RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing

Sven Woop Jörg Schmittler Philipp Slusallek
Saarland University[*]

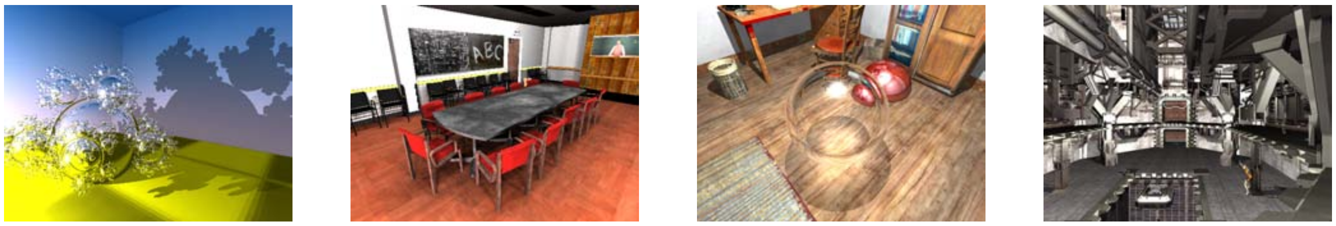


Figure 1: Realtime renderings on the RPU prototype using a single FPGA running at 66 MHz and 512×384 resolution: *SPD Balls* (1.2 fps, with shadows and refractions), a *Conference* room (5.5 fps, without shadows), reflective and refractive *Spheres-RT* in an office (4.5 fps), and *UT2003* a scene from a current computer game (7.5 fps, precomputed illumination).

Abstract

Recursive ray tracing is a simple yet powerful and general approach for accurately computing global light transport and rendering high quality images. While recent algorithmic improvements and optimized parallel software implementations have increased ray tracing performance to realtime levels, no compact and programmable hardware solution has been available yet.

This paper describes the architecture and a prototype implementation of a single chip, fully programmable Ray Processing Unit (RPU). It combines the flexibility of general purpose CPUs with the efficiency of current GPUs for data parallel computations. This design allows for realtime ray tracing of dynamic scenes with programmable material, geometry, and illumination shaders.

Although, running at only 66 MHz the prototype FPGA implementation already renders images at up to 20 frames per second, which in many cases beats the performance of highly optimized software running on multi-GHz desktop CPUs. The performance and efficiency of the proposed architecture is analyzed using a variety of benchmark scenes.

CR Categories: I.3.1 [Hardware Architecture]: Graphics processors—Parallel processing I.3.7 [Three-Dimensional Graphics and Realism]: Ray Tracing—Animation

Keywords: Ray Tracing, Hardware Architecture, Programmable Shading, Ray Processing Unit

1 Introduction

Rasterization has been the state-of-the-art technology for realtime 3D graphics for a long time and is still developing at a very fast pace. This success is mainly driven by the availability of low-cost hardware support through highly parallel graphics processing units (GPUs). The fundamental aspect of the rasterization algorithm and a major reason for its efficient hardware implementation is the purely local computation that conceptually renders a scene sequentially one triangle after another.

Unfortunately, this aspect is also the greatest weakness of rasterization as it does not allow for directly computing any global effects such as shadows, reflections, transparency, or indirect illumination. Global effects require direct access to potentially the entire scene database. As we move to more advanced, simulation-based 3D applications, efficient and easy to use support for these effects is becoming increasingly important.

The ray tracing algorithm [Appel 1968] directly *simulates* the physics of light, based on the light transport equation [Kajiya 1986] and can directly and accurately compute all global effects using ray optics. As a result ray tracing is well known for its ability to render high quality, photorealistic images. Through its simulation based approach, ray tracing supports declarative scene descriptions that are *completely* evaluated within the renderer. This greatly simplifies content creation compared to the inherently procedural interface of rasterization, where the application must handle all global effects (e.g. through precomputing texture maps).

Other important advantages of ray tracing are the “embarrassing” parallelism and scalability due to the independence of rays, its average case logarithmic complexity in terms of the number of scene primitives, and the ability to efficiently trace arbitrary rays instead of being limited to a fixed grid of pixels.

1.1 Spatial Index Structures in Rendering

In order to better compare rasterization and ray tracing it is useful to look at a slightly generalized description of the two algorithms:

Definition 1.1 (Rasterization Problem): *Given a set of rays and a primitive, efficiently compute the subset of rays that hit the primitive.*

Definition 1.2 (Ray Casting Problem): *Given a ray and a set of primitives, efficiently compute the subset of primitives that are hit by the ray.*

For rasterization the set of rays are defined by the screen pixels and primitives are triangles. The basic operation is applied to each

[*]{woop,schmittler,slusallek}@cs.uni-sb.de

triangle and a closest-triangle test (i.e. z-buffer) is performed per ray.

However, for increased performance, both rasterization and ray tracing require spatial index structures for quickly finding the respective subset of rays or triangles. Rasterization uses the regular arrangements of pixels in a 2D grid. While the grid structure greatly simplifies hardware it also limits the supported sets of rays to regular samples from a planar perspective projection. However, for advanced effects such as shadows, the set of query rays does no longer obey this restriction. Thus resampling of visibility information is required, which is prone to errors and artifacts. Little research has been performed on finding better 2D index structures to avoid this issue (but see [Aila and Laine 2004; Johnson et al. 2004]).

In addition, rasterization hardware cannot efficiently handle situations that require rendering of only a small subset of the scene, e.g. only the triangles seen through a few pixels or a particular reflection. This type of query resembles the ray tracing problem. However, rasterization hardware is missing a 3D spatial index to quickly locate the relevant triangles. Instead, the application must provide the missing functionality in software by using spatial index structures in conjunction with occlusion queries for instance. This split in the rendering process adds overhead and complexity while eliminating the options for complete hardware acceleration.

In contrast, ray tracing is fundamentally based on a 3D spatial index structure in object space. The traversal operation through this spatial index conservatively enumerates the set of triangles hit by the ray in front to back order. The index imposes no limits on the allowable set of rays and can answer even single ray queries efficiently. In most cases the spatial indices are *hierarchical* in order to better adapt to the often uneven distribution of triangles in space. Efficient hardware support for ray queries in hierarchical indices is a prerequisite for accelerated ray tracing. It would allow for fully declarative scene description by integrating the entire rendering process into hardware including any global effects.

One drawback of spatial indices in general are dynamic changes to the scene, as this would require partial or full re-computation of the index. This, however, applies to any rendering algorithm that uses a spatial index, including advanced rasterization. Little research on spatial indices for dynamic scenes has been performed in the past [Reinhard et al. 2000; Lext et al. 2000; Wald et al. 2003a].

1.2 Hardware Support for Ray Tracing

For a long time hardware support for ray tracing has been held back by three main issues: the large amount of floating point computations, support for flexible control flow including recursion and branching (necessary for traversal of hierarchical index structures and shading computations), and finally the difficulty to handle the complex memory access patterns to an often very large scene data base.

On the *software* side significant research has been performed on mapping ray tracing efficiently to parallel machines, including MIMD and SIMD architectures [Green and Paddon 1990; Lin and Slater 1991]. The key goal has been to optimally exploit the parallelism of the hardware architecture in order to achieve high floating point performance [Muuss 1995; Parker et al. 1999; Nebel 1997; Andrea Sanna and Rossi 1998; Badouel and Priol 1990; Keates and Hubbold 1995].

Realtime ray tracing performance has recently been achieved even on *single high-performance CPUs* [Wald et al. 2001; Wald et al. 2003b; Wald 2004]. However, higher resolutions, complex scenes, and advanced rendering effects still require a cluster of CPUs for realtime performance [Wald 2004].

This large number of CPUs is also the main drawback of these software solutions. The large size and cost of these solutions is preventing a more widespread adoption of realtime ray tracing. We speculate that the ray tracing performance needs to be increased by up to two orders of magnitude compared to a single CPU in order to achieve realtime, full-resolution, photorealistic rendering for the majority of current graphics applications.

