An FPGA target for the StarPU heterogeneous runtime system

Georgios Christodoulis*, Manuel Selva*,
François Broquedis†, Frédéric Desprez*, Olivier Muller‡
* Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
† Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG
‡ Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, TIMA
*,† *firstname.lastname@inria.fr,* ‡ *firstname.lastname@grenoble-inp.fr*

***Abstract*—Heterogeneity in HPC nodes appears as a promising solution to improve the execution of a wide range of scientific applications, regarding both performance and energy consumption. Unlike CPUs and GPUs, FPGAs can be configured to fit the application needs, making them an appealing target to extend traditional heterogeneous HPC architectures. However, exploiting them requires an in-depth knowledge of low-level hardware and high expertise on vendor-provided tools, which should not be the primary concern of HPC application programmers. In this paper, we present the first results of the HEAVEN project that aims at designing a framework enabling a more straightforward development of scientific applications over FPGA enhanced platforms. Our work is concentrated on providing a framework, which will require the minimum knowledge of the underlying architecture, as well as fewer changes to the existing code. To fulfill these requirements, we extend the StarPU task programming library that initially targets heterogeneous architectures to support FPGA. We use Vivado HLS, a high-level synthesis tool to deliver efficient hardware implementations of the tasks from high-level languages like C/C++. For the evaluation of our proposal, we present code snippets for a blocking version of matrix multiplication, illustrating the ease of development our approach delivers. We also show preliminary results regarding the performance of the FPGA version, which validate our proof-of-concept implementation.**

***Index Terms*—FPGA, StarPU, Vivado HLS, Task programming system, Heterogeneous runtime system**

I. Introduction

Computer architectures are getting more and more complicated, exposing massive parallelism, hierarchically-organized memories, and heterogeneous processing units. The complexity of the platforms above imposes a compromise between portability and performance. To exploit heterogeneous processing units integrated into a single machine, programmers often need to deal with hardware-specific low-level mechanisms explicitly. For example, they must expressly handle memory transfers between the main memory and the private memories of the accelerators.

Solutions have been proposed to tackle the portability versus performance dilemma. Existing parallel programming models and tools evolved towards the support of heterogeneous architectures, broadly adopting the Graphical Processing Units (GPU) as complementary computational resources to the general purpose processors (CPU).

Many of these parallel programming systems rely on the concept of tasks. A task is an autonomous work unit that can be seen as a portion of code with inputs and outputs, possibly depending on other tasks. Within this model, an application is organized in the form of a dynamically created task graph. Tasks are later assigned to processing units by a software layer called the runtime system.

Tasks prevalence is mainly due to their affordable cost and flexibility, enabling the implementation of dynamic load balancing strategies with very low overhead. Unfortunately, none of the well-established task-based parallel programming models for heterogeneous architectures support Field-Programmable Gate Arrays (FPGA).

Unlike CPUs and GPUs, FPGAs can be configured to fit the application needs. They contain arrays of programmable logic blocks that can be wired together to build a circuit specialized to the targeted application. For example, FPGAs can be configured to accelerate portions of code that are known to perform poorly on CPU or GPU [1]. The energy efficiency of FPGAs is also one of the primary assets of this kind of accelerators compared to GPUs, which encourages the scientific community to consider them as one of the building blocks of large-scale low-power heterogeneous platforms.

Unfortunately, their programming complexity has prevented High-Performance Computing (HPC) developers from using them broadly. When the entry point to hardware design was a Register Transfer Level (RTL) description, with Hardware Description Languages (HDL) like VHDL and Verilog, FPGAs were mostly confined to hardware designers. The capabilities delivered by modern High-Level Synthesis (HLS) tools bridge the gap between software programmers and FPGA. HLS tools generate an RTL description of an algorithm, provided typically in C or C++. Despite their rapid evolution, those tools, like Catapult from Calypto or Vivado HLS from Xilinx, still requires a certain level of expertise in hardware design to benefit from their abilities fully.

Our contribution is a framework, which enables developers to use FPGA as another accelerator within a traditionally heterogeneous HPC platform. This framework relies on existing HLS tools and heterogeneous task programming frameworks. We focused on ease of development and portability without sacrificing performance.

The remainder of the paper is organized as follows. Section II describes related work. Section III introduces the background for this work. Section IV presents the fundamental concepts behind our framework and the way we implemented it. Section V gives an evaluation of the HEAVEN framework, while conclusion and future works lie in Section VI.

II. Related Work

Numerous proposals to ease the programmability of an FPGA enhanced platform have appeared with different features regarding the types of supported processing units, the programming abstraction (from tasks to threads), the way they manage data transfers as well as their scheduling support.

