PAVLOV: A Programmable Architecture for Volume Processing

Kevin Kreeger and Arie Kaufman[†]

Center for Visual Computing (CVC)
and Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11749-4400, USA

Abstract

We present a parallel 2D mesh connected architecture with SIMD processing elements. The design allows for real-time volume rendering as well as interactive 3D segmentation and 3D feature extraction. This is possible because the SIMD processing elements are programmable, a feature which also allows the use of many different rendering algorithms. We present an algorithm which, with the addition of hardware resources, provides conflict free access to volume slices along any of the three major axes. The volume access conflict has been the main reason why previous similar architectures could not perform real-time volume rendering. We present the performance of preliminary algorithms on a software simulator of the architecture design.

CR Categories: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—Single-instruction-stream, multiple-data-stream processors (SIMD) ; I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors, Parallel processing; I.4.6 [Image Processing And Computer Vision]: Segmentation;

Keywords: Volume Rendering, Volume Processing, Segmentation, SIMD, 2D Mesh Array

1 Introduction

Real-time and near real-time rendering of volumetric datasets is finally becoming possible. Recent advances provide users with different choices depending on the requirements and resources of the application. Near real-time volume rendering systems are now available in software for users with access to parallel SMP computers such as the SGI Challenge [27] or multiple processor Intel machines with 3D graphics accelerator cards [3]. Recent special-purpose architecture designs that have actually been built include VIRIM [13] and VIZARD [22]. For higher quality rendering, real-time performance, and commercial availability, Mitsubishi Electric Research Labs plans to release a PCI bus plug-in card for performing volume rendering [31] which is based on Cube-4 [32]. Finally, users everywhere will be able to afford to add an accelerator card to their PC which will enable them to view their volumetric datasets in real-time. This advancement has the potential to greatly increase the productivity of scientists, medical professionals and technicians who analyze three dimensional data. The user will be able to interactively explore the volume, adapt the data-to-color mapping function, and manipulate other viewing parameters to more quickly analyze and interpret their data.

Now that high quality real-time volume visualization is on the horizon, the users will, of course, begin to demand more functionality. For example, researchers are already beginning to explore segmentation of volume data and feature extraction. We call this *volume processing.* If the volumetric data is static, it can be pre-processed one time on a general-purpose machine and the user can visualize the processed datasets. However, if the data is being collected in real-time or the user desires to interactively adapt the segmentation parameters, general-purpose processors cannot perform this process at the desired rate. In fact, it is reported that "segmentation of volume data will always require a high degree of user interaction" [4]. For this reason architectures are needed that will provide, at the least, sub-second runtimes for common volume processing algorithms. Meanwhile, proposed hardware rendering architectures are ASIC pipelines for performing volume rendering only. They have not been configured to perform such volume processing tasks.

Segmentation is the process of dividing a volume into two subsets, one of which contains all the voxels possessing a certain property and the other being the rest of the voxels. A simple example of a property is "all voxels whose density value are above a certain threshold". It would be trivial to compute these two sets. Add the requirement that all of the voxels must also be either 6 or 26 connected to some seed voxel and it gets more complicated. Now attempt to perform this operation robustly on data collected from a noisy sensor and it becomes quite a challenge [34].

We propose a programmable architecture for performing volume processing and viewing that we call **PAVLOV** — **P**arallel **A**rray for **VoL**ume pr**O**cessing and **V**iewing. The PAVLOV machine is a two-dimensional array of SIMD — Single Instruction Multiple Data — Processing Elements that would operate as a coprocessor for performing volume operations and rendering. This architecture is programmable for a variety of tasks. In [8] de Boer et al. presented lessons which they learned from building and testing the first ever operable volume rendering accelerator, VIRIM. They reported that the most important feature in the eyes of the user was that of segmentation. They went on to state

[†]{kkreeger,ari}@cs.sunysb.edu

that the main benefit of their system was its flexibility to perform various volume rendering algorithms.

SIMD computer architectures have experienced a decline in popularity over the last couple of years, reasons being the general downswing of massively parallel architectures (most SIMD machines are massively parallel), and SIMD processing elements not taking advantage of the multitude of recent work on micro-processors. Since one of the prevailing theories in most SIMD machine design is to highly replicate very simple processing elements, very few contain the complex processing elements associated with the recent advances in microprocessor design. Another reason is that SIMD architectures provide a worst-case approach to algorithm performance instead of an average case approach because, by design, they must also work on portions of the dataset that don't require processing.

Despite this, SIMD machines have been holding ground in the image processing field. Researchers in image processing have discovered that the advantages of SIMD architectures — no synchronization delay and no communication overhead — provide tradeoffs and outweigh the disadvantages for the types of fine grain data parallel processing found in image processing algorithms and, as we show, volume processing. In fact, most recent image processing architectures are designed either entirely SIMD or with a SIMD core for performing the low-level portions of the algorithms [1, 12, 6, 5, 18, 21, 24]. Volume processing contains many of the same inherent low-level features of image processing. They both contain fine grain data parallelism — where algorithms perform the exact same processing on each and every data element — and the data is represented as a regular array of scalar data. Furthermore, the processing of each data item not only requires minimal information from other data items, but also what it does require is confined to a localized neighborhood.

Additionally, some researchers still believe that there is a future for SIMD arrays because they offer the following benefits:

- SIMD arrays, almost by definition, maximize computational capability per unit hardware (chip, board, etc.).
- SIMD arrays are inherently easy to build and code.
- SIMD arrays provide the power of an ASIC implementation with the flexibility of a solution on a general-purpose machine for many applications (especially ones with fine-grain data parallelism) [15].

2 Previous SIMD volume rendering Work

Schroder and Stoll proposed an algorithm for the Connection Machine CM2 where the volume is stored one beam per processing element. However, the inherent latency of the CM2 limited their performance to 4 frames per second for a 128^3 volume [33]. Yoo et al. presented a method to perform volume rendering on the Pixel Planes 5 machine partly utilizing the 2D SIMD mesh pixel processors and partly the MIMD Graphics Processors [39]. They achieved 20 frames per second for a 128x128x56 volume. Hsu designed a segmented Ray Casting approach for the DECmpp SIMD mesh [16]. However it distributed the volume in sub-blocks and only achieved 4-5 frames/second. Both Vezina et al. [35] and Wittenbrink and Somani [38] proposed algorithms for the MASPAR MP-1 (a SIMD 8-connected mesh). Yet, neither achieved frame rates better that 2-5 frames per second. All of these methods suffered because of the latency inherent in large general-purpose machines. Doggett [9] presented a special-purpose architecture with a 2D array of processing elements for volume rendering. However, his processing elements are ASICs, not programmable SIMDs. Also, the machine is a shared memory design and the volume data flows from a single memory buffer through the array.

