2014 IEEE International Symposium on Parallel and Distributed Processing with Applications

FPGA-based design using the FASTER toolchain: the case of STM Spear development board

F. Spada*, A. Scolari*, G.C. Durelli*, R. Cattaneo*, M.D. Santambrogio*, D. Sciuto*, D.N. Pnevmatikatos†, G.N. Gaydadjiev‡, O. Pell§, A. Brokalakis¶, W. Luk‖, D. Stroobandt**, D. Pau††

*Politecnico di Milano, Italy
†Foundation for Research & Technology - Hellas, Greece
‡Chalmers University of Technology, Sweden
§Maxeler Technologies, UK
¶Synelixis, Greece
‖Imperial College London, UK
**Ghent University, Belgium
††STMicroelectronics, Italy

***Abstract*—Even though FPGAs are becoming more and more popular as they are used in many different scenarios like communications and HPC, the steep learning curve needed to work with this technology is still the major limiting factor to their full success. Many works proposed to mitigate this problem by creating a companion of tools to support the designer during the development phase for this technology.**

The EU FASTER Project aims at realizing an integrated toolchain that assists the designer in the steps of the design flow that are necessary to port a given application onto an FPGA device. The novelty of the framework relies in the fact that the partial dynamic reconfiguration, which FPGA devices can exploit, is seen as a first class citizen throughout the whole design flow. This work reports a case study in which the FASTER toolchain has been used to port a raytracer application onto the STM Spear prototyping embedded platform. The paper discusses the steps done for the realization of the prototype and the results obtained on the target device. It finally reports some improvements that can be exploited to improve the performance of the hardware implementation that has been realized.

***Keywords*—partial reconfiguration**

I. Introduction

In an ever-changing world, there is an increasing demand for embedded, multi-core systems that are able to adapt to the surrounding environment or to meet new application demands. Adaptability by means of changing the software running on the processors is not always adequate: many applications require hardware acceleration due to strict requirements in terms of performance and energy efficiency. It is thus imperative that this hardware can adapt to application's changes and partial dynamic reconfiguration is the key enabler for such kinds of systems. In fact, this provides the flexibility needed to add or substitute functionalities (i.e. hardware modules) after the system has been manufactured and deployed. These hardware-supported adaptation mechanisms provide a cost-effective way to cope with changing environmental requirements, improvements in system features, changing protocols and data-coding standards, etc.

However, the designers still need the ability to properly take reconfiguration issues into account directly from the application specification down to the final system implementation. Moreover, the mechanisms required to verify and support this functionality at run-time are currently lacking. The FASTER (Facilitating Analysis and Synthesis Technologies for Effective Reconfiguration) project [3] aims at providing a complete methodology that will enable the designers to easily implement and verify applications on platforms with one or more general-purpose processors and multiple accelerators, which have been implemented on the top of the latest reconfigurable technology. Our goal is that, for the selected application domains, the envisioned toolchain will be able to reduce the design and verification time of complex reconfigurable systems by at least 20%, providing additional novel verification features that are not available in any existing tool flows. In terms of performance, for these application domains, the toolchain could be used to achieve the same performance with up to 50% smaller cost compared to programmable SoC-based approaches, or exceed the performance by up to a factor of 2x for a fixed power consumption envelope. Previous research and EU projects such as hArtes [4], Reflect [6], ACOTES [1], Andres [2], Morpheus and others focus on the necessary toolchain and address similar issues as FASTER but focus more on system-level or architectural aspects of reconfiguration. Moreover, they do not explicitly emphasize on the design and runtime aspects of partial and dynamic reconfiguration, or on choosing the best reconfiguration grain-size.

The FASTER toolchain will accept input that can be in HDL or C whose initial decomposition could be described with existing formalisms such as OpenMP [5] and derive the corresponding task graph. Using new graph-theoretic algorithms we will partition the specification in space and time. Then we will pursue a task-cluster definition of a system specification by detecting recurrent structures in the specification itself. These modules are candidates for reconfiguration, thus saving device resources and reconfiguration time. FASTER will support both region-based [11] and micro-reconfiguration, a technique to reconfigure very small parts of the device [7]. The ability to

978-1-4799-4293-0/14 $31.00 © 2014 IEEE
DOI 10.1109/ISPA.2014.26

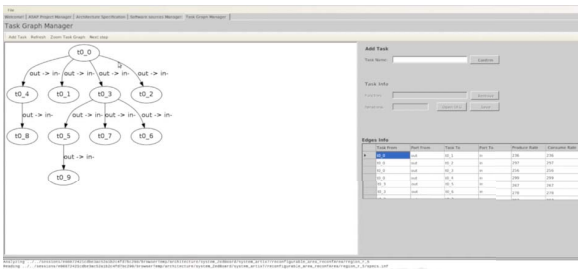

Fig. 1. Screenshot of the GUI application, with DFG exploration.

