A Runtime Controller for OpenCL Applications on Heterogeneous System Architectures*

Cristiana Bolchini
DEIB, Politecnico di Milano
cristiana.bolchini@polimi.it

Stefano Cherubin
DEIB, Politecnico di Milano
stefano.cherubin@polimi.it

Gianluca C. Durelli
DEIB, Politecnico di Milano
gianlucacarlo.durelli@polimi.it

Simone Libutti
DEIB, Politecnico di Milano
simone.libutti@polimi.it

Antonio Miele
DEIB, Politecnico di Milano
antonio.miele@polimi.it

Marco D. Santambrogio
DEIB, Politecnico di Milano
marco.santambrogio@polimi.it

ABSTRACT

Nowadays Heterogeneous System Architectures (HSAs) are becoming very attractive in the embedded and mobile markets thanks to the possibility to select the best computational resource among the available compute units to optimize the performance per Watt figure of merit. In this scenario, OpenCL is becoming the standard paradigm for heterogeneous computing supporting the programming of all types of units with a single abstraction level. However, the decision of the resource to use together with its architectural tuning is still left to the programmer; this issue is even more exacerbated when considering the fact that the choice depends also on the actual conditions in which the system is operating. This work aims at proposing a runtime controller, integrated in Linux Operating System (OS), for optimizing the power efficiency of a running OpenCL application deciding the system configuration. Our experimental results over a set of applications from the Polybench suite on the Odroid XU3 board show that our controller is able to obtain a power efficiency of more than 90% of the one achievable via offline profiling.

1. INTRODUCTION

HSAs [7] are becoming nowadays an attractive solution for achieving an optimal trade-off between performance and power/energy consumption thanks to the availability of different kinds of resources, such as Central Processing Units (CPUs), Graphic Processing Units (GPUs), Digital Signal Processors (DSPs) and other kinds of possibly reconfigurable HW accelerators. Examples are the Samsung Exynos 5 Octa chip [17], hosting an ARM big.LITTLE asymmetric octa-core CPU and an ARM Mali GPU, and the Xilinx Zynq [21], integrating an ARM dual-core CPU and a reconfigurable Field Programmable Gate Array (FPGA) unit.

However this increase in heterogeneity comes at the cost of new issues in *programmability* and *runtime management* of these resources to achieve the pursued performance/power consumption trade-off. In particular, various kind of processing units imply different type of programming languages and models thus introducing new implementation and integration challenges. Nevertheless, this abundance of resources has to be properly managed in the execution of the workload, since each type of processing unit offers a different level of performance/power efficiency to each single application and part of it.

In 2009, Khronos Group, including Apple, ARM, Samsung and many other industrial partners, has defined OpenCL [11], a cross-platform programming model designed around the *Single Instruction Multiple Thread (SIMT)* computational paradigm, to exploit data parallelism on heterogeneous accelerators. OpenCL, that has been implemented as an extension of C/C++ languages, enables the programmability and the usage of a large variety of processing units with a single programming model. However, even if enabling functional portability between different processing units, the OpenCL API still requires the programmer to explicitly select and tune the resources to be used for the execution of the application. This still constitutes a limitation since each application may have different optimal operating points on different platforms, and, also on the same platform, the optimal configuration may also vary on the basis of performance requirements expressed by the user or on the overall working conditions of the board; e.g., the system may be in a low-battery mode. Thus, there is a quest in self-adaptation of OpenCL applications to identify in each working scenario the optimal working point.

In this paper, we present a runtime controller integrated within OpenCL applications running on Linux, which enables the monitoring of system status and the automated adaptation of the application itself[1]. We also propose a novel policy integrated within this controller allowing the application to self-tuning by acting on the mapping and the Dynamic Voltage and Frequency Scaling (DVFS) of the processing units to optimize the performance/power consumption trade-off. Experimental sessions carried out on a widely-used OpenCL benchmark suite, called Polybench [4], show the efficiency of the controller to quickly converge to the optimal solution with less than 10% of error.

The rest of the paper is organized as follows. Section 2 briefly discusses the related work. Then, Section 3 introduces the working scenario and states the addressed optimization problem. The implementation of the proposed integrated runtime controller and of the mapping decision policy are provided in the Sections 4 and 5, respectively. After that, an experimental evaluation of the approach is provided in the subsequent Section 6, and, finally, Section 7 concludes the paper.

*EWiLi'16, October 6th, 2016, Pittsburgh, USA. Copyright retained by the authors.

