Extending a Run-time Resource Management framework to support OpenCL and Heterogeneous Systems

Giuseppe Massari
DEIB - Politecnico di Milano
Via Ponzio 34/5, 20133
Milano, Italy
giuseppe.massari@polimi.it

Chiara Caffarri
Università di Parma
Via Università 12I, 43121
Parma, Italy
chiara.caffarri@studenti.unipr.it

Patrick Bellasi
DEIB - Politecnico di Milano
Via Ponzio 34/5, 20133
Milano, Italy
patrick.bellasi@polimi.it

William Fornaciari
DEIB - Politecnico di Milano
Via Ponzio 34/5, 20133
Milano, Italy
william.fornaciari@polimi.it

ABSTRACT

From Mobile to High-Performance Computing (HPC) systems, performance and energy efficiency are becoming always more challenging requirements. In this regard, heterogeneous systems, made by a general-purpose processor and one or more hardware accelerators, are emerging as affordable solutions. However, the effective exploitation of such platforms requires specific programming languages, like for instance OpenCL, and suitable run-time software layers. This work illustrates the extension of a run-time resource management (RTRM) framework, to support the execution of OpenCL applications on systems featuring a multi-core CPU and multiple GPUs. Early results show how this solution leads to benefits both for the applications, in terms of performance, and for the system, in terms of resource utilization, i.e. load balancing and thermal leveling over the computing devices.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling, Kernels

General Terms

Design, Algorithms, Performance

Keywords

Runtime, OpenCL, Graphical Processing Units (GPUs), Multicore, Heterogeneous systems, OpenCL, Parallel programming, Profiling

1. INTRODUCTION

In the last decades, the growning request of performance in the computing systems has pushed the chip manufacturing technology towards always higher integration processes.

At first, this evolution bound the increment of the processors' performance to the rise of their frequency of work. In the last years, this trend has changed, due to the emerging thermal and power management issues. The frequency rising has given way to the increasing level of architectural parallelism (multi-core and many-core processors).

Nowadays, with the market of battery-powered mobile devices exploding, and the growth of data center and cloud computing systems, the control over the energy consumption has become another big issue. *Performance* and *energy consumption* determine a trade-off, that we can treat as a whole talking about *energy efficiency*. Roughly speaking, this concept is usually defined in computing systems as the ratio between performance achieved and power consumed for that purpose.

From the hardware perspective, it is well known how the maximum energy efficiency can be achieved by spending silicon area in ASIC, or exploiting application-specific computing units (e.g., DSP). In this case, the price is paid in terms of flexibility, time and costs. As an alternative, these drawbacks can be strongly mitigated by resorting to heterogeneous platforms. In this work, we will refer to platfoms usually featuring a general purpose multi-core processor along with hardware accelerators, as many-core processors or Graphical Processing Units (GPUs).

The effective exploitation is one of the most critical issues in this kind of platforms. Suitable programming languages are indeed required, along with run-time software layers. In this regard, OpenCL is a noteworthy and widespread parallel programming language, targeting code execution on both CPU processors and GPU adapters. Typically, an OpenCL application is made by *kernels*, namely code performing parallel computations on a selected device, and a sequential *host code*, in charge of setup, synchronize the kernels execution, and manage data transfers between host and device memory. The interaction between host and devices is based on OpenCL *commands*, that are properly queued.

According to this schema, as first step, the applications are in charge of query the OpenCL runtime, in order to know the set of all the available computing devices. Afterwards, the applications select the device(s) on which execute the kernels. This point becomes extremely critical, especially if we consider scenarios with several OpenCL applications running on heterogeneous platforms, featuring a multi-core CPU and multiple GPUs. Indeed, since each application is unaware of the others, and has no knowledge about the status of the computing devices, it is easy to observe a bad utilization of system resources. In other words, we would have underutilized devices alongside overutilized ones, with whatever concern loss of performance, and other side effects related to thermal and power management.

What is missing to manage a scenario like this is a third party arbiter, having a *system-wide* perspective. This in order to allocate computing resources, with the aim of maximizing the overall performance or balancing the usage of all the devices.

This paper introduces a work in progress extension of a framework, aiming at enabling run-time resource management of OpenCL applications running on heterogeneous systems, featuring a multi-core CPU and multiple GPUs.

