6

Content-Addressable Memories for Quadtree-Based Images

J.V. Oldfield, R.D. Williams, N.E. Wiseman, M.R. Brûlé

Quadtrees are attractive for storing and processing images with area coherence, but performance has been limited by software overheads. A Content-Addressable Memory (CAM) with ternary storage allows single-cycle searches by pixel coordinate, quadrant or rectangle. To use this feature effectively the authors have reviewed a range of quadtree processing functions relevant to computer graphics and image processing, and some new algorithms have been discovered. The proposed VLSI chip has microcoded logic on each row, as well as its CAM cells. This architecture has been simulated in fine detail with the aid of the Connection Machine as well as by much slower, conventional computers. The combination of quadtrees and CAMs offers significant improvement in performance for display systems and image processing.

CR Categories and Subject Descriptors:
B.3.2 *[Memory Structures]* : Design Styles — *Associative memories*
C.1.2 *[Processor Architectures]* : Multiple Data Stream Architectures — *SIMD*
E.1 *[Data]* : Data Structures — *Trees*
I.3.3 *[Computer Graphics]* : Picture/Image Generation — *Display Algorithms*

Keywords: Algorithms, Content-addressable memories.

1. Introduction

Content-addressable memories have been in and out of the literature (see, for example, [3]) for many years, and several aspects of their use were reported long ago when digital technology was totally different from today's. Indeed, most of the early work had no practical application because the technology was not ready — component cost was high and complexity in memory was a thing to avoid. Superconducting logic was a subject for research in the 1960s and CAM might have found its place there if only the research had born fruit. Several papers were published (for example [18]) about Cryogenic memory cell design with content-addressability.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

Now the situation is changed — semiconductor technology is ready to deliver devices with very large component count and designers are looking for ways to exploit complexity usefully. We may find that the day of the CAM has come. This paper describes a memory cell design, a memory architecture and some algorithms for use that promise high speed and operational simplicity for processing certain kinds of spatial data. Quadtree encoded image data is what we have studied most, but there are other forms of spatial data that could use similar methods to exploit a suitable CAM design.

Quadtree encoding is a form of recursive image decomposition that uses area coherence to compact (code) the image. A square array of pixels of image data is tested for homogeneity. If it is homogeneous, or if the square is sufficiently small, then it is issued as part of the image coding; otherwise it is divided into 4 equal smaller squares and the test is repeated on each one. The process starts with the whole image and continues until there are no more squares to test. There are several possible representations for the code: the squares can be attached by pointers from nodes in the decomposition tree, they can be inferred from some traversal order of the tree, or they can be listed as squares having some size and belonging to some image position. The latter method is of interest to us here. In this method the principle is that each square is represented by a record of two fields. One field contains a colour value, the other a *locational code*. The locational code is written in ternary where 0/1/* indicates left/right/both or above/below/both (both means no subdivision). A pair of such ternary digits (*trits*) then describes an arc from a node in the decomposition tree, and a sequence of pairs (referred to as the locational code) will describe a path to any particular node from the root, and hence identify that node. The locational code is stored left justified in its field with trailing * trits. Their number indicates the size of the square, and the rest of the code its position in the tree. Since each leaf node identifies the path by which it was reached, we need not store non-leaf nodes. We shall see that this is a reasonably compact and convenient way to hold quadtree data in CAM. A binary tree decomposition is equally possible.

A reason to be interested in tree-encoded image data is that some operations on images may be carried out more efficiently on the coded form than on a raster of pixels and, of course, the coded form (usually) saves storage space. Spatial indexing into locational coded quadtree data is particularly simple — one just uses the locational code of some desired position to match against the locational codes stored, with * interpreted as "match anything". One can pass over the list of leaves carrying out this test, or (better) store the list in CAM and use the spatial index value as an access key. Searching with any key requires only a single memory cycle (of the order of 100ns). There are other tree operations, some of which work dramatically better in a CAM implementation, and these issues are the subject of this paper.

