Extending OmpSs for OpenCL kernel co-execution in heterogeneous systems

B. Pérez, E. Stafford, J. L. Bosque, R. Beivide
Department of Computer Science and Electronics
Universidad de Cantabria
Santander, Spain
{perezpavonb,stafforde,bosquejl,beivider}@unican.es

S. Mateo, X. Teruel, X. Martorell, E. Ayguadé
Barcelona Supercomputing Center
Universidad Politécnica de Cataluña
Barcelona, Spain
{sergi.mateo,xavier.teruel,xavier.martorell,eduard.ayguade}@bsc.es

***Abstract*—Heterogeneous systems have a very high potential performance but present difficulties in their programming. OmpSs is a well known framework for task based parallel applications, which is an interesting tool to simplify the programming of these systems. However, it does not support the co-execution of a single OpenCL kernel instance on several compute devices. To overcome this limitation, this paper presents an extension of the OmpSs framework that solves two main objectives: the automatic division of datasets among several devices and the management of their memory address spaces. To adapt to different kinds of applications, the data division can be performed by the novel HGuided load balancing algorithm or by the well known Static and Dynamic. All this is accomplished with negligible impact on the programming. Experimental results reveal that there is always one load balancing algorithm that improves the performance and energy consumption of the system.**

***Index Terms*—Heterogeneous systems, OmpSs programming model, OpenCL, co-execution**

I. Introduction

The scene of high performance computing (HPC) is becoming more and more dependant on compute accelerators. This is not only due to their high performance but also their outstanding energy efficiency. Interestingly, the success of these devices comes in spite of the fact that it is difficult to develop efficient software for machines that integrate them.

Programming heterogeneous systems, that integrate compute accelerators like GPUs, can be accomplished through several frameworks like CUDA and OpenCL. However, instead of presenting the programmer with a convenient abstraction, both regard the system as a collection of independent devices. This allows accessing the computing power of the devices, but there is not an effortless way to squeeze all the power out of the heterogeneous system, as each device must be handled independently.

The OmpSs programming model presents a change of paradigm in many ways. It provides support for task parallelism due to its benefits in terms of performance, cross-platform flexibility and reduction of data motion [1]. The programmer divides the code in interrelating tasks and OmpSs essentially orchestrates their parallel execution maintaining their control and data dependences. To that end, OmpSs uses the information supplied by the programmer, via code annotations with pragmas, to determine at run-time, which parts of the code can be run in parallel. It enhances OpenMP with support for irregular and asynchronous parallelism, as well as support for heterogeneous architectures. OmpSs is able to run applications on symmetric multiprocessor (SMP) systems with GPUs, through OpenCL and CUDA APIs [2].

However, OmpSs did not support co-execution of kernels, that is to automatically run a single kernel instance on all the available devices in a heterogeneous system. Doing so would require extra effort from the programmer, who would have to decompose the kernel in smaller tasks so that OmpSs could send them to the devices. However, there would be no guarantee that the resources would be efficiently used and the load properly balanced. The programmer would also be left alone in terms of dividing and combining the data. This would lead to longer code, which would be harder to maintain.

As a solution to the above problems this article presents an OmpSs extension which enables the efficient co-execution of massively data-parallel OpenCL kernels in heterogeneous systems. By automatically using all the available resources, regardless of their number and characteristics, it presents an easy way to perform kernel co-execution and extracting the maximum performance of these systems. The extension takes care of load balancing, input data partitioning and output data composition. To suit the different behaviour of applications, the extension presents the programmer with three different load balancing algorithms. This behaviour may be *regular*, when every work unit represents the same running time, or *irregular*, if different work units have different running times.

The experimental results presented here indicate that, for all the used benchmarks, the utilisation of the whole heterogeneous system has a positive impact on performance. In fact, the results show that it is possible to reach an efficiency of the heterogeneous system over 0.85, if the right algorithm is chosen. Furthermore, the results also show that, although the systems exhibit higher power demand, the shorter execution time grants a notable reduction in the energy consumption. Indeed, the average energy efficiency improvement observed is $2.62\times$.

The main contributions of this article are the following:

- Extending the OmpSs programming model with a new scheduler, that allows a single OpenCL kernel instance

to be co-executed by all the devices of a heterogeneous system.
- The implementation of a set of load balancing algorithms that adapt to the needs of the different applications.
- Presenting an exhaustive experimental study that corroborates that using the whole system is beneficial in terms of energy consumption as well as performance.

The rest of this paper is organised as follows. Section II presents background concepts key to the understanding of the paper. Next, Section III describes the details of the load balancing algorithms. Followed by Section IV, that covers the implementation of the OmpSs extension. Section V presents the experimental methodology and discusses its results. Finally, Section VII offers some conclusions and future work.

II. Background

This section explains the main concepts of the OmpSs programming model that will be used throughout the remainder of the article.

