Design of a Parallel Accelerator for Volume Rendering

Bertil Schmidt

School of Applied Science, Nanyang Technological University, Singapore 639798,
asschmidt@ntu.edu.sg

Abstract. We present the design of a flexible massively parallel accelerator architecture with simple processing elements (PEs) for volume rendering. The underlying parallel computer model is a combination of the SIMD mesh with the instruction systolic array (ISA), an architectural concept suited for easy implementation in very high integration technology. This allows the parallel accelerator unit to be built as a programmable low cost co-processor, that suffices to render volumes with up to 16 million voxels (256^3) at 30 frames per second (fps).

1 Introduction

Volume visualisation [4] is a key technology for the interpretation of 3D scalar data generated by acquisition devices such as biomedical scanners, by supercomputer simulation, or by voxelising geometric models. Especially important for the exploration and understanding of the data are sub-second display rates and instantaneous visual feedback during change of rendering parameters. This is a challenging task due to its rigorous requirements. Firstly, the datasets are very large, typically over 16 MBytes and sometimes exceeding 150 MBytes. Secondly, to be useful the system must be able to produce images at interactive frame rates, preferably at 30 fps, but at least greater than 10 fps.

These tremendous storage and processing requirements have limited the widespread use of volume visualisation. Consequently, research has been conducted towards the development of dedicated volume rendering architectures [11,12,14]. VolumePro [12] is the first single-chip real-time accelerator for consumer PCs. However, the disadvantage of these special-purpose systems is the lack of flexibility with respect to the implementation of different algorithms, e.g. interactive segmentation, feature extraction and other tasks which are to be performed on volume datasets before rendering cannot make use of the special-purpose architecture. ISAs combine the speed and simplicity of systolic arrays with flexible programmability [6, 8], i.e. they achieve extremely high performance cost ratio and can at the same time be used for a wide range of applications, e.g. scientific computing, image processing, multimedia video compression, computer tomography, and cryptography [15-18]. Thus, the ISA architecture fits well for performing high-speed visualisation and processing of 3D datasets at low cost. In this paper we present an ISA architecture that can solve all components of a volume rendering application efficiently by taking advantage of their high degree of inherent parallelism. It has been designed in order to render volumes with up to 256^3 voxels at real-time.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1095-1104, 2000.

This paper is organised as follows. Section 2 gives an introduction to volume rendering algorithms. In Section 3 previous SIMD implementations of volume rendering are described. The concept of the ISA is explained in Section 4. Section 5 presents the new accelerator architecture. The parallel algorithms for volume rendering are explained in Section 6 and their performance is evaluated in Section 7. The outlook to further research topics concludes the paper in Section 8.

2 Volume Rendering Algorithms

Volume rendering involves the direct projection of an entire 3D dataset onto a 2D image plane. The data is sampled on a rectilinear grid, represented as a 3D array of volume elements, or *voxels*. Volume visualisation algorithms can simultaneously reveal multiple surfaces, amorphous structures, and other internal structures. These algorithms can be divided into two categories: forward-projection and backward projection. Forward projection algorithms iterate over the dataset during the rendering process projecting voxels onto the image plane. A common forward-projection algorithm is *splatting* [21]. Backward-projection iterates over the image plane during the rendering process by resampling the dataset at evenly spaced intervals along each viewing ray. *Ray casting* [10] is a common backward-projection algorithm.

In ray casting, rays are cast into the dataset. Each ray originates at the viewing position (eye), penetrates a pixel in the image plane (screen), and passes through the dataset. At evenly spaced intervals along the ray, samples are computed using interpolation. The sample values are mapped to display properties such as opacity and colour. A local gradient is combined with a local illumination model at each sample point to provide realistic shading of the object. Final pixel values are found by compositing colour and opacity values along a ray. Composition models the physical reflection and absorption of light. Ray casting offers room for algorithmic improvements by still allowing for high image quality. Several variants of traditional ray casting have been introduced, e.g. [7,23]. The modifications to the original ray casting algorithm to make it more suitable for our parallel accelerator architecture are presented in Section 6.

