Multiple Target Task Sharing Support for the OpenMP Accelerator Model

Guray Ozen[1,2], Sergi Mateo[1,2],

Eduard Ayguadé[1,2] Jesús Labarta[1,2] and James Beyer[3]

[1] Universitat Politècnica de Catalunya (UPC–BarcelonaTECH), Barcelona, Spain
[2] Barcelona Supercomputing Center (BSC-CNS), Barcelona, Spain
`name.surname@bsc.es`
[3] Nvidia Corporation
`jbeyer@nvidia.com`

Abstract. The use of GPU accelerators is becoming common in HPC platforms due to the their effective performance and energy efficiency. In addition, new generations of multicore processors are being designed with wider vector units and/or larger hardware thread counts, also contributing to the peak performance of the whole system. Although current directive–based paradigms, such as OpenMP or OpenACC, support both accelerators and multicore-based hosts, they do not provide an effective and efficient way to concurrently use them, usually resulting in accelerated programs in which the potential computational performance of the host is not exploited. In this paper we propose an extension to the OpenMP 4.5 directive-based programming model to support the specification and execution of multiple instances of task regions on different devices (i.e. accelerators in conjunction with the vector and heavily multithreaded capabilities in multicore processors). The compiler is responsible for the generation of device-specific code for each device kind, delegating to the runtime system the dynamic schedule of the tasks to the available devices. The new proposed clause conveys useful insight to guide the scheduler while keeping a clean, abstract and machine independent programmer interface. The potential of the proposal is analyzed in a prototype implementation in the OmpSs compiler and runtime infrastructure. Performance evaluation is done using three kernels (N-Body, tiled matrix multiply and Stream) on different GPU-capable systems based on ARM, Intel x86 and IBM Power8. From the evaluation we observe speed–ups in the 8-20% range compared to versions in which only the GPU is used, reaching 96% of the additional peak performance thanks to the reduction of data transfers and the benefits introduced by the OmpSs NUMA-aware scheduler.

1 Introduction

The use of accelerators has been gaining popularity in the last years due to their higher peak performance per watt ratio when compared to multicore-based multi-processors. A remarkable increase in the number of systems based on Nvidia and AMD GPUs and Xeon Phi co-processors can be observed in the Top500 supercomputers list. Besides, with the introduction of new ARM-based heterogeneous architectures, i.e. sockets that incorporate both general-purpose cores and specialized co-processors like GPUs, the potential applicability of these architectures seems evident in all forms of computing.

The programmability wall already risen by multicore architectures is even higher when heterogeneity needs to be considered. Programmers have to deal with different programming models, such as OpenMP [2], CUDA [8] or OpenCL [4], different vector intrinsics and use them with device specific optimization strategies, incurring portability and productivity issues. In addition the programmer may need to manage multiple memory address spaces and maintain their consistent view when necessary.

Existing directive-based programming models [9, 2] and their commercial or research implementations [5, 3] and variants [6, 10] attempt to lower the programmability wall by addressing some of these challenges. Nevertheless, all these proposals do not support offloaded task regions to be concurrently executed by multiple target devices (cores and accelerators) since the annotation of the same code-block for these different targets is not standard. Programmer has to manually transform the code, write multiple versions of these code-blocks and inject code to statically/dynamically offload them to the available devices. Usually this results in small performance benefits that do not compensate the programming effort devoted. This and the explicit management of data sharing over usually low bandwidth interfaces in discrete systems is favouring a programming style in which the bulk of the computation is performed on the accelerators and the general-purpose cores are just used to configure and manage the accelerators, remaining unused most of the time.

This paper contributes with an extension to the OpenMP 4.5 directive-based programming model to support multiple target device code generation and offloading. The proposal is partly inspired in the current `device_type` clause in OpenACC 2.5, the current capabilities of the OmpSs programming model and the internal discussions in the OpenMP accelerator model committee. The proposal has been included in the MACC [10] compiler, a research compiler for the accelerator model in the OmpSs programming model, and the new features in its runtime system to dynamically schedule tasks to all devices available in the system. In particular we propose new semantics to the already existing `device` clause, we adopt the conditional `if_device` construct to specify different thread hierarchies and clauses for different devices[4], and to add a new hint–to-the–runtime `resources` clause. The complete proposal allows the programmer to annotate offloadable task region for multiple different target devices, considering as devices both host cores as well as accelerators, and gives hints to the runtime system to appropriately schedule multiple instances of the task to the available devices, trying to balance the load assigned to them and taking into account data locality.

