistor count.

Tools to help programmers in this parallel and heterogeneous environment (debuggers, profilers, etc.) are slowly becoming available, but the programmer still needs to have an in-depth knowledge of the devices in which applications will run if performance and efficiency have to be taken into consideration. The performance models that have been proposed in the last years, and that we have tried to characterize in this review, aim to make the process of choosing the right options (device, program settings, optimizations, etc.) easier in order to efficiently run accelerator-based programs. From all the tools analyzed in this paper, a few of them stand out: MWP-CWP [15] as the model of choice to predict execution times on GPUs; the roofline model [24] to determine the factor limiting program performance in parallel systems (including accelerators); and the GPGPU-Sim simulator [49] of GPU-based accelerators, that can be enhanced with a power model. All currently available models have limitations, but they will be the foundation on which better tools will be constructed. In the following list we analyze some of these limitations, and different ways to overcome them.

- There is no accurate model valid for a wide set of architectures. Each model finds a different tradeoff between being more device-specific and therefore more accurate (e.g., [38]), or being more general purpose at the cost of losing accuracy (e.g., [23]). Related to this topic is the fast pace at which manufacturers market new products, with new or improved features, making models obsolete in a very short time. Performance models should be flexible enough to allow characterizing new devices.
- A majority of the models discussed in this review have been designed for CUDA, the most mature development environment for GPGPU. However the vendor neutrality of OpenCL and its availability for non-GPU accelerators is increasing its adoption by HPC programmers. In the past, OpenCL tools produced less efficient codes than their CUDA counterparts, but this is no longer true with the most current versions of OpenCL SDKs.
- Society is becoming aware of the great monetary and environmental costs derived from the high energy consumption of computing systems. The challenge is not only to squeeze the maximum performance out of a system, but also to do it with the minimum power. As often these two requirements cannot be optimized simultaneously, good tradeoffs have to be found. The power models reviewed in this paper can help to solve this bi-objective optimization problem.
- Reviewed models focus on outsourcing compute-intensive tasks to accelerator devices. This usually means leaving the CPU idle while the accelerator is busily crunching numbers. It is possible, however, to make both work simultaneously, significantly increasing system efficiency. There are proposals dealing with this workload distribution. Some of them divide the data to be processed into several chunks, obtaining performance measurements and, based on them, adjusting the optimal number of chunks to assign to each type of core [59]. Other authors run several benchmarks in the different compute resources using different balancing configurations, and use machine learning techniques to train a model to be able to predict the optimal chunk distribution for new applications [60]. Energy can also be included into the equation, making power-aware load balancing across heterogeneous systems [61].

As a final remark, we would like to point out an important difficulty we have found when crafting this survey: the lack of sufficient detail in different parts of the reviewed papers. For example, some authors neither indicate exactly which program features are needed by their models, nor the way used to obtain them. The methodologies used to test the accuracy of the models often lack detail too: proper testing methods require wide and representative data sets combined with state-of-the-art accuracy estimation techniques (such as cross-validation or bootstrapping) [62], taking care of not using test data in any step of the model creation process. Often, details about the exact accuracy estimation technique used with the models are missing: if this estimation is not performed in an sound way [63], models tend to over fit, providing (unrealistic) high levels of accuracy for the data used for training, but poor estimates for new, unseen data. Finally, it would be beneficial for the community to have access to models, tools and benchmarks, in order to cross-check results, to validate the models against applications not used by the developers, to test the viability of model variations, etc.

Acknowledgments

This work has been supported by the Saiotek and Research Groups (IT-609-13) programs (Basque Government), TIN2010-14931 (Ministry of Science and Technology) and COMBIOMED Network in Computational Biomedicine

(Carlos III Health Institute). U. Lopez is supported by a doctoral grant from the Basque Government. J. Miguel and A. Mendiburu are members of the HiPEAC European Network of Excellence.

REFERENCES

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J.Krüger, A.E. Lefohn, and T.J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80-113, 2007.

[2] GPGPU Community, http://www.gpgpu.org, Nov. 2013.

[3] J. Nickolls and W. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, no. 2, pp. 56-69, Mar./Apr. 2010.

[4] C. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50-59, Mar. 2011.

[5] J. Diaz, C. Munoz-Caro, and A. Nino, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era," *IEEE Trans. Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1369-1386, Aug. 2012.

[6] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the Limits of GPU Acceleration," *Proc. Second USENIX Conf. Hot Topics in Parallelism (HotPar '10)*, pp. 13-13, 2010.

[7] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 73-82, 2008.

[8] V. Lee et al., "Debunking the 100x GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 451-460, 2010.

[9] Khronos Group, *The OpenCL Specification*, 2008.

[10] NVIDIA, CUDA Home Page, http://developer.nvidia.com/cuda, Nov. 2013.

[11] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufman, 2013.

[12] G.F. Diamos, A.R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems," *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques*, pp. 353-364, 2010.

[13] R. Jain, *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

[14] T.M. Mitchell, *Machine Learning*. first ed., McGraw-Hill, Inc., 1997.

[15] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 152-163, 2009.

