On the roles of the programmer, the compiler and the runtime system when programming accelerators in OpenMP

Guray Ozen, Eduard Ayguadé and Jesús Labarta

Barcelona Supercomputing Center (BSC-CNS), Barcelona, Spain
Universitat Politècnica de Catalunya (UPC–BarcelonaTECH)

Abstract. *OpenMP* includes in its latest 4.0 specification the accelerator model. In this paper we present a partial implementation of this specification in the OmpSs programming model developed at the Barcelona Supercomputing Center with the aim of identifying which should be the roles of the programmer, the compiler and the runtime system in order to facilitate the asynchronous execution of tasks in architectures with multiple accelerator devices and processors. The design of OmpSs is highly biassed to delegate most of the decisions to the runtime system, which based on the task graph built at runtime (*depend* clauses) is able to schedule tasks in a data flow way to the available processors and accelerator devices and orchestrate data transfers and reuse among multiple address spaces. For this reason our implementation is partial, just considering from 4.0 those directives that enable the compiler the generation of the so called "kernels" to be executed on the target device. Several extensions to the current specification are also presented, such as the specification of tasks in "native" CUDA and OpenCL or how to specify the device and data privatization in the *target* construct. Finally, the paper also discusses some challenges found in code generation and a preliminary performance evaluation with some kernel applications.

Keywords: OpenMP accelerator model, OmpSs, OpenCL, CUDA

1 Introduction

The use of accelerators has been gaining popularity in the last years due to their higher peak performance and performance per Watt ratio when compared to homogeneous architectures based on multicores. However, the heterogeneity they introduce (in terms of computing devices and address spaces) makes programming a difficult task even for expert programmers.

Some alternatives have been proposed to address the programmability of these accelerator–based systems. CUDA [1] and OpenCL [2] provide low-level API's that allow computation to be offloaded to accelerators. the management of their memory hierarchy and the data transfers between address spaces. Other alternatives, such as OpenACC [3], have appeared with the aim of providing a higher–level directive–based approach to program accelerator devices. OpenMP

[4] also includes in its latest 4.0 specification the accelerator model with the same objective. These solutions based on directives still rely on the programmer for the specification of data regions, transfers between address spaces and for the specification of the computation to be offloaded in the devices; these solutions also put a lot of pressure on the compiler–side that has the responsibility of generating efficient code based on the information provided by the programmer.

The OmpSs [5] proposal has been evolving during the last decade to lower the programmability wall raised by multi–/many–cores, demonstrating a task–based data flow approach in which offloading tasks to different number and kinds of devices, as well as managing the coherence of data in multiple address spaces, is delegated to the runtime system. Multiple implementations were investigated for the IBM Cell (CellSs [6]), NVIDIA GPU (GPUSs [7]) and homogeneous multicores (SMPSs [8]) before arriving to the current unified OmpSs specification and implementation. Initially OmpSs relied on the use of CUDA and OpenCL to specify the computational kernels. This paper presents the latest implementation of OmpSs which includes partial support for the accelerator model in OpenMP 4.0 specification. We just adopted those functionalities that are necessary to specify computational kernels in a more productive way. The paper analyzes the roles of the programmer, the compiler and the runtime from this new OmpSs perspective.

2   "Pure" accelerator-specific programming

"Pure" accelerator-specific programming initially put all responsibility in the programmer, who should take case of transforming computational intensive pieces of code into kernels to be executed on the accelerator devices and write the host code to orchestrate data allocations, data transfers and kernel invocations with the appropriate allocation of GPU resources. Nvidia CUDA [1] and OpenCL [2] are the two APIs commonly used today.

In favor of programmability, the latest releases of the Nvidia CUDA architecture improved programming productivity by moving some of the burden to the CUDA runtime, including Unified Virtual Addressing (CUDA 4) to provide a single virtual memory address space for all memory in the system (enabling pointers to be accessed from GPU) no matter where in the system they reside) and Unified Memory (CUDA 6) to automatically migrate data at the level of individual pages between host and devices, freeing programmers from the need of allocating and copying device memory. Although these additions may be seen as a need for beginners, they make it possible to share complex data structures and eliminate the need to handle "deep copies" in the presence of pointed data inside structured data. Carefully tuned CUDA codes may still use streams and asynchronous transfers to efficiently overlap computation with data movement when the CUDA runtime is unable to do it appropriately due to lack of lookahead.

3   Directive-based approaches in OpenMP and OpenACC

With the aim of providing a smooth and portable path to program accelerator-based architectures, OpenACC [3] and OpenMP 4.0 [4] provide a directive-centric programming interface. Directives allow the programmer to specify code regions to be offloaded to accelerators, how to map loops inside those regions onto the resources available in the device architecture, the data mapping in their memory and data copying between address spaces. This directive–based approach imposes a high responsibility on the compiler that needs to be able to generate optimized device–specific kernels (considering architectural aspects such as the memory hierarchy or the amount of resources available) as well as taking care of accelerator startup and shutdown, code offloading and implementing data allocations/transfers between the host and accelerator[1].