[4] Currently under discussion in the accelerators subcommittee of the OpenMP Language Committee.

2 Accelerator support in directive-based approaches

This section briefly describes the heterogeneity support currently provided by the OpenMP 4.5 specification to accelerators programming [2], the OpenACC 2.5 API [9]and the features supported by the MACC compiler for the OmpSs programming model [10].

2.1 Heterogeneity support in OpenMP accelerator model

With the aim of providing a smooth and portable path to program accelerator-based architectures, OpenMP 4.5 provides a programming interface based on a set of directives to offload the execution of code regions onto accelerators, to map loops inside those regions onto the resources available in the device architecture, and to map and move data between address spaces.

The main directive is `target`, which creates the data environment and offload the execution of a code region on an accelerator device. The `device` clause is used in these directives to specify the actual device that will be used to offload the execution of that code region; an integer-expression value sets the device to be used. The specification also brings the `teams` directive to create a thread hierarchy on the target device, with a league of thread teams. In each team the threads other than the master thread do not begin execution until the master thread encounters a `parallel` region. The `distribute` directive specifies how the iterations of one or more loops are distributed across the master threads of all teams that execute the teams region. In addition, OpenMP 4.5 includes the `simd` directive to indicate that a loop can be transformed so that multiple iterations of the loop can be executed concurrently using SIMD instructions.

In the current 4.5 specification there is no sufficient support to tailor the information provided in the `target` directive to different device types. The conditional `if([directive-name-modifier:] scalar-logical-expression)` clause can be used with `target` and in combined/composite constructs that include `target` and `parallel`. In `if(target:scalar-logical-expression)`, when the condition evaluates to false the target region is executed by the host device in the host data environment. In `if(parallel:scalar-logical-expression)`, the evaluation of the condition is used to determine the number of threads to use in the region.

2.2 Heterogeneity support in OpenACC

OpenACC provides directives that allow programmers to specify code regions to be offloaded to accelerator devices and to control many features of these devices explicitly. The main construct is `kernels`, instructing the compiler to optimize the annotated code region and exploit the available parallelism. OpenACC also offers the `parallel` and `loop` constructs for detailed control of kernel offloading and parallel execution of loops.

For these directives, the current 2.5 specification offers different clauses that the programmer use to specify alternative options for different accelerators using the `device_type` clause. The argument to the clause is a comma-separated list of one or more accelerator architecture name identifiers, or an asterisk. A single directive may have one or several `device_type` clauses. Clauses on a directive with no `device_type` clause apply to all accelerator device types. Clauses that follow a `device_type` clause up to the end of the directive or up to the next `device_type` clause are associated with this `device_type` clause. Clauses associated with a `device_type` clause apply only when compiling for the named accelerator device type. For each directive, only certain clauses may follow a `device_type` clause. For example, for `parallel` the programmer can specify `async`, `wait`, `num_gangs`, `num_workers`, and `vector_length`.

2.3 Heterogeneity support in OmpSs

The OmpSs task-based programming model offers the programmer a single linear address space on which the data lays. Directionality clauses for the tasks are specified as the mechanism to provide the runtime the information to compute data dependences between tasks and to enforce data consistency on systems with multiple address spaces by managing transfers between different address spaces transparently to the programmer.

In the initial versions, OmpSs relied on the use of CUDA and OpenCL for the specification of tasks to be executed on accelerator devices. The `target` construct was used to simply direct the compilation of the source code to the appropriate backend CUDA or OpenCL compiler and to inject the necessary code for task offloading, including data movement. The `device` clause in the `target` construct is used not to specify the specific device to execute the region but the device type than can execute it, delegating to the runtime system the decision of which device to use in case multiple devices of that type are available. Different versions for the same task, each one tailored to the device kind specified in the `device` clause, can be specified through the `implements` clause.

