SoPC for 3D Point Rendering

Lars Middendorf, Felix Mühlbauer, Christophe Bobda, Georg Umlauf
Department of Computer Science
University of Kaiserslautern
Gottlieb-Daimler-Str. 48
67653 Kaiserslautern, Germany
muehlbauer,bobda,umlauf@informatik.uni-kl.de

***Abstract*—Real-time 3D visualization of objects or information becomes increasingly important in everyday life e.g. in cellular phones or mobile systems. Care should be taken in the design and implementation of 3D rendering in such embedded devices like handhelds devices in order to meet the performance requirement, while maintaining power consumption low. In this work, the design and implementation of a vertex shader on a reconfigurable hardware is presented. The main focus is placed on the efficient hardware/software partitioning of the vertex shader computation, in order to maximize the performance while maintaining a high flexibility. The resulting solution must be compatible to existing vertex shaders in oder to allow the large amount of existing program to be easylly ported to our platform. A prototype consting of a PowerPC, peripherals and some custom hardware modules is realized a on an FPGA-board. The implementation of a point rendering shows considerable speed up compared to a pure software solution.**

I. Introduction

Rendering of three-dimensional objects in real-time requires much arithmetic performance. This is a problem for embedded systems that are running at low clock speed and often lacks dedicated hardware processing modules like a floating point unit (FPU). In desktop computers, the expensive arithmetic computations related to the rendering of 3D objects are done by specialized stream processing hardware in video cards. Those cards are programmable using small programs called *shaders*. The execution of shaders is the main difference to the CPU. A new instance of the program is invoked for every primitive, vertex or pixel. There are three slightly different types of shaders for these elements. Each instance can be executed independently of the others because there is no communication possible between instances of the same type. This is advantageous when designing the hardware, because it allows the execution of an arbitrary number of instances in parallel, in order to gain the maximum computation speed. Also, pipeline technique can be used to allow some threads to feed parameters to other threads waiting for them in the pipeline. As a result the available hardware can be used more efficiently.

We developed a hardware accelerator for executing vertex shaders that is in particular useful for embedded systems, because it uses very few hardware resources, in this case FPGA slices. It is a kind of coprocessor that is directly connected to the CPU by a fast bus. The main program running on the CPU loads the shader code and all inputs into this coprocessor. While the coprocessor is running the shader, the main program accomplishes further computations in parallel until the results can be read back.

It is important to minimize the resource usage of the hardware, because the number of available slices in a FGPA is very limited. The clock speed is also very low and we have to maximize the utilization of sub-components in all cycles. The scheduling of the threads is therefore pre-calculated and stored as part of the shader code. Hence, the control-logic consists only of the program counter and a few multiplexers, that route the data flow, thus enough space is left on the device to implement floating point calculations. The multiplexer configuration is stored in a table with one row for each cycle. In this work a shader converter that generates the control table from a Direct3D9[1] vertex shader was developed. The shader converter performs the scheduling of all operation on the generated hardware unit, analyzes and optimizes the data flow and maps the calculations to operations of our ALU. Currently we can execute four threads on the ALU in parallel. This allows a speed-up of factor four compare to the software implementation of the shader.

The rest of the work is organized as follows: Section II provides the basics of vertex shader while section III introduces some work related to custom implementation of vertex shaders. Section IV-B explain our implementation. A naive co-design approach is first explained, followed by a more efficient one. Also the design decisions for the hardware software partitioning are explained. The results obtained on a prototype implemented on a Xilinx Virtex 4 evaluation platform are given in VI. Finally section VII concludes the work and provides some indication on the future directions.

II. Vertex Shader

In a rendering process, each 3D-point, also called *vertex* must traversed a set of computing stations, the *render pipeline* until the final step when it can be drawn on the display. The stations consist of a coordinate transformation (object → world, world → camera) stage, an illumination, a clipping, a projection, a scaling to screen resolution step, and finally the step to approximate the float values to integer values is performed.

To simplify spatial calculations in computer graphics *homogeneous coordinates* (x, y, z, w) are used. Transformations like *translations*, *rotations*, *scalings*, *shearings*, *projections*, etc. can be mapped to 4×4 matrices and can be combined to only one matrix by multiplying the corresponding matrices. Thus, a

transformation of a vertex by a certain list of transformations can be realized by one matrix-vector-multiplication.