The directive–based approach frees the programmer from the need to write accelerator-specific code for the target device (e.g. CUDA or OpenCL kernels). We think this is important in terms of programming productivity, but we also believe that the directive–based approach should allow a migration path for existing CUDA applications by reusing device–specific OpenCL or CUDA kernels already optimized by experienced programmers.

In the following subsections we briefly summarize OpenMP 4.0 and OpenACC constructs with the aim of splitting responsibilities between the compiler and the runtime system, with the overall objective of lowering the programmability wall.

3.1   Offloading, kernel configuration and loop execution

OpenMP 4.0 offers the *target* directive to start parallel execution on the accelerator device. Similarly, OpenACC offers the *parallel* directive with the same purpose. In OpenACC these regions can be declared asynchronous removing the implicit barrier at the end of the accelerator parallel region, allowing the host to continue with the code following the region.

Inside these accelerator regions in the OpenMP 4.0, the programmer can specify *teams*, representing a hierarchy of resources in the accelerator: a league of *num_teams* teams, each with *thread_limit* threads. The execution in the *teams* region initially starts in the master thread of each team. Later, the *distribute* and *parallel for* directives can be used to specify the mapping of iterations in different loops to the resources available on the accelerator. On the other hand, OpenACC offers *kernels*, i.e. regions that the compiler should translate into a sequence of kernels for execution on the accelerator device. Typically, each loop nest will be a distinct kernel. OpenACC also includes the *loop* directive to describe what type of parallelism to use to execute a loop and declare loop-private variables and arrays and reduction operations. Inside kernels and loops resources are organized in gangs, workers and vectors (indicated with the *num_gangs*, *num_workers* and

---

[1] If the accelerator can access the host memory directly, the implementation may avoid this data allocation/movement and simply use the host memory.

vector_length clauses, respectively), similar to the teams, threads and SIMD in OpenMP 4.0.

Figure 1 shows the use of the above mentioned directives and clauses in OpenMP 4.0. Lines 7 and 11 in the code on the left specify the mapping of iterations of the i and j loops among 16 teams and 512 threads inside each team, respectively as declared in lines 5–6. Similarly, line 6 in the code on the right informs the compiler to map iterations of the i loop to both teams and threads inside each team; lines 8 and 13 also map iterations of the j loop to threads, probably using the multidimensional organization available in current accelerator devices.

```
1 #define n 128                            1 #define nX 4
2 #define m 10240                          2 #define nelem 12000
3                                          3
4 #pragma omp target device(0)            4 #pragma omp target device(0)
5 #pragma omp teams                       5 #pragma omp teams thread_limit(64)
6      num_teams(16) thread_limit(512)    6 #pragma omp distribute parallel for
7 #pragma omp distribute                  7 for (i=0; i < nelem; i++) {
8 for (i = 0; i < n; i++)                 8     #pragma omp parallel for private(k)
9 {                                       9     for (j=0; j < nX*nX; j++)
10                                        10         for (k=0; k < nX; k++)
11      #pragma omp parallel for          11             // loop body
12      for (j = 0; j < m ; j++)          12     #pragma omp parallel for
13          // loop body                  13     for (j=0; j < nX*nX; j++)
14                                        14         // loop body
15 }                                      15 }
```

Fig. 1. Two simple examples using OpenMP 4.0 directives for offloading.

Figure 2 shows another OpenMP 4.0 code where the programmer defines the thread hierarchy (line 9) and maps to it the execution of the loop in line 10. The *target* region is inside a *task*, so in this case the execution in the device is asynchronous to the execution of the master thread in the processor.

```
1 for(begin=0 ; begin < n; begin+=stride)
2 {
3     int end = begin + stride − 1;
4     int dev_id = (begin / stride) % omp_get_num_devices();
5
6     #pragma omp task
7     #pragma omp target device(dev_id) \
8             map(to: y[begin:end], x[begin:end]) map(from: z[begin:end])
9     #pragma omp teams num_teams(16) thread_limit(32)
10    #pragma omp distribute parallel for
11    for(i = 0; i < stride; ++i)
12        z[i] = a * x[i] + y[i];
13 }
```

Fig. 2. Code in OpenMP 4.0 asynchronously offloading to multiple accelerator devices.

The *target* directive in OpenMP 4.0 includes the *device* clause, which offloads the execution of the region kernel to a given physical device (indicated by the integer value in the clause). This direct mapping makes it difficult to write applications that dynamically offload work to accelerators in order to achieve load balancing or adapt to device variability, since it forces the programmer to embed in the application logic code to manage resources.

