Extending OmpSs to support CUDA and OpenCL in C, C++ and Fortran Applications

Florentino Sainz, Sergi Mateo, Vicenç Beltran
*Barcelona Supercomputing Center
Barcelona, Spain
{florentino.sainz, sergi.mateo, vbeltran}@bsc.es

Jose L. Bosque
University of Cantabria
Santander, Spain
joseluis.bosque@unican.es

Xavier Martorell*† and Eduard Ayguadé*†
†Technical University of Catalonia
Barcelona, Spain
{xavim, eduard}@ac.upc.edu

***Abstract*—CUDA and OpenCL are the most widely used programming models to exploit hardware accelerators. Both programming models provide a C-based programming language to write accelerator kernels and a host API used to glue the host and kernel parts. Although this model is a clear improvement over a low-level and ad-hoc programming model for each hardware accelerator, it is still too complex and cumbersome for general adoption. For large and complex applications using several accelerators, the main problem becomes the explicit coordination and management of resources required between the host and the hardware accelerators that introduce a new family of issues (scheduling, data transfers, synchronization, ...) that the programmer must take into account.**

In this paper, we propose a simple extension to OmpSs –a data-flow programming model– that dramatically simplifies the integration of accelerated code, in the form of CUDA or OpenCL kernels, into any C, C++ or Fortran application. Our proposal fully replaces the CUDA and OpenCL host APIs with a few pragmas, so we can leverage any kernel written in CUDA C or OpenCL C without any performance impact. Our compiler generates all the boilerplat code while our runtime system takes care of kernels scheduling, data transfers between host and accelerators and synchronizations between host and kernels parts. To evaluate our approach, we have ported several native CUDA and OpenCL applications to OmpSs by replacing all the CUDA or OpenCL API calls by a few number of pragmas. The OmpSs versions of these applications have competitive performance and scalability but with a significantly lower complexity than the original ones.

***Index Terms*—Accelerator, CUDA, OpenCL, OmpSs**

I. Introduction

Hardware accelerators provide better performance and energy efficiency than traditional processors for a wide range of workloads. However, hardware design and software development costs have traditionally hindered a wider adoption of these technologies. This trend started to change with the introduction of programmable GPUs and the CUDA SDK by NVIDIA. The use of commodity hardware paired with a more developer-friendly programming model reduced both hardware design and software development costs.

With the introduction of CUDA [1] and OpenCL [2] the rate of adoption of hardware accelerators in general and GPUs in particular is growing at a fast rate. The development of the OpenCL standard has enabled the proliferation of many types of accelerators such as embedded SoC (ARM Mali), the Intel Xeon Phi, the Cell/BE or FPGAs boards, which have a native programming model, but also support the OpenCL standard. CUDA and OpenCL are a great advance over low-level and device specific programming techniques, which have helped the adoption of hardware accelerators on many fields where the performance gains paid off the higher development costs associated with CUDA and OpenCL. However, most large and complex applications still cannot benefit from hardware accelerator. The main reason is that rewriting these applications from scratch is not an option and modifying them incrementally is still a complex and error prone task that most developers cannot afford with current tools.

In this paper, we extend the OmpSs [3] programming model to ease integration of OpenCL C and CUDA C kernels into an existing application written in C/C++ or Fortran. OmpSs is a programming model that is able to execute sequential programs in a data-flow way. 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 run be in parallel. In this paper, we present a new *ndrange* clause that, in conjunction with the *input* and *output* clauses, allows OmpSs to transparently manage data transfers from the host to the accelerator and vice versa, providing the illusion of a single address space and transparently running CUDA C and OpenCL C kernels on the available accelerators.

Thus, OmpSs not only eliminates all the boilerplate code required to execute a kernel on a hardware accelerator but also effectively reliefs the programmer from manually scheduling kernels and managing data transfers from/to hardware accelerators. With the extensions to OmpSs presented in this work developers can use any CUDA C or OpenCL C kernel like a regular C function, even from a Fortran application. Hence, the programmer only needs to focus on writing kernels with CUDA C or OpenCL C, making the integration of these kernels with large and complex applications a straightforward task. Our approach also makes trivial the simultaneous use of several accelerators, even if the programmer mixes OpenCL and CUDA kernels, as the run-time system takes care of all the low-level details.

We have evaluated the performance of our proposal porting several CUDA and OpenCL applications to OmpSs. The results show that our approach is much more productive, and it has same or better performance than the hand-optimized native applications.

The rest of the paper is organized as follows: Section II introduces both CUDA and OpenCL and re-examines some relevant related works. Section III introduces the OmpSs programming models and the extensions presented in this paper. Section IV describes the applications used for the evaluation performed in Section V, where we study the productivity, performance and scalability of OmpSs. Finally, we draw conclusions and outline some future work in Section VI.

II. Programming models for hardware accelerators

The growing popularity of hardware accelerators has encouraged researchers and industry to develop novel programming models to make the most of these new compute devices with a moderate effort. CUDA was the first mainstream programming model widely used to exploit GPUs. A standardization effort to exploit hardware accelerators have led to the development of OpenCL, which is the only programming model that can exploit a variety of accelerators. In the rest of this section, we will summarize the main features of both CUDA and OpenCL, as well as some other related work.