OpenCL [2], [3] is a standard framework offering both a device-side language and a host management layer to exploit multiple hardware accelerators from a single application, created initially for GPU and later on extended for FPGA. The device-side language, called OpenCL C, implements both task and data parallelism and provides ways to map data efficiently on specialized processor devices. The application programmer can attach command queues to processing elements and explicitly assign them tasks from the application. This way, OpenCL stands as a great programming environment to experiment with heterogeneous architectures and implement higher-level layers, like runtime systems but may appear too low-level for application programmers, as the assignment of tasks and the data transfers between host and device memories are explicit.

At thread level, hthreads [4], is a unifying programming model aiming to specify application threads running within a hybrid CPU/FPGA system. A unified multiprocessor abstracts every component of the platform. Application developers can express the concurrency of their computations using threads, arranged in a single program, which can either be compiled to run on a CPU or synthesized to run on an FPGA.

The Heterogeneous System Architecture Foundation [5] (HSA) is a consortium of industrial and academic partners discussing ways of standardizing a whole ecosystem for heterogeneous programming. In particular, HSA provides a unified view of fundamental computing elements. HSA allows writing applications that seamlessly integrate CPUs (called latency compute units) with GPUs (called throughput compute units) while benefiting from the best attributes of each. The essence of the HSA strategy is to create a single unified programming platform providing a strong foundation for the development of languages, frameworks, and applications that exploit parallelism. The HSA platform is designed to support high-level parallel programming languages and models, including C++, OpenCL, OpenMP, Java, and Python, to name a few. HSA-aware tools generate program binaries that can execute on HSA-enabled systems supporting multiple instruction sets and also can run on existing architectures without HSA support. An HSA implementation is a system consisting of a heterogeneous hardware platform that integrates both CPUs and GPUs, which operate coherently in shared memory, a software compilation stack, a user-space runtime system, and some kernel-space system components.

The OpenMP accelerator support provides explicit ways of offloading portions of applications to accelerator devices. Relying on this OpenMP support, recent work [6] have been proposed to handle FPGA. This work focuses on making data transfers automatic based on dependencies provided at the OpenMP level and does not discuss heterogeneous scheduling questions. Relying on StarPU, our proposal will benefit from smart scheduling heuristics based on cost models for both computation and data transfers times.

OmpSs is a programming environment developed at BSC that enables the execution of task-based applications on heterogeneous platforms. It can be seen as a fork of OpenMP with original extensions. Like OpenMP, OmpSs provides compiler directives to turn serial C, C++ and Fortran applications into parallel applications initially able to offload computations on GPU. The programming environment comes with a dedicated compiler and a runtime system responsible for distributing tasks over the CPUs and the accelerator devices, managing data transfers between host and devices memories in a transparent way. OmpSs has then been extended to support offloading on FPGA [7]. Compared to our proposal, this extension focuses on embedded heterogeneous architectures such as the Zynq from Xilinx integrating a CPU and an FPGA on the same die. Blaze [8] provides programming and runtime support that enables rapid and efficient deployment of FPGA accelerators at warehouse scale. It builds upon Apache Spark, a widely used framework for writing Big Data processing applications.

Among many industrial attempts to use FPGA as an accelerator in a heterogeneous environment, Amazon EC2 F1 instances can be considered to deliver high flexibility to developers [9]. Instances come in two sizes, with up to eight 16 nm Xilinx UltraScale Plus FPGA devices connected to each one of them through PCIe. They provide all the necessary tools to develop an AFI (Amazon FPGA Image) that can later be used to configure one of the devices of the instance. The runtime decisions are yet to be managed by the developer though.

III. Background

We now introduce the background required for our work. In this paper, we use the term *host* to refer to the general purpose processor of the machine along with its main RAM, and with all its software environment including its operating system.

A. Vivado HLS and Vivado

Vivado HLS is one of the broadly used HLS tools. From a C++ description of a kernel, Vivado HLS automatically generates an RTL description. Vivado HLS delivers performance by allowing the developer to monitor and tune the behavior of its kernel by using some *pragmas* to exploit the fundamentally concurrent parts of it. From an HPC developer, a relatively efficient hardware description of a C/C++ application should arrive after a basic understanding of the design principles through the rich documentation and code examples the tool provides. Once an RTL description has been generated by

Vivado HLS, it needs first to be instantiated along with all the other required components for the final design. Then, the complete RTL description of the final design is synthesized to logic gates by Vivado.