handle both types of reconfiguration opens up a new range of application possibilities for run-time reconfiguration, as a much broader time frame for the reconfiguration itself is available and the underlying concepts are different for both types of reconfiguration. FASTER will also develop techniques for verifying static and dynamic aspects of a reconfigurable design at compile time using symbolic simulation - a powerful verification approach for static designs, and extending it to support both static and dynamic aspects of a reconfigurable design. We will also explore techniques for verifying selected static and dynamic aspects of a reconfigurable design at run time with a small impact on speed, area and power consumption. Finally, FASTER will provide a powerful runtime system that will be able to run on multiple reconfigurable platforms and manage the various aspects of parallelism and adaptivity with the least overhead.

Within this paper we are focusing our attention to present the work done by FASTER on global illumination and image analysis algorithms. In future graphics applications (games, visualization, etc.), it will be important to achieve photorealistic rendering in a coherent manner, in order to greatly improve picture quality with an ever-increasing scene complexity, with support for real reflection, soft shadows, area light source, indirect illumination, etc. This is a computationally intensive problem, addressed by the increasing interest in real-time global illumination approaches. Within FASTER, STMicroelectronics (STM) is developing a C global illumination pipeline. This system should be flexible enough to help accelerating different algorithms based on ray casting (ray tracing, path tracing, Monte Carlo ray tracing, etc.).

II. Toolchain for HW/SW Codesign

The methodology developed in FASTER is accompanied by an effective step-by-step Graphical User Interface (GUI) that helps the designer in completing each step of the design [13]. The designer is responsible of providing the task graph of the application, which is encoded in a XML format. The GUI, shown in Fig. 1, provides a simple frontend to manually generate and feed information to the XML file. Each task is associated with a function in the C source files that are also included in the XML file; also in this case the GUI provides a simple way to load application files and modify them if needed. This binding between the tasks and functions is the one that permits the analysis of the C code and the generation of the function Data Flow Graph (DFG) that can be used for further analysis and optimizations. The GUI, along with the XML file, provides a simple integration with third party tools. The exploration phase starts from the information

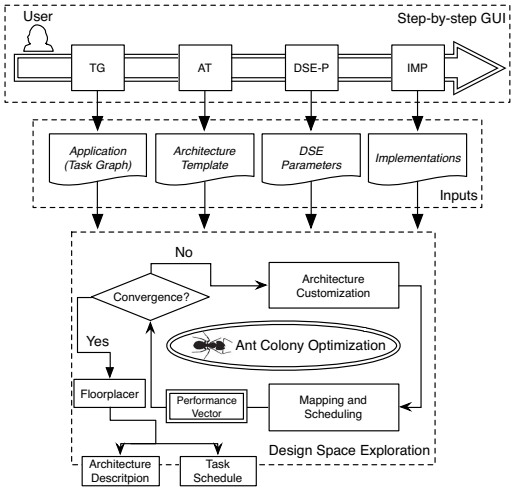

Fig. 2. Overview of the Design Space Exploration.

included in the XML, which are generated in several ways: by manually inserting the data to the XML with a text editor, by using the GUI, by profiling or by other FASTER related tools. The exploration phase represented in Figure 1 starts from reading 4 input parameters from the XML files, which are the application task graph, the architecture description, the implementations associated with each task and the Design Space Exploration parameters to be used. After analyzing the input task graph, the toolchain assembles the information into an intermediate representation format that will be further manipulated by the mapper [9] and the scheduler [10]. After this initial step, the task graph is manipulated by the mapper, which will tightly interact with the scheduler to generate a mapping characterized by some performance metrics like execution time (an information computed by the scheduler) and the amount of utilized resources, which is computed from the board specifications and implementation details. The mapping phase is implemented as an evolutionary algorithm, namely Ant Colony Optimization, which uses the information fed back by the scheduler to evolve towards increasingly better mapping solutions.

III. STM Spear Architecture

The STM SPEAR prototyping embedded platform is a development board that is intended to be used for feasibility studies, in particular to evaluate software systems developed for embedded platforms and how they could benefit from external hardware acceleration. As such, its primary target are R&D projects that aim to customize the architecture starting from an existing but flexible one. Therefore, this platform is not designed for production environments.