3 Previous SIMD Volume Rendering Work

Schröder and Stoll proposed an algorithm for the Connection Machine CM2 where the volume is stored one beam per PE. However, the inherent latency of CM2 limited their performance to 4 fps for a 128^3 volume [19]. Yoo et al. presented a method to perform volume rendering on the Pixel Planes 5 machine partly utilising the 2D SIMD mesh and partly the MIMD Graphic processors [24]. They achieved 20 fps for a 128x128x56 volume. Hsu designed a segmented ray casting approach for the DECmpp SIMD mesh [3]. However it distributed the volume in subblocks and only achieved 4-5 fps. Both Vezina [21] and Wittenbrink [23] proposed algorithms for the MASPAR MP-1 (a SIMD 8-connected mesh). Yet, neither achieved frame rates better than 2-5 fps. All of those methods suffered because of the latency inherent in large general-purpose machines. Dogett [2] presented a special-purpose architecture with a 2D array of PEs for volume rendering. However, his PEs are not programmable

ASICs. The PAVLOV design presented in [5] achieves 30 fps for a 256^3 volume on a 64x64 torus of 8-bit-parallel PEs. This architecture is close to our approach since it is a 2D mesh with simple PEs. However, its communication mechanism assumes enough memory to store two times the complete volume on-chip. This is extremely costly in terms of area requirements for PEs as compared to our design.

4 Principle of the ISA

The ISA is a quadratic array of identical processors, each connected to its four direct neighbours by data wires. The array is synchronised by a global clock. The processors are controlled by instructions, row selectors, and column selectors. The instructions are input in the upper left corner of the processor array, and from there they move step by step in horizontal and vertical direction through the array. This guarantees that within each diagonal of the array the same instruction is active during each clock cycle. In clock cycle k+1 processor (i+1,j) and (i,j+1) execute the instruction that has been executed by processor (i,j) in clock cycle k. The selectors also move systolically through the array: the row selectors horizontally from left to right, column selectors vertically from top to bottom. Selectors mask the execution of the instructions within the processors, i.e. an instruction is executed if and only if both selector bits, currently in that processor, are equal to one. Otherwise, a no-operation is executed. This construct leads to a flexible structure, which creates the possibility of very efficient solutions for a large variety of applications, e.g., numeric, image processing, video compression, and cryptography [16-19].

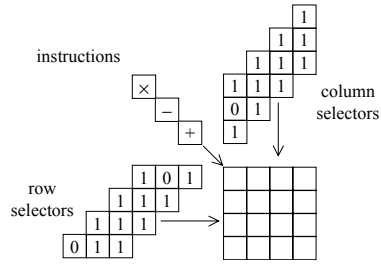

Fig. 1: Control flow in an ISA

Every processor has read and write access to its own memory. Besides that, it has a designated *communication register* (*C-register*) that can also be read by the four neighbour processors. Within each clock phase reading access is always performed before writing access. Thus, two adjacent processors can exchange data within a single clock cycle in which both processors overwrite the contents of their own C-register with the contents of the C-register of its neighbour. This convention avoids read/write conflicts and also creates the possibility to broadcast information across a whole row or column with one single instruction. This property can be exploited for an efficient calculation of row broadcasts, row ringshifts, and row sums, which are the key-operations in many algorithms.

5 Accelerator Architecture Design

Since our aim is to develop a special-purpose architecture for multimedia applications, it is highly desirable to design hardware that can be installed on PCs within the price range of a PC, i.e. to design add-on-boards for PCs. Due to experience gathered in the cause of designing and fabricating the Systola 1024 (also an add-on-board for PCs, with a 32x32 ISA built out of 16 processor chips of 64 processors each [9]), we are in the position of being able to make reliable performance predictions by extrapolation, based on the change of technology parameters. While the Systola 1024 is based on 1.0-micron technology, we are now able to use .25-micron technology, so that on the same chip area that contains 64 processor in case of Systola 1024 we can now place 1024 processors and on a single PC-board we can place 16K processors (together with memory chips, memory multiplexers and a controller with program memory).