B. RIFFA: The link between the host and the FPGA

RIFFA (Reusable Integration Framework for FPGA Accelerators) is a simple framework for communicating data from a host to a FPGA via a PCIe bus [10]. It is then made of a host part and a FPGA part. RIFFA is available on Linux and for several series of boards, from different vendors such as AVNet, Xilinx and Altera.

On the host side there is a clean interface to exchange data between the host and the device, that relies on a custom Linux kernel driver. The communication is managed using direct memory accesses transfers and interrupt signaling.

On the FPGA side, RIFFA provides a clean interface for accessing up to twelve signal controlled completely independent bidirectional channels. Each channel has two sets of signals; one for receiving and one for sending data. For a transaction, the sender has to notify the receiver for the upcoming operation, providing supplementary information regarding the data size, a potential offset, and whether other transactions are expected or not. Once they receive the acknowledgment from the receiver, they synchronize themselves using a two signals handshaking protocol for the consumption of the data.

C. StarPU

StarPU [11] is a task-based runtime system targeting heterogeneous machines comprising several kinds of processing units, from traditional multicore CPUs to GPU accelerators and coprocessors like the Intel Xeon Phi. StarPU comes with an API to express task parallelism from a C/C++ or Fortran application. StarPU is data-flow based, meaning the programmer has to specify for each task the input and output data, thus creating *dependencies* between tasks. This way, a StarPU application generates a *graph* of tasks at runtime, with edges representing the dependencies between them. Unlike most task-based parallel programming libraries, each StarPUtask can embed multiple implementations, one for each kind of processing unit for example. When a task becomes ready, the StarPU tasks scheduler dynamically assigns it to what it considers to be the best processing unit at the given time. To do so, StarPU takes into account the availability of target-specific implementations for the current task, the availability of the hardware resource at the given time and some model-based performance estimation of the task on the different processors/accelerators. On top of that, StarPU also handles data transfers automatically and transparently, and can make smart decisions on whether some data should stay in the accelerator memory or not, depending on the input data of the next tasks to be executed.

In this work, we focus on using the C/C++ API of StarPU. In this case, once the different tasks making the application have been identified, the programmer has to define task types, called *codelets* in the StarPU terminology. A codelet establishes the number of inputs and outputs of a task along with all its implementation for one or more different types of processing units. Once codelets have been defined, the application developer has to create the main StarPU program. This program first registers in the StarPU runtime, the inputs and outputs data that will be read and produced by tasks. Then the program creates tasks along with their codelets and their inputs and outputs and submits them to the StarPU runtime system. The task graph represents the execution of a StarPU application.As a consequence, it is fully dynamic and can, for example, depend on user-specified inputs.

When a task is submitted, the StarPU runtime first checks for input dependencies. If these dependencies are satisfied, the scheduler decides on which processing units the task will execute, among all the possible ones specified in the task codelet. At this time, the runtime pushes the task into the ready queue of the chosen processing unit. If some input dependencies are not satisfied, the runtime pushes the task into a global waiting queue instead.

At initialization time, StarPU creates a *worker* for each processing unit of the underlying machine. A worker is a thread, running on the host, responsible for popping tasks the runtime has scheduled in its ready queue and for executing them. The worker is also responsible for handling data transfers required for the task execution.

Regarding the scheduling decisions, StarPU comes with many policies which can be chosen by the application. Many of these strategies rely on cost models for task execution and data transfers. These models can either be initialized from a previous run of the application or from within the runtime system itself, by profiling during the execution.

IV. The HEAVEN Framework

This section describes in details the HEAVEN framework. We first present the components involved in the framework with a focus on our contributions. Then, we present the HEAVEN framework from the user point of view, showing how to use it.

A. Architecture

Figure 1 shows all the components involved in the HEAVEN framework. Because we target heterogeneous machines including FPGA accelerators, the framework is split between the host part shown on the top of the figure, and the FPGA part displayed on the bottom. The red components of the figure are the contributions of this work. Because we base our framework on StarPU, a complete figure would also include the already-supported GPU side. For clarity, only the FPGA side which is our contribution is shown here.

On the host's side, we then had to extend StarPU to integrate the capability to use FPGA accelerators. To ease the communication between this extended version of StarPU and the FPGA, we developed Conor, a library providing abstractions on top of RIFFA.