One solutions could be *multi-core CPUs* announced by all the major manufacturers. However, based on the publically announced roadmaps for multi-core chips, reaching the above goal will take at least another 5 to 10 years.

On the other hand, the computational requirements of ray tracing do not require the complexity of current CPUs. Smaller hardware that satisfies the minimum requirements but allows for greater parallelism seems to be a more promising approach. First examples are ray tracing on a DSP and the simulation for the SmartMemories architecture [Greg Humphreys 1996; Mai et al. 2000; Purcell 2001].

One particularly interesting example is the use of *programmable GPUs* already available in many of today's PCs. With more than twenty SIMD units they offer excellent raw floating point performance. However, the programming model of these GPUs is still too limited and does not efficiently support ray tracing [Carr et al. 2002; Purcell 2004]. In particular, GPUs do not support flexible control flow and only very restricted memory access.

On the other extreme, multiple *custom hardware* architectures have been proposed, both for volume [Meissner et al. 1998; Pfister et al. 1999; H. Kalte and Rückert 2000] and surface models. Partial hardware acceleration has been proposed by [Green 1991] and a different implementation is commercially available [Hall 2001]. In addition a complete ray tracing hardware architectures has been simulated [Kobayashi et al. 2002]. The first complete, fully functional *realtime ray tracing chip* was presented by Schmittler et al. [Schmittler et al. 2002; Schmittler et al. 2004]. However, these specialized hardware architectures only support a fixed functionality and cannot be programmed, an essential property for advanced rendering.

2 Design Decisions

As discussed above, ray tracing is a *compute intensive, recursive, and highly parallel* algorithm with *complex control flow* requirements. The raw algorithm would perform a large number of mostly *unstructured memory accesses*, which can be greatly reduced by exploiting the *coherence* between rays. Most operations in the ray tracing algorithm are *floating point vector operations*, especially for shading.

These properties of ray tracing result in the following basic design decisions for our RPU architecture:

Vector Operations: Similar to current GPUs we use four component, single precision floating point or integer vectors as the basic data type in the core *Shader Processing Unit (SPU)*, that is used for geometry intersection and shading computations. The use of 4-vectors takes advantage of the available instruction level parallelism, results in fewer memory requests of larger size, and significantly reduces the size of shader programs compared to a scalar

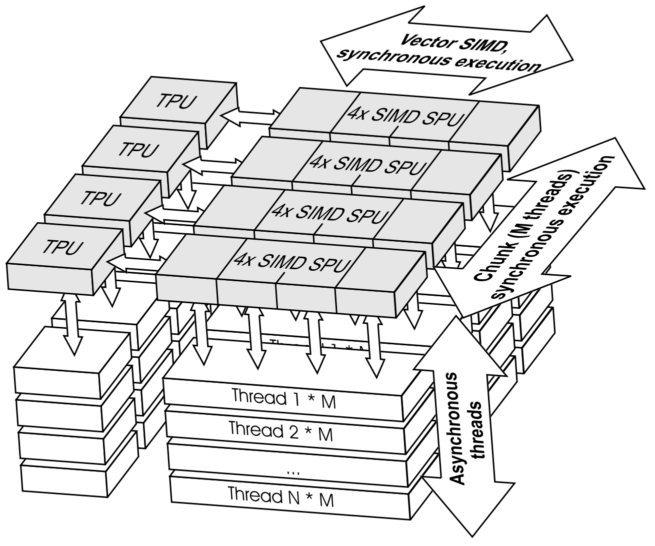


Figure 2: Multiple Shader Processing Units (SPUs) execute chunks of *M* threads synchronously in SIMD fashion. Each SPU operates on 4-component vectors as its basic data type. Multiple chunks are executed asynchronously on the multi-threaded hardware. Custom Traversal Processing Units (TPUs) synchronously traverse chunks of rays through a k-D tree but operate asynchronously to the SPUs.

program. Again similar to GPUs we implement dual-issue instructions that can operate on entire 4-vectors or split them into 2/2 or 3/1 elements as required.

Threads: We take advantage of the *data parallelism* in ray tracing through a multi-threaded hardware design. For every primary ray a new independent thread is started. The state of multiple of these threads is maintained in hardware and the execution in an SPU switches between threads as required. Multi-threading allows to increase hardware utilization by filling instruction slots that would otherwise not be used due to instruction dependencies or memory latency.

Chunks: The raw bandwidth requirement of the unmodified ray tracing algorithm is huge [Schmittler et al. 2002]. It can be reduced considerable by exploiting the high coherence between adjacent rays. To this end, we create chunks of *M* threads, where all threads are executed *synchronously in SIMD mode* in parallel by multiple SPUs, see Figure 2. Because all threads always execute the same instruction, identical memory requests are highly likely for coherent rays and can later be combined (see below). Using SIMD mode, these SPUs can share much of their infrastructure (e.g. instruction scheduling, caches), which greatly reduces the hardware complexity. The chunk size *M* and thus the number of parallel SPUs is determined by the expected coherence within a chunk and fixed for a concrete implementation of the architecture.

Control Flow and Recursion: In order to allow for complex control flow even in an SIMD environment the architecture supports conditional branching and full recursion using masked execution [Slotnick et al. 1962] and a hardware-maintained register stack [Sun Microsystems 1987]. The top most part of the register stack is easily available through the SPU register file (see Section 3.1). Recursively tracing rays from any location in a shader is required to offer maximum flexibility to shader writers. Other approaches that only allow for tail-recursion or impose other restrictions [Kajiya 1986] are too limiting for practical use.

A different control stack, hidden from the user, is used explicitly through function calls/returns and through a special `trace` instruction for recursively tracing new rays. This stack is implicitly also used for executing conditional branches, by splitting a chunk into a sub-chunk that perform the branch and one that does not. One of these sub-chunks is pushed onto the stack while waiting for the other to finish its execution.

Dedicated Traversal Units: Traversal of a ray through a k-D tree typically requires between 50 to 100 steps with scalar floating point operations. Using a fully programmable vector unit for these operations wastes precious cycles, since every step would correspond to several instructions. Instead, we provide a custom fixed function Traversal Processing Unit (TPU) that greatly improves performance and efficiency over software implementations. During traversal the TPU invokes the SPU for executing a data dependent shader for every entry encountered in the k-D tree. The TPUs share a dedicated Mailboxed List Processing Unit (MPU) that significantly reduces redundant computations by maintaining a small cache (per chunk) of previously processed entries (e.g. intersected triangles) [Amanatides and Woo 1987]. After traversal the SPUs can call a material shader for the found intersection point.

We use k-D trees as spatial index structures as they adapt very well to the scene geometry, even if the geometry is not distributed equally over the scene (good analysis has been performed by Havran [Havran 2001]). The traversal decision can be computed relatively easy by doing a substraction followed by a multiplication and some comparisons. Unfortunately, the k-D tree traversal algorithm is recursive and requires a stack which complicates the hardware. A k-D tree is a binary tree which simplifies the packet handling as at most two subchunks, one going to the left and the other one to the right of the tree, are generated.

Memory Access: Memory requests are a key problem with multi-core designs. We assume that the synchronous execution of chunks leads to many identical memory requests that can be combined and thus reduce bandwidth. Nevertheless incoherent chunks are allowed and cause no overhead but do not see improvements either. All memory accesses go through small dedicated caches in order to further reduce external bandwidth. Coherence and consequently cache hit rates are generally very high (see Section 5). Memory accesses refer to virtual memory which is mapped directly to DRAM or refers to host memory that is brought in on demand via DMA [Schmittler et al. 2003].

General Purpose Computing: The basic RPU architecture is a general purpose design. It supports random memory read and write operations as well as arbitrary address computations using integer arithmetic. However, the design has been optimized for algorithms with properties similar to those of ray tracing: *high data coherence*, *high instruction and data parallelism*, and a large number of *short vector operations*.

