A Survey of Performance Modelling Techniques for Accelerator-based Computing

Unai Lopez, Alexander Mendiburu, and Jose Miguel-Alonso

Abstract—In the last years high performance computing has become heterogeneous. Almost every computing resource is composed of traditional out-of-order execution cores and accelerator device(s), such as graphical processing units, to which compute-intensive tasks can be offloaded. The advent of this type of systems has brought about a wide 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 graphical processing units as the most widely available class of co-processors. We begin describing current heterogeneous systems and the development frameworks and tools that can be used for developing them. Next, a diverse collection of performance models proposed in the literature is reviewed. These models are valuable tools to understand the performance of a given device when running a particular application, and are aimed to help programmers make the most of the huge computing power available in state-of-the-art and future heterogeneous computing 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 commonly attached to a Central Processing Unit (CPU) which controls the offloading of software fragments and manages the copying and retrieval of the data manipulated at the accelerator. Most accelerators show important architectural differences with respect to the CPU to which they are attached. Commonly, the number of computing cores, instruction sets or memory hierarchy is completely different, which make them suitable for certain computations that would be inefficiently processed in the CPU.

Around 2007, the High Performance Computing (HPC) community began using Graphics Processing Units (GPU) as accelerators for general purpose computations [79], coining the term GPGPU (General Purpose computing on GPUs) [2]. These are hardware devices specifically designed for the manipulation of computer images. The increasing needs of the image processing field resulted in an important boost in the horsepower of the GPU, starting a manufacturer race in which each new generation of devices offered significantly higher (theoretical) FLOPS [68] [97]. The scientific community soon became aware of the power hidden inside graphic cards, devising smart (and complex) tricks to disguise scientific computations as graphics manipulations. GPU manufacturers became aware of this trend and provided APIs (Application Programming Interfaces) and SDKs (Software Development Kits) to facilitate a more direct programming of their devices for non-graphics tasks.

In subsequent years, GPGPU spread and many commercial applications relied on the use of GPUs as accelerators. GPU manufacturers not only improved general-purpose software development tools, but also made their devices evolve, taking into account the needs of this new market. Discrete accelerators built around GPUs but without video connectors were produced, with exceptional acceptance by the HPC community. The impact of the evolution of this fast market can be seen following the Top500 list [4], a ranking of the 500 most powerful supercomputers in the world which is renewed twice a year. In the November 2012 update, three out of the top 10 computers were built around accelerators. Out of the complete list, 62 supercomputers use accelerators, 53 of which are GPUs. Another example of the wide adoption of GPUs as compute co-processors can be seen in the Amazon Web Services portal, where this company offers virtual infrastructures for HPC with built-in GPUs [9].

Effectively using the theoretical power of an accelerator is a very challenging task. It requires not only the use of new tools, but also a completely new set of programming paradigms (compared to those for CPUs). In most cases, adapting an application to use GPUs requires 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 power of CPUs and accelerators, not to mention a massively parallel system with thousands of combined CPU+GPU nodes. While several works claim that it is difficult to efficiently use (homogeneous) massively parallel computing systems [34], it is even more difficult with heterogeneous, accelerator-based supercomputers [59] [93].

The challenge, therefore, is to carefully design and

• *U. Lopez, A. Mendiburu and J. Miguel-Alonso are with the Department of Computer Architecture and Technology, University of the Basque Country UPV/EHU, Gipuzkoa 20018, Spain*
E-mail: unai.lopez@ehu.es

implement applications to make the best possible use of the available resources. A common pitfall is to carry out code implementation or tuning at random, by trial-and-error, without the feedback of performance tools that would provide guidance during the development process, avoiding (or helping to remove) performance bottlenecks [54]. 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/or tuning. The main objective of this survey is to analyze, organize and summarize the large body of literature in this field. To better understand these performance tools, we also provide the reader with a picture of the evolution and state-of-the-art in accelerator hardware and the development tools used with them. To the best of our knowledge, there is no previous research document providing a similar assessment of performance tools for accelerator-based application developers.

In Section 2, this document begins reviewing the rapidly evolving landscape of heterogeneous hardware in current HPC systems. Then, in Section 3 we describe the novel programming paradigms and development frameworks that accelerators in general and GPUs in particular have brought. We continue in Section 4 with a classification of the existing performance analysis and optimization models and tools based on the intended uses (execution time prediction, bottleneck highlighting, simulation or power consumption estimation). We also analyze the proposals found in the literature, discussing their constraints, strengths and weaknesses. Furthermore, this information is summarized in several tables. Finally in Section 5, we provide some conclusions and discuss future research lines in this field.

2 Heterogeneous architectures

Before the GPGPU era, compute nodes in HPC systems were mainly built with one or several CPUs, which held one or several processing cores each. In most cases these compute nodes were powerful server-class computers, capable of efficiently running almost any type of workload. Evolution towards multi-cores was a result of excessive power consumption. The current trend is to package more general-purpose cores in each new CPU generation.

Multi-core processors show some weaknesses when dealing with highly data-parallel applications that require many threads to be spawned – thousands, millions of them. Thread management in multi-cores is mainly performed by the operating system (although there is some hardware support) and implies expensive context switching. This fact, together with the evolution of GPUs, made the HPC community turn to heterogeneous, accelerator-based computing to deal with this class of massively parallel workloads. However, not all applications fit into this model and, therefore, CPUs are still the main computing platform for most applications.

In this section we review the state-of-the-art hardware devices used for heterogeneous computing. As we have stated before, this is a rapidly evolving field. We have summarized in Figure 1 a timeline of the main hardware devices developed by different manufacturers. A dashed line indicates the absence of a product family, while a shortened line indicates the discontinuation of a product. Brief descriptions of the devices mentioned in the figure are given along this section. GPUs are explored in more detail, as they are most popular accelerators of today.

2.1 Graphics Processing Units

GPUs are processors originally intended for the rendering of 2D and 3D images. Currently the main GPU manufacturers developing HPC-class products are NVIDIA and AMD (who bought ATI in 2006), see Figure 1 which also shows the evolution of their product families. Both AMD and NVIDIA provide desktop and server versions of their cards. Desktop cards are marketed as graphic accelerators that can also be used for GPGPU, running punctual workloads at full performance. Server cards are actually GPU-based accelerators, designed to run intensive workloads in an uninterrupted way.

There are other players in the GPU field. Not included in the figure is ARM [1], designer of a line of GPUs called *Mali*. This product family targets the embedded and mobile market, but not HPC: they are designed to accelerate image processing in low-power portable devices. Its design is radically different from those of AMD and NVIDIA, using a simpler memory hierarchy and a smaller number of processing units. In 2012, ARM announced the second Generation of the Mali family, the T-600 series [5], which integrates support to be used as a co-processor for GPGPU.

In essence, a GPU is an autonomous compute system composed of a set of cores and a memory hierarchy. Inside this memory hierarchy, GPUs have their own global memory space, accessible by all the processing cores, which can be exclusive (this is the common case if the GPU is a discrete accelerator plugged into a PCI-Express 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). Discrete NVIDIA and AMD server-class accelerators (Tesla [62] and FireStream systems [21] respectively) include up to 8 GB of global memory.

In terms of processing elements, GPUs are arranged as a set of multiprocessors, each of them holding, among other elements, several computing cores, a set of registers and some shared memory for inter-core communication. Common additional elements include double-precision floating-point units, functional units for transcendental operations, and a cache hierarchy. The components and their interconnection vary depending on the manufacturer and card model. For example, a Tesla C2050 accelerator of the Fermi family from NVIDIA has 14 multiprocessors, each one with 32 single

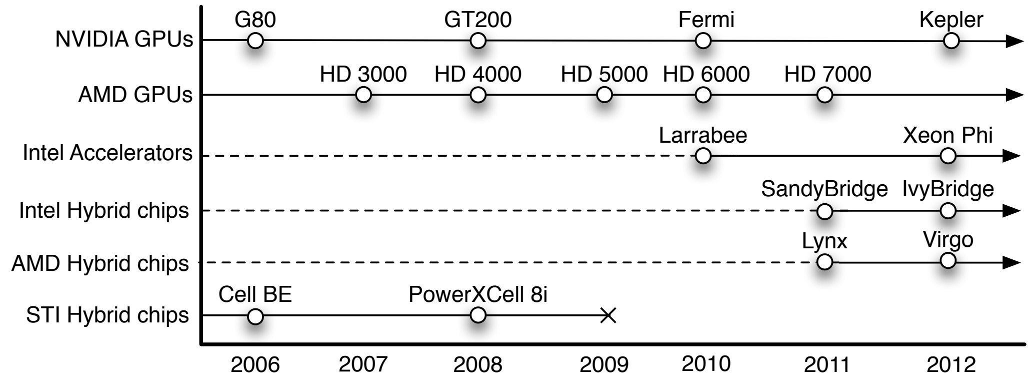

Fig. 1. Accelerator/hybrid devices timeline

precision processing cores, and 64 KB of configurable shared memory and L1 Cache [97]. In contrast, a Tesla K20 accelerator (with a newer GK 110 Kepler GPU) has 13 multiprocessors with 192 single-precision cores and 96 KB of configurable shared memory [76].

