Cube-4 Implementations on the Teramac Custom Computing Machine

Urs Kanus, Michael Meißner, Wolfgang Straßer
WSI/GRIS, University of Tübingen, Germany.

Hanspeter Pfister, Arie Kaufman
Department of Computer Science, SUNY at Stony Brook, NY, U.S.A.

Rick Amerson, Richard J. Carter, Bruce Culbertson, Phil Kuekes, Greg Snider
Hewlett-Packard Research Laboratories, Palo Alto, CA, U.S.A.

Abstract

We present two implementations of the Cube-4 volume rendering architecture on the Teramac custom computing machine. Cube-4 uses a slice-parallel ray-casting algorithm that allows for a parallel and pipelined implementation of ray-casting with tri-linear interpolation and surface normal estimation from interpolated samples. Shading, classification and compositing are part of rendering pipeline. With the partitioning schemes introduced in this paper, Cube-4 is capable of rendering large datasets with a limited number of pipelines. The Teramac hardware simulator at the Hewlett-Packard research laboratories, Palo Alto, CA, on which Cube-4 was implemented, belongs to the new class of custom computing machines. Teramac combines the speed of special-purpose hardware with the flexibility of general-purpose computers. With Teramac as a development tool we were able to implement in just five weeks working Cube-4 prototypes, capable of rendering for example datasets of 128^3 voxels in 0.65 seconds at 0.96 MHz processing frequency. The performance results from these implementations indicate real-time performance for high-resolution data-sets.

1 Introduction

Volume rendering is a key technology with increasing importance for the visualization of 3D sampled, computed, or modeled datasets. 3D volumetric data is delivered by acquisition devices such as biomedical scanners (MRI, CT) or acoustic wave devices for geophysical explorations, as well as super-computer simulations and scientific experiments, including aerodynamics, weather simulations, material tests, and many more. Volume rendering provides a powerful technique to reveal the information contained in these datasets.

The computational cost for volume rendering is very high and becomes worse for the visualization of dynamically changing datasets in real-time, a process that is called 4D (spatio-temporal) visualization. Numerous software approaches for interactive rendering, mainly based on algorithmic optimizations and large-scale parallelism, have been introduced. The highest performance for rendering of a 256^3 data set at over 10 frames per second was achieved on an expensive 16 processor SGI Challenge using the shear-warp algorithm [10]. This impressive achievement is only possible by using lengthy pre-calculations, storage of large auxiliary data structures, approximations, 2D instead of 3D interpolation, and expensive multiprocessor machines.

Providing real-time volume rendering at a reasonable cost with high image quality is the goal of special-purpose volume rendering hardware. The Cube project [9, 14, 15, 16, 13] for hardware accelerated volume rendering pioneered several volume rendering architectures using parallel rendering processors and a special interleaved memory organization to provide high processing performance and memory bandwidth.

Cube-4, the most recent approach, is a parallel and scalable architecture with modular rendering pipelines using only local and fixed bandwidth interconnections. Cube-4 is estimated to achieve real-time performance (30 frames per second) for example for a 512^3 dataset with 128 pipelines running only at 30 MHz. Cube-4 uses 3D interpolation and high-quality surface normal estimation without any precomputations or additional data storage. The performance of Cube-4 grows proportionally with increasing number of pipelines, ultimately limited only by memory speeds. The cost-performance ratio of Cube-4 is significantly better than existing solutions. The Cube-4 algorithm and dataflow have been simulated in C and VHDL.

This paper describes two prototype implementations of the Cube-4 architecture on the Teramac hardware simulator at the Hewlett-Packard research laboratories, Palo Alto, CA. Teramac belongs to a new class of machines called *custom computing machines* (CCM) which provide the user with a huge amount of programmable logic, thus combining the speed of special-purpose hardware with the flexibility of general-purpose computers.

In Section 2 we briefly describe the Cube-4 volume rendering algorithm and the rendering pipeline derived from this algorithm. We further describe a scheme for parallel volume rendering that has been implemented on the Teramac machine. We explain two partitioning schemes for rendering large volumes with a small number of rendering pipelines. Section 3 gives an overview of the Teramac hardware and software system. In Section 4 we discuss our two Cube-4 implementations on the Teramac in more detail and present results in the form of performance numbers and images.

2 Cube-4

Cube-4 implements *ray-casting*, one of the the most commonly used image-space volume rendering methods [7]. Rays are cast from the viewpoint into the volume. At evenly spaced locations along each ray a sample value is computed using surrounding voxels. A surface normal approximation for a sample point is obtained by computing the gray-level gradient [8]. The so computed surface normal together with the computed sample value is used to assign each sample a color based on a local shading model and a sample opacity. Shaded and classified sample values are composited along the rays into pixel values of the final image.

To achieve real-time performance we need to remove several bottlenecks of the ray-casting algorithm, the most important being the frequent and mostly random accesses to memory. Voxels may be addressed multiple times due to the non-uniform mapping of samples along the rays and the overlap of voxel neighborhoods during independent calculations, namely interpolation and gradient estimation. To get a one-to-one mapping of ray-samples onto voxels we use a template-based ray-casting technique first introduced by Yagel and Kaufman [17] and shown in Figure 1.

Discrete voxel rays with a constant stepping of one in major viewing direction are sent into the volume from each pixel on the base-plane, the face of the volumetric dataset that is most perpendicular to the viewing direction. After the volume has been rendered, the base-plane contains a distorted image which has to be warped and projected onto the view-plane [10, 17].

