Managing Heterogeneous Resources in HPC Systems

Giovanni Agosta
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Italy
giovanni.agosta@polimi.it

William Fornaciari
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Italy
william.fornaciari@polimi.it

Giuseppe Massari
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Italy
giuseppe.massari@polimi.it

Anna Pupykina
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Italy
anna.pupykina@polimi.it

Federico Reghenzani
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Italy
federico.reghenzani@polimi.it

Michele Zanella
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Italy
michele.zanella@polimi.it

ABSTRACT

To sustain performance while facing always tighter power and energy envelopes, High Performance Computing (HPC) is increasingly leveraging heterogeneous architectures. This poses new challenges: to efficiently exploit the available resources, both in terms of hardware and energy, resource management must support a wide range of different heterogeneous devices and programming models that target different application domains. We present a strategy for resource management and programming model support for heterogeneous accelerators for HPC systems with requirements targeting performance, power and predictability. We show how resource management can, in addition to allowing multiple applications to share a set of resources, reduce the burden on the application developer and improve the efficiency of resource allocation.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; **Concurrent programming languages**; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering** → *Process management*;

KEYWORDS

Parallel programming models, Resource Management, Heterogeneous Architectures, High Performance Computing.

ACM Reference format:
Giovanni Agosta, William Fornaciari, Giuseppe Massari, Anna Pupykina, Federico Reghenzani, and Michele Zanella. 2018. Managing Heterogeneous Resources in HPC Systems. In *Proceedings of 9th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and 7th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms, Manchester, United Kingdom, January 23, 2018 (PARMA-DITAM),* 6 pages.
https://doi.org/10.1145/3183767.3183769

1 INTRODUCTION

High-Performance Computing is quickly evolving at the hardware, software and application levels. From the hardware point of view, heterogeneity is emerging as a dominant trend for pure performance and, even more, for performance per watt, as shown by the dominance of such architectures in the Green 500 [1] and Top 500 [2] lists. From the application point of view, new classes of applications are emerging, as HPC is now seen as a valuable tool beyond the traditional application domains of oil & gas, finance, and meteorology and scientific computation. From the software point of view, the push towards cloud HPC [9] follows the hardware and application trends, aiming at providing computational resources to classes of users which could not afford them in the past. In this context, applications that are time-critical, such as financial analytics, online video transcoding, and medical imaging require predictable performance, which are at odds with the need to maximize resource usage while minimizing power consumption.

Extending the traditional optimization space, the MANGO project [4] [5] aims at addressing what we call the PPP space: *power*, *performance*, and *predictability*. In this scenario, the objective of MANGO is to achieve extreme resource efficiency in future QoS-sensitive HPC through an ambitious cross-boundary architecture exploration. The MANGO project investigates the architectural implications of the emerging requirements of HPC applications, aiming at the definition of new generation high-performance, power-efficient, deeply heterogeneous architectures with native mechanisms for isolation and QoS compliance. MANGO follows a disruptive approach that challenges several basic assumptions, and explores new many-core architectures specifically targeted at HPC. In particular, it focuses on deeply heterogeneous architectures, where multiple accelerators coexist and can cooperate for a single application composed of multiple kernels, or be partitioned among different applications.

To achieve resource efficiency when dealing with Quality of Service-sensitive HPC applications, resources cannot be fine-grain managed by the application developer. This would in fact lead to over-allocation of resources, thus limiting the ability of the system to balance between different applications or serve high QoS applications, even in scenarios where such applications would be

[1] https://www.top500.org/green500/lists/2016/11/
[2] https://www.top500.org/lists/2016/11/

compatible with the current availability of resources. We now highlight the *technical and scientific challenges* that will need to be tackled at the level of the runtime software support as heterogeneity increases its presence in the HPC field, along with the outline of the proposed solutions.

