2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Adaptive Raytracing Implementation using Partial Dynamic Reconfiguration

Gianluca Durelli[1], Fabrizio Spada[1], Riccardo Cattaneo[1],
Christian Pilato[2], Danilo Pau[3], Marco D. Santambrogio[1]

[1]Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy,
fabrizio.spada@mail.polimi.it, {durelli,rcattaneo,santambr}@elet.polimi.it

[2]Columbia University, Department of Computer Science, New York, NY, USA,
pilato@cs.columbia.edu

[3]ST Microelectronics, Milano, Italy,
danilo.pau@st.com

***Abstract*—The continuous strive for improvements in visual realism is progressively increasing the complexity of algorithms for simulating light physics to produce very realistic scenes. As a result, they are becoming more and more suitable for hardware acceleration, even if they introduce new challenges due to the high requirements in terms of resources. In this paper we propose a hardware implementation of the raytracing algorithm, which is a method for rendering 3D scenes. We exploit partial dynamic reconfiguration to adapt the hardware to the specific part of the image under analysis. This allows us to obtain up to 30% better performance with respect to the software baseline implementation on the AVNET ZedBoard platform.**

***Index Terms*—Field Programmable Gate Arrays, Embedded software, Image processing**

I. Introduction

The problem of performing 3D rendering in an effective way is becoming more and more important due to the continuous strive for realistic images, as for instance in both movies and video games industries over the recent years. The complexity of simulating light behavior makes these algorithms very computational intensive; for this reason, their implementations are generally carried out exploiting Graphical Processing Units (GPUs) or high-end General Purpose Processors. Harwdare acceleration has been also investigated, especially for Field-Programmable Gate Arrays (FPGAs), at the penality of a huge amount of resources needed to implement the application. On the other hand, Partial Dynamic Reconfiguration (PDR) [1] is a promising technique to cope with limited resources, but it requires the designer to have a high expertise to create the final system.

Raytracing is one of these algorithms for 3D scenes rendering which is able to simulate the physics of a light ray and thus can generate realistic results. The classical implementation of the raytracing algorithm consists in defining a rendering point in a 3D scene and shooting light rays from that point, simulating their reflections and refractions when these rays intersect objects in the scene. The objects are described as a composition of geometric primitives (2D or 3D) such as triangles, polygons, spheres, cones or other shapes. It is clear that the computational complexity of a rendering scene is proportional to the number and the nature of these primitives, along with their positions in the scene itself.

Recently progress in High Level Synthesis (HLS) improved the programmability of FPGAs, extending the range of designers able to exploit such devices as accelerators for specific application tasks. As an example of these HLS tools, we can mention both academic ones (e.g. GAUT [2], LegUp [3], DWARV [4], bambu [5]) and commercial ones (e.g. Xilinx's Vivado HLS [6], Cadence's C-to-Silicon [7], Forte's Cynthesizer [8]).

This paper proposes an implementation of the raytracing algorithm for a heterogeneous platform composed of an ARM processor and an FPGA. We adopt Xilinx Vivado HLS to realize the hardware cores and the corresponding interfaces for the integration with the rest of the system, based on the standard AXI bus. The proposed implementation exploits the partial dynamic reconfiguration capability of such device to adapt at run-time the hardware configuration of the board to speed up the computation in specific parts of the input scene, based on the nature of the primitives. A simple policy is proposed to adapt the hardware to a single block under analysis. This adaptation is done by monitoring, at run-time, which the most used primitives at a certain moment and then by configuring the hardware accordingly.

The remainder of the paper presents the related work in the area (Section II), the general description of the algorithm and the modification we apported (Section III), the implementation details about the hardware architecture (Section IV), the description of the adaptation policy (Section V), a discussion on the obtained results (Section VI), and finally presents the conclusions introducing possible future extensions of the work (Section VII).

II. Related Work

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

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.31

IEEE computer society

movies or computer games. The optimization of the algorithm execution time allows the designers to have a shorter time to market for the movies or the possibility to increase the graphical quality of a game without decreasing the number of frames per second rendered by the hardware device. 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 [9] and OpenCL [10]. For example, [11] 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 researches in the field of ray tracing focused in creating customized computing platforms using FPGAs as prototyping devices. These works present solutions that deploy multiple processing pipelines to increase the processing capabilities of the devices [12]–[14]. However, these works feature a static hardware, without the possibility of adaptation with respect to specific parts of the scene. For example, if in one part of the scene we have only one type of geometric primitives, it would be useful to have multiple hardware cores to perform a parallel computation of the intersections generated in that region. Furthermore, some of the proposed approaches [15] relies on the fact the the input image is composed only of triangles. Even if this is a common solution, it would be useful to support different kinds of intersection primitives in order to obtain a more realistic scene description which will lead in turn to a more realistic rendered image.