[1] The source code is publicly available at https://bitbucket.org/necst/opencl-cgroups-library-release

2. RELATED WORK

Many OpenCL runtime supports have been defined by vendors for their designed processing units; examples are Intel for last generations of Pentium, Xeon and HD Graphics units [8], NVIDIA and ARM for GPU devices [14, 1], AMD for their multi-core CPUs, GPUs and Accelerated Processing Units (APUs), and Xilinx for FPGAs [20]. Moreover, other open source runtime supports, such as [3, 10, 9], have been designed to overcome the unavailability of commercial solutions especially for some type of CPUs. Finally, OpenCL ICD loaders (e.g., [19]) have been also implemented to dynamically discover and use at the same time in the same application various runtime supports in computing systems containing devices from different vendors. When considering the mobile and embedded scenario, the main limitation of these OpenCL solutions is the lack of an advanced support to the widely-used ARM big.LITTLE device. In fact, ARM does not provide any runtime for the CPUs [1], while the mentioned open source solutions handle such a multi-core as a single device and spawn threads indistinctly on all the cores. Thus, the presence of two highly-different clusters that could be used separately is neglected.

Approaches for tuning and optimizing OpenCL applications on HSAs have been recently investigated by a number of works, such as [15, 16, 12]. In [15] a design space exploration is performed to identify the most efficient solution in terms of OpenCL kernel tuning (e.g. the workgroup size) and subsequent task mapping. Then, in [16], the authors define another design exploration approach to identify the optimal partition point for the amount of data to be processed by a single OpenCL kernel in order to parallelize the elaborations among CPU and GPU. Finally, in [12] a Domain Specific Language and a companion source-to-source compiler are proposed to generate an OpenCL kernel specifically optimized for a target architecture and at the same time transparently managing parallelization and data transfer among resources. Since all these works are based on design-time activities, they require a specific design optimization for each considered architectural platform and, nevertheless, do not feature runtime controllers able to adapt to changes in the working conditions.

Further works (e.g., [2, 22, 18, 13]) have proposed runtime controllers to perform dynamic resource management. Their goal is to optimize the trade-off between performance and power/energy consumption by adapting to the currently running workload and related execution requirements specified by the user. Unfortunately, none of such frameworks support OpenCL applications. In particular, the approaches in [2, 22] do not provide any mapping mechanism compliant with asymmetric CPUs. The approach in [18] defines a mapping mechanisms based on the Linux `sched_set_affinity()` to support the OpenMP parallelization for CPUs that is not compatible with OpenCL and, at the same time, cannot be extended to GPU. The same mapping mechanism is adopted in [22]. Finally, even if in [13] it is defined another mechanism exploiting Linux *cgroups*, that is more flexible than the previous one, unfortunately again the approach does not consider OpenCL code and therefore it does not support acceleration on GPU. As a conclusion, the goal of this work is to overcome all the discussed limitations and to propose a comprehensive controller supporting the mapping of OpenCL applications on all the types of devices and at the same time featuring a fast runtime decision policy.

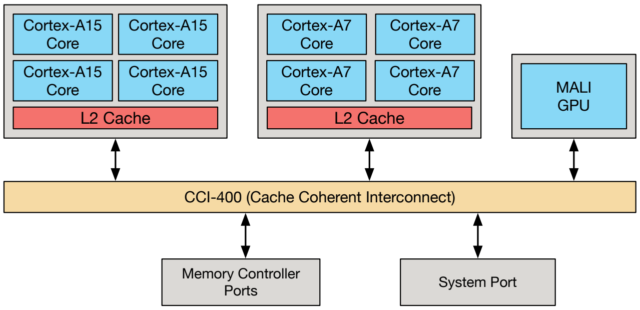


Figure 1: Architecture of the Samsung Exynos 5422.

3. PROBLEM DEFINITION

This section presents the working scenario considered in this paper, by introducing the target architectural platform and the class of executed applications. Then, we formulate the optimization problem addressed in the runtime controller we propose.

3.1 Target Architecture and Applications

In this work we consider an HSA as the Samsung Exynos 5422 [17]. As depicted in Figure 1, this chip features an ARM A15 quad-core cluster (called *big*) and an ARM A7 one (called $LITTLE$). The big cluster, targeted for performance-demanding tasks, can run at frequencies in the ranges from 200 to 2000 MHz, while the LITTLE one, suited for low-power mode, at frequencies in the 200-1300 MHz range. Moreover, the architecture contains an ARM MALI GPU which frequency can be configured in a range between 177 and 600 MHz. DVFS can be used to change the frequency at runtime with a *per-cluster* granularity. Moreover, the chip is provided with power monitoring sensors. Both sensors and actuators are accessible though standard interfaces provided by the loaded Linux OS. As a final note, our solution is valid for any alternative HSAs with a similar architecture and running a Linux OS.

