OmpSs@FPGA framework for high performance FPGA computing

Juan Miguel de Haro, Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesús Labarta

Abstract—This paper presents the new features of the OmpSs@FPGA framework. OmpSs is a data-flow programming model that supports task nesting and dependencies to target asynchronous parallelism and heterogeneity. OmpSs@FPGA is the extension of the programming model addressed specifically to FPGAs. OmpSs environment is built on top of Mercurium source to source compiler and Nanos++ runtime system. To address FPGA specifics Mercurium compiler implements several FPGA related features as local variable caching, wide memory accesses or accelerator replication. In addition, part of the Nanos++ runtime has been ported to hardware. Driven by the compiler this new hardware runtime adds new features to FPGA codes, such as task creation and dependence management, providing both performance increases and ease of programming. To demonstrate these new capabilities, different high performance benchmarks have been evaluated over different FPGA platforms using the OmpSs programming model. The results demonstrate that programs that use the OmpSs programming model achieve very competitive performance with low to moderate porting effort compared to other FPGA implementations.

Index Terms—FPGA, reconfigurable hardware, parallel architectures, task-based programming models, High-Level Synthesis

1 Introduction

FIELD Programmable Gate Arrays (FPGA) are becoming popular in recent times, due to their high flexibility to create custom hardware designs in a relatively short time. In a matter of hours, one can test and run an RTL design without having to implement it in silicon, which can potentially take months. This property has been exploited to accelerate many applications, ranging from digital signal processing to high performance computing, including machine learning. The main challenge comes from the fact that traditionally FPGAs are programmed directly with Hardware Description Languages (HDL). These languages, like Verilog or VHDL, offer the maximum flexibility and performance at the cost of programmability. Developing an RTL application directly in HDL usually takes significantly more effort and time than with a high level language.

In this paper we present new features and improvements to OmpSs@FPGA [1], our framework that allows to program heterogeneous systems with FPGAs. With OmpSs@FPGA, the programmer is able to accelerate applications in FPGAs easily and, most importantly, in a short time. The framework is an extension of OmpSs [2], a task-based programming model. Written in C/C++ or Fortran, the user code can be enhanced and parallelized with pragmas, in an OpenMP-like syntax. These pragmas are used to declare parts of the code, like functions, as tasks which are the basic unit of work of the programming model. In the traditional model, a task is executed on a CPU thread, and can run concurrently with other tasks. Dependencies can be declared for any task, avoiding that two tasks that operate over the same memory region execute in parallel, by establishing an implicit execution order through dynamic dependence graphs. In order to generate the executable from the original code, OmpSs uses its own compiler, Mercurium, and runtime system, Nanos++. The compiler processes the pragmas, transforms the code as needed and generates calls to the Nanos++ API [3]. The runtime manages everything needed to execute tasks concurrently, by analyzing task dependencies dynamically and scheduling them to the CPU threads.

OmpSs@FPGA extends OmpSs in the sense that it allows to execute a C/C++ task in the FPGA. To do so, it uses Mercurium to transform the code and build a hardware accelerator through High-Level Synthesis (HLS) of the transformed code. This way, several accelerators, which execute a specific type of task, can be used to easily speedup a previous CPU-only application. Of course, in order to get the best performance, the original code has to target the FPGA which needs a different optimization strategy. Moreover, the framework can coexist with the specific pragmas of the underlying HLS tool, to take advantage of the features provided and boost the accelerator even more. Currently, OmpSs@FPGA supports Xilinx software and FPGAs, thus the HLS software being Vivado HLS.

The main contributions of this article are the following:

- New compiler optimizations for FPGA accelerators that improve memory accesses by the use of a wide memory port.
- A new load/store mechanism that saves redundant memory copies in the FPGA accelerators.
- A new way to pipeline computations and memory accesses inside FPGA accelerators.
- A new complete hardware runtime that in conjunc-

- *All authors are with the Barcelona Supercomputing Center (BSC), Barcelona, Catalonia 08034, Spain. E-mail: {juan.deharoruiz, jaume.bosch, antonio.filgueras, miquel.vidal, eduard.ayguade, jesus.labarta}@bsc.es*
- *J. de Haro, J. Bosch, D. Jiménez, C. Álvarez, X. Martorell, E. Ayguadé and J. Labarta are with the Universitat Politècnica de Catalunya (UPC), Barcelona, Catalonia 08034, Spain. E-mail: {calvarez, djimenez, xavim}@ac.upc.edu*

tion with in-FPGA task creation and synchronization allows local in-FPGA task and dependence management.
- The extension of the framework to target different FPGA boards with diverse host connections in a transparent way to the programmer.
- Evaluation of all the above improvements with different HPC applications, including, to the best of our knowledge, the fastest N-Body implementation over FPGA published up-to-date.

The structure of the article is as follows. Section 2 explains the base implementation used to do our tests. Details on the proposed improvements are explained in Section 3. A qualitative and quantitative evaluation of the system can be found in Section 4. Section 5 lists the related work and finally, section 6 addresses the conclusions and future work.

2 Base OmpSs@FPGA framework

In this section we explain the original OmpSs@FPGA framework that is the baseline of our work. This basic system uses the classical master-slave model that is common to other models like CUDA or OpenCL. In such approach, the host, a Symmetric MultiProcessor (SMP), is responsible for issuing tasks to the accelerators (kernels in the FPGA). The kernels process the data and return a result to the host after finishing their execution. Such process is iterated until all the work is done and then the host finishes the process. In this model, the host is responsible for synchronizing all the work, orchestrating all the resources either in the FPGA or in the SMP when necessary.

2.1 Compiler FPGA-oriented modifications

Figure 1 shows the compilation process in OmpSs@FPGA. A C/C++ source file is read by the Mercurium compiler where a frontend phase splits the code into two different flows: SMP and FPGA. As outline tasks are not supported, this distinction is done through C/C++ function annotation with task declaration pragmas. In OmpSs@FPGA, tasks or kernels can target both SMP or FPGA devices. The SMP part of the code, i.e. main code and tasks that do not have an FPGA target, is separated and its compiler directives are replaced by Nanos++ API calls. The Nanos++ runtime has a dedicated API for FPGA tasks, which uses internally the xTasks library, containing the low-level code to communicate with the FPGA. It is separated from the main runtime because each hardware platform uses different communication protocols, depending on the board vendor and the memory model (e.g. shared like SoCs or distributed like PCIe attached FPGAs).

The FPGA code is also separated and integrated with a wrapper code, which communicates with a hardware runtime inside the FPGA, accesses main memory to load-/store local memories and starts the actual hardware task engine. This wrapper is in fact C++ code with Vivado HLS pragmas. Since generally FPGAs count with on-chip RAM (e.g. Xilinx BRAMs), the kernels can exploit this feature by storing data in this local memory. Depending on the memory model, main DRAM can be shared with the CPU or featured separately in the FPGA board. In both cases

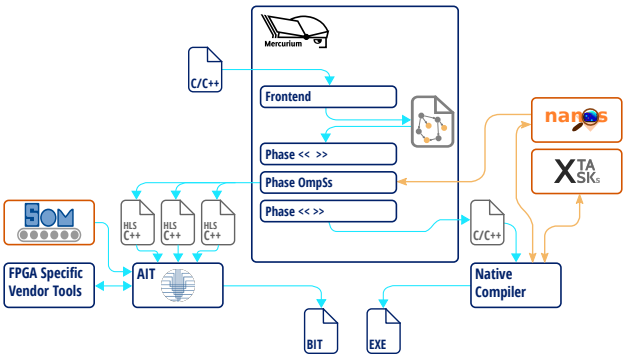


Fig. 1. Mercurium compiler process

accessing local memory is faster, and OmpSs@FPGA allows to declare arrays stored in local memory and use them inside a task. The difference with other approaches like CUDA or OpenCL is that the local copies are automated in the wrapper and thus transparent to the user, who otherwise has to code them explicitly in the kernel. Once the code is transformed by Mercurium, it is passed to AIT (Accelerator Integration Tool). This tool feeds all the high-level codes to the vendor-provided tools and integrates them inserting the proper connections with the hardware runtime and the FPGA I/O pins in order to generate the final bitstream.

This integrated compilation process has some useful features such as compilation of the whole system (bitstream and executable file) from a single command and automatic connection and integration of the hardware design, reducing the complexity of an otherwise error-prone process.

2.2 FPGA hardware runtime

As commented in section 2.1, AIT inserts a small hardware runtime, the Smart OmpSs Manager (SOM), in the final hardware design. This runtime basically receives commands from the software runtime and forwards them to the related accelerator. In the case of having more than one accelerator available for any given task, a simple round-robin scheduler dispatches the task to one of the accelerators. After the execution finishes, it informs the hardware runtime which forwards the information to the software control.