OmpSs is a programming model based on OpenMP and StarSs. Which has been extended in order to allow the inclusion of CUDA and OpenCL kernels in Fortran and C/C++ applications as a simple solution to execute on heterogeneous systems [1], [2]. It supports the creation of data-flow driven parallel programs that, through the asynchronous parallel execution of tasks, can take advantage of the computing resources of a heterogeneous machine. The programmer declares the tasks through compiler directives (pragma) in the source code of the application. These are used at runtime to determine when the tasks may be executed in parallel.

OmpSs is built on top of two tools:

- *Mercurium* is a source-to-source compiler that processes the high-level directives, and transforms the input code into a parallel application [3]. In this manner, the programmer is spared of low level details like the thread creation, synchronization and communication, as well as the offloading of kernels in a heterogeneous system.
- *Nanos++* is a run-time library that provides the necessary services for the execution of the parallel program [4]. Among others, these include task creation and synchronization, but also data marshaling and device management.

In the pragma annotations, the programmer specifies the data dependences between the tasks. Then, when the execution of the parallel program commences, a thread pool is created. Of these, only the master thread is active, and uses the services of the run-time library to generate tasks, identified by work descriptors, and adding them to a dependence graph. The master thread then schedules the execution of the tasks to the threads in the pool as soon as their input dependences are satisfied.

In terms of heterogeneous systems, OmpSs provides a *target* directive that indicates a set of devices in which a given task can run. In addition to a task, the target directive can be applied to a function definition. OmpSs also offers the *ndrange* clause that, together with the data-directionality clauses *in* and *out*, guides the data transfer between the devices and the host CPU, so the programmer perceives a single unified address space.

However, OmpSs does not support the execution of a single kernel instance in several devices. The extension proposed in this article modifies the Nanos++ runtime system so that it can automatically divide a kernel into sub-kernels and manage the different memory address spaces. In order to make the co-execution efficient, three load balancing algorithms have been implemented to suit the behaviour of different applications.

III. Load Balancing Algorithms

The behaviour of the three algorithms (Static, Dynamic and HGuided) is illustrated in Figure 1. It shows the ideal case in which in the execution of a regular application all devices finish simultaneously, thus achieving perfect load balance.

A. Static algorithm

This algorithm works before the kernel starts its execution by dividing the dataset in as many *packages* as devices are in the system. The division relies on knowing the computing power of the devices in advance. Then the execution time of each device can be equalised by proportionally dividing the dataset among the devices. As a consequence, there is no idle time in any device, which would signify a waste of resources. The idea of assigning a single package to each device is depicted in Figure 1.

A formal description of the algorithm can be made considering a heterogeneous system with n devices. Each device i has *computational power* P_i, which is defined as the amount of work that a device can complete per time unit, including the communication overhead. This value depends on the architecture of the device, but also on the application that is being run. These powers are input parameters of the algorithm and can be extracted by a simple profiled execution.

The application will execute a kernel over W work-items, grouped in G work-groups of fixed size $L_s = \frac{W}{G}$. Since the work-groups do not communicate among themselves, it makes sense to distribute the workload taking the work-group as the atomic unit. Each device i will have an execution time of T_i. Then the execution time of the heterogeneous system will be that of the last device to finish its work, or $T_H = max_{i=1}^{n} T_i$. Also, since the whole system is capable of executing W work-items in T_H, it follows that its total computational power of the heterogeneous system is $P_H = \frac{W}{T_H}$. Note that it also can be computed as the sum of the individual powers of the devices.

$$P_H = \frac{W}{T_H} = \sum_{i=1}^{n} P_i$$

The goal of the Static algorithm is to determine the number of work-groups to assign each device, so that all the devices finish their work at the same time. This means finding a tuple $\{\alpha_1, ...\alpha_n\}$, where α_i is the number of work-groups assigned to the device i, such that:

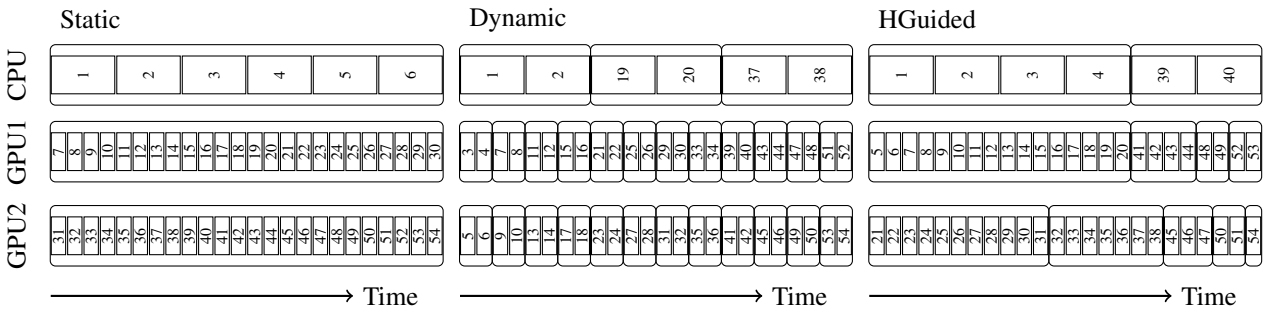

Fig. 1. Depiction of how the three algorithms perform the data division among three devices. The work groups assigned to each device, identified by numbers, are joined in packages shown as larger rounded boxes. The execution time of work groups in CPU is four times larger than in the GPUs.

