Application Acceleration on FPGAs with OmpSs@FPGA

Jaume Bosch*†, Xubin Tan*†, Antonio Filgueras*, Miquel Vidal*, Marc Mateu*†,
Daniel Jiménez-González*†, Carlos Álvarez*†, Xavier Martorell*†, Eduard Ayguadé*†, and Jesus Labarta*†

*Computer Science Dept.
Barcelona Supercomputing Center
Barcelona, Spain
Email: name.surname@bsc.es

†Computer Architecture Dept.
Universitat Politècnica de Catalunya
Barcelona, Spain
Email: djimenez,calvarez,xavim@ac.upc.edu

***Abstract*—OmpSs@FPGA is the flavor of OmpSs that allows offloading application functionality to FPGAs. Similarly to OpenMP, it is based on compiler directives. While the OpenMP specification also includes support for heterogeneous execution, we use OmpSs and OmpSs@FPGA as prototype implementation to develop new ideas for OpenMP. OmpSs@FPGA implements the tasking model with runtime support to automatically exploit all SMP and FPGA resources available in the execution platform.**

In this paper, we present the OmpSs@FPGA ecosystem, based on the Mercurium compiler and the Nanos++ runtime system. We show how the applications are transformed to run on the SMP cores and the FPGA. The application kernels defined as tasks to be accelerated, using the OmpSs directives are: 1) transformed by the compiler into kernels connected with the proper synchronization and communication ports, 2) extracted to intermediate files, 3) compiled through the FPGA vendor HLS tool, and 4) used to configure the FPGA. Our Nanos++ runtime system schedules the application tasks on the platform, being able to use the SMP cores and the FPGA accelerators at the same time.

We present the evaluation of the OmpSs@FPGA environment with the Matrix Multiplication, Cholesky and N-Body benchmarks, showing the internal details of the execution, and the performance obtained on a Zynq Ultrascale+ MPSoC (up to 128x). The source code uses OmpSs@FPGA annotations and different Vivado HLS optimization directives are applied for acceleration.

***Keywords*-Heterogeneous Parallelism; OmpSs; FPGAs;**

I. INTRODUCTION

Current trends in computer architecture are focused on providing heterogeneous execution environments. Heterogeneity comes in many different flavors. One important flavor is an environment that incorporates accelerators within an FPGA (Field-Programmable Gate Array), providing specialized hardware to better execute specific algorithms. FPGA devices are programmed by means of bitstreams, usually generated by vendor-proprietary tools, following an specification provided in the VHDL or Verilog hardware description languages. In addition, there is an additional characteristic to be taken into account: Vendor compilation tools to generate the place and route to configure the FPGA usually take from minutes to hours. This causes that the porting of new code onto these platforms is usually a slow process. Vendors also provide FPGAs integrated with a few cores, that can be used as the host cores. In this case, the FPGA shares the physical memory with the cores.

FPGA modules (accelerators from now on) may have additional but limited amount of local memory. This accelerator local memory may be needed in order to achieve high performance accelerators and, in this case, data movements are a must for them to work. On the other hand, the limited amount of memory forces the use of blocking techniques when the workload does not fit on the FPGA resources. Therefore, memory transfers from/to host memory to/from accelerator local memory should be optimized enough to reduce the communication overhead or be overlapped, with or without blocking execution, with the accelerator computation in order to hide it. Related to those memory transfers, the FPGA device incorporates the implementation of the bus protocol as part of its programming to perform FPGA external accesses. Thus, the programmer needs to be aware of it and should incorporate it in the bitstream generation process. In order to reduce the programmer effort, models have to provide the means to perform/indicate data transfers between host and accelerators in an easy way, allowing to reduce the impact of those communications.

In our work, OmpSs@FPGA ecosystem addresses previous challenges achieving high productivity by providing higher-level abstractions that could help the programmer to generate high performance code on them. For example:

- Making the memory allocation and data copies automatic, based on directives.
- Providing the programmer facilities to perform blocking from inside the accelerators.
- Automating the code generation of the CPU and FPGA binaries, provided the C/C++ implementation, by transparently running open or vendor tools.
- Allowing the use of parallelism based on tasking (instead of kernel invocations).
- Providing support for data dependent tasks, and managing the execution based on such data dependences.
- Providing FPGA execution trace generation support.

This makes the programming environment to (hopefully) completely hide the target architectures, providing a clean, high-level, abstract interface to the programmers, and incorporating all the intelligence on management and scheduling onto the runtime system.

II. Related Work

There are efforts similar to OmpSs@FPGA. The Vineyard project [1] aims at facilitating heterogeneous programming, based on OpenSPL [2], OpenCL [3] and SDSoC [4]. The Ecoscale project [5] targets applications written in MPI and OpenCL, to synthesize the OpenCL kernels for the FPGAs, and support distributed and heterogeneous computing. For both projects, the goal is to efficiently execute functionality in the FPGAs. In addition to this, we also provide heterogeneous execution on both the FPGAs and the available host cores.