2.3 OmpSs code

Listing 1 shows a function *vecSum* annotated with two OmpSs pragmas in the first two lines. As it can be seen in the listing, the first pragma specifies that the function is going to target an FPGA device (`target device(fpga)`) while the second pragma specifies that the function is going to be a task that has two dependencies: a vector of 16 elements `a` that is going to be input and output of the function, and another vector `b` also with 16 elements that is going to be an input of the function. From this second pragma Mercurium extracts the necessary information for inserting the code to perform the data copies of the dependencies to/from the FPGA external RAM if necessary. Although this copy is the default behavior, clauses `copy_in`, `copy_out`, `copy_inout`, `copy_deps` and `no_copy_deps` allow to specify the copies individually or force/disable the copies of all the dependencies.

```
#pragma omp target device(fpga)
#pragma omp task inout([16]a) in([16]b)
void vecSum(float a[16] , float b[16]) {
   for (int i = 0; i < 16; ++i)
      a[i] += b[i];
}
```

Listing 1: Minimal example of C code with OmpSs pragmas

By default, the compiler also uses local memory in order to optimize memory accesses. A variable stored in local FPGA memory is declared and data will be copied to/from this variable by the wrapper prior to/after the kernel execution. Accesses to the variable will reach the local memory instead of the global one. Although the default copy behavior is intended to provide good performance and programmability, it may be modified by the programmer to better adapt to specific cases. The clause `localmem_copies` enforces local copies of the task data (the default behavior), the clause `no_localmem_copies` disables copies (so the user code accesses to main FPGA memory) while the clause `localmem(...)` specifies a list of shaping expressions that define the data that must be copied by the accelerator wrapper.

In order to execute multiple instances of the same task call in parallel, resources, i.e. accelerators, can be replicated. The clause `num_instances(N)` allows specifying the number of times that a task accelerator is instantiated in the FPGA. All specific OmpSs@FPGA clauses and their usage are available in [4].

3 New Features and Improvements

3.1 Wide and unified memory port

Each accelerator needs to access memory to get data, whether it is stored in a local memory or directly accessed. Vivado HLS tool uses an AXI4 interface [5] to handle memory reads and writes, which is a very common interface used in the FPGA and ASIC world. The most straightforward solution to convert pointer accesses in C to AXI transactions is through the creation of an independent memory port per each pointer argument. Vivado HLS syntax to read and write from an AXI interface is the same as in C to access a pointer or array.

However, there are two main problems with this implementation. First, using a different interface for each pointer or array argument can easily outnumber memory access ports. The number of real memory ports on an FPGA system is quite low, for instance, discrete boards like the Xilinx Alveo family have only one per DDR module, whereas SoCs like the Xilinx Zynq UltraScale+ have up to six. Reducing several ports to one is indeed possible with an AXI interconnect, but it takes resources and hinders design routability. Moreover, performance-wise it is better not to have more than one memory port in a single Vivado HLS module. Our efforts to make the tool use more than one port have been unsuccessful, since Vivado HLS seems to always respect the access order between all external interfaces.

The second and most important handicap of the mentioned implementation is the bandwidth. The default behavior is to use a data bus with the same bit width as the data type. However, the FPGA memory controller may allow a wider data bus. Therefore, in order to exploit the memory bandwidth of the system, the AXI data bus used has to be as wide as supported by the memory controller. This way, each cycle the accelerator can read multiple data elements.

To conclude, to remove redundant resources and improve performance, we added the possibility to use a single memory port with a configurable data width. Specified as a Mercurium variable at compile time, the user can provide the bit width of the data bus, which has to match the FPGA memory data width to get the maximum performance. This port is shared across all array arguments, thus limiting the total required AXI interfaces to one per accelerator. Using this feature is only possible for array arguments that are stored in local memory. If the task directly accesses memory, the shared memory port is not supported. It could be possible to use it by replacing every access to the pointer with a cast to the data type of the shared port, which could even have different bit width. However, to benefit from the bandwidth of the wide memory port would require significant changes to the compiler. It should provide the user the possibility to load/store vector data types, such as the ones from Vivado HLS. At the time of writing we have not found any use case that would benefit from this feature without the use of local memory.

Listing 2 shows a portion of the wrapper HLS code generated by Mercurium from listing 1. The resulted wrapper contains a single memory interface, `mcxx_data` with a 512-bit data bus and two local memories. The compiler also generates the necessary code to copy the data from main memory, mainly loops enhanced with Vivado HLS pragmas to maximize bandwidth. Line 10 of listing 2 shows an `HLS pipeline` pragma used to pipeline the memory load with the store to local memory. The inner loop is fully unrolled automatically by the tool, and the Initiation Interval (II) depends on the partition of the local memory. In order to generate the unified memory interface with a specified width, the argument `--variable=fpga_memory_port_width:<width>` has to be provided to the compiler in the invocation command.

There are some restrictions to take into account when using the unified port. The HLS generated code uses an interface declared as a pointer to an unsigned integer type with the specified width. Therefore, all accesses must be aligned to that type. In the example shown in listing 2, the memory port is used to copy from a 512 bit unsigned integer pointer to a local array of 16 floats. Hence the address stored in *param[0]* has to be aligned to 64 bytes or the lower bits will be discarded in the division of line 12. Although unaligned accesses are supported in Mercurium with a compiler variable, they add significant resource overheads due to large bit shifts of non-constant length and they are disabled by default. The preferred approach is allocating memory aligned to the required width in software. Line 13 of listing 2 imposes that the accessed type width has to be multiple of the memory port width. Moreover, the union used to do the casting between types, mainly to avoid float to integer conversion, uses a 64-bit unsigned integer type. As a result, the casted type can not have more than 64 bits, and due to union restrictions it cannot have a non-trivial

```
void vecSum_hls_automatic_mcxx_wrapper(
   ...//Input/output streams from/to hwruntime
   ap_uint<512> *mcxx_data) {
#pragma HLS interface m_axi port=mcxx_data
static float a[16];
static float b[16];
...//Read task parameters, address of a in param[0]
int su = sizeof(ap_uint<512>), sf = sizeof(float);
for (j=0; j < (16*sf)/su; j++) {
#pragma HLS pipeline II=1
   ap_uint<512> tmpBuffer =
     *(mcxx_data + param[0]/su + j);
   for (k=0; k < su/sf; k++) { //fully unrolled
      union {
         unsigned long long int raw;
         float typed;
      } cast_tmp;
      cast_tmp.raw = tmpBuffer((k+1)*sf*8-1,k*sf*8);
      a[j*(su/sf)+k] = cast_tmp.typed;
} }
...//Read b, execute task code and copy back a
}
```

Listing 2: Part of the Vivado HLS wrapper code generated by Mercurium, that copies 16 floats from an AXI memory interface with a 512 bit data bus to a local memory

constructor.

Another important concept to have in mind is that in order to get the maximum bandwidth (II=1 in the copy loop), the local array has to be able to be written or read at the same rate as the memory. I.e. it has to have enough ports to read or write a memory word in one cycle. For instance, in the example the *a* and *b* memories should have at least 16 ports. Translated into Vivado HLS terms, they should have a cyclic partition of factor 8 if implemented as BRAMs, since each one has two ports.

3.2 Pipelined load-computation-store loops

A task with local memories requires to copy the input data at the start of the execution, and to store the output after finalization. This adds an overhead that is more significant as the task granularity decreases. Nevertheless, some iterative applications allow loading next iteration data in parallel with computation, or previous iteration data store. In these cases, it is possible to remove the load or store at the beginning or end of the execution, and hide this time within the computation phase. The idea is illustrated in figure 2.

Pipelining this process reduces the necessary local memory since the loop body only needs to access a subset of the whole original array. To implement this feature however, it is needed to use double buffering. While one of the buffers is used by the loop body, the other is read or written from/to memory, which doubles the number of local memory ports. This optimization does something similar to the dataflow pragma of Vivado HLS, but it is more flexible since the dataflow is applied to the whole code region and affects all variables. Also using the HLS pragma and making it work would require similar transformations to the user code that the compiler is already doing automatically in our proposal.

Depending on the memory and compute latencies, this technique is able to transform a computation kernel to a fully compute-bound or memory-bound problem. The loop logic waits for the load/store and computation parts before starting the next iteration, independently of the latencies of

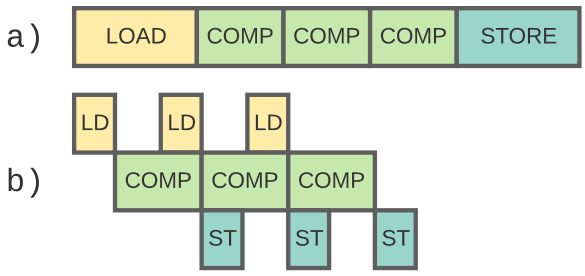


Fig. 2. Timing diagram of (a) load-computation-store loop and (b) its pipelined version

each part. Thus, the resulting latency is the maximum of both parts. The following formula describes the expected execution time of an FPGA task with a single loop and local memories.

$$n_{it} \times L_{ld} + n_{it} \times L_c + n_{it} \times L_{st}$$

Where n_{it} is the number of iterations of the loop, L_{ld} the latency to load all the data required on a single iteration, L_{st} the store latency for the data generated in one iteration, and L_c the computation latency of each iteration. Once the pipelining is applied to the loop, the expected execution time of the task becomes:

$$L_{ld} + max(L_{ld}, L_c) + (n_{it} - 2) \times max(L_{ld} + L_{st}, L_c) + max(L_{st}, L_c + L_{st})$$

In this case, execution time depends on the ratio between memory and computation latencies. If $L_c > L_{ld} + L_{st}$, with a big enough n_{it} the potential speedup becomes:

$$speedup \approx 1 + \frac{L_{ld} + L_{st}}{L_c}$$

On the opposite case, if $L_c < L_{ld} + L_{st}$ and n_{it} is big enough, the fraction is flipped. This means that the potential speedup of the whole task execution ranges from 1 to 2.