MACC Compiler. The MACC[5] compiler for the OmpSs programming model supports almost all directives of the OpenMP 4.5 accelerator model with some extensions to enhance code generation capability and minor semantic changes in order to allow their combination with other OmpSs directives.

The `target` construct is always associated to a `task` construct and the need for the OpenMP 4.5 `target data` and `target update` constructs is eliminated reducing programming complexity. The rest of device related OpenMP 4.5 constructs (`teams`, `distribute` and `parallel for`) are thoroughly implemented with the originally specified semantics. MACC adds to OpenMP 4.5 additional data sharing clauses for the `team` construct: `dist_private`, `dist_firstprivate` and `dist_lastprivate`; with the `chunk_size` provided in the `dist_schedule` clause, the compiler is able to make a better use of the memory hierarchy in the device (e.g. global, shared and local memory), allocating portions of arrays in different teams and performs the necessary data movement according

[5] **MACC** is an abbreviation for "**M**ercurium **ACC**elerator Compiler".

to `firstprivate` and `lastprivate` semantics. Also, MACC provides nesting of teams constructs and conditional clauses make the use of dynamic parallelism feature of GPU architecture that allows to launch a kernel within a kernel without CPU intervention [11]. It shows it is possible for the compiler to generate code that is then efficiently executed under dynamic runtime scheduling.

3 Proposal and implementation of multi target approach

In this section we propose an extension to the `target` directive to provide support for multiple device and conditional compilation. We also comment its implications in the implementation of the compiler and runtime system.

3.1 `Target` directive syntax extension

In this paper we propose to extend the `target` construct in three complementary ways: (i) by allowing the a use of the `any` keyword in the `device` clause, telling the compiler that the directive applies to more than one device type; (ii) by using new conditional construct `if_device` to provide different directives for different device types and (iii) with a new `resources` clause to give hints to the runtime system to appropriately balance the scheduling of tasks to the different devices in the system. Figure 1 shows the proposed syntax extension.

Multiple device targeting example

```
#pragma omp target device( any ) distribute parallel for
 for-loops
```

Multiple device targeting with conditional compilation example

```
#pragma omp target device( any )  [clause[ [,] clause] ... ]
#pragma omp
     if_device(archtype, subtype) construct-set resources(percentage-integer) \
     if_device(archtype, subtype) construct-set resources(percentage-integer)
 structured-block
```

Fig. 1: Usage of Conditional Multi Targeting support of OpenMP

The upper part in Figure 1 shows an example of use for the `any` keyword. When this keyword appears in the `device` clause, the compiler will have to generate multiple versions for the target region each one tailored to a different kind of device. In other cases, it is possible that the same directive set may not be appropriate for the different devices available. To that end we also propose the `if_device` construct, allowing the programmer to specify different directive set for different devices. The lower part of Figure 1 shows the use of this construct. In the `construct-set` the programmer can specify for the same code block different thread hierarchies and clauses according to the OpenMP 4.5 accelerator model, each one tailored to a device kind. At execution time the runtime system will decide among the different implementations provided the one that is more appropriate for each particular instance of the task to be offloaded.

Finally we also propose the `resources` clause, providing a hint to help the runtime system to decide where to schedule the execution of the next task instance. For each device the programmer specifies a value (or expression) over 100 which indicates the amount of "tokens" consumed every time a task is scheduled on that device; once a task finishes its execution, that number of tokens is restored. If at any time the number of tokens available is not sufficient, the runtime will not be able to schedule the task to that device, choosing a different device that requires less resources if possible.

A more specific example illustrating the use of the proposed extensions is shown in the upper part in Figure 2; the code corresponds to the main loop of the N-Body simulation kernel that is used later in the evaluation section. In this example, the programmer specifies in line 4 that the target region can be executed in different device types. Device specific directives are specified in lines 5 and 6 for Nvidia accelerators with compute capability 3.5 and for host, respectively. In addition, the programmer specifies the number of tokens consumed/released every time a task is scheduled to execute or finishes its execution on every possible device: 2 token (over 100) when task is offloaded to the Nvidia accelerator or 40 tokens (over 100) when executed in the host. In this case the programmer expresses that no more than 2 tasks should be scheduled to be executed in the host at any time. These combination can be estimated considering separate performance of multiple targets after some initial experiments, or could be auto-tuned by the runtime if `auto` is used instead of an integer number. These two aspects are not considered in this paper and are subject of current research.