For example the code in Figure 2 shows how the programmer could statically map consecutive *target* regions to the accelerators available in the target architecture (line 4 to compute the device identifier and *device* clause in line 7),

allowing in this case to use two devices. Observe that the iteration range for the *for* loop at Line 11 goes from 0 to *stride*, so the program is not "sequentially equivalent" since it should iterate from *begin* to *end*. This is how OpenMP 4.0 forces the specification of the work to be offloaded; we assume that this has to be done in this way in order to ease code generation by the compiler although at the expenses of reducing code portability and reusability, in addition to potential programming errors.

3.2   Data motion

Data copying clauses may appear on the *target* construct in OpenMP 4.0 and *parallel* and *kernels* constructs in OpenACC. With these clauses the programmer specifies the data motion needed to bring in and out the data for the execution of the region in the accelerator.

For the data items (including array regions) that appear in an OpenMP 4.0 *map* clause, corresponding new data items are created in the device data environment associated with the construct. Each data item has an associated map type which specifies the data copying on entry and exit (*to*, *from* or *tofrom*) or just allocation (*alloc*). OpenACC offers similar clauses (*copyin*, *copyout*, *copy* and *create*). With all this information, the compiler schedules the associated data allocations and transfers accordingly.

The example in Figure 2 shows the use of the *map* clause in Line 8. It is important to notice that *map(to: ...)* forces the movement of data when the *target* region is found; similarly for *map(from: ...)* which copies from device to host when the *target* region finishes.

Both OpenMP 4.0 and OpenACC offer the possibility of defining data environments in the accelerator for the extent of the region: *target data* and *data*, respectively. Inside these regions, multiple kernel offloading actions may occur without data copying between them, unless explicitly specified. An executable directive (*target update* in OpenMP 4.0 and *update* in OpenACC) is offered to the programmer to update, inside the scope of a *data* region, the data from the host to the device or vice-versa. The use of multiple accelerators within a *target data* region is not clear since at most one *device* clause can appear in the *target data* directive. The Jacobi code in Figure 6 shows the use of *data* regions and *update* in an OpenACC example.

3.3   Memory hierarchy in the accelerator device

Private, *firstprivate* and *reduction* clauses in *distribute* and *parallel for* directives give the compiler hints about the use of the memory hierarchy inside the accelerator. Again, OpenMP 4.0 and OpenACC rely on the programmer and the compiler for the management of the memory hierarchy, having a direct impact in the quality of the kernel codes to be executed on the accelerator device.

4   Accelerator support in OmpSs

The accelerator support in the OmpSs programming model [5] leverages the tasking model with data directionality annotations already available in the model (that influenced the new *depend* clause in OpenMP 4.0). These annotations are used by the OmpSs runtime system to dynamically compute task dependences and build a dependence task graph. This graph is used to dynamically schedule tasks in a data–flow way conscious of the resources available at any time.

The OmpSs programming model offers *target* directive with the following syntax:

```
#pragma omp target [clauses]
    task construct | function definition | function header
```

where clauses specify:

- copy_in, copy_out and copy_inout - shared data that needs to be available in the device before the construct can be executed or available after the construct is executed.
- copydeps - copy semantics for the directionality clauses in the associated *task* construct.
- device - kind of devices that can execute the construct (smp, cuda, opencl or acc).
- implements - an alternative implementation of the function whose name specified in the clause for a specific kind of device.

In order to make hybrid with native CUDA and/or OpenCL kernels, the directive includes two additional clauses:

- ndrange - specification of the dimensionality, iteration space and blocking size to replicate the execution of the CUDA or OpenCL kernel.
- shmem - arguments (and their size) to be mapped into team shared–memory.

The *copy_in*, *copy_out* and *copy_inout* clauses, together with the lookahead provided by the availability of the task graph, are used by the runtime system to schedule data copying actions between address spaces (movements between host and accelerator or between two accelerator devices if needed). *Copydeps* is a simple shorthand to reuse the directionality annotations in the *task* directive.