3.2.1 *Compiler transformations*

Though we have successfully accelerated the matrix multiply with this feature (see section 4), it has not been yet fully integrated into the programming model at the time of writing. In this paper we propose the necessary syntax to express a pipelined loop in a simple way. We also propose compiler transformations to generate Vivado HLS code that automatically pipelines the loop. An example of the proposal is in listing 3. This code performs a pipelined vector addition with blocks of 8 elements. The `pipeline_in`, and `pipeline_out` clauses are new additions to the OmpSs syntax, whereas the `linear` clause is taken from the OpenMP standard [6].

The first type of clause is used to declare which arrays have to be loaded or stored in pipeline mode, and to declare the size of the local buffer. The second type is used to declare how the addresses of each array are incremented or decremented. By encapsulating the loop body and loads/stores in different functions, the HLS compiler is able to schedule calls without dependencies in the same cycle. In the example

```
void vectorAdd(const float* a,
               const float* b,
                     float* c, int n) {
  #pragma omp for pipeline_in([8]a, [8]b) \
  pipeline_out([8]c) linear(a:8, b:8, c:8)
  for (int k = 0; k < n; ++k)
    for (int i = 0; i < 8; ++i)
      c[i] = a[i]+b[i];
}
void vectorAddTransformed(int n, float* c,
               const float* b, const float* a) {
  float a1[8], a2[8];
  float b1[8], b2[8];
  float c1[8], c2[8];
  load(a1,b1,a,b); //n is multiple of 2 and n >= 2
  for (int k = 0; k < n-2; ++k) {
    loadStore(a2,b2,c2,a+k*8,b+k*8,c+(k-1)*8,k);
    loopBody(a1,b1,c1);
    ++k;
    loadStore(a1,b1,c1,a+k*8,b+k*8,c+(k-1)*8,k);
    loopBody(a2,b2,c2);
  }
  int k = n-1;
  loadStore(a2,b2,c2,a+k*8,b+k*8,c+(k-1)*8,k);
  loopBody(a1,b1,c1);
  store(c1,c+k*8);
  loopBody(a2,b2,c2);
  store(c1,c+k*8);
}
```

Listing 3: Proposal of OmpSs pragma syntax (*vectorAdd*) and generated Vivado HLS code (*vectorAddTransformed*) to pipeline loads/stores with computation

of listing 3, the first *loadStore* function call of *vectorAddTransformed* is scheduled alongside the first *loopBody* call. The other two calls are also scheduled together after the first two. Variable *k* is passed as an argument to *loadStore* to avoid doing the store in the first iteration. The compiler unrolls the loop by a factor of 2 to code the double buffer explicitly. Therefore, the first and last iterations have to be expressed differently to generate correct code. Before the first iteration of the loop body, the data has to be loaded in the first buffer, thus a separate load function has to be called before the start of the loop. To avoid loading out of bounds data in the last iteration, this one is placed outside the loop. For readability purposes, the example assumes that *n* is multiple of two and hence after the loop there are two calls to *loopBody*, but only one to *loadStore*.

3.3 Dynamic copy optimization

Another way to mitigate the load/store overhead of an FPGA task is to use runtime information to suppress the local copies when possible. This can be achieved when two consecutive tasks operate over the same region of memory on the same accelerator. To detect these situations, the hardware runtime analyzes the ready queue, which is a buffer composed of multiple circular sub-queues, one per accelerator. Tasks that are scheduled for execution, sent by the host or by the hardware runtime itself, wait in this queue while the accelerator is busy executing another task.

In the load case, the first task fills the local memory, but the second one does not need to access to memory again. For writes, the first task does not need to store if the next task in the queue will perform the same copy, since the results would be overwritten. Applying these optimizations

Algorithm 1: Dynamic copy optimization

Input: `ready_task_queue`: circular queue with tasks made of the same number of arguments and copy flags, `current_slot`: index of the current slot in the queue

```
task_current = ready_task_queue[current_slot];
task_next = ready_task_queue[current_slot+1];
for i = 0; i < #args; i = i+1 do
    flags_current = task_current.flags[i];
    flags_next = task_next.flags[i];
    if task_current.args[i] == task_next.args[i] then
        if outCopyEnabled(flags_next) then
            suppressOutCopy(flags_current);
        if inCopyEnabled(flags_current) or
         chainBitEnabled(flags_current) then
            if inCopyEnabled(flags_next) then
                enableChainBit(flags_next);
            suppressInCopy(flags_next);
```

in both directions maintains correctness for `inout` copies. Each array argument has some flags associated with it, which are interpreted by the accelerator wrapper to know whether data has to be transferred to/from memory at the address specified in the argument value. These flags depend on the `copy_in/out` and `in/out` clauses in the target and task pragmas respectively. Before reaching the accelerator they are manipulated by the hardware runtime following algorithm 1. The chain bit is used to detect and thus disable chains of `in` copies over the same address, since in this case all except the first one can be suppressed. Chains of `out` copies are also disabled except for the last task.

This optimization is activated by default in the hardware runtime and operates transparently to the programmer so it has not been evaluated separately.

3.4 Internal task creation and management

Managing task creation, dependence resolution and task scheduling in software can become the bottleneck of an application, due to the CPU-FPGA communication latency and the overhead of the runtime, which is also influenced by the CPU speed. As it can be seen in section 4, moving these functions to hardware can improve significantly the performance.

In [7] it is introduced the capability to create tasks in an FPGA accelerator, i.e. hardware task nesting. However, it is limited to tasks without dependencies. Tasks with dependencies are forwarded to the CPU which significantly degrades performance. In this paper we overcome this limitation by adding a full dependence manager to the OmpSs@FPGA framework.

OmpSs@FPGA presents to the user the possibility to choose between two hardware runtimes: Smart OmpSs Manager (SOM, previously Task Manager) and Picos OmpSs Manager (POM). Both can execute the same code and are transparent to the programmer, but they present relevant performance differences.

TABLE 1
Absolute and relative primitive usage taken from Vivado synthesis on a Xilinx ZynqU+ for each OmpSs@FPGA hardware runtime and Picos

#primitives (% usage)	LUT	FF	BRAM18K
SOM (simple)	1743 (0.64%)	3288 (0.6%)	0 (0%)
SOM (extended)	4430 (1.62%)	5418 (0.99%)	7 (0.38%)
Picos Daviu	1738 (0.63%)	1714 (0.32%)	45 (2.47%)
POM (with Picos)	6952 (2.54%)	7819 (1.43%)	56 (3.07%)

Both SOM and POM manage task creation and scheduling of hardware/software tasks and optimize copies on the same accelerator (section 3.3). Nevertheless, SOM does not have a dependence management module, hence it relies on the software runtime. POM however includes Picos Daviu, an improved hardware dependence manager based on the Picos++ architecture [8]. It determines which tasks are ready for execution and forwards them to the hardware scheduler, which decides where to execute them. The main advantage of POM over SOM is the minimization of host-FPGA communication. Moreover, Picos manages dependencies much faster than the software runtime since it is a specialized hardware module, optimized to quickly build and analyze dynamic dependence graphs. Using POM, the programmer can offload big parts of the application to the FPGA, removing the need for host interaction between different kernel executions.

For simple or highly parallel applications it is still useful to use SOM for its reduced resource requirements. Picos Daviu utilizes memories (implemented as BRAMs and LUTRAMs) and extra logic which may limit the place&route of the design in some extreme cases. In addition, if the user does not need FPGA task nesting, SOM automatically removes unnecessary modules and leaves only the relevant logic. These operating modes are called simple and extended, the resources of which are quantified in table 1 alongside POM and Picos Daviu resources. The table shows Vivado synthesis results targeting a Xilinx Zynq Ultrascale+ board in terms of absolute primitive count and relative FPGA usage. LUT and FF refer to LookUp Tables and Flip-Flops, the basic FPGA primitives to build logic. BlockRAMs (BRAMs) are low latency memories embedded in the FPGA fabric. SOM (simple) is the runtime using fewer resources since it lacks Picos Daviu, the logic to handle task creation and the scheduler. In this mode SOM is controlled completely by the host, which has to do all the task management. Enabling extended mode of SOM gives more flexibility to the FPGA at the cost of more resources. In this mode, tasks can be created by accelerators and dependencies are forwarded to the host, but the scheduling is moved to the hardware. Finally, the POM hardware runtime incorporates dependence management, thus providing the best performance and consequently taking the most resources. Picos Daviu alone takes as many LUTs, half of FFs and six times more memories than SOM in simple mode. This is due to the necessity to store task runtime information, e.g. task dependencies.

3.4.1 POM internal design

Both SOM and POM share most of the internal code, written in Verilog and SystemVerilog. The interface with the host

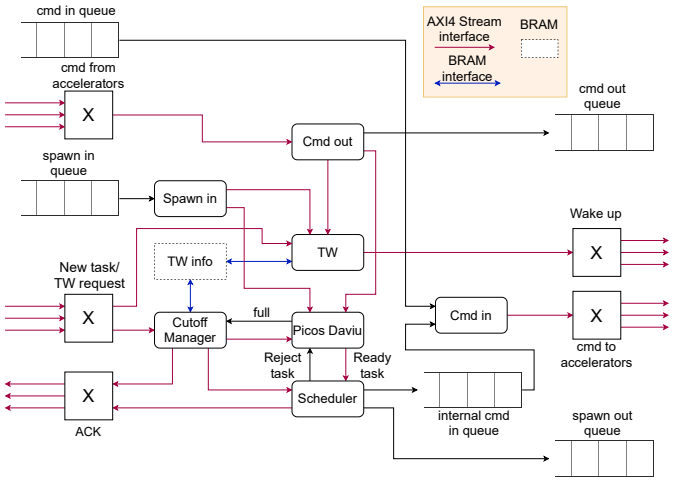


Fig. 3. High-level diagram of POM's internal design

is implemented with BRAM queues, which are easier to interface with RTL code than main memory. Internally, most communication is done through AXI-stream interfaces. Figure 3 shows how internal modules are interconnected. The host sends commands through the *command in* queue also called ready queue, which can be tasks ready for execution or hardware instrumentation commands. Tasks created by accelerators pass through the scheduler first (when ready), which decides where to execute the task. SMP tasks are put in the *spawn out* queue, which is read by the host. FPGA tasks are put in an internal queue, and the *command in* module acts as a multiplexer between the host queue and the aforementioned one. SMP finished tasks are put in the *spawn in* queue. The TaskWait (*TW*) module controls the number of created and finished tasks per context. In applications with nesting, there can be several contexts creating tasks at the same time, therefore it contains a memory to store the necessary information for each task creator. This information is needed when a hardware accelerator synchronizes with its child tasks. The *Cutoff Manager* decides if a new task, created on the accelerator, has to go to Picos or the Scheduler. This is needed because Picos does not accept tasks without dependencies, thus these have to be redirected to the scheduler. Moreover, it handles the allocation of the memory used to handle taskwait requests.