As on-chip communication can be clocked at significantly higher frequency than chip-to-chip communication, we have decided to use the 16 processor chips relatively independently, i.e. we assume that the applications allow simple data partitioning. Chip-to-chip communication is done exclusively via locally shared memory, i.e. each processor chip is connected to a memory chip via a simple multiplexer that also allows access to the memories of the four direct neighbours (NEWS) -- here we assume a torus architecture in order to be able to perform easily horizontal and vertical ringshift of data (see Fig. 2). Thus, by avoiding direct off-chip communication we can assume an on-chip clock cycle time of 200 MHz.

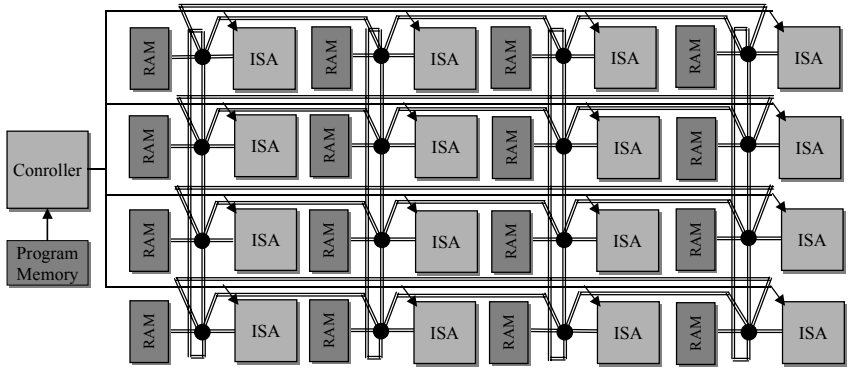

Fig. 2: Data paths of the accelerator architecture

The analysis of ray casting and its processing efforts leads to a fixpoint PE architecture (see Section 6). The PE needs a small local memory for the storage and fast supply of local voxel data. For 8-bit input voxels an intermediate operand length of 16 bits in most computations provides enough accuracy for ray casting [1]. Thus, the wordlength of the data items is set to 16 bits. To allow flexible use of the architecture the PEs must also be able to process longer operands and shorter operands efficiently, e.g. adding two 32-bit numbers in two instructions or adding two

8-bit numbers in one instruction. This idea is incorporated in the design of our computational units. Figure 3 depicts the PE architecture for volume rendering.