Regarding the target applications, a set of computational kernels that is of great interest in the context of embedded and mobile systems is the family of polyhedral applications. Polyhedral applications comprise algorithms for video processing, filters and algebraic transformations which are at the basis of control, infotainment and augmented reality software. These applications are characterized by a computational intensive kernel continuously executed in a loop on incoming data, such as the frames in a video to be processed. In this work we considered the OpenCL implementation of Polybench [4] benchmark suite as a representative set of polyhedral applications.

3.2 Problem Formulation

The addressed problem consists in the identification of the best operating point in terms of Performance per Watt for a single given application running on the considered architecture. In particular, we want to identify at runtime, without previous profiling information, which is the best processing unit to use and the related frequency level.

In a more formal way, let us consider a controlled application A running on the target architecture featuring three different kind of processing units $P = \{BIG, LITTLE, GPU\}$. Each processing unit is characterized by a set of operating frequencies (in MHz), that can be selected at runtime:

$$f_{BIG} = \{200, 300, \ldots, 1900, 2000\}$$
$$f_{LITTLE} = \{200, 300, \ldots, 1200, 1300\}$$
$$f_{GPU} = \{177, 266, 350, 420, 480, 543, 600\}$$

The running application will be characterized for each operating point (defined in terms of the used processing unit and the selected frequency level), by two direct metrics the throughput, Thr, and the overall power consumption W, and a derived metric called the power efficiency EFF:

$$EFF_{p,f} = \frac{Thr_{p,f}}{W_{p,f}}, p \in P, f \in f_p$$

The goal tackled in this work is to find $\hat{p} \in P$ and $\hat{f} \in f_{\hat{p}}$ such that:

$$EFF_{\hat{p},\hat{f}} \geq EFF_{p,f}, \forall p \in P \land \forall f \in f_p$$

In order to solve this optimization problem we need to address a set of technical issues related to the monitoring and controllability of the running application on the target system; specifically, we need i) the support for OpenCL for all the processors with the possibility to constrain and move the execution at runtime; ii) to measure the throughput of one iteration of the application and its power consumption; iii) a smart algorithm to explore the power efficiency curves and rapidly identify the best operating point.

4. CONTROLLER IMPLEMENTATION

The self-adaptive approach proposed in this paper has been implemented in a specific controller C++ class directly instantiated within the application source code. Figure 2 depicts the overall structure of the controller and its integration within the system. Moreover, in order to enable the actuation of the mapping on all OpenCL devices, the application has to be implemented according to a specific template. Listing 1 shows the defined application template and how the controller is instantiated and used. All the details of the controller in Figure 2 are discussed in the following paragraphs.

OpenCL runtime. To enable the support for all the devices available in the Exynos chip, we have installed both ARM OpenCL Mali SDK [1] and Portable OpenCL library [9], and we have enabled the concurrent discovery of both the platforms with the OpenCL ICD Loader provided by [19].

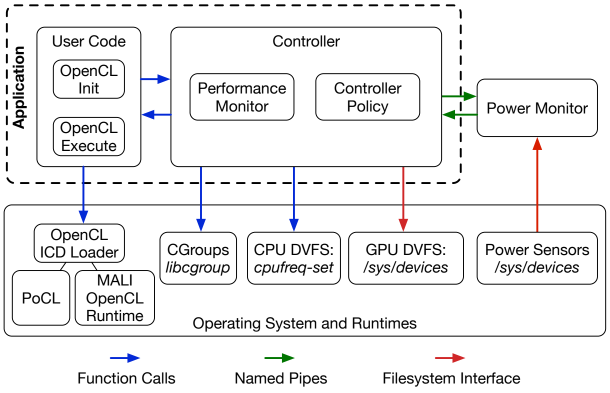


Figure 2: Overview of the implemented system.

OpenCL template. The defined application structure is shown in Listing 1; it slightly enhances the standard OpenCL template. In particular, the latter one requires the programmer to select and configure the desired platform and device to be used to execute the kernel. We extend this template to allow the defined controller to dynamically select at each iteration of the application which device to use.