3.4.2 Picos Daviu architecture

Picos Daviu bases its architecture on Picos++ [8], but improving the actual algorithm to detect dependencies. It is made of five components: GateWay (GW), Task Reservation Station (TRS), Dependence Chain Tracker (DCT) and Ready Task Dispatcher (RTD). The GW preprocesses dependencies to detect repetitions in the same task and distributes task data between the TRS and the DCT. The TRS keeps track of how many dependencies are free for each task, thus it notifies the RTD when a task can be executed, and also receives notifications of finished tasks. Then, it notifies the DCT to wake up potential dependencies of other tasks. The RTD is responsible for communicating with the POM scheduler, providing a ready task in the expected format.

The DCT is the critical component of Picos Daviu because it performs the actual dependence detection, and notifies the TRS when a task dependence becomes free. To do that, it uses a hash table of binary search trees not autobalanced. The hash function used is an XOR of randomized values in a ROM, indexed with the dependence address. The memory that stores the binary trees is called Dependence Memory (DM), and it only stores dependence addresses once. It is used to know if a new dependence address matches with another from a previous task. To analyze if there is a real dependence, the DCT uses another memory called Version Memory (VM). This one uses linked lists, using one entry for each task that depends on the same address. Figure 4 shows how the hash table, DM and VM are used to keep the dependence graph and determine if a new incoming dependence address is ready or not. As it can be seen, there are two types of VM nodes, depending on the direction of the dependence. Each DM node points to the last `out` dependence, or `in` in case there are no `outs`. On the other hand, the `in` dependencies are pointed by the last `out` dependence at the time they arrived at the DCT. These chains of `in` dependencies have to wait for the task with `out` direction to finish before being ready. In the example of figure 4, the task identifiers represent the order in which they arrived at the DCT. The dotted lines show how the pointers change when the last dependence from task 0x04 arrives at the system. First, the only task that can be executed is 0x00, assuming that all tasks have only one dependence. When it finishes, the TRS notifies the DCT and the next task in the VM list (0x01) is marked as ready. When this one finishes, the two `in` tasks (0x02 and 0x03) are marked as ready concurrently since there is no real dependence between both. Finally, when both finish, the last task (0x04) is ready to execute.

Although the OmpSs programming model distinguishes between three types of directions, `in`, `out` and `inout`, Picos Daviu only supports two types since `inout` can be treated as `out` without losing correctness nor performance.

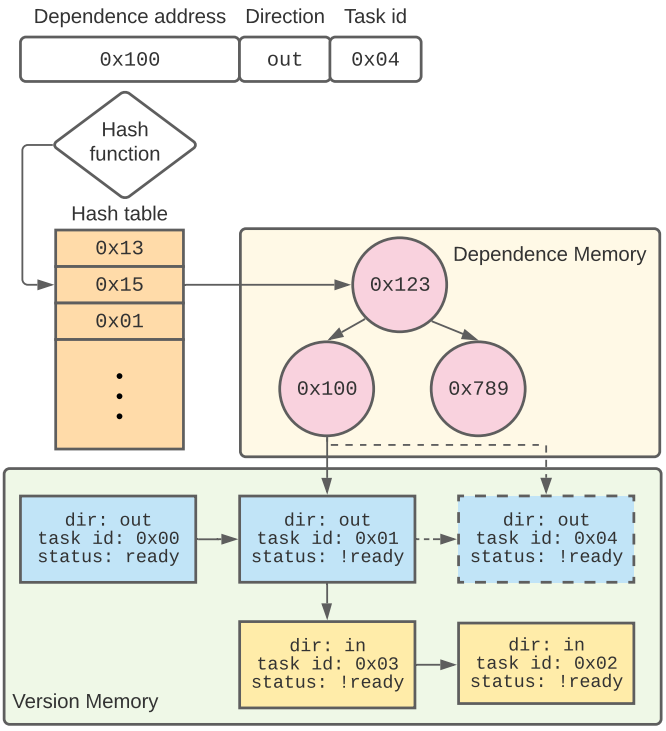


Fig. 4. Insertion of a new dependence in the Picos Daviu DCT structures. Dotted lines represent the DCT state change after the last insertion.

4 Performance Evaluation

We have selected a set of benchmarks to evaluate the peak performance we can get with OmpSs@FPGA: matrix multiply, N-body, Cholesky and Spectra.

4.1 Experimental setup

Except for the matrix multiplication, the benchmarks are coded only in C using OmpSs@FPGA and Vivado HLS pragmas. The only difference of the mentioned benchmark is that the main kernel HLS code is not generated by Mercurium. The pipeline optimization introduced in section 3.2 is not supported by the compiler, so it is implemented manually. The Vivado and Vivado HLS version used is 2020.1.

In our experiments we target two different boards: the Xilinx Zynq Ultrascale+ MPSoC ZCU102 (XCZU9EG-FFVC900) and the Xilinx Alveo U200 (XCU200-FSGD2104). The first one uses shared memory between four ARM Cortex-A53 cores @ 1.1GHz and the FPGA. On the other hand, the Alveo is a stand-alone discrete board and needs to be connected to an external host through PCIe. The server attached to the FPGA includes an Intel Xeon CPU X5680 @ 3.33GHz, with two NUMA nodes, six cores per socket and two threads per core. In these types of systems the communication is slower, but nevertheless the board features its own memory (plus the host memory) which is usually bigger than in SoCs. In addition, the FPGA itself includes more resources and the CPU is faster. Table 2 compares the resource count of each FPGA. The Alveo has roughly four times more LUTs and FFs, double DSPs and BRAM18K. Another advantage of the Alveo is that it features a new type of memory, the UltraRAM (URAM). These memories are significantly bigger than BRAMs (288kb against 18kb) and have a very similar interface and low read/write latency.

4.2 Benchmarks

4.2.1 Matrix multiplication

The matrix multiplication benchmark is a well-known embarrassingly parallel application with a regular dependence pattern. The application operates with three square matrices of size $N \times N$, A, B and C, and computes $C = C + A \times B$. In our implementation, we use float as the data type of the matrices. To program this with a task-based programming model, all matrices are decomposed in square blocks of $BS \times BS$ elements. A task just performs the multiplication of matrices with a fixed square size BS. Therefore, the amount of tasks created depends on the matrix size and the block size. Only the tasks that operate over the same block of C have a real dependency, thus the number of parallel tasks grows quadratically to N/BS. Though the typical implementation uses the *ijk* loop order ($C[i][j]$+=$A[i][k] \times B[k][j]$), we change it to *kij*, pipelining the i loop and fully unrolling the j loop. This way, in each cycle an entire row of a B

TABLE 2
Xilinx Zynq UltraScale+ and Alveo U200 resources

	LUT	FF	DSP	BRAM18K	URAM
Alveo U200	1182240	2364480	6840	4320	960
ZCU102	274080	548160	2520	1840	0

block is multiplied with an element of an A block and accumulated in a row of a C block.

This design has limited scalability due to the high amount of ports required by the C and B blocks, linearly dependent on the II of the pipelined loop. To ease this effect we can increase the II which consequently reduces the number of concurrent multiplications proportionally. For instance. although the latency of the computation is doubled, if we double both the block size and the II, the throughput is maintained whereas the computation-to-memory ratio is increased. I.e. with the same matrix size, the amount of tasks is reduced and thus the number of loads/stores between main and local memories. In addition, the amount of computation of a single accelerator grows by a factor of BS^3 whereas the required memory only by a factor of BS^2.

Using the aforementioned optimizations, we are able to fit up to 3 accelerators with $BS = 256$ and $II = 2$ on the ZCU102 board, and 5 accelerators with $BS = 384$ and $II = 3$ in the Alveo board, both working at 300MHz. Furthermore, it is possible to use the load-computation-store pipelining in this benchmark. While the main loop is using a row of B and a column of A, it is possible to load in advance the next row and column of both matrices. However, one restriction to make it work efficiently is that A has to be transposed to allow issuing read bursts of consecutive addresses to memory (blocks are stored by consecutive rows).