The Spear board has an ARM dual-core Cortex-A9 with two level of caching and 256MB DDR3 as Central Memory, which communicates with the ARM processors through an AXI 64 bits bus. The Spear is equipped with several interfaces that allow external peripherals to be connected to the board through standard interfaces like Ethernet, USB or UART. By means of the auxiliary EXPansion Interface (EXPI) present on the board, it is possible to plug in dedicated boards to expand the SOC with customizable external hardware. On this

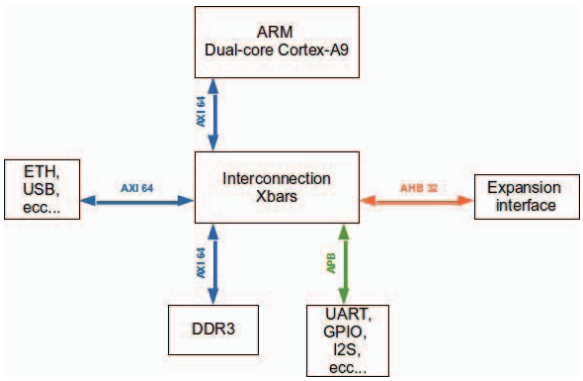

Fig. 3. Architecture of the STM Spear Development board.

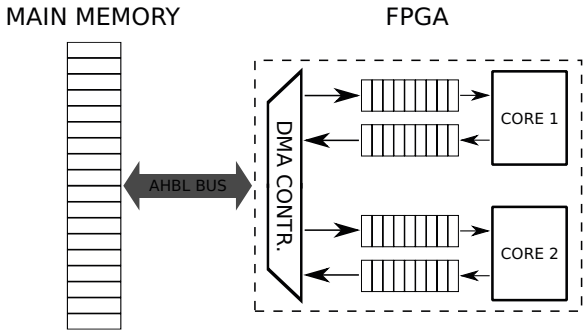

Fig. 4. The figure shows the main buses involved in data transmission between main memory and computing cores.

interconnection, the communication is driven by the AHBLite protocol. The choice of the EXPI interface and of the AHBlite protocol lets designers attach custom acceleration boards and interface them quickly, thanks to the simplicity of the AHBLite protocol. Overall, the interconnection scheme of the system is summarized in Fig. 3.

The performances of AHB bus are lower than the AXI one: for example, the AXI protocol defines double data channel and the read and write can be done simultaneously, while the AHB has only one data channel and the read and write cannot be executed in parallel. Moreover, the internal interconnection crossbars, with conversion from AHB bus to AXI bus, constitute a further physical limits that impacts on the performance when considering the overall transfer time, because it increases the latency of the transmission and can create bottleneck effects when multiple transmissions from main memory and the external components are happening simultaneously.

Virtex5 Daughter Board

The FPGA board ships an auxiliary bidirectional interface (EXPI) to connect external peripherals. Since the Spear board was designed for functional prototyping, the EXPI interface provides a single AHBLite bus with both master and slave interfaces on the board side, thus requiring external components to implement these same interfaces.

The AHBLite protocol uses a single data channel for communications, so that it is impossible to receive and send data at the same time, e.g., to have a continuous stream between memory and computing cores. This can be a major limitation in exploiting hardware acceleration, also considering the low frequency of the channel, which runs at 166 MHz, and the maximum transfer length limited to 1KB. Fig. 4 shows the architecture involved in data transfer from DMA to AHB bus.

IV. Raytracer Application

The raytracing algorithm has been widely studied over the recent years due to its great interest in computer graphics; this technique can be, indeed, used in rendering of images for movies or computer games. Different implementations of the algorithm have been thus proposed over the recent years. Most of the proposed approaches focus on implementing this algorithm on GPUs due to the possibility of programming them with frameworks such as CUDA [12] and OpenCL [16]. For example, [15] proposes a technique to implement a parallel version of the algorithm on GPUs, overcoming the limitation posed by the recursive structures of the algorithm which cannot be implemented in GPUs. Other works instead focused in creating customized computing platforms using FPGAs as prototyping devices [8, 14, 17].

This section introduces the version algorithm used in the work, its potential limitations for hardware accelerator and the solutions adopted to solve them.

General Characteristics

The raytracing algorithm provided by STM starts from a description of the scene as a composition of 2D/3D geometric primitives. The basic primitives that are supported by the algorithm are the following: triangles, spheres, cylinders, cones, toruses, and polygons. Each of these primitives is described by a set of geometric properties such as, for example, the position in the scene, the height of the primitive or the rays of the circles composing the primitive. The algorithm then performs the following steps:

1) the scene is divided in blocks, called voxels, and the number of these voxels is one of the contributors to determine the complexity of the algorithm; the more the voxels there are, the more intersections between rays and primitives have to be computed;
2) the algorithm generates a certain amount of rays from the current rendering point of the image and it computes the set of voxels traversed for each of these rays;
3) it then iterates all over these voxels and computes the intersection between the primitives in the voxel and the current ray;
4) assuming all the objects are opaque, the nearest intersection, if any, is considered and the algorithm computes the reflection and refraction of the light ray on the surface of the object;
5) the rays generated by this physic simulation continue to be propagated into the image until a maximum number of intersection (an input parameter of the application) is reached or no intersection is found at all;