To achieve the required high memory bandwidth we use a skewed memory organization [9] that allows

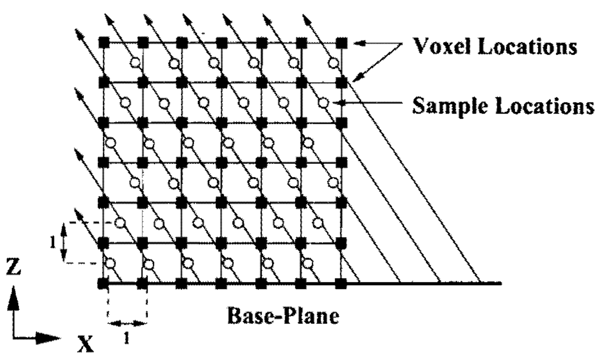

Figure 1: Base-plane for template based ray-casting.

for conflict free simultaneous access to a row of voxels (called a *beam*) parallel to one of the major viewing axis. A regular volumetric dataset with n^3 voxels is distributed over n logical memory modules, each containing n^2 words of 8 bits, using the *skewing function* $\sigma : [z,y,x] \longmapsto [k,i]$, which maps a voxel with coordinates $[z,y,x]$ (or address $[zyx]$) in *volume space* to logical memory module k at index i as follows:

$$\begin{aligned} k &= (x+y+z) \bmod n \qquad 0 \le k,x,y,z \le (n-1), \\ i &= y + z * n. \end{aligned} \tag{1}$$

Adjacent voxels of beams in X direction are placed in the same relative locations of adjacent memory modules, (i.e., rows across the memory). This choice of X direction storage is arbitrary. For all following descriptions, we choose Z as the major direction and beams in X-direction. For the five physical memory banks used, a *partitioning function* $\phi : [k,i] \longmapsto [k_p, i_p]$ has to be applied, which partitions the beams in X directions, thus re-mapping the skewed memory space as follows:

$$\begin{aligned} k_p &= k \bmod p, \\ i_p &= i\frac{n}{p} + \left\lfloor \frac{k}{p} \right\rfloor . \end{aligned} \tag{2}$$

For example, Figure 2a shows a 4^3 dataset in its local coordinate system, each voxel represented by its address a, which is the $[zyx]$ tuple of the local coordinates of the voxel. Figure 2b shows the dataset stored in $p = 4$ memory modules and Figure 2c shows the arrangement for $p = 2$.

For real-time performance the ray-casting algorithm needs to be parallelized. In Cube-4 we implement a form of parallelism called *slice-parallel* processing. During ray-casting, the volume is traversed by processing beams along consecutive slices parallel to the base-plane. The conceptual dataflow of slice-

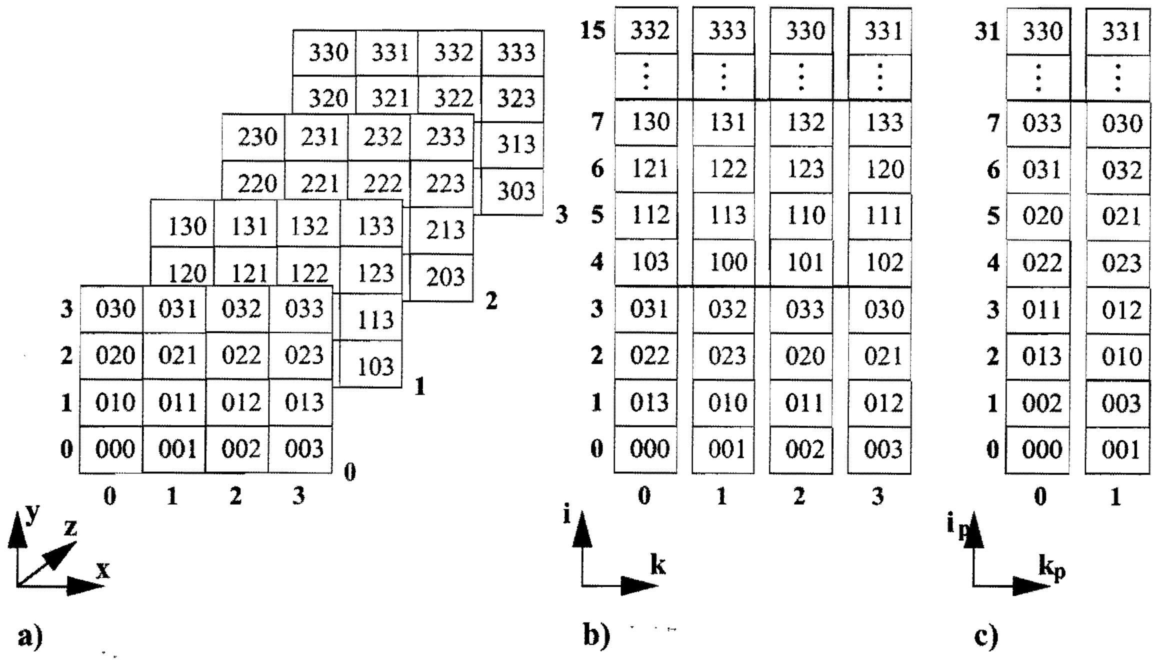
Figure 2: Memory organization for a $4 \times 4 \times 4$ dataset. a) Volume space b) $p = 4$ memory modules c) $p = 2$ memory modules.

parallel ray-casting is shown in Figure 3. Two consecutive slices are required for tri-linear interpolation. To reduce the number of memory accesses, the previously fetched slice is stored in a plane buffer (FIFO) so that it can be retrieved without further access to the voxel memory. The gradient is computed using samples from three slices of interpolated samples [14]. The two previously calculated slices of interpolated sample are stored in FIFO plane-buffers, delaying them by n and $2n$ cycles, respectively. After shading and classification each slice is composited onto the intermediate results of the previous slices, yielding the final base-plane image after n^2-cycles.

The slice-parallel approach discussed so far operates on beams of n voxels, thus requiring n processing elements, where n is the dimension of the dataset. This leads to an undesirable amount of hardware and limits the maximum size of datasets. To render datasets of size n^3 with $p < n$ processing elements, we developed two different approaches called *sub-volume partitioning* and *beam partitioning*.

In sub-volume partitioning, a volumetric dataset of size n^3 is divided into smaller sub-volumes of resolution p, each being processed by p pipelines. The images of each subvolume are combined to yield the final image. Our first prototype implementation on Teramac, described in Section 4, uses sub-volume partitioning.

