Efficient use of Multiple Hardware Components for Image Synthesis

Francisco Pereira João Paulo Moura José Afonso Bulas-Cruz Luís Magalhães
Universidade de Trás-os-Montes e Alto Douro
Quinta de Prados, Vila Real
{fsp,jpmoura,jcruz,lmagalha}@utad.pt

Alan Chamers
Warwick Digital Laboratory
University of Warwick
Warwick, UK
A.G.Chalmers@warwick.ac.uk

Abstract

With the advent of affordable and powerful computers systems, even in desktop configurations, the rapid rendering of complex scenes using global illumination algorithms is within reach. However, some significant problems still must be dealt with when performing such physically based lighting simulations if we are to achieve real time performance on current technology. The power offered by modern computer systems is not only due to the increasing CPU capabilities, but also to the fast development of others system elements, including graphics processing units (GPU) and other additional processing boards. Ray tracing implementations in such environments typically target only one hardware component, or at best map specific type of work to particular elements within the system. This approach leads to an unbalanced work distribution and marginalisation of certain types of resources. In this paper we present a framework for managing and interfacing all the available computational power within a modern PC in a balanced and efficient way. At the centre of proposed approach is an abstraction layer between the main rendering stage and the primitives of the rendering process. Several advantages of such organisation are discussed, including portability and easy extendibility to newer powerful hardware resources as they become available, optimizations at resource level, modularity, and load balancing. More practical issues regarding the management strategy and implementation for heterogeneous environments below this level of abstraction, where performance is a key aspect, are also presented. To validate this model a set of experiments were conducted. The results have shown that the introduction of an abstraction layer into the rendering system can improve the performance and better use the available resources.

Keywords

Parallel Rendering, Platform Independent, Ray Tracing, Real-time.

1. INTRODUCTION

The significant increase in computational power within standard computer systems and the range of resources available provided the necessary framework for the development of customised ray tracers. Implementations have targeted several types of commodity hardware [Buck04, Chalmers02, Parker99, Zemcik03] including single processing elements [Wald01] or multiple ones in a homogenous environment [Parker99, Wald01], specific hardware elements [Zemcik03] and graphics processing units (GPU) [Buck04, Parker99]. The design of such implementations was tailored to the underling hardware at the algorithm and data structures level, for performance purposes, and essentially targets one class of resources, although computers systems are composed of several hardware elements that are able to contribute to the rendering process. Portability to other architectures or resources is not a key issue, at least in the design stage. More details of the referred approaches are presented in section 2.

The work developed and presented in this paper is part of the Rendering on Demand (RoD) project [RoD09]. The project main objective is to produce high fidelity animations in real time. For this purpose three levels of speedup are considered:

- Perception - avoiding unnecessary calculation where users will not perceive any difference between a lower and full quality rendering [Cater03, Dumont03, Sundstedt04];

- Parallel processing - provide an extended computational power pool using several nodes and manage efficiently data and tasks within the work at hand [Chalmers02, Wald02, Wald03];
- Parallel processing at resources level - use all the resources available in each node and maximise each resources contribution throughput in the rendering process.

The work described in this paper is related to the parallel processing inside each node of the system, with the overall goal of real-time rendering in mind. This constraint is embedded in the framework design and low level support was included in the design.

A more clear view of the interaction between levels in the RoD project is shown in Figure 1.

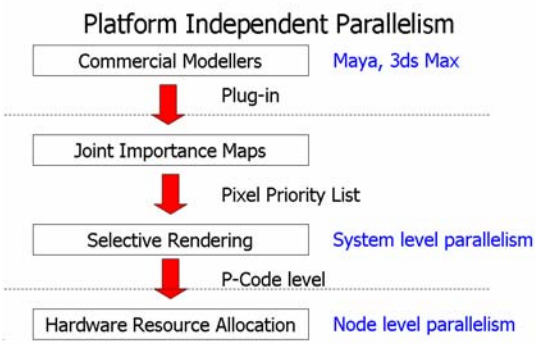

Figure 1 – Overview of RoD

2. RELATED WORK

Rendering of high fidelity images and animations has been undertaken on both CPUs and dedicated hardware. In the CPU domain, optimisation at low level, such as instruction level parallelism and data representation have been used to achieve higher performance [Wald02]. The main type of hardware targeted, besides the CPU, is the GPU. Other types of hardware have also been considered. This section will give an overview of ray tracer implementations carried out using graphics cards, processing boards and CPU optimisations.