Analyzing the application, we found two major roadblocks that may prevent us from efficiently porting the application into

hardware. First, the memory accesses to the objects stored in the main memory do not follow a regular pattern, but instead they depend on the path followed by the light ray in the scene and its subsequent reflections which cannot be predicted in advance. Accessing random memory locations may cause slowdowns in the hardware implementation; the hardware cores can be efficiently exploited when data is accessed with a fixed pattern, since data transfers between the memory and the accelerator can be carried out through a Direct Memory Access (DMA) mechanism. This exploits the principles of locality by moving an entire block of data.

Second, one of the primitives, the polygon, is characterized by a variable number of parameters, such as the number of vertexes. For its hardware implementation we need to determine in advance the amount of vertexes supported. On the other hand, each computation performed by this accelerator will require an amount of time which will be proportional to the number of vertexes.

To approach the first problem the flow of the application has to be restructured as described in the following paragraph, while for the second problem we left the computation of the intersections with polygon primitives to be computed only in software so that we do not have to constrain the core.

Data Access Pattern: To achieve the best performance in terms of memory accesses, the access pattern of the raytracing algorithm must be restructured. The original code computed the least amount of intersections needed to determine if a ray intersects any objects in the scene. It computes all the intersections until one intersection is found. In this case, it stops searching in the next voxels. This behavior has to be changed by precomputing all the intersections that has to be computed along the path of one light ray and we organized them in queues. These queues are then sent to the hardware cores and this requires only a linear memory access, since they can be moved from main memory to cores by using the DMA. After the intersections are computed by the hardware cores, the results are collected by the SW that merges the results and determines which is the nearest intersection found. Note that, since the intersections are computed in order, there is no need to perform any computation to determine the nearest one. Indeed, only the the first intersection found has to be considered, while the following ones can be safely discarded.

V. Implementation

This section reports the details of the implementation of a prototype of the Raytracer application for the STM Spear Development board. The first section shows the details of the DMA module that has been realized to interface the FPGA with the AHB bus, then the details about the realized HW cores by means of High Level Synthesis (HLS) are reported.

DMA Module and driver

No DMA controller IP was provided with the FPGA. Hence, a custom DMA controller has been designed from scratch according to the AHBLite protocol. This controller implements one master and one slave interface. Because of the limitations of the protocol, it is fundamental to add a queuing mechanism to keep the inputs and the outputs of the computing cores. With this mechanism, the software can send multiple inputs to a core, that computes the results and stores them in an exit queue. During the core computation, its data (inputs and outputs) are managed by the queues, so that the bus is available to support other transfers and the software must not wait for the results but can perform other tasks, like computation or data transfer management.

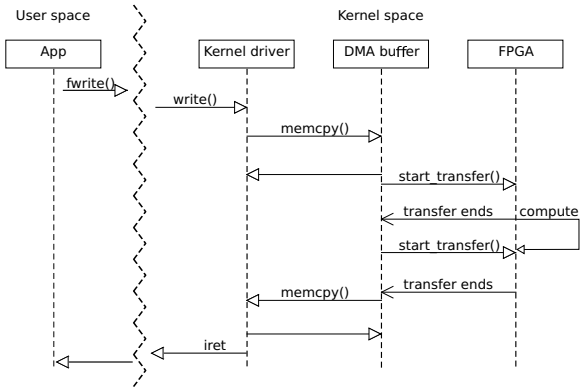

Fig. 5. Sequence of calls to send data to the cores by writing the device file: after the context switch, the control is passed to the driver, which copies data from the user buffer to the DMA buffer and starts the transfer. Then the cores on the FPGA compute the result and the driver copies than back to the application.

The master interface, whose registers are chosen by the designer, is mapped into main memory starting from address 0xD0000000. This allows designers to write a Linux driver in order to manage the communication, for example by representing the FPGA device as a file inside the /dev/ directory. Thus, the driver can explicitly handle any operation performed on this file, and translate user-space calls with a well-known syntax (open, read, write, etc.) to bus communications.

Moreover, the Linux file interface allows designers to give complete control of the device to user-space by the *mmap()* system call. Through this call, the device driver can map the DMA interface into the virtual memory of the application that performed the call. Thus, this application can take the control of the communication by directly handling the DMA interface, handling its specific syntax. This optimization slightly increases the complexity of the application, but potentially increases the overall performance as it avoids expensive context-switches from user-space to kernel-space (as from Fig. 6), which often require copying data from user-space to kernel buffers. Several aspects must be taken into account to achieve user-space control (virtual memory permissions, memory-cache coherency, etc.), but this is a worthwhile optimization, as visible when comparing the initial results of Fig. 10 with Fig. 11.