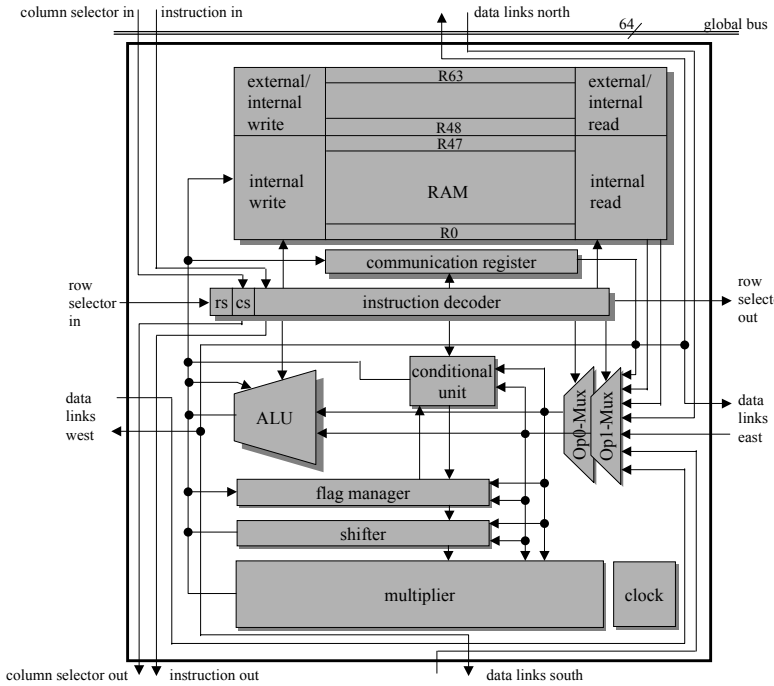

Fig. 3: Block diagram of the processor architecture

Due to the limited chip area the processor has to be very compact. This leads to our choice of a bit-serial data organisation. The bit-serial design allows a higher number of PEs per chip and a higher clock frequency than a corresponding bit-parallel design. The main components of the PE are a set of 64 data registers, a C-register, an ALU, a conditional unit, a multiplier, and a shifter. In addition to the registers there are flags (zero flag, negative flag, activation flag) that control the processing units depending on the state of the processor and several special registers. The wordlength of data items is 16 bits. Because the data is processed bit-serially, the execution of each instruction takes exactly 16 clock cycles. After receiving an instruction, the PE stores it in the instruction register, decodes the two operand addresses and the destination address, retrieves the operands from the register file, executes the instruction, writes the result back to the destination register, and passes the instruction to the next processor. The corresponding instruction set consists of 44 instructions. Since all this is done bit-serially, it can be pipelined on bit-level, such that a new instruction can be fetched and processed every 16 clock cycles.

Extrapolating the design parameters used for Systola 1024 allows us to predict that a 32x32 array of these PEs on a $1cm^2$ chip is realistic for a .25-micron CMOS process with a 200 MHz true single phase clock. For a word format of 16 bits the theoretical peak performance for one chip is 12.8 GIPS and for the complete board 204.8 GIPS.

There is already a SIMD single chip architecture with a 32x32 array of bit-serial PEs in .25-micron technology on the market [20]. But the architecture proposed in this paper achieves twice the clock frequency due to adhering as closely as possible to local communication, and its main advantage it gets through its unique control structure that allows the execution of aggregate/reduction functions in a fraction of time as compared to conventional SIMD architectures.

For the fast exchange of data with the processor array each PE has two memory banks. Each memory bank contains 8 interface registers. One of these banks is always assigned to the corresponding processor, the other to a neighbouring memory chip by means of a fast data channel. The exchange of data between ISA and the memory chip is done by bank switching. Both memory banks can be active at the same time, i.e. data transfer can be done concurrently to the execution of an ISA program.

6 Mapping of Ray Casting to the Accelerator Architecture

Fig. 4 shows three possible approaches to parallelising ray casting. According to the form of parallelism that is exploited, we call them *ray*, *beam*, and *slice parallel*.

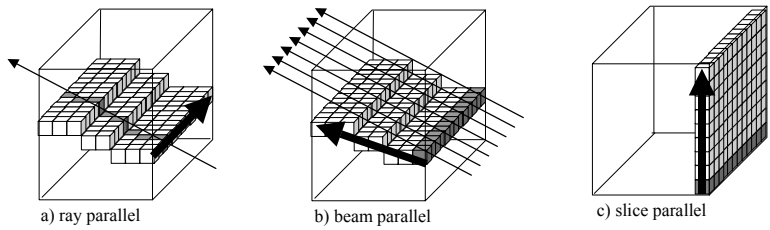

Fig. 4: Three different approaches to parallelising ray casting. Shaded voxels are processed simultaneously. The thick arrows indicate the direction in which the algorithm proceeds.

In the ray parallel approach, all voxels along a ray are processed simultaneously (the shaded voxels in Fig. 4a). The algorithm proceeds ray by ray in scanline order (the thick arrow in Fig. 4a). However, simultaneous access to all voxels along a ray requires irregular data transfer patterns between volume memory and PEs. An alternative to operating on all samples of a single ray is to simultaneously operate on samples of several neighbouring rays. Depending on how the algorithm proceeds, we call these approaches beam parallel (Fig. 4b) and slice parallel (Fig. 4c). A beam is a line of voxels that is parallel to a principle axis of the dataset. The beam parallel ray casting approach follows a group of rays by fetching consecutive beams in major viewing direction. However, the stepping along slanted planes of rays requires complicated addressing mechanisms. The slice parallel approach processes consecutive data slices that are parallel to the *base plane* of the volume dataset (Fig. 4c) and achieves a uniform data access. The base plane is the face of the volume that is closest to perpendicular to the major component of the viewing direction.