Newer approaches and platforms aim to combine and facilities a transparent programming between the CPU and the GPU world. Such examples include CUDA platform for parallel processing [Halfhill08] and OpenCL open standard for heterogeneous parallel programming [Saha09]. These approaches intend a more general view of problem solving beyond ray tracing [GPGPU09, Owens07].

2.1 Graphics card approach

The current graphics cards architecture is designed to speedup the rasterisation of images from a geometry/texture model, although slowly shifting towards a more powerful generic coprocessor in more recent times. Outside the computer graphics domain other algorithms have been implemented, for example, Fast Fourier Transform (FFT), Basic Linear Algebra and matrices calculations [Buck04]. It has been shown that the performance between a top of the line GPU and the CPU counterpart are very similar for a variety of problems. The main advantage of the GPU is the presence of multiple integer and floating units, in a parallel and pipelined architecture, that benefits the implementation of matrices based algorithms.

Although the GPU is not optimized for ray tracing, several primitives are shared with rasterisation making this type of hardware attractive for ray tracer developers. The fast growing computational power/low cost ratio in GPUs is also appealing. Ray tracing algorithms implemented on such architectures have achieved performances similar to those achieved with a standard CPU [Carr02] and several algorithms in the computer graphics domain have been adapted or implemented [Buck04].

The definition of a general-purpose stream language for GPUs by Buck et al. [Buck04] demonstrates the versatility and power of commodity graphics hardware. The language takes advantage of the GPU SIMD architecture, programmable vertex and fragment processors, texture memory and internal registers to implement a streaming paradigm environment. Streams are arrays of similar data types that can be processed in parallel by basic transformations defined as kernels. Results of one kernel are passed to the next one in an efficient way using internal memory. Single parameters are stored in internal registers as read only. All the specificity of the hardware is hidden from the programmer by the compiler. The programmer only needs to define the streams, input and output, and implement the intended algorithms in the kernels. This type of environment is well suited for problems with SIMD behaviour. The major drawback in GPUs programming is the lack of branching in the programmable vertex and fragment processors which difficult the implementation of certain algorithms such as octree crossing.

2.2 Processing boards

More recently some specific and general purpose types of hardware boards have been used in the computer graphics domain. The evolution of microelectronics has brought to the market faster and more resourceful devices such as Digital Signal Processors (DSP) and Field Programmable Gate Arrays (FPGA).

Zemcik et al. [Camea09, Zemcik03] implemented a particle rendering pipeline into a hardware board which includes a DSP and a FPGA. The DSP is mainly devoted to interfacing and managing the system, and FPGA programming. The FPGA is pro-

grammed with the rendering algorithm and accesses local DRAM present in the board for the data/results. The main advantage of a FPGA is its adaptation at instruction level to a specific problem, enabling the implementation of SIMD instructions for the particle rendering, in this case. The communication to the board is available through a serial or Ethernet link. A newer version of the board will include a PCI interface and several FPGAs on the same board.

The FPGAs provide a simple and powerful way to implement problem specific instructions with low impact in the algorithm pipeline. Data parallelism and pipelining at instruction level can be programmed into them to obtain higher performance in specific algorithms. Nevertheless, operations such as isolated integer and floating point arithmetic are not best suited for this type of hardware.

More powerful and specific boards have been used for rendering [Coulthurst08], but the interface and programming model is more close to the GPU model, with several processing elements and shared memory architecture.

2.3 CPU optimizations

At the CPU level, significant performance increases have been achieved by maintaining the CPU pipelines and functional units at full capacity, through data representation adaptation and algorithms re-engineering. Examples of this type of optimisations are CPU cache alignment and use of instruction level parallelism [Wald02, Wald01]. This includes SIMD instructions present in the Intel Pentium™ architecture. Wald et al. implemented a ray tracer optimised for the Pentium architecture [Wald03] especially at low level data representation and SIMD instructions. Rays Coherency was used to pack four intersection tests with triangles into the Pentium pipeline, taking advantage of the CPU SIMD instructions.

