EG UK Theory and Practice of Computer Graphics (2009)
Wen Tang, John Collomosse (Editors)

Hardware Accelerated Shaders Using FPGAs

Luke Goddard[†1,2] and Ian Stephenson[‡1]

[1]National Centre for Computer Animation,Bournemouth University,UK
[2]Double Negative Visual Effects, UK

Abstract

We demonstrate that Field Programable Gate Arrays (FPGAs) can be used to accelerate shading of surfaces for production quality rendering (a task standard interactive graphics hardware is generally ill-suited to) by allowing circuits to be dynamically created at run-time on standard commercial logic boards. By compiling shaders to hardware descriptions, they can be executed on FPGA with the performance of hardware without sacrificing the flexibility of software implementations. The resulting circuits are fully pipelined, and for circuits within the capacity of the FPGA can shade MicroPolygons at a fixed rate independent of shader complexity.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics Processors

1. Introduction

When rendering high quality images for film and video production, the process of shading (the calculation of surface colour, for specific lighting and observer) is still a significant bottleneck. Though realtime rendering systems have adopted hardware acceleration to implement similar techniques at interactive frame-rates, the limitations introduced by this hardware make it unsuitable for use in animation production.

By retargeting a RenderMan SL compiler [HL90] to produce FPGA circuits, we create a hardware accelerated shading engine, which dynamically rebuilds its hardware at render time, to create acceleration circuits for each surface type required. This allows complex programmable shaders [AG00] to be executed at performance levels approaching hard-wired shaders.

2. Background

In modern renderings systems the calculation of light/surface interactions is performed by programmable modules known as shaders. These are typically written in a custom programming language such as the RenderMan Shading Language (SL) [HL90]. Figure 1 shows a simple shader. Surfaces are diced into MicroPolygons (MPs) the size of a single pixel and the shader is executed on each of them, making shading one of the most costly part of the rendering process [Ele04]. As such it is a candidate for hardware acceleration.

```
surface plastic (
          float Ka = 1;
          float Kd = .5;
          float Ks = .5;
          float roughness = .1;
          color speccolor = 1;
          )
{
    normal Nf;
    vector V;
    Nf = faceforward (normalize(N),I);
    V = -normalize(I);
    Oi = Os;
    Ci = Cs*(Ka*ambient()+Kd*diffuse(Nf))+
     speccolor*Ks*specular(Nf,V,roughness);
    Ci*=Oi;
}
```

Figure 1: *A Plastic Shader in RenderMan SL*

Early attempts at accelerating programmable shading, such as PixelFlow [OL98], The Pixel Machine [AS93], and Pixar's RM-1 were based around custom hardware. Despite

† e-mail: luke@lukegoddard.info
‡ e-mail: ian@dctsystems.co.uk

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG09/109-116

the power of these systems, the high cost of developing hardware made them impractical to develop beyond initial prototypes.

In contrast to these systems, early hardware accelerators for interactive rendering were much more primitive, and provided only fixed shading models. However, driven by the games market they became progressively cheaper and more powerful. It was then demonstrated [POAU00] that procedural shading is possible on this hardware.

The demand for increased realism from the gaming community led to support for programmable shaders in interactive rendering hardware and these were rapidly developed to appear more like RenderMan SL [PMTH01, MGAK03].

However, despite the apparent similarities between real time shading languages and their off-line equivalents, the hardware accelerated shading provided by current graphics cards is still a poor fit for production rendering. While good results are possible [WGER05] the fixed graphics pipeline in interactive graphics cards is still very different to that found in high quality rendering [CCC87] where the distinction between vertexes and fragments is not made. Features which are essential in production rendering are often difficult or impossible to implement within these fixed pipelines, where trade-offs have been made between speed, flexibility and image quality in order to reach interactive frame rates.

The Sony Playstation architecture(s) [Son01] and Intel's Larrabee [SCS*08] offer some escape from this [Ste03], providing "soft" hardware acceleration, in the form of multiple flexible vector processors, but in doing so give up the absolute performance available in true hardware solutions.