A CAM [7] can compare its contents with a given search pattern for every row simultaneously. Present-day CAMs may take advantage of improvements in density and speed for random-access memory, since they can use similar MOS circuit techniques and

fabrication processes. It is now feasible to store quite wide words, e.g. 32-64 bits, in a single CAM chip, and so avoid the problem of resolving partial matches on portions of a word held in separate chips. There is advantage in organising a CAM application so that information is stored in a completely address-independent and order-insensitive manner. Often it is necessary to search on a specific field or fields of each word, and a mask register may be incorporated for field selection. Since a search pattern may match more than one row, multiple responses may occur. Sometimes these must be processed sequentially, and so a multiple-response resolver is included, but often a common operation, such as changing a selected field for every responder, can be carried out in one memory cycle (using a multiple-write operation).

This application is based on a relatively new CAM feature, namely the capability of storing trit values, which was originally applied in logic programming [2]. Storing "don't care" trits is distinct from specifying "don't care" trits in a search pattern. Wade and others [24] developed and tested a 2k trit chip, based on a 5-transistor dynamic CAM cell, and suggest that a 32k trit design with a cycle time of 100ns is feasible with a 2-micron enhanced CMOS process. It is also possible to use static CAM cells with trit storage, and a 15-transistor cell has been reported [22], though obviously the number of trits per chip will be much lower. Chips can readily be cascaded vertically to form CAM structures of virtually unrestricted size. Selection takes place as a result of operations by row logic in conjunction with row status bits. These allow a sequence of qualified search operations to take place.

With microcoded row logic and status bits, the CAM becomes a single instruction, multiple data stream (SIMD) computer. Since established algorithms for computer graphics and image processing are based on a von Neumann architecture, it is important to reconsider them to exploit the concurrency possibilities of a SIMD architecture.

2. Locational Codes for Quadtrees

The locational code method for representing quadtree leaves was introduced in the previous section. The code describes the position and size of the quadrant associated with the leaf. With binary image data, only the locational codes for the black leaves need be stored, because the codes for the white leaves can be inferred from them. With multi-valued image data an additional colour field must also be stored, although the background nodes can still be omitted. This method of representation was first reported by Gargantini [4] and Oliver and Wiseman [11, 12], and has found favour with a number of researchers since.

In deriving locational codes suitable for CAM the origin of the image is assumed to be at the top left-hand corner. For each quadrant the bit patterns of the x and y coordinates of the top left-hand corner are calculated. Depending on the size (width or height) of the quadrant, the bottom m bits of both the x and y patterns are replaced by "don't cares", which are represented by *'s. The two resulting trit patterns $(x_n \cdots x_1 x_0)$ and $(y_n \cdots y_1 y_0)$

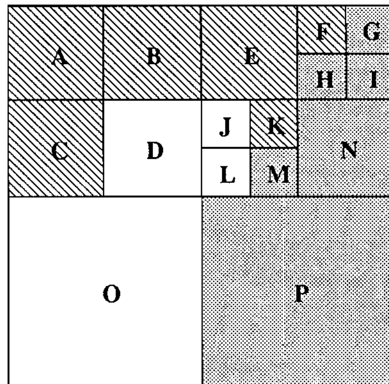

Figure 1: Sample image.

	id	location	colour
A	01	0000**	01
B	01	0010**	01
C	01	0001**	01
D	01	0011**	00
E	01	1000**	01
F	01	101000	01
G	01	101010	10
H	01	101001	10
I	01	101011	10
J	01	100100	00
K	01	100110	01
L	01	100101	00
M	01	100111	10
N	01	1011**	10
O	01	01****	00
P	01	11****	10

Table 1: CAM contents.

are interleaved to produce the locational code ($x_n y_n \cdots x_1 y_1 x_0 y_0$). The value m is log 2 of the quadrant's size and may be thought of as the level, assuming that pixel-sized quadrants are at level 0. The CAM contents for the 8×8 image shown in Figure 1 are given in Table 1. Note that background quadrants are always stored.