Resource Management. The biggest challenge for heterogeneous resource management is to optimize resource allocation while taking into account that: a) each application may be composed by multiple tasks, each of them possibly having data and timing dependencies with the other ones; b) executing a task on different computing units of an heterogeneous architecture would lead to different throughput, QoS, and power/energy consumption; c) especially in case of data dependencies, the performance of an application depends not only on where its tasks are executed, but also on where the data of its task is located in the system; and d) requirements coming from each application (usually throughput and QoS) must be complied with, while also addressing the system-wide (power/thermal/energy) requirements. To address this very complex problem, we aim at providing managed applications with a resource-agnostic view of the available resources. Application developers will focus only on what has to be done (optimize the tasks implementation and describe the inter-task dependencies) and how it should be done (define throughput/QoS requirements but also provide the resource manager with some meta-data about the tasks that will be executed), while the resource manager, which has a system-wide view of the available resources and the current workload, must optimally allocate the tasks and their data while making both applications and hardware comply with their requirements. Taking into account inter-task data dependencies is of paramount importance: tasks executing on different parts of an heterogeneous architecture want their data to be close to all of them, thus minimizing communication overheads; hence, memory management is indeed tightly coupled with resource management. How proposal includes a tight cooperation between resource and memory managers, so that memory management will serve memory requests in a resource allocation-aware fashion.

Programming Model Support. Here the challenge is to support programming models across a wide range of different accelerators. The proposed technique is to adopt an intermediate runtime support that exposes basic features which, while per se not sufficient for the application programmer needs, easily map on the hardware features which are common to all the accelerators (i.e., those provided by the communication architecture). The intermediate runtime support must expose basic services for communication, synchronization and task spawning. Higher level programming models will be then built over the intermediate model, and each accelerator will also be able to expose its own specific primitives, thus providing the programmer with the features that are needed to achieve performance.

To assess the impact of resource management in heterogeneous HPC systems, we compared our design to an industry-grade OpenCL, showing the overhead imposed by the two solutions, as well as the scalability of our design.

The rest of this paper is organized as follows. In Sections 3 and 4 we respectively show how we address the two challenges outlined above. In Section 5 we assess the impact of resource management on the execution time of the host side code. Finally, in Section 6 we draw some conclusions and provide a roadmap for future development.

2 RELATED WORKS

In case of heterogeneous platforms, programming models are essential to abstract from the increasing architecture complexity. *OpenMP* [1] is a well-known example of shared memory programming model, which supports, from version 4.0, the option to offload parallel regions to accelerators such as GPGPUs. *OpenCL* is a framework for programming parallel heterogeneous platforms [7], supporting data parallelism and, to some extent, also task parallelism. The architecture of the typical OpenCL platform is composed by a *host* and one or more computing *devices*. With respect to proprietary models such as CUDA [12, 13], OpenCL does not impose predefined limits on the number of executors. Instead, it relies on a platform introspection API that allows such limits to be retrieved at runtime. This allows OpenCL to support computing devices from multiple vendors and attached to the same host. *OpenACC, SYCL and C++ AMP* [8, 11, 16] provide similar features, in some cases even emitting OpenCL or CUDA code as a back-end. They attempt to provide easier-to-use interfaces by leveraging either directives or C++ features. None of these programming models provides automated resource management, which is instead our key distinguishing feature. In this regard, several resource management systems are worth considering. The *StarPU* [2] framework has been specifically designed to address the problem of scheduling tasks on systems featuring heterogeneous processing devices. The allocation policy that is used at runtime can be selected among a set of pre-defined ones. StarPU features a greedy policy that aims at balancing the workload and the energy consumption over all processing devices. However, maximizing the energy efficiency of a workload while complying with the QoS requirements of each application is not considered as an explicit objective. The *Simple Linux Utility for Resource Management* (SLURM), is another widespread resource manager [6]. It is modular, scalable and includes several greedy scheduling policies. However, from the hardware perspective, it focuses on Linux cluster systems. Power management can be performed by switching-off idle computing resources or explicitly setting the CPU frequencies according to the job input. Overall, this resource manager enforces coarse-grained decisions. Moreover, to the best of our knowledge, it does not take into account the possibility of controlling the bandwidth allocation of a Network-on-Chip (NoC) in a many-core processor; hence, its capability of isolating applications is limited in the context of a multi-accelerator system.