1.1 Related works

Several projects have investigated how to alleviate the programmer from the burden of managing hybrid or heterogeneous platforms, building systems where applications would spread across the entire machine.

StarPU [1] [2] is a runtime system targeting heterogeneous multicore architectures equipped with GPU adapters. It provides a unified high-level programming interface, but adopting an abstraction to model tasks, the *codelet*, that forces the developer to declare tasks with their data dependencies and write a driver for each new architecture that might be targeted. The programmer can specify the level of priority of a task, just like our target framework, but conversely from StarPU, the application developer should only rearrange its code in the reconfigurable execution model exported by the application run-time library, without specifying data dependencies and writing driver functions. SkePU [5] [6] is a C++ template library designed to support parallel programming and it provides a multi-GPU support. It is based on skeleton programming, so it requires the programmer to rewrite a program using skeletons, i.e., pre-defined generic components that capture, organize and mask to the user all the details involved in the parallel computation structure, that are not relevant to the user code. In addition, all those computations that do not fit a given skeleton must be rewritten.

Qilin et *al.*, face changes in runtime environments, like hardware/software configurations, or input problem size, with their heterogeneous programming system [7], featuring an automatic *adaptive mapping*. The solution is built on top of NVIDIA CUDA for executing code on GPUs, and thus the communication between the CPU and GPU is managed by CUDA drivers. Qilin et *al.* provide libraries that substantially wrap function calls to the vendor's ones, like we did in our solution, but they still allow the programmer to optionally select the processing elements to use. We believe that this approach does not fit well in the scenario described in the previous paragraph. Indeed, as we are going to see, we delegate mapping decisions to the resource manager scheduling policy, in order to pursue a system-wide optimization in resource allocation. Moreover, the solution proposed by Qilin et *al.* focuses on improving performance and energy consumption of a *single* application, while our target framework aims at taking into account all the applications to run.

Finally, OpenCLoSE middleware [3] exploits offline profiling and static priorities to take scheduling decisions at run-time, but it requests the application to be partitioned into chunks, namely entities including a set of computations and the associated memory transfers. Moreover, the OpenCLosE source code does not seem to be available, actually.

2. RUN-TIME RESOURCE MANAGEMENT

This section describes the work done to integrate the support to OpenCL and heterogeneous platforms into an existing run-time resource management framework. The integration flow is resumed in Figure 3, and shows how different parts of the framework have been involved. In this regard, next sections will provide some details.

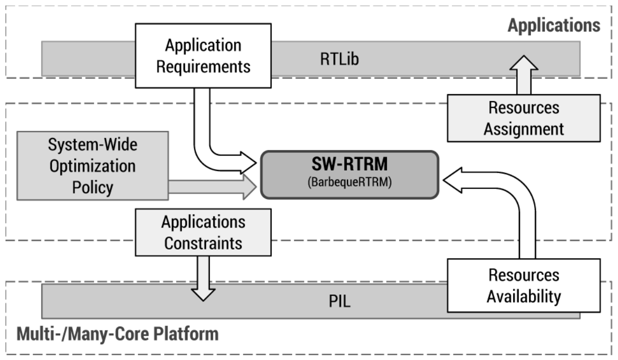


Figure 1: Overall view of the BarbequeRTRM.

2.1 The BarbequeRTRM framework

The run-time resource management approach that we aim to extend is the BarbequeRTRM framework[1]. Figure 1 provides an overall view of the interactions between its core module, i.e. the *System-Wide Run-Time Resource Manager* (SW-RTRM), the applications, and the hardware platform.

From an high abstraction level, the SW-RTRM module takes care of collecting two types of information: 1) resource requirements from running applications and 2) run-time variability of the computing resources availability. These two kind of inputs can trigger a *system-wide optimization step*, where in a scheduling and resource allocation policy is executed, in order to dynamically redefine the caomputational *resource partitioning* among the demanding applications. The resource management is performed on an *event-driven* basis (applications starting, finishing, sending requests, . . .). Once a new resource partitioning has been defined, the SW-RTRM module performs all the control actions required to setup platform specific constraints and to notify the applications involved.