At a higher level the accelerating structure was designed to benefit from the first and second level of CPU cache alignment. Access time to one level of cache is normally one order of magnitude less than the next level of memory. In the shading component of the ray tracer a similar approach to the ray intersection test was followed, but due to data rearrangement the speedup obtained was less than ray intersection. The conjugated speedup of the ray tracer implemented, compared with Rayshade and POV-ray, was of one order of magnitude. Compared results from standard rasterisation with the implemented ray tracer have shown that ray tracing is more affordable in scenes with a very large number of polygons, above one million. This is due mainly to the logarithmic and linear computational power increase with scene complexity, for ray tracing and rasterisation respectively [Wald03].

3. HRA OVERALL DESIGN

The Hardware Resource Allocation (HRA) is designed to provide an abstract interface between the Selective Renderer and the underling hardware, as shown in Figure 2. In addition to the typical data structures adaptation and optimisation of the low level rendering primitives in each type of resource, the HRA must also ensure that all the resources are working to their maximum capacity. The HRA maintains a status of each resource, such as work units dispatched and data/code present in each resource's local memory. The interface between the main rendering program and the HRA process is provided through two bins: One for depositing work units, by the selective renderer thread; the other for collecting processed results.

The HRA collects from the work units' bin a set depending on type and the resources' actual allocation and status. This set is then dispatched to a selected resource according to the associated communication strategy in one or more packets. After finishing the work, the resource returns the result to the HRA to be stored in the results' bin. Each resource contains a part of the rendering algorithm and partial data needed to complete the requested work units. All the global management and resources status is kept in the HRA main thread.

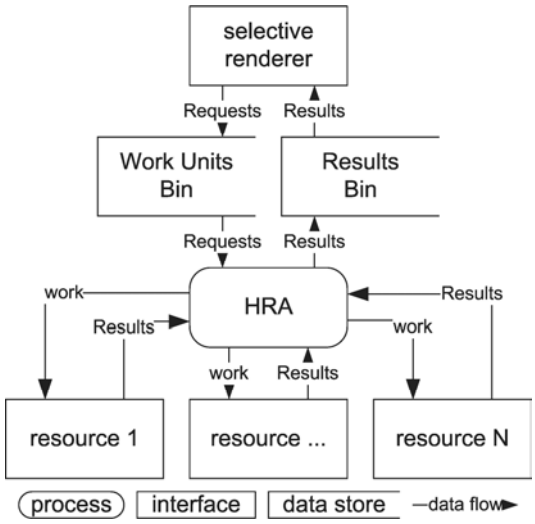

Figure 2 – HRA dataflow

3.1 Implementation issues

In order to get the maximum computational power from the underlying hardware, it is important to ensure that the workload is subdivided appropriately amongst the hardware. When considering several different hardware resources, a number of issues need to be considered. These include data representation and size, communication between the main loop in-

teraction and available resources, and the programming model followed in each type of resource.

The main purpose is to use all the available resources in the common goal, achieving higher performance and keeping the generated overhead, due to data conversion, management and communications, within a reasonable margin.

3.2 Data representation

There is a trade-off between portability and efficient implementation of the data structures involved in any algorithm. One aspect is the basic data types available in each type of resources and respective resolution. For example, ray tracing can be implement using integers or floats. Floating point is more appropriate for high fidelity graphics. However, graphics cards currently provide some non standard representations in the arithmetic units such as 16 bit floats, half resolution, or, 20 bit [Hutchins08].

At a higher level, the data structures representing the scene elements and basic primitives, rays and shapes for example, can be internally rearranged and adapted to the underling hardware caching scheme or optimized for the processing element pipeline at instruction level parallelism [Wald03]. When using heterogeneous hardware, the data alignment and organisation can be very distinctive between architectures. The overhead of converting on the fly from one representation to the other may cancel any gain obtained from the code/data optimisation. In addition, the organisation of the high level data structures, for example an octree, must take into account the caching schemes between the main program loop and the partial code running in the resource.