Finally, the datastructures used to represent the scenes have been also investigated. The literature proposes works focusing on Grids [16], Bounding Volume Hierarchies (BHV) [17] and KD-Trees [18]. However, there is not an optimal data structure to represent the input scene, but this depends on the computing platform adopted for the implementation of the algorithm.

III. Raytracer Algorithm

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

A. General Characteristics

The raytracing algorithm adopted in this work starts from a description of the scene as a composition of geometric 2D/3D primitives. In particular, 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) 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 generated 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.

On one hand, we approach the first problem by restructuring the flow of the application as described in the following paragraph. On the other hand, we limited the computation of the intersections with polygon primitives to be computed only in software so that we do not have to constraint the core.

B. Data access pattern

To achieve the best performance in terms of memory accesses, we restructured the access pattern of the raytracing algorithm. 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. We changed this behavior 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 part and this requires only a linear memory access, since it can be moved from main memory to cores by using the DMA. After the intersections are computed by the hardware core, the results are then collected by the raytracer 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

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

Core	LUT	FF	DSP	BRAM
Cone	11702	7471	33	4
Cylinder	11021	7041	33	4
Sphere	7051	4763	15	2
Triangle	6168	3432	32	4

first one that has been found has to be considered, while the following can be safely discarded.

IV. HARDWARE IMPLEMENTATION

The raytracing algorithm has been implemented on the AVNET ZedBoard, that is a development board for the Xilinx Zynq-7000 All Programmable SoC (AP SoC); this SoC features a ARM Cortex-A9 dual-core processor and a reconfigurable logic fabric. The computational cores have been realized using Xilinx Vivado HLS tool starting from the available C implementation and they have been integrated using Xilinx Vivado Design Suite [6]. The raytracer code which runs natively on the ARM processor has been adapted to support hardware execution and to exploit the partial dynamic reconfiguration of the device.

A. Realization of hardware cores

The hardware cores have been realized through 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 used as they are since they require sophisticated mechanism to be executed in hardware, whenever it is possible. Since pointers are generally used in function interfaces, in order to realize the accelerators required in this work, we rewrote the C functions implementing the intersections with each of the primitives 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 and feed the variables with the input data. On the software side, a similar modification 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 results of HLS process applied to the intersection functions of the raytracer are reported in Table I. All the cores meet the target frequency of 100 MHz, which is the one used for the programmable logic, and they occupy around the 20% of the FPGA resources.

B. Target architectures

All the architectures used for this work features the ARM processor that can exploit application specific accelerators to speed up some part of the application that it is running. In order to demonstrate how partial dynamic reconfiguration and adaptiveness can help while performing raytracing on an embedded heterogeneous system, we realized two different architectures reported in Fig. 1. The first one (Fig. 1-A) consists in a pure architecture where only the ARM processor (booting Linux) is used to perform the raytracing algorithm. The second architecture (Figure 1-B) consists in a heterogeneous architecture composed of the ARM processor and of three reconfigurable regions. The main memory is shared between the ARM processor and the accelerators and data can be moved by means of DMA cores interfacing each one of the accelerators with the main memory. The connection between the reconfigurable modules and the ARM is performed through AXI bus using Xilinx AXI DMA modules with AXI Stream protocol. The DMA core is not part of the reconfigurable region and the DMA interface is the one which is kept costant across all the reconfigurable cores. This allows reconfiguration and also the possibility to implement the same core in all of the reconfigurable modules without the need to design a superset of the interfaces in order to keep it constant for each single module. Furthermore since it is important to have a fast data transfer rate between the ARM memory and HW cores the DMA are interfaced with the ARM through the High Speed AXI interfaces the Xilinx Zynq subsystem has. We decided to have the maximum flexibility out of our design, so each of the hardware accelerators may be configured at run-time onto any regions. This implies that each region must be able to implement the biggest of the cores and so only three reconfigurable modules can fit into our architecture. In fact, considering the data in Table I, one can argue that up to four cores can fit into the programmable logic the design. However in order to implement reconfigurable regions we need to partition the design limiting the amount of resources available. It is not possible with the tested architecture to implement all the four module as reconfigurable region due to routing errors. Partial dynamic reconfiguration may affect only the functions implemented in the accelerators while the interface with DMA cores will remain the same for each of the implemented cores.