Results of different versions of the benchmark can be observed in figure 5, which shows the performance in GFlops for the ZCU102 and Alveo boards. The block size, II and frequency are the same for each version. In the figure we show the performance impact of the mentioned optimizations, applied incrementally. I.e. each bar uses all the optimizations of the versions at its left. The first uses simple SOM and 32-bit memory port width on each accelerator. The wide port version increases the memory port width to 128 on the ZCU102 and 512 on the Alveo. This one is the optimization with highest impact, doubling performance in both cases because a significant time of a task execution is spent copying data. The load-computation-store pipelining helps to mitigate this effect, especially in the ZCU102. The Alveo FPGA does not benefit as much from this optimization because it has more accelerators and fewer AXI slaves on the memory controller than the ZCU102. When more than one accelerator access memory, they start fighting in the last level interconnect, affecting each other's latency. Since the pipelining optimization adds loads in the computation loop, all accelerators are slowed down due to the high memory latency. The final version shown (SOM(e)), uses hardware task creation and SOM in extended mode. This last optimization exploits the round-robin scheduler of the hardware to send tasks with an `inout` dependency on the same C block to the same accelerator. This way, the runtime triggers the copy optimizations on C (section 3.3), while A and B copies are parallelized with the computation.

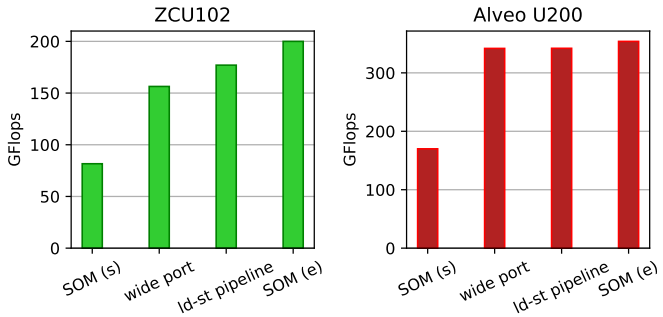


Fig. 5. Matrix multiply performance

With this optimization, the number of copies is reduced and the concurrent access effect is mitigated. Dependencies are not used in this benchmark because the scheduler would receive ready tasks in an unpredictable order, depending on the order the tasks finish. If the tasks are created without dependencies, then this order is enforced because they are scheduled right away. In this case, POM is not needed.

Pipelined accesses have a margin to tolerate concurrent accesses, the loop body time, before affecting performance. However, it is important to note that these last two optimizations in the Alveo need to work together in order to have some effect. That is, if the SOM(e) optimization is applied to the wide port there is no performance improvement. Contrary to the other cases the memory characteristics make it necessary to apply both optimizations to obtain some gains and break the memory bottleneck. The final peak performance achieved is 199 GFlops on the ZCU102 with a matrix size of 4096x4096 and 353 GFlops on the Alveo with a matrix size of 9600x9600.

4.2.2 *Cholesky*

The Cholesky benchmark performs a Cholesky decomposition of a Hermian, positive definite matrix into a lower triangular matrix, which multiplied by its transpose results in the original matrix. I.e. the application generates an output matrix L from an input C, assuring that $C = L \times L^T$ providing that C fulfills the restrictions.

The strategy to taskify this benchmark is the same as the matrix multiply. C is distributed in consecutive blocks of $BS \times BS$ elements, and each task operates over one or more blocks. The code uses four kernels: `gemm`, `trsm`, `syrk` and `potrf`. The `gemm` implementation uses the same strategy explained in section 4.2.1. However, the load-computation-store pipelining can not be applied directly because all blocks belong to the same matrix, but only one of them needs to be transposed. The `trsm` and `syrk` have similar access patterns to the blocks as `gemm` and can also benefit from loop unrolling and pipelining. On the other hand, the peculiarity of this benchmark compared to the others is that the `potrf` kernel is hard to accelerate in FPGAs due to the memory access patterns. Previous evaluations [1] demonstrate that it is faster to execute it in the host rather than in the ZCU102 FPGA. A similar study on the Alveo led to the same conclusions. We use the OpenBLAS [9] implementation of the kernel, which is in fact a Choleksy decomposition of a single block. When using the hardware

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3086106

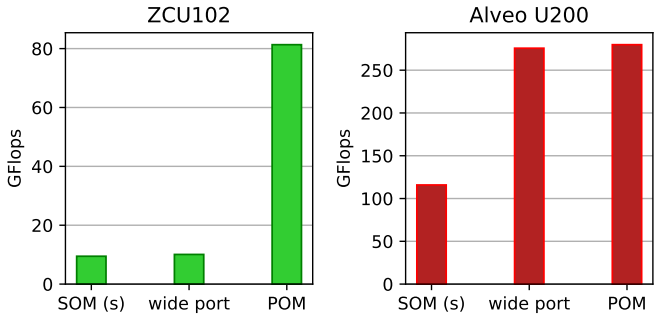


Fig. 6. Cholesky performance

runtime (SOM/POM) and creating tasks on accelerators, the FPGA sends the `potrf` ready tasks to the host through the *spawn in* queue. Then, the software runtime is responsible for performing the necessary copies between FPGA and host memories.

The challenge of Cholesky is to use the right amount of accelerators per kernel. The number of `gemm` tasks grows as a cubic power of the problem size, whereas `syrk` and `trsm` only grow with a quadratic exponent. Therefore, there must exist more accelerators for `gemm` than for any other kernel. We found that only one instance of the non-critical kernels is enough, because the `gemm` accelerators are the bottleneck of the application. This property can be exploited to gain more performance, by intentionally slowing the `syrk` and `trsm` kernels to give more space for `gemm`. This is achieved using a higher II for the two first loops and a lower one for the latter loop.

Our final implementation in the ZCU102 uses five `gemm`, one `trsm` and one `syrk`, all with $BS = 64$ and $II = 1$ at 350MHz. On the Alveo, we managed to put four `gemm` with $II = 2$, one `syrk` and `trsm` with $II = 4$, and $BS = 256$ at 300MHz. In figure 6 we can see the performance in GFlops of the simple SOM, wide port and POM versions for the ZCU102 and Alveo. Like in the matrix multiply, the simple SOM version uses a single 32-bit port per accelerator. Nonetheless, for this one and the rest of benchmarks, the last improvement uses internal task creation with POM because tasks have dependencies. We can observe that the ZCU102 FPGA, contrary to the Alveo, benefits greatly from the use of the hardware runtime. The main reason is the task granularity, being too fine-grained for the software runtime which is run on a cortex-A53. POM is able to manage very short tasks instead, without suffering from significant runtime overheads. The Alveo, on the other hand, does not get a big improvement due to the high parallelism of the `gemm` kernel. In this case, although the CPU task creation and sending are slower, this time is hidden in part by the ready queue of the hardware runtime.

To conclude, the ZCU102 reaches 81 GFlops with a matrix of 8192x8192 floats, and the Alveo 286 GFlops with a matrix of size 10240x10240. Yang et. al. [10] also provide a Cholesky decomposition for FPGAs, as part of a compressed sensing algorithm, programmed in VHDL. However, they report performance for small matrices only (2048x2048) with 21 GFlops in a Virtex-5 XC5VLX110T FPGA.

4.2.3 N-body

The N-body simulation calculates the interaction of a set of particles with different masses over a period of time. The interaction force is calculated with Newton's law of gravity, where the force between two particles is calculated with the formula

$$F_{ij} = \frac{G \times m_i \times m_j \times (p_j - p_i)}{||p_j - p_i||^3}$$

Where F_{ij} is a 3-dimensional vector with the force between particles i and j applied to particle i. The magnitude of the vector is the same for the force applied to particle j but in opposite direction. The particles are represented with a 3-dimensional vector as a position in the space p_i and a mass m_i. The gravitational constant is represented as G in the formula.

The input of the algorithm is a set of particles with initial positions, velocities and masses, the number of steps to simulate and the time interval between each step. The output is the set of final positions after simulating all the steps, or alternatively all the intermediate positions for each iteration. In each step there are two phases. The first one consists of calculating the accumulated force of each particle against each other. This part is computationally intensive as the number of force interactions is n^2 where n is the number of particles. In the second phase the positions and velocities are updated according to the force vector and the time interval, using the Euler method.

To implement this application efficiently with FPGAs, the particle data is distributed in tiles of consecutive vectors of the same attribute. I.e. each dimension of the positions, velocities and masses is separated in consecutive vectors of TS elements, forming a single tile. If the original data is not shaped with this format, it can be easily converted during the data transfer to FPGA memory space with low cost since it has to be done once, and only implies changing the indexes of the elements in the original data structure. Forces are also stored in the same layout. This way, for the force calculation phase, each task calculates the forces between two tiles of particles and updates one of them. On the other hand, the update phase task updates a single tile of particles. The update particles part is significantly shorter than the force calculation due to the linear against quadratic computation time. Therefore, our efforts are focused only on the critical task. In our C implementation, the programmer can decide how many force calculations can be performed in parallel by unrolling the innermost computation loop with a pragma. Moreover, it can be pipelined so several forces are calculated every cycle, depending on the unroll factor.