$$T_H = T_1 = \cdots = T_n \Leftrightarrow \frac{L_s \alpha_1}{P_1} = \cdots = \frac{L_s \alpha_n}{P_n}$$

This set of equations can be generalised and solved as follows:

$$T_H = \frac{L_s \alpha_i}{P_i} \Leftrightarrow \alpha_i = \frac{T_H P_i}{L_s} = \frac{T_H P_i G}{W} = \frac{P_i G}{\sum_{i=1}^{n} P_i}$$

Since α_i is the number of work-groups, its value must be an integer. For this reason, the expression used by the algorithm is:

$$\alpha_i = \left\lfloor \frac{P_i G}{\sum_{i=1}^{n} P_i} \right\rfloor$$

If there is not an exact solution with integers then $\sum_{i=1}^{n} \alpha_i < G$. In this case, the remaining work-groups are assigned to the most powerful device.

The advantage of the Static algorithm is that it minimises the number of synchronisation points. This makes it perform well when facing regular loads with known computing powers that are stable throughout the dataset. However, it is not adaptable, so its performance might not be as good with irregular loads.

B. Dynamic algorithm

Some applications do not present a constant load during their executions. To adapt to their irregularities, the dynamic algorithm divides the dataset into small packages of equal size. The number of packages is well above the number of devices in the heterogeneous system. During the execution of the kernel, a master thread in the host is in charge of assigning packages to the different devices, following the next strategy:

1) The master splits the G work-groups in packages, each with the package size specified by the user. This number must be a multiple of the work-group size. If the number of work-items is not divisible by the package size, the last package will be smaller
2) The master launches one package on each device, including the host itself if it is desired.
3) The master waits for the completion of any package.
4) When device i completes the execution of a package:
   a) The device returns the partial results corresponding to the processed package.
   b) The master stores the partial results.
   c) If there are outstanding packages, a new one is launched on device i.
   d) If all the devices are idle and there are no more packages, the master jumps to step 5.
   e) The master returns to step 3.
5) The master ends when all the packages have been processed and the results have been received.

This behaviour is illustrated in Figure 1. The dataset is divided in small, fixed size packages and the devices process them achieving equal execution time. As a consequence, this algorithm adapts to the irregular behaviour of some applications. However, each completed package represents a synchronisation point between the device and the host, where data is exchanged and a new package is launched. This overhead has a noticeable impact on performance. The Dynamic algorithm takes the size of the packages as a parameter.

C. HGuided algorithm

The two above algorithms are well known approaches to the problem of load balancing in general. But none satisfy three key aspects. First, take into account the heterogeneity of the system. Second, control the overhead of the synchronisation. And third, give reasonable performance with applications of different behaviours. Thus a new load balancing algorithm proposal called *HGuided* is proposed, which is based on the *Guided* method from OpenMP.

In this algorithm, the packet size is dependent on the computing power of the assigned device. Which allows taking full advantage of fast devices with large packets while assigning smaller packets to slower devices. This, also reduces the number of synchronization points and the corresponding overhead, compared to the Dynamic. As in the Guided algorithm, the size of the packets is reduced as the execution progresses, thus reaching a small package granularity towards the end of the execution to allow all devices to finish simultaneousy.

The package size for device i is calculated as follows:

$$packet\_size_H = \left\lfloor \frac{G_r P_i}{k \sum_{j=1}^{n} P_j} \right\rfloor$$

Where G_r is the number of pending work-groups and is updated with every package launch. Like the Static algorithm, the HGuided takes the computing powers of the devices as parameters. It also requires a minimum package size, which is a lower bound for the $packet\_size_H$. Due to the unpredictable behaviour of the irregular applications, the constant k is introduced to limit the maximum package size, taking a value of 2 in the experimental evaluation of Section V. Once the size of the packages is defined, they are assigned to the devices much like in the Dynamic.

Figure 1 shows how the size of the packages is large at the beginning of the execution, and decreases towards the end.

IV. Implementation

As stated before, the OmpSs infrastructure relies on the combination of two components: Mercurium, which is a source-to-source compiler, and Nanos++, which is a runtime capable of managing tasks, their data and the *Task Dependence Graph (TDG)* their dependences generate. As a first approach, the new load balancing algorithms have been implemented focusing on making the changes as self-contained as possible and minimizing the impacts on the OmpSs specification, Mercurium and the rest of Nanos++. As a result, neither directives nor clauses have been added to Mercurium. Nanos++ implements a set of different schedulers that deal with the management of the tasks submitted to the runtime. To offer the work distribution strategies for a single OpenCL task presented in the previous section, a new scheduler has been implemented as a Nanos++ plugin. The parameters of the algorithms are the following:

- The device computing powers for Static and HGuided.
- The package size for Dynamic.
- The minimum package size for HGuided.