V. DYNAMIC ADAPTATION

The concept of run-time adaptiveness proposed in the work consists in the possibility of varying at run-time the core on which the intersection with a particular primitive is computed. Each of the potential intersection cores may be implemented in software or in any of the available reconfigurable modules.

As a first idea, the designer can control the mapping of the intersection primitives, whether they are computed in software or in hardware, when a new rendering is requested on the basis of the amount and types of primitives in the scene. As an example, if the requested scene does not contain any sphere it is useless to have a hardware core to compute the intersection with a sphere while it may be useful to substitute it with a core computing the intersection for one of the other primitives present in the scene, especially the most used ones. However, to maximize the performance, this approach requires to design

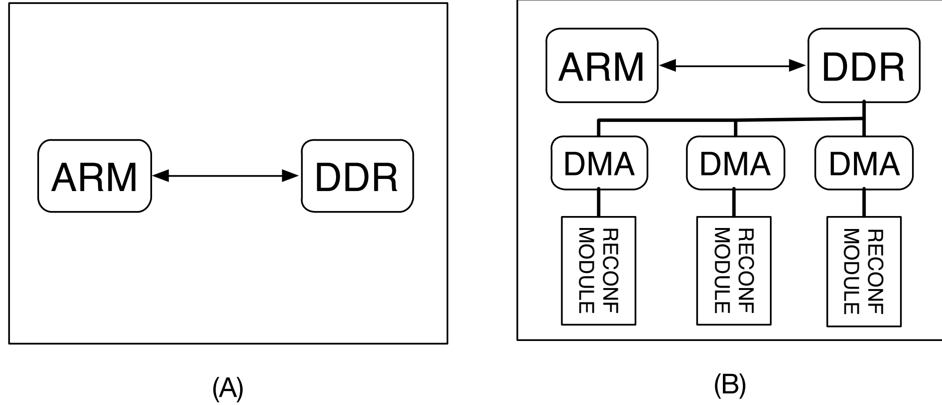

Fig. 1. Target architectures for the raytracing implementation. Fig. A reports the pure software architecture; Fig. B shows the architecture supporting partial reconfiguration with three reconfigurable modules.

a new architecture for each new scene that is requested to be analyzed by the raytracer algorithm.

For this reason, we envisioned an adaptation policy that exploits partial dynamic reconfiguration and applies the same concept, but at run-time (i.e., while a certain amount of intersections to compute are collected into the queues that linearize memory access). When one of the queues is full their content is checked and the reconfigurable modules are configured accordingly. If in a particular moment, there is only data in the queue used for computing intersections with spheres, we can configure the reconfigurable modules to parallelize the intersections on all the available resources by configuring on all of them to compute intersections with a sphere.

A. *Run-time adaptation policy*

The adaptation policy designed to optimize the raytracing aims at exploiting the behavior of the algorithm. In particular, using a grind as a representation of the scene, we have that rays traveling towards the same direction are likely to traverse the same regions of the image. The raytracing computation will be then characterize by phases each one having a similar number of intersections to compute for each primitives. As an example we may have the chunks of computation reported in Table II. If we analyze this table, we can see that we have a first part of the computation where only polygons are computed (chunks from 1 to 6) and then the second part combines polygons and spheres. This property is maintained throughout all the execution of the algorithm and the adaptation policy tries to exploit this pattern issuing the reconfigurations when it determines a change in the mix of the cores that are needed. In this case, at step 7, the policy will issue the reconfiguration for the polygon core while computing chunk 1 and then will perform the configuration only of the cone, since the polygon is already configured. This approach allows us to reduce as much as possible the number of reconfigurations by reusing the available cores. We want to highlight that the example reported in Table II does not represent the regular behavior of the system. For instance, it is not impossible to know for how

TABLE II
EXAMPLE OF CHUNKS OF INTERSECTIONS THAT HAVE TO BE COMPUTED AT RUNTIME. FOR EACH CHUNK THE AMOUNT OF INTERSECTIONS FOR EACH PRIMITIVE IS REPORTED.

CHUNK	POLYGON	SPHERE	CYLINDER	CONE	TRIANGLE
1	1004	0	0	0	0
2	1005	0	0	0	0
3	1002	0	0	0	0
4	1004	0	0	0	0
5	1004	0	0	0	0
6	1004	0	0	0	0
7	946	0	0	58	0
8	254	0	0	765	0
9	244	0	0	768	0
10	244	0	0	768	0

many chunks of computation the distribution of intersection to compute remains stable and more importantly the algorithm is unaware of the composition of the scene in term of primitives.