3.2.1 Heterogeneous communication models

A simple master/slave model can not fulfil all types of communication between the HRA and resources, because it does not take into account the broad range of latency values and communications paths within the system. This latency can vary from some nanoseconds in a shared memory approach to several milliseconds in a dedicated connection, such as an Ethernet link.

The communication model must ensure that when sending work packets, it can do it without requiring the results from previous work packets which have yet to be processed. In addition, the communication system must avoid contention and be fast enough to provide work packets rapidly to the faster processing resources. For the slower resources, a pipelining approach is more appropriate.

The communication strategy should also take into account the need to keep the hardware pipeline full or near its full capacity throughout at all the time during the rendering process, for each of the available resources.

3.2.2 Heterogeneous programming models

The different types of hardware available today in common desktop computers provide the systems with useful additional computational power. However, the underling programming model may vary significantly for each type of resource. Such programming models include the sequential imperative model and the streaming model. This last one is most commonly used in graphics cards. The proliferation of FPGAs, in several types of processing boards, has introduced a new paradigm, mixing traditional programming techniques with reprogramming/reorganisation at the hardware level, at setup or runtime. When integrating all these models together, the designed interaction for the ray tracing algorithm must take into account several layers with different types of granularity.

This mixing of several different programming models should not interfere with the main algorithm running on the CPU. Small and simple basic primitives are thus better to accomplish the interaction between the main renderer and the hardware resources, and easier to implement in any of the different programming models.

3.3 HRA element design

3.3.1 Primitives

Load balancing amongst the different hardware resources, heterogeneous or homogeneous, is best achieved using a fine grain work division. With this in mind, the ray tracing algorithm is divided into three levels: low level primitives, high levels primitives, high level recursive management. The lowest level includes all the basics primitives involved in ray intersection and shading with simple objects representation. Normally, this kind of primitives can be easily implemented in a variety of hardware, even with low memory capacity or programmability. Furthermore, there is no sequential dependency between the results, thus it is possible to compute them independently and concurrently. This approach helps to divide the work more uniformly amongst the available resources.

More general data structures and complex algorithms, which can be composed of low level primitives, are implemented as a single higher level primitive. An example of a high level primitive is ray intersection with an acceleration structure, such as the octree.

The most important feature required from either a low or high level primitive is that it must be self contained. This means that all the data necessary to obtain the result for that primitive and all the necessary code for completing the work is available; no other primitive will be invoked unless it is an integral part

of the one being executed. The primitive is also time bounded to compute the result. This last rule excludes recursive calls on the primitive itself. All the high level recursive nature of a global illumination algorithm is performed at the main rendering loop, using decomposed primitives and results previously calculated.

This decomposing of the rendering algorithm into three components, basic primitives, high level non self recursive primitives and high level data/function management permits the definition of a standard API which can be implemented in each resource. The implementation of the API in a specific type of hardware can be partial due to its memory size or programmability limitations. For this reason the HRA is configured with a list of capabilities for each type of hardware available.

Marginalisation of the hardware and workload unbalancing occurs whenever all the work units in a large time interval can not be processed by a specific type of hardware. One option to minimise this aspect in future developments is to start calculations ahead of schedule for the type of primitives implemented in the idle resource. This is not a trivial approach because it requires more state management from the main rendering loop or even from the HRA. Another solution can be to increase the work unit bin length, but also this can cause less interactivity in a real time system implementation. This issue will be further discussed in section 3.3.3.

3.3.2 Work units

The work submitted to the HRA is in the form of work units. A work unit is composed of a data set and the function to be applied to that data set. An example of a work unit is a simple intersection test between an object and a ray. In this case the test is the function, the object and rays are the data set. The work unit does not contain the actual data or the code itself, but references to it. This approach offers several advantages such as smaller packets sent by HRA to the resources when the data or code is already present in the resource local memory or smaller memory footprint in the HRA and resources. It also permits the grouping of several work units together by the HRA, optimising the resources allocation and the packet size. In this last case only a copy of each data set or code is needed even when it is not present in the targeted resource local memory. The advantage is even more obvious in the resources with shared memory access. The data set is an input to the primitive in a read only mode, making data blocking access unnecessary.