A. CUDA

CUDA (formerly Compute Unified Device Architecture) was first introduced back in 2007 by NVIDIA to exploit the GPUs computational power. The CUDA SDK provides a host API to manage computational kernels that are written in CUDA C. The host API is mainly used to configure the kernel arguments, to explicitly manage the GPU memory (allocation and memory transfers from/to the host memory) and to manage synchronization between host code and accelerator code. On the other hand, the CUDA C programming language is a C-based programming model specially designed to exploit the parallel nature of the GPU hardware. Hence, the host API works as a glue between the main program written in C or C++ and the kernels written in CUDA C. The complexity of developing a CUDA application is clearly split in two different parts. The first one is related to the development of optimized CUDA C kernels that can exploit the GPU hardware, and the second one is related to the integration of these optimized kernels with the main application. The first part can be alleviated with the use of kernel libraries, such as CUBLAS [4], or programming models such as OpenMP 4.0 or OpenACC that can run sequentially written code on accelerators, but the second one cannot be avoided and it is specially complex and hard to overcome for legacy applications that were not designed with CUDA in mind. Despite these shortcomings, CUDA is widely used in many fields where the benefits of using this hardware paid off the higher development cost. However, there are still many applications where the expected benefits of using GPUs are not enough to re-write them in CUDA or OpenCL.

B. OpenCL

OpenCL was first developed by Apple and then submitted to the Khronos group, which published the first specification at the end of 2008. Since then many vendors such as IBM, Intel, NVIDIA, Samsung or AMD, have announced their own OpenCL SDK for a wide range of hardware platforms: from many-core processors to SoC, GPUs or even FPGAs. The philosophy behind OpenCL is similar to CUDA: a host API to explicitly manage the hardware accelerators and a OpenCL C programming language to write the computational kernels. Thus, the same advantages and drawbacks from CUDA apply to OpenCL. The major difference is that the OpenCL API is a bit more low-level and all the kernels must be explicitly compiled at run-time before they can run on a given accelerator. Hence, the complexity of developing a OpenCL application is comparable to developing a CUDA application as well as the expected performance gain.

Listings 1 and 2 shows the required code to implement a program that performs a SAXPY operation. The first listing correspond to the kernel code written in OpenCL C, which will run in the accelerator.

Listing 1: OpenCL C kernel code

```
__kernel void saxpy(int n, float a,
    __global float* x, __global float* y)
{
   int i = get_global_id(0);
   if(i < n)
      y[i] = a * x[i] + y[i];
}
```

The second listing shows the part of the application that will run on the host side. As we can see, all the code between lines 9 and 46 are only to setup the environment. This steps include: selecting and initializing the device in which the kernel will run (lines 9 - 21), compile the OpenCL C kernel (lines 27-30), copy the data from the host to the device (lines 32-40) and set the parameters of the kernel (lines 42-46). Once everything is in place the kernel can be executed (lines 49-52). When the result is ready, we need to copy back the data to the host memory (line 54-55). Finally some more API calls are required to free all the resources used (lines 59-64).

Listing 2: OpenCL Host code

```
1 #include "CL/cl.h"
2 #define DEV CL_DEVICE_TYPE_DEFAULT
3
4 int main(int argc, char** argv)
5 {
6   float a, h_x[1024], h_y[1024];
7   // Init a, h_x and h_y;
8
9   cl_uint numPlats;
10  clGetPlatformIDs(0, 0, &numPlats);
11
12  cl_platform_id Plat[numPlats];
13  clGetPlatformIDs(numPlats, Plat, 0);
14
15  clGetDeviceIDs(Plat[i], DEV, 1, &id,0);
16
17  cl_context ctx = clCreateContext(
18                   0, 1, &id, 0, 0, 0);
19
20  cl_command_queue cmd =
```

```
21    clCreateCommandQueue(ctx, id, 0, 0);
22
23  cl_program program =
24    clCreateProgramWithSource(ctx, 1,
25                        KernelSrc, 0, 0);
26
27  clBuildProgram(program, 0, 0, 0, 0, 0);
28
29  cl_kernel ko_saxpy = clCreateKernel(
30                      program, "saxpy", 0);
31
32  cl_mem d_x = clCreateBuffer(ctx, 0,
33                 sizeof(float) * n, 0, 0);
34  cl_mem d_y = clCreateBuffer(ctx,  0,
35                 sizeof(float) * n, 0, 0);
36
37  clEnqueueWriteBuffer(cmd, d_x, CL_TRUE,
38     0, sizeof(float) * n, h_x, 0, 0, 0);
39  clEnqueueWriteBuffer(cmd, d_y, CL_TRUE,
40     0, sizeof(float) * n, h_y, 0, 0, 0);
41
42  clSetKernelArg(ko_saxpy, 0, 4, &n);
43  clSetKernelArg(ko_saxpy, 1, 4, &a);
44  size_t sze = sizeof(cl_mem);
45  clSetKernelArg(ko_saxpy, 2, sze, &d_x);
46  clSetKernelArg(ko_saxpy, 3, sze, &d_y);
47
48  size_t global = 1024, local = 128;
49  clEnqueueNDRangeKernel(cmd, ko_saxpy,
50          1, 0, &global, &local, 0, 0, 0);
51
52  clFinish(commands);
53
54  clEnqueueReadBuffer( commands, d_y,
55       CL_TRUE, 0, 4 * n, h_y, 0, 0, 0 );
56
57  printf("%f, h_y[0]);
58
59  clReleaseMemObject(d_x);
60  clReleaseMemObject(d_y);
61  clReleaseProgram(program);
62  clReleaseKernel(ko_saxpy);
63  clReleaseCommandQueue(commands);
64  clReleaseContext(ctx);
65  return 0;
66 }
```

C. Related Work

The widespread adoption of heterogeneous systems have raised the question about their programmability. Some early work, such as [5] and [6], already explored ways to integrate OmpSs dataflow model with CUDA and OpenCL. This first approach was limited to C, using an outline SMP task to call the native kernels using the CUDA syntax or linking the object code produced by the OpenCL compiler. The main drawback of this approach was that the outlined SMP task has to be hand-written by the programmer, who will see pointers from the accelerator address spaces on the host side of the application, making this approach less robust and more error-prone than the one proposed in this paper. Moreover, OpenCL was only supported for CPU devices.

