Procedia Computer Science

Volume 51, 2015, Pages 130–139

ICCS 2015 International Conference On Computational Science

AMA: Asynchronous Management of Accelerators for Task-based Programming Models

Judit Planas[1,2], Rosa M. Badia[1,2,3], Eduard Ayguade[1,2], and Jesus Labarta[1,2]

[1] Barcelona Supercomputing Center (BSC-CNS), Barcelona, Spain
{judit.planas, rosa.m.badia, eduard.ayguade, jesus.labarta}@bsc.es
[2] Universitat Politecnica de Catalunya (UPC), Barcelona, Spain
[3] Artificial Intelligence Research Institute, Spanish National Research Council (IIIA, CSIC), Spain

Abstract

Computational science has benefited in the last years from emerging accelerators that increase the performance of scientific simulations, but using these devices hinders the programming task. This paper presents AMA: a set of optimization techniques to efficiently manage multi-accelerator systems. AMA maximizes the overlap of computation and communication in a blocking-free way. Then, we can use such spare time to do other work while waiting for device operations. Implemented on top of a task-based framework, the experimental evaluation of AMA on a quad-GPU node shows that we reach the performance of a hand-tuned native CUDA code, with the advantage of fully hiding the device management. In addition, we obtain up to more than 2x performance speed-up with respect to the original framework implementation.

Keywords: accelerator management, asynchronous devices, programming models, multi-GPU systems

1 Introduction

Computational science is an interdisciplinary field where computational and numerical techniques are applied to study systems of real-world scientific interest. Such studies are usually done through computer simulation and modelling and allow us to simulate systems that were previously too difficult to study due to its complexity. Moreover, scientists are now able to reproduce or simulate studies that may be too dangerous, take too much time or simply be impossible to reproduce in a laboratory. In the past years, scientific computing has clearly benefited from the advances in computer science, as computers have massively increased their performance, popularity and usability. In this sense, heterogeneous high-performance computers have become a key evolution of regular homogeneous CPU computers due to their computing power. The TOP500 list (Nov 2014) [1] reflects this fact as half of the top 10 machines have either NVIDIA K20x GPUs or Intel Xeon Phi processors, the top 2 being heterogeneous computers.

However, heterogeneous computers make the programming task more difficult, especially for programmers or scientists that code their applications targeting heterogeneous systems (heterogeneous applications). Even in single-node systems, accelerators, offering massively parallel

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.212

hardware, may have their own separated memory space with limited capacity; therefore, programmers and scientists (who may not be expert programmers) need to care about when and which pieces of data are transferred between memory spaces. Data movements and synchronizations become particularly complicated if we want to split the computation between multiple devices and still want to get optimal performance. Several proposals have arisen in the last years to program accelerators, the most important being CUDA [10], which targets NVIDIA GPUs, and OpenCL [7], which works with Intel MIC devices and GPUs as well. However, none of them addresses the aforementioned challenges, since they both expose the underlying hardware to the programmer and only offer a resource management API. Ideally, programming models should be able to hide heterogeneity and hierarchy from the programmer point of view, so that they can focus on their application development and forget about the management of available resources.

In this paper we present AMA (Asynchronous Management of Accelerators), the combination of several optimization ideas that help to efficiently manage and schedule computations to accelerators. Since these devices are asynchronous, we can issue operations such as computations from the host and then do other work on the CPU while waiting for such device operations. Our contributions are a specialized accelerator-oriented work scheduler combined with an asynchronous, non-blocking management design for external devices. Our main target are task-based programming frameworks, where our techniques can improve the management of multi-accelerator systems with a minimized overhead. The objectives of such work are, first, to increase the performance of heterogeneous applications with no effort from the programmer side and, second, to improve the framework by making the accelerator management more efficient.

The paper is organized as follows: Section 2 explains AMA. Section 3 discusses the chosen framework to implement AMA. The implementation is described in Section 4 and its evaluation in Section 5. Related work can be found in Section 6. Section 7 concludes the paper.