However, this first prototype revealed two main problems with this approach. First, the voxel neighborhood required for tri-linear interpolation and gradient estimation at sub-volume boundaries can only be provided by overlap of sub-volumes. As Table 1 shows, this results in substantial memory overhead, which leads to higher execution time (see Section 6). The second problem is that rays can traverse multiple

Rendering pipelines p	Memory overhead in percent Subvolume size $p \times p \times 128$
8	61
16	34
32	18
64	10

Table 1: Memory overhead in percent due boundary-voxel overlap for sub-volume partitioning of a 128^3 dataset.

sub-volumes for non-orthogonal viewing directions, as illustrated in Figure 4. The intermediate compositing results for rays that cross the sub-volume boundary have to be stored in a buffer so that they can be accessed during processing of the next sub-volume. The order in which the sub-volumes have to be processed depends on the viewing direction and the compositing order (front-to-back or back-to-front). To access the

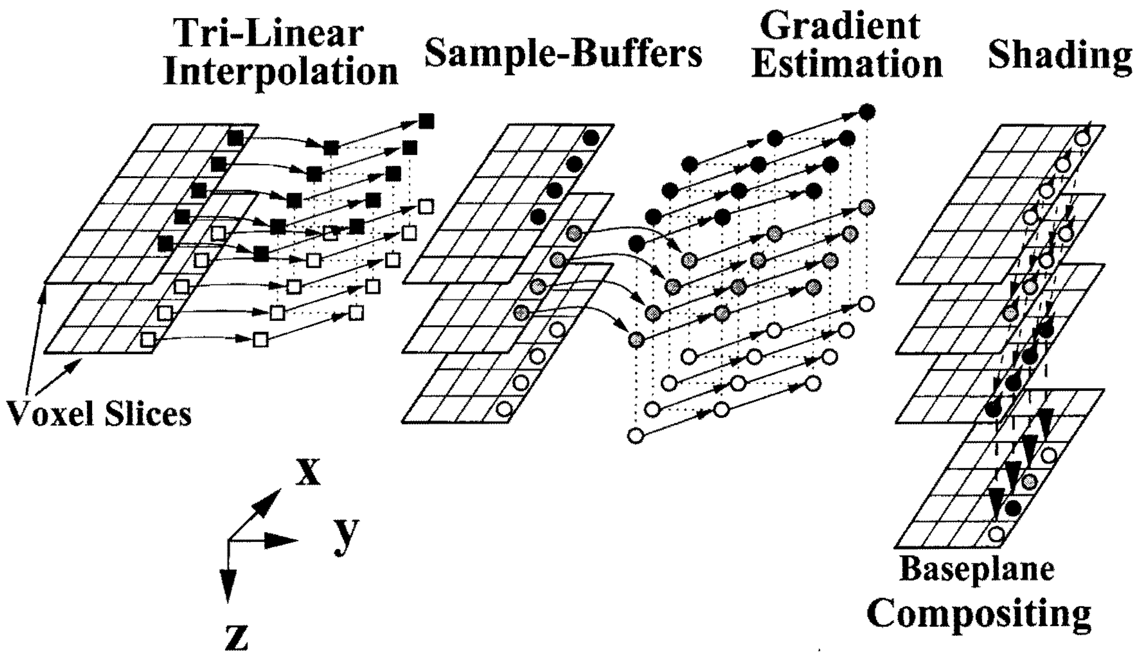

Figure 3: Dataflow of slice-parallel ray-casting.

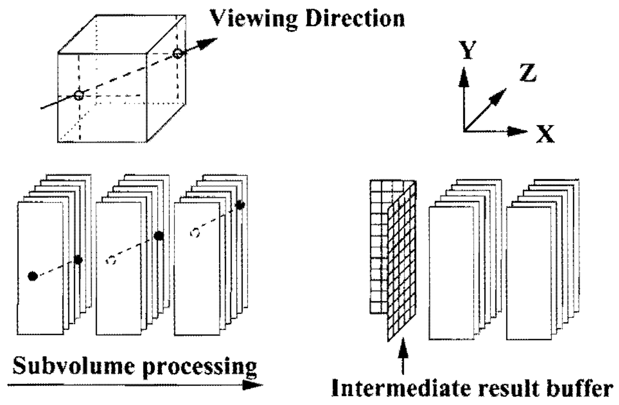

Figure 4: Sub-volume processing order for front-to-back compositing and given viewing direction. Intermediate results at sub-volume boundaries have to be stored for subsequent processing.

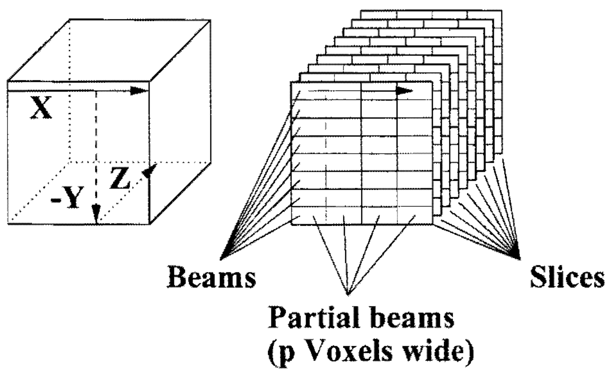

Figure 5: Volume traversal for beam partitioned slice-parallel ray-casting.

buffer of intermediate compositing results requires global connectivity between processing pipelines.

These problems with sub-volume partitioning lead to the development of beam partitioning. Instead of subdividing the volume into subcubes, the size of beams is adjusted to the number of processing elements (see Figure 5). With p processing units beams are partitioned into b *partial beams* of width p, which are subsequently processed.