A 2D array of ISA PEs can inherently process slice order algorithms very efficiently, since an entire slice of the volume can be processed in parallel. Therefore,

we choose the slice order approach to be mapped on our architecture. Our implementation combines the slice order ray casting approach [1] with segmented ray casting [3] for parallel projections: The volume is partitioned into subcubes. These subcubes are distributed evenly across the memory modules. Each ISA chip computes the colour and opacity values of the portion of the rays, which lie inside the subblock, and writes them into its adjacent memory module. After all subcubes have been processed the segments are composited using chip-to-chip communication. The algorithm consists of the following steps:

Subcube partitioning: Determined by the memory capabilities of PEs, the size of the non-overlapping subcubes is set to 64^3. Each slice is mapped onto a 32x32 ISA by loading 2x2 voxels into each PE. As the algorithm requires a small local neighbourhood of each voxel, three slices are stored in the processor array at any time and processors at the borderline need some data from neighbouring subcubes.

Gradient estimation: The first computing step is the determination of gradients to approximate surface normals for classification and shading. *x*-, *y*-, and *z*-gradient are computed for a voxels sample value $P_{i,j,k}$ at location (i,j,k) using central differences: $G_x = P_{i+1,j,k} - P_{i-1,j,k}$, $G_y = P_{i,j+1,k} - P_{i,j-1,k}$, $G_z = P_{i,j,k+1} - P_{i,j,k-1}$. Each PE can compute gradients for its 2x2 voxel samples of the current slice in parallel using neighbouring samples. Because the processor array holds three slices at the same time, samples needed from the ahead and behind slice are stored locally in each PE. Samples needed in the two dimensions within the current slice are either also stored locally or in one of the four neighbouring PEs. Other algorithms that use larger neighbourhoods and produce higher quality gradients at additional computational costs can also be mapped efficiently on our architecture. Afterwards gradient magnitude computation continues locally by taking the sum of the squares of the gradient components and then a Newton-Raphson iteration to compute the square root of this value, resulting in an approximation of the gradient magnitude.

Classification: Classification maps a colour and opacity to sample values. Opacity values range from 0.0 (transparent) to 1.0 (opaque). On special-purpose architectures [11,12,14] classification is typically implemented using look-up tables (LUTs). These LUTs are addressed by sample value and gradient magnitude and they output sample opacity and colou. In our architecture using LUTs is not appropriate, as the local memories of PEs are very small. Thus, we are using few low degree polynomials depending on the sample value (for colour) and the product of sample value and gradient (for opacity).

Shading: The Phong shading algorithm [13] is often used in shading subsystems within volume rendering architectures. It requires gradients, light, and reflection vectors to calculate the shaded colour for each sample location. The shading calculation can be expressed as: $I = A + D\ (L*N) + S\ (R*V)^s$, where N is the (normalised) gradient vector, I is the light vector, R is the reflection vector, V is the viewing vector, A, D, L represent ambient, diffuse, and specular material components, and s is the specular exponent. The shading equation can be computed in each PE locally. To normalise the gradient vector we compute the reciprocal of the gradient magnitude by a Newton-Raphson iteration, followed by three multiplications. Parallel view and light vectors are assumed in order to make the reflection independent of the place. Thus, L and V can be stored as constants within each processor. In this case also the computation of the reflection vector can be avoided by using the halfway vector between L and V instead.

Compositing: Compositing is responsible for summing up colour and opacity contributions from interpolated sample locations along a ray into a final pixel colour for display. The front-to-back formulation for composting is $C_{acc} = (1.0 - A_{acc})\ C_{sample} + C_{acc}$ and $A_{acc} = (1.0\text{-}A_{acc})\ A_{sample} + A_{acc}$, where C_{acc} is the accumulated colour, A_{acc} is the accumulated opacity, C_{sample} is the interpolated samples colour, and A_{sample} is the interpolated samples opacity.