3.2 Compiler and runtime support to the proposed extensions

Figure 3 shows the compilation pipeline in the MACC compiler once the Mercurium Intermediate Representation (IR) has been generated. Different device-specific IR lowering phases can be implemented, each one either transforming the IR (e.g. for the `host` device by inserting the appropriate calls to the OmpSs runtime system) or generating an output file to be compiled by a device-specific native compiler (e.g. CUDA for the `NVIDIA` device). The *multiple device dispatcher* unit is in charge of forwarding a new copy of the IR for each device type in the list of devices in the `device` clause to the appropriate lowering phase. The implementation is extensible as shown with dotted lines and the *nextgen_device* lowering phase. At the end of the compilation pipeline, the compiler driver compiles each output file with appropriate back-end compiler and links object files to generate the final executable file.

Before the execution of the *Host lowering* phase it can be necessary to execute a `host IR reducer` phase when one set of directives are used for all devices. This phase is in charge of adapting the thread hierarchy supported by the OpenMP 4.5 accelerator model (teams and threads) to the flat thread model. This step basically selects the outermost loop affected with a `distribute` or `parallel for` directive and transforms it into a `parallel for` directive. Other directives in the `target` region are ignored for host device, except for `simd` constructs which are then lowered to specific SIMD operations in the host.

All-Pairs N-Body Simulation with $\mathcal{O}(n^2)$ Complexity

```
//N-Body Computation
for (int k = 0; k < n; k+=BS)
{
 #pragma omp target device(any) map(tofrom:vx[k:BS],vy[k:BS],vz[k:BS]) nowait
 #pragma omp if_device(NVIDIA,cc35) teams distribute parallel for resources(2)
 #pragma omp if_device(host) parallel for resources(40)
 for (int i = k; i < k+BS; ++i) {
   float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

   #pragma omp simd reduction(+:Fx,Fy,Fz)
   for (int j = 0; j < n; j++) {
    float dy = y[j] - y[i];
    float dz = z[j] - z[i];
    float dx = x[j] - x[i];
    float distSqr = dx*dx + dy*dy + dz*dz+CONST;
    float invDist = 1.0f / sqrtf(distSqr);
    float invDist3 = invDist * invDist * invDist;
    Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
  }
  vx[i] += dt*Fx;  vy[i] += dt*Fy;  vz[i] += dt*Fz;
 }
}
```

Transformed NVIDIA-Task Code

```
#pragma omp teams distribute \
        parallel for resources(2)
for (int i ...) {
   for (int j ...) {
      ...
   }
}
```

Transformed Host-Task Code

```
#pragma omp parallel for resources(40)
for (int i ...) {
   #pragma omp simd [ clause ...]
   for (int j ...) {
      ...
   }
}
```

Fig. 2: N-Body example of MACC IR Code Transformation

The lower part in Figure 2 shows how the directives are interpreted for each device type in order to adapt the generic thread hierarchy to each specific device: `NVIDIA` on the left and `host` on the right.

3.3 Compiler and Runtime Support for the `resources` Clause

For the `resources` clause, the compiler just parses the two fields for each device kind and pass this information to the runtime system though an internal runtime call. This information is used by the runtime system to account for the total number of resource ”tokens” available at any time. When a task is ready for execution, the runtime checks if enough tokens are available for any of the possible target devices; if so, then the runtime subtracts the specified resource tokens for the selected device from the currently available tokens. When the task finishes its execution, the runtime adds the same amount of tokens to the total count. Both operations are done using *atomic* operations.