The CAPS HMPP [7], OpenACC [8] or OpenMP 4.0 [9] programming models are a set of compiler directives, tools and runtimes that supports parallel programming in C and Fortran. HMPP and OpenACC are based on codelets that define functions that will be run in a hardware accelerator. These codelets can either be handwritten for a specific architecture or be generated from sequential C or Fortran code like in OpenMP 4.0. However, the programmer still needs to explicitly orchestrate how memory is allocated and data transfered between the host and the accelerator address spaces with some specific pragmas. The synchronization of host and kernel code must also have to be explicitly done with some pragma annotations. Thus, the main benefit of these methods is that they relief the programmer from writting in CUDA C or OpenCL C, while our approach focus on replacing the host API of CUDA and OpenCL by the OmpSs data-flow model, which transparently do all the memory managment and code synchronization. The Intel offload directives [10] for the Xeon Phi works in a similar way to OpenACC, automatically compiling the code for the accelerator, but it also requires the explicit management of data transfers between the host and the accelerator. Offload [11] is a programming model for offloading portions of C++ applications to run on accelerators. Code to be offloaded is wrapped in an offload block, indicating that the code should be compiled for an accelerator, and executed asynchronously as a separate thread. Call graphs rooted at an offload block are automatically identified and compiled for the accelerator. Data movement between host and accelerator memories is also handled automatically.

Merge [12] encapsulates specialized languages targeting accelerators (GPUs, FPGAs) in C/C++ functions to provide a uniform interface for them. Encapsulation is based on EXOCHI [13], which uses pragmas to offload the domain specific language to be compiled with the compiler of the target device. Merge allows the specification of the same function for different targets, as new intrinsic functions, and it provides the mechanism for dynamic function selection at run-time. StarPU [14] provides numerical kernel designers with a convenient way to generate parallel tasks over heterogeneous hardware on the one hand, and easily develop and tune powerful scheduling algorithms on the other hand. StarPU is based on a tasking API and also on the integration of a data-management facility with a task execution engine. With regard to data management, StarPU proposes a high level library that automates data transfers throughout heterogeneous machines [15]. In StarPU codelets are defined as an abstraction of a task (e.g., a matrix multiplication) that can be executed on a core or offloaded onto an accelerator using an asynchronous continuation passing paradigm. StarPU offers low level scheduling mechanisms (e.g., work stealing) so that scheduler programmers can use them in a high level fashion, regardless of the underlying (possibly heterogeneous) target architecture. While OmpSs and StarPU present several similarities with regard to the execution model, StarPU is implemented as a library and therefore the programmer is exposed to low-level APIs and execution details that are hidden in the OmpSs case because the Mercurium compiler generates the necessary outline functions and boilerplat code with the information provided on the

ndrange clause. The management of GPUs in Charm++ [16] is done through the Charm++ GPU Manager. The Charm++ GPU Manager is a library designed to automate the management of GPUs. Users of the GPU Manager define work requests which specify the GPU kernel and any data transfer operations required before and after completion of the kernel. The system controls the execution of the work requests submitted by all the *chares* on a particular processor. This allows it to effectively manage execution of work requests and overlap CPU-GPU data transfers with kernel execution. In steady-state operation, the GPU Manager overlaps kernel execution of one work request with data transfer out of GPU memory for a preceding work request and the data transfer into GPU memory for a subsequent work request.

III. OmpSs to exploit hardware accelerators

OmpSs is a directive-based programming model that enables the execution of sequential programs in a data-flow way. Listing 3 shows the previous SAXPY example implemented in OmpSs (with no support for GPUs). The programmer only needs to specify that the *x* vector of size *n* is going to be read (*in*) and that the *y* vector of size *n* is going to be read and written (*inout*). The *pragma taskwait* is to ensure that the result is ready before it is actually printed. These annotations are interpreted by the Mercurium source-to-source compiler, which emits calls to the run-time system Nanos++. Nanos++ uses the information provided by these user annotations to build dynamically a task dependency graph, which is used to schedule tasks in a data-flow way. We have extended the OmpSs programming model to support directly tasks written in CUDA C or OpenCL C, freeing developers from writing all the boilerplate code required to explicitly schedule kernels and manage data transfers, specially on multi-accelerator and distributed systems. All the details are on Section III-C

Listing 3: OmpSs saxpy example

```
#pragma omp task in([n]x) inout([n]y)
void saxpy(int n,float a, float* x, float* y){
  for(int i=0; i<n; i++)
      y[i] = a * x[i] + y[i];
}

int main(int argc, char* argv[]) {
   float  a, x[1024], y[1024];

   saxpy(1024, a, x,  y);

#pragma omp taskwait
  printf("%f, y[0]);
  return 0;
}
```

A. Mercurium

Mercurium is a source-to-source compiler that provides the OmpSs programming model for C/C++ and Fortran languages. Mercurium parses and analyzes the pragma directives provided by the programmer, then the original code is augmented with calls to the Nanos++ run-time. The Mercurium compiler also generates the function stubs required to call CUDA C and OpenCL C kernels, from C/C++ or Fortran host code. Finally, Mercurium calls the appropriate native compiler (gfortran, icpc, xlc, ...) to generate the host side of the application, which is linked with Nanos++. The CUDA C kernels are also compiled and linked at compilation time with the nvcc compiler, while the OpenCL kernels are saved as strings in the host binary and automatically compiled at run-time the first time they are executed.