There are a large number of frameworks targeting the High-Level Synthesis from C/C++. Vivado HLS [6] is the Xilinx tool that we use from OmpSs@FPGA to generate the FPGA IP blocks. Xilinx SDSoC [4] runs on top of Vivado HLS, and better integrates the execution environment for the Xilinx Zynq platform (7000 and Ultrascale+), with the automatic generation of the complete Linux system for the target platform. LegUp [7], [8] 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. In OmpSs, the accelerators are also generated at compile time but they correspond to tasks with target device FPGA. Indeed, it is also possible to specify tasks that run in the SMP, that can run in both SMP or FPGA, or that can substitute other tasks. OmpSs runtime takes care of issuing data transfers and task executions among the cores and the IP accelerators based on the readiness of the task dependencies, if defined. To the best of our knowledge, OmpSs@FPGA is the first attempt to this kind of dynamic work distribution across SMP cores and the FPGA that integrate previous programmability features.

III. OmpSs@FPGA Ecosystem

The OmpSs [9], [10] programming model allows to express parallelism that will be executed in the available resources among the host SMP cores, or integrated/discrete GPUs and/or FPGAs. OmpSs is based on task parallelism, and very similar to OpenMP tasking. It is being used as a forerunner prototyping environment for future OpenMP features. On GPUs, both CUDA and OpenCL kernels are supported. For FPGAs, OmpSs uses the vendor IP generation tools (Xilinx Vivado and Vivado HLS [6], [11], or Altera Quartus [12]), to generate the hardware configuration from high-level code. OmpSs@FPGA can also leverage existing IP cores, provided they adhere to the same interface with our software platform.

OmpSs@FPGA [13] is a significant upgrade of the OmpSs infrastructure (Mercurium source-to-source compiler and Nanos++ runtime) to incorporate FPGA support. Figure 1 shows an example of an OmpSs application. In particular, function *matrix_multiply* is defined as a task with input dependencies *a* and *b* and input/output dependency *c*. Each call to this function will generate a task that will be run when its dependencies are ready. This

```
#pragma omp target device(fpga,smp) copy_deps num_instances(3)
#pragma omp task in([BS]a,[BS]b) inout([BS]c)
void matrix_multiply(float a[BS][BS],
     float b[BS][BS],float c[BS][BS]) {
#pragma HLS inline
#pragma HLS array_partition variable=a block factor=BS/2 dim=2
#pragma HLS array_partition variable=b block factor=BS/2 dim=1
  for (int ia = 0; ia < BS; ++ia)
    for (int ib = 0; ib < BS; ++ib) {
#pragma HLS PIPELINE II=1
      float sum = 0;
      for (int id = 0; id < BS; ++id)
        sum += a[ia][id] * b[id][ib];
      c[ia][ib] += sum;
} }
...
for (i=0; i<NBI; i++)
  for (j=0; j<NBJ; j++)
    for (k=0; k<NBK; k++)
       matrix_multiply(AA[i][k], BB[k][j], CC[i][j]);
#pragma omp taskwait
...
}
```

Figure 1. First version of FPGA Matrix Multiply code

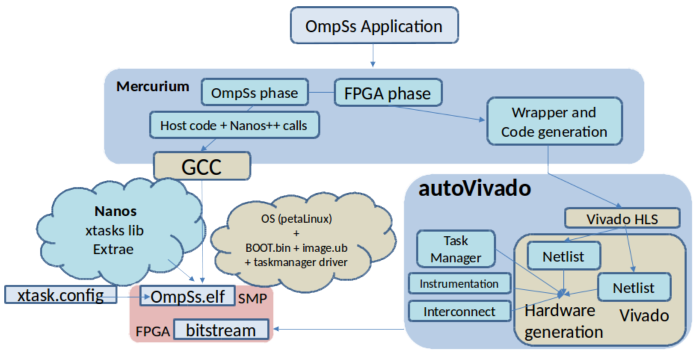


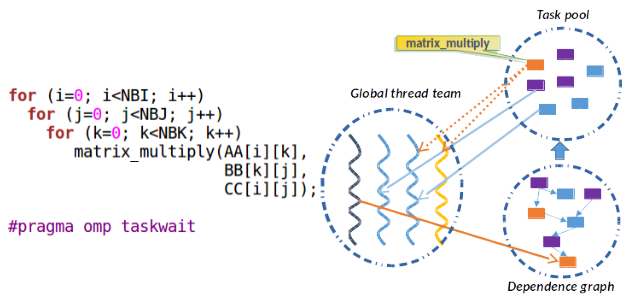
Figure 2. OmpSs compilation env. with FPGA support

task has also been defined to be potentially executed in two target devices: any of the cores of the *smp* running the application and three instances of an accelerator that will be built to do this task in the FPGA. The accelerator has been tuned by the programmer to exploit the parallelism of the FPGA by using some additional directives (*#pragma HLS*) not related to OmpSs programming model. In the following sections, we will describe how the OmpSs compilation and runtime ecosystem helps programmability, heterogeneity, memory transfers and tracing support, and finally, mechanisms to develop blocking techniques from inside the FPGA.

A. Programming Productivity

Figure 2 shows the toolchain flow. In particular, it currently supports Xilinx FPGAs using the Vivado HLS and Vivado tools through our *autoVivado* tool.