To avoid altering the OmpSs specification, the algorithm used and its parameters are selected through environment variables, which is the normal way to specify the scheduler in Nanos++.

Figure 2 represents the outline of an OmpSs implementation of the Binomial benchmark used later in the experimentation. It shows how a call to a function defined as a task is followed by a wait. The header of that function, which is shown in Figure 3, indicates that the task must be run in an OpenCL device, as well as its launch parameters, input and output data. Figure 4 shows the environment variables that need to be set to run the task with each of the three algorithms presented in Section III.

Despite the efforts made to minimize the impact on Mercurium, a minor change was unavoidable. The original implementation did not make OpenCL kernel configuration parameters available to Nanos++. This information is necessary for the operation of the plugin, as it defines the amount of work that will be performed. Nanos++ work descriptors do not hold this information either. Consequently, a new Mercurium work

```
//Initializations
binomial_options(NUM_STEPS, SAMPLES,
                 randArray, output);
#pragma omp taskwait
//Free resources
```

Fig. 2. Basic outline of an OmpSs application.

```
#pragma omp target device(opencl) copy_deps
        ndrange(1,samples*(numSteps+1),
                numSteps+1)
#pragma omp task in([samples]randArray) \
                 out([samples]output)
__kernel void binomial_options(int numSteps,
          int samples, const __global float4*
          randArray, __global float4* output);
```

Fig. 3. Header file for the task.

descriptor creation function has been implemented, which behaves like the original but including these parameters.

When a work descriptor is submitted, the new scheduler manages its division in as many work descriptors as the selected algorithm and parameters require. These work descriptors are considered as children of the one submitted, and represent an aggregate workload equivalent to that of their parent. For the Static and Dynamic algorithms, in which the number and size of the packages are known when the launch of the workload is made, all the work descriptors are created at the submission of their parent. They are stored in the scheduler and adequately returned when a thread is idle, receptive to another task. In the case of the HGuided algorithm, the packages have varying sizes that depend on the prior execution and the device that will run them. As a consequence, the children work descriptors will be created when required by an idle thread, considering the device it manages.

Each of the children work descriptors is identical to its parent except for two key differences. First, it has different OpenCL parameters, namely *global_work_size* and *offset*, defining the workload of the package it represents. Second, its output data is just a portion of that of its parent, which is conveniently offset so the results are written adequately. This is represented by an independent *CopyData* object, holding the start address and size that the package will have to work on. As a result, coherence problems are avoided in the OmpSs directory. Apart from the aforementioned details, data transfer relies on the methods used by standard OmpSs. To perform the correspondence between work descriptors and output data, an assumption is made: each OpenCL work-item will produce the result for the position of the output buffers indexed by its identifier. This may seem a strong requirement, but it is met by most kernels widely used in the industry and research.

The creation of the children work descriptors is performed by a modified version of the *duplicateWD* function that does this extra work. This function is also responsible for mak-

```
# Static load balancing
NX_SCHEDULE=maat
NX_ALGORITHM=static
NX_GPU_POWER=34.0

# Dynamic load balancing
NX_SCHEDULE=maat
NX_ALGORITHM=dynamic
NX_DYN_PACKAGE_SIZE=409600

# HGuided load balancing
NX_SCHEDULE=maat
NX_ALGORITHM=hguided
NX_GPU_POWER=34.0
NX_MIN_PACKAGE_SIZE=115200
```

Fig. 4. Environment variables to use standard OmpSs and the different load balancing algorithms.

ing the OpenCL parameters of the divided work descriptors available to the Mercurium code, which will trigger the actual kernel launches.

Once the submission of the original work descriptor is completed, the *done* function is called. This is a Nanos++ function that is used to signal the completion of a work descriptor. It also waits for the completion of the children of the calling work descriptor. In this way, no task dependent on the divided one will be run until all the children resulting from the work distribution are completed, so the dependencies of the task graph are maintained.

V. Evaluation

This Section begins with a description of the system and the benchmarks used in the experiments, as well as definitions of the metrics used in the evaluation. Additionally experimental results are showed and analysed.

A. System Set-up

The test machine has two processor chips and two GPUs. The chips are Intel Xeon E5-2620, with six cores that can run two threads each at 2.0 GHz. They are connected via QPI, which allows OpenCL to detect them as a single device. Thus, any reference to the CPU considers both processors. The GPUs are NVIDIA Kepler K20m with 13 SIMD lanes and 2496 cores. The experiments build upon a baseline system which uses a single GPU but consider the static energy of all the devices, regardless of if they are contributing work or not.