HW Cores

The hardware cores have been realized using Vivado HLS, which permits to generate an accelerator, along with its interfaces, that can be directly integrated in the development board architecture. However, the C code that can be synthesized using HLS presents some restrictions. As an example, pointers or any pointer logic cannot be generally used. Since pointers are generally used in function interfaces, in order to realize the accelerators required in this work, C functions implementing the

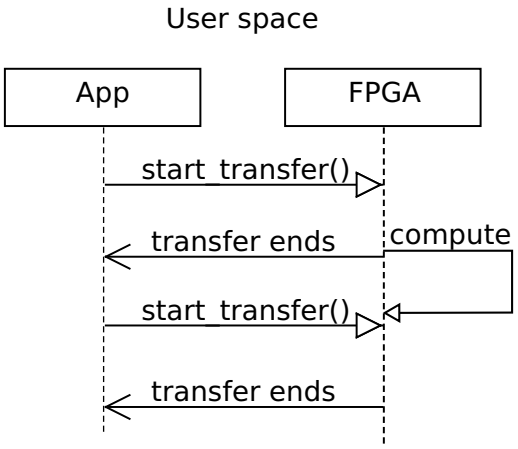

Fig. 6. Sequence of calls in case of user-space DMA handling, provided that the user data structure are memory-coherent. No context switch happens, and no data copying happens.

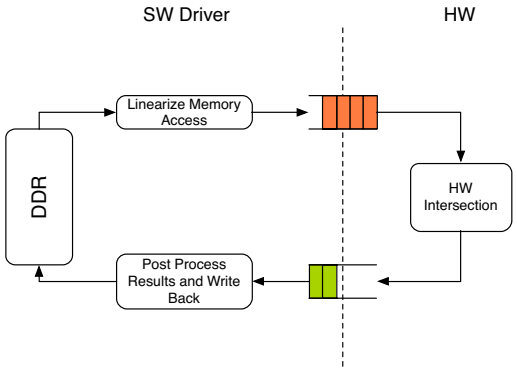

Fig. 7. Template of HW core generated with High Level Synthesis for compute intersection functions.

intersections with each of the primitives have been rewritten in order to remove the input pointers and substitute them with explicit variables. The interfaces of the cores have been then instructed to read data from an input FIFO queue and feed the variables with the input data. On the software side, a similar change has been made. Before executing a hardware function, the raytracer needs to dereference the involved pointers and organize them in a proper way in the main memory (in the same order which is expected to be read from the input FIFO). Once completed, the core can start the execution and fetch a region of memory to process. The core generated through HLS have been synthesized to meet the reference clock of 100 MHz, which is the one used in the programmable logic. The core ported to HW are the intersections functions that intersect a light ray with a given primitive. In particular the functions implemented in HW compute the intersection with the following primitives: cones, cylinders, spheres, and triangles. The structure of the the HW cores alongside with the SW interface/drivers can be summarized by Figure 7, while Table I reports the results of the HLS on the three functions when targeting a 100MHz reference clock.

TABLE I. SUMMARY OF HLS RESULTS OF THE HW CORES. ALL THE CORES MEET THE TARGET FREQUENCY OF 100MHZ.

Core	LUT	FF	DSP	BRAM
Sphere	7051	4763	15	2
Ring	5466	3372	18	2
Triangle	6168	3432	32	4

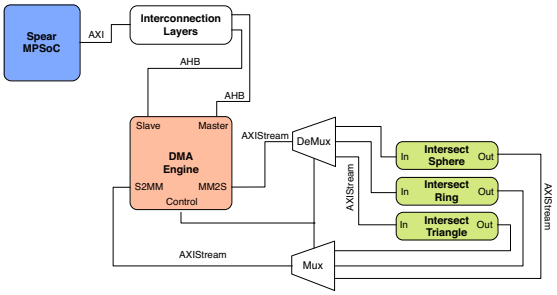

Fig. 8. Architecture used to port the Raytracer application on the STM Spear Evaluation Board.

HW prototype application

As from the previous section, a preliminary evaluation required the synthesis, by means of HLS solutions, of the cores that compute the intersections for spheres, rings and triangles primitives; those cores were deployed on the Spear platform with the described DMA interface. The resulting architecture is represented in Fig. 8. To evaluate the performance of the platform, no change was made to the application to optimize the DMA transfers. Therefore, the execution model of the application is unchanged, the software computationbeing replaced with the hardware counterpart. The application, thus, for each ray writes the input data to the DMA buffer, triggering the transfer to the needed FPGA core, and then reads the output data. Hence, the sequence of operations is the one shown in Fig. 5, with memory copies of inputs and outputs that are on the critical path.