4 Evaluation

In this section we present the performance evaluation of the multiple targeted task-sharing proposal and its implementation in the MACC compiler and OmpSs

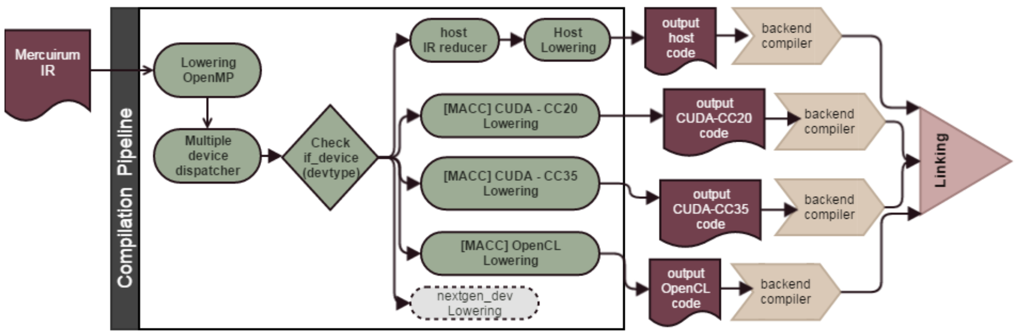

Fig. 3: Overview of device dispatcher and IR lowering units.

runtime system. To that end we use a variety of system configurations and three small kernel applications: N-Body, tiled matrix multiply and the Stream benchmark.

4.1 System configurations

Table 1 shows the main characteristics of the four systems that have been used for the experimental evaluation of the proposal in this paper. The different system configurations offer different ratios between the performance of the host and the performance of the accelerator devices.

The 1st system is based on an old generation of Nvidia GPUs (Fermi architecture) while the 2nd and 3rd systems are based on a more recent Nvidia GPU (Tesla K40). The first two systems are based on Intel hosts while the 3rd system is based on the emergent IBM Power8 architecture with high memory bandwidth and increased hardware thread counts. Finally the 4th system is based on ARM SoC with a tiny GPU which just includes one Streaming Multiprocessor Architecture (SMX).

System	Processor	Memory	Nvidia GPU
1	*2 x Intel Xeon(TM) E5649 sockets* *6-core/socket at 2.53GHz*	*24 GB*	*2 x Tesla M2090* *(Fermi, 512 cores)*
2	*1 x Intel Core(TM) i7-4820K socket* *4-core/socket, 2-hw threads/core at 3.70GHz*	*64 GB*	*2 x Tesla K40c* *(Kepler, 2880 cores)*
3	*2 x IBM Power S824L sockets* *12-core/socket, 8-hw threads/core at 3.52 GHz*	*1 TB*	*2 x Tesla K40m* *(Kepler, 2880 cores)*
4	*Nvidia Jetson TK1 SoC* *4-core Cortex-A15 up to 2.5GHz*	*2 GB*	*1 x GK20A* *(Kepler, 192 cores)*

Table 1: System configurations

All CUDA codes in this paper have been automatically generated by the MACC compiler and compiled with *nvcc* v7.0, except for the 4th system which makes use of v6.0. GCC 4.9 is used to compile host codes on all systems with -O3 optimization level. The `simd` construct in OpenMP 4.5 and auto-vectorization is performed by back-end GCC compiler.

4.2 OmpSs runtime configurations and thread binding

The OmpSs runtime is used to support the execution of work-sharing and tasking constructs. In addition, the OmpSs runtime manages host/GPU data transfers and concurrent kernel execution and CUDA streams. To that end OmpSs reserves a helper thread in the socket for each GPU device attached to it; the rest of threads are used to execute `host` tasks. The execution of `host` target regions is assigned to sockets in a round-robin way and work-sharing constructs inside an `host` target region are bound to the threads in a single socket,

For the 3rd system based on IBM Power8 processors we have activated the NUMA-aware scheduler feature in the OmpSs runtime. The runtime detects the socket architecture of the system and binds threads properly, distributing tasks according to the memory layout. Besides that, in order to investigate the effect of multithreading inside a core, we adjust the OmpSs thread binding (using an environment variable) to use 1, 2, 4 or 8 threads per core.

4.3 Performance results

N-Body. This kernel computes the motion of a set of bodies based on the forces between them. For this simulation, an all-pairs algorithm is used with $\mathcal{O}(n^2)$ complexity, as shown in the upper code in Figure 2. The `resources` values have been set to maximize load balancing between tasks executed on the host processors and the GPU devices.