Since we are processing in slice order fashion, the data is sampled in each slice at the point the ray would intersect the current slice by using bilinear interpolation. Because all the points needed for bilinear interpolation are contained within the slice of voxels currently being processed, it is simpler than trilinear interpolation performed in traditional ray casting. It is also more accurate than nearest neighbour interpolation. As the algorithm moves through the dataset, the point where the ray intersects the current slices moves off the current PE position. This offset is stored and accumulated. Once the ray moves closer to another PE position, the compositing information is shifted to be stored in the corresponding neighbouring PE. In other words, the compositing information of each ray is stored in the PE closest to the ray intersection with the current slice. For parallel projections the corresponding data movement pattern is regular. Thus, whenever a ray attempts to shift to another PE, all the rays in the entire slice buffer shift together.

The rays are cut into segments by the planes that separate the subcubes. These planes are either parallel to the *x*-*y*-plane, or the *x*-*z*-plane, or the *y*-*z*-plane. We refer to these planes as *x*-*y*-planes, *x*-*z*-planes, and *y*-*z*-planes. Without loss of generality we assume that the main viewing axis is the *z*-axis. Firstly, we composite rays that pierce the *x*-*z*-plane between subblocks and then we composite rays that pierce the *y*-*z*-plane. Due to the fact that *z* is the main axis, this can be done in one step. Afterwards compositing has to happen at the *x*-*y*-planes. This can be done for *k* *x*-*y*-planes in $\log_2 k$ steps using a binary tree approach.

Finally, a 2D warp depending on the viewing vector is computed to produce the image for display. Since this is only a 2D operation, it does not influence overall computing time significantly and can be neglected.

7 Performance Evaluation

We execute ray casting within a 256^3 volume by firstly executing ray casting within 64^3 subblocks (subblock processing) and secondly compositing results of rays that move through neighbouring subblocks (final compositing). We have written a C++ cycle accurate simulation of our architecture. During subblock processing we produce the rays slice by slice. Each slice needs 1385 instructions as shown in Table 1. (Table 1 also shows the number of instructions for each substep.)

Table 1: Instruction count (IC) for the ray casting algorithm of Section 6 of one 64x64 slice of a 64^3 subblock with 8-bit voxels on a 32x32 ISA module. For intermediate operands we mostly use a length of 16 bits.

Task	Gradient	Classification	Shading	Compositing	Sum
IC	369	208	504	304	**1385**

Assuming an instruction cycle of 80 ns and computing 64 slices per subblock and 4 subblocks per ISA module, leads to total execution time of 28.4 ms for subblock processing. The data I/O for these steps (based on 150 MBytes/s throughput between each ISA module and RAM) is totally dominated by above computing time and thus can be ignored (see Section 5). Because the final composition step does not require bilinear interpolation it is dominated by the data transfer time. In the worst case (45° viewing angles) it requires 392 KByte per module. The runtime for a 256^3 volume is shown in Table 2. The processing time for larger volumes scales linear with the volume size.

Table 2: Runtime for the rendering of a 256^3 volume with 8-bit voxels on the introduced accelerator architecture. It includes computing time on the ISA and data transfer time between ISA modules and RAM.

Task	Subblock	Final Compositing	Sum
Runtime Accelerator	28.4 ms	2.9 ms	**31.3 ms**

8 Conclusions

In this paper we have presented a massively parallel architecture for volume rendering combining the SIMD computing model with the ISA concept. The accelerator unit has been designed as a co-processor to fit into an inexpensive PC class machine. The global architecture of the accelerator engine has been discussed as well as the detailed implementation of PEs. It has been shown how a volume rendering application can be mapped on the new architecture in order to render a 256^3 volume in real-time.

The introduced architecture is faster, cheaper, and smaller than previous general-purpose SIMD mesh arrays. Different from special-purpose designs, it provides more functionality, e.g. it allows multiple rendering algorithms, and, more importantly, it allows volume processing such as segmentation and feature extraction. The design will give be benefits to a medical or scientific PC where normally users wish to do more than merely render volumetric data. Future work would include identifying applications that profit from this type of processing power. For example, some users may wish to analyse the frequency of local density patterns of a volume and subsequently visualise these measurements. It would be also interesting to study the performance of the new architecture in totally different application areas like scientific computing and multimedia video processing.