Each pair of trits effectively stores a 2 bit code identifying the quadrant (00 = NW, 01 = SW, 10 = NE and 11 = SE). This particular coding scheme is determined by the choice of origin and the order of interleaving (i.e. x before y). Different choices would have led to different coding schemes, but this would only have affected the decoding algorithms, and not the other software. With locational codes stored in normal memory the choice of coding scheme affects the order of storage, but with CAMs there is no concept of order as data can only be retrieved by content and not by address.

The fifth code for each pair of trits is the "don't care" pair (**), which can be translated as meaning all four sub-quadrants. Enumerating the "don't care" code in all its positions produces the locational codes of each pixel within the quadrant. However, it is unnecessary to actually perform the enumeration, because a "don't care" will match on either a "0" or a "1". This is the key to the efficiency and elegance of the CAM method. An enquiry quadrant will not only match on equal-sized or smaller quadrants, but also on larger quadrants.

Setting a different number of trits to "don't care" in x and y produces locational codes which describe certain rectangles. For instance, the top scan line is (*0*0*0) and the bottom half of the image is (*1****). When this type of code is used as a search pattern all quadrants either within the rectangle, or intersected by the rectangle, will match. If the image were represented by a binary tree, as opposed to a quadtree, the CAM could also

store rectangles. For the data in Figure 1, A and C combine to give (000***), G and I combine to give (10101*), and J and L combine to give (10010*). Although binary trees could be used instead, with some saving in space, they also complicate the algorithms; it is for this reason that quadtrees were chosen, and rectangles used only for searching.

3. Applying CAM to Quadtree Operations

Over the past decade a multitude of quadtree algorithms have been published for a variety of quadtree representations. In [16] Samet presents a comprehensive survey of much of the earlier work. It is beyond the scope of this paper to perform a detailed analysis and comparison of all the quadtree algorithms and their CAM counterparts. Instead, a more informal treatment of CAM quadtree algorithms will be presented, with the intention of giving the flavour of the techniques and methods used in their design, and also of indicating which operations are likely to benefit the most. A more rigorous survey is contained in [25]. In section 6, one of the algorithms will be examined in more detail, and sample code and simulation results will be given.

Conventional quadtree algorithms are generally designed to minimise the number of node accesses. The actual cost of an access depends on the type of quadtree, but whatever the representation the cost will only be constant for sequential access. Therefore many algorithms traverse or build their quadtrees in the order in which they are stored, and effectively sort the input or output data. Note that if such an algorithm requires no post or pre-sorting then it is in some sense optimal, and use of CAM will not improve it. Other methods make use of pointers or links to reduce the time taken to access a node randomly. With CAM a search is only a single cycle of the hardware. Consequently, algorithm design is not inhibited by node access times, and simple and efficient algorithms can be developed for quadtree operations.

What follows is a discussion of typical operations. For convenience these will be grouped into four broad categories: those that build quadtrees from some other form of data; those that traverse an existing quadtree to derive some non-quadtree result; those that perform set operations; and those that use neighbour finding techniques. First, the logical fields into which each CAM word is divided will be outlined.

3.1. CAM Fields

From now on each CAM word will be regarded as holding three fields: id, location and colour. The lengths of these fields depend on the application, but typical values for a 32 trit wide CAM might be: id (4), location (20) and colour (8).

The id field permits several quadtrees to be stored at once, and is also used to indicate 'free' words. Initially a multiple-write sets all words to 'free'. A new word can be found at any time by searching with an id pattern of 'free' and selecting the first responder. The id field is then set to the number of the quadtree concerned, along with new values for the other fields as desired.

The location field stores the locational codes described in the previous section. A 20 trit wide field allows a maximum image size of 1024 × 1024 pixels.

The colour field stores the value associated with the quadrant. The term colour is used purely for convenience, as any identifier can be stored here. The user may wish to view the field as being comprised of sub-fields, or may wish to store "don't cares". However, these possibilities have not been explored and it is assumed that the field contains only a simple bit pattern.