The code excerpt for the pseudo algorithm of the policy is reported below.

function ADAPTHW(chunkDescription, HWconfig)
  $HWupdated \leftarrow False$
  $sortedChunk = sort(chunkDescription)$
  $primitives = chunkToHWPrimitive(sortedChunk)$
  $newConfig = getFirstThree(primitives)$

  for $p \in primitives$ **do**
    if $p \notin HWconfig$ **then**
      $HWupdated \leftarrow True$
    end if
  end for
  if HWupdated **then**
    $newConfig = Reconf(HWconfig, newConfig)$
  end if
  return $newConfig$

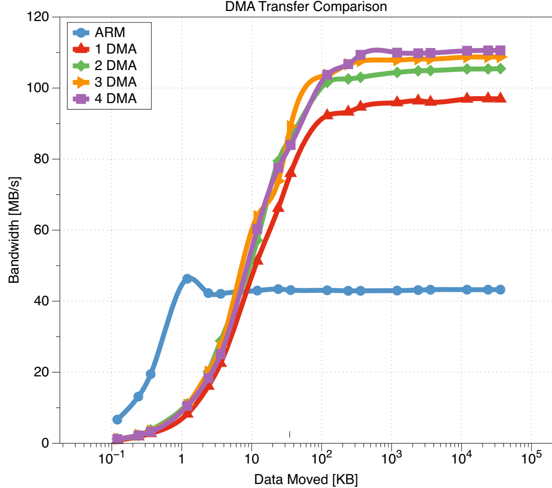

Fig. 2. Memory access performance. Comparison between memory access from ARM and access performed by the cores using DMA.

end function

The function $AdaptHW$ takes as inputs the chunk to be computed and the current hardware configuration. At first, it sorts the number of intersections to be computed ($sortedChunk$) and extracts which are the primitives that correspond to the sorted chunk ($primitives$). Since there is the possibility to configure only 3 modules on the device at a given time; the first three are selected and this is set to be the new configuration ($newConfig$). This new configuration is compared to the one in input ($HWconfig$) to check if there is any difference. In case, reconfigurations are performed where needed and the new configuration is returned.

VI. Results Evaluation

A. DMA Performance

As a first evaluation, we tested the performance of the DMA cores when moving data between DDR memory and the programmable logic. The DMA works using the AXI stream protocol and an apposite driver have been realized in order to use the DMA to move data from ARM DDR to the HW cores. The test has been performed copying an increasing amount of data up to 40MB blocks. Data is moved transferring at most 4KB at the time because that is the dimension of the FIFO used by the DMA core. Data is then transferred in parallel by using up to 4 DMAs. For each test, we measured the time needed to complete the transfer and we computed the transfer bandwidth in MB/s. For each test 10 measures have been taken and we averaged the results. Fig. 2 reports the results collected during this test and shows how the DMA transfers to the cores outperform the memory usage from the ARM processor when moving more than 40KB of data. This is the starting point of the evaluation. This behavior justifies the decision to accumulate the intersections to be computed and send all of them in chunks to the hardware accelerators.

TABLE III
Performance of the hardware cores and comparison with the corresponding software ones. Performance is reported as the number of intersections computed per second. For the Polygon primitive only the software implementation is available.

PRIMITIVE	SW [intersect/s]	HW [intersect/s]	Speedup
CYLINDER	0.30×10^6	40.13×10^6	130×
CONE	0.13×10^6	14.90×10^6	110×
SPHERE	0.14×10^6	15.97×10^6	108×
TRIANGLE	0.30×10^6	28.65×10^6	93×
POLYGON	0.31×10^6	-	-