We refer the interested reader to [103], where authors make an in-depth comparison between the architecture of NVIDIA and AMD GPU families.

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) or when there is control flow divergence (which appears when not all threads run exactly the same instructions). GPUs hide these latencies by rapidly context-switching stalled threads (actually warps / wavefronts) with active ones.

2.2 Dedicated accelerators

In this section we review other, non-GPU accelerators used for HPC. In all cases they are devices designed for the offloading of compute intensive tasks.

In the last few years, Intel has released a line of accelerator products to compete with the GPGPU computing trend. A preliminary proposal, called Larrabee, was an accelerator composed of 32 x86 cores plugged to a host using PCIe. Larrabee was announced in 2008 [84], and some prototypes were shipped in 2010. In 2012, Intel launched Xeon Phi Coprocessor [50] (codename Knights Corner), as an improved version of Larrabee. A Xeon Phi SE10 coprocessor houses 61 x86 cores at 1.1 Ghz top clock frequency and 8 GB of dedicated built-in memory. The computing cores used in these devices are less powerful than those in state-of-the-art Xeon processors, but they are still more general purpose than GPU cores. The greatest strength of the Xeon Phi is its ease of programming. Intel has focused on developing a compatibility layer to run both massively parallel and legacy multithreaded applications.

Field Programmable Gate Arrays (FPGAs) are programmable integrated circuits designed to be configured by the developer or user. Some of the most popular FPGA manufacturers are Altera and Xilinx. The main focus of FPGA manufacturers and developers is not HPC; however, they can be used to offload computationally intensive tasks after the hardware has been appropriately programmed, using tools based on languages such as VHDL, with which the HPC community is not familiar [25]. This flexibility has been enhanced by the availability of tools that automate the development of VHDL code from higher-level, conventional programming languages (such as C). In fact, both Altera and Xilinx are offering development environments based on the existing GPGPU frameworks, and (directly or through partners) PCIe-pluggable cards to be used as accelerators [29].

2.3 Hybrid chips

In recent years, manufacturers have introduced into the market hybrid chips that combine different types of cores in the same die. A prototypical example is the family of AMD's APUs (Accelerated Processing Units [23]), that include several x86 cores and a GPU composed of several STMD cores. Similarly, Intel has its own family of processors with integrated graphics. The target market of these hybrid processors is low cost, low power computers, with the focus on mobility.

GPUs integrated into hybrid chips are not very different from those used in discrete accelerators. They may be less powerful (fewer cores), due to cost and power limitations, and therefore not attractive for HPC. However, they have an important advantage when compared to discrete accelerators: memory spaces for GPU and CPU are not separated; in fact, both share the same memory. This drastically reduces the overheads associated to CPU-GPU communication and synchronization, because data movement through PCIe (a requirement for discrete accelerators) is not necessary. There are studies available comparing the performance of the AMD Fusion platform against a discrete CPU+GPU system [30][87], where the main conclusion is that current hybrid devices are more compute efficient than a discrete GPU for applications involving massive data movements through the system bus.

The STI group, formed by Sony, Toshiba and IBM, presented in 2005 and released in 2006 a hybrid processor called the Cell Broadband Engine [51]. It integrated in the same die a general purpose PowerPC core and eight co-processor elements designed to accelerate multimedia and vector applications. The chip was shipped mainly in two ways: as Cell-based IBM server blades for general purpose systems, and in the Sony PlayStation 3 (PS3) gaming console. The production of the Cell was discontinued in 2009, as shown in Figure 1. On March 31st, 2013, the largest Cell-based supercomputer (Los Alamos' Roadrunner, manufactured by IBM) was decommissioned [80].

ARM designs processors that mainly target embedded systems, where the main challenge is low power consumption. Current ARM Cortex processors are widely used in smartphones and other mobile platforms, but with its increasing computing power there are some proposals to use this family of processors for general purpose systems, targeting "greener" cloud infrastructures and also supercomputers [81].

In a step towards low-power but efficient computing, ARM presented in 2011 a technology called big.LITTLE [42] which enabled to seamlessly balance computations among two different processors that coexist in the same chip. With this configuration, a *big* processor is performance oriented while another *LITTLE* processor is power efficient but less capable. A task manager dispatches tasks to in the most energy efficient manner.

As previously described, ARM also designs a family of GPUs (Mali) that can be integrated with general-purpose processors in the same chip. The low-power market is clearly heterogeneous.

3 DEVELOPMENT TOOLS

Parallel applications for accelerators are generally developed using software development environments provided by the device manufacturer. These environments can be manufacturer specific toolchains, or implementations of standard APIs. This section presents most common APIs and tools targeting current heterogeneous computing: CUDA and OpenCL. Additional development tools built on top of these are also described briefly.

3.1 CUDA

Compute Unified Device Architecture (CUDA [72]) is a framework released by NVIDIA for the development of general purpose applications on NVIDIA GPUs. It includes a hardware (compute and memory) model, and a programming environment.

Regarding the hardware model, a CUDA capable GPU is formed by several multiprocessors called Streaming Multiprocessors (SMs), which hold the processing cores, also called Streaming Processors (SPs). As described in Section 2.1, in addition to the compute cores, each SM also contains a register file, a shared memory space for inter-SP communication, and (depending on the particular model) additional elements such as double-precision floating point units, transcendental functional units, and a cache hierarchy. NVIDIA assigns a number to each configuration, which defines the device's *Compute Capability*.

Outside the SMs, each GPU holds a dedicated, global memory space of several gigabytes. The host system (typically, the CPU) can copy data to/from this memory.

Each CUDA program is composed of a host side code and a collection of device *kernels*. The former runs on the host device and manages data copies from/to host memory to/from device memory, and also the execution of the kernels, which are the data-parallel portion of the program and run on the GPU. Kernels are written using a language called CUDA-C, an extension of the standard ANSI-C99 with several synchronization and management functions. The host part defines the number and geometry of the threads which will run a kernel, and the organization of these threads into groups called *blocks*. At run time, each block will be assigned to a SM for its execution, where the kernel will be executed in a SIMD way by the SPs. Internally, each block is divided into sub-groups called *warps*, which are the minimum scheduling unit of the GPUs. Up to version 5 of CUDA, warps have a fixed size of 32 threads.

CUDA has been widely adopted as a framework for the acceleration of massively data-parallel tasks, due to its maturity and good performance. Its popularity has led to the creation of an ecosystem of developers and tools around this programming environment. We will review some of these tools in Section 3.3. It is also worth mentioning the large body of available literature about CUDA, both printed [55] and on-line [78].

3.2 OpenCL

Open Computing Language (OpenCL [46]) is a framework for the development of data/task parallel applications defined by the Khronos Group. It aims to be the vendor-neutral, standard API for the development of applications in accelerator devices. Similar to CUDA in many aspects, it also provides a hardware model and a programming environment.

Regarding the hardware model, an OpenCL *platform* is a collection of compute *devices*. Each device comprises one or more *compute units* which, in turn, contain *processing elements*. All the compute devices in a platform are connected to a host device that controls the execution of applications. The processing elements run work-items, the equivalent of CUDA threads; each work-item keeps its own private memory, and can access a local memory in order to communicate and synchronize with other work-items in the same compute unit. Finally, there is a global memory per device accessible to all work-items in that device.

This abstraction of the hardware enables the OpenCL framework to launch any parallel application over any device, provided that the minimum hardware requirements are fulfilled. Each manufacturer that releases an

CUDA	OpenCL	Description
Thread	Work Item	One of a collection of parallel executions of a kernel
Warp	Wavefront (AMD GPUs only)	Minimum unit of thread level scheduling
Block	Work-group	Group of threads
Streaming Processor	Processing Element	Single processing core
Streaming Multiprocessor	Compute Unit	Multi-processor inside a device
GPU	Compute Device	CPU, GPU or other accelerator
Registers	Private Memory	Memory space for a single thread
Shared Memory	Local Memory	Memory space for a group of threads

TABLE 1
Comparison of CUDA and OpenCL terminology

OpenCL capable device makes a mapping of its hardware to the OpenCL model and releases its own OpenCL SDK. For example, a NVIDIA GPU can be mapped to a compute device, each SM to a compute unit, and each SP to a processing element. In the case of Intel CPUs, each (multi-core) CPU is a compute device, and each core is considered to be a compute unit with a single processing element.