Six applications have been chosen for the experimentation. Three of them: *NBody*, *Krist* and *Perlin* are part of the OmpSs examples offered by BSC, and the other three: *Binomial*, *Sparse Matrix and Vector product (SpMV)* and *Rap* have been specifically adapted to OmpSs from existing OpenCL applications. The first four applications (NBody, Krist, Binomial and Perlin) are regular, meaning that all the work-groups represent a similar amount of work. On the contrary, SpMV and Rap are irregular, which implies that each work-group represents a different amount of work. The parameters associated to each of the load balancing algorithms have been set to maximise performance.The computing power for a device/application pair has been obtained as the relative performance of the device, with respect to that of the fastest device for the application.

TABLE I
Maximum achievable speedup per application.

Application	NBody	Krist	Binomial	Perlin	SpMV	RAP
Max. Speedup	2.61	2.2	2.03	2,04	2.05	2,16

Perlin implements an algorithm that generates noise pixels to improve the realism of moving graphics. Krist is used on crystallography to find the exact shape of a molecule using Röntgen diffraction on single crystals or powders. Rap is an implementation of the Resource Allocation Problem. It has a certain pattern in its irregularity, because each successive package represents an amount of work larger than the previous.

The evaluation of the performance of the benchmarks is done through their response time. This includes the time required by the communication between host and the devices, comprising input data and result transfer, as well as the execution time of the kernel itself. The benchmarks are executed in two scenarios, the *heterogeneous system*, taking advantage of the GPUs and CPU, and the *baseline*, that only uses one GPU. Note that in both instances, the same version of the program is run, as there is no need to modify the source or recompile, only set environment variables.

Based on these response times, two metrics are analysed. The first is the speedup for each benchmark when comparing the baseline and the heterogeneous system response times. Note that, for the employed benchamrks, the CPU is much less powerful than the GPUs, then the maximum achievable speedup using the three devices is not 3, but a fraction over 2 which depends on the computing power of the CPU for the application. The speedup for each application using a perfectly balanced work distribution is shown in Table I. These values give an idea of the advantage of using the complete system.

The second metric is the load balancing efficiency, obtained by dividing the reached speedup by the maximum speedup, shown in Table I. The obtained value ranges between 0 and 1 giving an idea of the usage of the heterogeneous system. Efficiencies close to 1 indicate the best usage of the system is being made. The measured values do not reach this ideal because of the communication and synchronisation times between the host and the devices.

The energy consumption is measured with an in-house developed tool that access the different hardware counters of the devices. For the CPU it uses the *Running Average Power Limit (RAPL)* registers. And for the NVIDIA GPUs, it takes advantage of the *NVIDIA Management Library (NVML)*.

The performance and the energy consumption can be combined in a single metric representing the energy efficiency of the system. This paper uses the Energy Delay Product (EDP) [5] for this purpose.

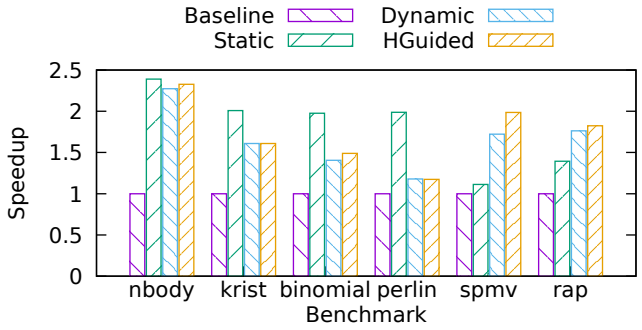

Fig. 5. Speedup per application.

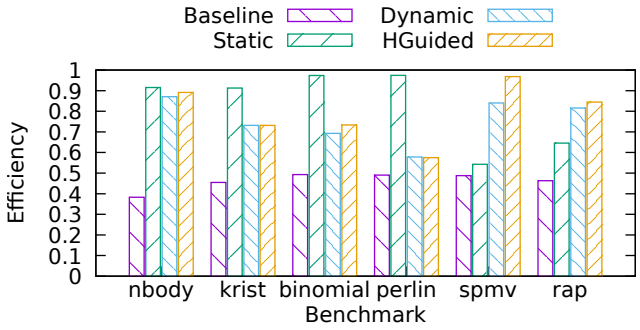

Fig. 6. Efficiency of the heterogeneous system.

B. Experimental results

Figure 5 shows the speedup obtained for each application calculated with respect to their execution time using baseline system, as was explained above. The results for the regular applications show that, as expected, the best option is the Static algorithm. The other two algorithms achieve good results, however suffer from a problem that hurts performance. If one of the last packages is assigned to the slowest device it will delay the execution of the whole application. This problem could be avoided by increasing the number of packages, but in that case overheads come into play and degrade performance too. The HGuided algorithm due to its very nature, partially solves this issue. In the case of the irregular applications, the Static algorithm fails to balance the load. This is a consequence of the behavior of these applications, with work-groups representing different execution times. The Dynamic and HGuided methods manage to balance the load more adequately, with the latter outperforming the former and achieving really good results.