2 Asynchronous Management of Accelerators

AMA targets any task-based programming framework with support for asynchronous accelerators such as GPUs. Our techniques can be applied to any asynchronous device supported by the framework. The implementation of such techniques does not require application modifications.

2.1 Event-driven Flow

The most widely-used accelerator programming languages (CUDA, OpenCL) support events and callbacks as a mechanism to manage asynchronous device operations. The AMA design uses these advanced features: each device event is managed from the host side and associated to one accelerator operation (data transfer or kernel launch). The event state will reflect the state of its associated operation. Each event is linked to a list of callbacks that will trigger the actions to be performed once the device operation is finished. Assuming the framework has information about task data, the AMA design lets the runtime prefetch data for several tasks ahead while waiting for the completion of previous tasks. Moreover, AMA supports having several data transfers and kernel executions concurrently, provided that the hardware offers this functionality. In the case of GPUs, two data transfers (one in each direction) can be overlapped with several kernel executions if they are issued in different CUDA streams or OpenCL command queues. This event-driven execution flow can largely speed-up application performance, as the overhead of task management and data transfers is hidden by the execution of other tasks. AMA establishes an additional scheduling policy called *First Transferred - First Run* (FTFR). This policy affects all the tasks targeting the same execution unit that have at least one event pending for completion. In FTFR, as soon as a task has its data transferred to

the device, it will be run, independently of other data transfers or task executions related to other tasks previously assigned. Consequently, the scheduling is now divided into two phases: first, the framework decides *which* processing unit will run the task and, second, FTFR decides *when* the task will be run. As a result of applying the AMA design, host-side threads are never blocked, so they can do other useful work. This gives us an opportunity to make other components smarter and more powerful, even if they increase framework's overhead.

2.2 Additional Runtime Enhancements

The specific modifications will mostly depend on the features of the target framework, but we will focus on those aspects that especially benefit the execution on accelerators. Generally, accelerators consume tasks faster than the host, so it is important to give priority to those tasks that open more parallelism (i.e., tasks with several successors) and also to those in the critical path of the data dependency graph. Thus, we propose to use the CPU spare time to traverse the graph and compute the appropriate priority for each task. In addition, we propose to propagate the priority of a task to its predecessors if they have lower priority, and do so for several levels of predecessors to ensure that the dependencies of higher priority tasks are satisfied as soon as possible. Assuming the target framework offers a data affinity-aware scheduling policy, we propose to compute the affinity information at the latest possible moment, when the task has its dependencies satisfied and is going to be assigned to one of the devices. Making a good affinity decision is very important to optimize the amount of data that is transferred. It may seem obvious to compute task data affinity as late as possible but, since the affinity computation is quite costly, some frameworks may compute this information much before to avoid additional overhead while running tasks. With our approach, such overhead is completely hidden. All these modifications increase the framework overhead, but AMA is able to hide it with the asynchronous execution of tasks and data transfers.

3 The OmpSs Programming Model

The OmpSs programming model [3] combines ideas from OpenMP [11] and StarSs [12]: on the one hand, it enhances OpenMP with support for irregular and asynchronous parallelism and heterogeneous architectures and, on the other, it incorporates StarSs dependence support and data-flow concepts that allow the framework to automatically move data as necessary and perform different kinds of optimizations. OmpSs is able to run applications on clusters of nodes that combine shared memory processors (SMPs) and other external devices, for example, GPU, FPGA and Xeon Phi [4, 5]. Tasks are the basic computation units and the OmpSs model assumes that a single address space exists from the programmer point of view. However, internally, it is able to manage shared data that reside in different memory spaces. It supports heterogeneity, data motion and several implementations of a task (the runtime, based on its own history records, decides which version is more suitable to run every time the task is called [14]).

The OmpSs runtime uses a thread-pool model with a master thread that starts the execution and several worker threads that cooperate executing the work created from task constructs (nesting of constructs is allowed as well). At run time, OmpSs dynamically creates a task data dependency graph to ensure program's correctness. The heterogeneity support handles data movements between system's memory spaces whenever needed. A directory structure is used to trace task data locations and reduce data movements. Task life is divided into five stages (from creation to completion), described in chronological order: *instantiation* (the task and all its related data structures are created, including data dependence computation), *ready* (the task becomes ready when its data dependences are satisfied, task scheduling occurs in this stage), *active* (the task has been scheduled to a system's processing unit; if needed, data is allocated

and transferred), *run* (the task is executed) and *completion* (the task has been run; if needed, output data is transferred and task memory space is freed).