Scalable Design: The RPU architecture is a scalable multi-core design supporting many independent RPUs working in parallel. Each RPU contains multiple SPUs and TPUs working on coherent chunks of rays. Each RPU on a chip has its own set of caches but they all share a common memory interface. RPUs are the main units for scaling performance. Similarly to software ray tracing on clusters of CPUs, multiple RPUs can be combined within a chip and across multiple chips, boards, or even clusters of PCs.

This RPU design can be understood as a mixed thread and stream programming model for highly data parallel tasks. From a stream programming point of view [Kapasi et al. 2002], our RPU features much more general purpose kernels with full recursion and branching. Instead of many small kernels operating on fine grained streams, our kernels are more complex and execute synchronously on small “chunks” of a single “seed” stream provided by the Thread

Generator (see Figure 3). However, more complex setups can easily be realized. Another main difference between an RPU and a stream processor is the support for flexible read and write memory access.

From a thread programming perspective, an RPU is simply a general purpose parallel processor. While it would, in principal, execute arbitrary threads, best performance is achieved for threads with high computational density, which can be partitioned into coherent chunks that access similar memory and execute similar code synchronously.

3 RPU Architecture

An general overview of the RPU hardware architecture is provided in Figure 3. The Thread Generator injects new threads whenever hardware threads are available by initializing some input and address registers of all SPUs in a chunk. The initialization data can come from DMA, memory, or may be derived from a 2D grid of pixels in scan-line or Hilbert curve order. Finally, control is passed to a specified primary shader code.

Chunks of M threads are then scheduled on demand to the SPUs. When shaders execute a `trace` instruction, control is transfered to the TPUs, which synchronously traverse chunks of rays through a spatial index. While the spawning thread is suspended during traversal, the SPUs continue executing instructions asynchronously from other threads.

The TPUs synchronously traverse the entire chunk through the k-D tree. For each node in the tree the traversal decisions of all rays are computed and the chunk is split using masking bits into sub-chunks as necessary. For each sub-chunk a joint decision for the next node to traverse is computed [Wald et al. 2001]. As a result TPUs can traverse even completely incoherent chunks of rays, but would not achieve much benefit in that case.

When traversal reaches a non-empty leaf node the MPU is called to iterate through the list of entries stored there (object or primitive pointers). The MPU uses a per chunk caching mechanism to ignore entries that have been encountered before. For every remaining entry an implicit function call is performed on all corresponding threads and the threads are scheduled onto the SPUs again.

The SPUs, TPUs, and the MPU are each connected to separate first level caches. These caches are fed through the Virtual Memory Management Unit (VMU), which caches pages of memory in the DRAM [Schmittler et al. 2003]. The scene description is contained in virtual memory, which may be distributed across the host's main memory system or other RPU boards. The Memory Interface (MEM-IF) manages and optimizes access to external DDR memory chips.

3.1 SPU Registers

The SPU is the central part of our programmable ray processing unit. It supports several types of registers both for internal use and for communicating with its environment (see Table 1).

The SPU contains 16 general registers `R0` to `R15`, which are mapped to the current frame on an internal hardware stack (Figure 4). The current shader uses this stack to pass and return arguments to and from other shaders. The stack frame is incremented by an amount specified by the `trace` or `call` commands and it is restored before control flow returns. The special register, address registers, and input registers must be saved by the calling shader as required.

Name	Direction	Description
`R0 to R15`	3r, 1w	standard registers (stack frame)
`C`	1r	constant vector $(0,1,+\varepsilon,+\infty)$
`P0 to P15`	1r	per shader user parameter registers
`S`	1r, (1w)	special operation register
`A`	1r, 1w	address base registers
`I0 to I3`	1r	memory input registers

Table 1: General registers of the SPU with the available read (r) and write (w) ports. All registers are 4-component single precision floating point vectors except for the address register, which contains four integers.

An RPU chip should contain enough on-chip memory for a stack of at least 32 vector registers. This is sufficient for a recursion depth of 4, when each recursion needs 4 local vector arguments and each shader uses all 16 registers. If on-chip memory is not sufficient to store the complete register stack, the hardware automatically and speculatively writes and restores parts of the stack to and from main memory as a background task. For optimizing this speculative technique the shaders provide the size of the required stack frame.

A hard-coded "constant" vector `C` contains the frequently used values $(0,1,+\varepsilon,+\infty)$, while the values for the 16 parameter registers `P0` to `P15` are provided by the application for each shader. Memory requests are performed relative to one of the four address register components and data read from memory is always written to one or more of the input registers.

3.2 SPU Instruction Set

Because ray tracing requires the efficient execution of mostly general purpose instructions with a flexible flow of control, the design of the instruction set is a crucial design aspect.

Supported instructions are simple assignment; per component addition, multiplication, multiply and add, and computation of the fractional part; different types of dot products; integer computation; arbitrary memory reads and writes relative to an address register; and 2D addressable nearest neighbor texture reads and writes. Supported source modifiers are: swizzling, negation, and multiply with a power of 2 (0.5,1,2,4). The result of an operation can be clamped to $[0,1]$ while a write mask specifies which components need to be written to the destination. Recursive function and shader calls are supported for increased flexibility. The only ray tracing specific instruction is a `trace` instruction to access the TPU.

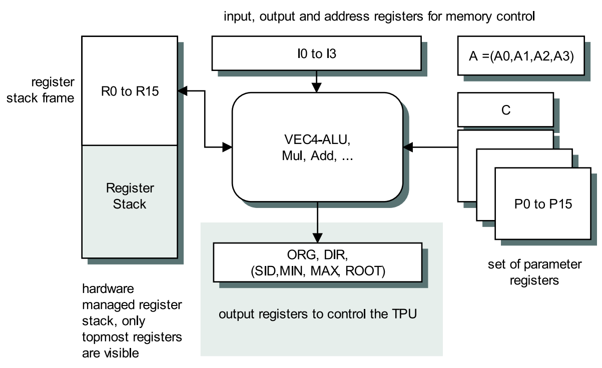

input, output and address registers for memory control
I0 to I3
A =(A0,A1,A2,A3)
register stack frame
R0 to R15
C
VEC4-ALU, Mul, Add, ...
Register Stack
P0 to P15
ORG, DIR, (SID,MIN, MAX, ROOT)
set of parameter registers
hardware managed register stack, only topmost registers are visible
output registers to control the TPU

Figure 4: Overview of the registers available to an SPU.