B. Nanos++

Nanos++ is a modular, extensible and portable execution run-time for parallel and distributed systems. The programming models supported by this run-time include OmpSs [3], OpenMP [17] and Chapel [18]. Nanos++ uses a task–based approach to exploit parallelism, while also providing support for synchronizations based on data–dependencies. Moreover, data parallelism is also supported on top of the task support. The cluster version of Nanos++ enables the distributed execution of task using a software directory and data–cache, which keeps program consistency and coherency across the whole cluster. GasNET [19] is used as the low–level networking layer used to exchange messages between nodes.

C. NDRANGE clause

To support CUDA C and OpenCL C kernels we have added the *NDRANGE* clause to the programming model. This clause contains the information required to configure and launch a kernel. With the information of the *NDRANGE* clause and the information provided by the *in, out* and *inout* clauses the compiler and the run-time have all the required information to transparently run CUDA or OpenCL kernels from C/C++ or Fortran.

Listing 4: *ndrange* clause definition

```
#pragma omp target device(opencl|cuda) \
    ndrange(work_dim, gwork_size, lwork_size)
#pragma omp task in(...) out(...)
cuda_or_opencl_kernel_declaration(....)
```

Listing 4 shows how the *target device* construct is used to specify the type of a kernel (either cuda or opencl) while the *ndrange* clause is used to configure the number of dimensions (*work_dim*), the global size of each dimension (*gwork_size*) and the number of work-items of each work-group (*lwork_size*). These parameters are the same used on the OpenCL *clEnqueueNDRangeKernel* (see Listing 2, line 49-50) API call and equivalent to the ones provided with CUDA special syntax.

Listing 5: Example of NDRANGE directive

```
#pragma omp task in([n]x) inout([n]y)
#pragma target device(opencl) \
        ndrange(1, n, 128) copy_deps
__kernel void saxpy(int n, float a,
    __global float* x, __global float* y)
{
   int i = get_global_id(0);
```

```
    if(i < n)
      y[i] = a * x[i] + y[i];
}

#define N 1024
int main(int argc, char* argv[]) {
   float  a, x[N], y[N];

   saxpy(N, a, x,  y);

#pragma omp taskwait
  print_result(y, N);
  return 0;
}
```

Listing 5 shows the previous SAXPY example written in OmpSs+OpenCL C. The example contains both host side and accelerator side of the application (our compiler support both single source as well as separate files for host and accelerator code). It is worth noting that no call to the OpenCL host API is necessary to launch a kernel, the syntax used is just plain C and three pragmas. Compared with the previous OmpSs version of SAXPY we have only added the pragma *target device* with the *ndrange* clause. The Mercurium compiler will generate all the required boilerplate code to call the OpenCL SAXPY kernel, as shown in Listing 6 (simplified for the sake of clarity). The original call to the SAXPY kernel has been replaced to a call to the Nanos++ API to create a new task, which will be inserted on the dependency graph (with the information from the *input* and *output* clauses). Once the task is ready and there is one accelerator available, the function *saxpy_outline* will be executed by a thread on the host. This function will compile (if necessary) and create a kernel that can run on the available accelerator. Then the information of the *ndrange* clause will be used to configure the kernel and the original parameters of the *saxpy* call will be forwarded to the kernel (and translated in the case of arrays). At this point, the Nanos++ run-time will take care of all the data managements (allocation, transfers, ...) using the plain OpenCL API on the selected device. It is worth nothing that Nanos++ do not rely on the OpenCL or CUDA API for inter-kernel and/or task synchronization because this work is done on the dynamic task dependency graph. Nanos++ directory and cache system have also been extended to manage data transfers across host and accelerators, ensuring data coherence and allowing mixed execution of OpenCL and CUDA kernels. With these extensions, a regular C/C++ or Fortran application can easily leverage the power of multi-hardware accelerators with just a few OmpSs directives, so the integration of CUDA and/or OpenCL kernels with an existing application requires a minimal effort.

Listing 6: NDRANGE code transformation

```
#define N 1024
int main(int argc, char* argv[]) {
  float  a, x[N], y[N];

  saxpy_args_t args = {N, a, x, y};
  ndrange_args_t ndrange = {1, N, 128, 0};
  nanos_create_new_task(&saxpy_outline,
                        &args, &ndrange);

  nanos_wait_for_tasks();
  print_result(y, N);
  return 0;
}

void saxpy_outline(saxpy_args_t *args,
                   ndrange_args_t *ndrange){
  void* kernel=nanos_create_kernel("saxpy",
                  path,opt);

  nanos_set_arg(0,kernel,args->n,sizeof(int));
  nanos_set_arg(1,kernel,args->a,sizeof(float)

  nanos_set_cache_arg(2, kernel,
    nanos_cache_translate_addr(args->x));
  nanos_set_cache_arg(3, kernel,
    nanos_cache_translate_addr(args->y));

  nanos_exec_kernel(ndrange->dim,0/*offsset*/,
    ndrange->global,ndrange->local,kernel);
}
```

IV. Selected applications

We have evaluated our approach with six benchmark that have been ported from CUDA to OpenCL or vice versa and then to OmpSs. The port of the computational kernels from CUDA C to OpenCL C have been straightforward as both languages are very similar, but the adaptation of the host side has been more complex, specially on the benchmarks that does not follow a fork-join parallelism and uses event to perform complex synchronizations. The conversion of native CUDA or OpenCL versions to OmpSs was easy, as we only need to get rid of CUDA and OpenCL host API calls and add a few pragmas to annotate the input and output of the computational kernels. Thus we have four version of each benchmark: native OpenCL, native CUDA, OmpSs + OpenCL C and OmpSs + CUDA C. It is worth noting that the host side of both OmpSs versions are the same, as well as the kernels used, which are copied verbatim from the native versions. The rest of the Section explains the most relevant characteristics of the selected benchmarks with special enphasis on the OmpSs version of each benchmark.