The code refers to one primitive previous defined, which make a work unit an instance with a specific data component. The work units can be seen as a function of code and data references. These parameters are used in the algorithm for management and dispatch of work units amongst the resources.

3.3.3 Bins for work units and results

The communication and flow control between the node Selective Renderer component and the HRA is implemented on top of two data structures called bins. The work units' bin has a maximum length that acts as a flow control mechanism. When the bin is full the "add work unit" call will block the caller, being awake after successful insertion. In the current implementation, the length value is statically defined as well as the notion of full and empty. By workload profiling, a strategy for dynamically reconfiguring the length value and full/empty definition by the HRA can be implemented in the future to improve system efficiency and interactivity. A higher value improves the system efficiency, bigger work units' set to choose from, but can delay results and adversely affect interactivity.

Both bins reside in a shared memory space in order to minimise the latency. Locking inside the bins is performed by mutexs.

Frequently, a sequence of submitted work units has some component in common, data or code. To accommodate such situations a set of group submission functions are provided, minimising the interaction and reducing the submission time. This option also reduces the time spent in the HRA sorting algorithm for work units' dispatch to the resources, due to pre-submission ordering reduction from the selective renderer.

The results' bin is simpler to manage because it is mainly a communication and storage element. No size limitation is imposed on it. The results from this bin are removed by the renderer main loop.

3.3.4 HRA data representation

The representation of the basic data types must take into account the diversity of hardware, namely the bitwise representation of values. Problems with size and alignment must be solved before passing the data to the resources. To overcome alignment problems, the implementation of the basic rendering data types, such as points, rays, intervals, pixels, colours, are defined as arrays. More high level and complex structures use heterogeneous elements that are mainly composed of the basic data types referred to previously. The basic rendering data types are defined on top of a standard float representation common to all the available hardware.

3.3.5 HRA work unit sorting and management

The main function of the HRA, besides providing an abstraction layer between the render and the hardware, is to maximise the efficiency of the whole sys-

tem. In this heterogeneous parallel environment the order of work units is not totally imperative, at least in a small scale, which enables order rearrangement of the work units. This approach is used to maintain the resources pipelines at a maximum capacity.

A group packet submission, of work units of the same type, can improve memory management and efficiency at the resource. Only a read copy of each data or code is needed, avoiding multiple tests in the resource local cache.

The HRA design follows the RoD project main specifications, in which it was defined that all the work presented to HRA must be performed. Each work unit has a life time validity and if one or more work units are constantly not dealt with, due to the arrival of new work to the bin, its validity will be overdue. To avoid this problem a maximum delay, sufficiently small relatively to the life time validity, is set for each work unit. All the work units with a delay greater than the maximum have highest priority. The influence of the bin maximum size in this matter is taken into account at setup time.

3.3.6 HRA/resources communication

When analysing the communications between the HRA and the resources, two properties are relevant: the bandwidth available and the latency of the connection. Taking into account these parameters different strategies need to be considered.

A simple master/slave approach is implemented for high bandwidth and low latency. The work unit or group of work units is sent regardless of the packet size. An example of this approach is the VirtualCPU. The result of a work unit is collected by the HRA as soon as the execution in the "VirtualCPU" thread finishes the work assigned. The latency, in this case, is very small and the resource is not dedicated 100% to the work unit execution. The idle time, when the VirtualCPU thread is waiting, can be used by the other threads in the system such as the HRA and the selective renderer.

For resources with high latency, the previous model does not provide an interaction capable of maintaining the resource pipeline fully occupied. The idle time will affect the resource efficiency, when waiting for a new work unit and sending a result back. For this type of situation the communication protocol must provide a mechanism that enables the transmission of several work units without results acknowledge. The sliding window protocol, also used in the TCP/IP protocol suite, is a solution for this case. Each packet is numbered in sequence order when it is sent. The window is defined by the number of packets that can be sent without receiving the results. Whenever results are received more packets can be sent to the resource. In the current implementation the work unit order is always preserved and it is assumed that none of the packets are lost in the communication. The implementation of the sliding window strategy is more complex than the master/slave approach and may be difficult to implement on certain types of hardware. A ponderable decision between efficiency and complexity must be taken regarding each type of hardware available. The sliding window model can be applied to almost any resource, but the window size must be tuned according with each resource capabilities and communications parameters.