At compilation level, the OmpSs application is split in two parts according to OmpSs directives. The part to be run in the SMP is transformed with all the runtime calls introduced by the source-to-source compiler. This part includes all the tasks code to be run in the SMP and is compiled with `gcc`. All functions annotated with the *target device(fpga)* directive are defined as tasks that will be transferred to the Vivado HLS tool for compilation to IP cores. Additionally, the Mercurium compiler generates a stub/wrapper function for each task, used to invoke the corresponding IP core from our Nanos++ runtime



```
for (i=0; i<NBI; i++)
  for (j=0; j<NBJ; j++)
    for (k=0; k<NBK; k++)
       matrix_multiply(AA[i][k],
                       BB[k][j],
                       CC[i][j]);

#pragma omp taskwait
```

Figure 3. High-level representation of the Nanos++ environment

system, adapting the parameter passing. *autoVivado* tool invokes Vivado HLS to transform wrapper functions and FPGA-annotated functions into IP cores. Then, *autoVivado* connects them to the Task Manager, local memory (BRAM) and AXI ports, using Vivado and generates the bitstream with the accelerators. Also, a configuration file (*xtasks.config*) with accelerator metadata is generated. This is necessary for the Nanos++ runtime in order to know the type and number of accelerators in the FPGA. This compilation process is automatically done by the compiler avoiding user errors and speeding up all the process of hardware generation for the supported platforms (Zynq 7000 and Ultrascale+ families). Programmers can deactivate the accelerators generation to reduce the overall compilation time, if they have been previously generated.

On the other hand, Nanos++ is the OmpSs runtime system. It takes care of executing tasks annotated by the programmer in the available resources. The high-level view of the execution environment is presented in Figure 3.

Nanos++ environment has a *thread team* created by default, the *dependence graph* used to organize tasks that still have pending data dependences to be resolved, and the *task pool* representing all the ready tasks. Running threads create tasks and insert them into the dependence graph. When data dependences have been fulfilled, the thread detecting this situation moves the dependence-free tasks to the task pool. When a thread finishes the execution of a task, it becomes idle and it looks for work in the task pool. The Nanos++ runtime will also take care of the possible heterogeneity expressed by the programmer and the necessary memory transfers (copies).

On the FPGA side, an special IP, called Task Manager, is in charge of the management of accelerator tasks execution and finalization. It will provide the accelerator with the information of a task, written by the Nanos++ runtime in shared memory. Then, once the accelerator finalizes, the Task Manager will be signaled by the output stream of the accelerator to indicate the end of the task. Finally, the Task Manager will notify the Nanos++ runtime of the finalization of the task.

B. Heterogeneity Support

Figure 1 example code includes a ***target device*** directive that indicates two target devices for the defined task: *smp* and *fpga*. This means that any invocation to this task can be run either in the *smp* or the *fpga*, transparently to the programmer. The code to be run is the same, one

```
#pragma omp target device(smp) copy_inout([BS]c) \
        implements(matrix_multiply)
#pragma omp task in([BS]a, [BS]b) inout([BS]c)
void matmulBlockSmp(float a[BS][BS],
    float b[BS][BS], float c[BS][BS]) {
  const float alpha = 1.0;
  const float beta = 1.0;
  cblas_gemm(CblasRM, CblasNT, CblasNT, \
          BS, BS, BS, alpha, a, BS, b, BS, beta, c, BS);
}
```

Figure 4. Implements version of SMP Matmul code (no castings done)

accelerated in the FPGA, and the another executed in the SMP. In the case of the *fpga* device, the ***num_instances*** clause is used to express how many instances of the given IP core (*fpga* task) the programmer decides to generate; in this case three instances. Those tasks, of the same type, may potentially run in parallel at runtime in both *smp* and *fpga*. The runtime system will know through the configuration file (*xtasks.config*) the number of instances available of each accelerator type defined by the programmer.

On heterogeneous environments, Nanos++ has a specific subset of threads that represent each of the heterogeneous devices. We call these threads *helper threads*. Figure 3 shows, on the left side, the code invoking the heterogeneous task *matrix_multiply* and, on the right side, the overview of threads and task pool in the runtime. The orange thread (thread number 4, on the right hand side of the Global thread team) in the figure is one of those helper threads. In this particular example, it may represent one FPGA accelerator.

Tasks can be also annotated with the ***implements(funcname)*** clause, indicating that such task is a different implementation of the same algorithm that *funcname* implements. This allows the runtime system to select the *best* version to run at any given point in time. This is done by applying a scheduling policy that takes these alternative implementations into account.

Tasks annotated with the *implements* clause implement the same functionality as other tasks but with a different code. At compile time, two (or more) versions of the task are built targeting different computing units. At runtime, those tasks can be executed on an SMP core or on one or more devices. This means that when the runtime system finds one of these tasks in the ready queue, it can be grabbed by a regular worker thread, that will execute the SMP version of the task in a SMP core. Or the task can be grabbed by one of the *helper threads*, and then the device version of that task will be executed in the device represented by the thread, transparently to the programmer (as shown in Figure 3 for the Matrix Multiply). Figure 4 shows the code that implements the function listed in Figure 1 in SMP by using OpenBLAS gemm.