A. Matrix Multiply (Matmul)

This benchmark performs a blocked matrix multiplication of two input matrices (A, B) and produce an output matrix (C). Listing 7 shows the declaration of the OpenCL C kernel used to perform each tile operation which is part of the AMD APP SDK [20]. Matrix C is initialized to zero and then a simple block-matrix multiplication algorithm is applied. Each call to the *Muld* kernel performs a matrix multiplication on blocks of *NB*NB* size. The OmpSs data-flow excution model enables the parallel execution of several of these kernels, that can simultaneously run on several hardware accelerators. As we can see, there is no need to specify any data transfer between host and accelerator or vice versa as our runtime system takes

care of all the details. The Native CUDA and OpenCL versions are much more complex because the programmer must explicitly orchestrate the parallel execution and synchronization of several kernels to obtain an acceptable level of performance, while in the OmpSs case the runtime automatically caches the data to improve data locality and minimize the host to/from device data transfers.

B. NBody

An N-body simulation [21] numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body. Each timestep (iteration) of the simulation process can be decomposed into several parts that can be computed in parallel (in a fork-join way). The aggregate output of all the parts for one iteration becames the input of the next one.

The code in Listing 8 shows how easy is with OmpSs to mix OpenCL C and CUDA C kernels. As we can see on the loop that iterates over the full vector of particles, we can easily combine calls to CUDA C and OpenCL C kernels. In this concrete example, even partitions are calculated with a OpenCL C kernel while odd partitions are calculated with a CUDA C kernel. It is worth noting that the Nanos++ runtime performs all the device-to-host and host-to-device copies between OpenCL and CUDA devices.

C. Krist

The Krist application [22] is used on crystallography to find the exact shape of a molecule using Rntgen diffraction on single crystals or powders. This application has been ported from C to Fortran to show how our *ndrange* clause works on Fortran. We take advantage of the standard *INTERFACE* construction used in Fortran to call C functions to be able to specify the signature of the OpenCL C or CUDA C kernels. That information is augmented with the *target device* pragma and the *ndrange* clause as well as with the usual OmpSs definition of *inputs* and *outputs*. As we can see on Listing 9, once the kernel have been declare with an appropiate interface and annotated with the corresponding pragmas, we can use it like a regular Fortran subroutine. This application is also interesting because it requires the use of shared memory inside a grid in CUDA terminology or local memory inside a workgroup in OpenCL terminology. To specify the amount of shared memory we have added an *shmem* clause that indicates the amount of memory that will be used as shared memory. This is enough for CUDA, because only one shared space can be used but in OpenCL we need to augment that clause to support multiple shared variables. To that end, we also support the specification of several sizes delimited by a comma, that are assigned in consecutive order to the pointer parameters of the OpenCL C kernels that are neither *in* or *out* parameters. Both C and Fortran versions of Krist have similar performance.

D. Julia

The Julia Set benchmark provided by the IBM OpenCL SDK [23] has an input describing both the fractal to be generated and the output image data. This code have been ported from OpenCL to CUDA and then to OmpSs, but the native versions only supported a single device so the host side code has been fully re-designed to work with several accelerators. An snippet of the OmpSs version of the code are shown on Listing 10. The benchmarks are annotated so that each task generates an horizontal slice of the image of height BS lines.

E. NBody MPI

The original NBody code has been augmented with MPI to run on a clusters of accelerators. There is a hierachical partition of each iteration which is first split across the number of nodes, and then split again across the two accelerators availables on each node. The OpenCL C and CUDA C kernels from the original NBody have not required any modification. Listing 11 shows how each iteration is split with MPI and then the result calculated by each node sent back to all other nodes because it will be used as the input of the next iteration. OmpSs is specially well suited to write hybrid MPI+OmpSs applications because its dataflow execution model allows to exploit multi-core processors inside each MPI process and also limits the effect of load inbalance across them. The *ndrange* clause presented in this paper perfectly suits this OmpSs+MPI hybrid approach, simplifying the development of distributed applications that can exploit hardware accelerators.

V. Evaluation

We have evaluated our approach with six applications described in the previous section. Each benchmark has been ported from CUDA to OpenCL or vice versa and then to OmpSs. Hence there are four different flavors of each application: native CUDA, native OpenCL, OmpSs+CUDA C and OmpSs+OpenCL C.

All the benchmarks, but the NBody_MPI, were evaluated on a compute node with two Intel Xeon E5645 24-Core at 2.4 GHz running Linux operating system with 104GB of RAM memory, 12 MB of cache, equipped with 8 NVIDIA M2050 cards each one with 448 CUDA cores and 3GB of GDDR5 memory. For the NBody_MPI benchmark, we have used 32 nodes interconnected with an Infiniband QDR network, but with only two NVIDIA M2090 cards per node.

On the rest of this section, we compare our approach from three different points of view: productivity, performance and scalability.