Figure 3 shows a simple example based on SAXPY. In this example, the task computing *saxpy* is written as a CUDA kernel and offloaded to a device with CUDA architecture; task *check_results* is defined to be executed in the host. Observe that the output of CUDA task instances are inputs of the host task instances. The dependences computed at runtime will honor these dependences and the runtime system will take care of doing the data copying operations based on the information contained in the task graph (dynamically generated at runtime). The *ndrange* clause is used to replicate the execution of the CUDA kernel in the device block/thread hierarchy (one dimension with $na*na$ iterations in total to distribute among teams of na iterations in this example).

```
1 #pragma omp target device(cuda) ndrange(1, na*na , na) copy_deps
2 #pragma omp task in(a[0:stride] , b[0:stride]) out(c[0:stride])
3 __global__ void saxpy (double * a, double * b, double * c, int stride) {
4    // CUDA kernel code
5 }
6 #pragma omp target device(smp) copy_deps implements(saxpy)
7 #pragma omp task in(a[0:stride] , b[0:stride]) out(c[0:stride])
8 void saxpy_smp (double * a, double * b, double * c, int stride) {
9    // CPU code with OpenMP directives
10 }
11 #pragma omp target device(smp) copy_deps
12 #pragma omp task in(static_correct_result[0:stride], c[0:stride])
13 void check_results ( double * precalc_result, double *  c, int stride) {
14    // CPU codes with OpenMP directives
15 }
16 int main (int argc, char ** argv) {
17    double a[SIZE], b[SIZE], c[SIZE];
18
19    for (begin=0 ; begin < nX ; begin+=stride)
20       saxpy(&a[begin], &b[begin], &c[begin], stride);
21
22    for (begin=0 ; begin < nX ; begin+=stride)
23       check_results(&precalc_result[begin], &c[begin], SIZE);
24 }
```

Fig. 3. Heterogeneous task example with OmpSs

With the *device* clause the programmer informs the compiler and runtime system about the kind of device that can execute the task, not an integer number that explicitly maps the offloading to a certain device as done in OpenMP 4.0. This is a big difference that improves programming productivity when targeting systems with different number and type of accelerators and regular cores. The code in Figure 3 could be executed on any number of devices without changes.

The *acc* device type is used to specify that the task will make use of OpenMP 4.0 directives to specify what to execute on the accelerator device, relying on the compiler to generate the kernel code to be executed on the device. We will describe in more detail the current compiler implementation in Section 5.

Multiple implementations tailored to different accelerators/cores can be specified for the same task (currently only available for tasks that are specified at the function declaration/definition). In this case, the programmer is delegating to the runtime system the responsibility of dynamically selecting the most appropriate device/core to execute each task instance, for example based on the availability of resources or the availability of the data needed to execute the task in the device (locality–aware scheduling). With the *implements* clause the programmer can indicate alternative implementations for a task function tailored to different devices (accelerator or host). Figure 3 shows the use of the *implements* clause: the *saxpy_smp* function in Line 6 is defined as an alternative implementation to the CUDA implementation of *saxpy* at Line 3. Observe that the programmer simply invokes *saxpy* in Line 20, delegating in the runtime the selection of the most appropriate implementation for each task instance.

5   MACC compiler

A new compilation phase (MACC[2]) has been included in the Mercurium [9] compiler supporting the OpenMP 4.0 accelerator model with the OmpSs runtime. MACC takes care of kernel configuration, loop scheduling and appropriate use

---

[2] MACC is abbreviation for "Mercurium Accelerator Compiler".

of the memory hierarchy for those tasks whose *device* is set to *acc* in the *target* clause. Some of the OpenMP 4.0 directives for accelerators (*target data target update* directives and *map* clause) are simply ignored because we delegate their functionality to the runtime system. Others have been extended to better map with the OmpSs model or to provide additional functionalities.

OpenMP 4.0	MACC	OpenMP 4.0	MACC
target	extended (*implements*, *ndrange* for CUDA and OpenCL kernels)	device(*int*)	extended
map(*to/from/tofrom*)	implemented but different names (copy_in/out/inout)	distribute	new clauses *dist_private*
map(*alloc*)	ignored	teams	implemented
target data	currently ignored	parallel for	implemented
target update	ignored	distribute parallel for	implemented

Fig. 4. MACC vs OpenMP 4.0

5.1   Kernel configuration, loop scheduling and thread mapping

When generating kernel code MACC needs to decide: 1) the dimensionality of the resources hierarchy (one-, two- or three-dimension teams and threads) and 2) the size in each dimension (number of teams and threads). In order to support the organization of the threads in multiple dimensions MACC allows the nesting of *parallel for* directives inside a *target* region (dimensionality equals the nesting degree). Other proposals considered the use of *collapse* which includes an integer to specify the number of nested loops with the same purpose [17].

MACC takes into account the restrictions of the device (for example maximum number of blocks and threads warp size in the CUDA computing capability) and the information provided by the programmer in the *num_teams* or *max_threads* clauses; if not specified, MACC simply assigns one iteration per block and one iteration per thread. MACC currently generates one dimensional teams (the current implementation does not support nesting of *distribute* directives). Thread dimensions are initially assigned in loop nesting order. As we will see in the experimental section[3], this ordering (for example outer loop for second thread dimension and inner loop for first thread dimension) may have a noticeable impact in performance; for now this is the responsibility of the compiler with no hints from the programmer in the current OpenMP 4.0 specification.