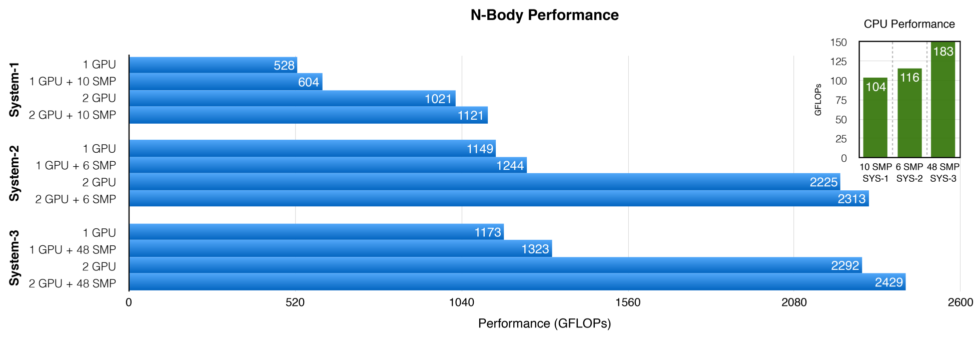

Fig. 4: N-Body simulation performance results.

Figure 4 shows the performance results obtained for the N-Body kernel. The plot on the top-right corner shows the GFLOPs achieved when using the cores in the host for the three first system configurations. The main plot in the same figure shows how much of that performance is actually contributed to the overall performance when using one and two GPUs, observing increases in the 8%-14% and 4%-10% ranges, respectively. This contributed performance is very close to the ideal performance which could be obtained by just adding the performance of the CPU to the GPU.

Finally, the performance of the N-Body kernel has also been evaluated on the 4th system based on the Jetson TK1. The left plot in the Figure 5 shows three different results: CPU only, GPU only and combined CPU/GPU. In this case, the performance benefit is up to 20% due to the relatively close performance of the ARM cores and the small GPU in the SoC.

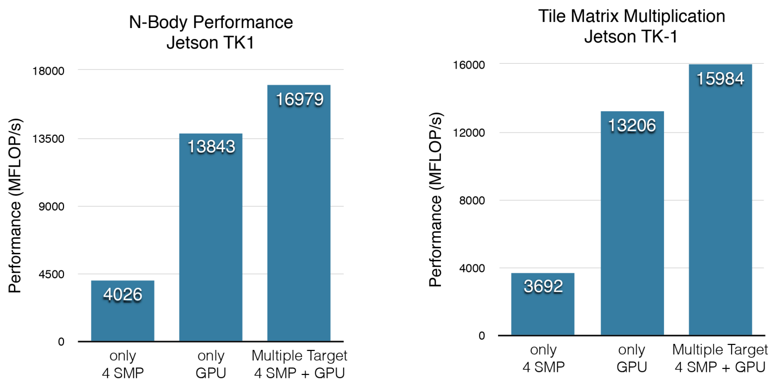

Fig. 5: N-Body and tiled-gemm performance on Jetson TK1.

Tiled matrix multiply. The kernel performs a dense matrix multiplication of two square matrices $A \times B = C$. Matrices are divided in blocks and each task is responsible for the computation of one of such blocks of the output matrix C. The matrix size is used 8192x8192 double-precision floating-point elements with 512x512 block size.

The matrix multiply kernel is written using six nested loops: the three innermost ones are annotated with MACC directives for multiple target devices. The MACC compiler transformed them into non-optimized CUDA code (current implementation lacks of many optimization phases that would be necessary to generate an optimized kernel) and highly optimized using expensive optimization features of back-end compiler for the host.

The performance for this kernel has been evaluated on the Nvidia Jetson TK1 (right plot in Figure 5) and IBM Power8 (Figure 6) platforms. For the TK1 system we can conclude that the work was shared among the entire SoC elements, with the cores being able to contribute to the performance of the GPU.