For illumination calculations the dot product (scalar product) is very important, because the light intensity depends on the angle between surface normals and light sources. Normals can be transformed similar to vertices which is advantageous when filling the surface normals together with the vertices of the scene into the render pipeline to speed up processing.

In conclusion, each stage of the render pipeline executes mainly matrix and vector operations using all values which are involved like coordinates, surface normals, surface attributes, lightning parameters, etc.

There are several vertex shader versions for different hardware. We focus on implementing a subset of the smallest version 1.1 [1]. All versions use a RISC instruction set. Each instruction can read from up to three registers and write to one result register. Almost every register is 128 bits wide and stores four 32 bit floating point numbers. Hence most of the commands operate on vectors with four components. The individual components of a vector can be reordered and duplicated while reading from a register and there is a write mask for every component of the result register. This improves flexibility and allows optimizing calculations. It is for example possible to get a cross product with two instructions. Because our hardware is scalar-based, all write and swizzle-masks are free and should be used to improve performance. The shader converter analyzes the data flow for each individual component and if a result is not used, the calculation is removed on a per-component basis.

There are global and local registers. Each instance of the shader has its own set of local registers consisting of temporary and output registers. It is not allowed to write global registers which makes parallelizing possible, because there is no synchronization required and no operation in one thread depends on results calculated in another thread.

Vertex Shader 1.1 does not support jumps or subroutines. A detailed description of the instruction set can be found in the DirectX SDK [1]. Table I shows some vertex shader commands and a minimal shader that reads the vertex position and performs a vector-matrix multiplication to calculate the projective position of the vertex.

Instruction	*Description*
add	addition
sub	subtraction
dp3	3D dot product
dp4	4D dot product
mad	multiplication & addition
mul	multiplication
rcp	reciprocal
rsq	inverse square root

```
# Vertex Shader 1.1
vs_1_1
# assign vertex
dcl_position v0
# vector-matrix multipl.
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3
```

Table I
SOME VERTEX SHADER COMMANDS AND AN EXAMPLE

III. RELATED WORK

Lots of work has been done already in the domain of accelerating graphics applications utilizing FPGAs in general. Some of them are listed in [2], [3]. Often a combination of a desktop computer and a FPGA builds the computing unit. The need of 3D graphics visualization in embedded systems is still growing with the increasing spreading of mobile multimedia systems in everyday life like cellular phones and PDAs. Even the MPEG H.264 standard which is the video coding for next-generation multimedia involves rendering of 2D and 3D deformable mesh geometry [4].

The still continuing miniaturization has led to highly integrated chips and finally to so-called SoCs (system on chip). Here all components and peripherals are placed on a single chip like processors, hardware accelerators, bus and peripheral controllers and allow a PLB (printed circuit board) independent redesign or update of applications which is an important advantage.

Sohn et al. introduced a multimedia co-processor for mobile applications using an ARM-10 processor and fixed-point arithmetic [5]. The company Bitboys developed an vector graphics processor targeting for high-end multimedia cellular phones which is available as IP core for SoCs integration and can process SVG and OpenVG object data [6].

We are particular interested in a system in which custom hardware can cohabit with software. Also, the system should provide enough flexibility to ease the redesign and also allow a run-time adaptation, while maintaining the performance high and the power consumption low. The next sections explain our solution to this problem.

IV. IMPLEMENTATION

Our target platform in this project was a Xilinx Virtex 4 evaluation board featuring a Virtex4-FX12 FPGA. This FPGA contains an embedded PowerPC 405 processor, on-chip memory (BlockRAM) and miscellaneous DSP functions[7]. We use the external DDR-RAM as video frame buffer to store 3D object data. A simple system on chip with DDR-RAM controller, VGA out module and system bus needs already half of the available slices of the FPGA. Because, floating point hardware modules are expensive, we tried to avoid or reuse them as much as possible. Thus, an efficient design considering speed and chip area has to be found.

A. Basic Design

In a first design a field of 32 registers combined with an adder and a multiplier unit and an instruction memory was drawn up. This *co-processor* is directly connected to the main processor via the FCM bus, which allows to extend the native PowerPC instruction set with custom instructions that are executed by a user-defined configurable hardware accelerator.