An OpenCL application consists of some host-side and some device-side code. The host-side code manages the execution of the device-side code, and also the data movements between devices. OpenCL device applications are called *kernels* (as in CUDA). Kernels are written in an extension of ANSI-C99, called OpenCL-C. Within a compute device, a number of developer-defined threads runs a kernel. Threads are aggregated into *work-groups*. Work-groups are assigned to compute units for their execution. The scheduling of the threads inside each compute unit is manufacturer-dependent. As an example, in AMD GPUs (for which AMD provides full OpenCL support) work-groups are divided into sub-groups called *wavefronts* (equivalent to NVIDIA's warps) with a typical size of 64 work-items.

It should be apparent that OpenCL and CUDA models and programming paradigms are very similar. There are differences in terminology – see Table 1, but also in other, more important aspects. While OpenCL has been designed to be generic, for any accelerator device, CUDA is tightly coupled with NVIDIAs GPUs. Also, as CUDA was developed first, it is a more mature environment. The overall result is that developing and tuning applications for NVIDIA GPUs is easier with CUDA, but if portability is a must and/or the program is being developed for another accelerator, currently OpenCL is the best choice.

It is also worth mentioning that CUDA applications are generally compiled off-line using the *nvcc* compiler. In contrast, OpenCL kernels are commonly compiled on-line, at run time when the application is launched. The OpenCL SDK for a given device will include the appropriate dynamic compiler for kernels. It is even possible to have several SDKs for the same device, which can result in different binaries for the same kernel. For example, both Intel and AMD offer OpenCL SDKs for multi-core CPUs, and they include different compilers.

The popularity of OpenCL comes from its wide support from the hardware manufacturers. Almost all the devices presented in Section 2 (GPUs, multi-core CPUs, hybrid chips, FPGAs) support the execution of OpenCL applications. OpenCL developers, thus, can create codes that are portable, being able to run in different accelerators. However, code portability does not translate into performance portability (an OpenCL code that runs efficiently in a GPU might behave poorly in a CPU). Therefore, extensive device-dependent tuning may be required.

For those readers interested in resources for OpenCL programming, we refer them to [11] [39] [77].

3.3 Higher level tools

Both CUDA and OpenCL are frameworks that provide a low level of abstraction over the hardware. The programmer has a high degree of control of what the application does, so that he is finally responsible for fine-tuning the code to effectively exploit the hardware. This is a difficult task, which becomes harder when we take into account the growing number of possible accelerators and families within them.

In order to simplify programming, debugging and tuning accelerator-based applications, an ecosystem of tools around CUDA and OpenCL has appeared, which try to hide hardware complexities and to offer programmers a higher-level view of accelerators. The characteristics of these tools are very different, because they were designed for diverse, sometimes very specific, purposes. Some of the tools included in this brief review are listed in the CUDA Tools and Ecosystem web site [73], a section inside the CUDA Developer zone where NVIDIA collects GPGPU-related applications, development tools and libraries.

A simple way to take advantage from accelerator devices is by making use of GPU-accelerated implementations of well-known function libraries. An application calling BLAS or FFT functions (these are just some examples) can take advantage of multi-core CPUs via linking to Intel's Math Kernel Library [49] or AMD's Core Math Library (ACML) [10], to mention just some options. Alternatively, GPU-ready versions are also available from the main GPU vendors: CUBLAS [71] and CUFFT [74] from NVIDIA, and APPML (Accelerated Parallel Processing Math Libraries) [12] from AMD.

Another option for the development of parallel versions of existing codes is relying on the abilities of parallelizing compilers. Some popular compilers, such as Intel Compilers, have built-in features to detect simple loops in an existing application and are able to automatically generate multi-threaded implementations. Similarly, the Par4All [14] open-source project is a compiler workbench that generates CUDA, OpenMP or OpenCL versions of programs from their source code in C or Fortran, detecting (and transforming) parallelizable loops.

An alternative that provides a trade-off between the simplicity of the previous tools (which completely hide the complexities of parallel hardware and programming) is to use an approach that has proven successful when programming for multi-cores: the utilization of programming directives for the declaration of parallel sections in the the code. OpenMP is a successful case [31]. Inspired by it, several companies have joined efforts to create OpenACC [3], a standard for the development of accelerator based applications using OpenMP-like compiler directives. OpenACC is endorsed by NVIDIA, CAPS, PGI and CRAY among others, who are interested in the definition of a common framework to simplify programming accelerator devices. Currently, PGI implements OpenACC support for NVIDIA GPUs in its Accelerator Compilers [98], while CAPS provides support NVIDIA and AMD GPUs, and also for Intel's Xeon Phi.

4 Performance Models

When developing a program, it is helpful to have access to a collection of support tools that complement the compilers, such as editors, profilers, debuggers, etc., even more so if they form part of an integrated development environment. When dealing with parallel programming, the need for these tools is even larger, because parallel code is generally more complex than sequential code, and the execution is harder to follow: multiple threads/processes running simultaneously, synchronization issues, memory sharing, etc. A working code is just the starting point for an optimized code. We have hinted before that developing optimized code for accelerator-based systems is not easy. However, the research community has made a step forward by designing powerful models, able to estimate execution times for different programs and platforms, to identify bottlenecks, to suggest the most appropriate optimization flags, or to provide detailed instruction-level information. These models and tools, which we review in this section, can help as a guide during the optimization process. We pay special attention to developing for GPUs, because this is the most popular class of accelerator devices and, therefore, it has a larger developer community.

The common scheme of a performance models is depicted in Figure 2. 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 is has been designed for, the input and output datasets can be notably different. We will discuss these aspects in the following sections.

4.1 Input data

We consider two different types of data: (1) related to the target hardware (the device or devices) and (2) related to the program that will run in that hardware.

4.1.1 Device characteristics

Information about the characteristics of a particular device can be collected in two main ways. The manufacturer usually provides product data sheets and detailed hardware manuals [11], [77]. Often, the development environment include mechanisms (an API) to query the device.

A complementary source of information is that provided by 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, with the advent of GPGPU, similar toolsets have been developed for NVIDIA [99] and AMD GPUs [89].

4.1.2 Application characteristics

There are different approaches to obtain information from a program. One of the most intuitive ways is the analysis of its code in order to identify memory access patterns, arithmetic operations, branches, etc. This can be done directly on the source code or using some transformations. For example, in [43] authors use Clang [57] (the front-end of the LLVM [58] compiler infrastructure) to generate an Abstract Syntax Tree. Next, they go throughout the tree counting the features they need to feed their model.

It is also common to parse an intermediate representation (IR) of the program, an assembly-like code representation produced by the compiler before the generation of the binary files. Intermediate formats are simpler, easier to parse, and usually provide additional information, such as the number of registers per thread, size of buffers allocated in shared and constants memory, and so on. A drawback of this approach relies on the potential optimizations that the compiler might have performed, and that may not exactly represent the original source code. In the case of CUDA applications for GPUs, the intermediate representation is in PTX, a virtual instruction set developed by NVIDIA. It can be obtained using the NVIDIA compiler *nvcc* (part of the CUDA SDK [72]) with the *-ptx* flag.

Regarding OpenCL, the IR will vary depending on the device that the application targets, and on the used SDK. As the same program can be executed on different platforms, each compiler will generate a different IR for each device family. Given these facts, it is not easy to develop a tool that relies on obtaining the IR of an OpenCL application, unless it is specific for a given device family

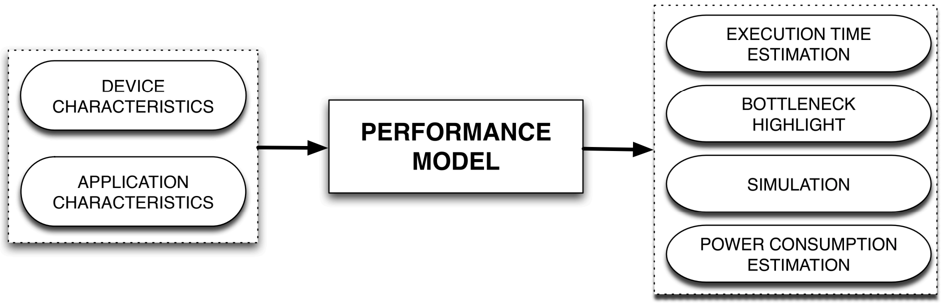
DEVICE CHARACTERISTICS
APPLICATION CHARACTERISTICS
PERFORMANCE MODEL
EXECUTION TIME ESTIMATION
BOTTLENECK HIGHLIGHT
SIMULATION
POWER CONSUMPTION ESTIMATION

Fig. 2. Input/output scheme of a performance model

– thus losing the portability expected from the OpenCL environment. To tackle this issue, in 2012 the Khronos Group released the 1.0 version of the Standard Portable Intermediate Representation (SPIR) [45], a proposal of a common IR for every OpenCL framework. This IR is based on that defined for the LLVM compiler toolchain.