GUI prototype

In order to test the hardware-ported application, show the hardware functionalities and check the correctness of the hardware rendering, a demo application was developed that shows a GUI displaying the images rendered on the Spear

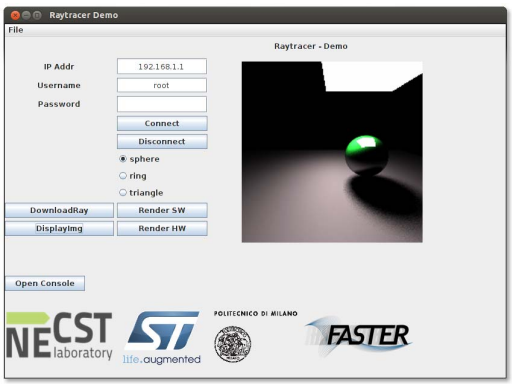

Fig. 9. The GUI prototype application to test the hardware port.

board, as visible in Fig. 9.
In particular, the main steps to use the application are:

1) the application connects to the Spear platform through an SSH interface
2) the application downloads the cross compiled binary application and the related configuration files on the Spear file system
3) the user selects on the GUI which primitive to render (sphere, ring or triangle) and how to render it (in hardware or in software)
4) the application runs the raytracing algorithm based on the choices made by the user and waits for the completion
5) the application loads the result image from the board to the host PC, automatically showing it on the screen

VI. Results

This section reports the performance analysis on DMA data transfer for the STM Spear Board used to implement the RayTracer algorithm and the details on the HW Cores realized using High Level Synthesis tools along with the results of executing the HW version of the raytracer.

DMA Performance Evaluation

In order to analyze overall performance of the system, some metrics should considered:

- CPU Speed
- AHB Bus Speed
- Bandwidth between main memory and FPGA

Since the frequency of CPU is 600 Mhz and the one of AHB bus is 166 Mhz, the calculation of bandwidth between central memory and FPGA requires a test bench. The way implemented to transmit data from memory to external board is through DMA core, and the test consists in an application that does a copy of data from one location of memory to another. These memory copies are performed with the use of the DMA core so data go to FPGA and then come back to memory, measuring the t time required to transmit d bytes and obtaining a bandwidth b=d/t. From the point of view of the operating system the DMA is an external peripheral and an application need a presence of driver in order to use this peripheral. Currently there are two ways of implementing the driver, the first one is let the driver totally in kernel space and export some interfaces into user space, for example with the use of device interfaces. The second one is move the logic as much as possible into user space and let in kernel space only the strict necessary to build infrastructure of communication. We calculated bandwidth with both solutions, the following figures represent the results obtained with this architecture. The blue line of Fig. 10 indicates the bandwidth with the first solution above (driver totally in kernel space), in this case the data must be copied from user buffer into kernel buffer and this impacts enormously on bandwidth. The green line of Fig. 11 indicates the second solution where the copies are eliminated and there is not context switch, the bandwidth here is the maximum theoretically achievable with this connections between SOC and FPGA. This two graphs should be compared

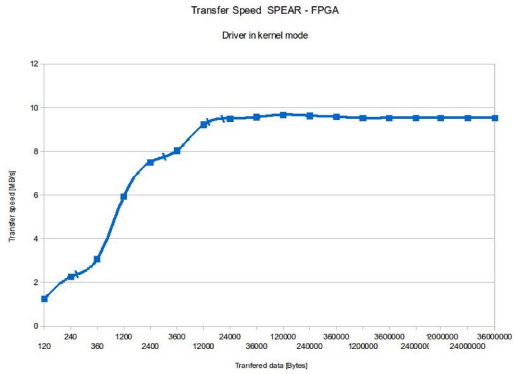

Fig. 10. Transfer speed of data copy between memory and hardware cores with driver.

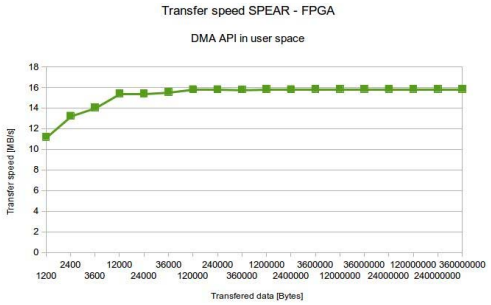

Fig. 11. Transfer speed of data copy between memory and hardware cores with user-space memory mapping.

with the same test bench executed purely in software by the ARM processor, looking at the orange line in Fig. 12.