Similar to sub-volume partitioning, the voxel-neighborhoods required for tri-linear interpolation and gradient estimation need to overlap at the border of partial beams. For example, tri-linear interpolation at the rightmost position of a partial beam requires voxels from the partial beam, which will be fetched in the next cycle. Using a technique called *beam extension*, these border cases can be handled without the overhead in computation and storage of sub-volume partitioning (see Figure 6). Partial beam i at time t is delayed by one cycle so that the necessary extension for partial beam i can be transfered

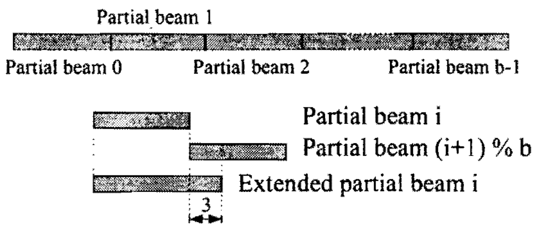

Figure 6: Beam extension provides the necessary data on partial beam boundaries.

from partial beam $i + 1$ at time $t + 1$.

The next section gives an overview of the Teramac system. In Section 4 we describe the sub-volume partitioned prototype implementation of Cube-4 on Teramac, and Section 5 describes our beam partitioned Cube-4 prototype.

3 Teramac - a CCM

The merits of general-purpose versus special-purpose computers have long been debated by computer architects. The configurable custom machine (CCM) [12, 4] is a new class of machine that falls half-way between these extremes. Teramac [2], the largest such machine built to date, achieves the massive parallelism of special-purpose computers and the reusability of general-purpose computers. Teramac provides large numbers of programmable gates, wires, and memories that can be configured to implement user designs.

Special-purpose architectures have often been proposed to accelerate the solution of compute intensive problems. However, these machines are often never built because they solve too narrow a range of problems to justify the cost of their construction. Because one custom computer can implement a countless variety of special-purpose machines, the cost of a custom computer can often be justified when a special-purpose computer cannot. When special-purpose hardware is built, its correctness and usability can be verified first with a custom computer. The high speed of custom computing, relative to conventional software simulations, makes much more exhaustive testing possible.

General-purpose computers have many virtues: they are ubiquitous, inexpensive, and easy to program. They typically also have significantly higher clock speeds than custom computers. However, because general-purpose computers execute at most a handful of instructions per clock cycle while custom computers perform hundreds, custom computers are potentially much faster. On many applications, Teramac has out-performed high-performance workstations by a factor of a hundred or more.

3.1 Teramac Hardware

Teramac is scalable, with systems comprising one to sixteen boards. Figure 7 shows four PCB boards with the attached controller boards and the board to board connections.

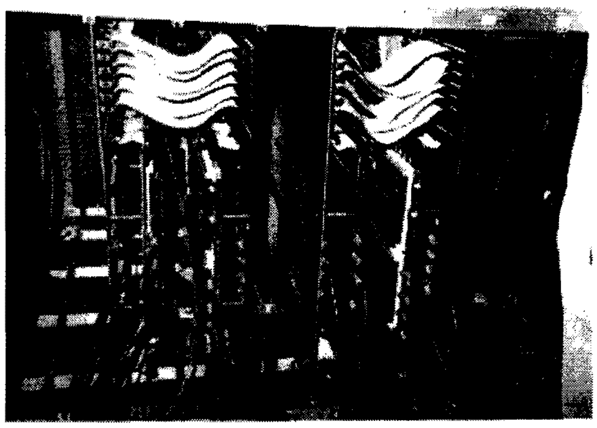

Figure 7: Four PCB boards, connected to each other with ribbon cable, and to a controller board. The pins of the MCM can be seen in the middle.

A full sixteen-board system is capable of running user designs with one million gates at speeds typically in the range of one megahertz.

A custom field-programmable gate array (FPGA), called Plasma [1], supplies the majority of Teramac's programmable resources: gates, crossbars, and multiported register files. Groups of twenty-seven FPGAs are assembled into large multi-chip modules (MCMs) [3] (see Figure 8). Each board contains four MCMs. Each board also contains four dual-ported two-megaword by 32-bit RAM's; thus, Teramac's memory resources are very ample in both capacity and bandwidth.

The Teramac routing resources, consisting of crossbars in the FPGAs and wires on the MCMs and boards, are sufficient for implementing almost any circuit topology. In particular, user circuits are not limited to systolic arrays, as they were in earlier custom computers. Users control Teramac from a host workstation, which connects to Teramac via a SCSI bus. The host also provides configurations and I/O.

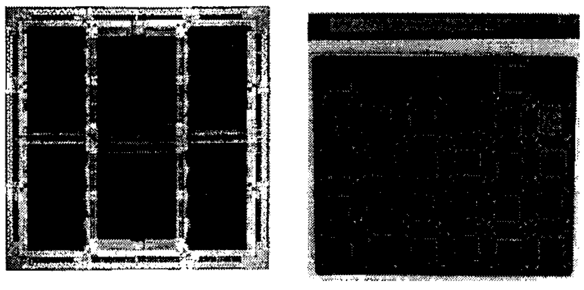

Figure 8: The Teramac hardware. a) A PLASMA chip configurable in three seconds. b) MCM with 27 PLASMA chips on it. The interconnections are handled in 39 layers and over 3,000 output pins.

3.2 Teramac Software

Configurable computers are of limited usefulness unless they include software to map designs onto them. Teramac was designed with the goal that user designs would be mapped onto it quickly and completely automatically. To insure that this goal was achieved, the Teramac hardware and mapping software were created in tandem. Large designs that fill our eight-board Teramac system typically are mapped onto the system in about half an hour, making design iterations reasonably painless.

Users enter their designs into software tools that transform them in two steps into configurations that are ready to run on Teramac. For design entry and the first step of the transformation process, we use general-purpose digital hardware design tools. To maximize user productivity, we have chosen tools that permit the user to express their designs at a high level of abstraction. These tools use logic synthesis to automatically convert the highlevel designs into netlists of simple gates.

The Cube-4 design was created with the Tsutsuji design system [6]. Tsutsuji accommodates large designs particularly well and synthesizes them into gates in just a few minutes. Tsutsuji designs are hierarchies of block diagrams. The blocks represent one of three things: subdesigns which are themselves block diagrams; data path elements (adders, multiplexors, etc.) for which Tsutsuji provides an extensive library of sophisticated module generators; and subdesigns whose behavior is described in Tsutsuji's textual Logic Description Format (LDF). LDF is intended for describing state machines, random logic, and truth tables. We have found that LDF is also useful for creating parameterized designs. Parameterized designs are ideal for parallel applications because they allow the degree of parallelism in the design to be scaled to fill the available hardware.