To simplify the algorithms still further one value from both the id and colour fields has been reserved for special use. These values are needed in some algorithms to indicate stages of intermediate processing and to avoid using an external stack.

3.2. Quadtree Creation

Two ways in which other representations can be converted into quadtrees are *divide and conquer* (recursive sub-division) and *node insertion*. With the *divide and conquer* method each quadrant is tested against the input data to determine its colour. When the quadrant is not homogeneous, it is sub-divided into four and the process applied to each sub-quadrant. The initial quadrant covers the whole image space, and sub-division is repeated until pixel level is reached. *Divide and conquer* has the disadvantage that the input data has to be rescanned for each quadrant. However, if re-scanning is not costly, this will be an efficient method for conventional quadtrees which store nodes in depth-first order. In this situation CAM has no particular advantage.

With the insertion method the input data is converted to a list of leaf nodes (background nodes are not inserted). The quadtree is initially set to all background and each node is inserted into its correct position, with background nodes sub-divided as necessary. In effect, this method transfers the input data search to a quadtree search. Therefore, in this situation CAM is advantageous, because a quadtree search is performed in a single cycle. Unfortunately, the insertion method as described does not guarantee to produce a minimal quadtree. This redundancy may be ignored and performance impaired, or the quadtree may be compacted locally "on the fly" or globally after the final node has been inserted (but there is no quick way of compacting quadtrees stored in CAM). For some types of data a better solution is to insert maximal nodes, as proposed by Shaffer and Samet [19]. Their method maintains a list of active nodes and only inserts a node when it is clear that it cannot be merged with others.

We have independently developed a maximal node method for inserting raster data into a CAM quadtree. It does not use an active list, but stores this information in the CAM as the quadtree is being built. The quadtree is initialised to a single-background coloured quadrant covering the whole image space. Each run-length (i.e. sequence of identically coloured pixels in the raster) is converted into a list of maximal quadrants that extend as far as possible into the unprocessed data. When adjacent quadrants can be represented by a single locational code they are merged into rectangles. For instance, assuming that the

image data in Figure 1 were represented by run-lengths, this algorithm would generate (00****), (1000**) and (101000) for the first run-length in row 1, and (00*0*1) and (1000*1) for the first run-length in row 2. The codes for the rectangles are used as the search pattern to discover the current (i.e. predicted) colour. If this colour is incorrect it must be changed, and if the matching quadrant is larger it must be successively subdivided. Because the quadrants that are inserted are maximal, there is never any need to compact redundant quadrants.

We have discovered an interesting alternative to *divide and conquer* for generating the quadtree entries of Manhattan rectangles. The x and y coordinate ranges are converted into minimal covering sequences of trits, i.e. using *s whenever possible. The entries are then combined in all combinations, expanding *s into 0 and 1 only when the numbers of * trits do not correspond. This scheme is faster than *divide and conquer*, particularly for small rectangles, and exploits the absence of ordering in a CAM-based quadtree.

3.3. Quadtree Traversal

There are a number of operations that involve visiting some, or all, of the quadtree's nodes, and performing some task at each node. When the order of the traversal is important, nodes can be sorted by location, such as preorder or raster scan order, or by colour. For a conventional quadtree the most efficient order for traversal is of course the order in which the nodes are stored, and if every node has to be visited then CAM cannot possibly make any improvement. The CAM algorithm to visit every node in any order consists of a single search with location and colour patterns of "don't care", and a loop to process each responder. The preorder, or depth first, traversal algorithm uses recursive sub-division and terminates when the multiple response resolver indicates a single match.