B. Core Evaluation

The cores generated with HLS have been tested and compared with the corresponding software counterpart. The software performance depends how the data is allocated into the memory and how it is accessed during the computation, while the hardware version streams data in a linear way since they have been reorganized for computation as described in Section III-B. For this reason, the software performance has been measured on different situations and we picked the best measured performance for each primitive as the nominal performance of the core. Figures 4 report the comparison between the hardware and software performance on the y-axes (measured in intersections per second) against the number of intersections requested. For the polygon, only the software implementation is available as explained in Section III with the performance reported in Table III. Despite of the different performances, all the four graphs show the same behavior, which is the one reflected also by 2. When a small number of intersections have to be computed the software implementation outperforms the hardware one of a 8× factor; while increasing the number of intersections, the hardware implementation greatly outperforms the software obtaining a speedup of up to 100×. In general, the break-even point for choosing between software and hardware implementation is very low for our solution; for each one of the cores, the hardware implementation starts to have a better performance than the software one when a bunch of 10 intersections have to be computed. Considering the number of primitives that we have on the scenes under evaluation, we obtained in average a speed up of about 20× for the primitives executed in hardware. Since this break-even point is so small, in the adaptive algorithm we do not include this information into the algorithm and we assume that, if there is any intersections to be computed for a primitive in a chunk, it is always worth the use of the hardware version of the primitive. It is worth noting that, if the break-even point was bigger, the adaptive policy could be slightly modified in order to include this information and decide which implementations have to be use accordingly. Finally, Table III reports the maximum performance achieved with the hardware cores designed in this work through HLS and the speed up with respect to the corresponding software implementations.

C. Adaptive Raytracing

The adaptive raytracing algorithm has been tested on the test scene in Fig. 3. The scene is composed of a set of

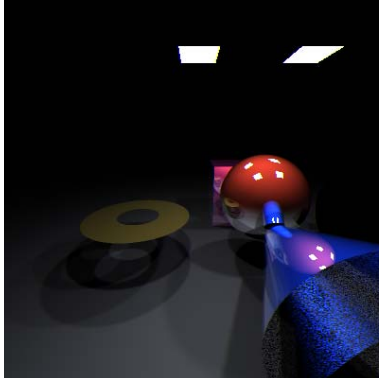

Fig. 3. Raytracer test scene under evaluation by the adaptive policy.

primitives and the raytracer was set up to create a 32x32x16 grid of voxels and render a 400x400 pixels output image. The maximum number of reflections of a ray in an image was set to 2. The part of the raytracer algorithm which traverses the grid and issues the intersections to be computed is executed on the ARM processor along with the adaptive policy. The reconfiguration is managed by the driver available in the Xilinx Linux distribution for the Zynq platform and is performed by writing the bitstream to the proper device file. The reconfiguration time of a single core in the architecture takes about 9 ms. The chunks used in the experiment are composed of 1,000 intersections in order to prevent an excessive memory occupation by the data to be arranged for the hardware accelerators.

The rendering test performed with this configuration allowed us to render the image computing 2,421 chunks performing only 7 reconfiguration throughout the execution. Considering the time required to compute the intersections, the speed up obtained with this configuration is of 31.8% and the algorithm spent 9% of the execution time waiting for reconfigurations to complete. Considering the number of intersections to be computed for the primitives for each chunk, even if the theoretical speed up (shown in Fig. 4) is in the order of 20×, the effective speed up is much less due to *1)* the reconfiguration overhead, *2)* the fact that not all the cores have been ported to hardware and *3)* the need to linearize the inputs of the cores before sending the data. On one hand, this step is faster than the software implementation. On the other hand, it suffers from the low memory access bandwidth illustrated in Fig. 2. A possible solution to this limitation could be a refactoring of the data structures, now based on pointers, to use arrays such as the FIFOs set up to accumulate intersections starting from the beginning of the algorithm.

VII. Conclusions and Future Work

This work proposes an adaptive implementation of the raytracing algorithm. The proposed solution dynamically configures the hardware at runtime in order to reflect the current need of the algorithm, exploiting the fact that a single block of the scene under analysis is likely to need the similar hardware cores to be processed. The policy designed to control the partial reconfiguration minimizes the number of reconfigurations needed by reusing the cores that have been already configured. Using this algorithm, it has been possible to obtain a speed up in computing intersections between light rays and primitives, up to 30% with respect to the baseline software implementation using the AVNET ZedBoard as a target device.

Future directions of research that stem from this work may focus on two sides. The first one consists in further optimizing this algorithm by working on the data structures used to represent the scene and optimizing them for the hardware cores used for the computation. The second one will try to investigate which algorithms may express a behavior similar to the one shown in Table II in order to apply the same adaptation policy to them in order to optimize their execution.

Acknowledgments

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

References

[1] M. Santambrogio and D. Sciuto, "Design methodology for partial dynamic reconfiguration: a new degree of freedom in the HW/SW codesign," in *Proc. of IPDPS*, Apr. 2008, pp. 1–8.