A lower level approach to source code parsing consists of disassembling the final binary files. These are the files loaded onto and executed by the devices, so they provide maximum information about what the hardware will do. The drawback of this approach is that it is device or architecture specific, and that the obtained assembly code might not be trivial to parse. Examples of this class of binary code analyzing tools are Decuda [92] and cuobjdump [75]; both are disassemblers for the CUDA binary format.

The Ocelot [33] modular compilation framework is a popular tool used to analyze CUDA kernels. Ocelot can transform a kernel from its PTX representation to LLVM's intermediate representation, and from this to other instruction sets (for example, x86 and ARM). Ocelot is also capable of emulating the execution of PTX codes and includes several source analysis tools. All together, it can be used to extract a variety of metrics from a PTX code (static information) and its execution (dynamic information).

Finally, it is worth mentioning that profilers can provide extremely accurate information about how an application runs on a given hardware. They can collect and summarize a variety of performance counters, useful to understand and model the way the application behaves and the use it makes of the resources available.

4.2 Modelling methods

A performance model is designed with the aim of predicting the behavior of a system. Regarding the literature, there are three approaches which are most widely used: analytical modelling, machine learning and simulation. An analytical model is an abstraction of a system in the form of a set of equations. These equations try to represent and comprise all (or most) of the characteristics of the system. Machine Learning (ML) [67] 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 will be used with new (unseen) data to classify it. Finally, simulators are tools designed to imitate the operation of a system. They are able to mimic each step providing information about system dynamics and behavior. The level of accuracy of the simulator's output information 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.

4.2.1 *Model training*

Independently of the nature of a model, most have a set of parameters that must be tuned according to the particular scenario to which they will be applied. Sometimes, these parameters are assigned values coming from experts or previous experimentation but, in general, models are trained with data obtained from scenarios similar to the target one. For example, a model designed to predict execution times of a particular application on a particular hardware with a particular configuration (number of threads, workgroup size, input data, etc.) may take as training data set the results obtained from running the same application in the same or similar hardware for a variety of different configurations. If the training dataset is large and representative enough, one could expect promising estimations of the execution time for the target application / hardware combination. However, this model would probably provide poor results for other applications (unless it is very close to the one used for training) or for other, different target hardware.

If we focus on modelling for a fixed hardware, the challenge is to design a general model, able to estimate the execution time of any application. The training process needs to be performed using a wide dataset, that is, extracting and gathering information from a representative set of applications (different computation/memory rates, memory access patterns, number of branches, etc.). In the literature we can find references to benchmark application suites that can be used for this purpose. Here are some of them:

- The Rodinia application suite [24][26], developed by the University of Virginia is one of the most popular benchmark suites. It contains diverse applications, each of them representing a compute behavior. There is a multi-core (OpenMP) and a GPU-accelerated (CUDA or OpenCL) implementation for each application.

- Parboil [88] is a suite of applications developed by the University of Illinois. It provides a diverse set of multi-core (OpenMP) and GPU (CUDA/OpenCL) implementations. Also, for each application, the suite includes a simple readable implementation, and also an optimized implementation.
- The Scalable HeterOgeneous Computing (SHOC) suite [32] is a collection of benchmarks that aims to be diverse in the nature of its applications. Authors provide CUDA, OpenCL and OpenMP implementations of each benchmark. SHOC applications are designed to be scalable if they are tested in a large enough cluster or set of devices. Applications are divided into three groups. Those in the first group are aimed to test low level hardware features (e.g. memory bandwidth or peak FLOPS). The second group is composed of implementations of common algorithms, such as FFT or SGEMM. The last group is a collection of compute intensive applications that aim to test the stability of large systems.
- NAS Parallel Benchmarks (NPB) [19] is a classic suite used to test the performance of multi-core computers and supercomputers. In 2011, Seoul National University adapted several of the NPB benchmarks to OpenCL and made them publicly available [85].
- Finally, test and demonstration programs included in vendor-provided SDKs (such as NVIDIA's CUDA or AMD's APP) have been also widely used for testing purposes. Some of them are FFT, N-Body, GEMM, etc.

4.3 Output data

Performance models can be classified according to different criteria. In this work we focus on the type of output information the model is able to provide, identifying four model classes (depicted in Figure 3). There are models designed to (1) predict execution times, (2) identify performance bottlenecks and propose performance optimizations, (3) provide detailed instruction-level information based on simulation, and (4) provide estimations of power consumption. In the forthcoming sections we make a literature review of the proposals inside these four classes.

4.4 Execution time estimation

In Table 2, we have compiled and summarized information about modelling tools for accelerator-based computing whose main purpose is to provide estimations of execution times. Each row corresponds to a model proposal, and includes the method used to build it (analytical or machine learning), the target hardware, the information required to feed the model, and its features and limitations.

Hong et al. present in [47] an analytical model for NVIDIA GPUs. This model is able to estimate the number of execution cycles needed by a given CUDA kernel (that can be easily transformed into execution time). As explained in Section 2.1, GPUs hide latencies by handling thousands of in-flight active threads, rapidly switching from warps stalled due to memory transactions with ready-to-run warps. Hong et al. model this behavior using two different metrics, called MWP (Memory Warp Parallelism) and CWP (Computation Warp Parallelism). This model represents a good trade-off between simplicity and hardware representation. Even though it does not take into account many device features, such as details of the memory hierarchies, it produces quite accurate results. Experiments show a geometric mean error of 13.3% between estimated and actual number of cycles for a collection of seven benchmark applications on four different GPUs. This model has been used in Ganestam et al. [37] as part of an auto-tuning tool for the optimized execution of a ray-tracing algorithm.

Kerr et al. [53] present an approach based on principal component analysis, cluster analysis and linear regressions. To train (calibrate) the models, authors use a dataset of performance features obtained from different CUDA kernels by means of Ocelot, and run these applications on the target hardware (NVIDIA GPUs or multicore CPUs). Once the model is trained, it can be used to estimate execution times for new applications or devices. They present four different models: application modelling, GPU modelling, CPU modelling, and CPU-GPU modelling. The paper states that GPU modelling (estimation of application execution times on an unknown GPU) provide fairly accurate results, with a maximum deviation of 16% in the worst case. Unfortunately the remaining models are not that accurate, producing poorer estimates and high variability.

Kothapalli et al. [56] present an analytical model to estimate the execution time of a kernel given its pseudo-code. They aim to create a model able to estimate the performance of a pseudo-code before it is actually coded in CUDA. The paper does not state clearly the syntax to be used for the pseudo-codes, but apparently it is very similar to CUDA. The model focuses on computing the most time-consuming task in a kernel, because it will determine the execution time of the whole kernel. Execution time is computed after differentiating the number of cycles required for computation from those for memory operations. Warp overlapping is taken into account; authors state that, in the best case, the total execution time will be determined by the maximum value between compute and memory cycles (what they call MAX model) and in the worst case both compute and memory cycles will overlap (the SUM model). In most of the tested cases, SUM and/or MAX are quite close to the real results but, unfortunately, authors do

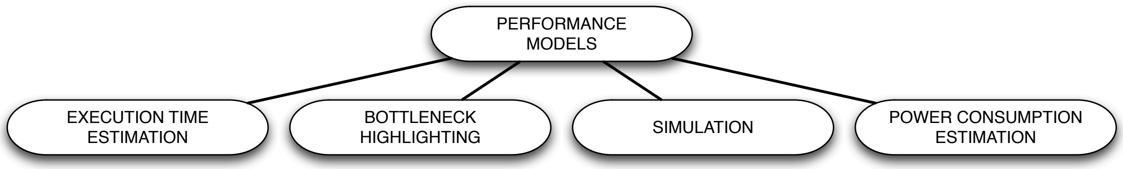

Fig. 3. Performance model taxonomy

not provide hints about how to choose the right one for a given application. This model has been extended to include more complex hardware features (memory copy overheads, branch-divergence latencies, etc.) in the MSc Thesis presented by Garcia et al. [38]. This extended model improves previous results, choosing in most cases the SUM model. As in the original work, no reason is given for this decision.

Similar to the previous work, Nugteren et al. [70] present a prototyping tool able to predict the performance of an unimplemented application when running on different hardware devices such as CPUs and GPUs. In contrast to the previous model, this tool needs to identify each computation block based on a parameterizable classification (e.g x-projection, 2D-convolution, etc.) that they proposed in a previous work [69]. For each block, their tool generates a performance graph, called Boat Hull, which shows the estimated performance in each available hardware device, together with some potential code optimizations. This work is based on the Roofline model presented by Williams et al. [96], which visually represents the limitations of a device (the roofs) and the limitations of an application (the ceilings).