Fig. 12 shows the performance of the memory hierarchy, which consists in 2 levels of cache connected to a main memory via an AXI 64 bus at 533MHz. The test shows that the memory bandwidth stabilizes around 28 MB/s, for large amounts of data moved. This measure serves as a baseline for following measures, to foresee the possibility of any performance bottleneck that might occurr when using the FPGA acceleration.

Fig. 10, instead, shows the performance obtained with the usual Linux driver interface, that consists in a sequence of write() and read() calls. These calls, according to Fig. 5, need

Fig. 12. Transfer speed of data copy inside main memory.

Fig. 13. Execution time of the software and hardware implementations of the Raytracing algorithm.

heavy context switch and data copy operations between user-space and kernel buffers that impact the final performance negatively. This, in our opinion, explains the performance loss with respect to Fig. 11. Nonetheless, even in the best case of Fig. 11 the performance is constantly below that achieved in Fig. 12; this definitely indicates the bus transfer as an unavoidable bottleneck when using hardware acceleration.

HW Raytracer Implementation

Fig. 13 shows the execution time of the software and hardware implementations, where the software one has unitary execution time. We immediately notice that the overall performance decreases, the hardware-application being effectively slowed-down by a 1.4 factor. This slowdown is due to the performance of the communication along the AHBLite channel: since only few data are sent on each call, the DMA transfer is done in small bursts with low throughput (as from Fig. 10) and synchronously with respect to the application execution, thus maximizing the impact of the communication and limiting the final performance.

Raytracer design with FASTER Toolchain

The FASTER toolchain has been used in the analysis and implementation of the application. In particular the mapper, scheduler and floorplan tools have been used to help in the realization of the HW architecture. The application task graph for relevant functions has been created manually and the SW and HW implementations of the functions computing the intersections have been profiled with tools or using profiling information provided by Xilinx Vivado HLS, which was used to create VHDL starting from original C code. At this point the task graph of the original application provided by STM with the related profiling information has been encoded in the XML format using the GUI developed in the FASTER project and the mapper and scheduler tool determined that with the original implementation the best architecture implementing the application is the one using only SW functions. This can be easily explain by the fact that the original code computes one intersection at the time and in this case the HW cores experience a slow down due to the continuous overhead of DMA transfer.

VII. Future work

This section identifies the main changes the hardware/software components need to better leverage the capabilities of the FPGA chip employed. To foresee a feasible scenario, we devised these changes with respect to the current features of the platform. However, also modifications to the platform are required, in order to alleviate the bottleneck effects we found in previous sections, which hinder the attempts to obtain a real acceleration.

Platform modifications

The results show the impact of the communication channel, that heavily affects the transmission and also the software design. In our opinion, the main limitation is the AHBLite protocol, which imposes a half-duplex communication pattern and provides a single channel. The presence of only one channel, for example, prevents the application from sending data to multiple accelerators, thus impeding full parallelism among multiple cores. Focusing on the single computational core, we envision the need of a full-duplex communication pattern, to allow a continuous stream of data that would increase the usage of each core (that now pauses waiting for new inputs) and would remove the need of long input/output queues, thus with area saving. A solution to some of these issues would be the adoption of the AHB full protocol, which is a ”superset“ of the current AHBLite protocol and, thus, should not require excessive changes to the existing design. But this protocol, albeit contemplating the presence of multiple channels, does not allow a full-duplex communication, thus with a potentially low gain in terms of exploitable parallelism and of benefits. Therefore, we finally envision the adoption of the AXI protocol, which, among other features, allows a high performance full-duplex communication. The adoption of this well-known protocol (for which many IPs already exist, easing the development) would introduce slight modifications to the architecture of the platform, but would definitely remove the main bottleneck found.

Hardware modifications

The current implementation of the transmission capabilities requires the software to specify to which core the data are directed by writing a register of the DMA memory interface. In case it is not possible to send large amounts of data to a single core, many small data amount are sent to various cores, re-programming the DMA interface on every transfer. This splits the data stream into multiple short bursts, thus penalizing the transfer rate (in Fig. 10, the rate increases with the burst size). To avoid this bottleneck, a component should be placed in front of the core that routes the data to the proper core, for example based on the value at the head of the queue.

Another optimization involves the output handling: in the current implementation, all the results are gathered back to the software component, which inspects them and discovers whether an intersection is present. Hence, output bundles containing no intersection, and thus useless for the following of the computation, are sent over the bus, wasting bandwidth. A component that performs this inspection directly on the FPGA would, on the contrary, save this bandwidth. Its main drawback is the increasing complexity of the transmission for output

collection: since it is not possible to predict how many outputs correspond to a real intersection, the amount of data to be sent back would become unpredictable, thus preventing software from specifying its measure to trigger the DMA transfer. A possible solution to this issue would be to expose the amount of the output data to software, for example through the memory mapped registers.