OmpSs offers several task scheduling policies, but we will focus on the data affinity-aware policy. This policy calculates the affinity between tasks and processing units of the system regarding data size and locality. Each processing unit is assigned an affinity score and the task will be scheduled on the unit with the highest score. For performance reasons, this score is computed at task instantiation stage. OmpSs also supports task priorities to establish a task execution order while preserving data dependencies: tasks with higher priority will be executed earlier. Task priority is only propagated to its direct predecessor task due to its overhead.

The flexible design and implementation of OmpSs runtime allows to easily extend any of its features, like adding a scheduling policy or support for a new architecture. Run time configuration arguments are used to set which plugins will be loaded in each application execution.

3.1 OmpSs CUDA Support

At run time, one CPU helper thread is created for each GPU device. Each helper manages all the operations on its own GPU. The original OmpSs implementation for GPUs [4] already supports asynchrony, but it is very restrictive and inflexible: each task execution cycle[1] can hold, at most, two data transfers (one in each direction) overlapped with one task execution. This means that for each task execution there can be two more operations: one device-to-host transfer that copies the output data of the previous executed task (if any) and one host-to-device transfer that prefetches the data for the next task (if any). Three CUDA streams are used to achieve the overlapping. This approach presents two main problems: first, the helper thread that manages the GPU gets blocked at the end of each task execution cycle waiting for all the operations of the cycle. This busy-waiting is done by calling *cudaStreamSynchronize()* for each stream. Second, operations can be overlapped inside one cycle, but there is an explicit synchronization between different cycles. Then, for each cycle, the thread has to wait for the longest operation before it can start a new cycle. This means that several CPU-time cycles are wasted just waiting for device operations. This gives us an opportunity to implement AMA on top of OmpSs to evaluate our proposal.

4 AMA Implementation

This section explains the implementation of AMA on top of the OmpSs CUDA device support. As mentioned before, this implementation is extensible to other task-based frameworks supporting asynchronous devices (OpenCL, FPGA, ...).

4.1 OmpSs CUDA Support with AMA

The host-device synchronization implementation of AMA is done by inserting a CUDA event after each device asynchronous operation (data transfer or kernel launch). Then, the helper thread links the event to the appropriate set of callbacks and holds a list of pending events (registered events that have not been raised) on its device. This list is checked by the thread (it queries CUDA for the state of each event), so when a raised event is detected, it will execute its associated callback actions. We use a combination of an event-polling mechanism and CUDA callbacks due to performance reasons. Several CUDA streams are used to overlap as many operations as possible. We keep two CUDA streams for data transfers (one for each direction) due to hardware limitations (although we could add more streams if the underlying hardware

[1] A task execution cycle is the set of all the operations needed to run a task: transfer input data to device, run the task on the device (usually, one kernel launch) and transfer output data back to host. Transfers can happen in a different cycle, depending on the cache policy used.

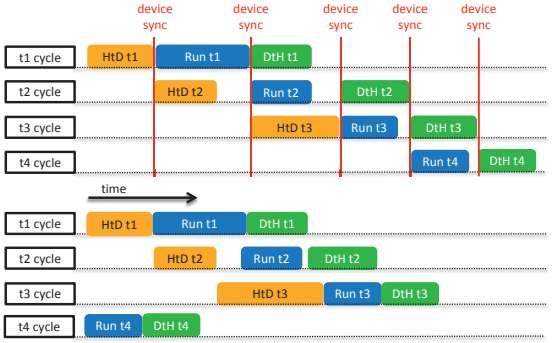

Figure 1: Task execution comparison on a GPU device

supported more simultaneous transfers). We also take advantage of CUDA's concurrent kernel execution feature by having multiple streams for kernel launches. The number of streams devoted to kernels is set accordingly to OmpSs run time configuration options.