For resources with low bandwidth others aspects must be taken into consideration. In this case the HRA have to optimize the work units/packet size ratio. This means more work units in smaller packets. This aspect is also taken into account with small size memory devices in order to minimise local cache and cache misses.

The choice between a master/slave approach or sliding window communication protocol must take into account the overhead in complexity implementation and the gain in using the more complex model. In a fast communication link between a outside resource (e.g. external processing board) and the main system, the gain in efficiency may not be worth if it leads to a complex communication model implementation.

The communication between the HRA and the resources does not confine itself to sending work units and receiving results. Additional packets or fields inside a packet are used to exchange system information such as workload or memory status. This information is used by the HRA to manage and distribute work units amongst available resources more efficiently.

4. TEST AND RESULTS

In order to test the proposed model in terms of an overall efficiency, a set of experiments were defined. The efficiency between the proposed model and a standard ray tracer implementation were compared. The test bed was designed only to measure the impact of the HRA in the overall rendering process speedup, and the introduction of more than one rendering resource in the rendering system. In this sense no low level optimizations were introduced for any kind of specific hardware that might alter the experiments results. Also an effort was taken to minimize the impact of external variables to designed goal, such as system or hardware level indirect parallelism. As example, Intel© Hyper Threading™ technology was disabled to avoid CPU level parallelism, since the base setup, the standard ray tracer implementation, is a sequential single threaded program while the others configurations were multithreaded.

4.1 Defined experiments

Three types of experiments were defined to evaluate the impact of the introduction of the HRA in the system (overhead) and the gains (speedup) achieved with the inclusion of two hardware rendering elements into the rendering pipeline, managed by HRA.

In this first type of experiments the main rendering loop calls the primitives directly, as in a standard implementation. The main rendering loop and the primitives are executed within the same thread and no form of parallelism is used from the user point of view.

The rendering time from this configuration was used as a base for relative index for others configurations. It was, in some sense, considered as a standard ray tracer implementation.

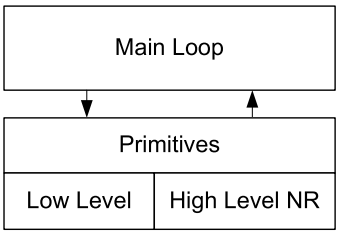

Figure 3 – CPU without HRA

On the second type, the HRA was introduced and instead of calling the primitives directly, the main rendering loop creates work units and puts them in the work unit's bin. The HRA was configured to use only the CPU (VirtualCPU) as a rendering resource. The main rendering loop, the HRA and Virtual CPU resource are executed within each own thread. A total of three threads are used and run concurrently, at least from the user point of view. This configuration is more prone to width system parallelism.

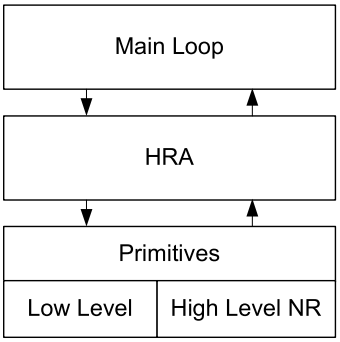

Figure 4 – CPU with HRA

Finally in the third and last configuration the HRA is used with two hardware rendering resources, the CPU (VirtualCPU) and the GPU. Besides the three threads found in the previous configuration, primitives are also executed in the GPU at the same time.