On the FPGA's side, we developed a connector allowing to send inputs and get outputs to and from hardware tasks generated from Vivado HLS. This connector enables the extended

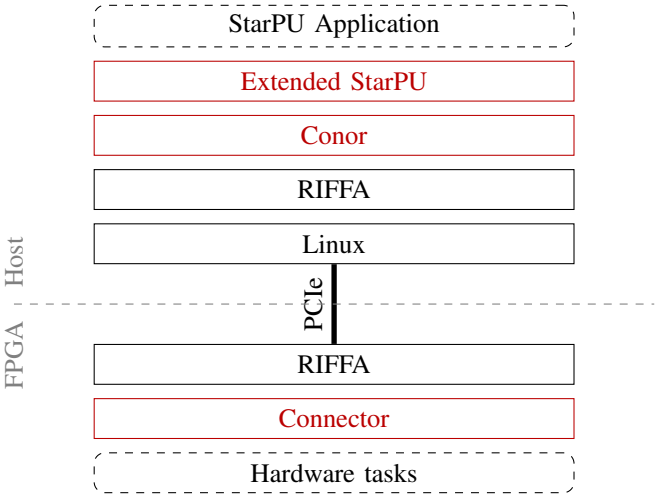


Fig. 1: Overview of the different components involved in the HEAVEN framework. The upper part of the diagram shows the components on the host's side, while the lower part shows the components on the FPGA's side. The red components are the contributions of this work and the dashed components are the ones to be provided by the application developer.

StarPU runtime running on the host side to communicate with the hardware tasks through PCIe without any intervention of the application developer. In the current version of the framework, the configuration of the FPGA must be done once, before the execution of the StarPU application. The framework supports up to five FPGA devices, while the number of tasks StarPU can execute on each one of them is constrained by the number of available communication channels as well as the spatial constraints. In the current version of the framework, if several hardware tasks are present in the system, either on several FPGA devices and a single FPGA device, they must all have the same type. Said differently, they must expect the same inputs and provide the same outputs.

We now describe the internal of the components at the heart of this work. The components provided by the application developer, shown with dashes in Figure 1 will be described in detail in Section IV-B.

1) Extended StarPU: To include FPGA support in StarPU, we created a new type of worker dedicated to FPGA devices. At initialization time, a worker is created for each hardware task present on the FPGA devices of the machine as shown in Figure 2. As any other StarPU's worker, each one of these FPGA workers has its own tasks ready queue which is filled by the StarPU's scheduler.

The role of the worker is then to pop tasks from this queue and to launch their execution on the FPGA side. When the worker is idle, it pops the ready task on the top of the queue and first send its input data to the FPGA through Conor. This sending operation is currently done by a function provided by the programmer as shown in Section IV-B. Nevertheless, the extended StarPU runtime has all the required information so that it can automatically perform this operation. Sending the inputs data triggers the computation on the FPGA side automatically, as soon as all data have been transmitted. Then, the function provided by the programmer must wait for the output data to come back from the FPGA.

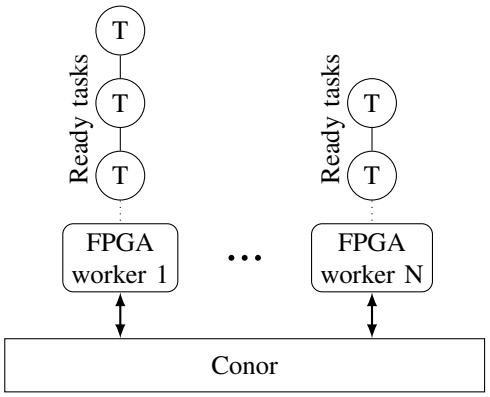


Fig. 2: One FPGA worker is created for each hardware task present in the machine. Workers communicate with the FPGA through Conor only.

To completely decouple the StarPU runtime and the StarPU application from low level concerns about communication with the FPGA, we developed a communication library called Conor. All the FPGA communications happening inside StarPU go through Conor using its simple API for sending and receiving data.

2) Conor: The Conor FPGA communication library is built on top of RIFFA as shown in Figure 1. RIFFA provides a primitive infrastructure to send and receive non-structured data to the device, where low-level information should still be managed for a clean adaptation.

Conor first provides functions for the transmission of structured data, such as arrays and vectors, over the PCIe in an optimal way. These functions abstract away all the low-level hardware concerns, and just let the caller say "send this buffer which size is that to the FPGA". Conor also provides functions to handle the control signals regarding the state of the device. These functions allow controlling the coherency of the state of the board.