Figure 1 compares the execution of several tasks with the original OmpSs implementation and with OmpSs+AMA. The top part shows a time line diagram of the execution of four tasks *t1*, *t2*, *t3* and *t4* on the same device with the original OmpSs implementation. The orange boxes labeled as *HtD* represent the amount of time spent in transferring data from host to device. The blue boxes labeled as *Run* represent the amount of time that task execution takes on the device. The green boxes labeled as *DtH* represent the amount of time spent transferring data from device to host. The red vertical lines show the synchronization points where the helper thread needs to block. In this example, we can see that waiting always for the longest operation in each task cycle delays other operations that could be done earlier. For example, the *DtH t2* transfer could be issued right after *t2*'s execution, but since the *HtD t3* transfer is longer, the *DtH t2* is delayed. Task *t4* does not need input data transfers, but its execution cannot be advanced due to the fixed in-order task execution of the original implementation. Moreover, although not shown in Figure 1, the helper thread spends most of its time waiting for the operations, as it calls *cudaStreamSynchronize()* to synchronize with the device.

The bottom part of Figure 1 shows the behavior of the OmpSs+AMA runtime proposal. We can see that the global execution time is lower because the synchronization points have been removed. We can still observe some gaps, but they are due to hardware limitations[2]. That is why tasks *t1* and *t2* can only partially overlap their execution. Since *t4* task does not need input data transfers, the FTFR scheduler can advance its execution right at the beginning: there is no need to wait for *t1*, *t2* and *t3* data transfers and execution. Then, *t4*'s output data transfers are overlapped with *t1*'s execution and *t2*'s input data transfers. The overall result is that the asynchronous runtime has significantly reduced the total execution time of these tasks. Moreover, *t4*'s dependences would be released at the end of its execution, so its dependent tasks would be ready to run much earlier.

4.2 Priority Propagation

Since the original implementation of OmpSs already supports tasks with priorities, we have modified this component in order to propagate task priority to several levels of predecessor tasks. With this modification, we can enhance the scheduling decisions for ready tasks, as predecessors of higher priority tasks will be scheduled earlier and thus dependences of higher priority tasks will be satisfied earlier as well. We have enabled the double-linked data dependency graph for

[2] A kernel can start only when all thread blocks of all prior kernels from any stream have started [10].

Feature	*Original OmpSs*	*OmpSs+AMA*
Host-device synchronization	Blocking sync with CUDA streams	Non-blocking CUDA events and callbacks with streams
Affinity computation	At task creation time	At task ready time
Task priority propagation	Direct parent only	Many levels of parents (configurable)
# task prefetch	One task ahead	Many tasks ahead (configurable)
# concurrent tasks	One task	Many tasks (if HW support) (config.)

Table 1: Differences between OmpSs and OmpSs+AMA

our approach, as it is disabled by default to avoid additional overhead. Then, we can navigate through task's predecessors and update their priority. The optimum number of predecessor levels to navigate and update priority is application-dependent. As we said before, the overhead that we are adding by enabling the double-linked graph is hidden by the execution of other tasks and their data transfers, so, effectively, it is cost-free in our implementation.

4.3 Affinity Scheduler

It will frequently happen in the original affinity scheduling policy that the computed affinity is outdated by the time the task becomes ready. Thus, we have modified this scheduler to better fit the execution of tasks on asynchronous devices. As we explained before, the original affinity computation is done too early, at task's instantiation stage. With AMA, the affinity computation is delayed until task's ready stage. We have also changed the affinity computation criteria to refine processing unit's scores. These changes give us an accurate and updated task affinity score for each processing unit that allows the runtime make better task scheduling decisions. In the same way as the priority propagation mechanism, the overhead of these modifications is hidden by the execution of other tasks and their data transfers.

Table 1 summarizes the differences between the original implementation of OmpSs (*Original OmpSs*) and our OmpSs+AMA proposal (*OmpSs+AMA*).