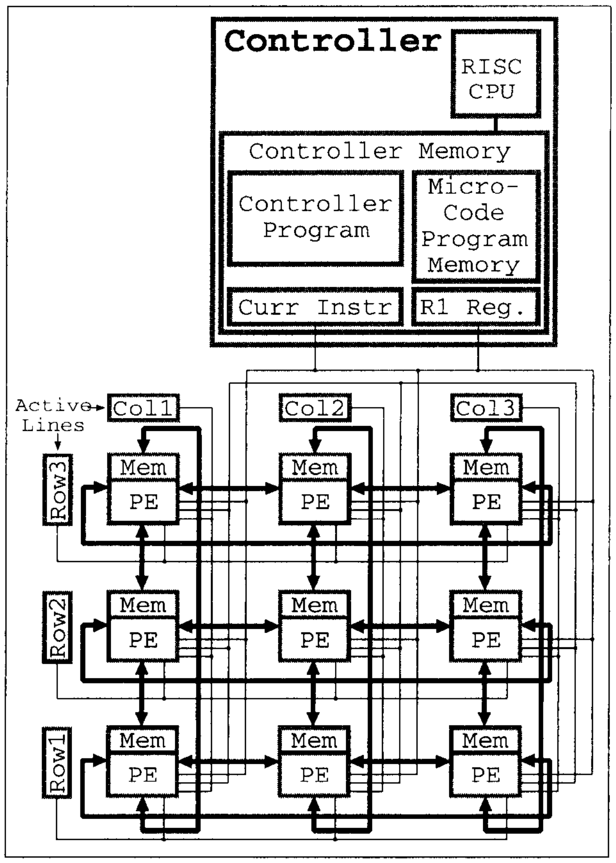


Figure 1: *System level PAVLOV design.*

3 PAVLOV Architecture

We propose a SIMD parallel architecture for performing volume processing. The processing elements are arranged in a 2D mesh with a RISC based controller processor. PAVLOV utilizes a distributed memory design. This means that a portion of the memory is associated with each processing element as shown in Figure 1. Processing elements communicate with each other through direct connections with the nearest processing elements in each of the two dimensions of the array. The processing elements on the edges of the array are connected with the associated processing element on the opposite edge as shown in Figure 1. This topology can be considered as either a 2D plane with wraparound, or as a torus (picture rolling the 2D plane to make a tube, then rolling the tube to make a donut) [17]. For volume processing applications we consider the topology as a 2D plane with wraparound (we sometimes utilize the wraparound and sometimes not). A 2D array of SIMD processing elements inherently processes slice order algorithms [2, 11, 20, 28, 30] very efficiently, since an entire slice of the volume is processed at one time. Therefore, proven slice-order algorithms can be easily ported to the PAVLOV system.

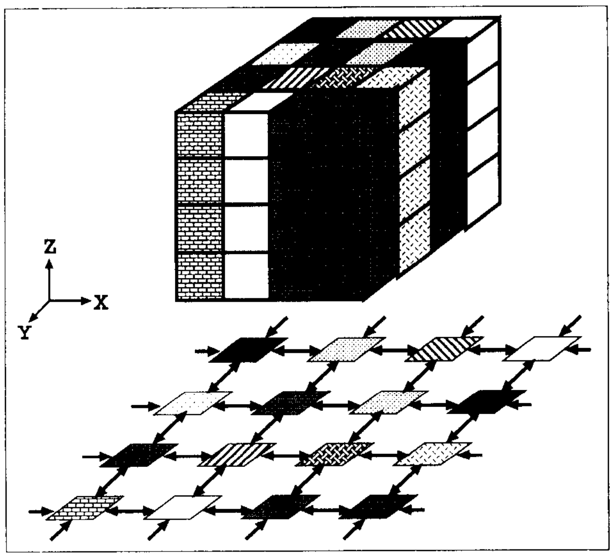


Figure 2: *Mapping a 3D volume onto a 2D array such that a complete beam of voxels along the Z-axis are stored on each processing element*

To access slices of a volume stored in the distributed memory of a 2D array, we propose storing the volume as follows. Each processing element in the 2D array of size N^2 has an (x, y) index. Since a 3D volume of size N^3 contains voxels with (x, y, z) indexes, the volume can be easily mapped onto the 2D array in such a way that all N voxels which have x index X' and y index Y' are stored in a processing element with index (X', Y'). This approach, shown in Figure 2, is used by [33, 35, 37]. Mapping the volume in this manner allows conflict free access to any Z-slice — plane of voxels with the same z index — by the plane of processing elements. However, slice-order algorithms sometimes require accessing the volume in X-slices or Y-slices. Previous attempts to allow conflict free access to these slices of the volume required either three copies of the volume data or some mechanism to transpose the volume before beginning processing. We propose the following algorithm and architectural enhancements to provide conflict free access to slices of the volume in any of the three major viewing directions.

Assuming the volume is stored as above, accessing the Z-slices conflict free is trivial. A 2D example is shown in Figure 3a for slice 0 and 3b for slice 2. Each case requires only one time step to load the entire slice since the N^2 voxels in each slice are distributed among all N^2 processing elements.

When accessing an X-slice of the volume, the entire slice of data is stored in one column of the 2D array. Thus, only one column of the slice can be read at one time. It requires N iterations of reading N voxels each to access the entire slice. Also, the data needs to be shifted across the array to align the slice with the N^2 processing elements. For example, in Figure 3c, slice 0 is being loaded. It is read one beam at a time, and only the processing elements on column 0 are reading voxel data. Between reading each beam, the slice is shifted to the left by one column of processing elements utilizing wraparound communication at the edges of the array. Therefore, there are two steps associated with each column of data being read. Time steps 1 and 2 show these two steps for the first column of data. The remaining 7 columns of data are read similarly, so that at time step 14 the entire slice is loaded, and one more shift operation aligns the slice

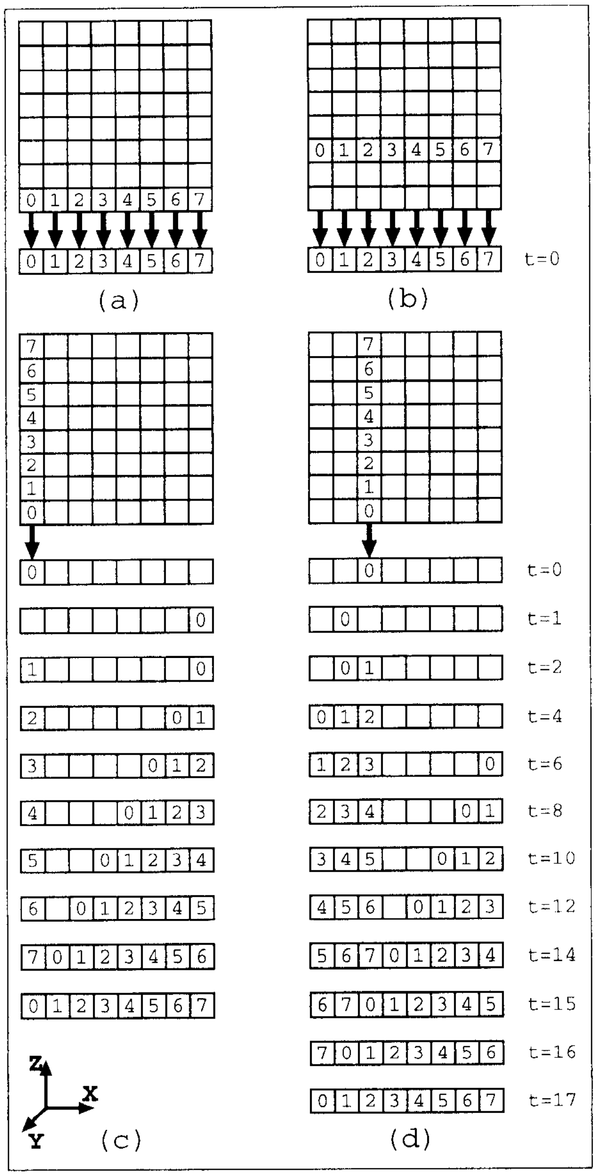


Figure 3: *2D view of loading volume slices. (a) Z-slice* 0 *,(b) Z-slice* 2 *,(c) X-slice* 0 *,(d) X-slice* 2.