[16] P. Ganestam and M. Doggett, "Auto-Tuning Interactive Ray Tracing Using an Analytical GPU Architecture Model," *Proc. Fifth ACM Ann. Workshop General Purpose Processing with Graphics Processing Units*, pp. 94-100, 2012.

[17] K. Kothapalli, R. Mukherjee, M. Rehman, S. Patidar, P. Narayanan, and K. Srinathan, "A Performance Prediction Model for the CUDA GPGPU Platform," *Proc. IEEE Int'l Conf. High Performance Computing (HiPC)*, pp. 463-472, 2009.

[18] C.Y. Gonzalez, "Modelo de estimación de rendimiento para arquitecturas paralelas heterogéneas," master's thesis, Univ. Politécnica de Valéncia, 2013.

[19] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling GPU-CPU Workloads and Systems," *Proc. Third Workshop General-Purpose Computation on Graphics Processing Units*, pp. 31-42, 2010.

[20] S. Che and K. Skadron, "BenchFriend: Correlating the Performance of GPU Benchmarks," *Int. J. High Performance Computing Applications*, DOI: 10.1177/1094342013507960, http://hpc.sagepub.com/content/early/2013/10/11/1094342013507960, 2013.

[21] K. Sato, K. Komatsu, H. Takizawa, and H. Kobayashi, "A History-Based Performance Prediction Model with Profile Data Classification for Automatic Task Allocation in Heterogeneous Computing Systems," *Proc. IEEE Ninth Int'l Symp. Parallel and Distributed Processing with Applications (ISPA)*, pp. 135-142, 2011.

[22] J. Meng, V.A. Morozov, K. Kumaran, V. Vishwanath, and T.D. Uram, "GROPHECY: GPU Performance Projection from CPU Code Skeletons," *Proc. ACM Int'l Conf. for High Performance Computing, Networking, Storage and Analysis(SC '11)*, pp. 14:1-14:11, 2011.

[23] C. Nugteren and H. Corporaal, "The Boat Hull Model: Enabling Performance Prediction for Parallel Computing Prior to Code Development," *Proc. ACM Ninth Conf. Computing Frontiers*, pp. 203-212, 2012.

[24] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Comm. ACM*, vol. 52, no. 4, pp. 65-76, 2009.

[25] J. Meng and K. Skadron, "Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs," *Proc. 23rd ACM Int'l Conf. Supercomputing*, pp. 256-265, 2009.

[26] J. Choi, A. Singh, and R. Vuduc, "Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 115-126, 2010.

[27] J. Rice and R. Boisvert, "*Solving Elliptic Problems Using Ellpack*," *Springer Series in Computational Math*, vol. 1, Springer, 1985.

[28] H. Jia, Y. Zhang, G. Long, J. Xu, S. Yan, and Y. Li, "GPURoofline: A Model for Guiding Performance Optimizations on GPUs," *Proc. 18th Int'l Conf. Parallel Processing (Euro-Par '12)*, vol. 7484, pp. 920-932, 2012.

[29] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison," *Proc. Many-Core Applications Research Community Symp. at RWTH Aachen Univ.*, pp. 38-44, 2012.

[30] J. Lai and A. Seznec, "Break Down GPU Execution Time With an Analytical Method," *Proc. 2012 Workshop Rapid Simulation and Performance Evaluation: Methods & Tools*, pp. 33-39.

[31] Y. Zhang and J. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," *Proc. IEEE 17th Int'l Symp. High Performance Computer Architecture (HPCA)*, pp. 382-393, 2011.

[32] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp, and W. Hwu, "An Adaptive Performance Modeling Tool for GPU Architectures," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 105-114, 2010.

[33] J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, 1987.

[34] S. Ramos and T. Hoefler, "Modeling Communication in Cache-Coherent SMP Systems: A Case-Study with Xeon Phi," *Proc. 22nd Int'l Symp. High-Performance Parallel and Distributed Computing*, pp. 97-108, 2013.

[35] S. Baghsorkhi, I. Gelado, M. Delahaye, and W. Hwu, "Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors," *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 23-34, 2012.

[36] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization," *Proc. IEEE 26th Int'l Parallel Distributed Processing Symp. (IPDPS)*, pp. 83-94, 2012.

[37] R. Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, http://books.google.es/books?id=pQV2nAEACAAJ, 2013.

[38] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 11-22, 2012.

[39] Y. Wang and N. Ranganathan, "An Instruction-Level Energy Estimation and Optimization Methodology for GPU," *Proc. IEEE 11th Int'l Conf. Computer and Information Technology (CIT)*, pp. 621-628, 2011.

[40] S. Hong and H. Kim, "An Integrated GPU Power And Performance Model," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 280-289, 2010.

[41] Y.S. Shao and D. Brooks, "Energy Characterization and Instruction-Level Energy Model of IntelOs Xeon Phi Processor," *Proc. ACM/IEEE Int'l Symp. on Low Power Electronics and Design (ISLPED)*, 2013.