5 Evaluation

We present in this section the performance results of three applications in order to evaluate our AMA proposal implemented on top of the OmpSs framework. We compare these results with CUDA native versions and the original OmpSs framework on a multi-GPU Linux system with two Intel Xeon E5-2650 at 2.00 GHz, 62.9 GB of main memory and four NVIDIA Tesla K20c with 2496 CUDA cores and 4.7 GB of memory. The native CUDA codes were compiled with CUDA 5.5 and the OmpSs versions were compiled with OmpSs compiler (using *nvcc* 5.5 and GCC 4.6.4). Optimization level $-O3$ was used in all codes. The same application source code was used with both OmpSs and OmpSs+AMA runtimes. We run the applications with different configurations of number of GPU devices and data set sizes and analyze its impact on performance. Results are computed as the mean value of several executions.

N-body Simulation. In the context of computational science, the N-body simulation is a molecular dynamics computation where a system of bodies (atoms, molecules) is allowed to interact for a period of time. The result of the simulation gives a view of the motion of the bodies whose trajectories are determined by forces between bodies and their potential energy. The CUDA native implementation comes from CUDA 5.5 SDK examples [10, 8]. We transformed this code into an OmpSs application by adding some task directives around GPU kernel calls and removing all data transfers and GPU management. The performance of this

Configuration	*App version*	*Runtime*	*Data size* [Bodies]
OmpSs 256Kbod	OmpSs CUDA	OmpSs	262144
OmpSs+AMA 256Kbod	OmpSs CUDA	OmpSs+ AMA	262144
OmpSs 512Kbod	OmpSs CUDA	OmpSs	524288
OmpSs+AMA 512Kbod	OmpSs CUDA	OmpSs+ AMA	524288
CUDA 256Kbod	Native CUDA	CUDA	262144
CUDA 512Kbod	Native CUDA	CUDA	524288

Table 2: N-body configurations

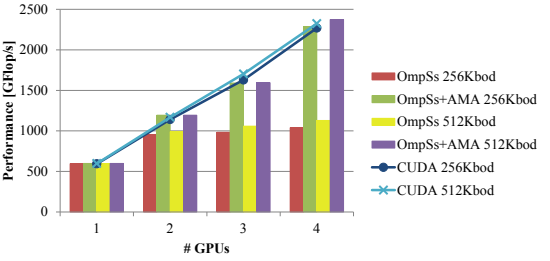

Figure 2: N-body simulation performance

Configuration	*App version*	*Runtime*	*Data set size* [DP FP elmts]
OmpSs 16K	OmpSs CUDA	OmpSs	16384×16384
OmpSs+ AMA 16K	OmpSs CUDA	OmpSs+ AMA	16384×16384
OmpSs 32K	OmpSs CUDA	OmpSs	32768×32768
OmpSs+ AMA 32K	OmpSs CUDA	OmpSs+ AMA	32768×32768
CUDA 16K	Native CUDA	CUDA	16384×16384

Table 3: Matrix multiplication configurations

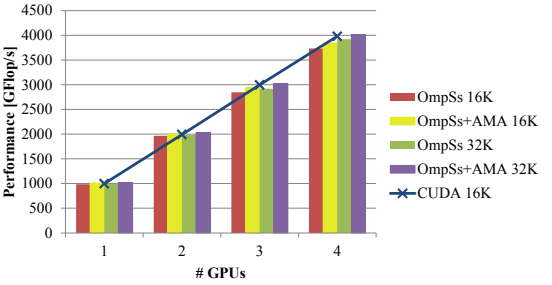

Figure 3: Matrix multiplication performance

simulation is limited by the amount of data that needs to be transferred between devices after each iteration. We have simulated 10 iterations of a body system with different number of bodies using double-precision floating-point data, described in Table 2.

Figure 2 shows the performance results of running the different configurations. The original OmpSs configurations (*OmpSs 256Kbod* and *OmpSs 512Kbod*) are negatively affected by the amount of data exchanged between iterations, up to the point of not being able to scale across several GPUs. In contrast, the OmpSs+AMA configurations (*OmpSs+AMA 256Kbod* and *OmpSs+AMA 512Kbod*) can scale at the same ratio as the original CUDA application does and, in some points, they even get slightly better performance than the native CUDA implementation. In this case, OmpSs+AMA gets up to 2.2x performance speed-up compared to the original OmpSs framework.