3) Connector: On the FPGA side, as shown on the bottom half of Figure 1, our framework includes a connector allowing the communication between hardware tasks created with Vivado HLS and RIFFA. Figure 3 shows the internals of this connector. On the hardware tasks side, the framework relies on the `ap_hs` protocol of Vivado HLS to handle the communication. The data transfers are controlled by a two signals handshake protocol. On the other hand, a RIFFA channel has two sets of signals, one for receiving and one for sending data. The transfers are also controlled by a handshake protocol on two phases, one to inform the parties about the type of the upcoming transaction, and another to perform the transaction itself. The connector ensures the consistency of the communication between these two different protocols using

information provided by the hardware tasks through the control stream.

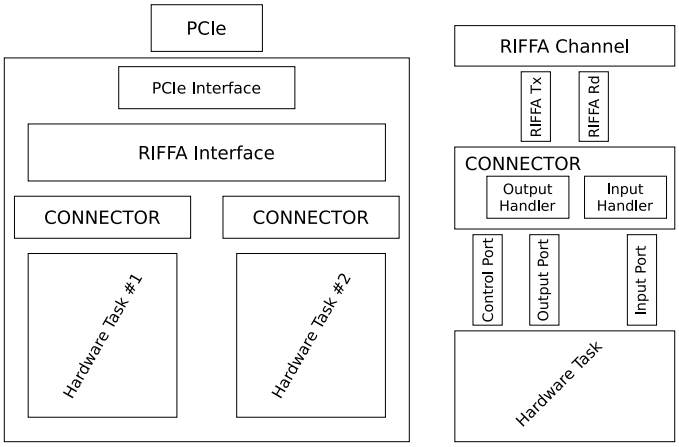


Fig. 3: The connector allows communication between RIFFA and the hardware tasks.

B. Developing an application

To illustrate how to use the HEAVEN framework, we will focus on a basic application consisting in computing the product of two matrices by blocks, as shown in Figure 4. In this example, each square block represents a sub-part of a matrix with a specific size, let's say 64 by 64. Then, the matrix A is of size 192x64, B is of size 64x192, and C is of size 192x192. To compute the content of the matrix C, we can perform in parallel the computation of the nine blocks making it. Each one of these sub-computations will require a sub-part of matrix A and a sub-part of matrix B as inputs.

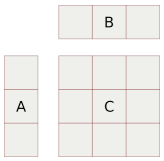


Fig. 4: Blocked matrix multiplication example. A is of size 192x64, B is of size 64x192 and C is of size 192x192.

To describe this application using the HEAVEN framework, the application developer must write both the StarPU application along with the hardware implementation of its tasks as shown by dashed boxes in Figure 1. In the case of the blocked matrix multiplication we use in this section, tasks are completely independent. Nevertheless, because the HEAVEN framework is an extension to StarPU, it also supports applications with task dependencies.

1) Writing the StarPU application: First, the programmer must describe the codelet for the tasks computing one block in the result matrix. Figure 5 shows how to do that using StarPU's C API. This codelet provides both a CPU implementation and an FPGA one. It also specifies that the task has two inputs and one output through the `nbuffers` and `modes` attributes.

```
static struct starpu_codelet cl = {
  .cpu_funcs =  {cpu_mult},
  .fpga_funcs = fpga_mult,
  .nbuffers = 3,
  .modes = {READ, READ, WRITE}
}
```

Fig. 5: Task codelet for a task computing one block of the result matrix. The task has two inputs and one output. The codelet provides both a CPU implementation and an FPGA one.

Then, the `fpga_mult` function referenced in the codelet must be defined has shown in Figure 6. This function is mainly divided in four steps:

- Ask to Conor to reserve an hardware task;
- Ask to StarPU where are inputs and outputs and on which part of them the task must operate;
- Ask to Conor for the result and then copy back this result into the C matrix.

In this function, the order used to send the inputs is crucial. It must be aligned with the expected inputs order in the hardware task implementations as explained in Section IV-B2.

Again, this function is not yet automatically generated by the extended StarPU runtime but all the information required to do so is available to it. Said differently, in the next version of the framework, this function will no more be written by the programmer. The runtime will automatically send and receive inputs and outputs using the information associated with the task.

Then the programmer must write the main part of the application responsible for allocating matrices, register them with StarPU, launch tasks and wait for their completion as shown in Figure 7.

The procedure for launching the task is completely uniform; it does not know anything about task types as shown in Figure 8. It consists of creating the tasks, associating them with the codelet defined in Figure 5, setting their dependencies, and finally submitting them.

2) Writing hardware tasks: On the FPGA side, the application developer must provide the implementation of hardware tasks. The HEAVEN framework relies on Vivado HLS for this task. Using this tool, the application programmer describes the implementation of its hardware tasks in C++. This description consists in:

- Receive the block from A and B;
- Compute the block of C as the product of the blocks from A and B;
- Send this C computed block.