[2] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, "GAUT: A high-level synthesis tool for DSP applications," in *High-Level Synthesis*. Springer, 2008, pp. 147–169.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 1–27, Sep. 2013.

[4] R. Nane, V. M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler," in *Proc. of FPL*, Aug. 2012, pp. 619–622.

[5] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *Proc. of FPL*, Sep. 2013, pp. 1–4.

[6] T. Feist, "Vivado design suite," *White Paper*, 2012.

[7] Cadence, "C-to-Silicon Compiler," http://www.cadence.com, 2013.

[8] Forte, "Cythesizer," www.forteds.com, 2013.

[9] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[10] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[11] A. Segovia, X. Li, and G. Gao, "Iterative layer-based raytracing on cuda," in *Proc. of the Int.l Performance Computing and Communications Conference (IPCCC)*, Dec. 2009, pp. 248–255.

[12] 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.

[13] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek, "Realtime ray tracing of dynamic scenes on an FPGA chip," in *Proc. of the Conf. on Graphics hardware*, Jul. 2004, pp. 95–106.

[14] S. Woop, J. Schmittler, and P. Slusallek, "Rpu: a programmable ray processing unit for realtime ray tracing," in *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3, 2005, pp. 434–444.

[15] A. S. Nery, N. Nedjah, and F. Frana, "GridRT: A Massively Parallel Architecture for Ray-Tracing Using Uniform Grids," in *Proc. of DSD*, Aug. 2009, pp. 211–216.

[16] S. Guntury and P. Narayanan, "Raytracing Dynamic Scenes on the GPU Using Grids," *IEEE Trans. on Visualization and Computer Graphics*, vol. 18, no. 1, pp. 5–16, 2012.

[17] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on GPU with BVH-based packet traversal," in *IEEE Symp. on Interactive Ray Tracing*, Sep. 2007, pp. 113–118.

[18] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," in *Proc. of the Conf. on Graphics hardware*, Jul. 2005, pp. 15–22.

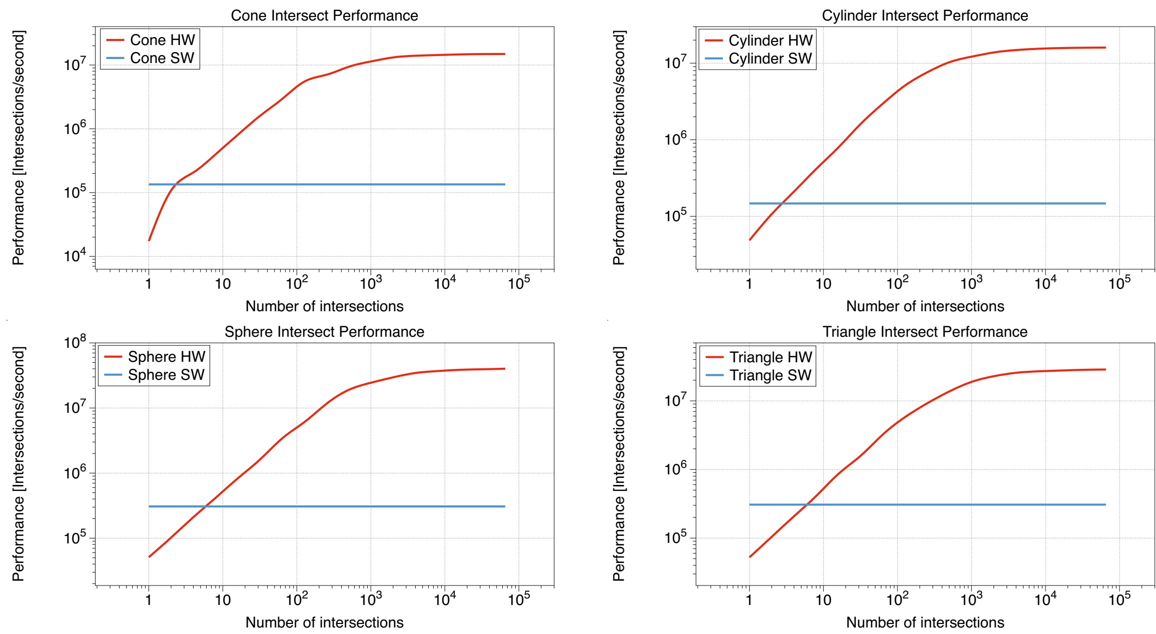

Fig. 4. Performance of hardware cores compared to the corresponding software ones. Performance is measured as the number of intersections per second against number of intersections to be computed.