The load balancing efficiency gives an idea of how well a load is balanced. A value of one represents that all the devices have been working all the time, thus achieving the maximum speedup. As the Figure 6 shows, there is at least one load balancing algorithm for every application that achieves an efficiency over 0.85. This is true even for the irregular applications, in which obtaining a balanced work distribution is significantly harder. In general it can be said that the efficiency can be largely improved, for instance it can be as high as 0.98, reached by Binomial and Perlin with the Static.

Nowadays, performance is not the only figure of merit used to evaluate computing systems. Their energy consumption and efficiency are also very important. Figure 7 gives an idea of the energy saving that comes through the usage of the whole heterogeneous system, instead of the baseline system that only uses the GPU while the other devices are idle, but still consuming. Therefore, the Figure shows for each benchmark the energy consumption of each algorithm normalised to the baseline consumption (Less is better).

The first fact worth mentioning is that, for all the benchmarks, it is possible to find an algorithm that reduces the energy consumption (The normalised energy is less than 1.) The use of more devices necessarily increases the instantaneous power at any time. But, since the total execution time is reduced, the total energy consumption is also less. This saving is further improved by the fact that idle devices still consume energy, so making all devices contribute work is beneficial.

The analysis of the algorithms shows a strong correlation between performance and energy saving. Consequently, the best algorithm for regular applications is also the Static, with an average saving of 26.5%. The saving with irregular applications is lower than with regular, around 12%, as the best algorithms is HGuided that adds communication overhead.

Another interesting metric is the energy efficiency, which combines performance with consumption. With the dual goal of low energy and fast execution in mind, the *Energy Delay Product (EDP)* is the product of the consumed energy and the execution time of the application. Figure 8 shows the EDP of the algorithms normalised to the EDP of the baseline.

Since the EDP is a combination of the two above metrics, the relative advantage of the different algorithms is maintained. Both, the Static algorithm on regular applications, and the HGuided on irregular, sensibly reduce the EDP measured on the baseline. This leads to an improvement of the energy efficiency of 2.9× and 2.13×, respectively. Considering all the benchmarks, the average improvement observed is 2.62×, if the best algorithm for each benchmark is selected. In some cases, the improvement can be well above these values, like in Binomial with 3.42×.

Judging by the results of all the experiments presented in this section, a set of conclusions can be reached. First, the utilization of all the devices of a heterogeneous system to execute the benchmarks significantly improves their performance, energy consumption and energy efficiency. However, it is important to choose the right load balancing algorithm to achieve these advantages.

Regarding this choice, it can be said that the Static algorithm is the best for regular applications. However, this algorithm is very sensitive to the selection of computing powers, making a calibration stage necessary to achieve the best results. The Dynamic algorithm can offer good results in some particular cases, but in general the results vary, and it is considered the less reliable algorithm. The HGuided algorithm is the best for irregular applications, and it would give the same results as

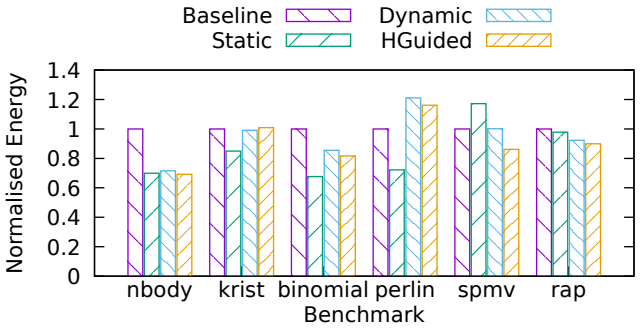

Fig. 7. Normalised energy consumption per application.

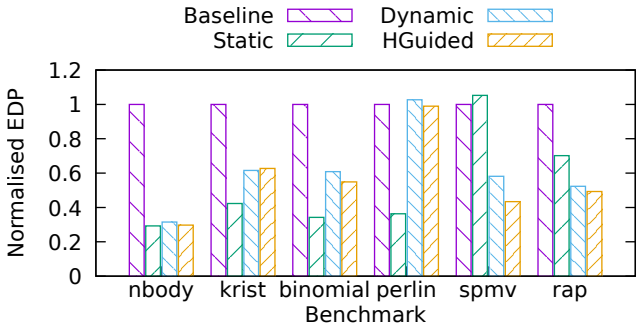

Fig. 8. Normalised EDP per application.

Static, if it were not for the communication overhead. Meaning that it adapts well to different execution patterns, and shows always outstanding balance among the devices.

VI. Related Work

Heterogeneous architectures show significant advantages in performance and energy consumption with respect to traditional systems. However, they also present the programmer with new challenges that make the use of such systems difficult. The keystones to make programming easy again are system abstraction, so the heterogeneous devices are handled transparently, and load balancing, so the resources are adequately used. Nevertheless, related as they are, these problems are often addressed separately.

To the load balancing problem, there are two main approaches found in the literature: *static* and *dynamic*.