C. Memory Transfers Support

Tasks executing in devices, with their own local memory, may need copy data from/to the device. In particular, the device memory space is main memory in the SMP, accessible from the accelerators and the SMP cores, physically contiguous, pinned and non-cacheable. With ***copy_deps*** clause the programmer indicates that all the dependences will require, at runtime, device memory space for copies between host memory and accelerators local memory. Alternatively, ***copy_in/out/inout(list-of-variables-with-size)*** clauses indicate the list of parameters of the task that needs to be copied to/from the accelerator and deactivate the, by default, *copy_deps* clause. In both cases, the runtime takes care of allocating device memory and copies between user and device memory for the list of parameters labeled as copies. Task parameters that are not indicated to be copied by the programmer should have been previously allocated in device memory (using our Nanos++ runtime API) so that the accelerator can access them. If any of the task parameters is a scalar, it is directly passed to the accelerator.

As aforementioned, a wrapper is generated for each task so that accelerators and runtime can communicate with each other. The work to be done by the wrapper is to get information of the task (address of each parameter) and output final signal, and optionally (based on some compilation flags), get/write tracing information, reserve local memory for some of the parameters, copy the data from/to device memory to/from local memory, and do timing instrumentation. In detail, the wrapper reads the address of each parameter of the accelerator using an input stream port of the wrapper IP, which is connected to our Task Manager IP (see Figure 2). Then, it maps the parameter address to an IP port connected to the external memory, using the AXI protocol, and copies the parameter data from/to device memory to/from local memory, if required. In case that copies are not requested, the programmer kernel code can access the device memory without any change. This makes programming easy and useful to apply blocking techniques from inside the accelerator, as shown in Figure 8 for the N-body.

D. Tracing Support

OmpSs@FPGA ecosystem allows tracing the execution of Nanos++ threads (running in cores) and accelerators. For the threads, it provides information at application and runtime levels so that the programmer can analyze both the application and runtime internals such as the creation of tasks, task executions, taskwaits, etc. For the accelerators, current support provides the user with information of execution time of the data movements done by the wrapper and computational time of the kernel. Figure 10 shows an execution trace where this information is shown for three accelerators of the Matrix Multiply application.

This tracing feature implies hardware support for timing within the bitstream and is transparently done to the programmer, which only has to activate the corresponding compilation flag. The execution trace generated is done using an internal tracing library and can be visualized with Paraver [14].

IV. Experimental Setup

Communication logic and hardware accelerators are coded in C with Vivado HLS directives. The final system designs are synthesized with Vivado Design Suite 2016.3.

Application	Description	Parameters
Matrix Multiply	Blocked matrix multiply in square blocks	Matrix size, Block size
Cholesky	Blocked Cholesky decomposition of a matrix	Matrix size, Block size
N-Body	Blocked N-body simulation	Number of particles, particles in a block, time-steps

Table I
Summary of applications' characteristics

The hardware platform contains a Zynq Ultrascale+ MPSoC Chip XCZU9EG-FFVC900 [15]. It includes an Application Processing Unit (APU) with 4 ARM Cortex-A53 cores (operating at 1.1GHz) and an FPGA. It has a 4GB DDR4 as main memory. Nowadays, bitstream download is done before executing the application.

The Zynq Ultrascale+ runs an Ubuntu Linux 16.04, where we perform the timing of sequential and parallel code. We also use performance tools Extrae and Paraver [14] to analyze the application behavior.

V. Applications

Three applications have been analyzed with our current workflow: Matrix Multiply, Cholesky and N-Body. Table I summarizes the characteristics of each application. Vivado HLS pragmas are not shown for Cholesky neither N-Body for simplicity and space constraints. Those are basically `array_partition` and `pipeline II` HLS pragmas. These pragmas allow to exploit operational and memory access parallelism, essential to obtain good performance.

Following subsections explain how each application has been implemented in the heterogeneous system through successive High Level Optimizations using the OmpSs@FPGA workflow.

A. Matrix Multiplication

The first application ported to the OmpSs@FPGA framework was the Matrix Multiplication. It is a key application because it includes several of the properties that are found in common HPC problems: regular dependence pattern and a blocked implementation that involves moving several times the same data to and from the accelerators. Figure 1 presents the initial code of a matrix multiply algorithm for a block of $BS \times BS$ size that can be used to implement a blocked matrix multiply.

In order to port this code to FPGA and use a good part of its resources, some directives should be added to take advantage of the parallelism in the innermost loop. A better performance would be achieved when all the multiplications and additions of this loop are performed in parallel. Figure 1 shows the code with the High Level Synthesis (HLS) pragmas used to obtain the parallel version of the loop. The key pragma in the code is *PIPELINE II=1* that says that an iteration of the second loop should start each cycle ($II = 1$). To obtain this performance, the innermost loop should be completely unrolled. To accomplish this goal, all the elements in a row of matrix a and all the elements in a column of matrix b should be read each cycle. Pragmas *array_partition* make the compiler to distribute a by columns (second dimension $dim = 2$) or

```
#pragma omp target device (smp,fpga) copy_deps num_instances(1)
#pragma omp task inout(A[BS*BS])
void spotrf(float *A);
... // Other kernels with OmpSs task definition
void Cholesky(float **A) {
  int i,j,k;
  for (k=0; k<NB;k++){
    spotrf(A[k*NB+k]);
    for (i=k+1;i<NB;i++)
      strsm(A[k*NB+k], A[k*NB+i]);
    for (i=k+1;i<NB;i++) {
      for (j=k+1;j<i;j++)
        sgemm(A[k*NB+i], A[k*NB+j], A[j*NB+i]);
      ssyrk(A[k*NB+i], A[i*NB+i]);
} } }
```

Figure 5. Cholesky application with its four composing kernels

b by rows (first dimension $dim = 1$) in different block RAM (BRAM) memories by a factor that is half the side size of such matrices (as each BRAM has 2 read ports).