To enable this capability, in the OpenCL initialization step, all platforms and devices are discovered and set up, as shown in the piece of code reporting the `cl_init()` function (Lines 8-19). In particular, the function iterates on all platforms and sets up all devices[2] and related OpenCL objects, such as the context, the memory objects and the program objects; all these objects are stored in arrays.

Then, a specific variable, `curr_device`, is used to specify the index of the current device to be used in the execution of the kernel. Thus, the `run_kernel()` function uses the execution context specified in such a variable to run the kernel (Lines 21-38).

Cgroups actuation. In order to enable cluster-level mapping on the big.LITTLE CPU, we have exploited OS facilities for task mapping to force the usage of a subset of the cores. Linux OS provides two different mechanisms: `sched_set_affinity()` and *cgroups*. `sched_set_affinity()` cannot be used in OpenCL applications, since it needs to know thread IDs; indeed, threads are generated within the OpenCL runtime and their IDs are not visible externally. Instead, *cgroups* offers the possibility to assign a set of cores, and, more in general, further resources such as CPU quota and memory amount, by specifying only the application PID; all the threads spawned by that PID are then managed automatically. Therefore, as in [13] we have integrated *cgroups* in the proposed controller.

Performance monitor. Instruction per Cycle (IPC) or other classical low-level metrics computed by the OS do not represent a useful information to the final user to perceive the actual progress of an application. As an example, IPC is not able to show if the video application in execution is providing a minimum Quality of Service (QoS) in terms of frame/s. Therefore, to enable run-time performance monitoring, we have integrated in the controller the Heartbeat mechanism [6], a state-of-the-art solution to acquire high-level information from the application.

The basic idea of the Heartbeat mechanism is to measure the throughput of periodic application by i) measuring the duration of the execution of each single loop of the application and ii) computing the ratio between the amount of processed data and such a duration. Therefore, the controller initialized all necessary data structures (timers and accumulators) during the initialization (Line 47). Then, at the end of the loop (Line 52), the invoked `send_heartbeat()` function collects the new timestamp and the size of the processed data (directly specified by the programmer) and based on such information computes the current throughput.

Power monitor. The considered Exynos chip integrated various sensors to monitor the status of the hardware platform, such as power consumption and temperature of the various clusters. Such sensors are exposed to the programmer through the virtual file systems of Linux OS.

In order to trace the power consumption of the big and LITTLE clusters and of the GPU, we have implemented

[2] For the sake of space, in the listing at Line 14 it is assumed to have a single device per platform.

Listing 1: Application template

```
1  //OpenCL objects
2  cl_platform_id platform_ids[MAX_PLATFORMS];
3  cl_device_id device_ids[MAX_DEVICES];
4  ...
5  cl_uint num_platforms;
6  cl_uint num_devices;
7
8  void cl_init(){
9    int i;
10   //setup OpenCL environment for all devices
11   clGetPlatformIDs(0, NULL, &num_platforms);
12   clGetPlatformIDs(num_platforms, platform_ids,
          NULL);
13   for(i=0; i<num_platforms; i++){
14     clGetDeviceIDs(platform_ids[i],
            CL_DEVICE_TYPE_ALL, 1, &device_ids[i],
             &num_devices);
15     //setup other OpenCL objects for device[i]
16     //i.e. context, queues, memory, programs
17     ...
18   }
19 }
20
21 void run_kernel(int currDevice){
22   //load application data
23   ...
24   //setup the workgroup sizes
25   localWorkSize[0] = ...
26   globalWorkSize[0] = ...
27   //write memory objects
28   clEnqueueWriteBuffer(cmdQueue[currDevice],
          mem_obj[currDevice], CL_TRUE, ...);
29   ...
30   // Set the arguments of the kernel
31   clSetKernelArg(clKernel[currDevice], 0,
          sizeof(cl_mem), (void *)&mem_obj[
          currDevice]);
32   ...
33   //execute kernel
34   clEnqueueNDRangeKernel(cmqQueue[currDevice],
          clKernel[currDevice], 2, NULL,
          globalWorkSize, localWorkSize, 0, NULL,
          NULL);
35   clFinish(cmdQueue[currDevice]);
36   //read memory objects
37   clEnqueueReadBuffer(cmdQueue[currDevice],
          mem_obj2[currDevice], CL_TRUE, ...);
38 }
39
40 int main(){
41   int curr_device, i;
42   Controller controller;
43   //applications variables and objects
44   ...
45   cl_init();
46   //setup CGroup, Heartbeat and policy
47   controller.init();
48   //application's main loop
49   for(int i=0; i<iterations; i++){
50     curr_device = controller.get_curr_config();
51     run_kernel(curr_device);
52     controller.send_heartbeat(DATA_SIZE);
53   }
54   //delete all objects
55   controller.destroy();
56   ...
57 }
```