with the processing element array. Similarly for slice 2 in Figure 3d the slice is loaded column by column so that the entire slice is read at time step 14. Now, however, it requires three extra shift operations to align the slice with the processing elements. For slices from a column numbered greater than than $\frac{N}{2}$, the slice is shifted the opposite direction after it is loaded. Therefore, this algorithm accesses slices along any axis of an N^3 volume in $2N + \frac{N}{2}$ steps. This works the same for Y-slices by accessing the volume in a row by row fashion.

The $2N + \frac{N}{2}$ latency involved with the previous algorithm is too large to enable real-time volume rendering. Simply increasing the clock rate would only decrease the latency, not remove it, and the latency would still be the same percentage of the overall runtime. This occurs because of the volume re-distribution required when processing is decomposed in image-order. Neumann has shown in [29] that dissecting the processing and storage this way creates greater commu-

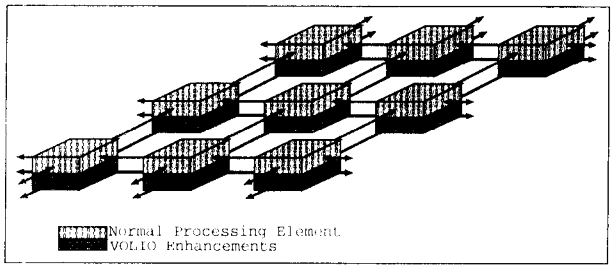


Figure 4: *PAVLOV enhanced SIMD mesh array to provide access to volume slices along any of the three axes.*

nication requirements than object decomposition processing. However, image decomposition allows simpler, and thus more efficient, processing once the data is redistributed. Therefore, we propose the following architectural improvements which completely hides the volume access latency by allowing each volume slice to be loaded while the previous slice is being processed.

- Add a register to each processing element which is utilized for volume I/O (VOLIO). Interconnect the VOLIO registers separately from the interconnections associated with the normal processing elements. With separate interconnections, the VOLIO registers can shift data without interrupting the processing element performing its normal instructions (see Figure 4).
- Provide a mechanism to allow only one row/column of the processing elements to perform an instruction. This is normally done with an activity flag on SIMD arrays.

One can visualize the proposed VOLIO enhancements as a layer of volume memory access elements which lie underneath the regular processing elements as shown in Figure 4. The extra PEs must be arranged this way, simply laying out a mesh that is twice as large only increases the memory access latency. Also, one could consider the enhancements as simply increasing the interprocessor bandwidth — a solution that has been shown to increase the communication performance of mesh connected arrays with minimal VLSI costs [7, 26]. Alternatively, the array plane of VOLIO registers can be envisioned as a fully associative volume memory cache. This cache utilizes the regular access patterns of slice-order algorithms to eliminate memory latency by prefetching the data that is about to be used.

Using the enhancements, Algorithm 1 loads the VOLIO registers (there is actually a 2D plane of registers since every processing element contains its own VOLIO register) with X-slices in $2N + \frac{N}{2}$ steps. Additionally, it does this without affecting the normal functioning of the processing element. Therefore, as long as it takes at least $2N + \frac{N}{2}$ steps to perform the normal processing on each slice of the volume, the next slice is available for the algorithm to use with no delay. The speedup achieved by the addition of the VOLIO enhancements ranges as a function of the number of cycles spent processing each slice. Specifically, the enhancements produce a speedup of 2 (over a normal 2D SIMD mesh) when the per-slice processing is exactly $2N + \frac{N}{2}$. If there are more or less cycles processed per slice, the speedup is less. The time to perform one slice of an algorithm which has C cycles per slice without the VOLIO enhancements is

$$T_1 = (2N + \frac{N}{2}) + C$$

```
for $i = 1$ to N
  Set column[CURRSLICE] active
  load VOLIO with VOLUMEMEMORY[$i$]
  Set all columns active
  shift VOLIO by 1 row
endfor
if CURRSLICE > NUMSLICES/2
  for $i = 1$ to CURRSLICE
    shift VOLIO back 1 row
  endfor
else
  for $i$ = CURRSLICE to NUMSLICES
    shift VOLIO by 1 row
  endfor
endif
```

Algorithm 1: *provides access to X-slices of the volume using the VOLIO enhancements. Since this runs concurrently with the processing of the previous slice, the slices are available with no delay*

while the time with the VOLIO enhancements is

$$T_2 = MAX((2N + \frac{N}{2}), C)$$

Notice that if C is much smaller than $2N + \frac{N}{2}$, then $2N + \frac{N}{2}$ dominates both T_1 and T_2 and therefore produce minimal speedup. The effect is similar when C is very large and C itself dominates both runtimes. Only when C is close to $2N + \frac{N}{2}$ will T_1 be 2 times larger than T_2. An alternative option would be doubling the VOLIO resources to provide better speedup at the maximum point as well as providing a larger range where speedup is achieved.

3.1 PE Architecture

Figure 5 shows all of the components of each of the SIMD processing elements. At the heart of the processor is an 8bit ALU with multiply capability. Two registers, RA and RB are the inputs to the ALU. There is a 256x8bit working memory. The Counter register can be loaded with a value from the processing element. The counter can be decremented and its value used to perform conditional loads. The most common mechanism for performing conditional execution with SIMD arrays is using such conditional loads. In essence, both branches of a condition are computed and only the correct results of the processing are loaded into result registers (or memory locations). Two registers are used for communication. The RV register provides access to the processing elements above and below each processor. The RH register provides left and right access. The VOLIO register, connections and Volume Memory provide the access to the volume in slice order along any of the three axes as described in the previous section. There is also a shader unit associated with each processing element. Currently, the shader is implemented as a reflectance vector shader [36]. We have not finalized the shader implementation yet. We have focused on the rest of the design since we wish to provide volume processing in addition to rendering and there are many good shader designs already proposed in the literature which we could incorporate [10, 19, 36, 23]. For the amount of shader hardware included we are considering the following tradeoffs:

- Shader LUT performed in software. The controller stores the reflectance vector table and loads the val-