The first option to create a good accelerator for the matrix multiplication in the FPGA was to try to create an accelerator that fully fits in the FPGA. We found that an accelerator with $BS = 128$ fits really well (even three of them) in the used FPGA but with $BS = 256$ it was also possible to obtain a successful compilation.

B. *Cholesky decomposition*

Cholesky Factorization is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. It computes $A = LL'$, with A an $n \times n$ matrix and L lower-triangular. The decomposition is made by blocks of BS size which results in four different kernels. Figure 5 shows the code of the Cholesky decomposition as it is decomposed in its four kernels: spotrf, strsm, sgemm and ssyrk. The SMP version is computed using the OpenBLAS version of the kernels while the FPGA version implements the kernels in C and compiles them through the HLS tool.

C. *N-Body simulation*

The N-Body simulation computes the problem of predicting the individual motions of a group of objects interacting with each other. Figure 6 shows the main loop (*nbody*) of the application. Data is split in blocks, and all the forces are computed, for each particle against all the other particles. Then, the particles velocity and positions are updated with the forces previously computed. This is done for as many time steps as desired.

In order to decompose the problem in simpler tasks, a routine that computes the interaction of BS objects against a set of other BS objects is used. The computation of the forces for a block of particles is done considering all other blocks of particles (no tree algorithm is used). Each call to `calc_forces_BLOCK` function is defined as a task, which allows the OmpSs programming model to execute them in parallel when possible.

VI. SUPPORT TO PERFORMANCE IMPROVEMENT

With the help of OmpSs@FPGA, different techniques can be used to improve the performance obtained with the different applications analyzed. In this section, some of these techniques will be reviewed.

```
#pragma omp target device(fpga)\
   copy_in([PARTICLE_SIZE*BS]block1, [PARTICLE_SIZE*BS]block2)\
   copy_inout([FORCE_SIZE*BS]forces)
#pragma omp task
void calculate_forces_BLOCK (for_ptr_t forces, \
  part_ptr_t block1,part_ptr_t block2,char safe);

void calc_forces(force_t *forces,
     part_t *bl, int n_blocks) {
  for (int i = 0; i < n_blocks; i++) {
    for (int j = 0; j < n_blocks; j++) {
      for_ptr_t f0 = (for_ptr_t)(forces+i);
      part_ptr_t b1 = (part_ptr_t)(bl+i);
      part_ptr_t b2 = (part_ptr_t)(bl+j);
      char safe = (b1 == b2);
      calculate_forces_BLOCK(f0, b1, b2, safe);
} } }
...
void nbody(part_t *part,force_t *forces,
     int n_blocks, int timesteps) {
  for(int t = 0; t < timesteps; t++) {
    calc_forces(forces, part, n_blocks);
    update_part(n_blocks, part, forces);
  }
}
```

Figure 6. N-body main loop and blocking version of the calculate_forces

Name	B_18Kb	DSP48E	FFs	LUTs
2 BLOCK	88 (9.7%)	1k (40.2%)	236k (43.2%)	171k (62.5%)
ALL FPGA	44 (4.9%)	0.5k (20.1%)	123k (22.5%)	88k (32.3%)
2 SUBBLOCK	107 (11.7%)	1k (40.2%)	242k (44.3%)	177k (64.6%)

Table II
RESOURCES USED BY N-BODY KERNELS IN XCZU9EG

A. *Blocking Performance impact*

There may be applications with a significant amount of work to be done that may not require any SMP computation, and then, could be completely executed in the accelerators. However, the limitation of the local memory inside the accelerators (due to the available FPGA resources) and also, the communication overhead associated, usually reduce the possibility of performing all the computation in the FPGA or the performance that can be achieved. A common approach in task-based programming models is to define tasks that can operate with a limited block size and then, perform the overall computation by blocks. In the case of FPGA accelerators, applying blocking in the code running in the SMP and using the accelerators to perform the block processing may imply several synchronizations and communications, in addition to several copies of task parameters between user and device memory. However, applying blocking from the accelerator can help to decouple SMP and accelerator task executions and reduce the total number of synchronizations between Nanos++ runtime and accelerators.

N-Body simulation is the selected application to demonstrate how to use blocking to improve the performance when using OmpSs@FPGA. The first implemented version simply puts the kernel `calc_forces_BLOCK` into the FPGA trying to make the computation as fast as possible. It was possible to fit 2 instances of the function that computes the problem for 128 particles in the FPGA. Table II, row **2 BLOCK** shows the resources used by this and all the other implementations presented in this section. The working frequency for all of them is 200MHz.