Productivity: The objective of the proposal presented on this paper is to improve the programmability of hardware accelerators, in particular the ones that support CUDA or OpenCL. To that end we propose to get rid of the CUDA API and OpenCL API in favor of the OmpSs programming model but leveraging the kernels written in CUDA C or OpenCL C. Our rationale is that, for most large and complex applications, the biggest challenge is to orchestrate and integrate the host side of the application with the accelerated part of the application. On the other hand, the development of high performance OpenCL C or CUDA C kernels can also be a hard task, but

Listing 7: Matrix Multiply

```
#pragma omp target device(opencl) ndrange(2, NB, NB, BL_SIZE, BL_SIZE) copy_deps
#pragma omp task inout([NB*NB]C) in([NB*NB]A,[NB*NB]B)
__kernel void Muld(__global REAL* A, __global REAL* B, int wA, int wB,
                   __global REAL* C, int NB);

void matmul(int m, int l, int n, int mDIM, int lDIM, int nDIM,
            REAL **tileA, REAL **tileB, REAL **tileC ){
  for(int i = 0;i < mDIM; i++){
    for (int k = 0; k < lDIM; k++){
      for (int j = 0; j < nDIM; j++){
        Muld(tileA[i*lDIM+k], tileB[k*nDIM+j], NB, NB, tileC[i*nDIM+j], NB);
      }
    }
  }
}
```

Listing 8: NBody code

```
#pragma omp target device(opencl) ndrange(1,size,128) copy_deps
#pragma omp task out([size] out) in([particles] part)
__kernel void calculate_force_opencl(int size, float time, int particles,
             __global Part* part, __global Part* out, int first, int last);

#pragma omp target device(cuda) ndrange(1,size,128) copy_deps
#pragma omp task out([size] out) in([particles] part)
__global__ void calculate_force_cuda(int size, float time, int particles,
                Part* part, Particle *out, int first, int last);

void Particle_array_calculate_forces(Particle* input, Particle *output,
                                     int particles, float time) {
  const int bs = particles/nanos_get_total_num_devices(); // OpenCL + CUDA devices
  assert(particles % nanos_get_total_num_devices() == 0);
  for (int i = 0; i < particles; i += bs ){
    if (i % 2 == 0)
      calculate_force_opencl(bs, time, particles, input, &output[i], i, i+bs-1);
    else
      calculate_force_cuda  (bs, time, particles, input, &output[i], i, i+bs-1);
  }
}
```

Listing 9: Krist Snippet

```
INTERFACE
  !$OMP TARGET DEVICE(OPENCL) COPY_DEPS NDRANGE(1, NR, 128) SHMEM(16384-2048) FILE(krist.cl)
  !$OMP TASK IN (A, H) OUT(E)
  SUBROUTINE CSTRUCTFAC(NA, NR, NC, F2, DIM_NA, A, DIM_NH, H, DIM_NE, E)
    INTEGER :: NA, NR, NC, DIM_NA, DIM_NH, DIM_NE
    REAL ::  F2, A(DIM_NA), H(DIM_NH), E(DIM_NE)
  END SUBROUTINE CSTRUCTFAC
END INTERFACE

[...]

DO II = 1, NR, NR_2
  IND_H = (DIM2_H * (II - 1)) + 1
  IND_E = (DIM2_E * (II - 1)) + 1
  CALL CSTRUCTFAC(NA, NR_2, MAXATOMS, F2, DIM_NA, A, &
                  DIM_NH/TASKS, H(IND_H : IND_H + (DIM_NH/TASKS) - 1),&
                  DIM_NE/TASKS, E(IND_E : IND_E + (DIM_NE / TASKS) - 1))
END DO

[...]
```

Listing 10: Julia Snippet

```
#pragma omp target device(opencl) \
        ndrange(2, jc.window_size[0]/4,jc.window_size[1], 16,1) copy_deps
#pragma omp task out(framebuffer[0;jc.window_size[0] * jc.window_size[1]])
__kernel void compute_julia(float muP0,float muP1,float muP2,float muP3,
                        __global uint32_t * framebuffer, struct julia_context jc);

for (int i = 0; i < iterations; i++) {
    getCurMu(currMu, morphTimer);
    morphTimer += 0.05f;

    compute_julia(currMu[0], currMu[1], currMu[2], currMu[3],
                framebuffer[OTHER_FRAME(display_frame)],jc);

    /* Alter the morphing matrix */
    if (morphTimer >= 1.0f) {
       [...]
    }
    display_frame = OTHER_FRAME(display_frame);
}
```

Listing 11: NBody MPI

```
Particle_broadcast_arguments();
Particle_array_initialize(particle_array, particles);
for (int i = 1; i <= timesteps; i++) {

  Particle_array_calculate_forces_cuda(particle_array, particle_array2, particles,
                                part_per_process, time_interval, first, last);

  #pragma omp target device(smp) copy_deps
  #pragma omp task out(particle_array[0;particles]) in(particle_array2[first:last])
  MPI_Allgather(particle_array,  part_per_process*8, MPI_FLOAT, particle_array2,
                                part_per_process*8, MPI_FLOAT, MPI_COMM_WORLD);
} /* for timestep */
#pragma omp taskwait
```

this is being alleviated with the proliferation of libraries that provide optimized kernels for different hardware accelerators and programming models such as OpenACC or OpenMP 4.0, which can automatically transform sequentially written codes and run them in parallel on hardware accelerators. Hence, in this section we focus on the complexity of the host side of the application to compare between the four different versions of each application: the two native versions (one with CUDA and one with OpenCL) and the two OmpSs versions (which only differs in the name of the kernel called).

To compare the productivity of each version we have used two metrics: the number of lines of code and the number of API/pragma calls on the host side of the application. We have not included the kernel code in the comparison because both CUDA C and OpenCL C kernels have about the same number of lines and are used verbatim on the OmpSs versions. On Table I the first value of each cell is the number of code lines and the second one is the number of calls to the native API or pragmas. If we look at the number of code lines, we can observe that OpenCL is the one that requires more lines of code closely followed by CUDA. The OmpSs version is the shorter one for all the benchmarks. However, the number of lines of code is not really representative of the application complexity because, on one hand most of the initialization code from OpenCL and CUDA can be easily reused from one application to another one but, on the other hand, the complexity introduced by the need to manage explicitly data transfers and synchronization can be far higher than what we can expect from just an increase of one hundred lines of code. The number of calls to the native API/pragmas better shows the reduction in complexity that OmpSs offers. As we can see, most applications only require three pragmas: one of them annotates the *inputs* and *outputs* of the kernel, another pragma provides the parameters to configure the kernel with the *ndrange* clause, and one more pragma waits for the completion of all the generated tasks. However the CUDA and OpenCL versions require a larger number of calls to their API to perform all the explicit synchronization and data transfer. This quantitative analysis already points out that the techniques presented in this paper have a positive impact on the complexity of the application, but from a qualitatively point of view, our approach is much more productive for two