5.2   Coalesced accesses and use of shared memory

MACC code generation makes use of coalesced accesses to access global memory in warps. To that end MACC performs a cyclic mapping of loop iterations

---

[3] **opt3** in the experimental evaluation of DG-kernel in Section 6.

and tries to eliminate redundant "one–iteration" loops and simplifies increment expressions for induction variables in order to improve kernel execution time[4].

MACC also makes use of shared memory for threads in a team based on the specification of *private* and *firstprivate* data structures in the *distribute* directive, so that each team allocates a private copy in its own shared memory. MACC analyzes the size of the data structure to be privatized and generates code for its allocation and copying from global memory to shared memory in each team. However, for very large private arrays this is not possible to apply. For these cases we have implemented 3 new clauses (*dist_private*, *dist_firstprivate* and *dist_lastprivate*); with these clauses and the chunk size provided in the *dist_schedule(static,chunk_size)* clause in the *distribute* directive or near by array variable the compiler just allocates a portion of the arrays to each team and performs the necessary copies according to the *firstprivate* and *lastprivate* semantics[5].

– *dist_private(list)* : shared memory is only allocated up to indicated *chunk_size*.
– *dist_firstprivate(list)* : shared memory is allocated up to indicated *chunk_size* and it is filled with own part of array at global memory.
– *dist_lastprivate(list)* : shared memory is allocated up to indicated *chunk_size*. End of the *distribute* scope, allocated area from shared memory is recopied to own location at the global memory.
– *dist_first_lastprivate(list)* : it is a short-cut for specifying *dist_firstprivate(list)* and *dist_lastprivate(list)* at the same time.

Figure 5 shows the use of shared variables with *distribute* and *team* directives for the DG_kernel application (which is used later in the evaluation section). In this example, *delta*, *der* and *grad* are small arrays which are privatized with *private* and *firstprivate* at line 13. However, *flx* and `fly` are specified as *dist_first_lastprivate* with a chunk size of `CHUNK` at line 14.

6   Preliminary performance evaluation

The objective of the performance evaluation in this section is to show how the OmpSs proposal to program accelerators behaves, which just integrates those directives from OpenMP 4.0 accelerator model that are used to specify the kernel computations. For the evaluation we use three codes: Jacobi, DG-kernel [11] from NCAR and CG from NAS Parallel Benchmark [12].

For the experimental evaluation we have used a node with 2 Intel Xeon E5649 sockets (6 cores each) running at 2.53 GHz and with 24 GB of main memory, and two Nvidia Tesla M2090 GPU devices (512 CUDA cores each, compute capability 2.0) running at 1.3GHZ and with 6GB of memory per device. For the compilation of OpenACC codes we have used the HMPP (version 3.2.3) compiler from CAPS [13]. For the compilation of OmpSs codes we have used the

---

[4] **opt2** in the experimental evaluation of DG-kernel in Section 6.
[5] **opt1** in the experimental evaluation of DG-kernel in Section 6.

```
1 #define nX 4
2 #define NELEM 90000
3 #define SIZE (NELEM*nX*nX)
4 #define CHUNK 256
5 #define NUM_TEAMS 5625
6
7 double delta[nX*nX], der[nX*nX], grad[nX*nX], flx[SIZE], fly[SIZE];
8
9 for (it=0; it<nit; it++)
10 {
11   #pragma omp target device(acc) copy_deps
12   #pragma omp task inout(flx[0:SIZE], fly[0:SIZE])
13   #pragma omp teams num_teams(NUM_TEAMS) private(grad) firstprivate(delta,der)
14   #pragma omp distribute parallel for dist_first_lastprivate(flx[CHUNK],fly[CHUNK])
15   for (ie=0; ie < NELEM; ie++)
16   {
17     #pragma omp parallel for private(j,i)
18     for (ii=0; ii < nX*nX; ii++) {
19
20       // <..computation..>
21
22       for (j=0; j < nX; j++)
23       {
24         // <..computation..>
25
26         for (int i=0; i < nX; i++)
27           // <..computation..>
28
29         // <..computation..>
30       }
31
32       // <..computation..>
33     }
34
35     #pragma omp parallel for
36     for (j=0; j < nX*nX; j++)
37       // <..computation..>
38   }
39 }
```

Fig. 5. Example to explain MACC implementation of shared memory - DG Kernel

Mercurium/Nanos environment [9],[10]. GCC 4.6.1 has been used as back-end compiler for CPU code generation and the CUDA 5.0 toolkit for device code generation. Performance is reported in terms of execution time for the kernels generated and speed–up, with respect to sequential execution on a single core, for the complete application.