For the Zynq board we found the best configuration is 6 accelerators to calculate forces with 8 parallel forces each, and one accelerator to update particles. For the Alveo, it is 8 accelerators to calculate forces with 16 parallel forces, and one accelerator to update particles. Both applications run at 300MHz. Without using any of the mentioned improvements in section 3, the base performance is around 11 Gpps (GPairs/s i.e. number of force vectors calculated per second) for the Zynq and 28 for the Alveo. Figure 7 shows the performance in Gpps when applying different optimizations for the ZCU102 and Alveo. The first optimization uses the wide memory port, which allows the accelerators to read

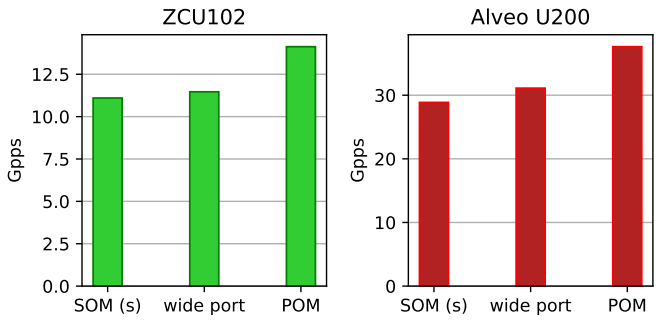


Fig. 7. N-body performance for 32768 particles and 16 steps with tiles of 2048 elements

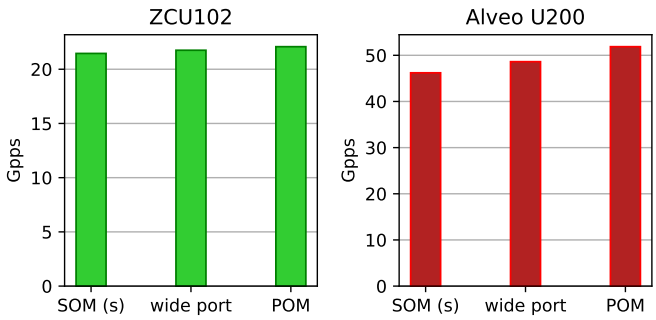


Fig. 8. Spectra performance for 2M particles with tiles of 2168 elements

faster each particle and force tile. Due to the relatively big tile size (2048), it does not have a big impact on performance. The Alveo gets a slightly better boost, since its memory port is 512-bit and the Zynq's port is only 128-bit wide. The use of the hardware runtime achieves the best performance, with a peak of 14 Gpps in the Zynq and 37 Gpps in the Alveo. Most of the speedup of the last optimization is caused by the fact that host-FPGA communication is minimized, in contrast to the Cholesky case where it was not as critical. By using POM, the time it takes for a task to go through the SoC or the PCIe is removed.

These results outperform other attempts to accelerate the N-body benchmark on FPGAs. In [11], Sozzo et. al. manage to get 13.4 Gpps with 36000 particles and a custom design coded in C++ and generated with SDAccel 2017.1 on a Xilinx Virtex UltraScale+ VU9P, similar in size to our Alveo FPGA. However, they only propose a design for the force calculation, and do not take into account more than one simulation step, which requires to update the positions and velocities. In [12] Sano et. al. introduce a design with the particle update phase. They use their own compiler for stream computation, SPGen, and Verilog modules used as a library for the compiler. They get 10.9 Gpps with 262144 particles in an Intel Arria10 FPGA, which is also similar in size to our Alveo. To the best of our knowledge the presented implementation is the fastest one done in FPGA for the N-body problem.

4.2.4 Spectra

The Spectra application [13] is similar to the N-body in the sense that the main kernel computes all particle-particle distances. Instead of using the distance to calculate and accumulate gravitational forces, it is used as an index to accumulate a histogram with the electronic weight between the particles, i.e. the multiplication of both electric charges. In order to parallelize the application, the histogram is replicated to allow different tasks to operate at the same time over different regions of memory. Also, the particle set is distributed in memory with tiles of TS consecutive particles. Each property of a particle, i.e. the 3-dimensional position vector and electric charge, is stored in a separate array. The reduction of the histograms into a single one is done by another task. Like with the N-body, the distance calculation loop is unrolled and pipelined, thus the accelerators compute more than one distance-vector per cycle.

We managed to put 3 accelerators to calculate distances with an unroll factor of 22 (number of distances per cycle) at 350MHz on the ZCU102 board, and 10 accelerators to calculate distances with 18 unroll factor at 300MHz on the Alveo. In both boards there is only one accelerator to reduce the histograms since, like in the N-body, its execution time is orders of magnitude shorter. The results, in figure 8, show the Gpps of some proposed optimizations for the ZCU102 and Alveo boards. The optimizations do not have as much impact as with other benchmarks. The wide port also has the same effect as the N-body: since the tile size is relatively big (2168), most of the time spent on a task is computation only. Furthermore, on the Alveo we increase only to a 128-bit port, as because of the high amount of accelerators Vivado does not route properly the design when using 512 bits. Using hardware task creation with dependencies slightly increases performance. Due to the tile size and the high amount of parallelism of the main kernel, the CPU exploits the POM ready queue like in the Cholesky benchmark. But even in this case, POM is able to speedup the execution by 6.25% on the Alveo, reaching 51 Gpps, and 1.7% on the ZCU102, reaching 22 Gpps.

4.3 Discussion

Table 3 puts together the best performance of each benchmark for each board. In addition, in order to have a reference, the table reports the performance of the pure CPU and GPU versions and power measures for FPGA and GPU. The FPGA power is an estimation reported by Vivado after bitstream generation, taking into account all the logic implemented in the design. The GPU power is measured dynamically with the Nvidia Management Library (NVML). All CPU-only benchmarks run with OmpSs software tasks, and the GPU ones use Nvidia CUDA. Both versions have been executed with the same problem size as their FPGA counterparts. The block/tile size is also maintained in the CPU benchmarks, but due to restrictions on the number of CUDA threads per block, it is reduced in N-body and Spectra to 256 and 512 respectively. SMP Matrix multiply and Cholesky use the OpenBLAS implementation at the block level, N-body and Spectra use a slightly modified code at the tile level, optimized for CPUs rather than FPGAs. The GPU versions use the pure cuBLAS implementation for matrix multiply and Cholesky, and a custom kernel optimized for GPUs in the other two benchmarks. In table 3,

TABLE 3
Peak performance of FPGA, SMP and GPU, Vivado power estimation and NVML GPU power measurements for all evaluated benchmarks

Benchmark (perf. metric)	Perf. Alveo	Perf. SMP Alveo	Perf. ZCU102	Perf. SMP ZCU102	Perf. GTX 1060	Power Alveo (W)	Power ZCU102 (W)	Power GTX 1060 (W)
matmul (GFlops)	353.87	236.18	199.98	13.69	3306.7	46.540	6.339	75.262
Cholesky (GFlops)	279.78	223.5	81.32	9.46	2375.6	37.636	21.425	64.217
N-body (Gpps)	37.62	0.83	14.13	0.057	35.34	64.606	24.504	67.924
Spectra (Gpps)	51.87	2.02	22.08	0.06	50.83	40.755	14.706	65.523

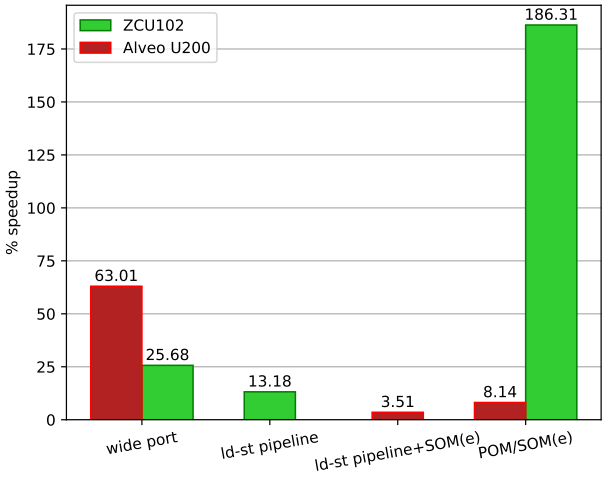


Fig. 9. Average speedup in % of each optimization across all benchmarks

SMP ZCU102 refers to the four ARM cortex cores, whereas SMP Alveo refers to the server attached to the FPGA used to do the evaluation (introduced in section 4.1). GTX 1060 refers to an Nvidia GPU GTX 1060 Mobile. All FPGA versions outperform their SMP counterparts, ranging from 25% speedup in the case of Alveo Cholesky, to 36700% in the ZCU102 Spectra case. Also it is observable that Spectra and N-body have the highest speedups. This is mostly due to the high amount of memory needed on matmul and Cholesky, which limits the block size and II since it makes the design quickly run out of BRAMs/URAMs. A similar effect can be observed when comparing with the GPU. Although in this case the GTX 1060 outperforms both FPGAs by a $\approx 10x$ factor in the matrix benchmarks. This is expected because GPU architectures are designed to execute efficiently embarrassingly parallel programs. On the other hand, N-body and Spectra present more challenges, as they have a very similar performance with the Alveo. Both need to calculate square roots for each pair of particles, which are more expensive compared to divisions/multiplications. In addition, Spectra introduces an atomic addition at the CUDA thread block level, since each thread accesses a shared array with an index that is calculated at runtime and thus can collide with other threads. Looking at the performance per Watt, both FPGAs are more efficient than the GPU in these two benchmarks, with a 13% improvement in N-body (Alveo) and 94% in Spectra (ZCU102).

Table 4 summarizes the resource usage of all the designs with the best performance. The ZCU102 has a high amount of DSP usage, which directly affects the GFlops and Gpps of the design because DSPs perform the floating point operations. On the contrary, the Alveo does not use as much in the matmul and Cholesky (around 47%), as increasing the II, which increases DSP usage, requires proportionally more BRAMs and URAMs. Another solution would be to put more accelerators, but it eventually leads to the same problem. Moreover, as the design gets bigger, the accelerator distance to the hardware runtime and main memory increases, affecting the critical path. A possible solution to solve this problem is to add registers to pipeline the interconnection paths to/from accelerators.