With the OmpSs@FPGA ecosystem it is easy to compare the performance obtained by the version that executes

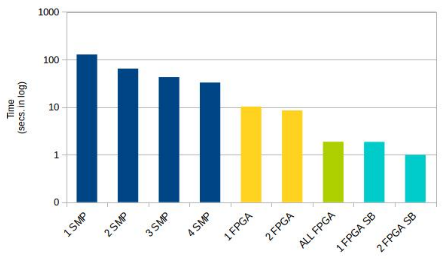


Figure 7. Time of N-Body execution with different blocking.

the tasks in the FPGA against the parallel version that uses the SMP cores to compute the tasks. Simply by changing the target device in the first line in Figure 6 from *smp* to *fpga* and back, the same code can be executed using the different resources in the system (using Nanos arguments to adjust the number of resources used in each execution). Figure 7 shows the time in logarithmic scale used by these OmpSs implementations when using a different number of resources to compute a 16384 particles problem with 8 time steps. As it can be seen in the figure, the runtime is able to obtain a near perfect parallelism when executing in the 4 cores available in the system. However, it can also be seen that the FPGA implementation is several times (15x) faster than the SMP implementation, making this problem a good fit for FPGA execution in the analyzed system. On the other hand, it can easily be observed that the 2 accelerators version (**2 FPGA**) has almost the same performance as the 1 accelerator version (**1 FPGA**). Nanos helps testing all the versions by simply changing the execution command line, allowing the programmer to see that there is a problem with this accelerator. In this case, the accelerators are so fast that the capacity of the threads to create tasks is the performance limiting factor.

To further improve the performance, a new accelerator was programmed that took care of executing the whole `calc_forces` function inside the FPGA (**ALL FPGA** column in Figure 7 and **ALL FPGA** row in Table II). Figure 8 shows this *calculate_forces* blocking version from inside the FPGA. In this case the *calculate_forces_BLOCK* is not defined as a task. The parameters *forces, block1, block2* are specified to be copied. However, we have specified, at compile time, that the wrapper does not reserve local memory for the parameters (neither perform copies) but, connects the parameter variables of the task to external memory ports of the IP. That means that each access to the data of the parameters is actually accessing the external device memory transparently to the user. Programmers can perform copies to local memory (local variables in the code) and process the local copy (in BRAM) to avoid continuously accessing external memory. In the code of the figure the programmer uses *memcpy* (actually this *memcpy* is interpreted and optimized by Vivado HLS) to perform copies to local variables. These local variables are usually mapped to BRAM of the FPGA. As it can be seen this accelerator is several times faster than the previous one, although it doesn't use even half the resources available in the FPGA fabric, it doesn't make sense to fit two of them in it because there is no parallelism

```
#pragma omp target device(fpga) \
        copy_in([PART_BSIZE*n_blocks]block1) \
        copy_in([PART_BSIZE*n_blocks]block2) \
        copy_inout([FORCE_BSIZE*n_blocks]forces)
#pragma omp task
static void calculate_forces(for_ptr_t forces,
        part_ptr_t block1, part_ptr_t block2, int n_blocks) {
   const int pbs = sizeof(float)*PART_BSIZE;
   const int fbs = sizeof(float)*FORCE_BSIZE;
   for (int i = 0; i < n_blocks; i++) {
      for_ptr_t  lforces[FORCE_BSIZE];
      part_ptr_t lblock1[PART_BSIZE];
      memcpy(lforces, forces + i*FORCE_BSIZE, fbs);
      memcpy(lblock1, block1 + i*PART_BSIZE , pbs);
      for (int j = 0; j < n_blocks; j++) {
         float lblock2[PART_BSIZE];
         memcpy(lblock2, block2 + j*PART_BSIZE, pbs);
         calculate_forces_BLOCK(lforces,
                      lblock1, lblock2, (i == j));
      }
      memcpy(forces + i*FORCE_BSIZE, lforces, fbs);
}}
```

Figure 8. FPGA Blocking version of the calculate_forces function

Name	B_18Kb	DSP48E	FFs	LUTs
1 128 Acc	287 (15.7%)	642 (25.5%)	76147 (13.9%)	54462 (19.9%)
1 256 Acc	648 (35.5%)	1280 (50.8%)	183646 (33.5%)	107207 (39.1%)
3 128 Acc	537 (58.9%)	1920 (76.2%)	311271 (56.8%)	169670 (61.9%)
3 256 DF	644 (70.6%)	1925 (76.4%)	341902 (62.4%)	208906 (76.2%)

Table III
RESOURCES USED BY MATRIX MULTIPLY KERNELS

available. To obtain some parallelism over this last version, a new FPGA accelerator was developed. This new version receives the list of blocks to compute and iterates over them. With this approach (**FPGA SB** columns in Figure 7 and **SUBBLOCK** row in Table II) two instances of the accelerator fit in the FPGA and were able to obtain a 1.87x over the previous version and a 128x over 1 SMP core.

B. Implements and Dataflow Performance impact

In order to obtain the best possible performance out of heterogeneous systems it is essential to use all the available resources whenever it is possible. In addition to making easy the programmability of FPGA accelerators and taking care of the necessary data transfers, OmpSs@FPGA also presents a *implements* clause that is really useful for heterogeneous systems.

Along with the implements and the Matrix Multiply presented above, different matrix multiply accelerator sizes were tested in the FPGA fabric available in the system. Table III shows in rows **1 128 Acc**, **3 128 Acc** and **1 256 Acc** how many resources took to synthesize one or three accelerators of **BS** size 128×128 (128), or one accelerator of **BS** size 256×256 (256). All the accelerators listed worked at 300MHz and as it can be deduced from the reported sizes it is not possible to fit four 128 size accelerators or two 256 size accelerators in the FPGA.