A type of breadth-first traversal algorithm, which visits the nodes in order of size, can be implemented very effectively with CAM. It involves searching for "don't cares" in the location field to determine the quadrant's size. For example, to discover whether the CAM has a "don't care" in the most significant trit of the location field a search pattern of (0*****) is used and the responders to this search are searched again for a pattern of (1*****). For the data in Table 1 no words respond after these two searches. However, when the 4×4 quadrants are searched for with (**0***) and (**1***), O and P respond. At the next level the search patterns are (****0*) and (****1*), and A, B, C, D, E and N respond, as do O and P again. Therefore, after every pair of searches a multiple-write must be used to change the id field to the reserved value. This stops quadrants from previous levels from responding again. After the 2×2 quadrants have been processed only the pixel-sized quadrants will remain and these will respond to a single search for (******). A final multiple-write is needed to reset the id field. When displaying quadtrees, breadth-first methods produce successively better approximations to the image, whereas depth first methods do not. As a result the observer perceives the image more quickly with breadth-first traversals.

Frequently image data is required in raster scan order. Samet [15] has presented a selection of algorithms for this type of conversion for fully pointered quadtrees. The simplest algorithm is a top-down approach that visits each run in a row in succession, starting at the root. The other algorithms are bottom-up and make use of neighbour finding techniques. The CAM algorithm for scan conversion is exceedingly simple. For each row the locational code of the left-most pixel is used as the initial search pattern. The size of the responding quadrant is added to the current x value to give the coordinates of the next search pixel. When the colour of the responding quadrant matches the colour of the current run the run's length is incremented by the quadrant's size; otherwise a new run is started. These steps are repeated until the end of the row is reached.

The traversal algorithms described so far have involved searching for a specific location pattern with a "don't care" colour pattern. The responders indicate the colour at the specified location. The inverse operation is to find the locations of the specified colours. A query, such as "find all red quadrants", will generally require a complete traversal of a conventional quadtree. Information stored at non-terminal nodes indicating the colours of the siblings would help to prune the search, or alternatively indices or quadtrees for each colour could be maintained. However, all these solutions generate an overhead for other operations. With CAM quadtrees a single search, using a "don't care" location pattern and the required colour pattern (red), will answer the query.

Using two searches and a microprogrammed logic operation it is possible to search for all nodes that are not a given colour. The first search finds all words for the quadtree under consideration and the second search finds all words for the colour to be eliminated. These are combined with an operation of (A AND NOT B).

To visit each node for every colour normally entails multiple passes over a conventional quadtree. For example, a plotter routine which minimised pen changes would be expensive. However, some operations, such as computing the area or centre of mass [20] for each colour, can be performed with a single traversal if an array of partial sums is maintained. For 8 bit colours, only a 256-element array is required, but if the colour field is large a dictionary is needed. The CAM approach to this type of operation is to search on each colour in turn. If these are not known in advance and the colour space is sparse, a binary chop can be used on the colour field. Starting with a colour pattern of "don't cares", the CAM is recursively searched. When there are multiple responses the colour code is refined in the most significant trit, and the search repeated on the left and right halves.

3.4. Set Operations

The three basic set operations for binary data are union (OR), intersection (AND) and complement (NOT) [20]. For multi-valued data a new definition of these operations is required. By assigning a priority to one of the quadtrees, useful counterparts for union and intersection can be defined. The union operation becomes a replacement operation, in

which the non-background data from one quadtree replace the data in the same location in the other quadtree. The intersection operation becomes an extraction operation, in which the non-background data in one quadtree are replaced with the data in the same location in the other quadtree. The corresponding multi-valued operation for complement is a general re-colouring, using a mapping from the original set of colours to the new set of colours. For conventional quadtrees, union and intersection type operations can be performed by traversing the two quadtrees in parallel, and complement is performed with a single traversal over the quadtree.

A CAM quadtree can be re-coloured with a sequence of searches and multiple-writes. For instance, to complement a black and white quadtree, all black nodes are set to the reserved colour, all white nodes are set to black, and all reserved colour nodes are set to white. The time taken by this algorithm is proportional to the number of colours in the quadtree and is independent of the number of quadrants.

The benefit of CAM for replacement and extraction is less dramatic, and if the two input quadtrees are similar in size there is probably no advantage at all. However, when one input quadtree is considerably smaller than the other, the set operation can be performed in situ, without visiting every node of the larger input quadtree. The minor edit problem, in which a small part of the main image has to be updated, is an example of a replacement operation between different sized quadtrees and is discussed in [10].