The integration of the framework in a hardware system is operated at two layers: the first one involves the hardware platform, while the second one the applications. For the former, a *Platform Integration Layer* (PIL) is in charge of monitoring the status of resources, and enforcing the constraints defined by the optimization policy. This is usually a platform-specific module. For the applications, a *Run-Time Library* (RTLib) is provided, to transparently setup the communication channel between the resource manager and the application. The `RTLib` exports a set of services, that allows the application to send resource requests, and get notifications about resource assignment changes. A detailed description of the framework and the optimization policy can be found in [4]. In the next section we briefly recall the policy, and focus on the changes introduced for the management of heterogeneous systems.

2.2 Run-time resource allocation

The resource allocation problem has been formalized as a Multi-dimensional Multi-choice Knapsack Problem. Accordingly, the BarbequeRTRM approach requires that each application provides a set of resource requests options. Such options are called *Application Working Modes* (AWMs), and are provided through an XML file, called *Recipe* in jargon. Every single AWM is characterized by a set of resources (type and amount required), along with an integer

[1]Project website at http://bosp.dei.polimi.it

value, expressing the level of QoS or performance associated.

The optimization policy is a heuristic, performing a system-wide multi-objective optimization. It allocates resources taking into account the overall QoS/performance, the fairness, the load balancing, the task migration and reconfiguration overhead.

Therefore, for each application to run, the policy selects an AWM, and *binds* the requested resources to the physical system resources, trying to make the best choice in a system-wide perspective. This is repeated for all the application priority classes. We extended this algorithm by enabling the binding step for requests of computing resources that must be bound to GPU, as well as to CPU. As a consequence, the optimization policy becomes able to schedule OpenCL applications, allocating heterogeneous computing resources.

2.3 OpenCL library-in-the-middle

The design and the implementation of the whole extension has been driven by the requirement of being as less invasive as possible. This means do not introduce customizations in the OpenCL specifications, and do not burden the developer with a lot of code to add or rewrite.

Since the OpenCL applications are usually linked to a library provided by hardware vendors, we decided to adopt a sort of *library-in-the-middle* strategy. The basic idea is to intercept the OpenCL function calls, with the main purpose of moving the control over the system resources, i.e. the computing devices, from the application to the resource manager. As shown in the previous section, the BarbequeRTRM provides a library (RTLib) that allows the applications to interact with the resource manager daemon. Therefore, the library-in-the-middle is an extension of the current `RTLib`, where in the OpenCL functions have been redefined, in order to wrap at the OpenCL function calls performed at run-time by the applications. It is worth to say that most of the wrapper functions simply forward the function call to the original OpenCL library functions. Actually, additional logics has been implemented just for the functions that can be considered "sensitive", from the point of view of the access to resources (e.g., `clGetDeviceIDs`).

More in detail, we integrated the Platform Integration Layer module with the code lines commonly used by the OpenCL applications to setup the access to the computing devices. This allows the BarbequeRTRM to enumerate the set of devices accessible to the OpenCL runtime. Then, whenever an OpenCL application asks for a device descriptor, the resource manager, through the `RTLib`, returns just the descriptor associated to a single computing device, assigned by the resource allocation policy. The overall picture depicts a very low overhead solution. From the application side, this configures the *automatic device selection*, that introduces benefits from the system-wide perspective, as we are going to see in Section 3, as well as from the application developer point of view, since it is lift from the burden of implementing a kind of logics for the selection of the computing device. Please consider that actually, the framework allows the correct management of single device applications, the multi-device option is indeed planned as a future extension.

In addition to the run-time resource management, this wrapping approach enables the possibility of profiling the execution of the OpenCL commands queued by the application. Next section moves the focus on this point.