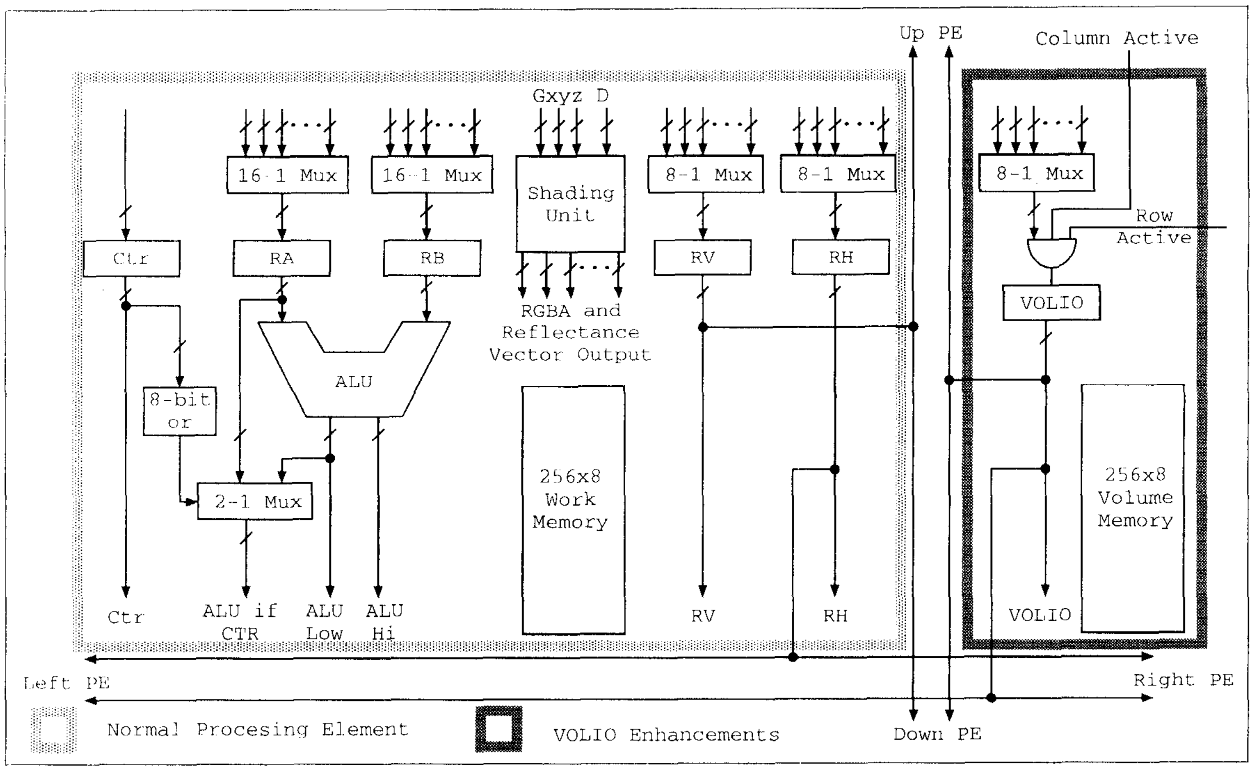


Figure 5: *PAVLOV SIMD Processing Element.*

ues into the instruction stream. It takes 3 clock-cycles per table element to implement this.

- One shader unit per processing element. This approach requires a large amount of VLSI resources.
- One shader per chip. Assuming multiple PEs per chip, we may require software pipelining. This approach allows the shading unit a whole slice time to sequence through all of the processing elements on the chip at the expense of increased working memory.

Herbordt provides a very good analysis of design tradeoffs for designing 2D SIMD mesh arrays in [14]. He analyzed meshes for performing image processing tasks. Since volume processing utilizes many of the same processing features as image processing, we used two of Herbordt's conclusions in our design.

- Increasing the datapath width to 8 bits provides enough speedup to rationalize the added VLSI requirements. However, increasing the width past 8 bits provides little speedup, if any, regardless of the increased VLSI requirements
- The inclusion of Multiply circuit also provides significant speedup to warrant its inclusion despite its increased VLSI requirements. (Many of the SIMD mesh arrays designed for image processing include a Wallace tree multiply circuit since it computes an 8bit x 8bit multiply in 1 cycle.)

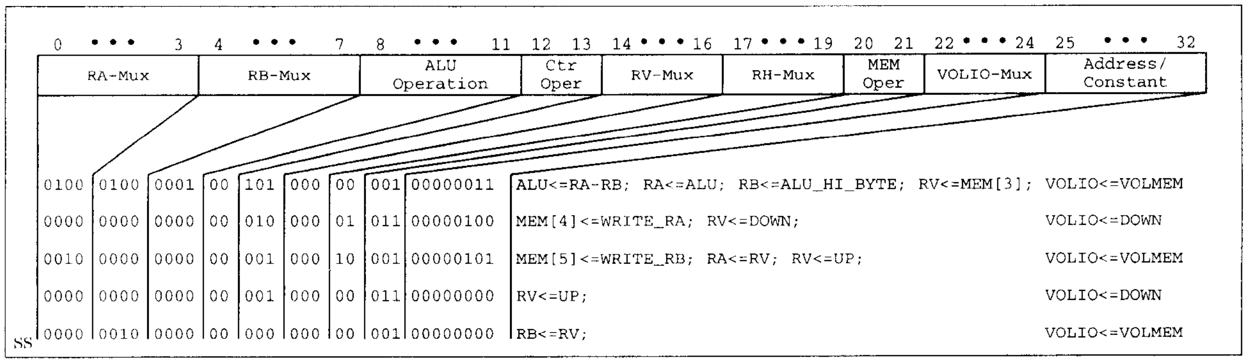
The Processing elements also contain activation lines. There are row and column lines for the entire array that are set by the controller processor. Each Processing element logically ANDs its activation lines together to use as an active flag for that processing element. Currently this is only used for the VOLIO register to allow rows or columns of the volume memory to be read independently.

3.2 Instruction set

In keeping with the *simple processing element* idea of SIMD architectures, we have designed the processing elements to utilize a microcode instruction format similar to many of the SIMD arrays designed for image processing [5, 6, 24]. What this means is that the processing elements are designed such that the control points (select lines on Muxes, read/write lines on memory, etc.) become the bit fields in the instruction. Instructions then set every control point for every cycle. Because of this, every instruction runs on the processing element is a 1-cycle instruction. This is advantageous because the programmer has more control of the processing element. For example, multiple operations can be performed in each cycle as long as they do not require any of the same resources. The VLSI costs of achieving this with regular microprocessors – superscaler, pipelining the instruction fetch/decode/execute – would be prohibitive to placing many processing elements on each chip.

Table 1 shows all of the fields in a microinstruction (grouped where multiple points control one function) along with the action performed for the values placed in the field. Since most fields are independent of each other, the processing element can be programmed to perform many different functions at one time. With this configuration, there are 25 control bits in the instruction (plus 8 address/constant bits) which yields 2^{25} or 33 million possible different instructions.