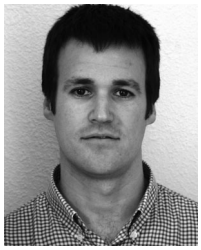
[42] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical Power Consumption Analysis and Modeling for GPU-Based Computing," *Proc. ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[43] X. Ma, M. Rincon, and Z. Deng, "Improving Energy Efficiency of GPU Based General-Purpose Scientific Computing through Automated Selection of Near Optimal Configurations," University of Houston, Technical Report UH-CS-11-08 2011.

[44] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical Power Modeling of GPU Kernels Using Performance Counters," *Proc. Green Computing Conf.*, pp. 115-122, 2010.

[45] S. Song, C. Su, B. Rountree, and K.W. Cameron, "A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures," *Proc. IEEE 27th Int'l Symp. Parallel and Distributed Processing (IPDPS)*, pp. 673-686, 2013.
[46] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N.S. Kim, T.M. Aamodt, and V.J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," *Proc. 40th Ann. Int'l Symp. Computer Architecture*, 2013.
[47] J. Lucas, S. Lal, M. Andersch, M. Alvarez Mesa, and B. Juurlink, "How a Single Chip Causes Massive Power Bills—Gpusimpow: A GPGPU Power Simulator," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, pp. 97-106, 2013.
[48] S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pp. 469-480, 2009.
[49] GPGPU-Sim Online Manual, http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual/, Apr. 2013.
[50] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS '09)*, pp. 163-174, 2009.
[51] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A Parallel Functional Simulator for GPGPU," *Proc. IEEE Int'l Symp. Modeling, Analysis & Simulation of Computer and Telecomm. Systems (MASCOTS)*, pp. 351-360, 2010.
[52] Barra GPU Simulator, https://code.google.com/p/barra-sim/, Apr. 2013..
[53] N. Goswami, R. Shankar, M. Joshi, and T. Li, "Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications," *Proc. IEEE Int'l Symp. Workload Characterization (IISWC)*, pp. 1-10, 2010.
[54] N. Brunie, S. Collange, and G. Diamos, "Simultaneous Branch and Warp Interweaving For Sustained GPU Performance," *SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 49-60, 2012.
[55] Gatech, Macsim Simulator for Heterogeneous Architecture, http://code.google.com/p/macsim/, Jan. 2012.
[56] V. Zakharenko, "FusionSim: Characterizing the Performance Benefits of Fused CPU/GPU Systems," PhD dissertation, Univ. Toronto, 2012.
[57] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing," *Proc. 21st Int'l Conf. Parallel Architectures And Compilation Techniques*, pp. 335-344, 2012.
[58] M. Arora, S. Nath, S. Mazumdar, S. Baden, and D. Tullsen, "Redefining the Role of the CPU in the Era of CPU-GPU Integration," *IEEE Micro*, vol. 32, no. 6, pp. 4-16, Nov./Dec. 2012.
[59] M.E. Belviranli, L.N. Bhuyan, and R. Gupta, "A Dynamic Self-Scheduling Scheme for Heterogeneous Multiprocessor Architectures," *ACM Trans. Architecture and Code Optimization*, vol. 9, no. 4, p. 57, 2013.
[60] D. Grewe and M. O' Boyle, "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL," *Proc. 20th Int'l Conf. Compiler Construction: Part of the Joint European Conf. Theory and Practice of Software*, pp. 286-305, 2011.
[61] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M. Sarrafzadeh, "Energy-Aware High Performance Computing with Graphic Processing Units," *Proc. ACM SOSP Workshop Power Aware Computing and Systems (HotPower)*, 2008.
[62] N. Japkowicz and M. Shah, *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge Univ. Press, 2011.
[63] J. Reunanen, "Overfitting in Making Comparisons between Variable Selection Methods," *J. Machine Learning Research*, vol. 3, pp. 1371-1382, Mar. 2003.

Unai Lopez-Novoa received the MSc degree in computer science from the University of Deusto in 2010. He is currently working toward the PhD degree at the Department of Computer Architecture and Technology of the University of the Basque Country UPV/EHU. His main research interests include parallel and distributed computing and specially, massively parallel computing using accelerators.



Alexander Mendiburu received the BS degree in computer science and the PhD degree from the University of the Basque Country UPV/EHU, Gipuzkoa, Spain, in 1995 and 2006, respectively. He has been a lecturer since 1999, and an associate professor since 2008 in the Department of Computer Architecture and Technology, UPV/EHU. His main research interests include evolutionary computation, probabilistic graphical models, and parallel computing. He has published 15 papers in ISI peer-reviewed journals and more than 20 papers in international and national conferences.

Jose Miguel-Alonso received the MS and PhD degrees in computer science from the University of the Basque Country (UPV/EHU), Gipuzkoa, Spain, in 1989 and 1996, respectively. He is currently a full professor at the Department of Computer Architecture and Technology of the UPV/EHU. Prior to this, he was a visiting assistant professor at Purdue University for a year. He teaches different courses, at graduate and undergraduate levels, related to computer networking and high-performance and distributed systems, and has supervised (and currently supervises) several PhD students working on these topics. He is a member of the IEEE Computer Society.