TABLE I: Productivity Comparison in lines of code / API calls

Benchmark	CUDA	OpenCL	OmpSs
NBody	800 / 7	922 / 26	798 / 3
Matmul	240 / 14	292 / 31	133 / 3
Julia	825 / 11	943 / 30	770 / 5
Krist	342 / 15	446 / 30	280 / 3
NBody (MPI)	1030 / 13	1089 / 26	975 / 3

reasons: the first one is that with OmpSs the programmer is only required to have a *local* view of the program, i.e. the programmer only needs to think about the specific inputs and outputs of a kernel, not on how this kernel will interact with the rest of the program on each invocation. The second one is that the run-time system is able to transparently manage any data transfer required and then schedule the kernels on the available resource without any additional effort from the programmer.

Performance: The main reason to use hardware accelerators on HPC environments are their potential to increase applications' performance. Hence, any proposal to improve programmability must also have competitive performance. In this section, we compare the performance of the two native versions of the benchmarks with the OmpSs counterparts. Table II shows for the four versions the execution time of each application in seconds. The performance of the OmpSs versions is equal or better than the hand-tuned native versions. In the Krist benchmark the OmpSs versions are noticeable faster due to the high number of kernels spawned and their fine grained nature, which benefit from data caching and kernel prefetching. In general, the potential overhead of OmpSs task management (creation, synchronization and scheduling) are very small compared to the granularity of the CUDA C and OpenCL C kernels. We expect that the productivity/performance trade-off of OmpSs will increase with the complexity of the applications and the heterogeneity of the hardware.

Scalability: In this section, we evaluate the scalability of the four versions of the applications. Figures 1 and 2 shows the

TABLE II: Performance Comparison. Exec. time (seconds)

	CUDA		OpenCL	
Benchmark	Native	OmpSs	Native	OmpSs
NBody	169.1	166.5	178.2	178.2
Matmul	50.8	51.0	70.6	52.2
Julia	186.5	185.2	73.4	73.1
Krist (C)	284.6	260.8	313.3	250.6
Krist (Fortran)	N/A	217.5	N/A	258.32
NBody (MPI)	247.8	247.2	307.2	306.4

scalability of OpenCL and CUDA versions respectively. Each figure shows the speedup of CUDA or OpenCL applications running with one, two, four and eight GPUs. Each benchmark version uses its own data of Table II as a baseline to calculate the speedup. Notice that the MPI version of NBody has only been evaluated with 2 GPUs per node (with a total of 32 nodes). Both figures show a similar scalability curve for all the applications but the Matmul one, which scales better with the CUDA versions (both native and OmpSs). As we can see, the scalability of OmpSs versions are as good as hand-tuned native versions or even better for some of the benchmarks.

As the number of in-node devices increases, the complexity of the host side of the application will also increase. To deal efficiently with issues such as kernel scheduling across devices, kernel synchronization (with other kernels, host code and/or in-fly data transfers), data locality (to avoid unnecessary data transfers), etc, the host code must be extended in a non-trivial way. Moreover, the scenario evaluated in this paper are still quite homogeneous and relatively easy to exploit because all eight accelerator devices are identical. If the heterogeneity of nodes continues to increase it will be even harder to develop and optimize by hand applications that should simultaneously exploit several hardware devices with potentially different performance and power characteristics (which probably will also depend on the type of workload executed).

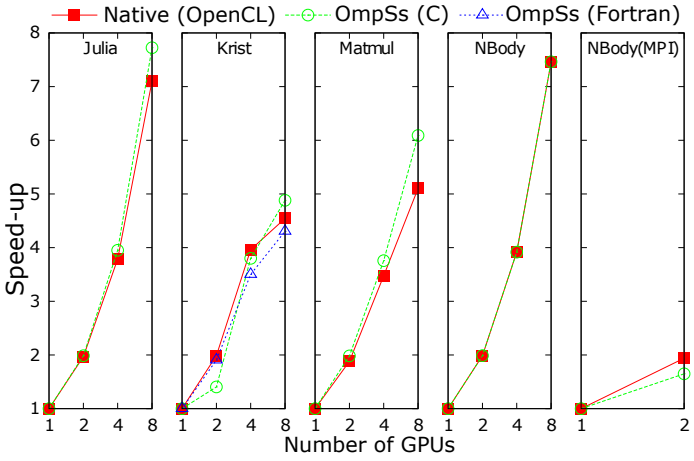

Fig. 1: Scalability comparison between five hand-coded OpenCL benchmarks and OmpSs counterparts (data from Table II used as a baseline)

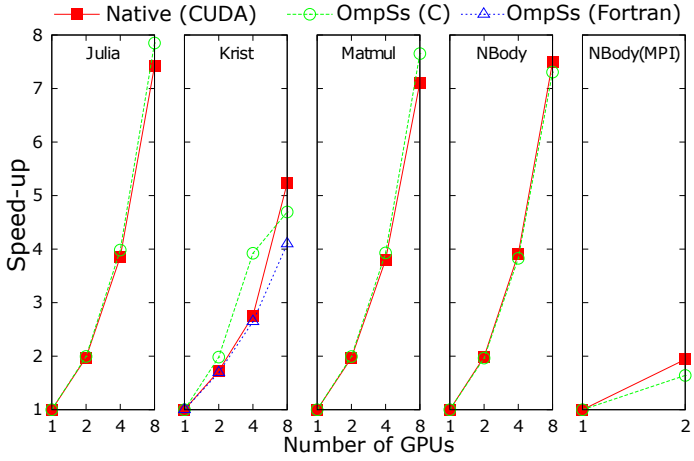

Fig. 2: Scalability comparison between five hand-coded CUDA benchmarks and OmpSs counterparts (data from Table II used as a baseline)