The second step of the process of creating configurations is called mapping. It is performed by the Teramac compiler, which was written expressly for Teramac. It reads the netlists, merges the simple gates into FPGA-specific gates, performs placement and routing, and ultimately creates configuration bitstreams. Figure 9 shows the design-flow for Teramac. In the following section we introduce the implementation of two Cube-4 prototype designs using the Teramac system and highlight the achieved results.

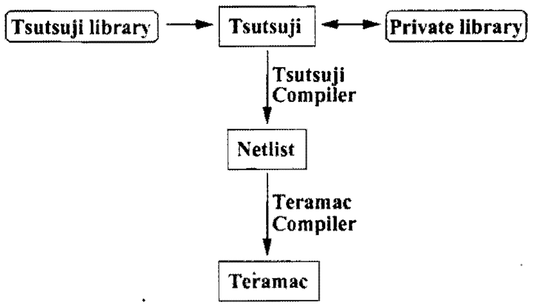

Figure 9: Design-flow for Teramac.

4 Cube-4 Prototypes on Teramac

Two prototype designs of Cube-4 were implemented on the Teramac custom computing system. The first design is based on the sub-volume approach, while the second uses beam partitioning.

The sub-volume partitioned approach has been implemented with eight parallel pipelines, shown in Figure 10. Each pipeline includes the Cubic Frame Buffer (CFB) volume memory, the CFB address generator, tri-linear interpolation unit (TRI), and gradient estimation unit (GRA). Shading and classification and compositing have been implemented in software. To provide the original volume data in a skewed and partitioned format we use a software front-end written in C. A dataset is transformed into a file containing the skewed data of all sub-volumes in sequential order. This file can then be downloaded into Teramac memory. Our implementation on Teramac performs memory access for arbitrary viewing directions, tri-linear interpolation between data slices, and ABC gradient estimation around sample points. The resulting sample values and gradient vectors are

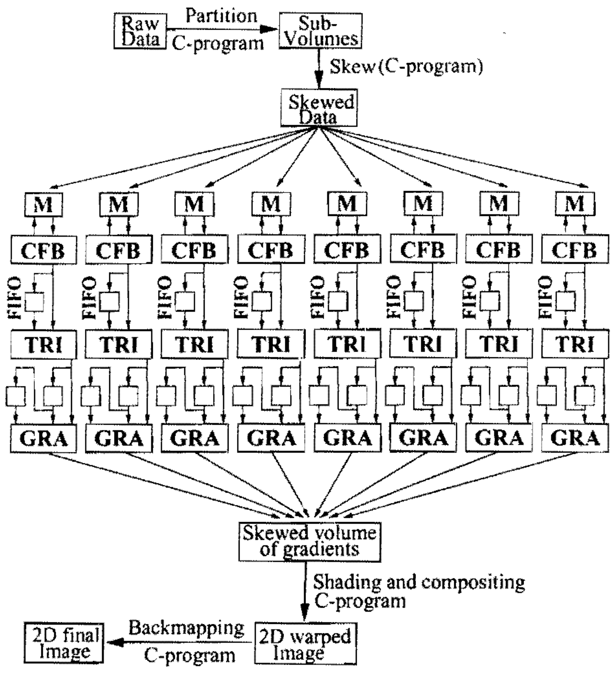

Figure 10: Block diagram of the sub-volume partitioned implementation.

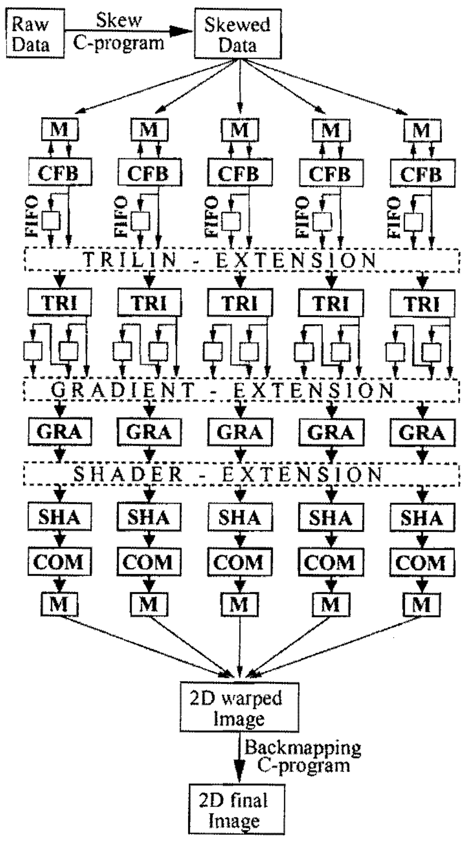

Figure 11: Overview of the beam-partitioned implementation with five rendering pipelines. The interconnections provided inside the extension units are only local, not global.

transfered from the Teramac memory onto the host computer and processed by the software back-end.

Our slice-parallel sub-volume partitioned Cube-4 design on Teramac is able to render datasets of size 128^3. Our implementation contains eight rendering pipelines, although available logic gates on Teramac would allow to implement a design with 16 pipelines. The timing results of this design (see Section 6) indicate high performance. The global connectivity required for the partial result buffers in the compositing units is the main drawback of the sub-volume partitioned design. Consequently, no further effort was put into this implementation.

Our second prototype design on Teramac uses beam partitioning and implements the complete rendering pipelines, including shading (SHA) and compositing (COM) (see Figure 11). Because all stages of the pipeline have been implemented on the Teramac system, the back-end software has been reduced to the 2D image warp. This reduces the software calculations from 3D to 2D, which proves to be a major breakthrough in performance.