an external monitor acting as a separate process daemon and periodically (i.e. every 50ms) collecting power values from the interface provided by the `sys` virtual file system of Linux OS. The external monitor can be triggered via a message over a *named pipe* (a Linux interprocess communication mechanism) and instructed to collect power information aggregating them over a period of time. Another message on the same pipe can stop the acquisition returning the average power consumption over the considered period.

DVFS actuation. The setting of the current frequency level for each CPU cluster or the GPU is performed by means of the interface provided by Linux OS through the `sys` virtual file system or by using the `cpufreq-set` utility.

Controller. The controller has been implemented in a single class encapsulating all the discussed mechanism and the decision policy. It exposes the following methods:

- `init()`, invoked at Line 47, sets up the environment by initializing the data structures of the Heartbeat, *cgroups* and of the decision policy; moreover the function connects to the external power monitor by means of the named pipe.

- `get_curr_config()`, invoked at Line 50, analyzes all collected metrics (power and throughput) and executes the decision policy to identify on which device to run the application kernel during the current loop iteration; moreover, it actuates on the *cgroup* library.

- `send_heartbeat()`, invoked at Line 52, processes current Heartbeat at the end of the main loop to compute the throughput.

- `destroy()`, invoked at Line 55, deallocates all data structures.

5. CONTROLLER POLICY

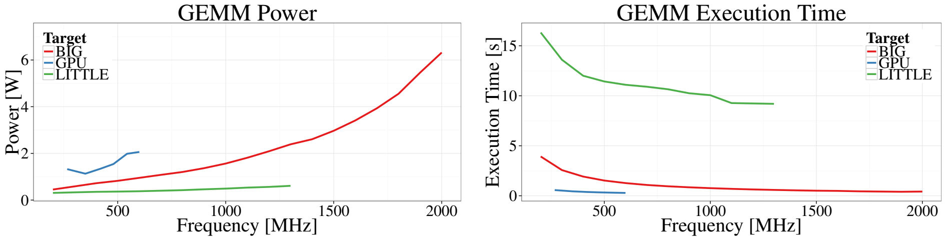
This section describes the controller policy that allows to solve the problem tackled in this paper, defined in Section 3. We will first discuss a preliminary profiling phase carried out on the target applications in order to understand which control strategy should be adopted, and, then, we will describe the policy itself.

5.1 Preliminary Analysis

In a preliminary phase, we carried out an experimental evaluation of the behavior of such applications on the considered heterogeneous platform. We measured the execution time, power consumption and power efficiency of each considered application on each processing unit (big, LITTLE and GPU) at each available frequency level.

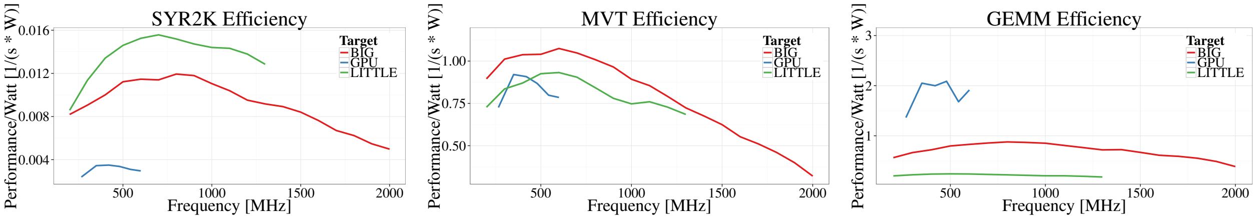
During this analysis, we noticed that the power profile is almost the same for all the considered applications presenting a quadratic relation with respect to the frequency, as shown for GEMM application in Figure 3(a). In the same way the execution time of the applications follows the same trend for all the benchmarks where an increase in the frequency level turns into an improvement of the execution time up until a certain value, as shown in Figure 3(b) for the GEMM application.

Combining these two curves we found how the power efficiency of the benchmarks varies (Figure 4). Also in this case, the trend of the curves is similar for all the benchmarks. An interesting aspect is that, depending on the actual values of the execution time and power consumption, three different situations can be found where either the big, the LITTLE, or the GPU outperforms the other units (as shown in Figure 4). Finally, all these curves present a maximum which is located around the middle of the frequency range for all architectures. The goal of the policy is then to find at runtime this maximum without any previous profiling information.