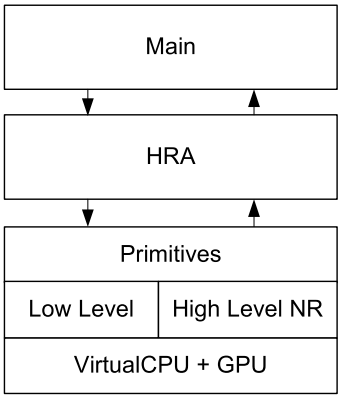

Figure 5 – CPU+GPU with HRA

In terms of software implementation, and according with the proposed model, three layers were defined.

- Main ray tracer loop – Responsible for analysing the scene description, produce the necessaries work units and gathering the results to produce the renderer image;
- HRA – With the goal of managing the submitted work units in accordance with the defined strategy and available resources;
- Rendering primitives – Implementation of all standard low level and high level primitives in the rendering algorithm, with no recursion (NR in the figures means No Recursion).

4.2 Hardware setup and results

Two hardware setups were used as platforms for testing the defined experiments. Platform 1 was composed by a 3GHz Pentium 4 single core, 1 GiB of memory (DRR400), and a NVIDIA GPU 6600GT (128MiB) with an AGP interface. Platform 2 was composed by a 3GHz Pentium 4 single core, 1 GiB of memory (DRR533), and a NVIDIA GPU 6600GT (256MiB) with a PCI Express interface.

Two scenes descriptions were considered, A and B, with different geometry and photometric characteristics. Scene A focused more on interreflections and a higher diffuse light component, while Scene B contains fewer interreflections and higher direct light component that means more coherency between casted rays and highest cache hit rate .

Table 1 and Table 2 present some of the default parameters and obtained results from running the experiments ten times in each different combination of configuration, scene, and platform.

PC1+GPU1	Time (in seconds)		
	CPU without HRA	CPU with HRA	CPU+GPU with HRA
Number of rays	100000000	100000000	100000000
Maximum bounces	5	5	5
Number of resources	1	1	2
Scene A			
Average - max - min	201,13	234,73	163,19
Relative index	**100%**	**117%**	**81%**
Scene B			
Average - max - min	166,81	188,71	139,53
Relative index	**100%**	**113%**	**84%**

Table 1 - Rendering scenes A and B (platform 1)

PC2+GPU2	Time (in seconds)		
	CPU without HRA	CPU with HRA	CPU+GPU with HRA
Number of rays	100000000	100000000	100000000
Maximum bounces	5	5	5
Number of resources	1	1	2
Scene A			
Average - max - min	154,99	172,72	123,00
Relative index	**100%**	**111%**	**79%**
Scene B			
Average - max - min	127,69	149,40	95,37
Relative index	**100%**	**117%**	**75%**

Table 2 - Rendering scenes A and B (platform 2)

The HRA was designed to work with several and distinctive types of resources at the same time. When only one resource is managed by the HRA no real speedup is achieve with its introduction in the rendering process. In this case it will only delay the work units' execution and overcharges the CPU with an extra burden in terms of computational requirements. Actually it can be observed that this overhead, about 13%, will reduce the system capacity which makes the use of the HRA in a standard sequential and single computational resource configuration unwise.

When comparing the first, CPU without HRA, and third test, CPU + GPU with HRA, the speedup obtained is 1,39. In this case both CPU and GPU are used in the scene rendering simultaneously.

In all the experiments the system, overall configuration and not only the render specific setup, was configured to avoid or minimize any side effect for operating system or hardware tuning. One example was turning off the hyperthreading option from the system, because it usually benefits programs with multiple threads. Also experiments were run more than once, trying to minimize the system caching interference. The presented average times from the set of experiment realized within the same type, do not include the best and worst time, to remove possible interferences from external elements, such as file system caching policies or operating system memory management.

5. CONCLUSIONS AND FUTURE WORK

The work presented in this paper shows that the inclusion of a hardware resources allocator (HRA) in a system with multiple computational resources available may reduce the rendering time. The HRA presents several advantages over similar approaches. The most important is the portability in system design, by separating the rendering process in three levels, with simpler and smaller work units. The inter-operability of different types of hardware is important, because it does not limit the use of such a system on a variety of platforms. The reduced size of some of the defined primitives and partial implementation is useful for testing and porting to newer and more powerful hardware. The development/test cycle can be shortened because it is unnecessary to implement the full rendering API specifications into the new resources.