6.1 Jacobi

Jacobi is a simple iterative program to get an approximate solution of a linear system A*x=b. In each iteration of an outer *while* loop two nested loops are executed, the second one performing the main computation and including a reduction operation on a scalar variable used to control convergence in the *while* loop. The structure of the code is shown in Figure 6, with three different annotations that correspond to three different versions[6]:

- "OpenACC baseline" – each loop is a *kernels* region with the individual specification of data copying.
- "OpenACC optimized" – a *data* region is defined, which includes the two *kernels* regions mentioned in the previous version.
- "MACC/OmpSs" – equivalent to "OpenACC baseline" in terms of *target* regions but written in OpenMP 4.0. In this version the programmer relies

---

[6] The OpenACC versions could have equivalent versions in OpenMP 4.0.

on the runtime system to do all data allocations and copying when necessary. Observe that all *target* regions are *tasks*. This is because the current OmpSs implementation just supports asynchronous *target* regions (not yet in OpenMP 4.0 specification); in this code this does not have any influence due to the serialization caused by data dependences.

The left plot in Figure 7 shows the total execution time of the kernels generated by HMPP and MACC compilers for a data size of 2048 elements. For this code there are no significant differences in the quality of the CUDA kernels generated. The right plot in the same figure shows the speed–up that is obtained for the three versions mentioned above for three different problem sizes: 512, 1024 and 2048. First of all, observe that in OpenACC (and in OpenMP 4.0) the programmer needs to define an external *data* region to minimize data copying between consecutive *kernels* regions, while taking care of updating the scalar *error* variable in the device and host. This achieves a relative speed–up of 25 between the OpenACC optimized and baseline versions. And second, the performance plot also shows that the runtime system in OmpSs is able to achieve a slightly better performance even with the overheads incurred by keeping track of memory allocations, data copying and orchestration of kernel execution.

OpenACC baseline

```
1 while ((k <= mits) && (error > tol)) {
2   error = 0.0;
3
4   #pragma acc kernels copyin(u)
5                    copyout(uold)
6   #pragma acc loop
7   for (i = 0; i < n; i++)
8     // <..computation..>
9
10  #pragma acc kernels copyin(uold)
11        copyin(u) copy(error)
12  #pragma acc loop reduction(+:error)
13  for (i = 1; i < (n - 1); i++)
14    // <..computation..>
15
16  error = sqrt(error) / (n * m);
17  k++;
18 }
```

OpenACC optimized

```
1 #pragma acc data copy(u) copyout(error)
2                  create(uold, error)
3 while ((k <= mits) && (error > tol)) {
4   error = 0.0;
5
6   #pragma acc kernels loop
7   for (i = 0; i < n; i++)
8     // <..computation..>
9
10  #pragma acc update device(error)
11  #pragma acc kernels loop reduction(+:error)
12  for (i = 1; i < (n - 1); i++)
13    // <..computation..>
14
15  #pragma acc update host(error)
16  error = sqrt(error) / (n * m);
17  k++;
18 }
```

MACC/OmpSs

```
1 while ((k <= mits) && (error > tol)) {
2   error = 0.0;
3
4   #pragma omp target device(acc) copy_deps
5   #pragma omp task in(u) out(uold)
6   #pragma omp teams distribute parallel for
7   for (i = 0; i < n; i++)
8     // <..computation..>
9
10  #pragma omp target device(acc) copy_deps
11  #pragma omp task in(uold) out(u) inout(error)
12  #pragma omp teams distribute parallel for reduction(+:error)
13  for (i = 1; i < (n - 1); i++)
14    // <..computation..>
15
16  #pragma omp taskwait
17  error = sqrt(error) / (n * m);
18  k++;
19 }
```

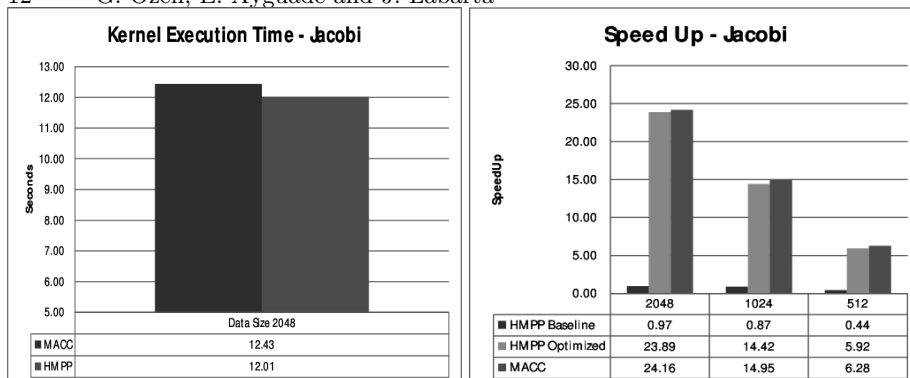**Fig. 6.** Annotated codes for Jacobi application