3. Field Programmable Gate Arrays

FPGAs are "soft" hardware — a required circuit is described in the programming language HDL (Hardware Description Language), and compiled. This compiled circuit description can then be downloaded into an FPGA board installed in a standard computer. Once downloaded, the circuit operates as if it had been fabricated by traditional means. This approach has been used [HL02, Knu05, Zem02] to develop custom graphics hardware at practical cost. Using an FPGA we can design and implement hardware required for production rendering and fabricate it on a development board costing no more than a standard graphics accelerator.

However as FPGAs are also re-programmable, new circuit descriptions can be download into them at run time. This allows the hardware to be tuned, not just for the generic task of rendering, but on a per scene, or per surface basis. This was noted in [SB94] which implemented a number of simple graphics routines in FPGA, but the application is limited by the problems of developing HDL descriptions of each graphics task. [SL00] and [CGT*05] propose flexible architectures, that can be more easily tuned to specific problems, but only on a per application level.

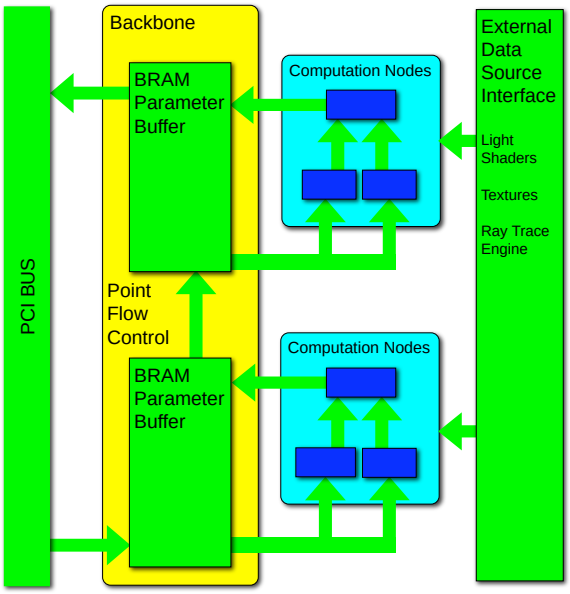

Figure 2: *The FPGA Shader Architecture*

By compiling RenderMan shaders from SL into HDL, we can create a hardware implementation of a specific shader. These can be loaded as required into the FPGA at render-time, allowing fully programmable shaders to be genuinely hardware accelerated rather than simply run on an array of dedicated processors. As SL is already a standard tool used in animation production, such a system would allow end users to reconfigure the system without knowledge of the hardware implementation.

4. Implementation

Generation of hardware descriptions is handled in multiple passes. Shaders are first compiled from high level SL into a custom assembly language, which describes the shader as a set of pipelined modules. An assembler then converts this description into HDL, combining standard modules with customised control hardware. Finally the HDL compiler turns this description into a circuit layout that can be loaded onto the FPGA board.

4.1. Hardware Architecture

Figure 2 shows an overview of the shading architecture. Shaders are constructed as a series of computation nodes connected together along a backbone of RAM and control circuitry. To describe the desired shader in HDL, the networks of these computation nodes or "shade-ops" are generated by the assembler along with a package file that customises a generic HDL backbone.

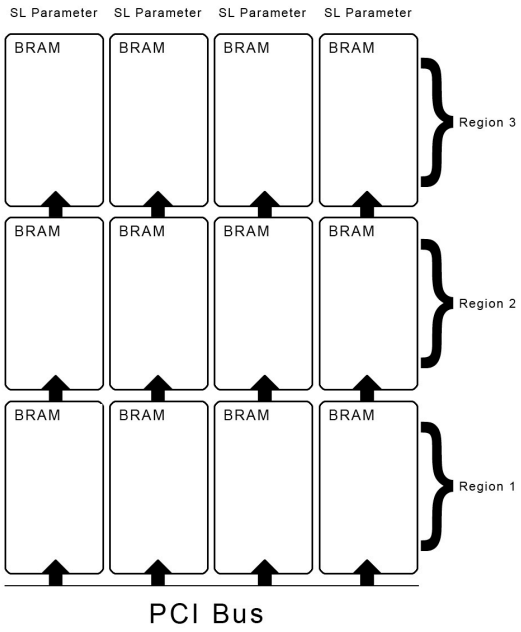

Figure 3: *Each BRAM within a region (row) holds a single SL variable for multiple MPs and acts as a circular buffer, looping the shade-ops over the parameter data. Regions can be connected together by allowing each BRAM to supply data to the one above it when a point has completed execution and space in the following becomes available.*