The example instructions in Figure 6 are from the gradient calculation portion of a rendering algorithm we wrote. The code segment picks up where G_x is being calculated.



```
0100 0100 0001 00 101 000 00 001 00000011 ALU<=RA-RB; RA<=ALU; RB<=ALU_HI_BYTE; RV<=MEM[3]; VOLIO<=VOLMEM
0000 0000 0000 00 010 000 01 011 00000100 MEM[4]<=WRITE_RA; RV<=DOWN;                        VOLIO<=DOWN
0010 0000 0000 00 001 000 10 001 00000101 MEM[5]<=WRITE_RB; RA<=RV; RV<=UP;                  VOLIO<=VOLMEM
0000 0000 0000 00 001 000 00 011 00000000 RV<=UP;                                             VOLIO<=DOWN
0000 0010 0000 00 000 000 00 001 00000000 RB<=RV;                                             VOLIO<=VOLMEM
```

Figure 6: *PAVLOV 33bit microinstruction format and example code from the gradient portion of a rendering algorithm.*

Table 1: *PAVLOV processing element microcode instruction Op-codes.*

RA⇐RA	0000	RB⇐RB	0000
RA⇐MEM	0001	RB⇐MEM	0001
RA⇐RV	0010	RB⇐RV	0010
RA⇐RH	0011	RB⇐RH	0011
RA⇐ALU	0100	RB⇐ALU_HI_BYTE	0100
RA⇐ALU_IF_CTR	0101	RB⇐ALU_IF_CTR	0101
RA⇐VOLIO	0110	RB⇐RA	0110
RA⇐0	0111	RB⇐0	0111
RA⇐LUT_R	1000	RB⇐PHONG_1R	1000
RA⇐LUT_G	1001	RB⇐PHONG_2R	1001
RA⇐LUT_B	1010	RB⇐PHONG_1G	1010
RA⇐LUT_A	1011	RB⇐PHONG_2G	1011
RA⇐CONST	1100	RB⇐PHONG_1B	1100
RA⇐CONTR_VAL	1101	RB⇐PHONG_2B	1101
RA⇐RB	1110	RB⇐CONST	1110
RA⇐255	1111	RB⇐255	1111
RV⇐RV	000	RH⇐RH	000
RV⇐UP	001	RH⇐LEFT	001
RV⇐DOWN	010	RH⇐RIGHT	010
RV⇐RA	011	RH⇐RA	011
RV⇐RB	100	RH⇐RB	100
RV⇐MEM	101	RH⇐MEM	10
RV⇐RH	110	RH⇐RV	11
ALU⇐ADD	0000	CTR⇐CTR	00
ALU⇐SUB	0001	CTR⇐DEC	01
ALU⇐AND	0010	CTR⇐CONST	10
ALU⇐OR	0011	CTR⇐RA	11
ALU⇐XOR	0100	VOLIO⇐VOLIO	000
ALU⇐MULT	0101	VOLIO⇐VOLMEM	001
ALU⇐NOT	0110	VOLIO⇐UP	010
ALU⇐CMP	0111	VOLIO⇐DOWN	011
ALU⇐RB	1000	VOLIO⇐LEFT	100
ALU⇐C ADD	1001	VOLIO⇐RIGHT	101
MEM NO-OP	00	VOLIO⇐RA	110
MEM⇐WRITE RA	01	VOLIO⇐RB	111
MEM⇐WRITE RB	10		

The left and the right samples have already been collected and stored in RA and RB. In the first instruction, the left sample is subtracted from the right sample and the result is stored in RA; the high-order byte of the output is stored in RB (we need to store the sign of the output, since the range of 8 bits minus 8 bits is -255..255). Concurrently with this operation, the current plane of samples is loaded into the vertical communication register, RV. The next instruction stores the output of the gradient calculation into memory and shifts the plane of samples up by one processing element. The third instruction stores the sign information from the gradient calculation into memory, stores the shifted sample plane in RA and shifts the sample plane back down to its original position. The fourth instruction shifts the sample plane down by one more processing element. Finally in the fifth instruction, the shifted sample plane is stored in RB. Now, in each processing element, the G_x has been computed, RA contains the sample below the current sample, and RB contains the sample above the current sample. The processing would continue by computing G_y, etc. Notice how the mathematical operation for G_x was performed concurrently with some of the communication required for G_y. Additionally, the VOLIO enhancement registers are busy loading the next Y-slice of the data without effecting the normal computations in the processing element.

We propose to utilize an off-the-shelf micro-processor for the controller. The row and column activate lines are mapped into the memory space of the micro-processor. There is also a memory mapped register whose value is broadcast to every processing element in the 2D array. This allows the controller to load a scalar value into the array. Finally the microcode instructions are loaded, one for each clock cycle, into another memory mapped location which is also broadcast to every processing element each cycle. The controller handles overlaying the instruction sequences from Algorithm 1 on top of the code that is being run. This creates a dual instruction stream for the PEs. Additionally, the controller ensures that it has finished loading each slice before the algorithm requires it.

4 Performance

Most of the current designs of SIMD arrays for image processing applications being fabricated today utilize 50 MHz clock frequencies [5, 18, 21, 24] (all from Computer Architectures for Machine Perception, 1997 and 1995). The future designs normally count on 100MHz. We analyze the performance of the PAVLOV design for performing a *Ray Casting* algorithm with a clock frequency of 50 MHz. To

Table 2: *Required subslice sizes to perform parallel projections at 30Hz on PAVLOV. Assuming 50 MHz clock yields 1.6 million cycles/frame and the algorithm utilizes 405 cycles per slice.*

Volume Size	$\frac{cycles}{slice}$	Sub-Slices	array size
128^3	13020	32	16x32
256^3	6510	16	64x64
512^3	3255	8	128x256
1024^3	1627	4	512x512

perform real-time rendering at 30 frames/second, each frame can utilize 1.6 million clock cycles. For a 256^3 volume, that means that 6510 instructions can be executed for each volume slice. We have developed a simulation of the PAVLOV array and written a parallel ray casting algorithm which utilizes slice-order object-access. The algorithm requires 405 clock cycles per volume slice to render an image. Therefore, we can provide 30Hz rendering of a 256^3 volume with only a 64^2 PAVLOV array and processing 16 different subvolumes. (Of course with sub-slice processing there is both processing and storage overhead which needs to be addressed in the final system.) With a 64^2 Pavlov array, the sub-slices of the volume can be loaded in 160 cycles according to our previous analysis. This shows how our VOLIO enhancements and Algorithm 1 provide conflict-free access to volume slices along any of the three major directions with no delay. Table 2 presents the same analysis for different volume sizes.

To show how this design can be used to implement various rendering algorithms, we have coded a Shear-Warp projection algorithm which requires 409 clock cycles per slice and a port of the Cube-4-Light [2] perspective algorithm which uses 313 clock cycles per volume slice. The previous analysis can be similarly applied to the runtime of these algorithms.

Figure 7(also in Color Section) shows three views of the skull from the CT head dataset projected along each of the major axes of the volume on a software simulator of the PAVLOV system. The volume is 128x128x113 and the simulator is configured as 128x128 processing elements. Both the X and the Y projections were computed in 52K cycles, and the Z projection was computed in 45K cycles (because there are only 113 Z-slices). The X and Z projections are from viewpoints centered directly in the front and above the volume while the Y projection is rendered from slightly to the left of center.

We have extracted the common primitives used in volume segmentation and analyzed the speed of computing them on a general-purpose computer versus the PAVLOV array. Šrámek [34] reports that it takes approximately 10 seconds to perform *Thresholding, Median Filtering, Erosion* or *Dilation* for a 256^3 volume on an HP9000. We notice that for all of these operations that the direction of the slices does not matter when we process the volume, so we can utilize the main storage direction. Therefore, the next volume slice is always available within 1 cycle. This is important because of the small cycle count of these operations. The $2N + \frac{N}{2}$ steps required to access non-storage-direction slices would dominate the runtime and cause the extra VOLIO enhancements to provide minimal advantage.

Specifically, the number of cycles to perform each of these operations on the PAVLOV architecture ranges from 4 cycles per slice for thresholding to 29 for erosion and dilation. According to Šrámek, these operations are usually repeated, possibly requiring many erosion, dilation and floodfill operations to perform a robust segmentation. Additionally, deriving the order to perform the primitives is an interactive process taking substantial human input.

For example, in Figure 8(also in Color Section) the meat of a 128x128x29 lobster volume is segmented from the shell. Figure 8a shows the entire volume rendered with a translucent shell making the meat visible. In Figure 8b, the meat is segmented by simply performing a threshold above the $\frac{105}{255}$ density level. Clearly, the segmentation achieved is of low quality. Notice the pieces of the shell that are marked as part of the lobster meat as well as antenna around the head. To perform a more robust segmentation, we spent 2 hours testing different combinations of the previous primitives (it took 5 to 10 minutes per segmentation attempt on a HP9000). We developed the following primitive sequence as the best segmentation from 13 different attempts:

```
Threshold - Dilate - Erode - Median_Filter -
     - Erode - Dilate
```

Figure 8c shows the results of our effort to develop a robust primitive sequence for segmenting the meat from the lobster. Unfortunately, this sequence is most likely a one of a kind, and the same process must be repeated whenever another volume is to be segmented.

It would take 4 cycles per slice to perform the segmentation, 29 per slice for the erodes and dilates, and 25 per slice for the median filter. This means that the PAVLOV system is able to perform the sequence of segmentation primitives in 4205 cycles (29 slices times 145 to perform all the primitives) or 80 milliseconds at 50MHz. Furthermore, the PAVLOV system is capable of producing real-time rendering of the segmented data, immediately, so the user can examine the segmentation performance and re-adjust the parameters and primitive operation ordering.

The most challenging segmentation primitive to implement on a SIMD architecture is floodfill. While a naive sequential algorithm performs floodfill with a simple recursive call to all connected neighbors, the worst case SIMD algorithm (here is one of the trade-off areas where SIMD performs worse) may take 54 million instructions to completely floodfill a 256^3 volume if the portion to be filled is the perfect 3D snake. This, however, is still in the 1 second range, still faster than on a general-purpose processor. Therefore, the PAVLOV system can still produce true *interactive* segmentation.

5 Resource Estimates

To get an idea of the VLSI costs of a system based on the PAVLOV design, we compared our design to SIMD array designs recently presented in the image processing community. For example, at the recent Computer Architectures for Machine Perception conference the following Chips were presented.

- NEC has built a chip based on the IRAM [25] design. It includes 32 8bit processors (each with 1KByte of RAM) per chip with 208 pins on a chip [12].
- The *SLIM-II* will be built with 64 8bit processors (each with 256 bytes of RAM and each includes a multiplier circuit) per chip and also 208 pins on a chip [5].
- Komuro et al. submitted a design for fabrication with 64x64 1bit ALU's on 1 chip [24].

Using estimates based on these, we propose to put 8x8 processing elements on each chip. With 8 processing elements along each side of a chip, there will be 32 required connections to the neighboring chips. Additionally, because of our proposed enhancements, each connection is actually 2 lines. With 8bit data-paths, PAVLOV will require 512 pins just for communication in addition to the required power, instruction and row/column activation pins. This design will require time-multiplexing communication pins or the next generation of chip with sufficient pin resources to be feasible. Beyond this, to create the 64x64 processing element array described in Table 2 for real-time rendering of a 256^3 volume will require 64 chips in an 8x8 layout.

An alternative implementation along the line of Komuro et al.'s with 1bit datapaths may provide a more efficient design. Assuming that a 1bit rendering algorithm would take 8 times as many cycles as the algorithm with 8bit datapaths, the rendering algorithm would then take 3240 cycles per slice. With this count, only two subslices may be processed using the same 50MHz clock and 30Hz frame rate for a 256^3 volume. However, following Komuro et al.'s design with 64x64 processing elements per chip, a 256x128 array can be built with only 8 chips in a 4x2 layout, 8 times fewer chips than with 8bit datapaths.

6 Conclusions

We have proposed architectural improvements to 2D mesh architectures which allow conflict-free access to volume slices along arbitrary axes. Overcoming this restriction allows 2D mesh architectures to achieve true real-time volume rendering rates for the first time. We have proposed a SIMD design for the PEs that enjoys the innate advantages of no synchronization delay and no communication overhead. We have demonstrated that the fine grain data parallelism inherent in volume rendering and processing utilizes these advantages to such a degree that they overcome the SIMD disadvantage of requiring worst-case processing. The design can perform various volume rendering algorithms in real-time. Additionally, the design is capable of accelerating volume processing tasks. We believe that with the advent of real-time volume rendering architectures, that interactive volume processing will become desirable. We show segmentation as an example of volume processing, but feel that future applications of volume processing are unpredictable and therefore wanted to provide a programmable solution.

We are currently analyzing the performance of the processing element design. For example, the current design contains two registers as inputs to the ALU and two registers for communication. These could be combined. We would need to analyze the impact on algorithm runtimes.

We need to complete a study of the tradeoffs on algorithms to process volumes larger than the processing element array (since our design utilizes it). For instance, the volume can be stored, and subsequently processed as subblocks or interleaved.

Finally, the 8x8 layout of 64 chips is not as concise as such state of the art volume rendering architectures as the EM-Cube design that Mitsubishi Electric Research Labs is currently building [31]. However, because PAVLOV is programmable, it provides more functionality. It allows multiple rendering algorithms and, more importantly, it allows volume processing such as segmentation and feature extraction. Therefore, it would have a different market than as a PC plug-in card.

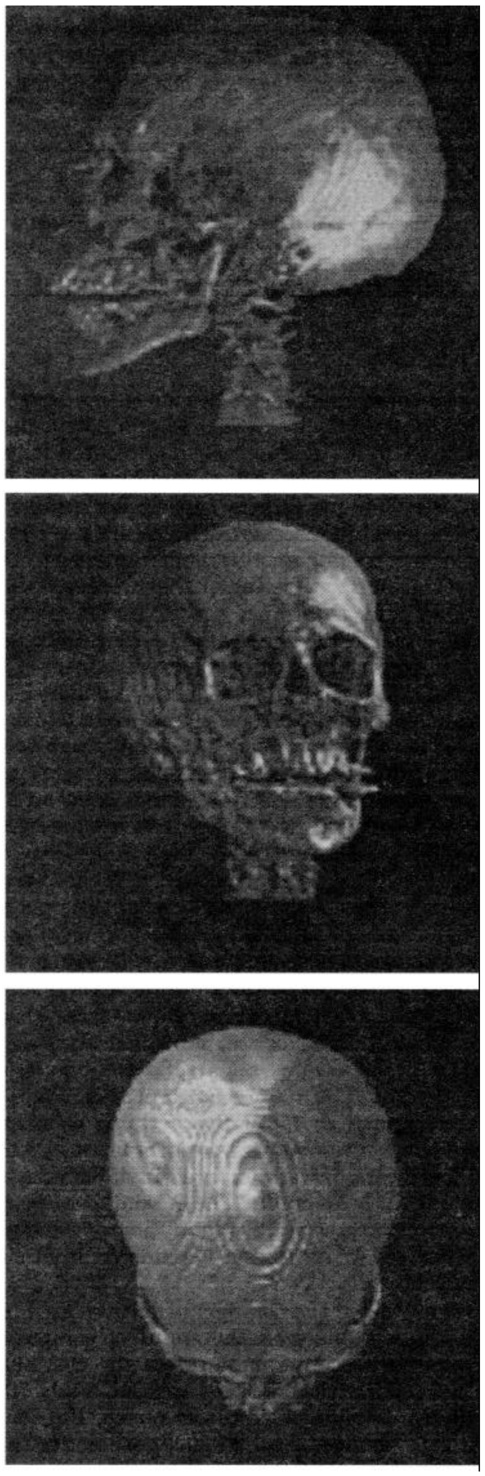


Figure 7: *Parallel projections of 128x128x113 CT head along three axes from our software simulator using parallel projections. X and Y projections were computed in 52K cycles, Z projection in 45K cycles (only 113 Z-slices). The Y projection shows non-orthogonal parallel rendering.*

Acknowledgements

This work was supported by the National Science Foundation under grant MIP9527694, Office of Naval Research un-

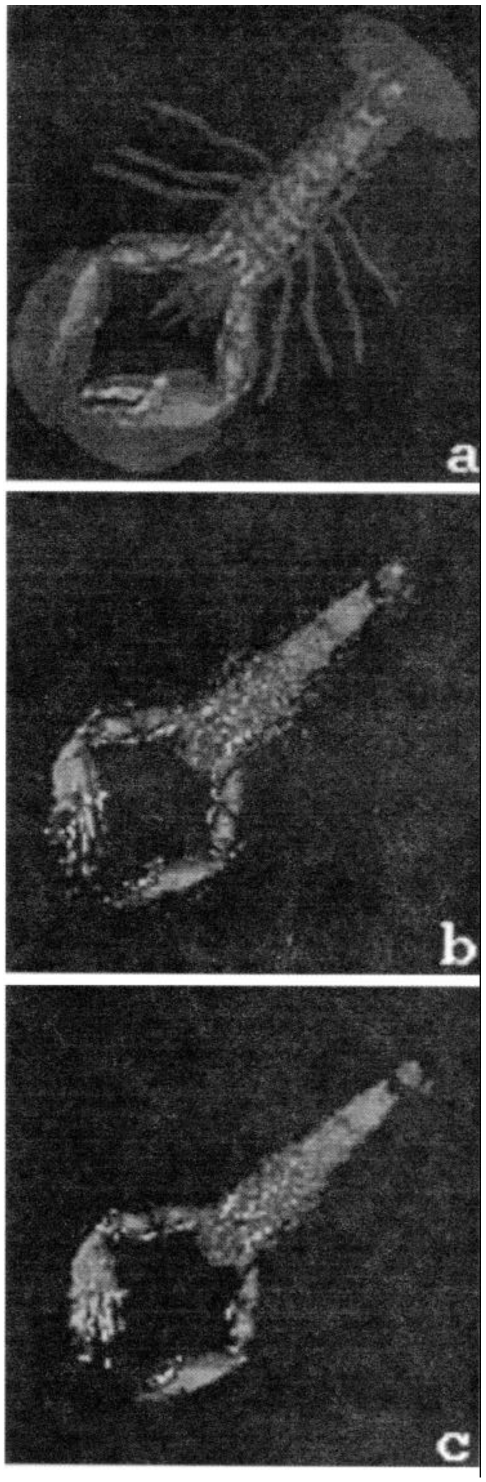


Figure 8: *Segmenting the lobster. (a) Original dataset, (b) meat segmented by performing $\frac{105}{255}$ threshold, (c) meat segmented by performing Threshold - Dilate - Erode - Median Filter - Erode - Dilate.*

der grant N000149710402, Mitsubishi Electric Research Lab, Japan Radio Corp., Hewlett-Packard, and Intel Corp. The CT head dataset is from the UNC database. The lobster dataset is courtesy of AVS Inc.

References

[1] P. Baglietto, M. Maresca, and M. Migliardi. Euclidean Distance Transform on Polymorphic Processor Array. In *Proceedings of the Third IEEE International Workshop on Computer Architectures for Machine Perception*, pages 288–293, Como, Italy, Sept. 1995. IEEE.

[2] I. Bitter and A. Kaufman. A Ray-Slice-Sweep Volume Rendering Engine. In *Proceedings of the 1997 Siggraph/Eurographics Workshop on Graphics Hardware*, pages 121–130, Los Angeles, CA, Aug. 1997. ACM.

[3] M. Brady, K. Jung, H. Nguyen, and T. Nguyen. Two-Phase Perspective Ray Casting for Interactive Volume Navigation. In *Proceedings of Visualization '97*, pages 183–189, Pheonix, AZ, Oct. 1997. IEEE.

[4] I. Carlbom, I. Chakravarty, and W. M. Hsu. SIGGRAPH '91 Workshop Report: Integrating Computer Graphics, Computer Vision, and Image Processing in Scientific Applications. *Computer Graphics*, 26(1):8–10, Jan. 1992.

[5] H. Chang, S. Ong, C. Lee, M. H. Sunwoo, and T. Cho. A General Purpose SLiM-II Image Processor. In *Proceedings of the Fourth IEEE International Workshop on Computer Architectures for Machine Perception*, pages 253–259, Cambridge, MA, Oct. 1997. IEEE.

[6] E. L. Cloud. The Geometric Arithmetric Parallel Processor. In *Proceedings of Second Symposium on Frontiers of Massively Parallel Processors*, George Mason University, Oct. 1988.

[7] W. J. Dally. Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6), June 1990.

[8] M. de Boer, J. Hesser, A. Gropl, T. Gunther, C. Poliwoda, C. Reinhart, and R. Manner. Evaluation of a Real-Time Direct Volume Rendering System. In *Proceedings of the 11th Eurographics Workshop on Graphics Hardware '96*, Poitiers, France, Aug. 1996. Eurographics.

[9] M. Doggett. An array based design for Real-Time Volume Rendering. In *Proceedings of the 10th Eurographics Workshop on Graphics Hardware '95*, pages 93 101, Maastricht, The Netherlands, Aug. 1995. Eurographics.

[10] M. C. Doggett and G. R. Hellestrand. A hardware architecture for video rate smooth shading of Volume data. In *Proceedings of the 9th Eurographics Workshop on Graphics Hardware '94*, pages 95- 102, Oslo, Norway, Sept. 1994. Eurographics.

[11] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. In *Computer Graphics, SIGGRAPH 88*, pages 65 74. ACM, Aug. 1988.

[12] Y. Fujita, S. Kyo, N. Yamashita, and S. Okazaki. A 10 GIPS SIMD Processor for PC-based Real-Time Vision Applications Architecture, Algorithm Implementation and Language Support —. In *Proceedings of the Fourth IEEE International Workshop on Computer Architectures for Machine Perception*, pages 22 32, Cambridge, MA, Oct. 1997. IEEE.

[13] T. Gunther, C. Poliwoda, C. Reinhart, J. Hesser, R. Manner, H.-P. Meinzer, and H.-J. Baur. VIRIM: A Massively Parallel Processor for Real-Time Volume Visualization in Medicine. In *Proceedings of the 9th Eurographics Workshop on Graphics Hardware '94*, pages 103 -108, Oslo, Norway, Sept. 1994. Eurographics.

[14] M. C. Herbordt. *The Evaluation of Massively Parallel Array Architectures*. PhD thesis, University of Mass., Department of Computer Science, 1994. , also TR95-07.

[15] M. C. Herbordt. Univ. of Houston-CAAD Lab, Mar. 1998. http://indus.ee.uh.edu/ideas.html#SIMD Coprocessors are Neat!

[16] W. M. Hsu. Segmented Ray Casting for Data Parallel Volume Rendering. In *Parallel Rendering Symposium*, pages 7 1 4, San Jose, CA, Oct. 1993. IEEE.

[17] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993.

[18] M. Johannesson and M. Gokstorp. Video-rate Pyramid Optical Flow computation on the Linear SIMD Array IVIP. In *Proceedings of the Third IEEE International Workshop on Computer Architectures for Machine Perception*, pages 280–287, Como, Italy, Sept. 1995. IEEE.

[19] S. Juskiw and N. G. Durdle. Interactive Rendering of Volumetric Data Sets. In *Proceedings of the 9th Eurographics Workshop on Graphics Hardware '94*, pages 86–94, Oslo, Norway, Sept. 1994. Eurographics.

[20] A. Kaufman and R. Bakalash. A 3-D Cellular Frame Buffer. In *Eurographics '85*, pages 215–220, Nice, France, Sept. 1985. Eurographics.

[21] H.-N. Kim, M. J. Irwin, and R. M. Owens. MGAP Applications in Machine Perception. In *Proceedings of the Third IEEE International Workshop on Computer Architectures for Machine Perception*, pages 67–73, Como, Italy, Sept. 1995. IEEE.

[22] G. Knittel and W. Strasser. VIZARD: Vizualization Accelerator for Realtime Display. In *Proceedings of the 1997 Siggraph/Eurographics Workshop on Graphics Hardware*, pages 139–146, Los Angeles, CA, Aug. 1997. ACM.

[23] G. Knittel and W. Strauser. A Compact Volume Rendering Accelerator. In *Symposium on Volume Visualization*, pages 67–74, Washington, DC, Oct. 1994. IEEE.

[24] T. Komuro, I. Ishii, and M. Ishikawa. Vision Chip Architecture Using General-Purpose Processing Elements for 1ms Vision System. In *Proceedings of the Fourth IEEE International Workshop on Computer Architectures for Machine Perception*, pages 276–279, Cambridge, MA, Oct. 1997. IEEE.

[25] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, 30(9):75–78, Sept. 1997.

[26] K. A. Kreeger and N. R. Vempaty. Bandwidth Equalization to achieve Hypercube Performance from a Mesh Connected Massive Parallel Processor. In *Proceedings of 5th Australian Supercomputing Conference*, pages 690–698, Melbourne, Australia, Dec. 1992. CSIRO.

[27] P. Lacroute. Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):218–231, Sept. 1996.

[28] P. Lacroute and M. Levoy. Fast Volume Rendering using a Shear-warp Factorization ot the Viewing Transform. In *Computer Graphics, SIGGRAPH 94*, pages 451–457, Orlando, FL, July 1994. ACM.

[29] U. Neumann. Communication Costs for Parallel Volume-Rendering Algorithms. *IEEE Computer Graphics and Applications*, 14(4):49–58, July 1994.

[30] K. L. Novins, F. X. Sillion, and D. P. Greenberg. An efficient method for volume rendering using perspective projection. *Computer Graphics*, 24(5):95–100, Nov. 1990.

[31] R. Osborne, H. Pfister, H. Lauer, N. McKenzie, S. Gibson, W. Hiatt, and T. Ohkami. EM-Cube: An Architecture for Low-Cost Real-Time Volume Rendering. In *Proceedings of the 1997 Siggraph/Eurographics Workshop on Graphics Hardware*, pages 131–138, Los Angeles, CA, Aug. 1997. ACM.

[32] H. Pfister and A. Kaufman. Cube-4 - A Scalable Architecture for Real-Time Volume Visualization. In *Symposium on Volume Visualization*, pages 47–54, San Francisco, CA, Oct. 1996. ACM.

[33] P. Schroder and G. Stoll. Data Parallel Volume Rendering as Line Drawing. In *Workshop on Volume Visualization*, pages 25–32, Boston, MA, Oct. 1992. ACM.

[34] M. Šrámek. *Visualization of Volumetric Data by Ray Tracing*. Austrian Computer Society, Austria, 1998. ISBN: 3-85403-112-2.

[35] G. Vezina, P. A. Fletcher, and P. K. Robertson. Volume Rendering on the MasPar MP-1. In *Workshop on Volume Visualization*, pages 3–8, Boston, MA, Oct. 1992. ACM.

[36] D. Voorhies and J. Foran. Reflection Vector Shading Hardware. In *Computer Graphics, SIGGRAPH 94*, pages 163–166, Orlando, FL, July 1994. ACM.

[37] C. M. Wittenbrink and A. K. Somani. Permutation Warping for Data Parallel Volume Rendering. In *Parallel Rendering Symposium*, pages 57–60, San Jose, CA, Oct. 1993. IEEE.

[38] C. M. Wittenbrink and A. K. Somani. Time and Space Optimal Data Parallel Volume Rendering using Permutation Warping. *Journal of Parallel and Distributied Computing*, 46(2):148–164, Nov. 1997.

[39] T. S. Yoo, U. Neumann, H. Fuchs, S. M. Pizer, T. Cullip, J. Rhoades, and R. Whitaker. Direct Visualization of Volume Data. *IEEE Computer Graphics and Applications*, 12(4):63–71, July 1992.