For receiving inputs and sending outputs, the application developer is provided two high-level objects which type is Vivado HLS `hls::stream`. The first one is used to receive inputs and the second one to send outputs. The function also has a third `hls::stream` parameter used for control. The interface between these three high-level objects with the PCIe

```
void fpga_mult(void *d[]) {

  /* Ask Conor for a channel, or
   * equivalently for a hardware task
   */
  int chnl = conor_reserve_a_chanel();

  /* Get inputs from STARPU */
  int* subA = SPU_MATRIX_GET_PTR(d[0]);
  int* subB = SPU_MATRIX_GET_PTR(d[1]);
  int* subC = SPU_MATRIX_GET_PTR(d[2]);

  /* Get info on which part of the
   * inputs the task must operate
   */
  uint32_t nyA= SPU_MATRIX_GET_NY(d[0]);
  uint32_t ldA= SPU_MATRIX_GET_LD(d[0]);
  // Same for B and C

  /* Send A and B */
  int buf_s[nyA], buf_r[nxC*nyC];
  conor_trans sent, recv;
  for (uint32_t j = 0; j < nxC; j++){
    for (uint32_t k = 0; k < nyA; k++)
      buf_s[k] = subA[j+k*ldA];
    conor_data_send(chnl, buf_s, nyA);
  }
  for (uint32_t i = 0; i < nyC; i++){
    for (uint32_t k = 0; k < nyA; k++)
      buf_s[k] = subB[k+i*ldB];
    conor_data_send(chnl, buf_s, nyA);
  }
  /* Receive C. This is blocking */
  conor_data_recv(chnl, buf_r, nxC*nyC);
  for (uint32_t i = 0; i < nxC; i++){
    for (uint32_t j = 0; j < nyC; j++)
      subC[j + i*ldC] = buf_r[i*nyC+j];
  }
  conor_release_chanel(chnl);
}
```

Fig. 6: Definition of the fpga_mult function.

```
/* Init and regist A, B and C */
init_and_register_data();
/* Partition data into blocks */
partition_data();
/* Submit all tasks */
ret = launch tasks();
/* Wait for termination */
starpu_task_wait_for_all();
```

Fig. 7: Main part of the StarPU application.

bus is handled transparently by the framework through the connector and RIFFA as described in Section IV-A.

For the receiving part of the function, the programmer must read the blocks for the A and B input matrices in a given order and save them in local variables which are automatically translated to BRAM blocks by Vivado HLS. This reading order must be respected in StarPU when inputs are sent to the FPGA as presented in Section IV-B1.

For the computation step, the application developer has first to declare the C matrix as a local variable that will also be allocated into BRAM blocks by Vivado HLS. Then the

```
for (uint32_t x = 0; x < 9; x++) {
  for (uint32_t y = 0; y < 9; y++) {
    spu_task* task = spu_task_create();
    task->cl = &cl;
    /* Get handlers for each block */
    task->handles[0] = spu_get_sub_data(
      A_handle, 1, y);
    task->handles[1] = spu_get_sub_data(
      B_handle, 1, x);
    task->handles[2] = spu_get_sub_data(
      C_handle, 2, x, y);
    starpu_task_submit(task);
  }
}
```

Fig. 8: Submit tasks to the StarPU runtime.

effective computation of C is done using three nested for loops just as in a software implementation.

Finally, the C matrix must be sent back to host side using the second `hls::stream` object provided to the function. In case of tasks with several outputs, as for the inputs, the sending order must be respected on the StarPU side when receiving the data.

3) Generating the bitstream: Once the RTL description of the hardware tasks has been generated with Vivado HLS, the application developer must generate the final bitstream to be used to configure the FPGA. For that task, the HEAVEN framework provides a parametric wrapper written in Verilog. This wrapper is only configured by the application developer to specify the number of hardware tasks to be instantiated. The wrapper combines the specified number of hardware tasks along with the hardware part of RIFFA, and the connector between RIFFA and the hardware tasks. The application developer then uses Vivado to synthesize the complete design to the final bitstream.

V. Preliminary Evaluation

A. Environment of experiments

We executed the experiments in a hybrid machine, where the FPGA was connected to the host via PCIe. The host CPU was a 64-bit Intel Xeon W3530, a two-way hyperthreaded quad-core, for a total of 8 virtual cores, running at the speed of 2.80GHz, with a smart cache of 8MB, 256KB of L2 and 8192KB of L3, a thermal design reference power of 130W, and a semiconductor size of 45nm. It was coupled with 12GB of DRAM at 1333MHz.