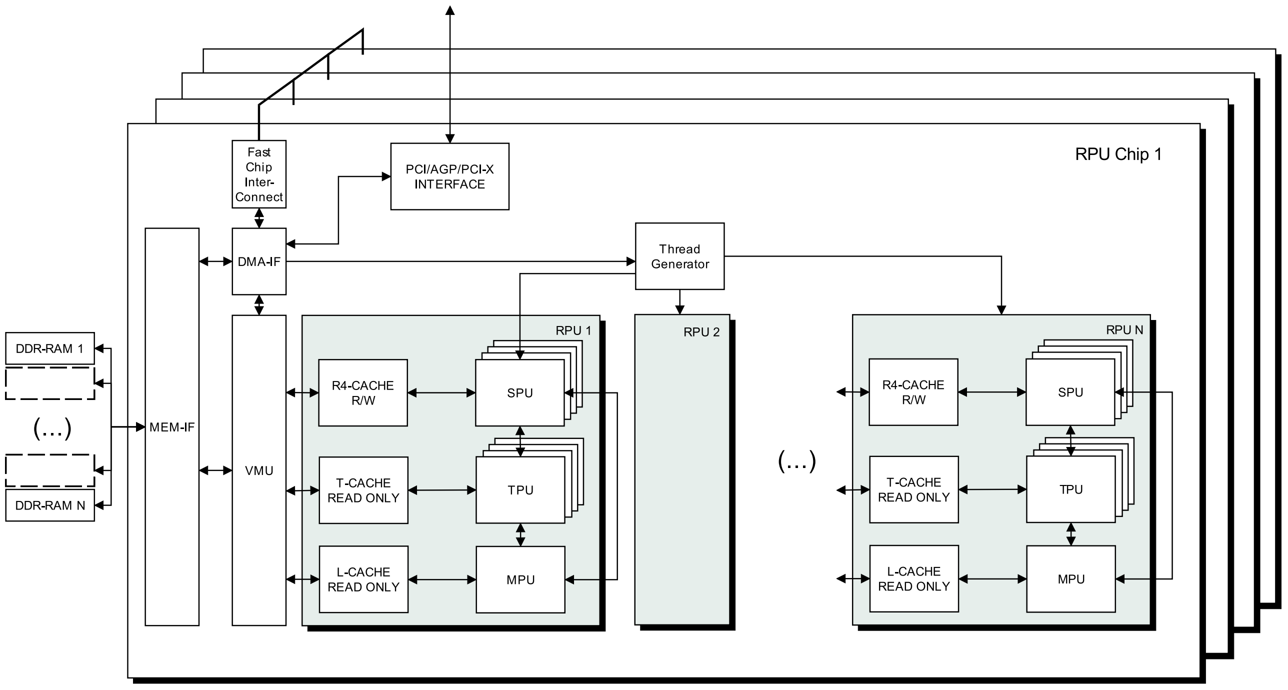
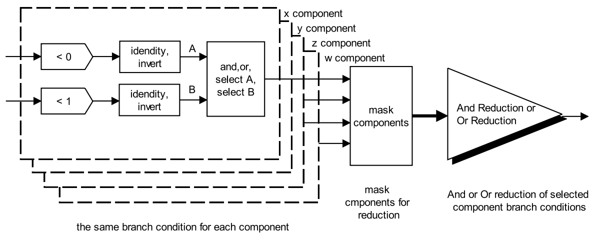


Figure 3: Each RPU chip is connected to external I/O and memory and may contain multiple independent RPU units. Each RPU contains *M* Shader Processing Units (SPU), *M* Traversal Processing Units (TPU), and one Mailboxed List Processing Unit (MPU), as well as corresponding first level caches. A central Thread Generator injects new threads/rays by itself or reads them from memory or DMA. The DMA interface can transfer data from the PCI/AGP/PCI-X interface, the Memory Interface (MEM-IF), and from neighboring RPU chips via the Fast Chip Interconnect Unit. Virtual to physical address translation and second level caching is performed by the Virtual Memory Unit (VMU).

The instruction set of the SPUs is strongly based on that of current GPUs [Nvidia 2004]. Functionality not supported by GPUs are recursive function calls, memory writes, and the `trace` instruction. On the other hand some GPU instructions, such as `SLT`, `SGT` [Nvidia 2004], have not been implemented, as we offer more flexible support for conditional branching. Except for the ability to read 4 texels of a 2D memory array at once, we provide no special hardware functionality for texture filtering such as anisotropic filtering. This gives more space for functional RPU units on the chip and texture filtering can of course be implemented in software.

Pairing: In order to improve the usage of hardware units, current CPUs provide out of order, dynamic scheduling mechanisms that submit instructions to the arithmetic units from a large instruction window. Dynamic scheduling is complex and requires significant hardware resources. Instead we choose to provide only static scheduling with two slots per instruction. An instruction word is divided into a *primary* and a *secondary* part making the SPU a kind of large instruction word (LIW) machine [Fisher 1983]. A few instructions, such as “load from texture”, require both the primary and secondary slots, while others are restricted to the secondary slot only, like the `load` instruction.

As vector SIMD units are utilized inefficiently for operations on scalars or too short vectors, we support splitting each vector into $2/2$ or $3/1$ components. The two instruction slots can then be used to execute arbitrary arithmetic operations on each sub-vector. Thus, optimizing compilers can take advantage of instruction level parallelism in shaders for improved static scheduling. An example for a valid $3/1$ pairing is `dp3 R0.xy,R1,R2 + mov R4.w,R5.w`. This instruction writes the 3 component dot product of registers R1 and R2 to the x and y component of R0 and moves the w component of R5 to the w component of R4.



x component
y component
z component
w component
< 0
< 1
idendity, invert
idendity, invert
A
B
and,or, select A, select B
mask components
And Reduction or Or Reduction
mask cmponents for reduction
And or Or reduction of selected component branch conditions
the same branch condition for each component

Figure 5: Computing a branch condition by comparing a 4D vector against a unit 4D hypercube.

Branch Instruction: The branch instruction is always paired with an arithmetic instruction. All 4 components of the arithmetic result can contribute to the branch condition. First, each component is compared to be smaller than 0 and smaller than 1 and both results can optionally be inverted. Each component then forwards either one of the results or combines them with `and` or `or`. Finally, a mask selects a subset of all component results and performs a final `and` or `or` reduction to derive the branch condition (see Figure 5). Powerful branch conditions improve the performance of the architecture as otherwise more jumps are required which causes a higher probability of splitting chunks into sub-chunks. Conditional returns are supported to further reduce the number of branches.

This branch condition can, for instance, be used for a weak in-triangle test by checking the barycentric coordinates against a bounding square. Combined with a `mad` instruction the branch condition allows for comparing a 4D point against an arbitrary 4D hypercube.

Load Instruction: The load instruction is executed asynchronously, which allows for hiding memory latency by overlapping it with other instructions. Up to four independent memory request can be outstanding per thread. An auto-increment addressing mode allows to perform a burst access of four vectors into consecutive input registers using a single `load4x` instruction. This can significantly reduce the instruction count when more than one vector needs to be read (e.g. a matrix or all triangle vertices).

A special texture addressing mode allows to address memory locations as a 2D array using two floating point coordinates. It can also load 4 adjacent texels for later bilinear interpolation in software. Texture loads support several memory formats: Full and half precision floating point vectors as well as 8 bit fixed point data.

Scheduling: The SPU does not perform any out of order execution or control flow analysis. All arithmetic instructions can be performed with throughput one and constant latency. Only instructions with special modifiers and the `load` and `trace` instructions have longer latency. The SPU continues scheduling instructions from a thread until a special *dependency bit* of the instruction indicates a dependency on previous instructions. In this case, scheduling switches to another thread and returns not before all previously scheduled instructions have been retired. We expect a compiler or assembler to automatically handle this flag.

Flexible Control Flow: It is not guaranteed, that the logical control flow in all threads of a chunk is always identical, even though this is required by the SIMD hardware. Rays hitting different geometry may execute different shaders or threads may branch to different instructions. Whenever threads in a chunk are about to execute different instructions the chunk is split into sub-chunks. Each sub-chunk maintains a mask of active threads, which determines if an SPU is actively executing instructions or simply sleeps. Only one sub-chunk is then executed while the remaining chunks are put onto a control stack until they have finished execution. Thus, all threads that have been active before are implicitly synchronized upon return of a function. This property of function calls can also be used by programmers to explicitly synchronize threads after a certain code segment.

This SIMD design assumes that there is enough coherence between threads that on average sub-chunks still maintain a high number of active threads. In the worst case performance could drop to $1/M$ in which case threads would be executed sequentially. However, for ray tracing coherence in a chunk is generally very high (see Section 5).

Special Instructions: Two complex operations are supported as special instruction modifiers: reciprocal and reciprocal square root. If attaching such an instruction modifier to an arithmetic instruction, its result is first computed normally and written to the destination. Then the special operation is applied to the 4th component of the result. The result of the special operation is replicated to all 4 components and written to the special operation destination register S using the write back mask of the command.