2.4 Command-level profiling

One of the features offered by the `RTLib` is the possibility to collect performance counter based statistics, to profile the behavior of parallel applications running on CPU. In this work, we extended

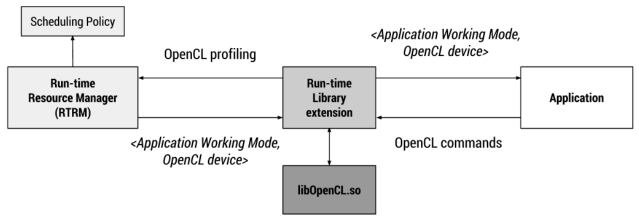

Scheduling Policy
OpenCL profiling
<Application Working Mode, OpenCL device>
Run-time Resource Manager (RTRM)
Run-time Library extension
Application
<Application Working Mode, OpenCL device>
OpenCL commands
libOpenCL.so

Figure 2: Interaction between the OpenCL application and the BarbequeRTRM with the RTLIb extension acting as library-in-the-middle ahead of the original OpenCL library.

this capability with the support to the OpenCL command-level profiling. To do that, we started from the event-based profiling API provided by the OpenCL API, thanks to which it is possible to extract timing information about the permanence of a command inside a queue, or about its execution.
Actually, the OpenCL application developer can declare an event object (`cl_event`), and bind it to a command to queue. The run-time fills the object with the timing information, related to each stage of the command processing (i.e., *queued*, *submitted* and *executed*). With the `RTLib` extension, the developer can avoid the declaration of such event objects, and let the library to collect (and dump) profiling information on all the OpenCL functions supporting the event profiling, by simply defining an environmental variable. What happens in this case is that event objects are defined by the library in a way that is completely transparent to the developer.

An interesting point to consider is that, since a large part of OpenCL applications falls in the class of streaming processing, their execution is usually characterized by a pretty regular processing cycle. As a consequence, by collecting timing information for each command queued in each cycle iteration, the `RTLib` can build profiling statistics for each command *type*, and call *instance*. Such statistics are useful, not only to support the developer in performance optimization, but also for resource management. Indeed, we can exploit such statistics both at *design-time* and at *run-time*. In the former case, it can be done by supporting the identification of the set of Application Working Modes. In the latter, by driving the resource allocation policy with an online profiling of the application. In Paragraph 3.3 an example of the former case is provided.

Finally, Figure 2 resumes the interaction between the application and the BarbequeRTRM. The RTLib extension intercepts the OpenCL function calls. In case of OpenCL profiling support is enabled, the timing information are collected. Whenever the application performs a query to retrieve the set of the available computing devices, the `RTLib` looks at the device ID assigned by the policy, and returns the device descriptor to the application.

3. EXPERIMENTAL RESULTS

To evaluate the benefits introduced by this framework extension, we executed scenarios featuring some OpenCL applications on a heterogeneous system made by a multi-core CPU and multiple GPUs. In Paragraph 3.2 we are going to see how the automatic device selection mechanism, driven by the run-time resource manager policy, positively affect the performance of the applications applications and the system load and thermal balancing.

3.1 Experimental Setup

We characterized the performance of our proposed work exploiting three OpenCL applications taken from the samples included in the AMD Accelerated Parallel Processing (APP) SDK (version 2.8), and a OpenCL implementation of the Stereo-matching algo-