We implemented a minimal Cube-4 configuration with five parallel rendering pipelines. The limitation to five pipelines was given due to the structure of the Teramac system. A total of 256 MByte of memory, distributed across several memory banks, is available on Teramac. Our implementation uses memory banks to realize the plane-buffers, the look-up tables for opacity, color transfer-functions, and shading parameters, as well as the intermediate image buffers in the compositing units. The restriction of having one read and one write access to a single memory bank per cycle forced us to use all available memory banks.

The beam partitioned implementation with five parallel rendering pipelines is able to process datasets of 128^3 voxels. A dataset is downloaded into Teramac memory, processed, and the final base-plane pixels are stored in memory modules at the end of each rendering pipeline. A software program uploads the pixel values and performs the 2D image warp from the base-plane to the image plane. In the following section we describe the design of the different pipeline stages in more detail.

5 Beam Partitioned Design

The address of a voxel in volume space can be described in terms of a *slice index* (*S_INDEX* or *S*) in major viewing direction, a *beam index* (*B_INDEX* or *B*) in scanline direction, a *partial beam index* (*PB_INDEX* or *PB*) and a (*PIPELINE_INDEX*) for the location inside a partial beam. For $p = 5$ memory banks, we obtain the address A with the following formula:

$$A = S * \frac{125^2}{5} + B * \frac{125}{5} + PB \qquad (3)$$

This formula is used in the CFB to address the memory banks. The CFB is the main control unit of each pipeline. It is split up into four sub-units as shown in Figure 12. The first is the *TRAVERSAL_UNIT* which keeps track of the position of the currently fetched voxel inside the volume. It consists of three cascaded counters, one for PB_INDEX, one for B_INDEX, and one for S_INDEX (see Figure 5). The values of the three counters are provided to the other sub-units of the CFB unit. The *ADDRESS_UNIT* is connected to the voxel memory of each pipeline, one 8 MByte bank of Teramac memory. The *TEMPLATE_UNIT* generates the resampling weights for the tri-linear interpolation which are forwarded to the TRI unit. To reduce the amount of logic, weights are updated incrementally every time the S_INDEX changes. The current resampling weights in X and Y are updated by simply adding the components of the viewing vector *VIEW_X* and *VIEW_Y*, respectively, modulo 256 (we use 8 bit for resampling weights).

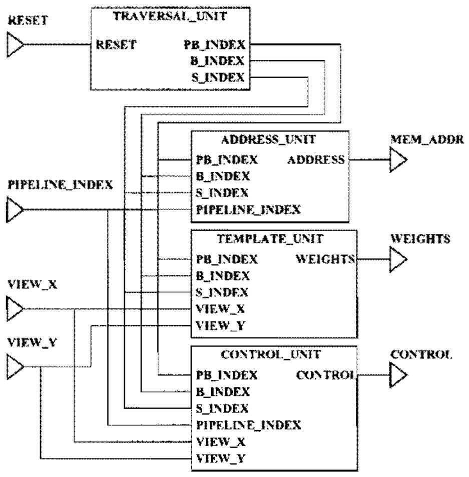

Figure 12: CFB blockdiagram, PB_INDEX indicates the index of the current partial beam, while B_INDEX and S_INDEX indicate the current beam- and slice-index.

The *CONTROL_UNIT* provides the control information (13 bit, shown in Table 2) forwarded with data, allowing the other stages of the pipeline to correctly align the data. *Start* and *End* indicate the beginning and the end of a volume. *Forget* marks invalid values. *X-wrap* and *Y-wrap* indicate that a value is the last one along a ray. *old-X-step*, *old-Y-step*, *X-step* and *Y-step* (are needed to perform the gradient correction and compositing) mark discrete steps along rays inbetween slices. This information is required to reconstruct the rays. In the TRI unit the interpolation of the samples is performed using the weights calculated in the CFB. Seven linear interpolators, each implementing the following formula, are able to calculate one sample per cycle:

Bit-No.	0	1	2
Signal	Start	End	Forget
Bit-No.	3/4	5,6/7,8	9,10/11,12
Signal	X/Y-wrap	old-X/Y-step	X/Y-step

Table 2: Control signals for the beam partitioned approach.

$$w * v0 + (1 - w) * v1 = (v0 - v1) * w + v1 \qquad (4)$$

In the GRA unit, samples out of three consecutive slices are aligned to compute the gray-level gradient [8]. This unit also performs a correction of the values to generate a gradient parallel to the Z-axis and to prevent aliasing [16].

The SHA unit uses the three components of the gray-level gradient for a look-up table based implementation of Phong shading [5]. This Phong model can be very efficiently implemented using only 1.5 kBytes of memory and four memory accesses per cycle. We used a four times wider implementation with 6 KByte lookup-tables because the Teramac limits its memory access to one read and write per cycle.

The resulting intensity value is then multiplied with the classified samples, resulting in red, green, blue, and opacity values which are then forwarded to the COM unit. The tables for the classification of the samples are 32 bit wide and 256 entries deep, corresponding to the eight bit representation of voxel values.

In the COM unit, the shaded samples delivered by the SHA unit are composed to final pixels. The slice by slice order requires a base-plane buffer for one slice of intermediate compositing results which has 125*25 entries per pipeline. Incoming shaded samples are directly composed with the corresponding previous values from the base-plane buffer.