The results of the native CUDA version include less data transfers than the OmpSs version, as the initial and final transfers are not taken into account in native CUDA. All OmpSs results include all the data transfers. We demonstrate with these results that our approach can efficiently overlap data transfers and computations because we appreciate no differences between OmpSs+AMA performance and the native CUDA code performance.

Matrix Multiplication. The application performs a dense matrix multiplication of two square matrices: $A \times B = C$. The GPU computation is done by calling the *cublasDgemm()* function from CUBLAS library (CUDA 5.5). The different configurations tested are explained in Table 3. In the native CUDA version, matrices A and C are split into as many chunks as GPUs, so each GPU receives a set of consecutive rows. Matrix B is fully copied to all GPU devices. Then, all GPUs can compute in parallel with the others. In the OmpSs version, each matrix is divided into square blocks of 2048×2048 double-precision floating-point elements; each created task performs a matrix multiplication on a given block of the destination matrix.

Figure 3 shows the performance results of matrix multiplication run with the different configurations. We present the *CUDA 16K* performance as a chart line because this is the peak performance of this application (only kernel execution time is taken into account, without

Configuration	*App version*	*Runtime*	*Data set size* [DP FP elmts]
OmpSs 16K	OmpSs CUDA	OmpSs	16384 × 16384
OmpSs+ AMA 16K	OmpSs CUDA	OmpSs+ AMA	16384 × 16384
OmpSs 32K	OmpSs CUDA	OmpSs	32768 × 32768
OmpSs+ AMA 32K	OmpSs CUDA	OmpSs+ AMA	32768 × 32768
CUDA 16K	Native CUDA	CUDA	16384 × 16384
CUDA dgemm 16K ceiling ref.	Native CUDA	CUDA	16384 × 16384

Table 4: Cholesky configurations

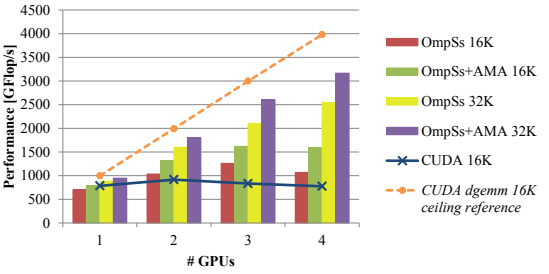

Figure 4: Cholesky performance compared to native CUDA DGEMM performance

data transfers). Since the application is highly parallel, the performance of *OmpSs 16K* and *OmpSs 32K* get close to our peak values, even though data transfers are taken into account in our measurements. The performance increases as we use bigger data set sizes because the application creates more tasks and thus, opens more parallelism. In the case of OmpSs+AMA, a bigger data set size (*OmpSs+AMA 32K*) is also better and we achieve the same performance as our peak line. This proves that we can fully occupy GPU resources and completely hide the overhead of data transfers. The 4% benefit that we get compared to the OmpSs implementation comes from the concurrent kernel execution, as data transfers are optimal in both cases.

Cholesky Factorization. The Cholesky factorization is a matrix operation that calculates a triangular matrix (L) from a symmetric and positive definite matrix (A): $Cholesky(A) = L$, where $L \cdot L^t = A$. The source code is the main algorithm of a blocked Cholesky factorization. The matrix A is organized in square blocks of 2048 × 2048 double-precision floating-point elements. The computation is done inside a set of nested loops that operate on these blocks by calling four different kernels: *dpotrf*, *dsyrk*, *dgemm* and *dtrsm*. We use a customized implementation of *dpotrf* based on its corresponding function from MAGMA library [9] and CUBLAS library (version 5.5) is called for the other kernels. The different configurations used are described in Table 4. Note that *CUDA dgemm 16K ceiling reference* is the performance of matrix multiplication previously evaluated in this section. Cholesky factorization cannot scale as good as matrix multiplication, but we add these results as a reference of scalability. The CUDA native version uses an OpenMP-like fork-join approach due to its complexity. In the OmpSs version, each kernel is annotated as a task and task data dependencies are managed by the OmpSs runtime. We used different task priorities to give more priority to critical tasks. OmpSs+AMA propagates task priorities up to five levels upwards.