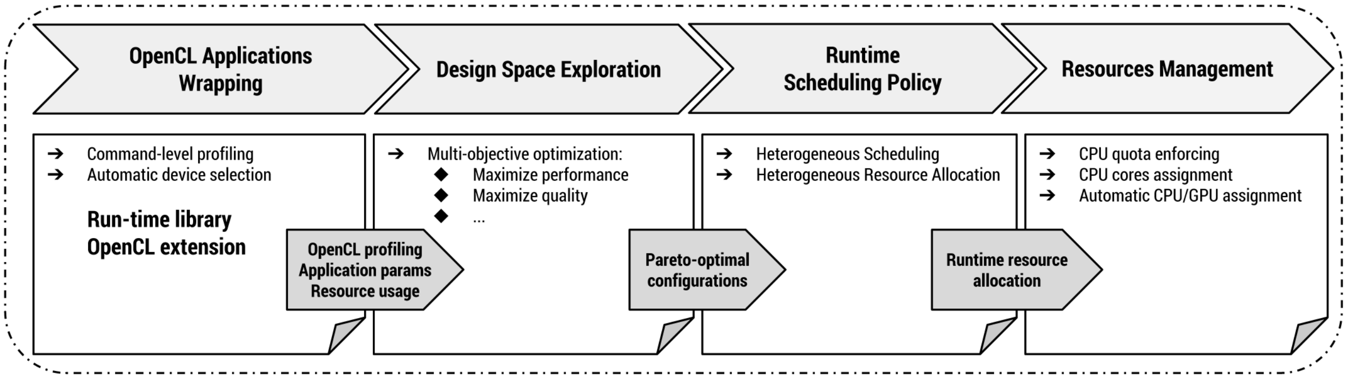

OpenCL Applications Wrapping
Design Space Exploration
Runtime Scheduling Policy
Resources Management
➔ Command-level profiling
➔ Automatic device selection
Run-time library OpenCL extension
OpenCL profiling
Application params
Resource usage
➔ Multi-objective optimization:
◆ Maximize performance
◆ Maximize quality
◆ ...
Pareto-optimal configurations
➔ Heterogeneous Scheduling
➔ Heterogeneous Resource Allocation
Runtime resource allocation
➔ CPU quota enforcing
➔ CPU cores assignment
➔ Automatic CPU/GPU assignment

Figure 3: The integration flow: RTLib extension, DSE exploitation and run-time resource management.

rithm. Each application executes a single kernel at a time, and the host program consumes the output of the kernel.

The hardware system was composed by 1 CPU Intel i7 3770 Quad core with Hyper-Threading, and 2 GP-GPUs ATI Radeon HD 7750, running a Linux OS (kernel version 3.8) and providing the AMD OpenCL Runtime version 1.2. The tests have been executed keeping the frequencies of the CPU and the GPUs set to a fixed value. This in order to avoid the influence of uncontrolled frequency scaling activities. More in detail, the GPU frequencies have been locked to the minumum values, i.e., 300 Mhz for the cores and 400 MHz for the memory bus. Conversely, the CPU cores have been locked to the maximum frequency value, corresponding to 3,40 GHz.

3.2 Automatic device selection

This experiment has been performed by using the *NBody* sample from the AMD APP SDK (single device version). We observed the variation of completion time of the execution, *load* percentage and the *temperature* of the GPUs, considering scenarios with a number of instances ranging from 1 to 4. Since NBody represents a computational intensive application, it is easier for us to show the benefits of exploiting run-time resource management with OpenCL applications. We compared the BarbequeRTRM-managed execution to the usual unmanaged one.

Concerning the completion time, it is immediate to see in Figure 4a, how the run-time resource management is necessary to avoid performance penalty. Indeed, in the unmanaged case all the NBody instances select the same GPU device (ID 0). Accordingly, since the execution of each instance leads to a full utilization of the GPU (100% load), the completion time scale up by a factor equal to the number of running instances. Conversely, whenever more than one instance is running, the BarbequeRTRM-managed case distributes the allocation of the NBody instances over both the GPUs, with a consequent halving of the execution time. Accordingly, the GPU load reaches the 100% for both the devices in the BarbequeRTRM-managed cases (multiple instances), avoiding the extreme unbalancing reported in the unmanaged cases, i.e. 100% versus 0% (Figure 4b and Figure 4c).

As expected, the load balancing led also to a temperature levelling among the GPUs. The GPU temperature (Figure 5b) is almost the same in both cases, while Figure 5a shows that in the BarbequeRTRM-managed case the GPU 1 temperature increases from 32° C to 40° C, since it passes from idle to active.

The overall result is shown in Figure 5c, with the difference of temperature between GPUs dropping from 13-14° C to a range of 3-7° C. Temperature levelling is a very important result, that must be considered jointly to the reduction of execution time. Indeed, by distributing the kernels on both GPUs, the single GPU works for a shorter time span, with a two-fold benefit: a) less energy required to supply the fan cooling system; b) mitigation of the aging effects on the devices, due to thermal stress.

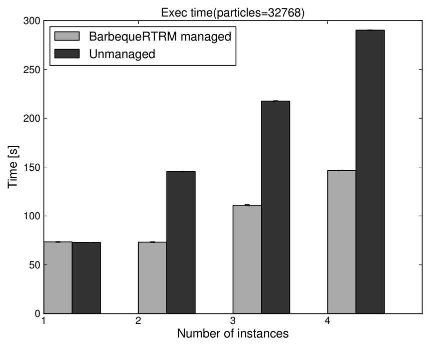
3.3 Command-level profiling exploitation

To identify a suitable set of Application Working Modes, exploiting a Design Space Exploration (DSE) is often a reasonable choice. Given a big space of configurations/options, DSE tools usually perform an effective exploration of the space, returning the Pareto optimal configurations, according to a set of metrics to optimize (e.g, performance and QoS maximization, energy consumption minimization, etc…).