3 PROGRAMMING MODEL SUPPORT

The goal of a parallel programming model for heterogeneous architectures is to allow programmers to easily develop applications that target different types of accelerator architectures. In particular, we aim at supporting three types of accelerators: *Symmetric Multiprocessors (SMP)*, which are characterized by good capabilities in terms of OS support and execution flexibility; *Single Instruction Multiple Data (SIMD) accelerators*, which are programmable but not able to run a complex runtime; and *hardware accelerators*, which do not need or support any kind of software runtime.

Applications, on the other hand, may be developed either by domain experts with limited knowledge of parallel computing and programming models, or by more experienced programmers. Thus the programming model needs to be simple enough for non-specialists to understand, should allow functional portability across a wide range of different platforms, and should integrate automated resource management. To this aim, the programming model support must include at least two runtime layers, one at the *host-side* and the other at the *device-side*. In this section, we first introduce our proposed programming model runtime support, the *MANGO Application Library (libmango)*. Then we discuss the key differences with respect to existing industry standards such as OpenCL.

3.1 Host-side low-level runtime

The *host-side low-level runtime* (HLR) provides an interface to access the functionality of the accelerators, from code running on a, typically CPU-based, general purpose node (GN).

Listing 1: Example of HLR use: a FIFO communication is set up between the host and the single kernel, which is loaded from an external file.

```
/* Initialization */
hlr_init("app_name", "app_recipe");
kernelfunction *k = hlr_kernelfunction_init();
hlr_load_kernel("./test_fifo", k, ACC_TYPE, BINARY);

/* Registration of task graph */
hlr_kernel_t *k1 = hlr_register_kernel(KID, k);
hlr_buffer_t *b1 = hlr_register_memory(BID, FIFO, 1, 1, k1, k1);
hlr_task_graph_t *tg = hlr_task_graph_create(1,1,0,k1,b1);

/* Resource Allocation */
hlr_resource_allocation(tg);

/* Execution setup */
hlr_arg_t arg1 = { (void*)b1->phy_addr, sizeof(uint64_t), FIFO };
hlr_arg_t arg2 = { (void*)((uint64_t)b1->event->eid),
                   sizeof(uint32_t), SCALAR };
hlr_args_t *args = hlr_set_args(k1, 2, &arg1, &arg2);

/* Data transfer and kernel execution */
hlr_write(argv[1], b1, BURST,strlen(argv[1]));
hlr_event_t *e2 = hlr_start_kernel(k1, args, NULL);
hlr_wait(e2);

/* Deallocation and teardown */
hlr_resource_deallocation(tg);
hlr_task_graph_destroy_all(tg);
hlr_release();
```

Kernel loading and launching. The HLR exposes functionalities and data structures that can be used to represent and manipulate kernels. Kernels are stored either in memory or external files and are processed in a unit-specific way – source code for a GPGPU-like accelerator would be compiled, whereas a hardware accelerator typically executes a fixed kernel. The HLR also provides developers with an interface to set the arguments of kernels and to trigger their execution. The programmer may specify more than one implementation of each kernel – specifying for each implementation the target accelerator. The resource manager will then use the available options to optimize the mapping of kernels to accelerator units based on its own global view of the system, as well as on kernel QoS requirements, which are specified as part of a *recipe*. A more in-depth discussion of recipes will be provided in Section 4 within the presentation of the runtime resource management approach.

Task graph management. The HLR API allows developers to indicate to the runtime which components (kernels, memory objects, and synchronization events) need to be allocated within the heterogeneous node. These components are then connected into a task graph, thus providing the resource manager with the information needed to generate the best feasible resource allocation for the requested QoS. The task graph represents data and control dependencies among the components, and provides the resource manager with a picture of the application behaviour.