Cholesky's performance results are shown in Figure 4. The native CUDA version cannot scale across several GPUs due to the synchronization and data exchange bottlenecks. This is the reason why the values of *CUDA dgemm 16K ceiling reference* are shown. The two OmpSs configurations (*OmpSs 16K* and *OmpSs+AMA 16K*) cannot scale beyond two GPUs because there is a lack of parallelism. In contrast, when we increase the data set size, we can see that both (*OmpSs 32K* and *OmpSs+AMA 32K*) scale better. We can observe that OmpSs+AMA configurations get better performance than original OmpSs thanks to its enhanced non-blocking data management and task priority propagation, increasing its performance up to 1.5x speed-up.

6 Related Work

Computational science can benefit from the performance of new computer architecture designs of heterogeneous multi-cores. However, their programmability complexity must be addressed.

OpenACC aims at simplifying offloading of tasks to accelerators. It defines a set of functions and compiler directives to specify parts of a program whose computation may be offloaded to an accelerator, to transfer data between main memory and the accelerator and to synchronize with the execution of those accelerator computations. In its current 2.0a version, OpenACC only addresses machines with one accelerator, while OmpSs+AMA addresses systems with multiple accelerators of different types as well. In the CUDA context, OpenACC also exposes low-level optimizations, like the use of CUDA streams, to the programmer. OmpSs+AMA is able to manage these low-level optimizations transparently.

hiCUDA [6] proposes the use of directives to give hints to the compiler about regions of code that can be exploited on GPUs and data motion. While hiCUDA is completely focused on CUDA, OmpSs+AMA also focuses on other accelerator types. Regarding the OmpSs CUDA support, hiCUDA directives are translated into CUDA API calls, but they do not use advanced CUDA features, like the use of streams or events to control device operations.

StarPU [2] is based on a tasking API and on the integration of a data-management facility with a task execution engine. With regard to data management, StarPU proposes a high level library that automates data transfers throughout heterogeneous machines. In StarPU, codelets are defined as an abstraction of a task that can be executed on a core or offloaded onto an accelerator using an asynchronous continuation passing paradigm. StarPU offers low level scheduling mechanisms to be used in a high level fashion, regardless of the underlying architecture. OmpSs+AMA and StarPU present several similarities with regard the execution model, but StarPU is not proposing a programming model and therefore the programmer is exposed with lower APIs and execution details that are hidden in the OmpSs+AMA case.

Pienaar et al. [13] describe a runtime framework to schedule Direct Acyclic Graphs (DAGs) in heterogeneous parallel platforms. The proposal is based on four important criteria: Suitability, Locality, Availability and Criticality (SLAC) and show that all these criteria must be considered in order to achieve good performance under varying application and platform characteristics. As StarPU, the authors propose a runtime scheduler while OmpSs+AMA is proposing a whole programming model and execution model.

Ueng et al. [15] propose tools to better map algorithms to machine's memory hierarchy. Programmers provide straight-forward implementations of the application kernels using only global memory and CUDA-lite tools do the transformations automatically to exploit local memories. The main difference with OmpSs+AMA is that CUDA-lite mainly focuses on kernel code, while OmpSs+AMA manages the whole application execution, including host-device communications.