The data words read from the BlockRAM (see Figure 1) specify which registers supply the input values for the

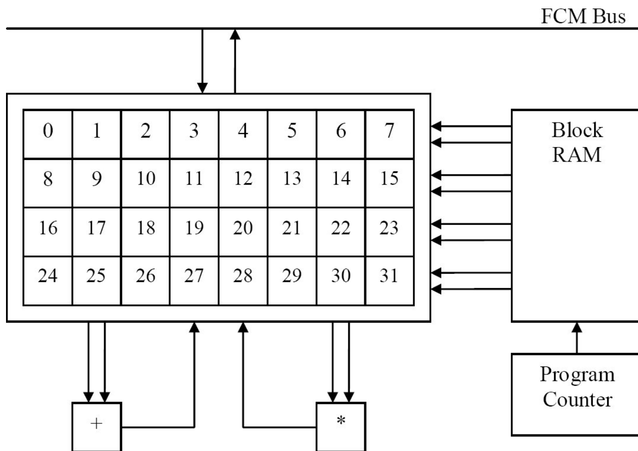

Figure 1. First design: Field of registers

arithmetic units and to which register each result should be written back.

The implementation of this design is very straight forward and also expandable for further operations like division or square root. So the two(or more) operations are executed simultaneously. Unfortunately, the design needed huge multiplexers and address decoders leading to very high resources consumption. The complete chip area was filled by this first version of the design.

B. Final Design

In order to improve the first design, the idea was to exchange the expensive registers, previously realize using the on-chip available logic (LUTs) to on chip memory, namely dual ported BlockRAM. These can hold up to 512 values each (compared to 32 for the register field) but only two read respectively write accesses are possible simultaneously. Because the dot product needs 8 input values and since only one value can be provided by a BlockRAM, we duplicated data to 8 BlockRAMs in order to be able to read eight values simultaneously. In order to keep the consistency in all 8 BlockRAMs all write request are dispatched to all 8 BlockRAMs (Figure 2). In the following this BlockRAM unit is called *register array*. This new design consumes very few slices and also provides much space for provisional results. Compared to registers a memory read access takes one clock cycle and could cause additional delays in the computation. However, due to the saving of slices, an efficient design of the ALU will compensate the lost in the BlockRAM usage (Figure 3).

We next explain the components of the final design (FCM Controller and ALU) in more detail.

1) FCM Controller: The control module implements the interface to the FCM bus. The CPU is able to write to the register array, which can hold up to 128 4D vectors, and to the instruction memory with a maximum of 512 opcodes. The FCM controller is also able to read back the results from the output memory. Because the FCM instructions to handle double words provide only 5 address bits an additional 2 bit register is used to access all 512 possible memory locations.

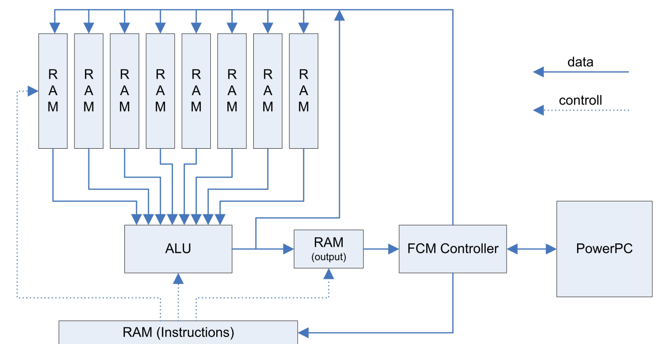

Figure 2. Final design

The final result of the shader is stored in a special output RAM that is written by the ALU and read by the CPU and the FCM control module. The output RAM is addressed independently from the register array which allows copying the shader results to an arbitrary position. The additional RAM also saves one multiplexer that, otherwise, would be needed at the address lines of the register array to switch between the ALU and the control module.

2) ALU: The ALU has eight floating-point input variables, one input port that is used to select the equation, and one output port which can be used to get the result as shown in Figure 3.

In every cycle there are nine different outputs available. One of them is selected by the output multiplexer and controlled by the current instruction code. Some results are intermediate result of longer calculations and have therefore a shorter latency. Table II lists the instruction set of the ALU. The arithmetic units use pipelining to save hardware resources and cause delays which are also shown in the table. When generating instructions for the ALU this behavior has to be taken into account. Still, the ALU can accept one set of values per clock cycle.