Communication and synchronization. Finally, the HLR provides developers with functions to synchronize with the executing kernels (wait for completion) and for communication purposes. In particular, it supports two forms of asynchronous communication: a simple copy of memory objects, and a burst copy through a FIFO memory buffer.

The code shown in Listing 1 shows the use of the HLR API. In this case, a FIFO communication is set up between the host and a single kernel, which is loaded as a binary from an external file. For the sake of simplicity, the kernel does not return data. The execution is not unlike that of a native kernel in OpenCL, but the resource assignment is controlled by the resource manager, which is therefore able to optimize the use of resources in a multi-application scenario.

3.2 Device-side low-level runtime

The *device-side low-level runtime* support (DLR) serves as a baseline to implement more complex programming models. It only implements the minimal functionality needed to work with the heterogeneous node (HN), featuring one or more accelerators, without a full operating system layer. The DLR allows developers to spawn tasks from the host-side and wait for their completion. It also provides synchronization mechanisms at accelerator level, between different accelerators, and between the host and the accelerators. Finally, it allows the device-side to perform memory mapping of buffers that are allocated in the shared memory, generating virtual addresses for them.

Listing 2: Example of DLR use: the DLR API is used to access the shared memory region, which is then read by the parallel tasks that are spawned from the main executor.

```
char *shared_memory;
dlr_event_t *fifo;

void *task(task_args *a){
    int i; char d;
    for(i = 0; i < N; i ++) {
        dlr_barrier(a, fifo);
        d = shared_memory[tid(a)];
        /* do something with the data */
    }
    return dlr_exit(a);
}

#pragma dlr_kernel
void kernel(char *buffer, dlr_event_t ev){
        shared_memory=buffer;
        fifo=&ev;
        dlr_event_t *e = dlr_spawn(&task, SIZE);
        dlr_join(e);
        return;
}
```

The code shown in Listing 2 demonstrates the use of the DLR API. In this case, which matches the HLR code shown in Listing 1, the DLR API is used to access the shared memory region, which is then read by the parallel tasks that are spawned from the main executor. Once more, for the sake of simplicity, the actual operation of the kernel and the generation of results are omitted.

3.3 Discussion

To better explain the choices that drove the selection of the primitives in the our runtime support, we compare them to possible alternatives, namely POSIX and OpenCL.

With respect to the POSIX API, the set of primitives presented above is much more restricted. The reason behind this choice is that communication and synchronization primitives must be homogeneous among the different accelerators, as they all access the same memory, communication and synchronization resources. Moreover, some of the accelerators lack the ability to efficiently support more complex activities, such as context switching or control divergence. Thus, the selected primitives focus on a set of functionality that can easily be supported by all the accelerators.

With respect to the OpenCL API, the set of primitives presented above lacks introspection capabilities; however, it can rely on the resource management support instead. This is a key goal, as there is a widespread reluctance of users towards the manual selection of computational resources. Moreover, with a manual selection there is no way to perform global optimization of the resource usage.

4 RUNTIME MANAGEMENT

Several studies have remarked the opportunities offered by heterogeneous computing platforms [15] to improve energy efficiency in HPC systems. In the case of heterogeneous architectures, resource management must address the complex problem of distributing and scheduling tasks over processing resources, characterized by different architectures, instruction sets, and support for the software layer. Addressing such a problem requires the cooperation of several actors, and, among those, the resource manager plays a key role.

4.1 Hierarchical and Distributed Management

The overall resource management strategy relies on a hierarchical and distributed approach comprising two levels. The Global Resource Manager (GRM) is at the top of the hierarchy, and it runs on a general purpose node, also referred to as the “master node”. The Local Resource Manager (LRM) runs on the managed nodes (“slave nodes”), which can be either general purpose or heterogeneous. Each instance of the LRM is in charge of managing the resources of one slave node, thus acting as a local manager.

To actually launch the application on multiple nodes, the GRM triggers the *Remote Application Launcher* through the runtime support provided by already known distributed parallel programming paradigms, e.g., Message Passing Interface (MPI): an instance of the application is started on the selected node, where the LRM instance will manage the resource assignment.