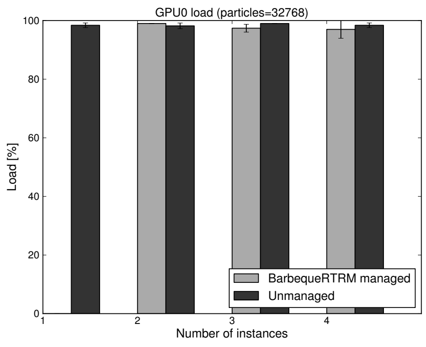
In this experiment, we tested the integration of the OpenCL profiling support with the DSE, considering an application performing StereoMatching computations. The choice has been driven by a pair of reasons. The former is that, the application performs a run-time monitoring of the frame-per-second rate (FPS). This let us to consider the FPS as a performance metrics, and thus an objective, to maximize. The latter reason is due to the presence of several parameters that, according to the value assigned, affect the computational capacity required to keep a certain QoS level or performance goal.

Briefly, our design space was composed by the set of values of four application-specific parameters (*hypo step* (HS), *confidence* (C), *max arm length* (MAL), *max hypo value* (MHV)), and two parameters related to heterogeneous resource usage (*CPU quota*(CPU) and *GPU assignment*(GPU)). For reasons of space, we cannot provide further details about how each parameter is related to the application behavior. The goals of the multi-objective exploration were: a) maximize the level of quality of the images processed (IQ); b) maximize the frame-per-second (FPS) rate, i.e. minimize the time of the processing cycle (CT); c) minimize the overhead (OH) due to data transfer, in case of execution on GPU. The exploration has been performed by MOST, a DSE tool developed in the context of the MULTICUBE project [8].

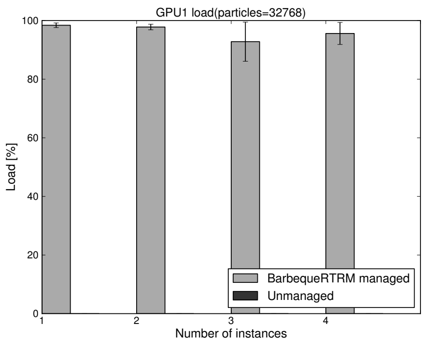
It is worth to remark that the last objective has been introduced thanks to the OpenCL command-level profiling support of the application run-time library. More in detail, we have defined the *overhead* as the ratio between the time spent to perform data memory transfers (time spent in executing `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` commands), and the time spent to perform real computations on the device (`clEnqueueNDRangeKernel`):



(a) *Execution time*

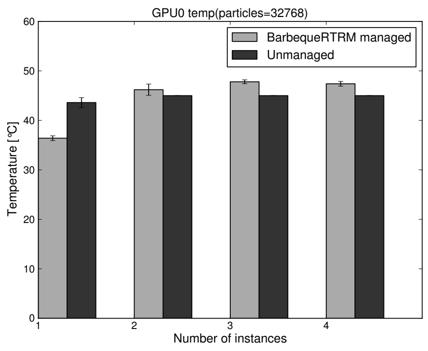


(b) *GPU 0 Load*

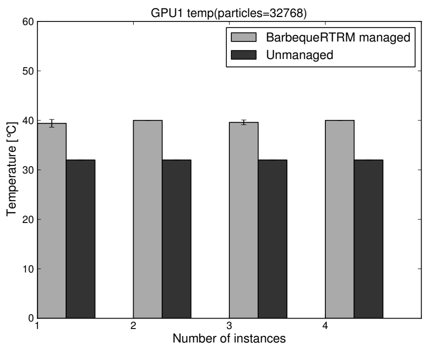


(c) *GPU 1 Load*

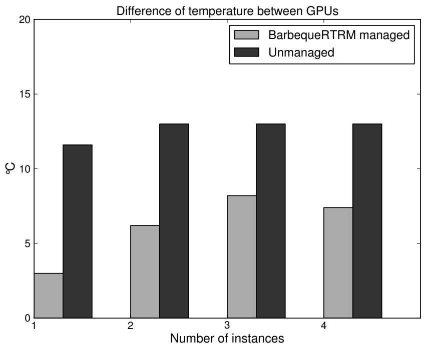
Figure 4: Automatic OpenCL device assignment: Completion time and GPUs load for the execution of the NBody sample.