3.5. Neighbour Finding Operations

Operations involving neighbour finding are well suited to CAM quadtrees, because the properties of locational codes enable search patterns to be generated for rectangular areas. Each of the one-pixel wide border rectangles, for any quadrant, can be represented by a single locational code. For the seed quadrant in Figure 2 and the data of Figure 1, the locational codes for the North, East, South and West borders are (0010*1), (10010*), (0110*0) and (00011*) respectively, and B, J, L, O and C will respond to these search patterns if the colour pattern is "don't care". Operations requiring the 8-connected neighbours must also check the four corner pixels.

In [13] Rogers describes a naive method for flood-filling an array of pixels. Initially the seed is pushed onto the stack. Every time a pixel is popped from the stack its colour is changed to the new colour, and each of the four neighbours which match the original colour are pushed onto the stack. The stack is processed in this manner until it becomes empty. The method is robust and works for convoluted regions, but requires a large stack and is inefficient because a single pixel is likely to appear on the stack many times. Rogers goes on to describe a more efficient scan line version, but this is also more complicated and more restrictive. With CAM the naive method can be used on quadtrees, but it does not require any stack space and only processes each quadrant once. First, the seed quadrant's colour is changed to the new colour. The quadrants that are the same colour as the seed and intersect the border rectangles are pushed onto the "stack", using a

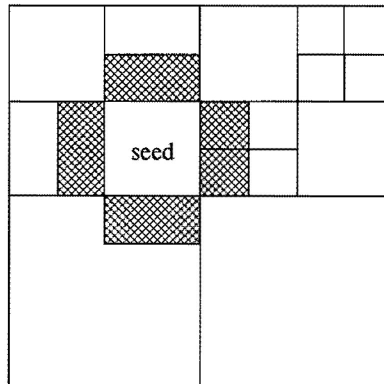

Figure 2: Neighbour finding.

multiple-write to change the colour field to the reserved colour. The next word with the reserved colour is retrieved (i.e. popped from the stack) and becomes the seed, and the whole process repeats. However, because the colour field has been changed to the reserved colour a quadrant cannot be found again by a subsequent search, so each is only processed once.

The operation of connected component labelling assigns separate labels to each distinct black region. A labelling algorithm for CAM quadtrees can be implemented as a series of flood fills. Any black quadrant can be chosen as the first seed quadrant, and the new colour corresponds to the first label. The reserved value of the identifier field is used to indicate black quadrants which have already been processed. Thus when the first flood fill terminates, the next unprocessed black quadrant can be fetched, the label incremented, and the whole process repeated until all black quadrants have been processed. A similar nested loop technique is also used in region growing algorithms.

With the 8-connected neighbour search patterns a number of boundary operations can be performed. For example, to find the boundary pixels in a black and white image the white neighbours of each black quadrant are searched for. A single pixel border of the reserved colour can then be inserted into the matching white quadrants. When all the black quadrants have been processed they are set to white using a multiple-write, and the reserved colour pixels are set to black. Alternatively, the black regions can be enlarged by setting the reserved colour pixels to black, or shrunk by reversing the roles of black and white.

4. CAM Architecture

Our design is based on prior experience with CAM prototypes for logic programming applications [9, 2], including actual fabrication and testing of prototype designs. Figure 3 shows the overall architectural arrangement of our trial design. CAM words are divided into three fields, but trit storage is only used for the locational code.

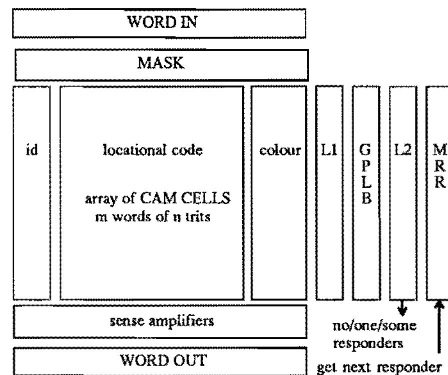

Figure 3: Chip floor plan.