3.3 Example Shader

The following is an excerpt from a ray triangle intersection shader used in the prototype. It nicely demonstrates the benefits of the chosen instruction set for typical ray tracing computations. The intersection test is performed by first transforming the ray from world coordinates to a unit triangle space [Schmittler et al. 2004] where the computation is greatly simplified.

```
 1 load4x A.y,0          ; load triangle
                         ; transformation
 2 dp3_rcp R7.z,I2,R3    ; transform ray dir to
 3 dp3 R7.y,I1,R3        ; unit triangle space
 4 dp3 R7.x,I0,R3
 5 dph3 R6.x,I0,R2       ; transform ray origin to
 6 dph3 R6.y,I1,R2       ; unit triangle space
 7 dph3 R6.z,I2,R2
 8 mul R8.z,-R6.z,S.z    ; compute hit distance d
   + if z <0 return      ; and exit if negative
 9 mad R8.xy,R8.z,R7,R6 ; compute barycentric
                         ; coordinates u and v
   + if or xy            ; and return if
       (<0 or >=1)       ; hit is outside
         return          ; the bounding square
10 add R8.w,R8.x,R8.y    ; compute u+v and test
   + if w >=1 return     ; against triangle diagonal
11 add R8.w,R8.z,-R4.z   ; terminate if last hit
   + if w >=0 return     ; distance in R4.z is
                         ; closer than the new one
12 mov SID,I3.x          ; set shader ID
   + mov MAX,R8.z        ; and update MAX value
13 mov R4.xyz,R8         ; overwrite old hit data
   + return              ; and return
```

The usage of arithmetic units is particularly high for the first part of the shader during transformation and intersection. The later tests are mostly scalar operations but take perfect advantage of the two instruction slots. Once a new hit is found, the dual issue feature is again used to quickly and efficiently update the hit information.

3.4 Programming Model

The shading model necessary for ray tracing has some fundamental differences to GPU programming, due to the fact that GPUs only perform *local shading* while RPUs can compute *global shading effects* by querying the scene with secondary rays. Most shading effects possible with GPUs can be performed with the RPU architecture as well.

An important advantage of our programming model is that it completely separates geometry from shading as well as different shaders from each other. Information is transfered along rays using clearly defined interfaces. This allows us to support fully declarative scene descriptions that can be completely evaluated in hardware. Technically, this is possible through tracing rays that indirectly call other shaders based on information both from the calling shader as well as from data associated with the objects found in the scene. If calling a material shader, a shader table is specified that is indexed by the shader ID derived from the scene. Different types of rays (e.g. primary or shadow rays) would use different shader tables in order to compute different effects.

Procedural Lighting: In addition full support of recursive function calls allows for separating common functionality into additional shaders. One particularly interesting example is the separate computation of global illumination that can be called from any material shader [Slusallek and Seidel 1995; Slusallek et al. 1995]. Instead of computing illumination itself by tracing rays directly, a global *lighting shader* is called that iteratively computes any incident light contribution at the point of interest. This allows for easily modifying the lighting scenario without having to change any material shaders. Another good example are *BRDF shaders* that encapsulate the evaluation of BRDFs and separate it from normal appearance shaders that simply specify BRDF parameters [Pharr and Humphreys 2004].

Procedural Geometry: *Geometry shaders* are shaders called for any k-D tree entry encountered during traversal. This allows to render any geometric representation that has a ray intersection procedure and can be put into a spatial index structure. This would, for instance, allow for directly ray tracing of quadrics, bilinear patches, and bi-cubic splines [Benthin et al. 2004] in hardware.

Having the option of evaluating procedural geometry on demand during the rendering process offers a number of benefits. For instance, it allows for increasing the computational density [Buck et al. 2004] by reducing the amount of memory required to represent complex surfaces while increasing the amount of data parallel computation. As the gap between computational power and memory bandwidth continues to increase, such representations will become more and more important for high performance graphics.

To a limited degree these shaders can also implement the functionality of *vertex shader* by modifying input vertices. This is possible as long as all possible positions have been taken into account when building the spatial index.

Dynamic Scenes: The concept of geometry shaders also allows to elegantly implement support for dynamic scenes as suggested by [Wald et al. 2003a]. In this approach a scene is divided into objects that each have their own spatial index. These objects can then be instantiated multiple times at any position simply by providing an affine transformation. Only the top-level index structure built over the bounding boxes of all instantiated objects is dynamically updated whenever any instance is transformed. The leaf nodes of the top-level k-D tree then only contain references to object instances and their respective transformations.

A primary ray is then first traced through the top-level k-D tree while specifying a *shader table* that lists specific *geometry shaders*. Each object encountered during traversal of the top-level k-D tree contains a *shader ID*, that is used to lookup the shader table. The object shaders use its affine transformation to transform the ray into the local coordinate system of the object. The shader then spawns a new ray starting at the root of the object's k-D tree. The leaf nodes of the object's k-D tree finally contains shader IDs for geometry shaders.

The RPU architecture provides no special support for building spatial index structures. This task has to be performed by the host CPU which is possible if the number if instantiated objects is in the range of several hundreds. Supporting full dynamic scenes using position shaders for triangle vertices would require fast hardware support for building spatial index structures. Future research is needed here as naively building these spatial index structures would result in a high computational overhead.

Programmable Materials: Of course, the RPU architecture also allows for the implementation of typical material shaders, that locally compute reflected light but can take the global environment into account. These shaders are similar to previous shading systems.

Technically, materials are determined during traversal by geometry shaders that update a material shader ID, which is to be called after traversal has terminated. Before shooting a ray this material ID is set to a default shader that is called if no intersection has been found. Again geometry shaders and material shaders can easily communicate with each other through the common stack and/or preallocated thread local storage. Shaders simply need to agree on parameter passing conventions similar to traditional programming standards. All the commonly used features of programmable shading systems can easily be realized on top of this infrastructure.

Unit	fadd	fmul	frac	rcp/rsq	mem
Shader	16	16	4	4	173.6 kB
Cache	-	-	-	-	15.0 kB
Traversal	4	4	0	0	44.5 kB
Cache	-	-	-	-	11.0 kB
Mailboxed List	-	-	-	-	0.8 kB
Cache	-	-	-	-	7.0 kB
Total	20	20	4	4	251.9 kB

Table 2: Hardware complexity of the RPU prototype with a chunk size of 4, 32 hardware threads, and 512 entries for each direct mapped cache. For each unit the number of specific floating point units and the required on-chip memory is given. Dual ported memory bits are counted twice.

4 Prototype and Implementation

A fully functional prototype of the RPU architecture described above has been implemented using FPGA technology. Our prototype platform uses a Xilinx Virtex-II 6000-4 FPGA [Xilinx 2003], that is hosted on the Alpha Data ADM-XRC-II PCI-board [Alpha-Data 2003]. The FPGA has access to four 16 bit wide DDR memory chips used in parallel for a 64 bit wide memory interface. This memory interface delivers a peak memory bandwidth of 1 GB/s at 66 MHz. However, we can currently utilize only about 352 MB/s due to a very simple DRAM interface with no support for bursts, auto-precharge, and other possible optimizations. The DMA capabilities of the PCI bridge are used to upload scene data to DRAM and to download frame buffer contents to the application for storage or display via standard graphics APIs. The entire scene representation including k-D trees, shader code, and any shader parameters is downloaded from the host via DMA.

We were able to fit a single RPU onto the FPGA chip, with four SPUs (chunk size of $M = 4$) and 32 concurrent hardware threads. The MPU remembers the first 4 intersected objects and triangles the chunk has been intersected with, to prevent further intersection with them. This simple mechanism is sufficient for our test scenes. This design uses almost all FPGA logic slices (99%), about 88% of the on-chip block memories, and 20 of the block multipliers (13%). Almost all of the 48 floating point units are in the SPU unit with the remaining dedicated to the TPU (see Table 2). On the FPGA we are limited to a 24 bit floating point representation, because we had to use the custom 18 bit fixed point multipliers already available on the chip. Fully synthesizing all floating point units would have used too much FPGA logic. This limited floating point accuracy is sufficient for most scenes.