In addition to the previous works, which can be considered general purpose models, that is, suitable to any application, there are also other proposals which focus on specific programs or program classes. For example, in [65] authors propose a framework that automatically selects the best parameters for an ISL (Iterative Stencil Loops) application, given some parameters of the domain and some features of the target GPU. ISL codes distribute computations into overlapping regions (tiles); neighboring regions interact via halo regions. This technique is applied in many domains, including molecular dynamics simulation and image processing. Choi et Al. present in [27] an auto-tuned implementation of the SpMV (Sparse Matrix-Vector) multiplication. The difficulty in computing with sparse matrices is related to using compression algorithms (such as BCSR or ELLPACK [82]) 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 the execution time.

4.5 Bottleneck highlighting and performance optimization

Once a parallel application has been designed and implemented, it is necessary to carefully analyze its execution looking for potential bottlenecks and, if possible, finding alternatives to eliminate or minimize them. Unfortunately, this is not an easy task, especially when programming accelerators. For example, typical optimizations that provide excellent results in CPUs, such as loop unrolling or tiling, when applied in GPUs may cause a high use of scarce resources, such as on-chip registers, and may cause more harm than benefits.

Parallel application SDKs commonly include a profiler (e.g. NVIDIA Visual Profiler for the CUDA SDK or CodeXL [13] for the AMD APP SDK), which should be the first tool to use when trying to optimize a kernel. Profilers for GPUs analyze the execution of the code and detect bottlenecks. They can even make recommendations to the programer about code changes that might reduce those bottlenecks. However, profilers 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. Our review of the literature has been summarized in Table 3. We now briefly analyze some of the main features of these models.

A first contribution that pointed out the difficulty of optimizing a CUDA kernel was presented by Ryoo et al. [83]. They review the most common performance optimizations (block size, tiling, prefetch, loop unrolling, shared memory and cache usage, etc.) and their impact on run times, and present a methodology to narrow the search space of promising optimization combinations. They first propose two metrics, called efficiency and utilization. These metrics are calculated using kernel features obtained from PTX instructions, information about resource usage, and some manual annotations (such as the average iteration counts of the major loops). Then, a Pareto-optimal front can be created, based on both efficiency and utilization metrics, and therefore only those configurations belonging to the Pareto set need to be tested (run). According to the experiments, a reduction in the search space between 74% and 98% is observed. In addition, the optimal configuration always belongs to the Pareto set for the applications tested. However, the model suffers some limitations: in order to simplify the calculations, group synchronization instructions are considered together with long latency memory

Model	Method	Target platforms	Tested devices	Input preprocessing	Limitations	Additional features
[47]	• Analytical	• NVIDIA GPUs	• NVIDIA 8800 GT • NVIDIA FX5600 • NVIDIA GTX280	• μBenchmarks execution • PTX analysis	• Cache misses and branch instructions not modelled • Memory model too simple	• Seems easily extendable to non NVIDIA GPUs
[53]	• Machine Learning	• GPUs • Multi-core CPUs	• 4 NVIDIA GPUs • 3 Multicore CPUs	• PTX analysis using Ocelot	• No insights about prediction process	• Model validated using a large benchmark
[56]	• Analytical	• NVIDIA GPUs	• NVIDIA GTX280	• Kernel pseudo-code analysis	• No hints about preferred model • Overheads of intra-thread synchronization and atomic operations not modelled	• Seems easily extendable to non NVIDIA GPUs
[70]	• Analytical	• GPUs • Multi-core CPUs	• NVIDIA GTX470 • NVIDIA GTS250 • Intel Q8300 • Intel i7-930	• μBenchmarks execution • Serial code analysis and classification following taxonomy in [69]	• Loss of accuracy due to high level abstraction in modelling	• Seems easily extendable to new architectures • Graphical output • Optimization proposals
[65]	• Analytical	• NVIDIA GPUs	• NVIDIA GTX280	• μBenchmarks execution • ISL input analysis	• Only for ISL applications	• An automated framework template for trapezoid optimization
[27]	• Analytical	• NVIDIA GPUs	• NVIDIA Tesla C870 • NVIDIA Tesla C1060	• Input matrices analysis	• Only for SpMV multiply operation • Thread block scheduling latencies not taken into account	• Detailed explanation of the BELLPACK implementation

TABLE 2
Summary of models for the estimation of execution time

operations; cache conflicts are not taken into account, and not all the configurations belonging to the Pareto set are close to minimizing the execution time of the kernel.

Baghsorkhi et al. [17] present a more elaborate model, that predicts the number of cycles spent by threads in different stages: compute, bank conflict, divergence, etc. The main input information is a program dependence graph (PDG [36]) obtained parsing the source code of a CUDA kernel. The PDG represents the workflow of the thread. This information is combined with the runtime parameters of the program and characteristics of the target device. Authors validate their model against several implementations of four CUDA Kernels, a naive one and others that include a diversity of optimizations using, for example, tiling and shared memory. Experiments show that the model is quite accurate highlighting the most time consuming stage in each execution.

In the same line, Baghsorkhi et al. [18] present a memory hierarchy model for GPUs through a framework that predicts the efficiency of the memory subsystem. In particular, they predict the latency of the main memory accesses of a GPU and hits and misses in the L1 and L2 caches. The memory model is based on the hardware memory hierarchy of the Tesla card, and validations with this particular hardware provide good results. In particular, they compare the results of their model against the hardware counters obtained through profiling, obtaining average absolute errors of 3.4 % for L1 read hit ratios, 1.9 % for L2 read hit ratios and 0.8 % for L2 write hit ratios.

Zhang et al. [102] approach performance prediction differently, analyzing first the GPU parts that may lead to bottlenecks (the instruction pipeline, the global memory and the shared memory). This information is used to highlight bottlenecks in a CUDA Kernel. A characterization of the hardware is obtained by means of a collection of micro-benchmarks, and the Barra simulator (see Section 4.6) is used to get the instruction count of the application. The model uses these two sets of input data and outputs information about the GPU part that may cause a performance bottleneck. It also estimates the performance benefit of removing it. Experiments show a relative error in the predictions within 5% - 15%.

Sim et al. present GPUPerf [86], a framework for the prediction of potential benefits of different optimization decisions over a CUDA Kernel: use of shared memory, tiling, prefetching, etc. The target application requires being processed by the NVIDIA Profiler, by a framework based on Ocelot, and by a static instruction analyzer based on cuobjdump. This tool relies on the use of an analytical model which is an extension of MWP-CWP (introduced in Section 4.4) that takes into account much more architectural details of GPUs, such as the effect of caches or the behavior of SFUs (the functional units in charge of performing trascendental operations). Authors claim that the potential performance benefits suggested by the model are attainable, but no numeric values are provided.

4.6 Simulation

A simulator is a system representation able to mimic, step-by-step, the behavior of the target (real) system. In the field of heterogeneous computing, these tools are useful for different purposes: to obtain detailed information about the behavior of a device, to analyze the performance of a particular code, to test changes in the structure or organization of a device, to propose completely new device schemes, etc.

In the next two sections we summarize our literature review of simulators for heterogeneous computing. We first introduce those intended for the simulation of discrete GPUs, followed by those designed for heterogeneous devices. As we did in the previous sections, our literature review has been condensed in a table (see Table 4).

4.6.1 GPU Simulators

GPGPU-Sim [7][20] is a functional simulator of CUDA-capable NVIDIA GPUs. It can simulate CUDA and OpenCL applications in two ways: a fast functional way, which only executes the application and generates its output, and a slower way, which collects performance statistics. The first mode is reported to be 5 to 10 times faster than the performance counter gathering mode, and should be used only to verify that an application runs properly on the GPGPU-Sim environment. The simulator accepts a wide set of configuration parameters in order to better model the target hardware, from multiprocessor features to the topology of the interconnection network. Regarding validation, authors state that version 3.1.0 obtains an accuracy of 98.3% for the instructions per cycle (IPC) metric when simulating a GT-200 card. With a Fermi card the accuracy is similar: 97.3%. It is worth mentioning that, as an add-on, GPGPU-Sim includes a visual interface called AerialVision [15].

Barra [28][6] is another functional simulator for NVIDIA GPUs, somewhat less flexible because it only works with CUDA applications. Barra generates an output file with performance counters for each instruction in the simulated kernels, such as the addresses of the memory accesses or the number of used SIMD channels. Authors state that in most of their tests, the performance information retrieved using Barra exactly matches the performance counters provided by the CUDA Profiler. In particular, they report relative difference values from 0% to 22.99%, except for two cases where the difference raises to 45.59% and 81.58%. The strongest point of Barra is that it is a fast, multi-core parallel tool, running on average four times faster than the source level emulation in debug mode of the CUDA SDK.