(a) *GPU 0 Temperature*



(b) *GPU 1 Temperature*



(c) *GPUs Temperature difference*

Figure 5: Automatic OpenCL device assignment: GPUs temperature during the execution of the NBody sample.

$$OH = \frac{EnqueueReadBuffer_{time} + EnqueueWriteBuffer_{time}}{EnqueueNDRangeKernel_{time}} \quad (1)$$

This because data transfers are typically source of bottlenecks for the GPU performance exploitation. Therefore, we try to reward the configurations such that if a "big" data transfer is required, it should be worth only if the time spent for the execution of the kernel is "big" as well.

Considering just a finite number of values for each parameters, we started from a design space of about 200 configurations. The exploration has identified the Pareto optimal configurations listed in Table 1. Such configuration are actually called *Operating Points*. Then, from the list of Operating Points we can extract the Application Working Modes (grey rows), by ignoring application-specific parameters and "negligible" difference in the resource usage configurations. Therefore, at run-time, the resource allocation policy will allocate resources to StereoMatching considering just this set of options, and all the possible resource bindings, i.e. the available set of CPU cores and GPUs.

Once the AWM has been assigned, the application can select an Operating Point from a range, carrying out a re-tuning of the application parameters, aiming at following a runtime goal, i.e. for instance a certain FPS value. This is another service provided by the `RTLib`, but a detailed description is out of the scope of this work.

3.4 Heterogeneous resource allocation

In this test, we evaluated the behavior of the framework in a mixed workload scenario, with several applications competing for the system resources. The workload features OpenCL applications at different priorities. We considered two instances of NBody (high priority), two instances of StereoMatching (low priority), and an instance of MonteCarloAsianDP (from the AMD APP SDK samples) having a medium priority, at different instants of time. Higher priority has been assigned to the applications that have been profiled as the most GPU intensive, in order to effectively exploits the computing resources.

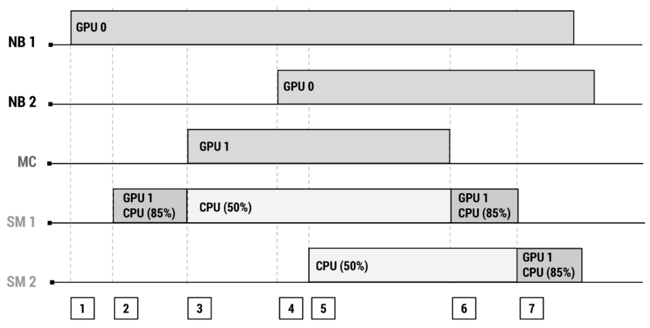


Figure 6: Heterogeneous resource allocation in a mixed workload scenario.

From the point of view of the resource allocation policy, we will focus on the choices made in terms of selection of AWM and resource binding for two StereoMatching instances, considering the results of the design space exploration previously discussed.

Figure 6 shows the timeline of the execution, along with the computing resource allocated to the applications. Please consider the

OP	HS	C	MAL	MHV	CPU	GPU	CT [s]	IQ [%]	OH	FPS	**AWM**
1	3	45	1	32	85	1	0.035	33	2.044	28.827	0
2	2	45	16	32	80	1	0.067	50	0.814	14.840	-
3	1	45	1	32	60	1	0.090	100	2.218	11.142	-
4	1	45	1	32	30	1	0.097	100	1.976	10.317	1
5	3	45	16	32	300	0	0.225	33	0.023	4.446	2
6	2	45	16	32	300	0	0.317	50	0.023	3.151	-
7	1	45	16	32	300	0	0.587	100	0.022	1.704	-
8	1	45	16	32	100	0	1.914	100	0.007	0.522	-
9	2	45	16	32	50	0	1.988	50	0.004	0.503	3
10	1	45	16	32	25	0	7.510	100	0.002	0.133	4

Table 1: Operating Points of the StereoMatching application resulting from the Design Space Exploration. The Application Working Modes (AWMs) are highlighted in grey.