For the Power8 system, the plot on the top-right corner shows the performance that is obtained when using different numbers different numbers of SMT threads. The main plot in that figure then shows how this performance is contributed to the hybrid system, observing performance increases of 30% and 16% when one and two GPUs are used, respectively. Observe that the best result is obtained when two SMT threads per core are activated, since each Power8 core includes two vector unit and load/store unit.

STREAM. This code [7] is commonly used to benchmark the memory bandwidth. It consists of four micro-benchmarks accessing three vectors `a`, `b` and `c` and a scalar variable, inside an iterative loop that repeats their execution a

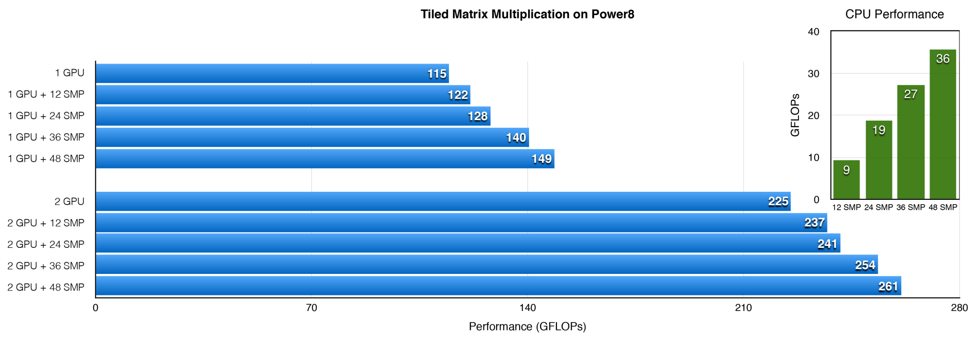

Fig. 6: Matrix multiplication performance results

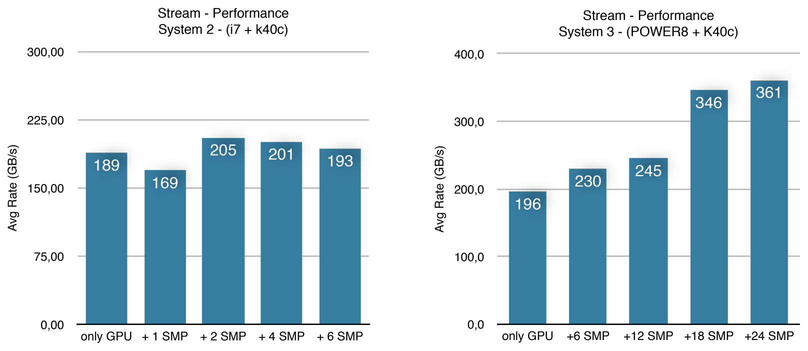

Fig. 7: Stream bandwidth performance avg rate (GB/s)

number of times. Loop tiling has been applied to the outermost loop in these four operations in order to divide the iterations into multiple tasks and to run them in parallel. Task dependencies are specified between the tasks computing the four different operations.

This benchmark is evaluated using the 2nd and 3rd system. Bot use the same Nvidia GPU (with reported memory bandwidth of 288 GB/s). However, the processors in the 2nd system report a memory bandwidth of 59.7 GB/s while the processors in the 3rd system report average 192 GB/s (with a maximum of 275GB/s on the individual micro benchmarks [1]). Therefore, comparing these two systems provides a good opportunity to see how the runtime is able to fully exploit the additional bandwidth in the Power8 system.

Figure 7 shows the average bandwidth reported by the Stream benchmark when different numbers of host threads that are called also SMP workers are used, for both systems evaluated. For the Power8 system (right plot), an speed up to 84% over GPU baseline is achieved when using all the cores in the entire system. For the i7-based system (left plot), the best performance is achieved when only two cores are used, showing the memory bandwidth bottleneck of the socket. When GPU tasks are finished, the runtime steals tasks which were initially assigned to the CPU, forcing the runtime to copy data from host to device.