Special attention has to be paid when values not coming out of the same partial-beam range need to be forwarded (cases 0, 2, 3 and 4). Figure 13 shows these cases in which intermediate results from adjacent partial beams are required. The dark square

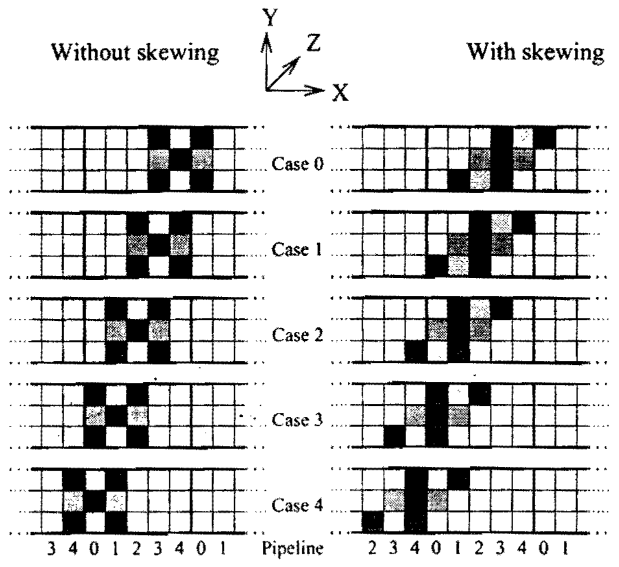

Figure 13: The five pipelines and their possible values for the compositing.

indicates the intermediate result that would be forwarded if no discrete steps in X or Y occur. The surrounding lighter squares indicate the intermediate values that are required depending on discrete steps in X and Y. With skewing, values from adjacent partial beams have to be forwarded in pipeline 0, 1, 2, and 4.

Figure 14 gives an overview over the COM unit. An instance of the TRAVERSAL_UNIT of the CFB is used in the COM unit to keep track of the current slice, beam, and partial beam indices. The *ADDRESSING_UNIT* addresses the intermediate compositing results which have to be composited with the shaded samples delivered by the SHA.

Compositing is performed in the *COMPOSITING_UNIT* using front-to-back compositing [11] and the *BASE-PLANE BUFFER* is implemented using Teramac memory. After a ray is finished, its value is output to the Teramac memory together with the address in X and Y on the base-plane. Otherwise, the intermediate result is stored inside the base-plane buffer.

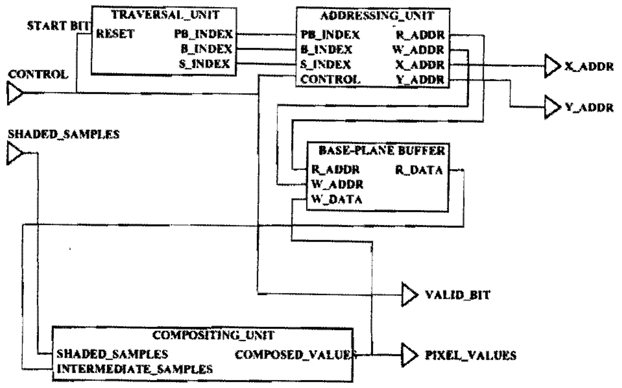

Figure 14: The COM unit.

6 Results

With the sub-volume design we achieved a frame-rate of 1.5 Hz using eight parallel rendering pipelines. Using multiple register stages in the pipeline allowed us to optimize the design from the initial 0.37 MHz to a final frequency of 0.96 MHz. The complete design of the eight parallel rendering pipelines uses 162,816 marketing gates, where one CFB unit requires 5,578 gates, one TRI unit requires 8,557 gates, and one GRA unit requires 6,142 gates. The TRI unit requires more gates than any of the other units due to the multipliers for the seven interpolators used for tri-linear interpolation.

Figure 15 shows volume rendered images of a CT scanned lobster with different transfer functions and different light sources rendered with the sub-volume partitioned Cube-4 design. The beam partitioned implementation with five pipelines is not optimized for speed. A SPICE-estimated maximum clock-rate of almost 0.2 MHz was achieved for 128^3 datasets. The resulting frame-rate of 0.5 Hz could be increased to 2.5 Hz by pipelining the design further to a clock frequency of 0.96 MHz. However, our non-optimized beam partitioned design with five pipelines is still faster than the optimized sub-volume partitioned design with eight pipelines.

The complete design uses 380,341 marketing gates, where one CFB unit requires 3,918 gates, one TRI unit requires 11,037 gates, one GRA unit requires 18,030 gates, one SHA unit requires 13,858 gates, and one COM unit requires 12,861. The logic needed to implement the beam extensions for TRI, GRA and SHA stage requires 80,932 marketing gates. TRI and GRA units have a larger size due to the necessary partial-beam buffers. Many gates can be saved if the partial-beam buffers are implemented with Teramac memory or hardware FIFOs instead of

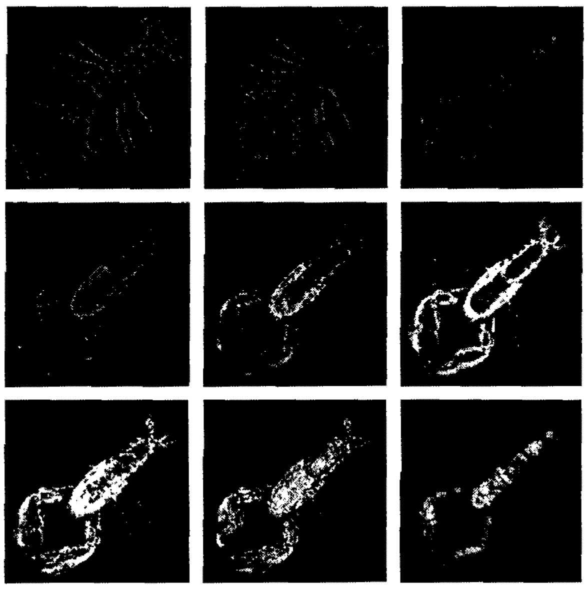
Figure 15: Volume rendered images of a 128^3 dataset of a CT lobster rendered with the sub-volume implementation of Cube-4 on Teramac.

using the expensive Teramac registers.

In Figure 16 we show the theoretical performance of Cube-4 beam partitioned design depending on the size of the dataset and the amount of rendering pipelines at a low 30 MHz processing frequency. Figure 17 shows orthogonal projections of several

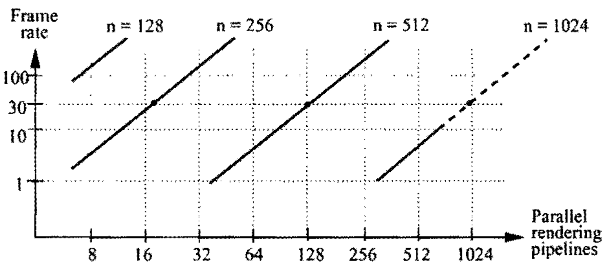

Figure 16: Possible frame-rates assuming a low 30 MHz processing frequency for different dataset dimensions n.

datasets. Those images were rendered completely on Teramac. Additionally, we implemented a protocol for generating frames for animations on Teramac.