Providing a foundation upon which all shaders can be constructed, the backbone is composed of Block Ram (BRAM) FPGA primitives. Each BRAM stores a different RenderMan shader parameter or local variable for multiple MPs. The parameters and constants of the shader for each MP are written into these BRAMs through memory-mapped registers on the PCI bus. Therefore, one MP resides in a single memory address, with its parameters distributed between several BRAMs.

Each BRAM acts as a circular buffer. This enables the shade-ops connected to them to loop over each point's parameters. As new points are passed in they could be placed at any memory address within the BRAM. Therefore an attribute (called a label) is also held with each MP to ensure that the shade-ops operate on it in a correct order. Labels are also used to instigate a method of control flow. Once a MP has finished its execution it is transferred back over the PCI bus.

4.2. Regions

Depending on its size and complexity, a shader may require larger BRAM resources than those available in a single block on the target FPGA. In this case, the shader is divided into sequential "regions" that each feed data into the next as illustrated in Figure 3, where each column represents a single variable. Points that have completed executing on all the shade-ops in one region are shifted to the next when space is made available. This allows the pipeline to extend beyond the length of a single BRAM.

This maps extremely well to current FPGA layouts that provide distributed BRAM resources arranged in rows, such as Xilinx's Spartan3 or Virtex families, leading to faster design synthesis and better system performance. Allowing the shader to be separated in such a way makes the architecture very scalable by permitting different regions to be spread over multiple FPGAs.

In addition to providing scalability, spreading the shader over several BRAMs improves performance by reducing the number of nodes accessing each BRAM. The performance of iterative structures may also be improved, as each region is an independent circular buffer. By ensuring that each loop resides in its own region, multiple loops can be implemented without additional performance loss.

4.3. Control Flow

Control flow within the system is elegantly implemented through the use of "labels": an attribute assigned to each point as it enters into the pipeline, used to represent its level of completion. As a point progresses through the pipeline the shade-ops check its label and only operate upon it if it has a corresponding number. Once a shade-op has affected a change upon a point, it increments the point's label to the proceeding label, progressing it through the pipeline.

Through the introduction of a conditional node that can set a point's label depending on the result of a data comparison, selective constructs can be created to bypass whole branches of unwanted shade-ops. In addition, as each point remains within a circular buffer until it has completed all required operations within that region and has a valid label to progress, iterative structures can be created by forcing a point to loop until a particular condition is met. Figure 4 illustrates how these control flow structures can be created using the label system.

After finishing all of the required operations in a region, points remain circulating within the buffer until they can be moved to the next region, or in the case of the final region, retrieved through the PCI interface. Removing a point clears space for another to enter that region of the pipeline. The shaded points are passed back to a software rendering system for hiding, though implementation of a simple hardware hider within the FPGA would be possible.

4.4. External Data Sources

With the exception of the simplest shaders, it is necessary to access external data such as texture, light source and ray

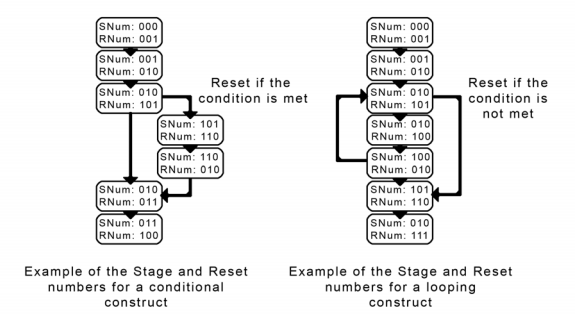

Figure 4: *Each stage (usually a network of shade-ops) in the pipeline compares the point's label to its stage number (SNum). If the label and stage number are equal, the result of the node is applied to the point's parameters and the label is set to the reset number (RNum). In doing so, the point is progressed to the next node in the pipeline, which can now operate on its parameters. The (above) illustration shows two sections of a pipeline that use the stage and reset numbers of the nodes within them to construct iterative and selective structures.*