At slave node level, resource allocation will take into account both static information, which will be collected during an off-line analysis of hardware and applications and will be stored locally on each node; and runtime information, which will be collected from applications and on-chip sensors and sent as a feedback to the LRM. All the runtime information that is relevant to understand the status of the node will also be forwarded to the GRM, which, merging the information coming from all the nodes, will be able to have a system-wide view of the system architecture. This view will be used by the GRM to adapt the application dispatching and the thermal management actions to the real system response. We used the aforementioned *SLURM* and the *BarbequeRTRM* [3] as global and local resource manager, respectively.

4.2 Runtime-manageable applications

From the LRM perspective, a managed application is a collection of tasks, each of which will be allocated a private share of the available system resources. Depending on the support provided by hardware, such a share can be dynamically reconfigured according to the resource management objectives and the application requirements. In such a case, applications running on heterogeneous systems must adapt accordingly.

The BarbequeRTRM has already introduced a support for application runtime adaptivity [10]. We extended this support to handle programming models relying on task-graph based descriptions of the applications. To this purpose, we developed the *Synchronization Library*, which provides suitable mechanisms to synchronize the execution of the application, and its tasks, with the management actions of the BarbequeRTRM daemon. In the approach described in this work, the HLR API represents an abstraction layer on top these mechanisms. More in detail, the launch of an application creates an instance of the *Execution Synchronizer*, a C++ object in charge of: (1) synchronizing the application tasks execution with the runtime-variable resource allocation; (2) profiling the task execution through user-defined metric counters; (3) transparently collecting the task performance/resource usage statistics and forwarding them back to the BarbequeRTRM, which will tune the resource allocation accordingly. As per the API previously described, this object is allocated when the `hlr_init()` function is called. A suitable control thread is then responsible of calling the member functions of the *Execution Synchronizer* according to both application-side events (e.g., *libmango* function calls) and resource manager actions. The control thread is started when the `hlr_resource_allocation()` function is invoked, and it drives the application execution by calling the following *Execution Synchronizer* member functions:

onSetup(): called when a task-graph has been built. It checks the consistency of the provided task-graph, then it forwards it to the resource manager;

onConfigure(): called when the resource allocation changes. The BarbequeRTRM had at this point allocated the buffers on the HN memory by interacting with the *Memory Manager* and it had offloaded the tasks on the assigned units. Once buffers and kernels are ready, the synchronizer must simply wait for the sequence of `hlr_start_kernel()` function calls. A *kernel execution control thread* is spawned to monitor the execution of each kernel;

onRun(): this function is called once all the tasks are running. It waits for the current “step” to complete;

onMonitor(): this function is called after each `onRun()` execution. It forwards application profiling information to the resource manager;

onRelease(): by means of this function, the application explicitly releases the resources. This function also notifies the application termination to the resource manager. It is invoked when the `hlr_resource_deallocation()` function is called.

Figure 1 provides a simplified view of the synchronization mechanisms underlying the application execution flow. The `onRun()` and `onMonitor()` are part of a loop that may occasionally enter also the `onConfigure()` in case of changes performed by the BarbequeRTRM from the resources allocation point of view, e.g., we need to migrate tasks among accelerators. What is worth to remark is that this happens in a transparent manner with respect to the application. The synchronization loop comes to an end when all the application tasks are completed. In such a case the main thread of the application, waiting for the termination of the tasks can resume its execution and eventually terminates as well.