However, due to the limited size of the FPGA not all features could be enabled for the prototype: Integer operations are not included, which limits memory reads to offsets of precomputed addresses. Write support is limited to a single vector per shader (similar to GPUs). Shader length is currently limited to 512 instructions of 80 bit each, which must be resident in the RPU. Ideally, the on-chip instruction memory would just be an instruction cache. Currently, only 4 parameter registers per shader and a total of 16 different shaders are supported. A fixed register stack of 16 entries is provided with no automatic spilling to memory. Only floating point textures and 32 bit frame buffers are implemented.

Table 2 shows the number of floating point units and memory available in the prototype. Only the memory bits actually used are shown and dual ported memories are counted twice. With this configuration the hardware provides a theoretical peak performance of 2,9 GFlops at 66 MHz.

Scene	triangles	objects	SaarCOR	RPU	OpenRT	RPU / OpenRT	RPU / SaarCOR
Scene6	806	1	44.6 fps	20.8 fps	12.9 fps	1.6	0.46
Office	34 312	1	35.9 fps	14.6 fps	10.4 fps	1.4	0.40
Quake3	39 424	1	24.6 fps	12.5 fps	11.1 fps	1.1	0.51
Quake3-p	52 790	17	19.6 fps	9.7 fps	7.9 fps	1.2	0.49
UT2003	52 479	1	18.6 fps	7.5 fps	8.0 fps	0.9	0.40
Conference	282 805	54	16.2 fps	5.5 fps	8.1 fps	0.7	0.34
Castle	20 891	8	17.5 fps	2.8 fps	9.2 fps	0.3	0.16
Terrain	10 469 866	264	11.6 fps	2.2 fps	3.5 fps	0.6	0.18
SunCOR	187 145 136	5 622	23.5 fps	4.0 fps	7.5 fps	0.5	0.17
Spheres-RT	2 spheres + 15 653	4	-	4.5 fps	-		
SPD Balls	820 spheres + 12	821	-	1.2 fps	-		

Table 3: Performance of the RPU prototype for a number of benchmark scenes of varying complexity regarding the number of triangles and dynamic objects. We compare the results by providing performance numbers (frames per second) for three different platforms: the fixed function SaarCOR implementation [Schmittler et al. 2004] (performance scaled down to match the lower frequency of the RPU prototype), the fully programmable RPU (running at 66 MHz), and the highly optimized OpenRT software ray tracer running on an Intel Pentium-4 2.66 GHz [Schmittler et al. 2004]. All numbers are for an image resolution of 512×384 pixels using primary rays only (in the upper half) and with advanced shading effects (as seen in the images on the first page). All measurements include fully textured shading with sample nearest filtering.

			usage		efficiency		cache hit rate			external bandwidth	
Scene	fps	RPP	TPU	SPU	combining	chunking	TPU	MPU	SPU	absolute	usage
Scene6	20.8	1.0	46.7%	70.4%	90.7%	99.1%	98.4%	98.6%	89.8%	51.4 MB/s	16.2%
Office	14.5	1.0	40.5%	66.8%	86.7%	97.8%	95.6%	92.8%	81.4%	83.4 MB/s	24.8%
Quake3	12.5	1.0	52.4%	52.0%	84.9%	96.7%	94.0%	93.6%	81.0%	83.5 MB/s	24.5%
Quake3-p	9.7	1.0	66.4%	46.0%	85.9%	96.5%	86.7%	94.4%	81.7%	133.2 MB/s	35.4%
UT2003	7.5	1.0	39.4%	58.9%	88.0%	96.3%	88.3%	89.1%	82.2%	105.6 MB/s	30.4%
Conference	5.5	1.0	44.9%	53.7%	90.3%	95.5%	88.8%	89.4%	83.6%	100.0 MB/s	31.1%
Castle	2.8	1.0	70.3%	51.5%	94.9%	98.2%	91.9%	98.1%	92.7%	76.1 MB/s	25.1%
Terrain	2.2	1.0	30.7%	24.8%	84.0%	88.5%	57.8%	62.7%	50.1%	193.4 MB/s	60.8%
SunCOR	4.0	1.0	24.9%	20.4%	79.0%	82.1%	47.8%	31.6%	23.6%	198.6 MB/s	64.4%
Spheres-GL	11.1	1.0	57.8%	61.6%	91.4%	98.5%	96.3%	98.3%	90.1%	53.9 MB/s	16.9%
Spheres-RT	4.5	2.2	50.8%	62.7%	83.3%	96.2%	95.4%	97.5%	84.3%	136.9 MB/s	20.3%
SPD Balls	1.2	3.2	34.8%	58.2%	77.3%	79.8%	85.4%	82.1%	93.7%	73.0 MB/s	27.6%

Table 4: Detailed statistics of the RPU prototype for all benchmark scenes. From left to right it shows the overall performance in frames per second, the number of rays per pixel (RPP), and the usage values for the TPU and SPU. The chunking efficiency is the average number of threads that are expected to be active in a chunk. The memory combining efficiency is the percentage of threads active in the memory requests after the combining circuit. Cache hit rates are given for each cache in the design. Finally we provide the absolute external memory bandwidth and its usage (including control cycles). The upper 10 scenes are measured with standard shading using nearest neighbor texture lookup, while *Spheres-RT* and *SPD Balls* also use shadows and refractions on triangles and spheres.

5 Results

We have chosen a number of benchmark scenes in order to compare the performance of the RPU architecture with other approaches. The test scenes of Figures 1, 6, and Table 3 have been chosen to cover a large fraction of possible scene characteristics.

The *Scene6* and *Office* scenes are both closed room scenes of low complexity. More realistic examples are taken from computer games [Scenes 1999–2003], such as *Castle*, *UT2003*, and *Quake3*. The latter one is used for two benchmarks: as a single object, and again with several moving players. The Conference scene shows a conference room with several instantiated chairs. The highly complex *Terrain* and *SunCOR* scenes use even more instantiated geometry resulting in scenes with millions of triangles. The *Spheres* and the *SPD Balls* scenes combine triangles and spheres as geometric primitives and additionally show advanced shading effects such as reflection and refraction on curved surfaces.

5.1 Performance Comparison

Table 3 compares the performance of the RPU prototype against different hardware and software ray tracing implementations. First of all its interesting to compare a programmable ray tracing hardware to the fixed function SaarCOR prototype [Schmittler et al. 2004]. This allows us to estimate the overhead imposed by programmability. Looking at the measurements we see that the RPU prototype delivers between 20% to 50% of the total frame rate. However, one has to keep in mind that a reduced performance is well balanced by much greater functionality and flexibility.

A detailed analysis shows that the peak floating point performance of the RPU prototype is similar to that of the fixed function SaarCOR prototype. Furthermore, the fixed function design allows for better utilization as the hardware directly corresponds to the implemented algorithm. Similar to other programmable designs like CPUs and GPUs we see reduced utilization and efficiency because not all available arithmetic units can be kept busy all the time. For instance, we can only schedule instructions to a subset of all units per clock and SIMD units must

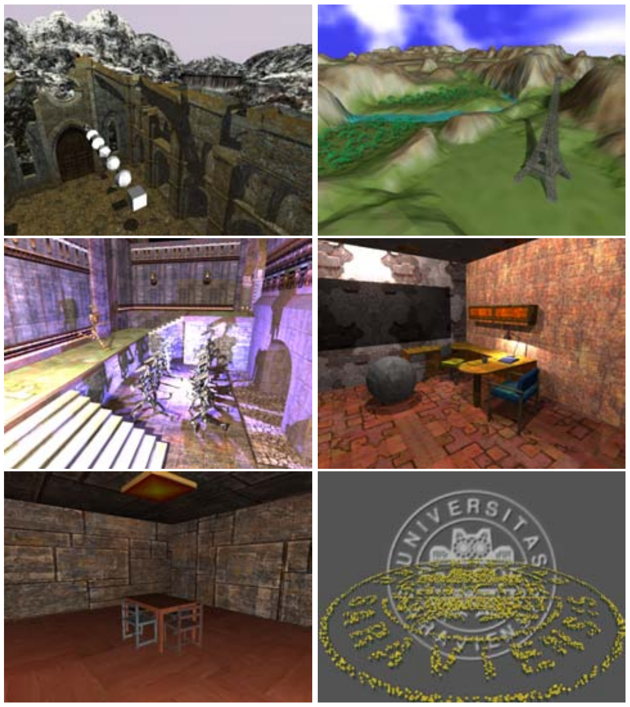


Figure 6: Some of the scenes used for benchmarking the prototype: *Castle*, *Terrain*, *Quake3-p*, *Office*, *Scene6*, and *SunCOR* (from left to right and top to bottom). More images are shown in Figure 1 and details on the complexity are given in Table 3.

sometime be used for scalar data only.