Every input variable can be pre-multiplied by -1 before it is read into the ALU. This is implemented as an exclusive-or between the sign bit of the floating-point number and the corresponding instruction bit. Because of delays the `slt` instruction that is used for comparisons, minimum, maximum and absolute value, the parameters e, f, g, h must be provided twice. The `rsq` command returns a rough approximation for the inverse of square root which is much more likely to be used as the square root itself, e.g. for normalization of vectors. Usually for a more precise result one step of the Newton iteration (formula: $x_{n+1} = \frac{1}{2}x_n(3 - x_0{x_n}^2)$) is sufficient [8].

3) Instruction Format: All instructions have a fixed length of 128 Bit, because of the eight input registers. The whole instruction can be divided into four words with the layout shown in Table III. The input values are read from memory at position `src*` and inverted according to `inv*`. The ALU result is stored at index `dst`, if `we` (write enable) is set and in the extra output RAM if `oe` (output enable) is set. To avoid an extra function de-multiplexer for each ALU command a selection bit was arranged (see remaining entries in Table III).

command	*result*	*delay*	*notes*
dot4	$a \cdot b + c \cdot d + e \cdot f + g \cdot h$	19	4D dot product
dot2	$a \cdot b + c \cdot d$	14	2D dot product
mult4	$a \cdot b \cdot c \cdot d$	18	multiplication
mult2	$a \cdot b$	9	multiplication
div	a/b	27	division
rsq	0x5F3759DF $-(a \gg 1)$	2	start value for newton iteration of $\frac{1}{\sqrt{a}}$
slt	if $(a \cdot b + c \cdot d < 0)$ then $(e \cdot f)$ else $(g \cdot h)$	14	input values are needed after 5 clock ticks again
int2float	$float(a)$	6	converts integer to float
float2int	$int(a)$	6	converts float to integer

Table II
ALU COMMANDS

Word	*0:8*	*9:17*	*18:26*	*27*	*28*	*29*	*30*	*31*
cmd0	src0	src1	dst	we	inv0	inv1	inv2	inv3
cmd1	src2	src3	out	oe	inv4	inv5	inv6	inv7
cmd2	src4	src5	-	div	rsq	slt	mult2	dot2
cmd3	src6	src7	-	f2i	i2f	mult4	dot4	-

Table III
INSTRUCTION FORMAT

C. *Vertex Shader Converter*

To generate the ALU opcodes a given vertex shader program is compiled with DirectX SDK[1]. Using the syntax analysis for the resulting code a data flow graph is build up, which points out the dependencies between input, provisional and output values. Now, vector operations are mapped to scalar (ALU) operations and long processing chains are move to the program start, while considering the delays caused by the arithmetic sub-units. Multiplications by -1 can be handled directly by the ALU input stage. Diversions which are not available in *Vertex Shader 1.1* and therefore are realized by multiplication with the inverse, can be processed directly by the ALU. Sometimes algebraic conversions can help to map calculations to the optimized dot product (e.g. $(a + b)c \rightarrow ac+bc$. Also the usage of the `slt` command is more practical.

V. GENERATING OPCODES

A. *Overview*

Writing the shader instructions is time-consuming and error-prone, because the format is optimized for simple decoding. Even when using an assembler that translates mnemonics into their corresponding binary format, it would be necessary to pay attention to the latency of the opcodes and the dependence between them. Therefore we developed a shader converter that reads a binary compiled Direct3D9[1] vertex shader and optimizes it for the ALU. Unfortunately it has not been possible to support all instructions, because of the limitations of the FPGA. The Direct3D9 vertex shader format has been chosen, because is well documented[9] and there are several tools like compilers, assembler and linker available[1]. The display driver gets the shader also in this format. Another possibility would have been to compile the shader directly from source code. The shader converter actually decompiles the input shader into a data flow graph and so it would have been only insignificant more complex to generate it from a simple C-style language. But by using a standard format, it is possible to develop and test shaders with Direct3D and to execute already existing shaders.

B. *Compiling the Shader*

The shader is originally written in HLSL(High Level Shading Language), which is very similar to C, except that it has build in support for special types like vectors and texture samplers. The HLSL compiler of the DirectX SDK is then used to compile it into its binary format. The compiler is available both as a standalone command line tool and as a DLL(dynamic link library), that can be used from other applications. Using this approach, the shader converter can directly process HLSL source code.