4.2.1 *Design-time Requirements.*

QoS requirements, i.e. the minimum requirements of the task in terms of Quality of Service, can be expressed in terms of minimum resource usage, e.g. “the task needs

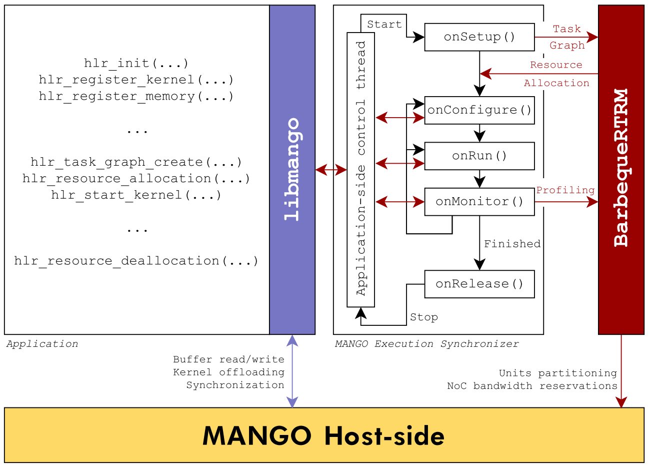

Figure 1: The synchronization mechanisms between the application execution and the BarbequeRTRM control actions.

at least 100 Kbps of network bandwidth", or in terms of throughput. Resource allocation policies that partially or totally base their choices on Design-time profiles take into account this information.

The Design-time profile of a task is contained in an XML file called *recipe*. stored in a local system folder. The typical structure of a recipe is shown in Listing 3. The example describes an application with a static priority level equal to 4. The priority level is used by the resource allocation policy to prioritize the application, (0 is the maximum priority value). The `<platform>` section identifies the target system we are referring to: in this case, a system running a Linux OS (`org.linux.cgroup`), whose architecture is qualified by a specific hardware attribute. The proper section will be parsed at the application start-time depending on the actual system. The `<platform>` section contains one `<awm>` section that defines several attributes: 1) a progressive numeric identifier (`id`); 2) a descriptive name, which is used only for logging and debugging purposes; 3) the aforementioned preference score (`value`); and 4) the profiled configuration time, expressed in milliseconds (`config-time`), which keeps track of the time overhead experienced by the application to enter this resource configuration. The `<resources>` subsections contain the resource requirements. In the example, such requirements are expressed in terms of CPU time quota (`pe` under `cpu`, expressed in percent), amount of memory (`mem`), number of accelerator cores (`pe` under `acc`) and network bandwidth (`net`).

Concerning the specific case of the CPU time, the values reported must be read as percentages. Therefore, values greater than 100 simply expresses the usage requirement of more than one CPU core. Generally, a good practice is to have the recipe generated by suitable profiling or DSE tools.

Then, for multi-tasking applications it is possible to specify the performance requirements at task level. In this regard, the section `<tasks>` comes into play. Here we can define for each task (identified through a suitable identification number), its performance goal in terms of completion time or throughput. Optionally we can also speed-up the learning process of the resource allocation policy through the attribute `hw_prefs` which includes the list of computing units types on which mapping the task, in order of preference.

Listing 3: Example of application Recipe.

```
<?xml version="1.0"?>
<BarbequeRTRM recipe_version=" 0.8 ">
  <application priority="4">
    <platform id="org.linux.cgroup" hw="...">
      <awms>
        <awm id="0" value="1" config-time="150">
          <resources>
            <cpu>
              <pe qty="100"/>
              <mem qty="2" units="MB"/>
            </cpu>
            <net qty="50" units="Kbps"/>
          </resources>
        </awm>
      </awms>
      <tasks>
        <task name="..." id="0" ctime="10" hw_prefs="peak,nup"/>
        <task name="..." id="1" ctime="50" hw_prefs="cpu"/>
        <task name="..." id="2" ctime="30" hw_prefs="nup,peak"/>
      </tasks>
    </platform>
  </application>
</BarbequeRTRM>
```

4.2.2 Runtime Monitoring. The Execution Synchronizer drives the execution of applications by invoking `onSetup`, `onConfigure`, `onRun`, `onMonitor`, `onSuspend` and `onRelease` callbacks, following the proposed execution flow previously described. Meanwhile, it also monitors the execution of applications and takes care of the communication with the resource manager, thanks to the *kernel execution control* threads. Such control threads collect timing information, by capturing events regarding the execution of the tasks (kernels), or the processing of a buffer. This fits well the case of application offloading resident tasks, performing continuous computing on a big set of data for which memory transfers are a typical time consuming activity. In addition, according to application-specific run-time conditions, we may need to redefine the tasks performance goals. But this has been left as a future extension.