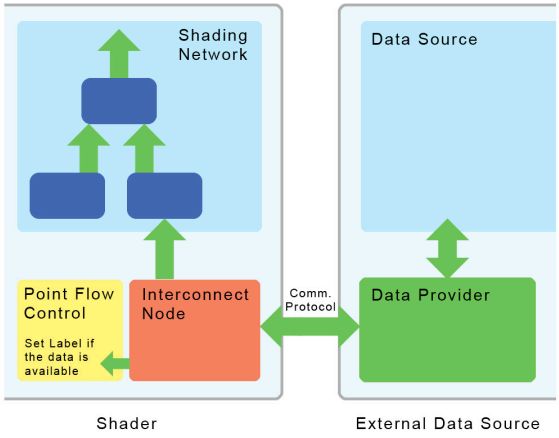

Figure 5: *The access of data external to the shader: The data is requested from the interconnect node which communicates with the external data source. If it is available, the data is supplied to the shade-ops that require it and the MP's label is incremented to the next stage. If no data is available, the MP's label is left unchanged and it is forced to wait until it is.*

traced information, in addition to a MP's parameters. To provide these features, an interface is included in the architecture to allow a uniform method of data access between the shader and any other data provider. The external interconnection works as a transparent layer between the shading networks and the external data source, providing the information to each MP as if it were simply another parameter BRAM.

When a MP attempts to read or write data from the parameter BRAM representing the external source, an interconnect node requests the data required from the external source's data provider. If the data is available, the interconnect node passes the information to the shade-ops that require it and increments the MP's label, allowing it to progress through the pipe. If the data is not available, the MP's label is not changed, effectively stalling it until it is. Figure 5 shows an illustration of the system.

To ensure the communication with a multitude of different data sources, all external data providers have a common interface with the shader. This allows them to be customised to most efficiently supply the data being requested. For example, a provider giving access to an external RAM that holds texture values may consist of a paged caching system but will use the same interface and communication protocol as a Ray Trace unit.

One problem that was immediately obvious when designing such a system was how to compensate for the variable latencies of the data providers and how to combat the negative effect that large delays would have on performance. To minimise any bottleneck created by a variable latency, an additional point cache is employed. The cache acts as a FIFO buffer into which points that are waiting for external data are branched. Upon exiting the buffer, each point has its status re-evaluated and is either permitted to continue or delayed further. When the maximum latency of the external data source is less that the size of the FIFO buffer, there is no degradation to the performance of the pipeline. This is due to the FIFO essentially being a wait state within the pipeline, which as a result, only extends its initial latency but not its throughput.

4.5. Adjacency Information

While simple surface shaders can be computed at each point using only its local parameters, more complex shaders can also require values from the local neighbourhood. One such example is a displacement shader that uses the position of two adjacent points to recalculate its normal using the cross product.

Our system implements the access of adjacency information through the use of local BRAM caches that are addressable using the absolute or relative value of a point's ID number. As each point passes through the pipeline, it can make data available globally by writing it to a BRAM cache under the address of its ID number. This provides other points with the means to access the data by either using an absolute address or a relative address generated using their own ID number offset by a variable.

As a massively pipelined SIMD processor, our system

avoids all of the structural and control hazards usually associated with traditional SISD processors. However, with the introduction of access to adjacency information, potential data hazards such as read after write, write after read and write after write are introduced.

To prevent such situations from occurring, two approaches are adopted. These are the regulated/ordered entry of points to the pipeline for linear shaders and, when iterative constructs are involved, the use of flags or semaphore bits to control data access.

For most shaders that only execute a linear control flow and can therefore guarantee in-order execution of their MPs, by regulating the order that points are entered into the pipeline we eliminate any potential race conditions and data hazards. However, it is unlikely that this will always be the case as it is common for points to be delayed due to data retrieval times and iterative structures that shuffle their orders. This problem can be solved using a pre-existing device: a condition node that checks if a semaphore bit indicating that the data has been written, has been set.