The row logic is repeated identically on every row, and is controlled by vertical microcode control lines. Figure 4 shows the arrangement. The match line is the output of the corresponding CAM row, and will be high if the row was selected for the previous CAM search operation, and a perfect match occurred. Its state may be latched in L1 if the control signal ld1 is high. The general-purpose logic block (GPLB) is identical in function to the OMII data path version [8], but is implemented in dynamic CMOS circuits. It allows any of the 16 functions of two boolean inputs to be generated. Note that this includes the case of all outputs high, e.g. to select every row for a CAM search. There is a latch bit L2 which may be selected to keep the GPLB result. It is part of the multiple response resolver (MRR), and the row select line will be driven by it if the switch is set to 0. Alternatively the (unique) output of the MRR may control the row select line, i.e. if a single-word operation such as read is desired.

In each CAM cycle, row logic may be performed as well as a memory operation, i.e. read, write or search, or a mask change. There is also a memory 'no operation' code for use if only row logic is to be performed in a given cycle.

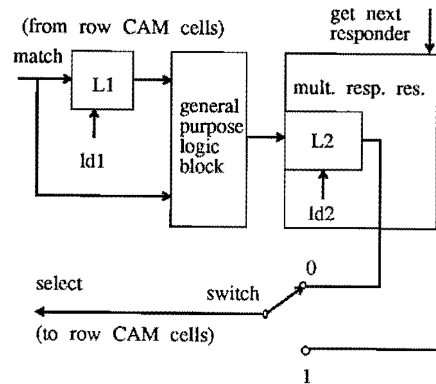

Figure 4: Row logic.

5. Simulation Studies

In order to explore the benefits of the CAM approach more fully, a software simulation of the hardware was needed. It was decided that this simulation should provide an interface that precisely models the chip at a functional level. That is, the procedure calls provided by the simulator should exactly mimic the functions that the proposed chip will perform, once it has been fabricated. There are many reasons to simulate the chip at such a detailed level. By actually implementing algorithms that have been developed, the completeness of the set of functions that the chip needs to perform can be verified. By using the simulation to operate on a variety of different sized problems, statistics can be gathered on the size of the CAM array that is required, as well as the number of operations that the chip performs. This can be useful for determining which functions are critical, as well as indicating under what circumstances significant benefits can be gained by pipelining operations. Finally, the simulator provides a testbed for new ideas in chip functionality, encouraging the development of a better architecture.

A CAM array is an example of an SIMD architecture. Every word in the array can be thought of as an individual processor, each with the capability of performing comparison and masking operations. Also, certain functions can be performed on a global level, such as multiple response resolution. In order to simulate a large array of CAM, of the order of 8K words or more, a significant amount of processing needs to be done. To provide a simulator that can operate on such an array in a timely fashion, the Connection Machine, produced by Thinking Machines, Inc., was utilized. The Connection Machine [5] is another SIMD architecture, where each processor is actually an ALU with a small amount of memory. The system is configured with a VAX-8800 front-end processor, which acts as a control processor, and provides instructions to each of the 32K processors in the

Connection Machine array. Each processor executes the same instruction in lock step, thus allowing operations on large CAM arrays to be simulated by a sequence of a fixed number of instructions.

Each processor in the Connection Machine array is used to simulate one row of the CAM array. Each processor holds the data bits and "don't care" mask for that row, as well as the two registers, match line, and select line. Memory operations, such as read or compare, are executed by a simple, non-iterating sequence of instructions. After each memory operation is performed, the row logic must be evaluated, which again is a simple sequence of instructions. Each Connection Machine processor executes the instructions on its own set of data, just as a CAM array would operate on its data in parallel over all the rows. When utilised in this manner, the Connection Machine provides a straightforward, highly efficient simulation of the proposed CAM array. For example, generating a quadtree of depth 8 containing 3668 leaf nodes, using the Connection Machine provides a factor of 6 speed-up over a VAX 8800-based implementation.