Regarding the static, Lee *et al.*[6] propose the automatic modification of OpenCL code that executes on a single device, so the load is balanced among several ones. De la Lama *et al.* [7] propose a library that implements static load balancing by encapsulating standard OpenCL calls. In [8] the distribution is obtained with automatic code modifications. Zhong *et al.* [9] use performance models. These works fail to address the importance of adaptability to irregular loads.

In the dynamic approach [10], [11] propose different techniques and runtimes. However, these focus on task distribution and not on the co-execution of a single data parallel kernel. The work of [12] deals with the dynamic distribution of TBB parallel_for loops, adapting block size at each step to improve balancing. FluidicCL [13] does focus on co-execution but for systems with a CPU and a GPU. SnuCL [14] also tackles data parallelism, but is mostly centered on the distribution of the load among different nodes using an OpenCL-like library.

Kaleem's *et al.* proposal in [15] and Boyer's *et al.* in [16] use the execution time of the first packages to distribute the rest of the load and focus on integrated systems. Scogland *et al.* [17] also use a mixed strategy to adapt OpenMP to heterogeneous systems.

Some papers propose algorithms to distribute the workload between CPU and GPU taking performance and power into account. For instance, GreenGPU dynamically distributes work to GPU and CPU, minimizing the energy wasted on idling and waiting for the slower device [18]. To maximize energy savings while allowing marginal performance degradation, it dynamically throttles the frequencies of CPU, GPU and memory, based on their utilizations. Wang and Ren [19] propose a power-efficient load distribution method for single applications on CPU-GPU systems. The method coordinates inter-processor work distribution and frequency scaling to minimize energy consumption under a length constraint. SPARTA is a throughput-aware runtime task allocator for Heterogeneous Many Core platforms [20]. It analyzes tasks at runtime and uses the obtained information to schedule the next tasks maximizing energy-efficiency.

With respect to the problem of transparently managing a heterogeneous system, Tupinamba [21] proposes a framework for OpenCL, that enables the transparent use of distributed GPUs. In this same vein, Cabezas *et al.* [22] present an interesting architecture-supported take on efficient, transparent data distribution among several GPUs. Nevertheless, this works overlook load balancing, which is essential when trying to make the most of several heterogeneous devices.

You [23], Zhong [24] and Ashwin [25] do address both load balancing while abstracting the underlying system and data movement. Nevertheless, their focus is on task-parallelism instead of on the co-execution of a single data-parallel kernel. Kim *et al.* [26] approach the problem by implementing an OpenCL framework that provides the programmer with a view of a single device by transparently managing the memory of the devices. Their approach is based on a Static load balancing strategy, so it can not adapt to irregularity. Besides, they only consider systems with several identical GPUs, lacking the adaptability that OmpSs offers.

There are also some contributions that focus on scheduling and load balancing for OmpSs tasks. For instance, the scheduler presented in [27] is closer to the idea of co-execution. It holds several implementations of a task, targeted for different devices, that will be run iteratively. The scheduler stores the execution time of each implementation, so it can take load balancing decisions on what implementation is best to run next. However, the programmer is responsible for mapping the computation on several iterative tasks, which may not be an easy and natural approach for the application at hand.

VII. Conclusions and Future Work

This paper presents a new scheduler of the OmpSs programming model that allows to efficiently co-execute a single OpenCL kernel instance using all the devices in a heterogeneous system. Due to the different behaviour of applications, there is a need of more than one load balancing algorithms. Thus, the scheduler implements three load balancing algorithms, the well known Static and Dynamic, plus a novel algorithm taylored for hetergeneous systems called HGuided. The scheduler has been conceived so that it is fully transparent to the programmer, who only needs to select the algorithm and set its parameters through environment variables.

From the evaluation presented above, a set of conclusions can be highlighted. The use of the whole heterogeneous system is beneficial, both from the performance and the energy points of view, with at least one load balancing method. Regarding the individual algorithms, the Static is the most adequate for regular applications and when the computing power of the devices are known, as it minimises overheads. The other algorithms represent suboptimal yet good solutions for this kind of applications. In the case of irregular applications the HGuided method is the best choice. The use of Static is strongly discouraged as it may result in poor performance due to imbalance. Lastly, the Dynamic algorithm is a good all-around option when a priori information of the computing powers is not available, at the cost of more modest speedups.

The future of this extension will see compatibility with new devices, like Intel Xeon Phi, FPGAs or integrated GPUs. From the OmpSs perspective, a modification of the pragma specification would allow the programmer to select different algorithms or parameters for different kernels of the same application. It would be interesting to extend the evaluation to different systems and device configurations.

Acknowledgment

This work has been supported by the University of Cantabria with grant CVE-2014-18166, the Generalitat de Catalunya under grant 2014-SGR-1051, the Spanish Ministry of Economy, Industry and Competitiveness under contracts TIN2016-76635-C2-2-R (AEI/FEDER, UE) and TIN2015-65316-P. The Spanish Government through the Programa Severo Ochoa (SEV-2015-0493). The European Research Council under grant agreement No 321253 European Community's Seventh Framework Programme [FP7/2007-2013] and Horizon 2020 under the Mont-Blanc Projects, grant agreement n 288777, 610402 and 671697 and the European HiPEAC Network.