Fig. 7. Performance evaluation of Jacobi application

6.2   DG Kernel

DG is a kernel version of a climate benchmark developed by National Center for Atmospheric Research [11]. The structure of the code has been omitted in this submission version but will be included if the paper is accepted for publication. The code consists of a single *target* region that is executed inside an iterative time step loop that is repeated for a fixed number of iterations. Inside the *target* region the iterations of two nested loops are mapped to the teams/thread hierarchy as specified by the programmer.
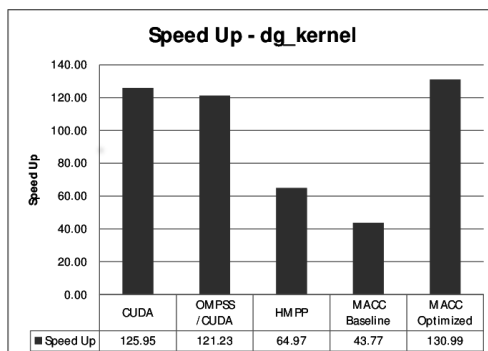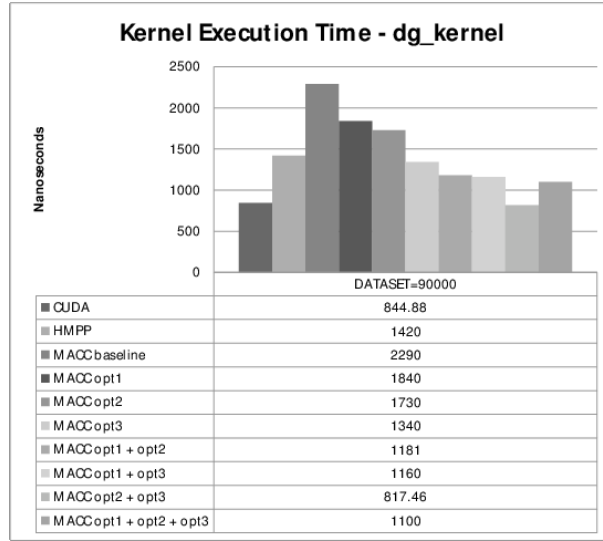

Fig. 8. Performance evaluation of for DG kernel

Figure 8 plots the performance that is achieved by different versions of the code, described in the following bullet points:

- CUDA: hand–optimized CUDA version of the application (with host and kernel code written in CUDA) available from NCAR.
- OmpSs/CUDA: OmpSs version of the application leveraging (only) the computational kernels written in CUDA.
- HMPP: OpenACC version available from NCAR.

– MACC: different versions of our OpenMP 4.0 implementation in the MACC compiler, including additional clauses to influence kernel code generation by the compiler.

Comparing bars labelled CUDA and OmpSs/CUDA in Figure 8 one can extract a first conclusion: OmpSs is able to leverage existing CUDA kernels with similar performance as full host/device CUDA codes. In this case we observe a small performance degradation probably due to overheads of the runtime in generating tasks in each iteration of the time step loop.

Kernel Execution Time - dg_kernel

	DATASET=90000
■ CUDA	844.88
■ HMPP	1420
■ MACC baseline	2290
■ MACC opt1	1840
■ MACC opt2	1730
■ MACC opt3	1340
■ MACC opt1 + opt2	1181
■ MACC opt1 + opt3	1160
■ MACC opt2 + opt3	817.46
■ MACC opt1 + opt2 + opt3	1100

Fig. 9. Performance evaluation of CUDA Kernel for DG kernel

The second conclusion from this evaluation is the important role of the compiler in generating efficient kernel codes for the target device. The first 3 bars at Figure 9 show the execution time for the original CUDA kernel, the kernel generated by the HMPP compiler and the initial kernel generated by the MACC compiler. As one can observe, the manually programmed CUDA kernel clearly outperforms the kernels generated by the two compilers, which directly translate into significant performance degradation in terms of speed–up for the whole application (first, third and fourth bar, in Figure 8).

Thanks to the previous observation and to the study of the kernels available and generated by the compilers, we have been investigating alternative code generation schemes and proposed a new clause for the *distribute* directive (*dist_private*, as explained in Section 5). The impact of these optimizations is shown in the performance plot at Figure 9. Observe that there is plenty of room for improvement by using and combining these optimizations which result in a clear impact in the overall speed–up of the application (last bar in the left plot).

6.3 CG from NAS Parallel Benchmarks

The last code we have selected for the experimental evaluation in this paper is
NAS CG [12]. The main computational part of the application contains several
loops that can be made tasks and offloaded to a device or executed on the host.
The loop that contributes the most to the execution time performs a sparse
matrix vector operation. To execute this loop we want to use the two GPUs
available in the node.