Figure 9 displays the performance obtained by the three different approaches. Columns labeled **0 SMP** show the performance obtained by executing the application in the FPGA accelerators alone, while columns **1 SMP** to **4 SMP** display the result of using, in parallel with the accelerators, from 1 to 4 threads running tasks with the OpenBLAS code in Figure 4. As it can be seen, the best approach in terms of performance is not to fit a single large accelerator

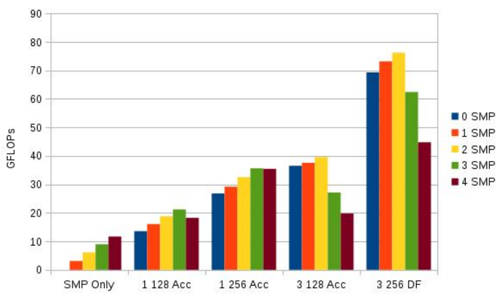


Figure 9. GFLOPs for Matrix Multiply with different FPGA accelerators

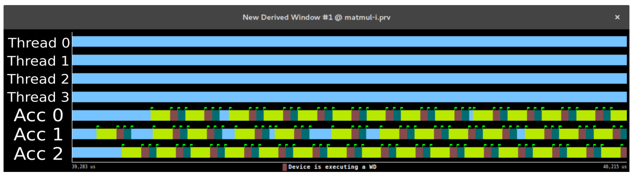


Figure 10. FPGA exec. trace with 3 128 Matrix Multiply accelerators

Name	B_18Kb	DSP48E	FFs	LUTs
fgemm32	68 (3.7%)	160 (6.4%)	19771 (3.6%)	15559 (5.7%)
fsyrk32	36 (2.0%)	160 (6.4%)	19822 (3.6%)	16149 (5.9%)
ftrsm32	36 (2.0%)	104 (4.1%)	11482 (2.1%)	10875 (4.0%)
fpotrf32	10 (0.6%)	22 (0.9%)	3487 (0.6%)	3302 (1.2%)
fgemm64	74 (4.1%)	160 (6.4%)	23887 (4.4%)	30032 (11.0%)
fsyrk64	42 (2.3%)	160 (6.4%)	23849 (4.4%)	30727 (11.2%)
ftrsm64	42 (2.3%)	250 (9.9%)	28734 (5.2%)	25753 (9.4%)
fpotrf64	28 (1.5%)	22 (0.9%)	3514 (0.6%)	3350 (1.2%)

Table IV
RESOURCES USED BY CHOLESKY KERNELS IN XCZU9EG-FFVC900

(**1 256 Acc**) but to use three smaller accelerators in parallel. Also, note that for any possible solution, the use of the SMP to compute matrix multiplication blocks in parallel always improves the resulting performance. When using four threads with 1 accelerator or three or more threads with 3 accelerators the performance drops from the maximum. This is due to the fact that there is over-subscription. Effectively, with 1 accelerator, 1 thread is used to send tasks to the FPGA. With three accelerators, 2 threads should be used in order to feed the accelerators properly. However, the runtime is intelligent enough to always use the best approach in the default configuration given the maximum performance for every configuration. Also, it is important to note that the exploration of all these possibilities is done with the same code, changing only the *BS* size and the number of instances of the accelerators, so reducing the programmability effort to a minimum.

In order to further improve the performance results, the trace of the execution with 3 128 Matrix Multiply accelerators was extracted. Figure 10 shows the trace of this execution. The four top lines in blue represent the threads and do not show any information. The three bottom lines show in blue when the corresponding FPGA accelerator is not working and in yellow when it is reading data. In brown it can be seen the computation time and in green the writing of the output matrix back to the memory. As it can be seen from the trace, the computation time is around 4 times shorter than the data movement time. Furthermore, the data copies are not overlapped with the computation. From this observation a way to improve the accelerators was devised. The idea is that doubling the accelerator size increases the data size by four times but the computation size by eight times. Therefore, if the number of cycles per operation is not significantly increased this will result in a better balance in the accelerator design while not increasing the DSP usage.

Following this idea to design better balanced accelerators, the number of computations were limited by setting the initiation interval of the innermost loop to 2 cycles (so making the same effective computations in the 256 DF accelerator as in the 128 one). Also the *DATAFLOW* pragma was used in order to overlap the data copies with the computation. The result of these changes can be observed in Table III row **3 256 DF** and Figure 9 column **3 256 DF**. These new accelerators fit in the FPGA available in the system while nearly doubling the performance of the previous 3 128 accelerators. They take the same time to compute the results but transfer half the blocks and overlap part of these transfers with the computation. In addition the *implements* clause still adds some performance to the FPGA by using the SMP achieving 76 GFLOPs with little more that 5 Watts consumption.

C. *Heterogeneity and Programmability impact*

Another common problem when dealing with complex applications applications composed by different kernels in heterogeneous environments is how to distribute such kernels over all the different resources. The OmpSs@FPGA environment can help with this distribution by allowing different possible mappings to be tested easily and without burden to the programmer.