VI. Conclusions and Future Work

This paper shows how OmpSs, a programming model that runs sequential applications in parallel following a data-flow execution model, has been augmented with the *ndrange* clause to exploit accelerators. With the use of a simple and concise syntax, OmpSs can fully replace the host API of both CUDA and OpenCL in a portable way. The productivity evaluation shows how OmpSs clearly simplifies the development of hardware accelerated applications, both quantitatively – in number of code lines and API calls– and qualitatively – relieving the programmer from a whole class of scheduling, synchronization and data transfers issues that otherwise the programmer must have in mind–. In terms of performance, our evaluation shows that OmpSs successfully compares with hand-tuned code directly using CUDA or OpenCL host APIs. Moreover, our opinion is that the productivity and performance advantages of OmpSs will even increase if the current trend of heterogeneity continues. It is worth noting that the approach presented in this paper is specially well suited to port large and complex applications because it can be incrementally applied and really simplifies a mixed use of OpenCL C and CUDA C kernels on the same application.

As a future work, we are investigating how to augment OmpSs with OpenACC or OpenMP 4.0 *pragmas* to automatically generate accelerated code from sequential code, thus simplifying the development of code that run on the accelerator side. We are also considering run-time auto-tuning of kernel parameters on different devices and automatic selection of the best accelerator device given a specific criteria (performance, performance/watt, etc) and workload. We are also working to augment the approach presented in this paper to support kernels written in High-Level Synthesis to exploit FPGAs devices.

Acknowledgment

This work has been developed with the support of the grant SEV-2011-00067 of Severo Ochoa Program, awarded by the Spanish Government and by the Spanish Ministry of Science and Innovation (contracts TIN2012-34557 and CAC2007-00052) by the Generalitat de Catalunya (contract 2014-SGR-1051) and the Intel-BSC Exascale Lab collaboration project.

References

[1] D. Kirk, "Nvidia cuda software and gpu parallel computing architecture," in *Proceedings of the 6th international symposium on Memory management*, ser. ISMM '07. New York, NY, USA: ACM, 2007, pp. 103–104. [Online]. Available: http://doi.acm.org/10.1145/1296907.1296909

[2] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008. [Online]. Available: http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[3] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 555–566. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033345.2033405

[4] *CUDA CUBLAS Library*, nVidia Corporation, Aug. 2010. [Online]. Available: {http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf}

[5] A. Duran, E. Ayguad, 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. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626411000151

[6] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. González, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Optimizing the exploitation of multicore processors and gpus with openmp and opencl," in *LCPC*, ser. Lecture Notes in Computer Science, K. D. Cooper, J. M. Mellor-Crummey, and V. Sarkar, Eds., vol. 6548. Springer, 2010, pp. 215–229.

[7] F. Bodin and S. Bihan, "Heterogeneous multicore parallel programming for graphics processing units," *Sci. Program.*, vol. 17, no. 4, pp. 325–336, Dec. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1662626.1662632

[8] S. Bihan, "Caps openacc compilers: Performance and portability. caps openacc compilers: Performance and portability," Feb 2013.

[9] " OpenMP Application Program Interface, Version 4.0, July 2013." [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[10] L. Koesterke, J. Boisseau, J. Cazes, K. Milfeld, and D. Stanzione, "Early experiences with the intel many integrated cores accelerated computing technology," in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, ser. TG '11. New York, NY, USA: ACM, 2011, pp. 21:1–21:8. [Online]. Available: http://doi.acm.org/10.1145/2016741.2016764

[11] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell, "Offload - automating code migration to heterogeneous multicore systems," in *HiPEAC*, ser. Lecture Notes in Computer Science, Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds., vol. 5952. Springer, 2010, pp. 337–352.

[12] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," *SIGPLAN Not.*, vol. 43, no. 3, pp. 287–296, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1353536.1346318

[13] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, "Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 156–166. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250753

[14] S. Henry, "Modèles de programmation et supports exécutifs pour architectures hétérogènes," Ph.D. dissertation, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, Nov. 2013. [Online]. Available: http://tel.archives-ouvertes.fr/tel-00948309

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par*, ser. Lecture Notes in Computer Science, H. J. Sips, D. H. J. Epema, and H.-X. Lin, Eds., vol. 5704. Springer, 2009, pp. 863–874.

[16] L. Wesolowski, "An application programming interface for general purpose graphics processing units in an asynchronous runtime system," Master's thesis, Dept. of Computer Science, University of Illinois, 2008, http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.

[17] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, pp. 404–418, March 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1512157.1512430

[18] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286120.1286123

[19] D. Bonachea, "Gasnet specification, v1.1," Berkeley, CA, USA, Tech. Rep., 2002.

[20] " AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) ." [Online]. Available: https://www.surfsara.nl/systems/gpu-cluster/intro-opencl

[21] L. Nyland and M. Harris. (2007) Fast N-Body Simulation with CUDA. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.156.7082

[22] " SurfSARA, University of Amsterdam, VU University Amsterdam and CWI." [Online]. Available: http://developer.amd.com/sdks/amdappsdk

[23] " IBM. OpenCL Development Kit for Linux on Power, 2011." [Online]. Available: http://www.alphaworks.ibm.com/tech/opencl