C. *Creating the Graph*

First the binary Direct3D vertex shader code is read and executed symbolically. The instructions are interpreted, but the calculations are performed on variables instead of real values. Each register stores a node of data flow graph that describes the calculations that were applied to this register. In the beginning there is usually only one node in every register, that is labeled with the register's type and index. When two registers are added, a new "'+"'-node is created with both registers as inputs. It represents the result of the calculation and is assigned to the result register for this instruction. After repeating this step for every instruction, each register contains an expression tree for its value at the end of the shader executing. There are of course node that belong to more than one tree, because of common sub-expressions. The union of all these trees is the data flow graph of the shader. It contains less information than the original instruction list, because it describes only the dependencies and not the exact order of operations. So one data flow graph is usually equivalent to a large number of programs which allows the shader converter to select the program most suitable for the ALU.

D. *Optimizations*

Except eliminating common subexpression, the shader converter does mainly architecture-dependent optimizations, because the Direct3D shader compiler already outputs code, that

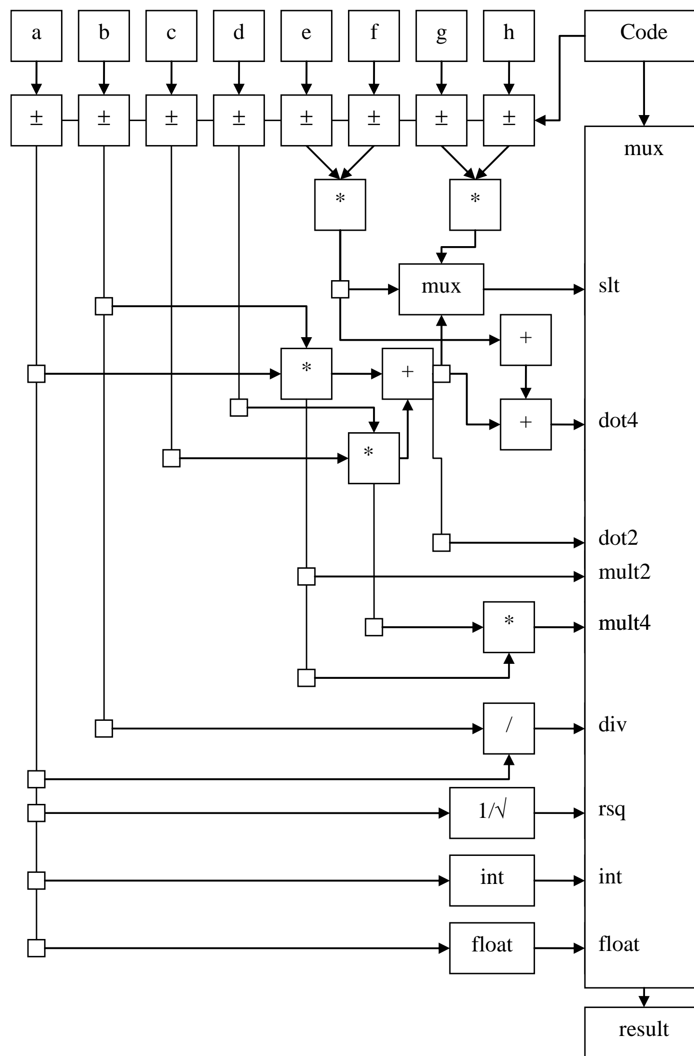

Figure 3. ALU

is very optimized, but written for a more abstract execution model. It is preferable to merge several simple command into one node, because almost every instruction takes one cycle. The ALU can multiply the inputs of every calculation by (-1) for free and so these nodes are pushed against the flow direction of the graph. This means that for an addition instead of only the output both inputs are multiplied by (-1). Although the number of nodes increases, the number of cycles remains does not and there is a higher probability that the resulting nodes can be combined into a larger calculation. There is a greedy algorithm starting at the output nodes that collects additions and multiplications as much as it is possible and creates the smallest equivalent dot2, dot4, mult2 or mult4 node (see Table II). Up to four additions are converted into a dot product. If a summand is the result of a product, the multiplication is also included, otherwise it is simply multiplied by 1, which does not create additional costs, because the multiplication is always done as part of the dot product (see Figure 4). The ALU has got a very expensive full divider, but the vertex

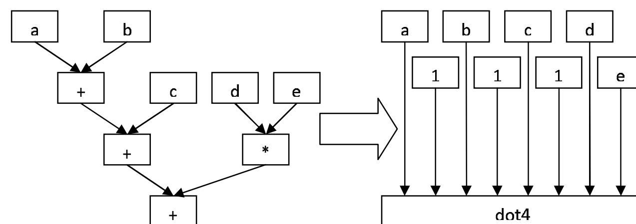

Figure 4. Optimizing an expression tree.

shader uses only reciprocals. So multiplication and reciprocals are also combined. Also the distributive law is used to convert the expression $(a+b)c$ into $ac+bc$ which seems to be more expensive, but takes one instead of two instructions, because it is implemented as the dot2 instruction.