(a) Power consumption (b) Throughput

Figure 3: Typical performance/power figure of the Polybench applications.



(a) LITTLE cluster is the most efficient. (b) big cluster is the most efficient. (c) GPU device is the most efficient.

Figure 4: Power efficiency of three different Polybench applications.

5.2 Policy Definition

The controller policy is triggered each time the user code requests a configuration to use (i.e. at the beginning of each iteration of the application loop).

Considering the structure of the power efficiency curves, if we fix a device, we can use a ternary search algorithm to solve our optimization problem. In fact, this algorithm allows to find a maximum of a mathematical continuous function F (with a single maximum point), and, actually, the curves characterizing the computational efficiency of the considered applications (refer to Figure 4) present such a shape.

The algorithm needs to find three initial points $a, b, c \mid a < b < c \wedge F(b) \geq F(a) \wedge F(b) \geq F(c)$; to do this we execute the first three iterations of the application loop on a given device at its minimum, maximum and middle frequencies. Once the three initial points are evaluated, the search space is divided in two parts: the interval $[a, b]$, and the interval $[b, c]$, where the point b represents the current estimate for the configuration having the best performance/Watt ratio. During an iteration step a new candidate is selected from one of the two intervals. Our implementation of this selection algorithm picks as candidate point the intermediate frequency of the interval and it gives priority to the interval $[a, b]$ until it converges and then selects the new candidate from the interval $[b, c]$.

Figures 5 and 6 illustrate a typical step of the algorithm. The new candidate frequency, x in the figure, is selected and used to execute the next execution of the application. Once the execution terminates the policy knows, the value $f(x)$ for the candidate point. Figure 5 (a) illustrates the case where x is chosen from $[a, b]$, while Figure 6 (a) shows when x is selected from $[b, c]$. Once the performance/Watt value of the new point is measured, the selection of the a', b', c' points for the next iteration is done by selecting from the points a, b, c, x the point with the maximum performance/Watt as b' and the points to its left and right (when sorted based on frequency) as a' and c'. Figures 5 (b) and 6 (b) illustrate the selection of the new a', b', c' points from the 4 available. Both the figures assume that b is still the best estimate of the frequency attaining the best performance/Watt ration and for this reason the points next to it are selected.

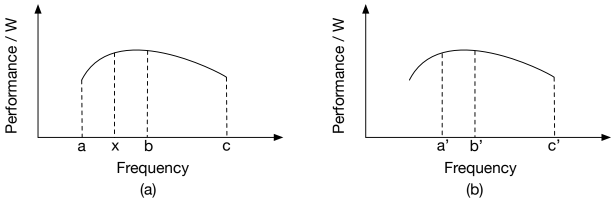


Figure 5: Ternary search: when the new point x is selected in $[a,\ b]$ (a) and corresponding outcome (b).

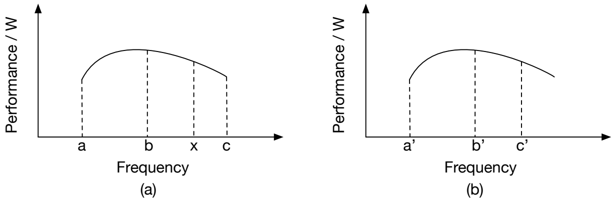


Figure 6: Ternary search: when the new point x is selected in $[b,\ c]$ (a) and corresponding outcome (b).

The iterative execution continues until the points a, b, c are contiguous and there is no possibility to select a new point in the two intervals. When the algorithm converges, the point b represents the frequency that allows to attain the best performance/Watt on the device currently analyzed.

The ternary search algorithm has been implemented in

Listing 2: Selection of next configuration

```
1  Configuration Controller::getConfiguration(){
2    for(auto d : devicesList){
3      if(!d->hasConverged())
4        return d->getNextConfiguration();
5    }
6
7    std::vector<std::pair<float, int>>
         bestEfficiency;
8
9    for(int i=0; i<devicesList.size(); i++){
10     float xMax, yMax;
11     xMax = devicesList[i]->maxEstimation;
12     yMax = devicesList[i]->
           getEstimationAtFrequency(xMax);
13     bestEfficiency.push_back( std::pair<float,
           int>(yMax, i) );
14   }
15
16   std::sort(bestEfficiency.begin(),
         bestEfficiency.end());
17   std::pair<float, int> bestDevice =
         bestEfficiency[bestEfficiency.size()-1];
18
19   return devicesList[bestDevice.second]->
         getNextConfiguration();
20 }
```