4.6.2 Simulators for hybrid CPU+GPU chips

Multi2Sim [8] is an open-source, modular and configurable instruction set architecture-level simulator of x86 CPUs and AMD GPUs. It supports the simulation of sequential and parallel applications in the CPU side [91], and OpenCL applications in the GPU side [90]. Multi2Sim combines a functional simulator that interprets the compiled binaries and reproduces their behavior at ISA level, and an architectural simulator that generates traces from the functional simulation. The output of the architectural simulation is represented using a visual tool, which provides a cycle-based interactive navigation, i.e. a way to explore, in a fine grained way, the performance details of the application; this is the main mechanism for finding bottlenecks. Another tool integrated in Multi2Sim allows the researcher to launch and control large sets of simulations (for example, to carry out a parameter sweep) over a cluster of computers. Regarding the performance of the GPU-side simulation, authors report average errors between 7% and 30% when estimating the execution cycles of OpenCL kernels, and an average slowdown of 8700x and 44000x for the functional and the architectural simulation modules respectively.

MacSim [40] is a trace-driven and cycle-level simulator for heterogeneous architectures, able to simulate state-of-the art hardware such as Intel's Sandy Bridge processors (for both modes, CPU-only and fused CPU+GPU) and also GPU devices such as NVIDIA's Fermi cards. In the case of CPU devices, it handles x86 binary files, while for GPUs it parses CUDA source applications. Although there is no current heterogeneous chip combining x86 CPU and CUDA GPU cores, MacSim uses the discrete GPU simulation engine (CUDA based) for their heterogeneous system simulation. As other simulators, it provides a simple framework to collect device global or per-core performance statistics during the simulation. Authors do not provide metrics to assess the accuracy of this tool.

FusionSim [101] is a simulation environment for both discrete and heterogeneous systems for x86 CPU and

Model	Method	Target platforms	Tested devices	Input preprocessing	Limitations	Additional features
[83]	• Analytical	• NVIDIA GPUs	• NVIDIA 8800 GTX	• PTX and CUBIN analysis	• Global memory assumed not to be a bottleneck • Synchronization and long latency memory operations considered to have similar latencies	• N/A
[17]	• Analytical	• NVIDIA GPUs	• NVIDIA 8800 GT	• μBenchmarks execution • Kernel source analysis	• Compiler optimizations not fully taken into account	• Detailed breakdown of the execution time into compute or memory transaction stages
[18]	• Analytical	• NVIDIA GPUs	• NVIDIA Tesla C2050	• μBenchmarks execution • Kernel instrumentation	• Focused only on the memory hierarchy	• Detailed breakdown of memory operations
[102]	• Analytical	• NVIDIA GPUs	• NVIDIA GTX280	• μBenchmarks execution • Kernel analysis using Barra simulator	• Enough warps assumed to hide latencies • Overheads of branch and synchronization instructions and effects of caches not modelled	• Detailed hardware analysis through μBenchmarks
[86]	• Analytical	• NVIDIA GPUs	• NVIDIA Tesla C2050	• μBenchmarks execution • Kernel analysis with NVIDIA profiler, Ocelot & a static analysis tool based on cuobjdump	• Seems not easily portable to non NVIDIA GPUs	• Estimation of potential benefits from applying different optimizations

TABLE 3
Summary of models for bottleneck highlighting

CUDA GPU, aimed to provide detailed timing information. GPU-side simulation is based on the GPGPU-Sim simulator, and CPU side simulation is based on PTLSim [100]. Currently, FusionSim only supports single-core CPUs. This tool can be used to explore the potential benefits of using fused systems, instead of discrete GPUs connected with CPUs using PCIe. The combined chip that FusionSim simulates is a non-existent x86 CPU + CUDA GPU. Authors of FusionSim use an analytical model that estimates the expected performance benefits of such a chip and use it to verify the correctness of the simulations. Their experimental validation focuses on demonstrating that fusion-like architectures are the best choice for applications operating on small data inputs, and for those that, when running in a discrete accelerator set-up, would require massive data movements through PCIe. In other scenarios, a discrete GPU would be a better choice.

MV5 [66] is a reconfigurable and modular simulator that targets modelling heterogeneous CPU+GPU chips. It presents an abstraction from contemporary programming interfaces and threading mechanisms, such as CUDA, and offers the ability to simulate chips with vastly different, hybrid core configurations under the same address space with the same ISA. Its target is to explore the design and performance of unreleased architectures instead of providing insights about the behavior of current hardware, as other simulators in this section do. MV5 does not offer compatibility with current GPGPU programming paradigms and, therefore, target applications must be implemented using a generic programming model specific for MV5.

4.7 Power consumption estimation

A trending topic in the field of HPC is improving the power efficiency of computing systems. To that extent, significant effort is being devoted to modelling the power characteristics of systems, taking into consideration the applications that run on them. Application power modelling aims to estimate the power required

Model	Method	Target platforms	Tested devices	Target applications	Limitations	Additional features
[20]	• Cycle level simulation	• NVIDIA GPUs	• NVIDIA 8600 GTS	• CUDA • GPU OpenCL	• Physical GPU required to compile OpenCL applications	• Power modelling features • Visual analysis tool
[28]	• Cycle level simulation	• NVIDIA GPUs	• NVIDIA 9800 GX2	• CUDA	• N/A	• Multi-core implementation of the simulator
[90]	• Cycle level simulation	• AMD GPUs • Multi-core CPUs	• AMD Radeon HD 5870	• CPU Sequential • CPU Parallel • GPU OpenCL	• Fused architectures not simulated	• Visual analysis tool • Module for massive simulation
[40]	• Cycle level simulation	• NVIDIA GPUs • Multi-core CPUs • Hybrid Chips	• NVIDIA G80 • NVIDIA GTX280 • NVIDIA GTX465 • Intel Sandy Bridge CPUs	• CPU Sequential • CPU Parallel • CUDA	• GPU cores in hybrid systems modelled as NVIDIA GPU multiprocessors	• Power modelling features for CPU simulation • Both discrete and hybrid CPU+GPU systems simulation capabilities
[101]	• Cycle level simulation	• NVIDIA GPUs • Multi-core CPUs • Hybrid chips	• NVIDIA Quadro FX 5800	• CUDA	• GPU cores in hybrid systems modelled as NVIDIA GPU multiprocessors • Only single CPU core simulation	• Both discrete and hybrid CPU+GPU systems simulation capabilities
[66]	• Event driven simulation	• Hybrid chips	• N/A	• Simulator specific	• Necessary to reimplement applications using their API	• Suitable for the design of new architectures • Power modelling features

TABLE 4
Summary of simulators

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 run-time characteristics. We discuss in this section several contributions to this field. A common feature of all of them is the mechanisms used to obtain the data required to feed the power model: a set of micro-benchmarks that must be executed while measuring the energy usage of the computing platform with a power meter.

Hong et al. [48] proposed an analytical power and performance model for NVIDIA GPUs, based on the previously presented MWP-CWP model (introduced in Section 4.4). They compute the total power consumed by a GPU as the addition of the idle power plus the active power (that caused by running an application). To model the power used by the multiprocessors when active, they add the consumption of each of the units that form a SM, such as the ALUs, double-precision floating-point units, or the register file. The MWP-CWP analytical model is used to estimate the execution cycles of the application. This information, together with the power model, is combined, providing an estimation of the power consumed while running the application. The model is actually more complex, taking into consideration additional factors such as the thermal effects of the device.

Wang et al. in [94] present a simpler analytical model to estimate power consumption. They compute the energy consumed by a kernel as the energy consumed by all the thread blocks which, in turn, is computed as the energy consumed by all its threads. The power used by a thread is the sum of the costs of its instructions. The list of instructions run by a thread is obtained parsing the PTX intermediate code. The power signature of each instruction is gathered via a collection of micro-benchmarks. This power model is simple but accurate. Authors state that, although gathering low-level information from the binary would produce more accurate results, the PTX-level information is good enough for obtaining reasonable predictions. In their experiments, they prove the effectiveness of their model reducing the power consumption of a CUDA kernel between 7.31% and 11.76% on average, causing an increase of the execution time of 4.92% in the worst case.