The HRA design does not only limit itself to manage the workload between available resources, but has additional indirect benefits at other levels, such as code optimisation, execution reordering, resources caching schemes and more high levels caching strategies.

The current implementation aim is to demonstrate the versatility and performance enhancements of this type of approach. Further work is needed to develop dynamic parameters tuning, such as the bins length, work units sorting algorithm taking into account available resources in the running system, or HRA/resources communications protocols. Also more localised values and profiling, such as data/code distribution, resource occupation and cache hit at several levels must be evaluated in order to fine tune the overall model to particular and specific conditions.

6. ACKNOWLEDGMENTS

The work reported in this paper was formed from part of the Rendering on Demand (RoD) project within the 3C research programme, who's funding and support is gratefully acknowledged.

This work was partially supported by the FCT - Fundação para a Ciência e a Tecnologia (Portugal).

The authors would like to thank all of RoD project team, especially its leader Prof. Alan Chalmers, for all the support and help provided.

7. REFERENCES

[RoD09] 3CR Projects Rendering on Demand <http://www.3cresearch.co.uk/renderingondemandprojectpage.htm>

[GPU09] General-Purpose computation on Graphics Processing Units <http://www.gpgpu.org>

[Camea09] Computation accelerators - Camea – signal and image processing <http://www.camea.cz/en/products/technologies/computation-accelerators/>

[Buck04] Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat. Brook for gpus: Stream computing on graphics hardware, 2004.

[Carr02] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[Cater03] K. Cater, A. Chalmers, and G. Ward. Detail to attention: Exploiting visual tasks for selective rendering. In Eurographics Symposium on Rendering 2003, pages 270–280. ACM, June 2003.

[Chalmers02] A. Chalmers, T. Davis, and E. Reinhard. Practical Parallel Rendering. AK Peters Ltd, July 2002.

[Coulthurst08] David Coulthurst, Piotr Dubla, Kurt Debattista, Simon McIntosh-Smith and Alan Chalmers. Parallel Path Tracing using Incoherent Path-Atom Binning. Spring Conference on Computer Graphics 2008, April 2008.

[Dumont03] R. Dumont, F. Pellacini, and J. A. Ferwerda. Perceptually driven decision theory for interactive realistic rendering. ACM Trans. Graph., 22(2):152–181, 2003.

[Halfhill08] Tom Halfhill. Parallel Processing with CUDA. Microprocessor Journal, 2008. <http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf>

[Hutchins08] E. A. Hutchins and B. K. Angell, "US Patent 7199799 - Interleaving of pixels for low power programmable processor." vol. 2008, 2007.

[Owens07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. "A Survey of General-Purpose Computation on Graphics Hardware". Computer Graphics Forum, volume 26, number 1, 2007, pp. 80-113.

[Parker99] S. Parker, W. Martin, P.-P. J. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In Symposium on Interactive 3D Graphics, pages 119–126, 1999.

[Saha09] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson, "Programming model for a heterogeneous x86 platform," PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, pp. 431-440, 2009.

[Sundstedt04] V. Sundstedt, A. Chalmers, K. Cater, and K. Debattista. Topdown visual attention for efficient rendering of task related scenes. In VMV 2004 - Vision, Modelling and Visualization. Stanford, November 2004.

[Wald01] I. Wald, P. Slusallek, C. Benthin, and M.Wagner. Interactive rendering with coherent ray tracing. In A. Chalmers and T.-M. Rhyne, editors, EG 2001 Proceedings, volume 20(3), pages 153–164. Blackwell Publishing, 2001.

[Wald02] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive Global Illumination using Fast Ray Tracing. In Proceedings of the 13th EUROGRAPHICS Workshop on Rendering. Saarland University, Kaiserslautern University, 2002. Available at http://www.openrt.de.

[Wald03] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In Eurographics State of the Art Reports, 2003.

[Zemcik03] P. Zemcik, P. Tisnovsky, and A. Herout. Particle rendering pipeline. In SCCG '03: Proceedings of the 19th spring conference on Computer graphics, pages 165–170. ACM Press, 2003.