Similar to fixed function units on GPUs we see significantly increased efficiency and performance due to the fixed function TPU.

Another major difference is the much less efficient DRAM interface resulting in greatly reduced memory bandwidth for random accesses (352 MB/s) compared to the fast SRAM (1 GB/s) used in [Schmittler et al. 2004]. Different implementations of the traversal and intersection algorithms also reduce performance.

When comparing the RPU hardware to the highly optimized OpenRT software ray tracer we see similar performance levels, varying between 50% and 160% depending on the scene. Note that this performance is achieved at a 40 times slower clock rate while offering essentially the same flexibility in implementing extensions and advanced rendering features. Performance drops mainly towards more complex scenes due to the limited memory performance.

It has long been shown that realtime ray tracing in software scales extremely well to tens and even hundreds of CPUs [Wald et al. 2003b; Parker et al. 1999] even for complex shaders. Current rasterization graphics cards implement up to 16 fragment shading units that are more complex than our RPUs. As these chips are running at about 450 MHz and provide more than 30 times the external memory bandwidth than our maximal memory bandwidth one can imagine that building a chip with 27x times ($16/4 \times 450/66$) the performance of our prototype should be possible using today's chip technology. The use of multithreading and the transfer of complete cachelines from DDR memory to the caches allows to efficiently utilize the available external memory bandwidth.

A careful analysis of the usage values for the prototype show that the balancing between the TPUs and SPUs is near to optimal for small shaders. If writing complex shaders, such as the sphere shading of the *SPD Balls* scene, the more complex SPU unit becomes the bottleneck, see Table 3. Computing a finer k-d tree can fix this issue, by avoiding object intersections for the cost of more traversal operations.

5.2 Scalability

Because all pixels are independent, the number of rays and shading computations are linear in the number of pixels in an image. Higher resolution images take advantage from better coherence between rays [Schmittler et al. 2002] which improves performance. The cost of tracing rays is mostly independent of the ray type (e.g. primary or shadow ray) but shading costs can differ significantly. Increased image quality through antialiasing currently still requires super sampling. But adaptive sampling, filtered texture samples [Igehy 1999], and advanced sampling and reconstruction techniques can reduce the overhead considerably.

We can scale the parallel computation for ray tracing arbitrarily as long as we can supply the parallel units with the required bandwidth to the scene data. A major limitation of the current prototype is the suboptimal DRAM interface that does not take advantage of the burst capabilities. This would greatly increase performance for loading triangle data, which accounts for significantly more than 50% of the bandwidth compared to the traversal and list units. It needs exactly four 128-bit memory transfers from consecutive addresses, which perfectly aligns with DRAM burst requirements. In particular, this would significantly help with the large test scenes that currently show lower performance and hit rates. We believe that similar results are achievable for the other types of memory accesses as well.

Doing inter chip parallelization, as shown in Figure 3, is possible for scenes with a reasonable number of visible triangles as the RPU units are fed from primary caches. Simulations have shown that distributing the scene to memory connected independently to several FPGA chips is easily possible, when we use this memory mainly as caches [Schmittler et al. 2003]. Of course, due to the caching approach we still depend on the working set fitting into the caches. Mostly, only small changes in the set of visible scene parts must be transfered per frame, unless the camera abruptly changes the view (e.g. by walking around a corner). Still pre-fetching can be used to reduce this effect to some degree. As these virtual memory techniques always work on larger block sizes a key issue here is the spatial locality of mapping objects from 3D into memory.

We have already tested the scalability of the prototype by simultaneously using two FPGAs on separate PCI boards in the same PC. In this simple setup, each one board holds a complete copy of the scene. Similar to distributed processing on clusters of PCs [Wald et al. 2003b], this approach scales mostly independent of any scene characteristics and is only limited by the available PCI bandwidth to upload scene changes and download final pixel values.

5.3 Chunking Efficiency

As can be seen from the detailed performance analysis in Table 4, chunking is very efficient for our test scenes. Even for the complex *SunCOR* and *SPD Balls* scene the average number of active threads per clock is only reduced to about 80%. The *SunCOR* scene contains many small triangles which might suggest that rays would often be traversing and intersecting different scene components. Similarly, the *SPD Balls* scene has many adjacent pixels showing different materials resulting in the need to execute different material shaders within one chunk. Both potential problems have only a small impact on the overall performance showing that coherence between rays is sufficiently high for these complex scenes and a chunk size of 4 even on a relatively low image resolution.

The efficiency of combining memory requests of synchronously executed threads is slightly lower than the chunking efficiency since different data is read more often than different instructions are executed. For most scenes we see an efficiency between 95% and 99%, when measured as the average number of threads that participate on an external memory request (after combining).

Chunks have been used to decrease the hardware complexity, especially for the memory interface and the SPU. The usage of larger chunks, that are possibly processed sequentially, can decrease the complexity of the memory interface further. Unfortunately, this causes increasing inefficiency for incoherent computations, as then chunking and memory combining efficiency drops, limiting peak floating point performance and/or memory bandwidth. We consider a chunk size of 4 to be a good trade-off.

6 Conclusion and Future Work

The RPU architecture as presented above aims at combining the best features of CPUs, GPUs, and custom hardware in order to implement a fully programmable, parallel processor that can accelerate the entire ray tracing algorithm in hardware to achieve realtime rendering performance. The RPU's programming model closely resembles that of current GPUs but extends it significantly towards general purpose computing. This includes full support for recursion, branching, and much more general memory operations. These features are implemented in an efficient multi-level SIMD design, where SIMD is used both at the instruction level (short 4-vectors) as well as on the thread level where synchronously executed, coherent threads can significantly increase memory efficiency. Finally, each general purpose SPU has been extended by a custom k-D tree traversal unit accessible through a special SPU instruction.

With this approach we have been able to create the first fully programmable hardware for ray tracing which already achieves realtime frame rates. Although the prototype hardware is implemented using only FPGA technology and runs at a rather low clock rate of 66 MHz, it provides about the same ray tracing performance as running essentially the same algorithm on a high-performance, multi-GHz, general purpose processor. It is this high efficiency of the programmable units tightly coupled with a fixed function TPU that allows realtime performance even at very low clock rates.

Our programmable hardware maintains an efficient high utilization for its functional units through a combination of an extended SIMD execution model and a strongly multi-threaded design. The extended SIMD model combines the efficiency of SIMD with the flexibility for threads to deviate from the fixed control flow where necessary with very low overhead and still high utilization.

Multi-threading is sometimes perceived to be inefficient due to the need to maintain state for all threads in costly on-chip memory. However, our memory requirements are reasonable. Furthermore, this memory is implemented in many small and local register files distributed across the architecture, which supports efficient memory designs.