References

[1] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[2] F. Sainz, S. Mateo, V. Beltran, J. L. Bosque, X. Martorell, and E. Ayguad´, "Leveraging ompss to exploit hardware accelerators," in *Int. Symp. on Computer Architecture and High Performance Computing*, Oct 2014, pp. 112–119.

[3] "Mercurium C/C++/Fortran source-to-source compiler," last accesed April 2017. [Online]. Available: https://github.com/bsc-pm/mcxx

[4] "Nanos++ Runtime Library," last accesed April 2017. [Online]. Available: https://github.com/bsc-pm/nanox

[5] E. Castillo, C. Camarero, A. Borrego, and J. L. Bosque, "Financial applications on multi-cpu and multi-gpu architectures," *J. Supercomput.*, vol. 71, no. 2, pp. 729–739, Feb. 2015.

[6] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent CPU-GPU Collaboration for Data-parallel Kernels on Heterogeneous Systems," in *Proc. of PACT*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 245–256.

[7] C. S. de la Lama, P. Toharia, J. L. Bosque, and O. D. Robles, "Static multi-device load balancing for opencl," in *Proc. of ISPA*. IEEE Computer Society, 2012, pp. 675–682.

[8] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 9:1–9:27, Aug. 2015.

[9] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-gpu platforms using functional performance models," *Computers, IEEE Trans. on*, vol. 64, no. 9, pp. 2506–2518, Sept 2015.

[10] T. Gautier, J. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IPDPS*, May 2013, pp. 1299–1308.

[11] A. Haidar, C. Cao, A. Yarkhan, P. Luszczek, S. Tomov, K. Kabir, and J. Dongarra, "Unified development for mixed multi-GPU and multi-coprocessor environments using a lightweight runtime environment," in *Proc. of IPDPS*, May 2014, pp. 491–500.

[12] A. Navarro, A. Vilches, F. Corbera, and R. Asenjo, "Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures," *J. Supercomput.*, vol. 70, no. 2, pp. 756–771, Nov. 2014.

[13] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM CGO*. ACM, 2014, p. 273:283.

[14] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An opencl framework for heterogeneous CPU/GPU clusters," in *Proceedings of the ACM ICS*. New York, NY, USA: ACM, 2012, pp. 341–352.

[15] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated GPUs," in *Proc. of PACT*. New York, NY, USA: ACM, 2014, pp. 151–162.

[16] M. Boyer, K. Skadron, S. Che, and N. Jayasena, "Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability," in *Proc. of the ACM Int. Conference on Computing Frontiers*. New York, NY, USA: ACM, 2013, pp. 21:1–21:10.

[17] T. Scogland, B. Rountree, W. chun Feng, and B. de Supinski, "Heterogeneous task scheduling for accelerated openmp," in *Proc. IPDPS*, May 2012, pp. 144–155.

[18] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures," in *2012 41st Int. Conf. on Parallel Processing*, Sept 2012, pp. 48–57.

[19] G. Wang and X. Ren, "Power-efficient work distribution method for cpu-gpu heterogeneous system," in *Int. Symp. on Parallel and Distributed Processing with Applications*, Sept 2010, pp. 122–129.

[20] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt, "Sparta: Runtime task allocation for energy efficient heterogeneous many-cores," in *Int. Conf. on Hardware/Software Codesign and System Synthesis*, ser. CODES '16. New York, NY, USA: ACM, 2016, pp. 27:1–27:10.

[21] A. L. R. Tupinambá and A. Sztajnberg, "Transparent and optimized distributed processing on gpus," *IEEE Trans. on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3673–3686, Dec 2016.

[22] J. Cabezas, I. Gelado, J. E. Stone, N. Navarro, D. B. Kirk, and W. m. Hwu, "Runtime and architecture support for efficient data exchange in multi-accelerator applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1405–1418, May 2015.

[23] Y.-P. You, H.-J. Wu, Y.-N. Tsai, and Y.-T. Chao, "Virtcl: A framework for OpenCL device abstraction and management," in *Principles and Practice of Parallel Programming*, ser. PPoPP 2015. ACM, 2015.

[24] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *CoRR*, vol. abs/1303.5164, 2013.

[25] A. M. Aji, A. J. Peña, P. Balaji, and W.-c. Feng, "Multicl: Enabling automatic scheduling for task-parallel workloads in opencl," *Parallel Comput.*, vol. 58, no. C, pp. 37–55, Oct. 2016.

[26] J. Kim, H. Kim, J. Lee, and J. Lee, "Achieving a single compute device image in OpenCL for multiple GPUs," in *Proc. of the ACM PPoPP*. ACM, 2011, pp. 277–287.

[27] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Self-adaptive ompss tasks in heterogeneous environments," in *2013 IEEE 27th Int. Symp. on Parallel and Distributed Processing*, May 2013, pp. 138–149.