Cholesky decomposition application is composed of four different kernel tasks that present a complex dependence pattern that grows exponentially with the size of the problem. Table IV shows the FPGA resources used by each kernel of the Cholesky application when implemented in the FPGA when different blocking sizes were used in the application. As it can be extrapolated from the results, it is impossible to fit all the 4 kernels with a blocking size of 128 in the FPGA due to the limited resources available. However, from previous results it is known that the bigger the accelerator, the better the obtained performance.

One of the first implications of using one accelerator for each kernel is that the execution in the FPGA would be a sequential one. However, using the *implements* clause explained in the previous section would help us to improve the performance significantly by allowing threads to also execute kernel functions. Figure 11 shows the time used when executing 8 different versions of the same Cholesky problem to solve a 2k equation matrix. Two implementations with different block sizes (32 and 64) were tested using the SMP cores to solve the problem (using OpenBLAS implementations of the kernels), using the FPGA accelerators to solve the same problem and also using both (through the implements clause).

As it can be seen in Figure 11 the initial FPGA

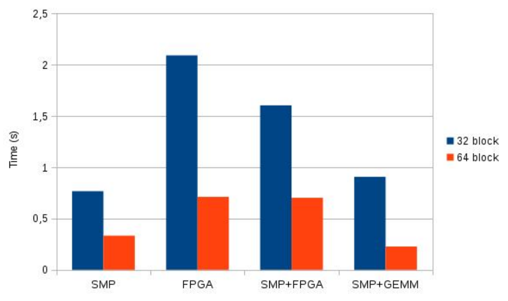


Figure 11. Time of Cholesky execution with different task mappings

alternative (column FPGA) of the code do not lead to good performance results compared against the SMP only alternative. Even the version that uses both SMP and FPGA (SMP+FPGA) to compute the result is slower than the SMP only version. On one hand, the reason behind this behavior is that including four different accelerators in the FPGA limits the performance of each accelerator. On the other hand, the Cholesky algorithm is not well-balance among all its kernels. As an example, when using tasks that operate over blocks of 32 by 32 elements, there are 41664 *gemm* tasks and only 4096 tasks of the other types. Even when the *implements* clause is used and the tasks can be executed in both SMP cores and FPGA (**SMP+FPGA**), mainly all the tasks executed in SMP cores in parallel with the accelerators are *gemm* tasks.

To solve the aforementioned unbalance, a second FPGA accelerated version of Cholesky (**SMP+GEMM**) is implemented with 4 *gemm* accelerators on the FPGA. The remaining kernels were implemented using the SMP cores. From the point of view of the programmer, this new version only implies increasing the number of instances of the *gemm* FPGA accelerator and not including the instances of the other accelerators, a change that can easily done in the source code. The rest of the whole program remains exactly the same. As it can be seen in Figure 11 columns **SMP+GEMM**, the performance is significantly better with this approach. This last version outperforms the initial SMP only version and illustrates how using the OmpSs@FPGA framework simplifies the accelerator space exploration keeping the necessary changes made by the programmer to a minimum.

VII. Conclusions

This paper presents the OmpSs@FPGA ecosystem that greatly improves programmability when dealing with heterogeneous systems that involve SMPs and FPGAs. OmpSs@FPGA not only offloads application functionality to FPGAs, it also takes care of data movements, replication of accelerators, implementation of the same task in different computing units and parallel execution in all the available resources decided at runtime.

This paper demonstrates how the framework can be used to accelerate different applications through different techniques. The results show that OmpSs@FPGA facilitates high-level language programming on systems that integrate FPGAs, being key to obtain performance out of them with a reasonable amount of effort.

Acknowledgments

This work is partially supported by the European Union H2020 program through the EuroEXA project (grant 754337), and HiPEAC (GA 687698), by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (TIN2015-65316-P) and the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d'Execució Paral·lels (2014-SGR-1051).

References

[1] Vineyard, "Objectives and rationales of the project," 2018, vineyard-h2020.eu/en/project/objectives-and-rationale-of-the-project.html.

[2] Maxeler, Inc., "The open spatial programming language," 2014, openspl.org.

[3] Khronos Group, Inc. (2018) Opencl. [Online]. Available: www.khronos.org/opencl/

[4] Xilinx, Inc. (2018, July) Sdsoc development environment. [Online]. Available: www.xilinx.com/sdsoc

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

[6] Xilinx, Inc. (2017, September) Vivado High-Level Synthesis. [Online]. Available: www.xilinx.com/hls

[7] A. Canis *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.

[8] B. Fort *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 2014, pp. 120–129.

[9] F. Sainz *et al.*, "Leveraging ompss to exploit hardware accelerators," in *26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France*, 2014, pp. 112–119.

[10] A. Duran *et al.*, "Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[11] S. Neuendorffer and F. Martinez-Vallina, "Building Zynq®Accelerators with Vivado®High Level Synthesis," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 1–2.

[12] Intel Corp. (2017, September) Quartus Prime. [Online]. Available: www.altera.com/products/design-software/fpga-design/quartus-prime/what-s-new.html

[13] PM - BSC. (2018, September) OmpSs@FPGA. [Online]. Available: pm.bsc.es/ompss-at-fpga

[14] BSC-CNS. (2016) Performance Tools. [online]. www.bsc.es/computer-sciences/performance-tools.

[15] Xilinx, Inc. (2017) ZYNQ UltraScale+ MPSoC Overview. [Online]. Available: www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf