

# Exploring Architectures for Volume Visualization on the Teramac Custom Computer

W. Bruce Culbertson, Rick Amerson, Richard J. Carter, Philip Kuekes, Greg Snider

Hewlett-Packard Laboratories  
1501 Page Mill Road, Palo Alto CA 94304

*Abstract*—In the past year we have gained experience in custom computing by porting a number of large applications to Teramac. Teramac is a custom computer capable of executing million-gate user designs at speeds approaching one megahertz. Teramac includes software that fully automates the conversion of high-level user designs to configurations that are ready to run on Teramac. Two applications, both in excess of a quarter million gates, are described here: an artery-extraction filter that locates and highlights arteries in medical MRI datasets; and *Cube*, a volume-rendering engine. We have discovered the importance of scalable, parameterized designs. We have been successful in parameterizing some aspects of our designs and we have identified improvements to our tools which would facilitate further parameterization. Our users have benefitted from the speed at which their applications run on Teramac and have gained confidence in their designs after seeing them run on actual hardware.

## 1 Introduction

During the last year, we have had the experience of putting a number of large applications on the Teramac Custom Computer [1,2,3]. Teramac is a modern bread-board for the VLSI age; it was designed to facilitate architectural experimentation by allowing designs to be quickly implemented and quickly executed, which in turn allows many design-and-test iterations to be performed in a short time.

Two of our applications come from the field of volume visualization: an artery-extraction filter and a volume-rendering engine, called *Cube*; they turned out to be particularly well suited to Teramac. They require large numbers of repetitive computations and have relatively few dependencies between computations. These characteristics allow us to create designs which are scalable and highly parallel. Scalable designs can be sized to make full use of the available Teramac hardware. Paral-

lism enables the designs to run at high speed, especially when compared to conventional software simulations on workstations.

Once we realized the value of scalable designs in custom computing, we began to enhance our design methods to facilitate scaling. We now capture our designs at a higher level which includes scaling parameters and we have automated the generation of specific scaled designs from the parameterized designs.

## 2 Why Custom Computers?

For decades computer architects have debated the merits of general-purpose versus special-purpose computers. Teramac is the largest machine built to date in a new class, the configurable custom computer [4,5,6,7], that falls half way between these extremes. It attempts to have the best of both worlds, the massive parallelism of the special-purpose machines and the complete reusability of general-purpose machines.

It is always possible to design and build special-purpose hardware systems that will significantly out-perform custom computers on a given problem. Custom computers, however, have other virtues which can compensate for their lower performance.

Special-purpose computers have often been proposed to accelerate the solution of compute intensive problems. Because these machines are expensive and time-consuming to build, and because they usually solve only a narrow range of problems, they frequently do not get built. However, they can be implemented on a custom computer. Because a custom computer can implement a variety of such machines, and thus accelerate the solution of many problems, the cost of a custom computer can often be justified when the construction of special-purpose hardware cannot.

The least practical special-purpose computers to build are those whose designs must change with each specific instance of the problem being solved. Such computers often solve graph problems (routing, partitioning, finding Hamiltonian paths) and some part of the design typically has the same topology as the graph. Custom computers make such solutions feasible since they can be reconfigured to suit the problem.

Custom computers can be configured dynamically while they solve a single problem [8,9,10]. Thus, several phases of the solution of a problem can be accelerated, each by a separate configuration. Very efficient use is made of the hardware since the hardware is reused during each phase.

General-purpose computers have many virtues: they are fast, ubiquitous, inexpensive, and easy to program. High sales volumes of microprocessors fund the development of very efficient designs and microprocessors do not have the configuration overhead of custom computers. Consequently, microprocessors typically have clock speeds about two orders of magnitude higher than custom computers.

However, microprocessors only execute one to five operations per clock cycle. Custom computers can execute hundreds or thousands of operations per cycle, assuming the application has sufficient parallelism. It follows that custom computers can considerably outperform microprocessors but they must be large to do it. Teramac is large and typically outperforms microprocessors by a factor of one hundred.

Sometimes, of course, a problem is important enough to warrant the design of special-purpose hardware to solve it. Because building hardware is costly, it is important to get it right on the first try. The high speed of custom computing, relative to conventional software simulations, makes much more exhaustive architectural exploration and verification possible. Also, user interaction with the proposed machine, which might be impossible to assess at software simulation speeds, can be quite feasible at custom computer speeds.

### 3 Teramac

Teramac is a large hardware and software system for custom computing built by the Hewlett-Packard Company. It is scalable, with useful systems consisting of one to sixteen boards. There are currently two Teramac systems in operation: an eight-board system at Hewlett-Packard Laboratories, Palo Alto, California, USA; and a one-board system at Brigham Young University, Provo, Utah, USA.

#### 3.1 Teramac Hardware

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 FPGA, called *Plasma*, supplies the majority of Teramac's programmable resources: gates, cross-bars, and multi-ported register files. Each board 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 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. See figure 1.

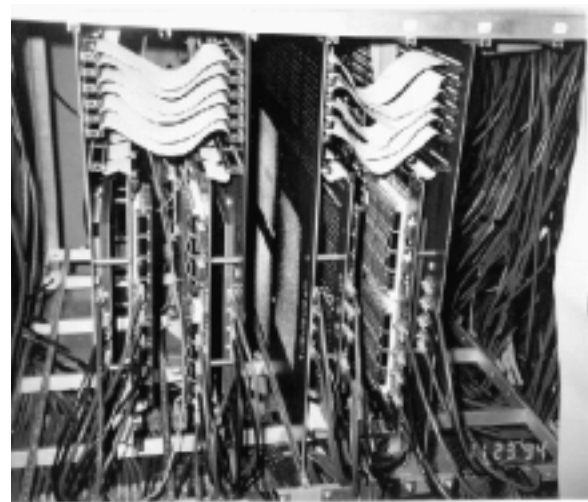


Figure 1. A four board Teramac system.

#### 3.2 Teramac Software

Configurable computers are of limited usefulness unless they include software to map designs onto them. The development of the Teramac compiler began before the Teramac hardware design, which enabled us to design the hardware to be an ideal compiler target. Large designs that fill our eight-board Teramac typically run through our hands-off synthesizing and compiling process in less than an hour, making design iterations reasonably painless.

Synthesis is performed by a conventional design system and converts a design, expressed at a high level of abstraction, into a netlist of simple gates. The Teramac compiler reads the netlist, merges gates into FPGA lookup tables, performs placement and routing, and ultimately creates configuration bitstreams.

We use high-level designs to maximize designer productivity. Tools are currently available to synthesize and map two formats of high-level designs. Most of our applications, including the two described in this paper, were created with the Tsutsuji [11] design system. Recently, Professor Brent Nelson, at Brigham Young University, has developed a process for creating Teramac designs in VHDL.

Tsutsuji encourages graphical design, which closely matches the way most engineers think about designs and describe them to other people. Tsutsuji designs are hierarchies of block diagrams. The blocks represent one of three things: subdesigns which are themselves block diagrams; data path elements (adders, multipliers, multiplexers, 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 was originally intended for describing state machines, random logic, and truth tables but we have recently found it useful for creating parameterized designs.

To be suitable for use in custom computing, a synthesis tool must be able to produce large designs quickly and easily. Tsutsuji has run-time and host system requirements proportional to design size and it synthesized both of our quarter million gate volume-visualization designs in less than ten minutes. Some popular synthesis systems bog down on designs as small as several thousand gates, which discourages design exploration. Such systems need the support of a synthesis specialist to synthesize realistically large designs, putting them beyond the means of many potential custom computer users.

## 4 Creating User Designs

When faced with a new problem, an engineer proceeds through several steps to create a Teramac implementation. See figure 2. First, one or more algorithms to solve the problem are selected. For example, for the problem of rendering volumes, the Cube designers chose algorithms like *ray-casting* and *tri-linear interpolation*. Next, the designer considers the implementation technology: ASIC's, multi-board systems, or custom computers, for example.

Rarely does the most natural formulation of the algorithm exactly fit the chosen technology. A given custom computer or ASIC might provide many times more or less gates than are required to perform the operations dictated by the algorithm. And, a straight-forward implementation of the algorithm might not perform the task fast enough.

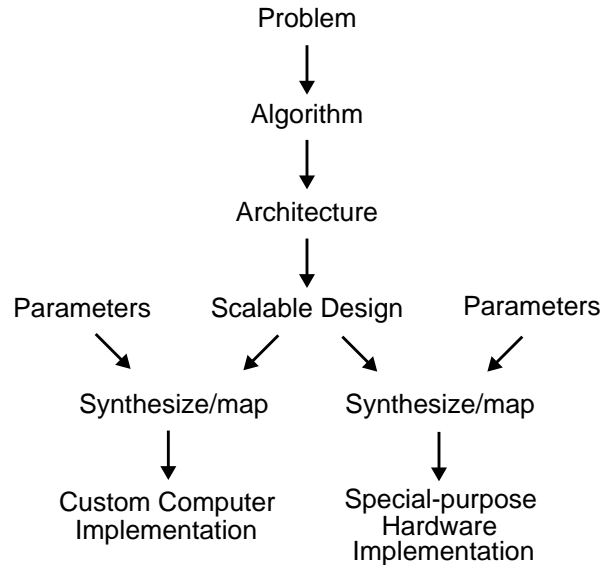


Figure 2. Design Method.

Thus, an architecture is needed which tailors the algorithm to the technology and performance requirements. If insufficient gates are available to perform all the computations simultaneously, then the computations must be divided into subsets, or slices, which can be implemented with the available resources. Also, a looping mechanism must be added to sweep the hardware over all the slices until the complete algorithm is executed.

On the other hand, there may be enough hardware resources to implement the algorithm multiple times. In this case, the hardware implementation of the algorithm can be duplicated and multiple instances of the problem can be solved simultaneously, with an attendant increase in speed. Extra hardware resources may also be used to pipeline the implementation to increase the throughput.

In actual practice, all of these techniques (slicing, duplication and pipelining) can be applied to different parts of the same design. In fact, they were all used in the artery filter and the Cube rendering engine. Unfortunately, it is seldom immediately apparent what degree of slicing, duplicating, and pipelining will make the most effective use of a given amount of hardware; invariably experimentation is required. Therefore, it is very desirable to have a single, abstract, parameterized design from which all the possibilities can be generated, synthesized, and checked for fit and performance.

Parameterized designs have some additional advantages. Suppose an algorithm is first developed and tested on a custom computer, then implemented as a product in an ASIC, and eventually re-implemented in an improved ASIC technology. All three implementation

technologies might offer different mixes of computational resources but the same design can generate all three implementations if it is parameterized. Furthermore, some algorithms can be implemented with different degrees of precision or resolution (typical in signal processing and graphics), and parameters can capture this variety without the need for separate designs.

We have learned a lot as we introduced parameters into the artery filter and Cube designs. Some kinds of parameters were trivial to add while others were added with considerable effort. And, some types of design variation will undoubtedly defeat our parameterization efforts for the foreseeable future.

Tsutsuji includes a feature called *flow-through-bus-sizing*. Once the sizes of input buses are specified, Tsutsuji can figure out the sizes of the rest of the buses. This usually makes parameters for precision and resolution trivial to implement. For example, the voxel bit widths for both Cube and the artery filter were parameterized in this way.

We introduced parameters for slicing and duplication by writing C-language programs that generate LDF. The parameter values are given on the command line when the C program is run; the LDF output is specific for those values. Due to an oversight, the current version of Tsutsuji does not allow instances of subdesigns to be created from LDF. This made our parameterizing efforts considerably more difficult than necessary. An improved LDF might not only fix this problem, but could also include more parameterization features, eliminating the need for the C programs.

Many algorithms specify calculations to be applied to each element in a multidimensional space. The space can be sliced in a number of different ways and the most

effective choice can be among the least obvious. This was the case with Cube: the architecture evolved through a number of versions as different slicing approaches were tried. Changing the direction of the slicing is far more profound than changing the size of the slice and currently requires manual redesign. Adding pipeline stages also requires manual redesign.

Memory is a difficult issue for both custom computer designers and users. Any specific set of memory resources will almost certainly be considerably less than ideal for a given application. Even if enough memory bits have been provided, one large memory can almost never meet the needs of an application that requires lots of small memories. On the other hand, it is inefficient and tedious to configure a large memory if the custom computer only supplies numerous small ones. Both the artery filter and Cube faced memory problems for which we developed *ad hoc* solutions.

## 5 Volume-Visualization Applications

Volume Visualization consists of techniques for transforming three dimensional arrays of volume elements, called *voxels*, with the ultimate goal of producing two dimensional images which convey useful information. Volume datasets come from a number of sources including medical imaging scanners and supercomputer simulations. Most volume-visualization techniques at some point apply a set of computations to every voxel in the dataset, making them very compute-intensive. This is exacerbated by the improving resolution of scanning devices and the attendant increase in dataset sizes. The large number of computations involved with these applications and their potential for parallelism make them

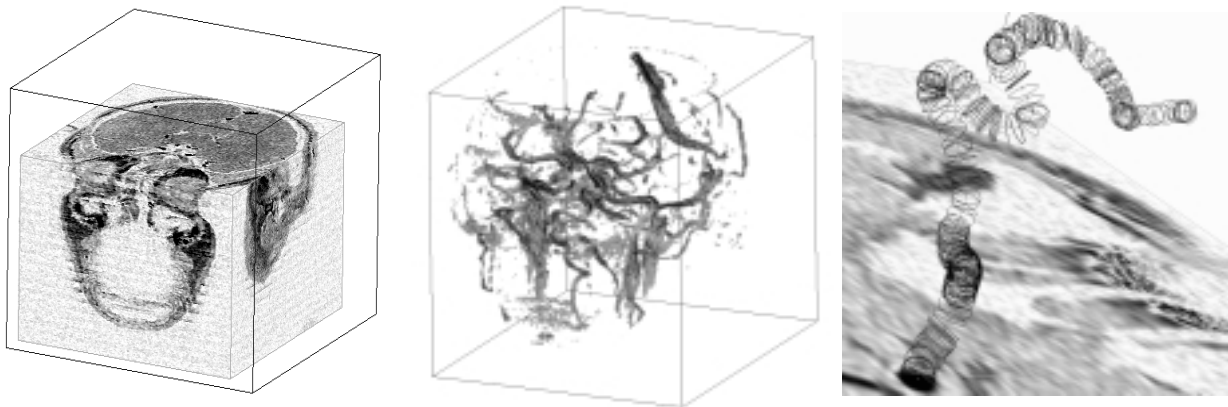


Figure 3. These three images are derived from the same MRI scan of a human head. The raw data is shown at left. The middle image shows the data after Teramac has filtered it to extract arteries. At right, the filtered data has been post-processed to trace a single artery.

ideal candidates for implementation on custom computers.

### 5.1 Artery-Extraction Filter

The artery-extraction filter [12] locates and highlights arteries in medical magnetic resonance imaging (MRI) datasets. See figure 3. Uses for the enhanced images include checking for arterial disease and planning surgeries that must avoid major arteries. Often, the time between scanning and surgery must be minimized, which makes the speed of the artery extraction very important.

MRI datasets can be difficult to interpret if the 3D volumes are simply projected onto 2D images without enhancement. Objects of interest can be occluded by other objects or they can be surrounded by other objects of similar brightness, making them hard to isolate. The artery-extraction filter increases the brightness of cylindrical objects in the dataset that have diameters which are reasonable for arteries; it attenuates the brightness of everything else.

A simple convolution can answer the question: is there a disk with a given location, orientation, and diameter in a volume? A sampling of voxels within the potential disk is multiplied by +1 and a sampling just outside the disk is multiplied by -1; then the products are summed to produce a score. A high or low score indicates that a bright or dark disk is present; a mid-range score indicates that no disk is present. By selecting sample points near the edge of the disk, we increase the sensitivity of the convolution to a specific diameter. See figure 4.

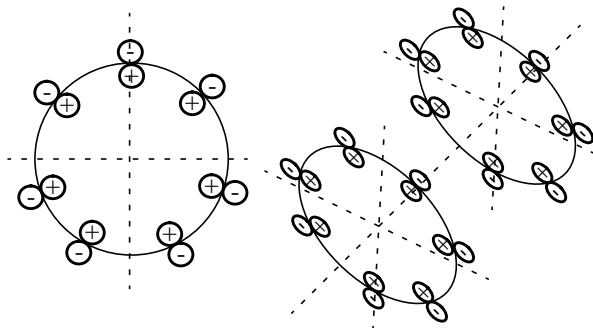


Figure 4. On the left is a convolution for identifying disks. Two such convolutions are combined on the right to create a new convolution which identifies cylinders.

If we find two disks which are both bright or both dark, separated somewhat from each other, and centered on and perpendicular to the same axis, then we have found a potential cylinder (i.e. artery). Thus, the sum of two disk convolutions can answer the question: is there a

cylinder with a given location, orientation, and diameter in a volume? We used disk convolutions with seven sample points inside the disk template and another seven outside. Hence, the convolution to identify a cylinder computes 28 multiplies and a carry-save adder computes the sum. Multiplying by +1 and -1 keeps the hardware simple.

The computationally intense part of the artery filter was implemented on Teramac. See figure 5. For each point in a  $256^3$  volume, Teramac computes a set of cylindrical convolutions to sample the range of all possible orientations and artery diameters, centered at the point. The maximum convolution in the set computed for each point is found and its value, orientation, and diameter are saved. The hardware to compute the convolutions and the maximum is swept over the input dataset, producing a new 3D output dataset.

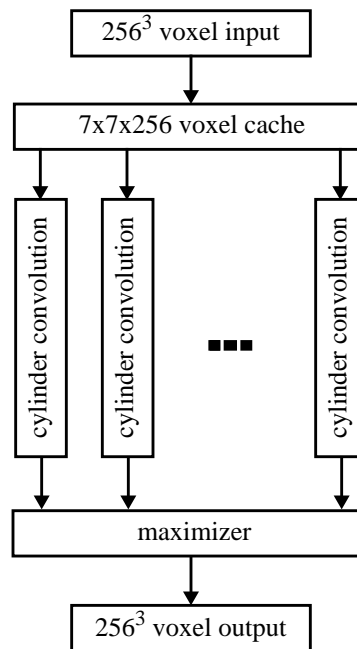


Figure 5. Artery-extraction filter block diagram.

At the beginning of the project, it was not known how many convolutions were needed at each point to produce a good quality output image. Hence, we wrote a C-language program to generate LDF to implement arbitrary convolution set sizes. The LDF instantiates all the hardware to compute, in parallel, all the convolutions centered at one point and find their maximum. For performance reasons, the LDF also inserts pipeline stages into the maximizer tree, with the number of stages automatically computed as a function of the number of convolutions. Through experimentation, we found that sets of 72 convolutions produced good results and, fortu-

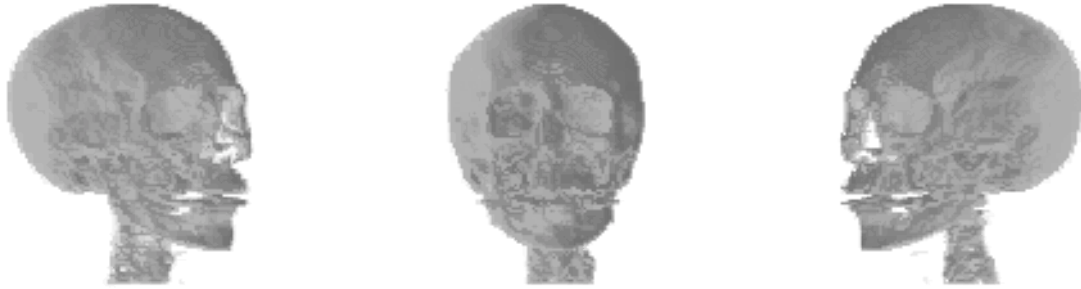


Figure 6. Several frames from an animation rendered by the Cube volume-rendering engine on Teramac.

nately, the required hardware fit nicely on our 8-board Teramac.

The convolutions to compute one output voxel need the values from many (say  $N$ ) nearby input voxels. In fact, all the required input voxels fit in a  $7 \times 7 \times 7$  subset of the input volume. An ideal memory for this application would have room to store the dataset and would have  $N$  read ports. Needless to say, Teramac's memories don't come close to meeting this requirement. Fortunately, the calculations for two adjacent points share  $7 \times 7 \times 6$  input voxel values. Thus, if we cache  $7 \times 7 \times 6$  of the inputs from the previous calculation, then we only need  $7 \times 7$  new inputs for the current calculation. For a further improvement, we cache  $7 \times 7 \times 256$  values as the hardware sweeps along one dimension. Then, when we step by one in the next dimension, we only need to fetch 7 new values per output voxel. By storing the dataset in 7 memories with one read port each, we can fetch the new values for the next calculation in one memory read cycle.

For our final 72 convolution version of the filter, less than one second was needed generate the LDF. Tsutsuji needed two minutes and fifty seconds to create a netlist of size equivalent to 270,000 2-input NAND gates. The Teramac compiler converted the netlist into a configuration in 27 minutes. The design ran on Teramac at 650 KHz and the  $256^3$  dataset was filtered in 27 seconds. A C-language program, written to implement the same filter, ran in 50 minutes or 110 times slower than Teramac.

## 5.2 The Cube Volume-Rendering Project

The Cube Project at State University of New York at Stonybrook [13] is an on-going effort to create an efficient, real-time volume-rendering architecture. Volume

rendering can be used for viewing medical images, viewing the results of 3D simulations, creating animations, and so forth. See figure 6. Given a viewpoint, volume rendering projects a 3D dataset onto a 2D plane to produce an image. Surfaces in the dataset are identified and shaded appropriately for a given light source. Data in the volume is composited, causing opaque objects in the foreground to occlude objects in the background.

Cube has the objective of rendering 30 images per second. Such a frame rate will probably require special-purpose hardware but Teramac has been useful for refining the architecture. The Cube design is scalable so that more hardware will produce images faster or process bigger datasets without loss of speed. No time consuming calculations need to be precomputed for a dataset, which allows the dataset and the projection parameters to change between successive images.

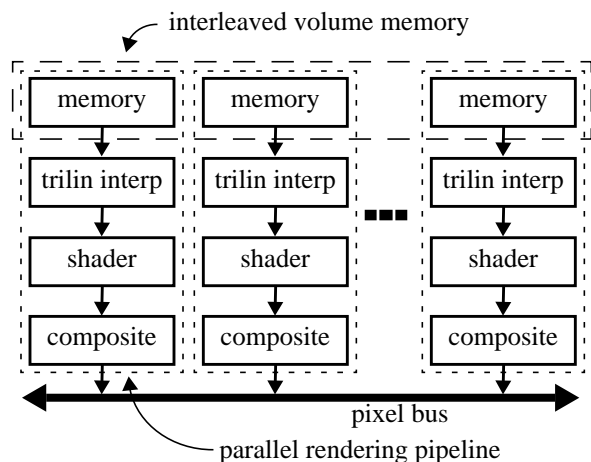


Figure 7. The Cube rendering engine block diagram.

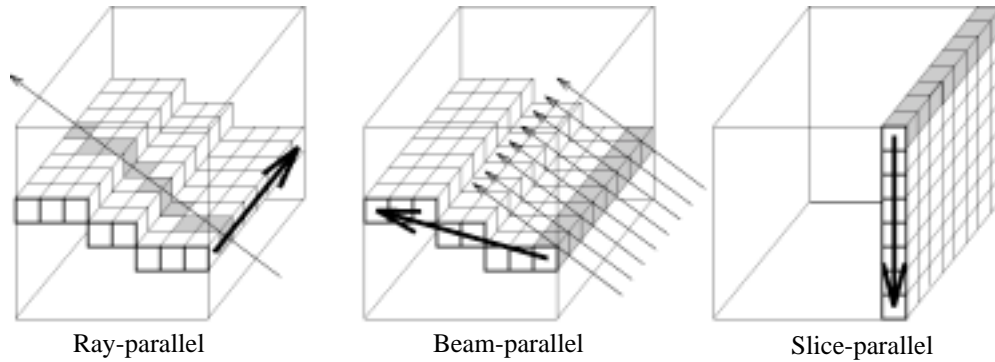


Figure 8. Three ways to parallelize the Cube volume-rendering algorithm.

The Cube algorithm produces images through a series of four steps. Rays are cast from the front of the dataset toward the back, relative to the viewpoint, to produce pixels in the final image. In general, sample points along the rays do not coincide with voxels in the dataset. Hence, small  $2 \times 2 \times 2$  neighborhoods of voxels are fetched from the dataset and used to compute values of sample points on the rays, using tri-linear interpolation. A small  $3 \times 3 \times 3$  neighborhood of interpolated ray samples is used to compute the gradient at each sample. The gradient is combined with the light source and viewpoint angles, using a lookup table, to produce a shaded value for each sample. Finally, each sample is combined with composited values of the earlier samples along the ray to produce a new composited value. Thus, the four rendering steps are: memory fetch, tri-linear interpolation, shading, and compositing. See figure 7.

The Cube architecture has been refined through a number of versions that have primarily varied the order in which the calculations required by the algorithm are carried out, in other words the way Cube is parallelized. In fact, two versions have been implemented on Teramac. The way Cube is parallelized is very critical if memory bandwidth and system interconnect are to be minimized, which in turn is critical if the Cube hardware is to be fast enough and feasible to build. A clever memory interleaving scheme allows voxels from the dataset to be fetched in parallel at a maximum rate, regardless of the scanning order dictated by the viewpoint.

The four-step rendering process is applied to multiple adjacent voxels by parallel pipelines. A variety of ways of sweeping this rendering hardware across the dataset have been tried in various versions of Cube. See figure 8. The *ray-parallel* approach applies the hardware simultaneously to adjacent voxels along a ray. *Beam-parallel* applies the hardware simultaneously to adjacent voxels along the axis in the dataset most perpendicular to the rays, and sweeps the hardware from back to front

along a plane that projects onto a single scan-line in the final image. Both of these methods of parallelizing Cube lead to difficulties like unscalable global interconnect and complicated memory addressing. Finally, *slice-parallel* was chosen, which sweeps the hardware over one plane at a time in the dataset, starting at the front and preceding toward the back.

The first implementation of Cube on Teramac had a parameter for the number of rendering pipelines. This was dropped in the second implementation, partly due to frustration with the parameterization facilities in our tools. However, all bit widths in both designs were parameterized.

Cube uses a number of small lookup tables. The Teramac memories were far deeper than necessary for this purpose and somewhat less numerous than ideal. Teramac probably had enough gates to implement another rendering pipeline but did not have enough memories for another pipeline.

The first version of Cube implemented on Teramac has been verified for correctness and tuned for performance. It is highly pipelined and executes at 920 KHz.

The implementation of the second version of Cube on Teramac is continuing. To date, the emphasis has been on refining the architecture, and debugging and verifying the implementation of the architecture. So far, performance has received little attention. Due to some long paths currently in the design, it executes at only 131 KHz. However, the Cube developers believe the second version, once it is tuned for speed, will be approximately as fast as the first.

The current Cube design on Teramac implements five parallel rendering pipelines. The design is synthesized by Tsutsuji in 8 minutes to 342,000 2-input NAND gate equivalents. The Teramac compiler produces a configuration in 32 minutes. The design produces an image in

390,625 cycles or 2.9 seconds. A C-language simulation produces the same result in 8 seconds on a 250 MHz workstation. A VHDL simulation produces the same result in 3 hours and 21 minutes on a 100 MHz workstation. Both the C and VHDL simulations are carried out at a behavioral level.

## 6 Conclusion

In the past year we have ported a number of large applications to Teramac and learned a lot about custom computing in the process. Volume-visualization applications, like the artery-extraction filter and Cube, are especially suitable to custom computing because they are scalable and inherently parallel. Scalable architectures allow us to size designs to use all the available custom computing resources. Parallelism enables us to run applications at speeds well in excess of what is possible on general-purpose CPU's, assuming a large custom computer is available.

We have found that parameterized designs, and the tools to automatically synthesize them, are useful for several reasons. They greatly expedite the process of sizing the design to the custom computer. They also speed the search for architectures that are economical to build and, yet, provide quality results. We have had some success parameterizing designs using our current tools and we now know how to improve our tools to make parameterization easier.

Architectural exploration generally requires many design-and-test iterations which, in turn, requires that it be possible to make such iterations quickly and easily. We have shown that we can automatically convert large, high-level designs into custom computer configurations in less than an hour. Furthermore, we have shown that, with careful design, custom computers can execute applications significantly faster than workstations.

## 7 Acknowledgments

Developing Teramac has consumed almost all the time of the Teramac team. We are very indebted to our collaborators who have actually used Teramac more than we have; without them we would know much less about Teramac's strengths and weaknesses. On the artery-extraction filter team, we would like to thank Tom Malzbender, who created the filter architecture, and Alte de Boer and Barthold Lichtenbelt, who implemented the filter on Teramac. We would like to thank Professor Arie Kaufman and Hanspeter Pfister, who created the Cube

Project architecture, and Urs Kanus and Michael Meissner, who created the Teramac implementation of Cube.

## References

- [1] R. Amerson, R. Carter, B. Culbertson, P. Kuekes, G. Snider, *Teramac—Configurable Custom Computing*, Proceedings of the 1995 IEEE Symposium on FPGA's for Custom Computing Machines.
- [2] G. Snider, P. Kuekes, B. Culbertson, R. Carter, A. Berger, R. Amerson, *The Teramac Configurable Computer Engine*, Proceedings of the 5th International Workshop, Field Programmable Logic, 1995, Oxford, England, pages 44-53.
- [3] B. Culbertson, R. Amerson, R. Carter, P. Kuekes, G. Snider, *The Teramac configurable custom computer*, Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, John Schewel, Editor, Proc. SPIE 2607, 1995.
- [4] P. Bertin, D. Roncin, and J. Vuillemin, *Introduction to programmable active memories*, Systolic Array Processors, Prentice-Hall, 1989, pages 301-309.
- [5] J. M. Arnold, D. A. Buell, and E. G. Davis, *Splash 2*, Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, 1992, pages 316-322.
- [6] J. Babb, R. Tessier, and A. Agarwal, *Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators*, Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines, Napa, CA, April 1993, pages 142-151.
- [7] S. Casselman, *Virtual Computing*, Proceedings of the IEEE Workshop on FPGA's for Custom Computing Machines, Napa, CA, April 1993, pages 43-48.
- [8] J. Hadley, B. Hutchings, *Design Methodologies for Partially Reconfigured Systems*, Proceedings of the 1995 IEEE Symposium on FPGA's for Custom Computing Machines.
- [9] C. Jones, J. Oswald, B. Schoner, J. Villasenor, *Issues in Wireless Video Coding using Run-time-reconfiguration FPGA's*, Proceedings of the 1995 IEEE Symposium on FPGA's for Custom Computing Machines.

[10] E. Lemoine, D. Merceron, *Run Time Reconfiguration of FPGA for Scanning Genomic Databases*, Proceedings of the 1995 IEEE Symposium on FPGA's for Custom Computing Machines.

[11] B. Culbertson, T. Osame, Y. Otsuru, J. B. Shackleford, M. Tanaka, *The HP Tsutsuji Logic Synthesis System*, Hewlett-Packard Journal, August 1993, pages 38-51.

[12] A. de Boer, *Application of large-scale programmable logic for artery extraction filtering of 3D MRI data*, HP Laboratories Technical Report, HPL-95-95, Hewlett-Packard Laboratories, Palo Alto, CA 1995.

[13] H. Pfister, A. Kaufman, F. Wessels, *Towards a scalable architecture for real-time volume rendering*, Proceedings of the 10th Eurographics Workshop on Graphics Hardware, pages 123-130, Maastricht, The Netherlands, August 1995.