4.6. Assembler Optimizations

When a shader is compiled from SL to a hardware layout, it is the job of the assembler to generate an HDL description optimised for either speed or area. As there are several factors that can affect the execution speed of the shading engine, the assembler produces a circuit that utilises all available logic resources to increase performance unless otherwise specified.

Performance within the shading engine is measured by the number of clock cycles taken to advance a point through the pipeline. This is affected by two main factors: the number of reads/writes per BRAM and the extent to which the shade-ops are multiplexed.

An excessive number of reads or writes to a single BRAM per pipeline stage is the main cause of loss in execution speed. However, this can be combated (if resources permit) by splitting the parameter contained within the BRAM over several different regions, reducing the number of accesses per BRAM.

Excessive BRAM accesses are often caused by the writing of a variable only to be read back again several stages down the pipeline. This can be avoided (at the expense of chip capacity) by introducing a FIFO to supply the data directly to the second shade-op rather than write it back to parameter RAM. Usually, if only one other source requires the data, a FIFO is implemented, otherwise when supplying the data to several sources spread throughout the pipeline, it is more efficient to write it to RAM. An additional factor that is also considered when solving this issue is whether the target FPGA has greater shift register resources than BRAMs and how much of each remains for use.

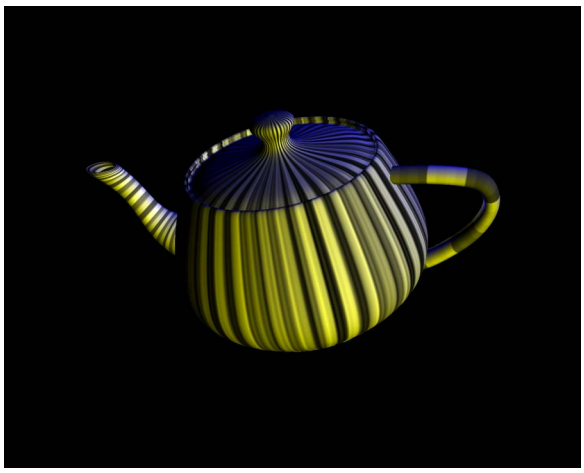

Figure 6: *The test scene shaded with 1D Noise*

The multiplexing of shade-ops is the other main cause of a decrease in performance but multiplexing produces more compact circuits at the expense of speed, allowing more complex shaders to fit within a given chip. By multiplexing the inputs and outputs of the shade-ops to increase the number of data sources that each one is able to execute upon, the number of cycles required to advance a point though the pipeline is linearly increased but the use of logic resources is reduced.

Often a compromise can be found for both shade-op multiplexing and the number of BRAM accesses per pipeline stage, as their effects on performance are not additive. Therefore the number of BRAM accesses can be increased and the logic footprint of the shading engine decreased with only a minimal decrease in performance.

5. Results

Table 1 shows the results of synthesising and executing a number of SL Shaders on a Xilinx Spartan-3 FPGA [Xil08] mounted on an EnterPoint Raggedstone1-1500 development board. A standard scene of 11,000 MicroPolygons (shown in Figure 6) was rendered using a number of simple shaders: forward-facing ratio, matte, metal and plastic, matte with 2D Perlin noise applied as a uv texture map, and a displacement based on cellnoise. Though all of the features required by more complex shaders are supported within the architecture (including conditionals, loops, derivatives and area operations), these shaders quickly exceed the 1.5 million gate capacity of the Spartan-3 FPGA.

Increasing shader complexity produces longer pipelines, which increases latency, but as the system is fully pipelined, the throughput is unaffected by this, as shown in Figure 7. The current implementation uses a clock rate of 33MHz, and uses 4 clock cycles per pipeline stage, giving a poten-