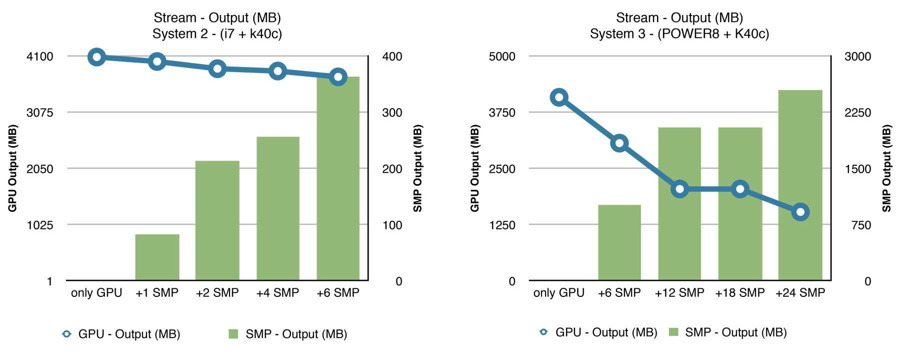

Fig. 8: Output data size of stream benchmark

Finally, Figure 8 shows the total amount of data that is produced by the different devices. The largest amount of data produced by the CPU is achieved when 24 cores are used on the Power8 system.

5 Conclusions and future work

In this paper we have proposed an extension to directive-based programming models to support the possibility of allowing the execution (of multiple instances) of a `target` region on different devices in an heterogenous architecture. We have also analyzed its implementation in the OmpSs compiler and runtime system and evaluated its performance for a variety of system configurations and some kernel applications. The proposed extensions ease the use of multiple accelerators in conjunction with the vector and heavily multithreaded capabilities in multicore processors without any code modification. The new proposed construct and clause convey useful insight to guide the scheduler while keeping a clean, abstract and machine independent programmer interface. The performance evaluation shows that with the new resources-based scheduler the runtime is able to take benefit of all devices available in the heterogeneous system.

As part of our current work, we plan to extend our implementation to support other architectures, like Xeon Phi and FPGA devices. Then, we will also need to tune our scheduling parameters to fit all of them. We are also investigating the use of nested (dynamic in CUDA terminology) parallelism in `target` regions [11] and how the proposed extensions in this paper interact with them.

6 Acknowledgments

This work is partially supported by the IBM/BSC Deep Learning Center Initiative, by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology through TIN2015-65316-P project and by the Generalitat de Catalunya (contract 2014-SGR-1051).

References

[1] Andrew V. Adinetz, Paul F. Baumeister, Hans Böttiger, Thorsten Hater, Thilo Maurer, Dirk Pleiter, Wolfram Schenck, and Sebastiano Fabio Schifano. Performance evaluation of scientific applications on POWER8. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation - 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers*, pages 24–45, 2014.

[2] OpenMP ARB. OpenMP application program interface, v. 4.5. 2015.

[3] Carlo Bertolli, Samuel F. Antao, Alexandre E. Eichenberger, Kevin O'Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. Coordinating GPU threads for OpenMP 4.0 in LLVM. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 12–21, Piscataway, NJ, USA, 2014. IEEE Press.

[4] Khronos OpenCL Working Group. The OpenCL specification, version 2.0. 2014.

[5] The Portland Group. PGI accelerator compilers.

[6] Seyong Lee and Jeffrey S. Vetter. OpenARC: Open Accelerator Research Compiler for directive-based, efficient heterogeneous computing. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, pages 115–120, 2014.

[7] J.D McCalpin. Stream: Sustainable memory bandwidth in high performance computers. *Technical report, University of Virginia*, 2007.

[8] NVIDIA. CUDA C programming guide version 7.0. *NVIDIA Corporation*, 2013.

[9] OpenACC-Standard.org. OpenACC application programming interface, v. 2.5. 2015.

[10] Guray Ozen, Eduard Ayguadé, and Jesús Labarta. On the roles of the programmer, the compiler and the runtime system when programming accelerators in OpenMP. In *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings*, pages 215–229, 2014.

[11] Guray Ozen, Eduard Ayguadé, and Jesús Labarta. Exploring dynamic parallelism in OpenMP. In *Proceedings of the Second Workshop on Accelerator Programming using Directives, WACCPD 2015, Austin, Texas, USA, November 15, 2015*, pages 5:1–5:8, 2015.