The FPGA we used for our experiments was Xilinx Virtex-7 VC709, a board using the XC7VX690T chip, with 693,120 logic cells, 3,600 DSP slices, 52,920kb of BRAM memory, up to 4GB of RAM at 1866Mbps, and an 8-lane PCIe edge connector.

B. Application set-up

Within the context of this study, we opted to evaluate the performance we could obtain using our platform on the blocking version of matrix multiplication introduced in Section IV-B. For the given analysis, we did not consider

	Light-weight Task	heavy-weight Task
FPGA	1.40 msec	7.01 msec
CPU	1.63 msec	113.37 msec

TABLE I: Performance of a single task for each architecture and task size.

hybrid scenarios, where the scheduler could use within the same execution CPU and FPGA implementations for the tasks. Thus, for every experiment under test, we went through two executions, one using only FPGA tasks and one using only CPU tasks.

We expected that the communication overhead would significantly downgrade the performance of the FPGA version. As a consequence, to study this effect, we focused on two constant task sizes for the entire range of experiments. A *light-weight* task operates on blocks of 64x64 integers while a *heavy-weight* task operates on 256x256 blocks.

We also vary the size of the input matrices A and B. We evaluated nine different sizes as shown in Figure 9. For each one of these nine scenarios, the number of tasks that can be deduced from the way the input matrices were sliced. Each scenario is further divided in two by using either light-weight of heavy-weight tasks. For example, the scenario in Figure 9e corresponds to a setup for the 64x64 task size and to another setup for the 256x256 one. In this scenario, for both 64x64 and 256x256 setups the A and B matrices are split into five slices leading to a total of 25 tasks. In the 64x64 setup the size of A is then 320x64 and the size of B is 64x320. In the 256x256 setup the size of A is then 1280x256 and the size of B is 256x1280.

In all our experiments, the FPGA was configured with four instances of the hardware task computing a single block of the result matrix. Hence for both architectures, CPU and FPGA we had an equal amount of independent computational entities because in the CPU case, StarPU allocates by default a number of workers equals to the number of physical cores in the host (4 in our case).

C. Results

We first evaluated the per task execution time for each architecture. This corresponds to the scenario presented in Figure 9a. In that case, the computation of the block corresponded to the entire application. The per-task performance we obtained is presented in I. The bigger task shows a significant performance difference when executed on an FPGA compared to when executed on a CPU. In the smaller task, the ratio between the computation and the communication is small, resulting in comparable behavior for both architectures. With the increase of the task size, the cost of data transfer (linear) has a minor impact compared to the cost of the computations (cubic).

Then we evaluated how the extended StarPU runtime behaves when the number of tasks increases. In this case, the expected total execution time should theoretically, follow the formula:

$$Time = \begin{cases} tpt + \alpha & ,\#tasks < \#conTasks \\ \dfrac{\#tasks}{\#conTasks} \times tpt + \alpha & ,\#tasks \geq \#conTasks \end{cases} \quad (1)$$

In this formula, $Time$ refers to the total execution time of the application, tpt to the execution time of a single task which value is shown in Table I, $\#tasks$ to the number of tasks, $\#conTasks$ to the number of concurrent tasks and α to a constant representing the initialization overhead. Since within our experiments we could execute up to four tasks concurrently in every scenario, the expected total execution time should follow the:

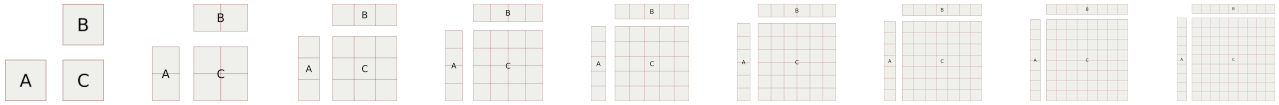
$$Time = \begin{cases} tp + \alpha & ,\#tasks < 4 \\ \dfrac{\#tasks}{4} \times tpt + \alpha & ,\#tasks \geq 4 \end{cases} \quad (2)$$

Figure 10 shows how the total execution time evolves with the number of tasks both for the CPU only version and the FPGA only version. For both light-weight and heavy-weight tasks, Figures 10a and Figure 10b, we observe a linear evolution of the total execution time validating the theoretically expected one. We can then conclude that for the sizes studied in our experiments, there is no additional overhead coming from task management.