	33 mHz, 4 cycles per stage, Max Pipe Throughput is 8.25 Million MPs a second.					
	N.I	Matte	Matte and Noise	Metal	Plastic	Noise Displaced
Number of Micropolygons	11040	11040	11040	11040	11040	11040
RM Parameters (Write)	Num, N, I	Num, N, P	Num, N, P, U, V	Num, N, P, I	Num, N, P, I	Num, N,P, I, U, V, S, T
RM Parameters (Read)	Num, Cs	Num, Cs	Num, Cs	Num, Cs	Num, Cs	Num, Cs, P
Number of 32 bit writes per Micropolygon	8	8	10	11	11	15
Number of 32 bit reads per Micropolygon	5	5	5	5	5	8
Pipeline Length (Capacity)	19	115	141	270	330	102
Latency (Pipe Length * Cycles Per Stage)	76 Cycles	460 Cycles	564 Cycles	1080 Cycles	1320 Cycles	408 Cycles
Average Execution Time	0.551	0.581	0.618	0.724	0.728	0.841

Table 1: *Results*

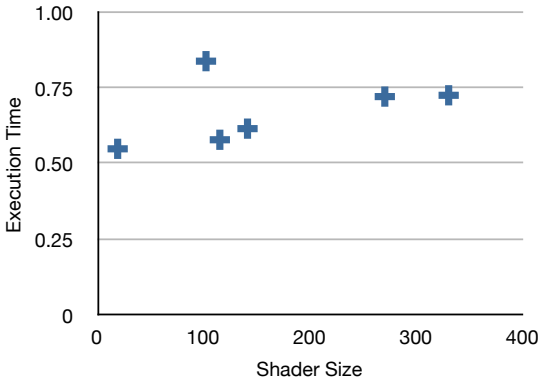

Figure 7: *Performance Vs Shader Complexity (stages)*

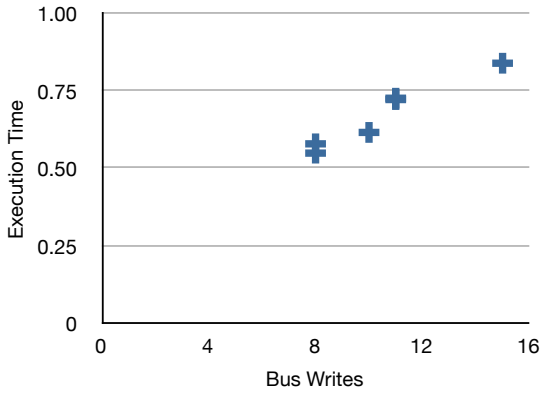

Figure 8: *Performance Vs Bus Writes*

tial throughput of 8.25 Million MP/s. FPGAs are currently available at speeds of up to 600Mhz, which would scale to 150 Million MP/s. By comparison a 2.4GHz Intel Core 2 Duo processor can evaluate Perlin noise [Per85] approxminately 16 Million times per second (approximately 150 clock cycles per evaluation). Noise is a standard building block found in SL shaders (including two of the test shaders), and any real shader would execute many operations of similar complexity in order to arrive at a final shading result.

A modern graphics accelerator might have 128 cores running at 1500Mhz. If these also used 150 clock cycles to evaluate the noise function, they would be able to evaluate 1280 Million noise functions per second. However a shader which used more than 8 functions of similar complexity would be faster on a 600MHz FPGA.

The weak correlation seen between shader size, and performance are not directly caused by the increased shader size, but rather that complex shaders typically take more parameters which must be fed to the FPGA. Though the points can be shaded at a constant rate, the parameters to each shader must first be calculated on the CPU, and written into memory mapped registers on the FPGA. The test shaders used between 8 and 15 32bit writes to set up each MP.

When the number of writes to the FPGA per MP is plotted against execution time, in Figure 8, we see that system throughput is proportional to the amount of data transferred over the PCI bus (note that though the plastic shader is more complex than the metal shader its performance is identical as they have the same number of parameters, and hence the data points are coincident on this graph). The performance of the current implementation is limited by the speed at which points can be transferred to the system of the PCI bus, rather than by the time taken to shade the points.

6. Future Work

The system is currently limited by the speed at which the parameters for MPs can be loaded into the FPGA over the PCI bus. While improved transfer methods such as block DMA would improve this, a better solution would be to generate these parameters directly on the FPGA. Most of the parameters are directly related to geometry, so only the high level per object parameters need be transferred. Even when user supplied attributes must be transferred these are at the resolution of the control hull, rather than the MP level.

While retrieving the results of the shader back from the FPGA has less impact on performance (as only a small number of parameters need be retrieved), by implementing Hiding on the FPGA, the full REYES pipeline could be could be run in hardware.