4.3 Memory Management

For memory resource management, we support both private and shared buffers, depending on the number of kernels accessing a given buffer according to the task graph. Memory allocation of physically partitioned but globally addressed memory is performed by the resource manager, in accordance to a strategy designed to ensure the availability of the communication bandwidth between memory and processing units, coming from the QoS requirements. The allocation is performed for an entire task graph, to give the resource manager the leeway to co-optimize the allocation for all kernels, with the goal to guarantee QoS for the application, while optimizing the location of remaining free memory and bandwidth to ensure the best management of future requests, based on the knowledge of previous requests [14].

5 EXPERIMENTAL EVALUATION

In this section, we report the results of the experimental campaign. The goal is to assess the overhead imposed by the resource management with respect to that caused by the manual management of heterogeneous resources by the user. To this end, we employed a test application (matrix multiplication), implemented both in our runtime managed programming model and in OpenCL, using manual resource management. To perform a fair comparison, we ran both versions on an Intel Core i7-6700K CPU used as accelerator device. This processor is equipped with 4 physical cores with Hyper-threading support, 8 MB of L3 cache and 16 GB of RAM.

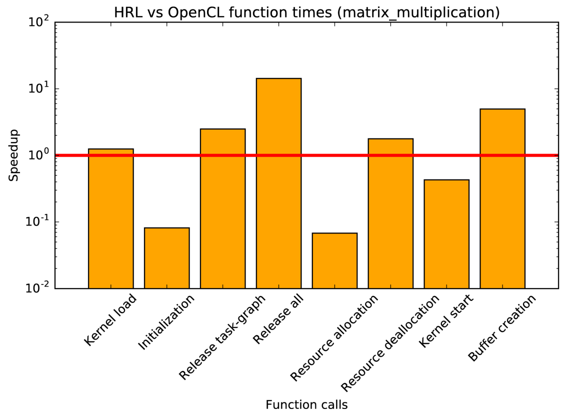

Figure 2: Comparison of resource management APIs

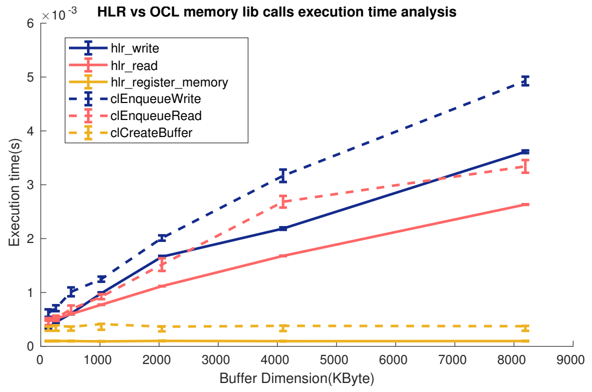

Figure 3: Scalability of memory operations

As can be seen in Figure 2, automated resource management takes over the burden of selecting the appropriate resources at a cost in terms of slowdown in the resource allocation calls. This overhead is due to the `hlr_resource_allocation` call, which takes about 76.4 *ms*. However, this overhead is justified considering that no attempt is made in the OpenCL code to explore different devices – the first available CPU is selected in the test code, thus minimizing the time spent in the allocation phase.

A further investigation aims to evaluate how the resource manager is able to handle a range of requests with different memory sizes. This exploration is run on a synthetic benchmark performing read and write operations on buffers of increasing size. Results are shown in Figure 3, where it can be seen that our runtime compares favourably to the one provided for OpenCL, in terms of both average execution time and its standard deviation.

6 CONCLUSIONS

We have discussed the goals, requirements and solutions for an HPC software stack that targets deeply heterogeneous architectures composed of general purpose nodes and a variety of accelerators. We employed runtime resource management techniques to control the allocation of compute and memory resources to different applications under QoS requirements; and a low-level runtime support to provide a minimum common base among different accelerators, thus allowing functional portability of applications on different computing devices. While our results show that the proposed runtime is competitive with industry standard ones on the host side, further work is needed to provide a full evaluation of the programming model and runtime management system on the actual target heterogeneous accelerator platform.