AWMs highlighted in Table 1.

1. The first application launched is an instance of NBody (NB 1), which gets the access to GPU 0.
2. The first instance of StereoMatching (SM 1), to which the BarbequeRTRM assigns the AWM 0, featuring a GPU and 85% of the CPU quota.
3. MonteCarloAsianDP starts, and since it has a higher priority than StereoMatching, it obtains the GPU 1. This leads to the migration of SM 1 from GPU to CPU, with the assignment of the AWM 3 (50% of CPU).
4. A second instance of NBody (NB 2) is co-scheduled on GPU 0.
5. A second instance of StereoMatching (SM 2) is co-scheduled on CPU (50%) with the first one, since higher priority applications are still running.
6. MonteCarloAsianDP terminates and frees GPU 1, that is reassigned to SM 1, along with 85% of CPU quota (AWM 0).
7. When SM 1 terminates, the second instance of StereoMatching (SM 2) can take its place with GPU 1 and 85% of CPU quota assigned.

In summary, what happened is that the StereoMatching instances have been dynamically migrated between CPU and GPU, at runtime, without the implementation of any logics by the application developer. In the case of CPU allocation, the BarbequeRTRM has reserved to the application the amount of quota resulting from a multi-objective exploration performed at design time. Accordingly, the amount of resources allocated has always been a Pareto optimal choice. In some cases, like at (2), (6) and (7) an heterogeneous allocation has been performed, by assigning a CPU quota, and granting the access to a GPU. At (5) instead, five OpenCL applications were spread among different devices. Overall, the the BarbequeRTRM framework, along with the support of DSE has enabled an effective utilization of computing resources, from an heterogeneous system, by properly assigning devices to OpenCL applications, with the optimal amount of quota in case of CPU.

4. CONCLUSIONS AND FUTURE WORKS

In this paper we introduced a working run-time resource management framework, extended to support OpenCL applications, running on systems featuring a multi-core CPU and multiple GPUs. We have shown early benefits in terms of performance, load balancing and thermal leveling, deriving from the exploitation of a system-wide perspective over the availability of resources and the application requirements. We have also provided an example of integration of our OpenCL command-level profiling with a Design Space Exploration activity. It has been shown how this can support the resource allocation policy at run-time, by identifying only a subset of resource usage configurations that are worth to consider, given a multi-objective optimization. On going works are focusing on the exploitation of online profiling, to drive the resource allocation policy, the multi-device support, and a policy to improve the co-scheduling of kernels on the same GPU.

5. ACKNOWLEDGEMENTS

This work has been partially supported by the EU HARPA project FP7-ICT-2013-10-612069.[2]

6. REFERENCES

[1] C. Augonnet and R. Namyst. A Unified Runtime System for Heterogeneous Multi-core Architectures. In *Euro-Par 2008 Workshops-Parallel Processing*, pages 174–183. Springer, 2009.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In *Concurrency and Computation: Practice and Experience*, pages 187–198, February 2011.

[3] A. Bahga and V.K. Madisetti. A Dynamic Resource Management and Scheduling Environment for Embedded Multimedia and Communications Platforms. *Embedded Systems Letters, IEEE*, 3(1):24–27, March 2011.

[4] P. Bellasi, G. Massari, and W. Fornaciari. A RTRM proposal for multi/many-core platforms and reconfigurable applications. In *ReCoSoC*, 2012.

[5] U. Dastgeer, J. Enmyren, and C. Kessler. Auto-tuning SkePU: a Multi-Backend Skeleton Programming Framework for Multi-GPU Systems. In *Fourth Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2011)*, page 132, 2011.

[6] J. Enmyren, U. Dastgeer, and C.W. Kessler. Towards a Tunable Multi-Backend Skeleton Programming Framework for Multi-GPU Systems. In *MCC'10, Third Swedish Workshop on Multi-Core Computing, Göteborg, Sweden*, November 2010.

[7] C.K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2010.

[8] C. Silvano, W. Fornaciari, and E. Villar. *Multi-objective Design Space Exploration of Multiprocessor SoC Architectures: The MULTICUBE Approach*. Springer, 2011.

[2]http://www.harpa-project.eu/