VI. Conclusion And Perspectives

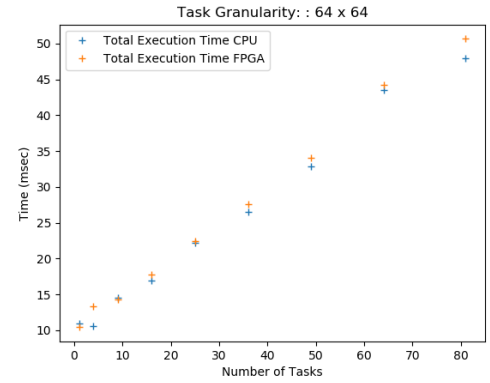
We have presented our ongoing effort to create a task programming systems targeting heterogeneous architectures including FPGA. Leveraging the StarPU programming system and its runtime along with HLS tools, the HEAVEN framework provides an easy and portable way for writing and running applications on top of heterogeneous architectures including FPGA.

The perspective of this work is numerous. First, as mentioned in Section IV-B1 we must make the communication with the FPGA completely transparent for the application developer. This is possible because the StarPU runtime already knows what the inputs and the outputs of each task are. The only thing the programmer will have to specify is the order in which inputs must be sent and outputs must be received. We also plan to extend the framework so that different types of hardware tasks could be used at the same time. This new feature will impact mainly the Conor library that must be aware about the type and the location of each hardware task in the FPGA devices. In this paper we have presented results for executions using only the cores of the host or only the hardware tasks in the FPGA. Because our goal is to leverage heterogeneous architectures, now that we have an operational framework, an immediate perspective is to execute applications combining CPU tasks, GPU tasks and FPGA tasks. We already started to study these executions and we believe that it may lead to interesting research directions regarding the scheduling heuristics of the StarPU runtime. As presented in Section IV-B2, the framework is currently using

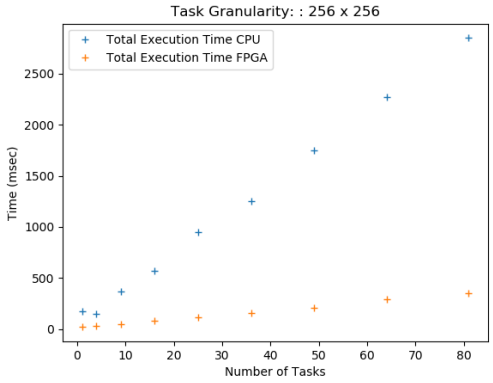


(a) 1 task (b) 4 tasks (c) 9 tasks (d) 16 tasks (e) 25 tasks (f) 36 tasks (g) 49 tasks (h) 64 tasks (i) 81 tasks

Fig. 9: The different inputs matrices size evaluated. Both the A and B matrices are split into N slices, with N from 1 to 9.



(a) Light-weight tasks.



(b) Heavy-weight tasks.

Fig. 10: Performance results for block sizes of 64x64 and 256x256 and both architectures (CPU, FPGA).

the FPGA BRAM memory to store inputs and outputs for the hardware tasks. Because the amount of available BRAM is usually in the order of megabytes, we are planning to investigate how the framework should be extended so that we can benefit from the biggest main memory coming along with the FPGA in boards such as the VC709 we used in our experiments. In this case, the application developer will be provided pointers into this main memory as the entry point for the implementation of hardware tasks. The StarPU runtime will then be responsible for managing the memory allocations in the board's main memory with the help of an additional hardware component in the FPGA. Finally, we want to study how dynamic reconfiguration could be integrated into the framework. For sure, this would be viable only if reconfiguration time is low compared to the task granularity. The StarPU scheduler will then need to take into account the reconfiguration time.

Acknowledgement

This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement davenir

References

[1] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 161–170.

[2] "Opencl specification," https://www.khronos.org/opencl/.

[3] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL: Revised OpenCL 1.* Newnes, 2012.

[4] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving programming model abstractions for reconfigurable computing," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 1, pp. 34–44, 2008.

[5] T. H. Foundation, "The hsa foundation specifications version 1.0," *http://www.hsafoundation.com/standards/*, 2012.

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

[7] A. Filgueras, E. Gil, D. Jiménez-González, C. Álvarez, X. Martorell, J. Langer, J. Noguera, and K. A. Vissers, "Ompss@zynq all-programmable soc ecosystem," in *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*, 2014, pp. 137–146.

[8] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and runtime support to blaze fpga accelerator deployment at datacenter scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: ACM, 2016, pp. 456–469.

[9] "Amazon ec2 f1," https://aws.amazon.com/fr/ec2/instance-types/f1/.

[10] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "Riffa 2.1: A reusable integration framework for fpga accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, p. 22, 2015.

[11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.