To conclude the study, figure 9 reports the speedup in percentage that each evaluated optimization gets over its previous version. For instance, wide port reports the speedup over the simple SOM version, *ld-st pipeline* (only matrix multiply) over the wide port version and so on. A special column is the *ld-st pipeline + SOM(e)*, which includes the speedup of the combination of the two versions over the wide port version for the matrix multiply benchmark on the Alveo. In this board the pipelining optimization is not able to extract much performance on its own, due to the concurrent memory access problem introduced in section 4.2.1. This effect is mitigated by the use of internal task creation, which optimizes copies, getting a performance boost of 3.5%, whereas the pipelining alone achieves 13.18% on the ZCU102, demonstrating that the optimization has more potential, especially in boards that feature higher memory bandwidth.

The wide port reports good results for both boards, getting 25.6% and 63% on the ZCU102 and Alveo respectively. The latter benefits more from this optimization due to the wider memory port (512 vs 128 bits). However, the use of a wide memory port makes the routing of the design more difficult. One open question is if narrower ports could lead to the capability to fit more accelerators and get more performance in applications where the memory is not the bottleneck. This possibility seems adequate to work with the pipeline optimization where there are more frequent but smaller accesses hidden by the computation. In fact it was the option chosen for the Spectra benchmark on the Alveo that was able to accommodate only a 128-wide port but with 10 parallel accelerators.

Using internal task creation (column POM/SOM(e) in figure 9), achieves the best results for the ZCU102. This is mainly because of the Cholesky benchmark, in which the hardware runtime boosts the execution by around 700%. However, other benchmarks still get an average speedup of 12.5% out of the SOM/POM optimization in the ZCU102. The Alveo does not see as much improvement because we have not tested any benchmark that involves fine-grained tasks. Still, the N-Body benchmark, which involves a lot

TABLE 4
Absolute and % FPGA resource usage of the evaluated benchmark's designs

Benchmark	LUT Alveo	FF Alveo	BRAM18K Alveo	URAM Alveo	DSP Alveo	LUT ZCU102	FF ZCU102	BRAM18K ZCU102	DSP ZCU102
Matmul	502499(38)	890439(38)	863(20)	647(67)	3240(47)	199034(72)	352418(62)	841(46)	1931(77)
Cholesky	435052(37)	748110(32)	2324(54)	328(34)	3221(47)	165555(60)	305840(56)	1576(86)	2242(89)
N-body	650237(55)	938399(40)	2855(66)	0(0)	5146(75)	206787(75)	339429(62)	814(45)	1927(76)
Spectra	605378(51)	766797(32)	2025(47)	547(57)	4892(72)	168569(62)	276693(50)	1575(86)	1776(70)

of synchronization between the FPGA and the host, is able to get nearly a 21% performance improvement out of this optimization alone. This effect is expected to grow as FPGAs grow bigger and can accommodate more accelerators or more FPGAs are attached to a single host CPU. If the communication time increases (like FPGA cloud environments) the performance boost of this optimization is also expected to become critical to obtain good performance.

5 Related work

Other efforts try to improve the efficiency and programmability of FPGAs from High-Level Languages (HLL). The Vineyard project [14] aims at facilitating heterogeneous programming from OpenSPL [15], OpenCL [16] and SDSoC [17]. The Ecoscale project [18] proposes a hybrid MPI+OpenCL programming environment and a minimum runtime system to orchestrate a large group of workers using reconfigurable accelerators. In both cases the approach is similar to the OmpSs@FPGA [1] baseline used in this work aiming at an easy usage of FPGA-based execution units. Over these works our approach improves the system performance with high-level data access optimizations, task-based parallel execution and the inclusion of a complete hardware runtime in the FPGA fabric that also improves programmability. The Unilogic system [19] also proposes a small runtime to coordinate several FPGA accelerators at the same time, but it is based on low-level code, not implemented from HLL. Mbongue et. al. [20] introduce automatic kernel extraction directly from LLVM IR code and uses RapidWright [21], a tool that improves the placement and routing of the FPGA design by pre-compiling and replicating the kernels. As future work, the same idea could be applied to the OmpSs@FPGA accelerators potentially increasing the operating frequency of the whole design.

Several frameworks target High-Level Synthesis from C/C++. Vivado HLS [22] is the Xilinx tool that is used by OmpSs@FPGA to generate FPGA IP blocks. Xilinx Vitis [23] works on top of Vivado HLS to better integrate the execution environment with Xilinx boards. It is an evolution of Xilinx SDSoC [17] and SDAccel [24] environments that includes a minimum runtime to manage communication between the FPGAs and the SMP host and facilitates the use of several already programmed FPGA library functions. In the same direction, Intel oneAPI [25] and Quartus [26] allow the use of HLL for Intel FPGAs. LegUp [27], [28] is another HLS tool that synthetizes C code with Pthreads and limited OpenMP annotations. Each thread (code) is synthesized as an accelerator at compile time. The remaining (sequential) portions are executed in the processor, invoke accelerators and use synchronization functions to retrieve their return values. Now, it only targets Microchip FPGAs [29]. ROCC [30], [31] was another interesting HLL compiler tool that was agnostic of the FPGA target. Our system is designed to be able to work over any HLS tool focusing on improving data movements and parallel execution of several accelerators, not the accelerators themselves. It also features a unique hardware runtime that allows more complex algorithms to be executed in the FPGA while improving performance.

There have been some works related to hardware task management and using tasks to program FPGAs also. Tan et al. [8] present a HW manager that supports task dependencies resolution and heterogeneous task scheduling for parallel task-based programming models. The proposal, however only allows task offloading to the accelerators and was not integrated woith any compiler framework. Cabrera et al. [32] and Sommer et al. [33] propose extensions in OpenMP to support the definition of tasks that target an FPGA device. Bosch et al. [7] also proposed to create tasks from inside the FPGA. However, these works were not integrated with a hardware dependence management system to allow fast task management inside the FPGAs. To the best of our knowledge, this work is the only one that allows task and dependence management inside the FPGA from a pragma annotated high-level source code that is automatically compiled into the final executable.

6 Conclusions

This paper presented the latest improvements and updates of the OmpSs@FPGA framework for easy FPGA programming. Local memory caching allows hardware accelerators to load/store data much faster. With a wide memory data bus, matching the memory controller width, the copies to/from main memory are improved. In some cases, the load/store phase can be overlapped with the main kernel in a pipeline manner. The OmpSs@FPGA compiler, Mercurium, is able to automatically apply these optimizations, except for the pipelining which is still a proof-of-concept. Its complete implementation is the main future work derived from this paper. In addition to the above, the compiler integrates an advanced hardware runtime to handle tasks created both from hardware accelerators and CPU. This runtime is able to optimize copies of the same memory region scheduled to the same accelerator. It is also able to manage dependencies inside the FPGA which minimizes host-FPGA communication and accelerates task creation.

The mentioned optimizations and features are used in several benchmarks, with different dependence patterns, on two boards: a Xilinx ZCU102 (SoC) and a Xilinx Alveo U200 (PCIe attached). Results show that the wide port optimization affects more the benchmarks with smaller granularity due to the bigger ratio between copies and compute time. Moreover, hardware task creation in FPGAs is most useful in

cases with many host-FPGA communications like N-body, or when granularity is small for the software runtime, like the Cholesky case on the ZCU102. For the latter benchmark and the matrix multiplication in the Alveo, we believe we can still get better results by solving the concurrent memory access problem that makes accelerators slower when accessing memory at the same time.

Only by changing C/C++ pragmas and compilation variables, the user is able to explore and extract a competitive performance with relatively low effort, compared to CPUs, other FPGA implementations and even GPUs in some cases. OmpSs@FPGA automatically performs the necessary steps to generate a final bitstream from the original code, so the programmer does not have to struggle with the intermediate hardware designs. The OmpSs@FPGA framework tools, like compiler sources and benchmarks (except Spectra) code are available in GitHub repositories [34].

Acknowledgments

This work has received funding from EuroEXA project (European Union's Horizon 2020 Research and Innovation Programme, under grant agreement No 754337), from Spanish Government (projects PID2019-107255GB and SEV-2015-0493, grant BES-2016-078046), and Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328).

References

[1] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, "Application acceleration on fpgas with ompss@fpga," in *International Conference on Field-Programmable Technology, FPT 2018, Naha, Okinawa, Japan, December 10-14, 2018*. IEEE, 2018, pp. 70–77.

[2] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Process. Lett.*, vol. 21, pp. 173–193, 2011.

[3] (2021, March) Nanos++ c/c++ api. [Online]. Available: https://pm.bsc.es/ftp/nanox/doxygen/group__capi.html

[4] (2021, April) Ompss@fpga user guide. [Online]. Available: https://pm.bsc.es/ftp/ompss/doc/user-guide

[5] ARM. (2020, December) Axi4 specification. [Online]. Available: https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification

[6] (2020, December) Openmp linear clause specification. [Online]. Available: https://www.openmp.org/spec-html/5.0/openmpsu106.html#x139-5760002.19.4.6

[7] J. Bosch, M. Vidal, A. Filgueras, C. Álvarez, D. Jiménez-González, X. Martorell, and E. Ayguadé, "Breaking master-slave model between host and fpgas," in *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, R. Gupta and X. Shen, Eds. ACM, 2020, pp. 419–420.

[8] X. Tan, J. Bosch, C. Álvarez, D. Jiménez-González, E. Ayguadé, and M. Valero, "A hardware runtime for task-based programming models," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 1932–1946, 2019.