The performance plot in Figure 10 shows the speed–up of the GPU accel-
erated version of NAS CG (bars HMPP, MACC and MACC/2 GPU) and the
speed–up using 8 processors in the host, for three different classes of NAS CG.
The speed–up with 2 GPU is significant although we only refined one of the
loops. Note that data transfers between GPUs will take place, automatically
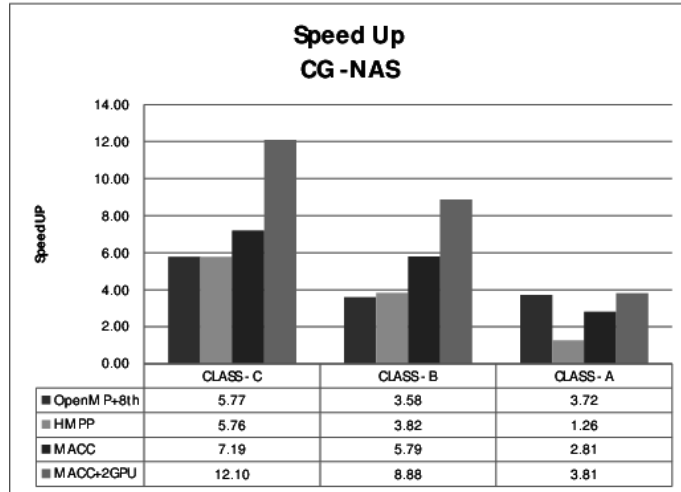handled by the runtime.

Speed Up
CG -NAS

	CLASS - C	CLASS - B	CLASS - A
■ OpenM P+8th	5.77	3.58	3.72
▨ HM PP	5.76	3.82	1.26
■ MACC	7.19	5.79	2.81
▨ MACC+2GPU	12.10	8.88	3.81

Fig. 10. Performance evaluation for NAS CG

7 Conclusions

In this paper we presented the main design considerations that are embedded in
our current implementation of the OpenMP 4.0 [4] accelerator model in OmpSs,
making emphasizing on the roles of the programmer, compiler and runtime sys-
tem in the whole picture. The compiler plays a key role and for this reason pre-
vious efforts have been devoted to the automatic generation of device–specific
programs from high-level programs annotations such as OpenMP and OpenACC
[3], including both research efforts at academia [15–17] as well as commercial im-
plementations [13, 14]. Our compiler implementation in Mercurium [9] has been
useful to experiment with different code generation strategies, trying to fore-
see the need for new clauses in current OpenMP 4.0 specification. OmpSs [5] is
strongly rooted on the assumption that the runtime system should play a key

role, making appropriate use of the information that can be gathered at execution time. In this paper we tried to emphasize this aspect supported by an experimental evaluation on three application kernels.

8   Acknowledgments

References

1. Nvidia CUDA parallel computing and programming. url="`www.nvidia.com/cuda`".
2. OpenCL Open Computing Language. url="`www.khronos.org/opencl/`".
3. OpenACC: Directives for Accelerators. url="`www.openacc-standard.org`".
4. The OpenMP API Specification for Parallel programming. url="`www.openmp.org`".
5. Barcelona Supercomputing Center. The OmpSs programming model. url=`pm.bsc.es/ompss`
6. Pieter Bellens, Josep M. Perez, Rosa M. Badia and Jesus Labarta. CellSs: a programming model for the Cell/B.E. architecture. In *ACM/IEEE Supercomputing*, November 2006.
7. J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade and J. Labarta. Productive programming of GPU clusters with OmpSs. In *Parallel Distributed Processing Symposium (IPDPS), IEEE 26th International*, May 2012.
8. Josep M. Perez, Rosa M. Badia and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *IEEE International Conference on Cluster Computing*, September 2008.
9. Barcelona Supercomputing Center. Mercurium source-to-source compiler. url="`pm.bsc.es/mcxx`".
10. Barcelona Supercomputing Center. Nanos++ runtime library. url="`pm.bsc.es/nanos`".
11. S. Vadlamani, Youngsung Kim, and J. Dennis. DG-kernel: A climate benchmark on accelerated and conventional architectures. In *Extreme Scaling Workshop (XSW)*, August 2013.
12. NAS Division. NAS parallel benchmarks, url="`www.nas.nasa.gov/resources/software/npb.html`".
13. CAPS Entreprise, CAPS Compiler. url="`www.caps-entreprise.com`".
14. PGI Accelerator Compilers. url="`www.pgroup.com/resources/accel.htm`".
15. T. D. Han and T. S. Abdelrahman. hicuda: A high-level directive-based language for gpu programming. In *2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, March 2009.
16. S. Lee, S-J. Min and R. Eigenmann. OpenMp to GPGPU: A compiler framework for automatic translation and optimization. In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2009.
17. C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan and B. Chapman Early experiences with the OpenMP accelerator model. In *International Workshop on OpenMP (IWOMP-2013)*, May 2013.