Ma et al. have published several works [63] [64] in which they discuss how to generate statistical models to predict the power consumption of GPU kernel, using machine learning techniques. They create training datasets running several CUDA applications on a GPU, retrieving two types of metrics: performance counters (which are obtained using a profiler) and power consumption measurements (gathered through an external power acquisition system). This training dataset is used together with different types of regressors. The main contribution of this approach is the methodology used for retrieving power consumption metrics. Experimental results show good levels of accuracy in the predictions, being the mean square errors within 1.7% and 33.7% [63]. Authors point out that some inaccuracies are due to non-deterministic power consumptions issues not captured during the training process. They also state that the bus communication or the memory access patterns affect power consumption, and that these facts must be taken into account in future, more detailed models.

GPUWattch [60] is a power model integrated with GPGPU-Sim version 3.2 and later. It models the main components of an NVIDIA GPU, i.e., the streaming multiprocessors, interconnection network, memory controllers, and main memory. The model generates an output file per kernel executed in GPGPU-Sim, which includes among other metrics the power consumption of each hardware component. This output can be later used by GPGPU-Sim to perform runtime optimizations in the simulation, such as emulating dynamic voltage and frequency scaling (DVFS). Authors tested the accuracy of their power model measuring the actual power utilization of two GPUs with different architecture (NVIDIA GTX 480 and Quadro FX5600). Tests were carried out running a suite of micro-benchmarks and some complete applications (which included the Rodinia benchmark suite). Average errors for the micro-benchmarks were 15% and 16.2% (respectively for the GTX and the Quadro). Average errors were smaller for the applications: 9.9% and 13.4%. Authors claim that using DVFS, GPU energy consumption could be reduced up to 14.4%, with less than 3% of performance loss. GPUWattch is based on the McPat power modelling framework [61], which targets multi-core and many-core architectures.

5 Conclusions

With the appearance of new computing devices and the design of new algorithms in different fields of science and technology, the way to use high performance computing is changing. Designing and developing programs that use currently available computing resources efficiently is not an easy task. As stated in [16], 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 [54], and improving power efficiency reducing the total transistor count.

Tools to help programers in this parallel and heterogeneous environment (debuggers, profilers, etc.) are slowly becoming available, but the programer 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 easier the process of choosing the right options (device, program settings, optimizations, etc.) in order to efficiently run accelerator-based programs. These models are not perfect, all of them have some 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. Every model finds a different trade-off between being more device-specific and therefore more accurate (e.g. [86]), or being more general purpose at the cost of losing accuracy (e.g. [70]). Related to this topic is the fast pace at which manufacturers introduce new products, with new or improved features, into the market, making models obsolete in a very short time. Performance models should be flexible enough to allow characterizing the evolution of devices.
- Most of the models discussed in this review have been designed for NVIDIA GPUs and the CUDA Platform. This is due to the fact that CUDA was the first "popular" environment for GPGPU, with a larger, more mature ecosystem of applications and tools than OpenCL. However the vendor neutrality of OpenCL and its availability for non-GPU accelerators is increasing its use by HPC programmers. In the past, OpenCL toolsets (compilers, optimizers) produced less efficient codes than the CUDA counterparts, but this is no longer true with the most current, better fine-tuned versions of OpenCL SDKs [35],[52].
- Society in general and the HPC community in particular is becoming aware of the great costs, in monetary and also in environmental terms, derived from the high power consumption of computing systems. The challenge nowadays 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 trade-offs have to be found. The models published in the literature and reviewed in this paper are tools that can help solve this bi-

Model	Method	Target platforms	Tested devices	Input preprocessing	Limitations	Additional features
[48]	• Analytical	• NVIDIA GPUs	• NVIDIA GTX280	• μBenchmarks execution • PTX analysis using Ocelot	• Complex control flow kernels not correctly modelled	• Thermal effects modelled
[94]	• Analytical	• NVIDIA GPUs	• NVIDIA GTX460	• μBenchmarks execution • PTX analysis	• Overheads of branch instructions not modelled	• Simple yet accurate
[63] [64]	• Machine Learning	• NVIDIA GPUs	• NVIDIA 8800 GT	• μBenchmarks execution • Kernel analysis using profiler	• Power consumption peaks not modelled accurately	• N/A
[60]	• Analytical	• NVIDIA GPUs	• NVIDIA GTX480 • NVIDIA Quadro FX5600	• μBenchmarks execution • Kernel execution in GPGPU-Sim	• Short running kernels not modelled accurately	• Seems easily extendable to new architectures • Module to simulate feed-back driven runtime optimizations using GPGPU-Sim

TABLE 5
Summary of models for power consumption estimation

objective optimization problem.

- We have repeatedly stated that computing systems are becoming hybrid. When different resources are available, all of them able to run a given code (for example, an OpenCL kernel), a decision has to be made about which device should be used. This decision has to consider performance, power efficiency, or both. Grewe et al. [44] propose a tool that generates an OpenCL application from an existing OpenMP one. Both versions are packed into a single binary code, together with a machine learning module. This module analyzes the underlying hardware and the application parameters to decide, at run time, which code version must be executed and on which device.
- Reviewed models focus on outsourcing compute-intensive tasks to accelerator devices. However, this usually means leaving the CPU idle while the GPU (or any other accelerator) is busily crunching numbers. It is possible, however, to share the computation effort between all resources available, significantly increasing system efficiency. There are proposals in the literature that deal 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 [22], [95]. 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. [41], [43].

When preparing a literature survey such as this, most of our frustrations are caused by the lack of detailed information about tools and models, which makes reproducing the experimental work almost impossible. Some of the papers we have reviewed neither indicate clearly which program features are needed by the models nor the way used to obtain them (from the program source? after running an application? using programmer-embedded traces? using external tools?). In addition, in order to properly validate the accuracy of a model, a wide and representative data set must be obtained, and cross-validation techniques must be properly used. If this is not performed correctly, the models tend to overfit, providing very accurate results for the data used for training, but poor estimates for new, unseen data. Finally, it would be beneficial for the community to have free access to the source codes of the models and the 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 partially supported by the Saiotek and Research Groups 2013-2018 (IT-609-13) programs (Basque Government), TIN2010-14931 (Ministry of Science and Technology), COMBIOMED network in computational biomedicine (Carlos III Health Institute), and by the NICaiA Project PIRSES-GA-2009-247619 (European Commission). Unai Lopez holds a grant from Basque Government. Authors of this paper are members of the HiPEAC European Network of Excellence on High Performance and Embedded Architecture and Compilation.

REFERENCES

[1] Arm home page. http://www.arm.com, 2012.

[2] Gpgpu community. http://www.gpgpu.org, 2012.

[3] Openacc - directives for accelerators. http://www.openacc-standard.org/, 2012.

[4] Top500 supercomputing sites. http://www.top500.org, 2012.

[5] Arm mali graphics plus gpu compute. http://www.arm.com/products/multimedia/mali-graphics-plus-gpu-compute, Jan. 2013.

[6] Barra gpu simulator. https://code.google.com/p/barra-sim/, Apr. 2013.

[7] Gpgpu-sim online manual. http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual/, Apr. 2013.

[8] Multi2sim. http://www.multi2sim.org, Apr. 2013.

[9] Amazon. Hpc in aws. http://aws.amazon.com/hpc-applications/, Apr. 2013.

[10] AMD. Accelerated parallel processing (app) sdk. http://developer.amd.com/tools/hc/AMDAPPSDK, 2012.

[11] AMD. Accelerated parallel programming opencl best practices guide. http://developer.amd.com/tools/hc/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf, Jan. 2012.

[12] AMD. Core math library (acml). http://developer.amd.com/tools/cpu-development/amd-core-math-library-acml/, 2012.

[13] AMD. Codexl. http://developer.amd.com/tools/heterogeneous-computing/codexl, Apr. 2013.

[14] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. McMahon, F. Pasquier, G. Péan, and P. Villalon. Par4all: From convex array regions to heterogeneous computing. In *2nd International Workshop on Polyhedral Compilation Techniques, Impact (Jan 2012)*, 2012.

[15] A. Ariel, W. W. Fung, A. E. Turner, and T. M. Aamodt. Visualizing complex dynamics in many-core accelerator architectures. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 164–174. IEEE, 2010.

[16] M. Arora, S. Nath, S. Mazumdar, S. Baden, and D. Tullsen. Redefining the role of the cpu in the era of cpu-gpu integration. 2012.

[17] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp, and W. Hwu. An adaptive performance modeling tool for gpu architectures. In *ACM SIGPLAN Notices*, volume 45, pages 105–114. ACM, 2010.

[18] S. Baghsorkhi, I. Gelado, M. Delahaye, and W. Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 23–34. ACM, 2012.

[19] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, et al. The nas parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165. IEEE, 1991.

[20] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. Ieee, 2009.

[21] A. Bayoumi, M. Chu, Y. Hanafy, P. Harrell, and G. Refai-Ahmed. Scientific and engineering computing using ati stream technology. *Computing in Science and Engineering*, 11(6):92–97, 2009.