The programming model made possible by the RPU architecture offers many advantages over rasterization, directly and accurately supporting many advanced global rendering features including shadows, reflections, and refraction effects. Even more importantly, the design allows for a declarative approach to scene description, where geometry and different shaders are fully orthogonal to each other. This type of scene description greatly simplifies content creation and can be evaluated with full acceleration entirely in hardware without any support from the application.

Of course many open questions still remain. Creating spatial index structures and maintaining them across scene changes is still a challenge for highly dynamic scenes. Efficiently computing glossy reflections and performing anti-aliasing (beyond simple but costly super-sampling) is still a largely unsolved problem. We also plan to evaluate the potential of a full ASIC implementation of the RPU architecture for comparing it with today's GPUs in terms of chip area, speed, and power consumption. Finally, we want to explore the use of fast, hardware accelerated ray tracing for other applications than rendering, including general visibility queries, collision detection, and the simulations of non-optical global transport problems.

In summary, our RPU architecture demonstrates that realtime ray tracing can efficiently be implemented by a hardware architecture that is fundamentally very similar to current GPUs. We hope that fast, hardware accelerated ray tracing will eventually become widely available, as it provides a robust, easy to use, and powerful basis for advanced 3D graphics and enables interesting new applications.

References

AILA, T., AND LAINE, S. 2004. Alias-free shadow maps. In *Proceedings of EUROGRAPHICS Symposium on Rendering 2004*, Eurographics Association, 161–166.

ALPHA-DATA. 2003. ADM-XRC-II. *http://www.alphadata.uk.co*.

AMANATIDES, AND WOO. 1987. A fast voxel traversal algorithm for ray tracing. In *Proceedings of EUROGRAPHICS 1987*, 3–10.

ANDREA SANNA, P. M., AND ROSSI, M. 1998. A Flexible Algorithm for Multiprocessor Ray Tracing. Tech. rep.

APPEL, A. 1968. Some Techniques for Shading Machine Renderings of Solids. *SJCC*, 27–45.

BADOUEL, D., AND PRIOL, T. 1990. An Efficient Parallel Ray Tracing Scheme for Highly Parallel Architectures. IRISA - Campus de Beaulieu - 35042 Rennes Cedex France.

BENTHIN, C., WALD, I., AND SLUSALLEK, P. 2004. Interactive Ray Tracing of Free-Form Surfaces. In *Proceedings of Afrigraph 2004*.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. In *Proceedings of SIGGRAPH*.

CARR, N. A., HALL, J. D., AND HART, J. C. 2002. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS 2002 conference on Graphics Hardware*, Eurographics Association, 37–46.

FISHER, J. A. 1983. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the 10th Symposium on Computer Architectures*, 140–150.

GREEN, S. A., AND PADDON, D. J. 1990. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer 6*, 2, 62–73.

GREEN, S. A. 1991. Parallel processing for computer graphics. *MIT Press*, 62–73.

GREG HUMPHREYS, C. S. A. 1996. Tigershark: A hardware accelerated ray-tracing engine. Tech. rep., Princeton University.

H. KALTE, M. P., AND RÜCKERT, U. 2000. Using a dynamically reconfigurable system to accelerate octree based 3d graphics. Tech. rep., System and Circuit Technology, University of Paderborn.

HALL, D. 2001. The AR350: Today's ray trace rendering processor. In *Proceedings of the EUROGRAPHICS/SIGGRAPH workshop on Graphics Hardware - Hot 3D Session*.

HAVRAN, V. 2001. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.

IGEHY, H. 1999. Tracing ray differentials. In *SIGGRAPH*, 179–186.

JOHNSON, G. S., MARK, W. R., AND BURNS, C. A. 2004. The Irregular Z-Buffer and its Application to Shadow Mapping. Tech. rep., The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-04-09, April 15.

KAJIYA, J. T. 1986. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, vol. 20, 143–150.

KAPASI, U. J., DALLY, W. J., KHAILANY, B., OWENS, J. D., AND RIXNER, S. 2002. The Imagine Stream Processor. In *Proceedings of the IEEE International Conference on Computer Design*, 282–288.

KEATES, M. J., AND HUBBOLD, R. J. 1995. Interactive ray tracing on a virtual shared-memory parallel computer. *Computer Graphics Forum 14*, 4, 189–202.

KOBAYASHI, H., ICHI SUZUKI, K., SANO, K., AND OBA, N. 2002. Interactive Ray-Tracing on the 3DCGiRAM Architecture. In *Proceedings of ACM/IEEE MICRO-35*.

LEXT, J., ASSARSSON, U., AND MÖLLER, T. 2000. BART: A Benchmark for Animated Ray Tracing. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, May. Available at http://www.ce.chalmers.se/BART/.

LIN, T. T., AND SLATER, M. 1991. Stochastic Ray Tracing Using SIMD Processor Arrays. *The Visual Computer*, 187–199.

MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W., AND HOROWITZ, M. 2000. Smart Memories: A Modular Recongurable Architecture. *IEEE International Symposium on Computer Architecture*.

MEISSNER, M., KANUS, U., AND STRASSER, W. 1998. VIZARD II, A PCI-Card for Real-Time Volume Rendering. In *EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware*.

MUUSS, M. J. 1995. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium '95*.

NEBEL, J.-C. 1997. A Mixed Dataflow Algorithm for RayTracing on the CRAY T3E. In *Third European CRAY-SGI MPP Workshop*.

NVIDIA, 2004. http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_fragment_program2.txt.

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P. P. 1999. Interactive ray tracing. In *Interactive 3D Graphics (I3D)*, 119–126.

PFISTER, H., HARDENBERGH, J., KNITTEL, J., LAUER, H., AND SEILER, L. 1999. The VolumePro real-time ray-casting system. *Computer Graphics 33*.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically Based Rendering : From Theory to Implementation*. Morgan Kaufmann.

PURCELL, T., 2001. The SHARP Ray Tracing Architecture. SIGGRAPH course on Interactive Ray Tracing.

PURCELL, T. J. 2004. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic Acceleration Structures for Interactive Ray Tracing. In *Proceedings of the Eurographics Workshop on Rendering*, 299–306.

SCENES, 1999–2003. *Unreal Tournament 2003* by Epic Games, *Return to Castle Wolfenstein* by Activision, *Quake3-Arena* by Id-Software, and *Mafia* by Illusion Softworks.

SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. 2002. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 27–36.

SCHMITTLER, J., LEIDINGER, A., AND SLUSALLEK, P. 2003. A Virtual Memory Architecture for Real-Time Ray Tracing Hardware. *Computer & Graphics 27*, 5 (October), 693–699. ISSN 0097-8493.

SCHMITTLER, J., WOOP, S., WAGNER, D., PAUL, W. J., , AND SLUSALLEK, P. 2004. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of Graphics Hardware*.

SLOTNICK, D. L., BORCK, W. C., AND MCREYNOLDS, R. C. 1962. The SOLOMON Computer. In *Proceedings of the Fall Eastern Joint Computer Conference*, 97–107.

SLUSALLEK, P., AND SEIDEL, H.-P. 1995. Vision: An Architecture for Global Illumination Calculations. In *IEEE Transactions on Visualization and Computer Graphics, 1(1)*, 77–96.

SLUSALLEK, P., PFLAUM, T., AND SEIDEL, H.-P. 1995. Using procedural RenderMan shaders for global illumination. In *Computer Graphics Forum (Proc. of EUROGRAPHICS '95)*, 311–324.

SUN MICROSYSTEMS, 1987. The SPARC Processor. http://www.sun.com/.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3, 153–164. (Proceedings of EUROGRAPHICS).

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*.

WALD, I., PURCELL, T. J., SCHMITTLER, J., BENTHIN, C., AND SLUSALLEK, P. 2003. Realtime Ray Tracing and its use for Interactive Global Illumination. In *EUROGRAPHICS State of the Art Reports*.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

XILINX. 2003. Virtex-II. *http://www.xilinx.com*.