a function called `getNextConfiguration()` that is used to pick for a given device the frequency to use. This function is then used in the `getConfiguration()` routine (Listing 2) that decides which device to use to execute the application kernel alongside its frequency. In the `getConfiguration()` function, the controller checks for all the available devices (`devicesList`) whether there exists an estimation for the best efficiency. If such estimation does not exist the controller uses the ternary search on the device to find the maximum as explained before (Line 2). When all the estimations are available, the algorithm selects the device with the best power efficiency (Lines 9-14) and asks that device for the next configuration (Line 16).

The whole configuration decision process is invoked when the application code calls the `get_curr_config()` (Line 50 in Listing 1). The code of this function, which bridges the application code with the controller policy is represented in Listing 3. The API calls the `get_curr_config()` function and then invokes the function to set the desired configuration: only the frequency for the GPU, both frequency and *cgroups* configuration for the CPU. Finally, it returns to the user the device id to use when invoking the OpenCL kernel, as explained in the application template section.

6. EXPERIMENTAL RESULTS

This section illustrates the results of our controller. Experiments have been conducted on an Odroid XU3 board [5] hosting an Exynos 5420 chip. We used a set of 7 applications from the Polyhedral benchmark suite [4], namely 3DCONV, 3MM, ATAX, BICG, GEMM, MVT, SYR2K. All these applications have been extended with the inclusion of the controller proposed in this work.

In order to test the performance of our controller, we modified each application to execute the computational kernel for a fixed number of 50 iterations to see if the controller was able to converge to the most power efficient solution possible. The best configuration for power efficiency has been identified with an offline profiling phase which performed an exhaustive exploration of all the possible configurations.

Listing 3: Code of the get_curr_config() API

```
unsigned int OpenClController::get_curr_config
    (){
  this->currentConfiguration = getConfiguration
      ();

  // Set frequency using either cpufreq-set for
       CPU or using filesystem for GPU
  // For CPU devices also calls the proper
       cgroups functions
  this->currentConfiguration.d->
       setConfiguration(this->
       currentConfiguration.frequency);

  // Return the device id to use in the
       application
  return this->currentConfiguration.device->
       openCl_deviceId;
}
```

Figure 7 illustrates the comparison among the profiled power efficiency and the one found at runtime by our controller. Note that the power measures are subject to a little variability due to different working condition (i.e. background system processes, out of our control). As the figure shows, the runtime power efficiency adheres for all the benchmarks to the best power efficiency found during the profiling phase. In particular, for the first five benchmarks (from 3DCONV to GEMM) we have that the GPU has the best power efficiency and the controller is able to converge and use the same device at runtime. In the last two benchmarks we have that MVT has the best efficiency on the BIG processors while SYR2K[3] is optimal on the LITTLE ones; nonetheless the controller found also in these situations a solution near to the optimum. More in details the controller reaches a power efficiency which is at least 90% of the best value found in profiling. The reason behind the 10% error is due to runtime measurements variability. This causes the policy to converge to a frequency that is near to the optimal one; e.g. GEMM converges to 350 instead of 480 MHz.

In all these experiments the time needed to execute our policy is included in the execution time of the kernel iteration and concurs in defining the power efficiency of the application. Since all the controller functionalities are invoked also when the policy has converged, we can state that the overhead introduced by our solution is negligible.

Concerning the convergence time we have that the controller converges by testing less than 20 different configurations; this represents about 50% of the overall design space (that is composed of 38 configurations) as shows in Table 1. At the opposite, the offline profiling strategy has always to explore all the 38 configurations. Furthermore, we have to keep in mind that the offline profiling requires that the working conditions are exactly the same at the moment the application is in execution. At the opposite, runtime adaptation allows to converge also when the working conditions change.

7. CONCLUSIONS AND FUTURE WORK

This paper has presented a novel runtime controller integrated within an OpenCL application. The controller fea-

[3] All the SYR2K power efficiency values have been multiplied by 100 for sake of clarity.

Table 1: Time to converge for the applications.