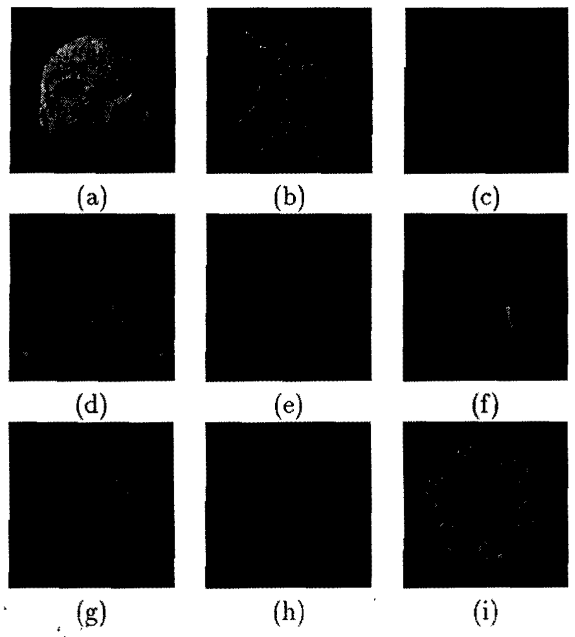

Figure 17: Volume rendering images of 128^3 datasets produced by the beam partitioned implementation of Cube-4 on Teramac. Each image took 1.5 seconds at 200KHz clock-rate. a) Human MRI head. b) Hippocampal pyramidal cell. c) UNC CT head dataset, 45^o rotated. Images d) through f) show the effect of different opacity and color transfer functions on a simulated high-potential iron protein dataset. g) MRI brain. h) Bullfrog ganglion cell. i) Volume sampled sphere flake.

7 Conclusions

We introduced the slice-parallel Cube-4 design on the Teramac. Additionally we showed two scalable and modular partitioning schemes for the slice-parallel approach and proved the feasibility of the proposed architectures by implementing them on the Teramac system. Simulating architectures of this size is not a trivial task. Teramac was a valuable tool that allowed us to efficiently implement those designs in a very limited time-frame. An important future extension to the Teramac system is a frame-buffer to display graphics without uploading results to a host. Furthermore, porting designs to Teramac will be easier in the future when the software is able to directly compile a VHDL description.

Implementing Cube-4 on the Teramac system was a mayor step towards a full-fledged real-time rendering system. We were able to prove the feasibility of the scalable and modular Cube-4 design and got a first impression of the image quality with the rendered images. The next logical step is to use this experience to develop a VLSI implementation of Cube-4 which will then provide real-time performance for up to 1024 × 1024 × 1024 datasets.

References

[1] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, and G. Snider. Plasma: An fpga for million gate systems. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 10–16, 1996.

[2] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac – configurable custom computing. In *Proceedings Of The 1995 IEEE Symposium On FPGA's For Custom Computing Machines*, pages 32–38, April 1995. Napa, CA.

[3] R. Amerson and P. Kuekes. The design of an extremely large mcm-c-a case study. In *The International Journal of Microcircuits and Electronic Packaging*, pages 337–382, 1994.

[4] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–322. 1992.

[5] M. Bosma, J. Smit, and J. Terwisscha van Scheltinga. Design of an on-chip reflectance map. In *Proceedings Of The Tenth Eurographics Workshop On Graphics Hardware*, August 1995. Maastricht, The Netherlands.

[6] W. B. Culbertson, T. Osame, Y. Otsuru, J. B. Shackleford, and M. Tanaka. The HP Tsutsuji logic synthesis system. *Hewlett Packard Journal*, pages 38–51, August 1993.

[7] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *ACM SIGGRAPH Workshop on Volume Visualization*, pages 91–98, October 1992. Boston, MA,.

[8] K. H. Hoehne and R. Bernstein. Shading 3D-images from CT using gray-level gradients. *IEEE Transactions On Medical Imaging*, MI-5(1):45–47, March 1986.

[9] A. Kaufman and R. Bakalash. Memory and processing architecture for 3D voxel-based imagery. *IEEE Computer Graphics & Applications*, 8(6):10–23, November 1988.

[10] P. Lacroute. Fast volume rendering using a shear-warp factorization of the viewing transformation. Technical report, Stanford University, 1995. CSL-TR-95-678.

[11] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[12] J. V. P.Bertin, D. Roncin. *Introduction to programmable active memories, Systolic Array Processors*. Prentice-Hall, 1989.

[13] H. Pfister and A. Kaufman. Cube-4 – a scalable architecture for real-time volume visualization. In *ACM/IEEE Symposium On Volume Visualization*, San Francisco, CA, Oct. 1996.

[14] H. Pfister, A. Kaufman, and T.-c. Chiueh. Cube-3: A real-time architecture for high-resolution volume visualization. In *ACM/IEEE Symposium On Volume Visualization*, pages 123–130, October 1994. Washington, DC.

[15] H. Pfister, A. Kaufman, and F. Wessels. Towards a scalable architecture for real-time volume rendering. In *Proceedings Of The 10th Eurographics Workshop On Graphics Hardware*, pages 123–130, August 1995. Maastricht, The Netherlands.

[16] H. Pfister, F. Wessels, and A. Kaufman. Sheared interpolation and gradient estimation for real-time volume-rendering. In *Proceedings of the 9th Eurographics Workshop On Graphics Hardware*, pages 70–79, September 1994. Oslo, Norway.

[17] R. Yagel and A. Kaufman. Template-based volume viewing. *Computer Graphics Forum*, 11(3):153–167, September 1992.