ACKNOWLEDGEMENTS

This work was supported in part by EU Commission through MANGO project (Horizon 2020 FET-HPC, grant number 671668).

REFERENCES

[1] ARB 2008. *OpenMP Application Program Interface, version 3.0.* ARB. http://www.openmp.org

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.* 23, 2 (Feb. 2011), 187–198. https://doi.org/10.1002/cpe.1631

[3] Patrick Bellasi, Giuseppe Massari, and William Fornaciari. 2015. Effective Runtime Resource Management Using Linux Control Groups with the BarbequeRTRM Framework. *ACM Trans. Embed. Comput. Syst.* 14, 2, Article 39 (March 2015), 17 pages. https://doi.org/10.1145/2658990

[4] José Flich, Giovanni Agosta, Philipp Ampletzer, David Atienza Alonso, Carlo Brandolese, Etienne Cappe, Alessandro Cilardo, Leon Dragić, Alexandre Dray, Alen Duspara, et al. 2017. MANGO: Exploring Manycore Architectures for Next-GeneratiOn HPC Systems. In *2017 Euromicro Conference on Digital System Design (DSD).* 478–485. https://doi.org/10.1109/DSD.2017.51

[5] Jose Flich, Giovanni Agosta, Philipp Ampletzer, David Atienza Alonso, Alessandro Cilardo, William Fornaciari, Mario Kovac, Fabrice Roudet, and Davide Zoni. 2015. The MANGO FET-HPC Project: An Overview. In *Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on.* IEEE, 351–354.

[6] Morris Jette and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *ClusterWorld Conference and Expo.*

[7] Khronos OpenCL Working Group. 2014. The OpenCL Specification, Version 1.2. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf. Aaftab Munshi eds.

[8] Khronos OpenCL Working Group – SYCL subgroup. 2014. SYCL^{TM} Specification, Version 1.2. https://www.khronos.org/registry/sycl/specs/sycl-1.2.pdf. Lee Howes and Maria Rovatsou eds.

[9] Bastian Koller, Nico Struckmann, Jochen Buchholz, and Michael Gienger. 2015. *Towards an Environment to Deliver High Performance Computing to Small and Medium Enterprises.* Springer International Publishing, Cham, 41–50. https://doi.org/10.1007/978-3-319-20340-9_4

[10] G. Massari, E. Paone, P. Bellasi, G. Palermo, V. Zaccaria, W. Fornaciari, and C. Silvano. 2014. Combining application adaptivity and system-wide Resource Management on multi-core platforms. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV).* 26–33. https://doi.org/10.1109/SAMOS.2014.6893191

[11] Microsoft Corporation. 2013. C++ AMP: C++ Accelerated Massive Parallelism, Version 1.2. http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf.

[12] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *ACM Queue* 6, 2 (2008), 40–53.

[13] nVidia Corp. 2008. CUDA Technology. http://www.nvidia.com/CUDA. (September 2008).

[14] A. Pupykina and G. Agosta. 2017. Optimizing Memory Management in Deeply Heterogeneous HPC Accelerators. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW).* 291–300. https://doi.org/10.1109/ICPPW.2017.49

[15] Ehsan Totoni, Babak Behzad, Swapnil Ghike, and Josep Torrellas. 2012. Comparing the Power and Performance of Intel's SCC to State-of-the-art CPUs and GPUs. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS '12).* IEEE Computer Society, Washington, DC, USA, 78–87. https://doi.org/10.1109/ISPASS.2012.6189208

[16] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC: First Experiences with Real-world Applications. In *Proceedings of the 18th International Conference on Parallel Processing (Euro-Par'12).* Springer-Verlag, Berlin, Heidelberg, 859–870. https://doi.org/10.1007/978-3-642-32820-6_85