References

1. Bitter, I., Kaufman, A.: A Ray-Slice-Sweep Volume Rendering Engine, *Proc. SIGGRAPH/Eurographics'97*, ACM (1997) 121-130
2. Doggett, M.: An array based design for Real-Time Volume Rendering, *Proc. Eurographics'95*, Eurographics (1995) 93-101
3. Hsu, W. M.: Segmented Ray Casting for Data Parallel Volume Rendering, *Parallel Rendering Symposium*, IEEE (1993) 7-14

4. Kaufman, A.: *Volume Visualization*, IEEE CS Press (1991)
5. Kreeger, K., Kaufman, A.: PAVLOV: A Programmable Architecture for Volume Processing, *Proc. SIGGRAPH/Eurographics'98*, ACM (1998) 77-86
6. Kunde, M., et al.: The Instruction Systolic Array and its Relation to Other Models of Parallel Computers, *Parallel Computing* 7 (1988) 25-39
7. Lacroute, P.: Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization, *IEEE Trans. on Visualization and Comp. Graphics* 2 (3) (1996) 218-231
8. Lang, H.-W.: The Instruction Systolic Array, a parallel architecture for VLSI, *Integration, the VLSI Journal* 4 (1986) 65-74
9. Lang, H.-W., Maaß, R., Schimmler, M.: The Instruction Systolic Array - Implementation of a Low-Cost Parallel Architecture as Add-On Board for Personal Computers, *Proc. HPCN 94*, LNCS 797, Springer Verlag (1994) 487-488
10. Levoy, M.: Display of Surfaces from Volume Data, *IEEE Computer Graphics and Applications* 5 (3) (1988) 29-37.
11. Meißner, M., Kanus, U., Straßer, W.: VIZARD II: A PCI-Card for Real-Time Volume Rendering, *Proc. SIGGRAPH/Eurographics'98*, ACM (1998) 61-67
12. Pfister, H., et al.: The Volume Pro Real-Time Ray-Casting System, *Proc. SIGGRAPH'99*, ACM (1999) 251-260
13. Phong: Illumination for Computer Generated Pictures, *Comm. ACM* 18(6) (1975) 311-317
14. Ray, H., et al.: Ray Casting Architectures for Volume Visualization, *IEEE Trans. On Visualization and Computer Graphics*, 5 (3) (1999) 210-223
15. Schimmler, M., Lang, H.-W.: The Instruction Systolic Array in Image Processing Applications, Proc. *Europto 96*, SPIE 2784 (1996) 136-144
16. Schmidt, B., Schimmler, M.: A Parallel Accelerator Architecture for Multimedia Video Compression, *Proc. EuroPar'99*, LNCS 1685, Springer Verlag (1999) 950-959
17. Schmidt, B., Schimmler, M., Schröder, H.: Long Operand Arithmetic on Instruction Systolic Computer Architectures and Its Application to RSA cryptography, *Proc. Euro-Par'98,* LNCS 1470, Springer Verlag (1998) 916-922
18. Schmidt, B., Schimmler, M., Schröder, H.: The Instruction Systolic Array in Tomographic Image Reconstruction Applications, *Proc. PART'98*, Springer Verlag (1998) 343-354
19. Schröder, P., Stoll, G.: Data Parallel Volume Rendering on the MasPar MP-1, *Workshop on Volume Visualization*, ACM (1992) 25-32
20. Teranex Inc.: *Parallel Processing Solves the DTV Format Conversion Problem*, http://www.teranex.com/whitepapers.html (1999)
21. Vezina G., Fletcher, P. A., Robertson, P. K.: Volume Rendering on the MasPar MP-1, *Workshop on Volume Visualization*, ACM (1992) 3-8
22. Westover, L.A.: *Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm*, PhD thesis, Dept. of Computer Science, Univ. of South Carolina in Chapel Hill (1991)
23. Wittenbrink, C.M., Somani, A.K.: Time and Space Optimal Data Parallel Volume Rendering using Permutation Warping, *Parallel and Distrib. Comp.* 46(2) (1997) 148-164
24. Yoo, T.S. et al.: Direct Visualization of Volume Data, *IEEE Computer Graphics and Applications* 12 (4) (1992) 63-71