Benchmarks	Iterations
ATAX	20
BICG	19
3DCONV	17
GEMM	19
3MM	19
MVT	18
SYR2K	20

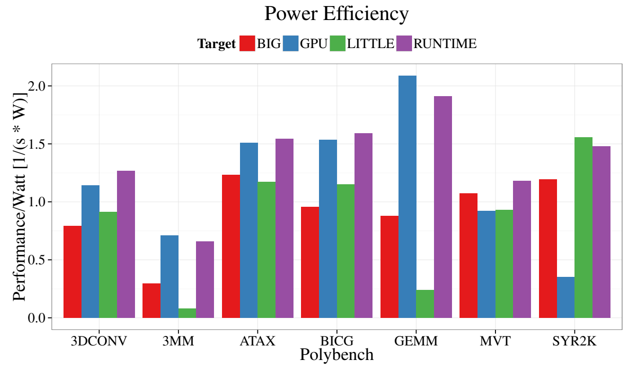


Figure 7: Comparison of the solution found by our controller with profiled information.

tures a novel policy allowing the application to autonomously adapt by acting on the mapping and the DVFS of the processing units to optimize the performance/power consumption trade-off. Experimental results have demonstrated the efficiency of the controller to quickly converge to the optimal solution with less than 10% of error. Future work deals with the adoption of further actuation knobs for resource usage, such as quota assignment and finer-grained mapping, the improvement of the proposed policy and controller to support the concurrent execution of several applications.

Acknowledgments

This work has been partially funded by the EU FP7 SAVE project (#610996-SAVE).

8. REFERENCES

[1] ARM. Mali OpenCL SDK. https://developer.arm.com/products/software/mali-sdks/mali-opencl-sdk.

[2] C. Bolchini, G. C. Durelli, A. Miele, G. Pallotta, and M. D. Santambrogio. An orchestrated approach to efficiently manage resources in heterogeneous system architectures. In *Intl. Conf. on Computer Design*, pages 200–207, 2015.

[3] R. Brochard and N. Nikolaev. FreeOCL. https://forge.imag.fr/projects/ocl-icd.

[4] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Proc. of Innovative Parallel Computing*, pages 1–10, 2012.

[5] Hardkernel co. Odroid XU3. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127.

[6] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats for software performance and health. *ACM Sigplan Notices*, 45(5):347–348, 2010.

[7] HSA Foundation. http://www.hsafoundation.com/, 2015.

[8] Intel. SDK for OpenCL Applications. https://software.intel.com/en-us/intel-opencl.

[9] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.

[10] G. Jo, W. J. Jeon, W. Jung, G. Taft, and J. Lee. OpenCL Framework for ARM Processors with NEON Support. In *Proc. of Workshop on Programming Models for SIMD/Vector Processing*, 2014.

[11] Khronos Group. OpenCL. https://www.khronos.org/opencl/, 2016.

[12] T. Lepley, P. Paulin, and E. Flamand. A novel compilation approach for image processing graphs on a many-core platform with explicitly managed memory. In *Proc. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10, 2013.

[13] S. Libutti, G. Massari, and W. Fornaciari. Co-scheduling tasks on multi-core heterogeneous systems: An energy-aware perspective. *IET Computers Digital Techniques*, 10(2):77–84, 2016.

[14] NVIDIA. OpenCL. https://developer.nvidia.com/opencl.

[15] E. Paone, F. Robino, G. Palermo, V. Zaccaria, I. Sander, and C. Silvano. Customization of OpenCL Applications for Efficient Task Mapping Under Heterogeneous Platform Constraints. In *Proc. Conf. on Design, Automation & Test in Europe (DATE)*, pages 736–741, 2015.

[16] A. Prakash, S. Wang, A. E. Irimiea, and T. Mitra. Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms. In *Proc. Int. Conf. on Computer Design (ICCD)*, pages 208–215, 2015.

[17] Samsung. Exynos 5 Octa. http://www.samsung.com/global/business/semiconductor/product/application/detail?productId=7978&iaId=2341.

[18] E. D. Sozzo, G. C. Durelli, E. M. G. Trainiti, A. Miele, M. D. Santambrogio, and C. Bolchini. Workload-aware power optimization strategy for asymmetric multiprocessors. In *Proc. of Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 531–534, 2016.

[19] B. Videau and V. Danjean. OpenCL ICD Loader. https://forge.imag.fr/projects/ocl-icd.

[20] Xilinx. SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[21] Xilinx. Zynq-700 All Programmable SoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[22] J. Yun, J. Park, and W. Baek. HARS: A Heterogeneity-aware Runtime System for Self-adaptive Multithreaded Applications. In *Proc. of Design Automation Conference (DAC)*, pages 107:1–107:6, 2015.