Software modifications

With the previous hardware modifications in mind, the software component must be tailored in order to leverage the new, proposed hardware capabilities.

First of all, the computation part should be separated from the transmission, making the two parts working asynchronously in order to send over the bus large amount of data without stopping the software computation. To this aim, we envision a solution with two threads, exploiting the dual core CPU available. The first thread continuously computes input data and serializes them inside a single buffer, from which they are sent to the accelerators for the computation of intersections. This thread also receives the results inside another buffer, from which it computes the color contribution of each intersection on the final image. The second thread, instead, is in charge of managing the communication, sending data to the core when the input buffer is full enough and gathering results from the output queue when this is full.

These modifications heavily impacts on the raytracing algorithm, changing its design. Indeed, while the current implementation generates reflected rays starting from the original ones in a depth-first manner, the need of producing input data for the hardware cores without waiting for intersections requires a breadth-first approach in the rays generation. By shooting long sequences of independent rays, it is possible to rapidly fill the input buffer, allowing the transmission of many data in a single burst while the software continues to compute inputs. But, allowing long input sequences to be buffered for transmission requires the two threads to be carefully coordinated, to avoid, e.g., buffer overflows or not-yet-used output data to be overwritten by newer outputs, loosing data.

VIII. Conclusion

This paper reports the work done in the EU FASTER Project; in particular it reports a case study involving the use of the toolchain developed during the project in order to assist the designer in port an application on a HW device. The case study discussed in this work use the raytracer algorithm as target application and the STM Spear Evaluation Board as target device. The raytracer algorithm is a state of the art solution for high quality scene rendering which exploits physics simulation of ray lights and their reflections on object surfaces to obtain realistic images. The STM SPEAR evaluation board is a development board that is intended to be used for rapid prototyping, in particular to evaluate software systems developed for embedded platforms and how they could benefit from external hardware acceleration. To this end the board can be expanded with an external daughter board which in our case features a Xilinx Virtex5 FPGA that we used to prototype and investigate a possible HW implementation of the raytracer application.

The paper reports the steps done for the realization of the prototype and the results obtained on the target device. Even though the results are not yet sound we believe that, with few changes in both HW and SW stack as discussed in Section VII, the solution proposed in this work might be promising.

Acknowledgments

This work was partially funded by the European Commission in the context of the FP7 FASTER project (#287804).

References

[1] http://www.hitech-projects.com/euprojects/ACOTES/, [Online; accessed March 2012].
[2] http://andres.offis.de/, [Online; accessed March 2012].
[3] http://www.fp7-faster.eu/, [Online; accessed May 2012].
[4] http://hartes.org/hArtes/, [Online; accessed March 2012].
[5] http://www.openmp.org/, [Online; accessed May 2012].
[6] http://www.reflect-project.eu/, [Online; accessed March 2012].
[7] K. Bruneel, "Efficient Circuit Specialization for Dynamic Reconfiguration of FPGAs," PhD thesis, Ghent University, 2011.
[8] C. B. Cameron, "Using FPGAs to supplement ray-tracing computations on the Cray XD-1," in *Proc. of the DoD High Performance Computing Modernization Program Users Group Conference*, Jun. 2007, pp. 359–363.
[9] R. Cattaneo *et al.*, "Smash: A heuristic methodology for designing partially reconfigurable mpsocs," in *Rapid System Prototyping (RSP), 2013 International Symposium on*, Oct 2013, pp. 102–108.
[10] R. Cattaneo *et al.*, "Para-sched: a reconfiguration-aware scheduler for reconfigurable architectures," in *Reconfigurable Architecture Workshop (RAW) 2014*, May 2014, pp. 102–108.
[11] P. Lysaght *et al.*, "Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs (Invited Paper)," in *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*, August 2006, pp. 1–6.
[12] J. Nickolls *et al.*, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.
[13] C. Pilato *et al.*, "A2b: An integrated framework for designing heterogeneous and reconfigurable systems," in *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, June 2013, pp. 198–205.
[14] J. Schmittler *et al.*, "Realtime ray tracing of dynamic scenes on an FPGA chip," in *Proc. of the Conf. on Graphics hardware*, Jul. 2004, pp. 95–106.
[15] A. Segovia *et al.*, "Iterative layer-based raytracing on cuda," in *Proc. of the Int.l Performance Computing and Communications Conference (IPCCC)*, Dec. 2009, pp. 248–255.
[16] J. E. Stone *et al.*, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.
[17] S. Woop *et al.*, "Rpu: a programmable ray processing unit for realtime ray tracing," in *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3, 2005, pp. 434–444.