E. Generating the Instruction Table

The instruction table is generated recursively. Each instruction is assigned to start at the first free cycle after all of its

inputs are calculated. Because the instruction bits are used directly to control the multiplexers, one logical instruction is distributed over multiply rows in the table and the parameters are delayed according to the internal pipeline of the ALU. The eight input addresses are written in the line the instruction starts, the (-1) inversion bits are written into the next line and the opcode selecting the correct output is inserted after the corresponding number of cycles (see Table II). The minimal count of cycles required to calculate the shader is bound by the length of longest path in the data flow graph. A simple optimization is to generate this path first so that optimally the first and the last instruction belongs to this path and the others can fill the gaps between caused by long latencies. Unfortunately this decreases the total length only by very few cycles. Because of the high latency there are still often large gaps of empty rows in the table where no new operations can be started and the ALU simply waits for intermediate results. To further reduce these gaps, there are always four instances of the shader executed in parallel. This is achieved by simply duplicating the nodes two times after creating the graph. Now even the large latency of the dot product(16 cycles) does not lead to empty rows in the table if there are at least four independent dot products per instance which is very common for matrix-vector multiplication. The hardware does not know about multiple threads but its simplicity allows to improve the performance significantly by generating optimal instructions.

VI. Results

The most important disadvantage of this implementation is the limitation to one result per cycle. This means that a matrix-vector multiplication takes at least four cycles. The high latency of certain operations is not really a problem, but it is different for almost every instruction, so that it can be difficult or impossible to fully load the ALU. Because of the strict requirements, not all commands could be directly implemented in hardware. For example a single `mul` instruction that multiplies two vectors component wise can take up to four cycles. But usually the instruction is part of a more complex calculation and the shader converter can merge the previous and following calculations so that the whole block may be mapped to four larger instructions that also take four cycles.

On the other hand the ALU can calculate a 4D dot product every cycle. It has been chosen to be specially optimized, because it is a very important and often used operation. Even the most simple but useful shader does a vector-matrix multiplication that can be calculated using four 4D dot products. A large number of other instructions using only multiplications and additions can be reordered and mapped to dot products. But the most important reason for the dot product is the fact that it has only one scalar result and fits perfectly to the limited register array that can only write one result value. It is also slightly cheaper than a parallel component wise multiplication and addition because it only needs three addition modules.

There is the possibility to output directly the intermediate 2D dot product and to skip the last addition for a lower latency. This can be useful when interpolating between two vectors. The additional multiplier outside of the dot product gives the ability to multiply four floating-point numbers in one cycle. This is important for calculating multi-linear functions that could otherwise only be achieved by a large number of cumbersome repeated high latency dot products.

This design cannot be enhanced any further. Adding another instruction type extends the multiplexer at the output of the ALU and leads to increased complexity. The timing constraints will not be met and the required clock speed of 100MHz cannot be achieved.

The new hardware component has been tested with a mesh viewer. The viewer is running on the PowerPC CPU, but the vertex shader can be calculated either in software or hardware to compare the performance. The triangles are not filled and the mesh is rendered as a point model (bunny model from [10]). We want to measure the speed of the vertex calculations and in a real application the expensive triangle filling would also be done in hardware. For each configuration 100 frames have been rendered several times with different point counts. The time spans are very precisely measured directly on the board with a special 64 bit register that counts the CPU cycles. The vertex shader consists of six instructions that calculate the coordinates and the lighting from a directional light source.

Comparing the results both for software and hardware it is obvious that the hardware accelerated version is much faster, see Table IV and Figure 5. The last column of Table IV contains the ratio between software and hardware performance. The ratio is higher when rendering more vertices, because there is a fixed overhead per frame for clearing the color and depth buffers.

Vertex Count	*Hardware*	*Software*	*Software/Hardware*
5000	3.607s	13.32s	3.693
10000	4.832s	24.25s	5.019
15000	6.104s	35.22s	5.770
20000	7.409s	46.26s	6.244

Table IV
Performance results for 100 frames [sec].