Reprogramming of the FPGA on the RaggedStone devel-

opment board used to prototype the system is slow, making it impossible to change the shader while the system is executing. Real FPGA rendering systems will need to support multiple shaders, which can be achieved by more powerful FPGA and/or the use of multiple chips. Through use of appropriate caching, and the ability to reprogram one (or one part of a) chip while another is in use it should be possible to support scenes with multiple shaders efficiently.

Standard graphics cards achieve high performance though the use of massive parallelism, using 128 or more processing elements in parallel to scale performance. In a similar fashion multiple FPGA shading engines could be operated in parallel to achieve even higher levels of performance.

7. Conclusions

Though the current system is limited by its supporting hardware, we have shown that compiling shaders to FPGA hardware is a practical method of hardware accelerating production shaders, capable of performance far beyond a CPU implementation, and with potential performance similar to current generation dedicated graphics cards.

The system's performance is independent of shader complexity, as larger shaders produce only an increase in latency.

References

[AG00] APODACA A. A., GRITZ L.: *Advanced RenderMan*. Morgan Kaufman, 2000.

[AS93] AMANATIDES J., SZURKOWSKI E.: A simple, flexible, parallel graphics architecture. In *Proc. Graphics Interface* (1993), pp. 155–160.

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The reyes image rendering architecture. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), ACM Press, pp. 95–102.

[CGT*05] CHEN J., GORDON M. I., THIES W., ZWICKER M., PULLI K., DURAND F.: A reconfigurable architecture for load-balanced rendering. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), ACM, pp. 71–80.

[Ele04] ELENDT M.: *Production Rendering*. SpringerVerlag, 2004, ch. Shading.

[HL90] HANRAHAN P., LAWSON J.: A language for shading and lighting calculations. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 289–298.

[HL02] HOLTEN-LUND H.: Fpga-based 3d graphics processor with pci-bus interface, an implementation case study. In *NORCHIP 2002 Proceedings* (2002), pp. 316–321.

[Knu05] KNUTSSON N.: *An FPGA-based 3D Graphics System*. Tech. rep., Linkopings universitet/Institutionen for systemteknik, 2005.

[MGAK03] MARK W. R., GLANVILLE R. S., AKELEY K., KILGARD M. J.: Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), ACM, pp. 896–907.

[OL98] OLANA M., LASTRA A.: A shading language on graphics hardware: The pixelflow shading system. In *Proceedings of SIGGRAPH* (1998), ACM, pp. 159–168.

[Per85] PERLIN K.: An image synthesizer. *SIGGRAPH Comput. Graph. 19*, 3 (1985), 287–296.

[PMTH01] PROUDFOOT K., MARK W. R., TZVETKOV S., HANRAHAN P.: A real-time procedural shading system for programmable graphics hardware. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM, pp. 159–170.

[POAU00] PEERCY M. S., OLANO M., AIREY J., UNGAR P. J.: Interactive multi-pass programmable shading. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 425–432.

[SB94] SINGH S., BELLEC P.: Virtual hardware for graphics applications using fpgas. In *Proceedings. IEEE Workshop on FPGAs for Custom Computing Machines* (1994), pp. 49–58.

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), ACM, pp. 1–15.

[SL00] STYLES H., LUK W.: Customising graphics applications: Techniques and programming interface. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines* (2000).

[Son01] *Emotion Engine User's Manual*. Sony Computer Entertainment Inc., 2001.

[Ste03] STEPHENSON I.: Implementing renderman on the sony ps2. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications* (New York, NY, USA, 2003), ACM, pp. 1–1.

[WGER05] WEXLER D., GRITZ L., ENDERTON E., RICE J.: Gpu-accelerated high-quality hidden surface removal. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), ACM, pp. 7–14.

[Xil08] *Spartan-3 FPGA Family Data Sheet*. Xilinx, 2008.

[Zem02] ZEMCIK P.: Hardware acceleration of graphics and imaging algorithms using fpgas. In *SCCG '02: Proceedings of the 18th spring conference on Computer graphics* (New York, NY, USA, 2002), ACM, pp. 25–32.