7 Conclusions and Future Work

This paper presents AMA, a set of techniques for the Asynchronous Management of Accelerators. AMA can be used in task-based programming frameworks to increase application performance and to reduce runtime overhead on multi-accelerator systems. The first technical contribution is a new design for the management of asynchronous devices, such as GPUs. Our proposal avoids blocking states of host threads devoted to device management; instead, it offers a blocking-free host-device communication mechanism for device computations, data transfers and synchronizations. For that purpose, events and callbacks are used to manage communications. This mechanism allows us to spend the previously wasted blocking times on doing other useful work in the runtime. In that sense, our second and third contributions rely on the fact that they could not be implemented without our first contribution due to overhead reasons. They are related to task scheduling techniques: having removed the host-side blocking points, we have some free time to spend on making smarter scheduling decisions. On the one hand, we propose an enhancement of the data- affinity scheduling policy. This policy's criteria focuses on

data locality to decide which computing unit runs each task. On the other hand, we propose a task priority mechanism that promotes the execution of critical tasks and their predecessors (for example, high-priority tasks, or tasks in the critical path of the data dependency graph).

We have implemented our proposal on top of OmpSs, a task-based programming framework with multi-GPU support. Our experiments show that our proposal outperforms in all cases the performance of the original implementation, reaching more than 2x speed-up. Furthermore, we can reach the same performance, or even get better results, than a native hand-tuned CUDA application, with the advantages of task-based programming models that leverage the programmer from many issues related to heterogeneous and asynchronous devices (like device management, memory coherency or data transfers). As future work, we plan to extend our implementation to support other architectures, like OpenCL or FPGA devices. Then, we will also need to tune our scheduling parameters to fit all of them. In addition, we will consider implementing AMA for a cluster architecture as well.

Acknowledgments

European Commission (HiPEAC-3 Network of Excellence, FP7-ICT 287759), Intel-BSC Exascale Lab and IBM/BSC Exascale Initiative collaboration, Spanish Ministry of Education (FPU), Computación de Altas Prestaciones VI (TIN2012-34557), Generalitat de Catalunya (2014-SGR-1051). We thank KAUST IT Research Computing for granting access to their machines.

References

[1] TOP500 Supercomputing Site. June 2014. http://www.top500.org/lists/2014/06.

[2] C. Augonnet et al. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, 2011.

[3] E. Ayguade et al. Extending OpenMP to Survive the Heterogeneous Multi-core Era. *International Journal of Parallel Programming*, 38(5-6):440–459, June 2010.

[4] J. Bueno-Hedo et al. Productive Programming of GPU Clusters with OmpSs. In *Proceedings of the 26th IEEE Int. Parallel and Distributed Processing Symposium*, IPDPS 2012, May 2012.

[5] A. Filgueras et al. OmpSs@Zynq All-programmable SoC Ecosystem. In *Proc. of ACM/SIGDA Int. Symp. on Field-programmable Gate Arrays*, FPGA '14, pages 137–146, NY, USA, 2014.

[6] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems*, 22:78–90, 2011.

[7] Khronos OpenCL Working Group. *The OpenCL Specification, version 2.0*, March 2014.

[8] Mark Harris Lars Nyland and Jan Prins. Chapter 31: Fast N-Body Simulation with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007.

[9] R. Nath, S. Tomov, and J. Dongarra. An Improved MAGMA GEMM for Fermi GPUs. Technical Report UT-CS-10-655, University of Tennessee Computer Science, July 2010.

[10] NVIDIA. *CUDA C Programming Guide Version 5.5*. NVIDIA Corporation, July 2013.

[11] OpenMP ARB. OpenMP Application Program Interface, v. 4.0, July 2013.

[12] J.M. Perez, R.M. Badia, and J. Labarta. A Dependency-aware Task-based Programming Environment for Multi-core Architectures. *IEEE Int. Conf. on Cluster Comp.*, pages 142–151, 2008.

[13] J.A. Pienaar et al. MDR: performance model driven runtime for heterogeneous parallel platforms. In *Proc. of the Int. Conf. on Supercomputing*, ICS '11, pages 225–234, NY, USA, 2011. ACM.

[14] J. Planas et al. Self-Adaptive OmpSs Tasks in Heterogeneous Environments. In *IEEE 27th Int. Parallel & Distributed Processing Symp. (IPDPS)*, pages 138–149, 2013.

[15] S. Ueng, M. Lathara, S.S. Baghsorkhi, and W.W. Hwu. CUDA-lite: Reducing GPU Programming Complexity. In *21st Languages and Compilers for Parallel Computing (LCPC)*, 2008.