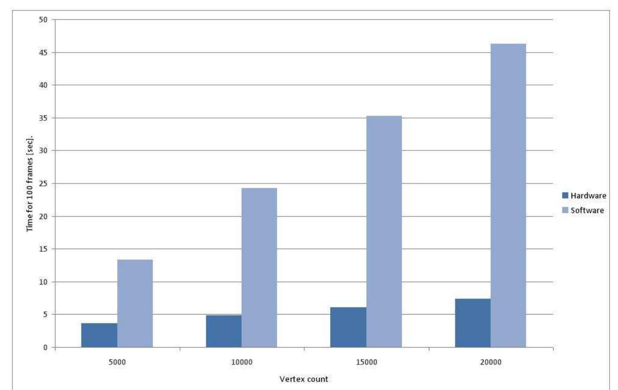

Figure 5. Performance results: Time in seconds for 100 frames and varying vertex counts

VII. Conclusion

We have introduced a hardware accelerator for a vertex shader. Our design consumes few resources (slices) on FPGA,

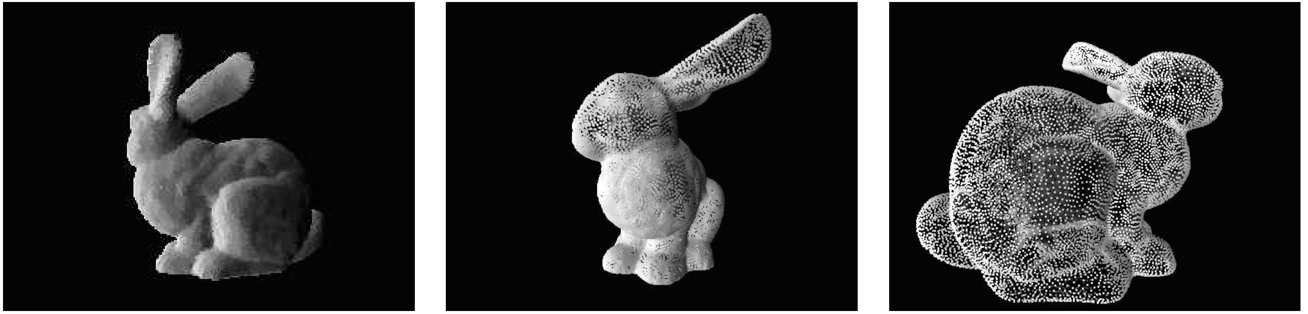

Figure 6. This bunny consist of approximately 20.000 vertices. [10].

while supporting almost all functions of the common language for such data processing *Vertex Shader 1.1*. Compared to a software only version a significant speed advantage could be achieved. This application is suitable for the domain of embedded systems.

REFERENCES

[1] Microsoft Corporation, "DirectX SDK," 09/12/2006. [Online]. Available: http://www.microsoft.com/directx

[2] D. Thomas and W. Luk, *Implementing Graphics Shaders Using FPGAs*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, p. 1173.

[3] H. Styles and W. Luk, "Customising graphics applications: Techniques and programming interface," in *IEEE Symposium on Field-Programmable Custom Computing Machines 2000*, 2000.

[4] I. Richardson, *H.264 and MPEG-4 - video compression*. Wiley, 2003.

[5] J. Y. Ju-Ho Sohn, Jeong-Ho Woo and H.-J. Yoo, "Design and test of fixed-point multimedia co-processor for mobile applications," in *DATE 2006*, 2006.

[6] symbian.com, "Bitboys introduces vector graphics processor for mobile devices at game developers conference," *www.symbian.com*, 2005.

[7] Xilinx Inc., "Virtex-4 documentation," 09/12/2006. [Online]. Available: http://www.xilinx.com/virtex4

[8] C. Lomont, "Fast inverse square root," 2003. [Online]. Available: http://www.math.purdue.edu/~clomont/Math/Papers/2003/InvSqrt.pdf

[9] Microsoft Corporation, "Windows Vista Display Driver Model Reference," 09/12/2006. [Online]. Available: http://msdn.microsoft.com

[10] L. Kobbelt, "Hauptpraktikum: Special effects SS05," 09/12/2006. [Online]. Available: http://www-i8.informatik.rwth-aachen.de/old-site/teaching/ss05/praktikum_sfx/