[9] (2020, December) Openblas, an optimized blas library. [Online]. Available: https://www.openblas.net

[10] D. Yang, G. D. Peterson, and H. Li, "Compressed sensing and cholesky decomposition on fpgas and gpus," *Parallel Comput.*, vol. 38, no. 8, pp. 421–437, 2012.

[11] E. D. Sozzo, M. Rabozzi, L. D. Tucci, D. Sciuto, and M. D. Santambrogio, "A scalable FPGA design for cloud n-body simulation," in *29th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2018, Milano, Italy, July 10-12, 2018*. IEEE Computer Society, 2018, pp. 1–8.

[12] K. Sano, S. Abiko, and T. Ueno, "Fpga-based stream computing for high-performance n-body simulation using floating-point DSP blocks," in *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, HEART 2017, Bochum, Germany, June 7-9, 2017*, D. Göhringer and M. Hübner, Eds. ACM, 2017, pp. 16:1–16:6.

[13] C. Gonzalez, S. Balocco, and R. Pons, "Accelerating pp-distance algorithms on fpga, with opencl parallelization directives and data transfer optimizations," in *IWOCL '20: International Workshop on OpenCL, Munich, Germany, April 27-29, 2020*, S. McIntosh-Smith, Ed. ACM, 2020, pp. 28:1–28:2.

[14] Vineyard. (2018) Objectives and rationales of the project. [Online]. Available: http://vineyard-h2020.eu/en/project/objectives-and-rationale-of-the-project.html

[15] Maxeler, Inc. (2014) The open spatial programming language. [Online]. Available: https://openspl.org

[16] Khronos Group, Inc. (2018) Opencl. Https://www.khronos.org/opencl.

[17] Xilinx, Inc. (2020, December) Sdsoc development environment. [Online]. Available: https://www.xilinx.com/sdsoc

[18] Ecoscale Consortium. (2018) Project description. [Online]. Available: http://ecoscale.eu/project-description.html

[19] A. D. Ioannou, K. Georgopoulos, P. Malakonakis, D. N. Pnevmatikatos, V. D. Papaefstathiou, I. Papaefstathiou, and I. Mavroidis, "Unilogic: A novel architecture for highly parallel reconfigurable systems," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 4, Sep. 2020. [Online]. Available: https://doi.org/10.1145/3409115

[20] J. M. Mbongue, D. T. Kwadjo, and C. Bobda, "Automatic generation of application-specific FPGA overlays with rapidwright," in *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*. IEEE, 2019, pp. 303–306.

[21] C. Lavin and A. Kaviani, "Rapidwright: Enabling custom crafted implementations for fpgas," in *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. IEEE Computer Society, 2018, pp. 133–140.

[22] Xilinx, Inc. (2020, December) Vivado High-Level Synthesis. [Online]. Available: http://www.xilinx.com/hls

[23] ——. (2020, December) Xilinx Vitis Unified Software Platform. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis.html

[24] ——. (2020, December) Sdaccel development environment. [Online]. Available: https://www.xilinx.com/sdaccel

[25] Intel Corp. (2020, December) Intel oneAPI Toolkits. [Online]. Available: https://www.intel.com/oneapi

[26] ——. (2020, December) Quartus Prime. [Online]. Available: https://www.intel.com/quartus

[27] A. Canis and et al., "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 2, pp. 24:1–24:27, September 2013.

[28] B. Fort and et al., "Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis," in *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*, Aug, pp. 120–129.

[29] Microchip Technology Incorporated. (2020, December) Microchip Acquires High-Level Synthesis Tool Provider LegUp. [Online]. Available: https://www.microchip.com/en-us/about/news-releases/products/microchip-acquires-high-level-synthesis-tool-provider-legup

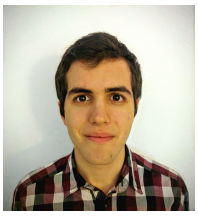
[30] J. R. Villarreal, A. Park, W. A. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0." in *FCCM*, R. Sass and R. Tessier, Eds. IEEE Computer Society, 2010, pp. 127–134.

[31] W. A. Najjar and J. R. Villarreal, "Fpga code accelerators - the compiler perspective," in *DAC*, 2013, p. 141.

[32] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jiménez-González, "Openmp extensions for fpga accelerators," in *International Symposium on Systems, Architectures, Modeling, and Simulation*, 08 2009, pp. 17 – 24.

[33] L. Sommer, J. Korinth, and A. Koch, "Openmp device offloading to fpga accelerators," in *IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 07 2017, pp. 201–205.

[34] (2020, December) Ompss@fpga github. [Online]. Available: https://github.com/bsc-pm-ompss-at-fpga



Juan Miguel de Haro received his BS degree in Informatics Engineering in 2018 and MS degree specialized in High Performance Computing in 2020 from Universitat Politècnica de Catalunya (UPC). He is currently a Ph.D. student at the Computer Architecture Department of UPC. He also works with the OmpSs@FPGA team in the programming models group of the Barcelona Supercomputing Center (BSC). His research interests cover parallel and reconfigurable architectures, hardware acceleration and runtime systems for heterogeneous and distributed architectures.

Jaume Bosch received the B.S. and M.S. degrees in Computer Science from the Technical University of Catalunya (UPC) in 2015 and 2017, respectively. Currently, he is a Ph.D. student in the Department of Computer Architecture of UPC and in the Programming Models Group of Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS). His research interest lies in parallel, distributed and heterogeneous runtime systems for High Performance Computing, especially systems with hardware accelerators like FPGAs or many-core architectures.

Antonio Filgueras earned a degree in Computer Science from Universitat Politècnica de Catalunya (UPC) in 2012. He currently works in the OmpSs@FPGA team within the Programming Models group at BSC, focused primarily on programming models for reconfigurable systems in High Performance Computing. He has been a researcher in the AXIOM, Legato, EuroEXA and MEEP European projects among others.

Miquel Vidal received a BSc degree in Computer Engineering in 2015 and an MSc in High-Performance Computing in 2017 from the Universitat Politècnica de Catalunya (UPC). Since 2015 he has been part of the Programming Models group at Barcelona Supercomputing Center (BSC-CNS), first as a resident student and currently as a research engineer. His research interests are focused on parallel architectures, multiprocessor systems, and heterogeneous and reconfigurable solutions for high-performance computing; as well as their application in the bioinformatics field.

Daniel Jiménez-González received the M.S. and Ph.D. degrees in Computer Science from UPC in 1997 and 2004, respectively. He currently holds a position as a Tenured Assistant Professor in the Department of Computer Architecture in UPC and is an associated researcher at BSC. His research interests cover the areas of parallel architectures, runtime systems and reconfigurable solutions for high-performance computing systems. He has been participating in the HiPEAC Network of Excellence and the SARC, ACOTES, TERAFLUX, AXIOM, PRACE and EUROEXA European projects.

Carlos Álvarez received his MS and Ph.D. degrees in Computer Science from Universitat Politècnica de Catalunya (UPC), Spain, in 1998 and 2007, respectively. He is currently a Tenured Assistant Professor in the Department of Computer Architecture at UPC and an associated researcher at the BSC. His research interests include parallel architectures, runtime systems, and reconfigurable solutions for high-performance multiprocessor systems. He has participated in the HiPEAC Network of Excellence and the different European projects.

Xavier Martorell received the M.S. and Ph.D. degrees in Computer Science from the Universitat Politecnica de Catalunya (UPC) in 1991 and 1999, respectively. He has been an associate professor in the Computer Architecture Department at UPC since 2001. His research interests cover the areas of operating systems, runtime systems, compilers and applications for high-performance multiprocessor systems. Dr. Martorell has participated in several long-term research projects with other universities and industries, primarily in the framework of the European Union ESPRIT, IST, FET and H2020 programs. He has coauthored more than 80 publications in international journals and conferences. He is currently the Manager of the Parallel Programming Models team at the Barcelona Supercomputing Center.

Eduard Ayguadé received his degree in Telecommunications Engineering in 1986 and his Ph.D. in Computer Science in 1989, both from the Universitat Politècnica de Catalunya (UPC), Spain. He has been full professor of the Computer Architecture Department at UPC since 1997. Prof. Ayguadé has co-advised more than 25 Ph.D. thesis, in topics related to his research interests: multicore architectures, parallel programming models and their architectural support, and compilers for HPC architectures. Prof. Ayguadé has published more than 400 papers in these research topics and has participated in several research projects in the framework of the European Union (in recent years, ExaNoDe, EuroExa, EPI, MEEP, . . .) and research collaborations with companies related to HPC technologies (IBM, Intel, Huawei, Micron, Microsoft and Samsung). Currently Prof. Ayguadé is associate director for research of the Computer Sciences Department at the Barcelona Supercomputing Center (BSC-CNS), the National Center for Supercomputing in Spain.

Prof. Jesús Labarta received his Ph.D. in Telecommunications Engineering from UPC in 1983, where he has been a full professor of Computer Architecture since 1990. He was Director of European Center of Parallelism at Barcelona from 1996 to the creation of BSC in 2005, where he is the Director of the Computer Sciences Dept. His research team has developed performance analysis and prediction tools and pioneering research on how to increase the intelligence embedded in these performance tools. He has also led the development of OmpSs and influenced the task-based extension in the OpenMP standard. He is now responsible for the POP center of excellence and leads the RISC-V vector accelerator within the EPI project. He was awarded the 2017 Ken Kennedy Award for his seminal contributions to programming models and performance analysis tools for High Performance Computing, being the First Non-US Researcher receiving it.