[22] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):57, 2013.

[23] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *Micro, IEEE*, 32(2):28–37, 2012.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[25] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107. IEEE, 2008.

[26] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2010.

[27] J. Choi, A. Singh, and R. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. *ACM Sigplan Notices*, 45(5):115–126, 2010.

[28] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: a parallel functional simulator for gpgpu. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 351–360. IEEE, 2010.

[29] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From opencl to high-performance hardware on fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534. IEEE, 2012.

[30] M. Daga, A. Aji, and W. Feng. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 141–149. IEEE, 2011.

[31] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[32] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[33] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT 10, page 353364, New York, NY, USA, 2010. ACM.

[34] J. Diaz, C. Muñoz-Caro, and A. Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[35] J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[36] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[37] P. Ganestam and M. Doggett. Auto-tuning interactive ray tracing using an analytical gpu architecture model. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 94–100. ACM, 2012.

[38] G. García and C. Yenyxe. Modelo de estimación de rendimiento para arquitecturas paralelas heterogéneas. 2013.

[39] B. R. Gaster. Introductory opencl tutorial. http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/introductory-tutorial-to-opencl/, Apr. 2013.

[40] Gatech. macsim simulator for heterogeneous architecture. http://code.google.com/p/macsim/, Jan. 2012.

[41] I. Grasso, K. Kofler, B. Cosenza, and T. Fahringer. Automatic problem size sensitive task partitioning on heterogeneous parallel systems. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 281–282. ACM, 2013.

[42] P. Greenhalgh. Big. little processing with arm cortex.-a15 & cortex-a7. *ARM Whitepaper*, 2011.

[43] D. Grewe and M. O Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Compiler Construction*, pages 286–305. Springer, 2011.

[44] D. Grewe, Z. Wang, and M. F. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *CGO '13: Proceedings of the 11th International Symposium on Code Generation and Optimization*. ACM, 2013.

[45] K. Group. Opencl standard portable intermediate representation (spir) 1.0. http://www.khronos.org/registry/cl/specs/spir_spec-1.0-provisional.pdf, 2012.

[46] K. O. W. Group et al. The opencl specification. *A. Munshi, Ed*, 2008.

[47] 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*, 37(3):152–163, 2009.

[48] S. Hong and H. Kim. An integrated gpu power and performance model. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 280–289. ACM, 2010.

[49] Intel. Math kernel library (mkl) home page. http://software.intel.com/en-us/intel-mkl, 2012.

[50] Intel. Xeon phi coprocessor: Parallel processing, unparalleled discovery. http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html, 2012.

[51] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005.

[52] K. Karimi, N. G. Dickson, and F. Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[53] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling gpu-cpu workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42. ACM, 2010.

[54] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W.-m. Hwu. Performance analysis and tuning for general purpose graphics processing units (gpgpu). *Synthesis Lectures on Computer Architecture*, 7(2):1–96, 2012.

[55] D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[56] K. Kothapalli, R. Mukherjee, M. Rehman, S. Patidar, P. Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu platform. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 463–472. Ieee, 2009.

[57] C. Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference, Ottawa, Canada*, 2008.

[58] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[59] V. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.

[60] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th annual international symposium on Computer architecture*, 2013.

[61] 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. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[62] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[63] X. Ma, M. Dong, L. Zhong, and Z. Deng. Statistical power consumption analysis and modeling for gpu-based computing. In *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[64] X. Ma, M. Rincon, and Z. Deng. Improving energy efficiency of gpu based general-purpose scientific computing through automated selection of near optimal configurations. 2011.

[65] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, pages 256–265. ACM, 2009.

[66] J. Meng and K. Skadron. A reconfigurable simulator for large-scale heterogeneous multicore architectures. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 119–120. IEEE, 2011.

[67] T. Mitchell. Machine learning. wcb, 1997.

[68] J. Nickolls and W. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, 2010.

[69] C. Nugteren and H. Corporaal. A modular and parameterisable classification of algorithms. Technical report, Technical Report No. ESR-2011-02, Eindhoven University of Technology, 2011.

[70] C. Nugteren and H. Corporaal. The boat hull model: enabling performance prediction for parallel computing prior to code development. In *Proceedings of the 9th conference on Computing Frontiers*, pages 203–212. ACM, 2012.

[71] NVIDIA. Cublas home page. http://developer.nvidia.com/cuda/cublas, 2012.

[72] NVIDIA. Cuda home page. http://developer.nvidia.com/cuda, 2012.

[73] NVIDIA. Cuda tools and ecosystem. http://developer.nvidia.com/cuda/cuda-tools-ecosystem, 2012.

[74] NVIDIA. Cufft home page. http://developer.nvidia.com/cuda/cufft, 2012.

[75] NVIDIA. cuobjdump application note, 2012.

[76] NVIDIA. Nvidia's next generation cuda compute architecture: Kepler gk110. Technical report, White Paper, 2012.

[77] NVIDIA. Opencl best practices guide. www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf, Jan. 2012.

[78] NVIDIA. Cuda education and training. http://developer.nvidia.com/cuda-education-training, Apr. 2013.

[79] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[80] E. F. Patterson, B. J. Archer, and C. L. Wampler. Roadrunner-on the road to trinity. Technical report, Los Alamos National Laboratory (LANL), 2013.

[81] N. Rajovic, N. Puzovic, L. Vilanova, C. Villavieja, and A. Ramirez. The low-power architecture approach towards exascale computing. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, pages 1–2. ACM, 2011.

[82] J. Rice and R. Boisvert. Solving elliptic problems using ellpack. *Springer Series in Computational Mathematics, New York: Springer, 1985*, 1, 1985.

[83] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, and W. Hwu. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204. ACM, 2008.

[84] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008.

[85] S. Seo, G. Jo, and J. Lee. Performance characterization of the nas parallel benchmarks in opencl. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148. IEEE, 2011.

[86] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 11–22. ACM, 2012.

[87] K. Spafford, J. Meredith, S. Lee, D. Li, P. Roth, and J. Vetter. The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In *Proceedings of the 9th conference on Computing Frontiers*, pages 103–112. ACM, 2012.

[88] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[89] R. Taylor and X. Li. A micro-benchmark suite for amd gpus. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 387–396. IEEE, 2010.

[90] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2sim: a simulation framework for cpu-gpu computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 335–344. ACM, 2012.

[91] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. Oct. 2007.

[92] W. van der Laan. Decuda and cudasm, the cubin utilities package, 2009.

[93] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. On the limits of gpu acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pages 13–13. USENIX Association, 2010.

[94] Y. Wang and N. Ranganathan. An instruction-level energy estimation and optimization methodology for gpu. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 621–628. IEEE, 2011.

[95] Z. Wang, L. Zheng, Q. Chen, and M. Guo. Cap: co-scheduling based on asymptotic profiling in cpu+ gpu hybrid systems. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 107–114. ACM, 2013.

[96] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[97] C. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. *Micro, IEEE*, 31(2):50–59, 2011.

[98] M. Wolfe. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010.

[99] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. Ieee, 2010.

[100] M. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34. IEEE, 2007.

[101] V. Zakharenko. *FusionSim: Characterizing the Performance Benefits of Fused CPU/GPU Systems*. PhD thesis, University of Toronto, 2012.

[102] Y. Zhang and J. Owens. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393. IEEE, 2011.

[103] Y. Zhang, L. Peng, B. Li, J. Peir, and J. Chen. Architecture comparisons between nvidia and ati gpus: Computation parallelism and data communications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 205–215. IEEE, 2011.

Unai Lopez Unai Lopez Novoa received the MSc in Computer Science from the University of Deusto in 2010. He is currently a PhD student at the Department of Computer Architecture and Technology of the University of the Basque Country UPV/EHU. His main research area is parallel and distributed computing and specially, massively parallel computing using accelerators.

Alexander Mendiburu Alexander Mendiburu received a B.S. degree in Computer Science and a Ph.D. degree from the University of the Basque Country, Gipuzkoa, Spain, in 1995 and 2006 respectively. Since 1999, he has been a Lecturer in the Department of Computer Architecture and Technology, University of the Basque Country. His main research areas are evolutionary computation, probabilistic graphical models, and parallel computing.

Jose Miguel-Alonso Jose Miguel-Alonso received in 1989 the MS and in 1996 the PhD, both in Computer Science, from the University of the Basque Country (UPV/EHU), Gipuzkoa, Spain. He is currently Full Professor at the Department of Computer Architecture and Technology of the UPV/EHU. Previous positions include one year as Visiting Assistant Professor at Purdue University (USA). 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 in these topics. Dr. Miguel-Alonso is a member of the IEEE Computer Society.