6. Example

All the algorithms presented in this paper have been implemented and tested with various CAM simulators. The software is written in C and uses a two-level interface to CAM. The top level interface is specific to quadtree applications and consists of the following procedures:

```
Search (id, location, colour)
ReSearch (GPLB_op, id, location, colour)
MultipleResponse ()
SingleWrite (mask, id, location, colour)
MultipleWrite (mask, id, location, colour)
Read (id, location, colour)
NextResponder ()
```

The ReSearch procedure enables two searches to be combined with a GPLB operation. The MultipleWrite procedure writes to all selected words (switch setting 0), whereas the SingleWrite procedure only writes to the word selected by the multiple response resolver (switch setting 1). The top level interface is responsible for converting the quadtree fields into a single 32 trit word, setting the appropriate registers, and calling the application-independent low level interface. The low level interface can either use the Connection Machine or a crude array version of the simulator which is used for checking.

Connected component labelling is a quadtree operation that has received attention from several authors [1, 14, 17, 23]. In [21] an algorithm which uses CAM is described, but the CAM is used only for the "union-find" operation and quadtrees are not used. The code for the CAM quadtree version of the algorithm is given below:

```
void CAMlabel (id)
trits id;

/* label each distinct black region */
{
        trits location, next, code [4], dummy;
        int i, n, counter = 2;

        if (! Search (id, ANY, BLACK)) return;
        MultipleWrite (ID, RESERVED, ANY, ANY);
        while (Search (RESERVED, ANY, ANY)) {
                Read (&dummy, &location, &dummy);
                next = EncodeColour (++counter);
                while (TRUE) {
                        SingleWrite(ID | COLOUR, id, ANY, next);
                        n = Get4Neighbours (location, code);
                        if (n == 0) break;
                        for (i = 0; i < n; i++) {
                                if (Search (RESERVED, code [i], BLACK))
                                        MultipleWrite(COLOUR, ANY, ANY, RESERVED);
                        }
                        if (Search (RESERVED, ANY, RESERVED)) {
                                Read (&dummy, &location, &dummy);
                        } else
                                break;
                }
        }
}

int Get4Neighbours (location, code)
trits location, code [];

/* return border locational codes for 4-connected neighbours */
{
        int n = 0, x, y, s, dummy;

        DecodeRect (location, &x, &y, &s, &dummy);
        if (x > 0) code [n++] = EncodeRect (x-1, y, 1, s);
        if (y > 0) code [n++] = EncodeRect (x, y-1, s, 1);
        if (x+s < SIZE) code [n++] = EncodeRect (x+s, y, 1, s);
        if (y+s < SIZE) code [n++] = EncodeRect (x, y+s, s, 1);
        return (n);
}
```

The encoding and decoding routines are omitted here as these are straightforward, as are the constants which should be obvious, with the possible exception of ANY which is "don't care" and RESERVED which is the reserved value (all 1's). The outer loop of the CAMlabel procedure processes each distinct BLACK region and the inner loop processes each quadrant within that region. The running time of the algorithm is therefore proportional to the number of black nodes in the quadtree, as can be seen from the simulation statistics in Table 2 for the data shown in Figure 5. On average, five calls are made to search for each black node (i.e. inside the inner while loop).

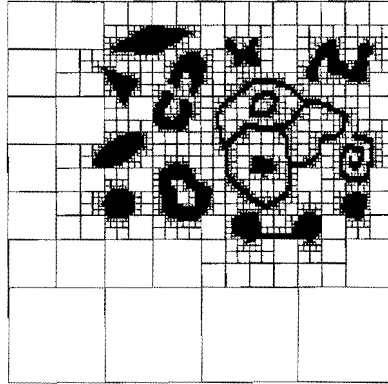

Figure 5: Depth 7 data.

depth	6	7	8	9
black nodes	328	904	2324	3683
total nodes	883	2191	5095	7861
searches	1689	4537	11637	18432
reads	328	904	2324	3683
writes	582	1688	4225	6579
no-ops	1689	4537	11637	18432
set-masks	374	1148	2722	4350
cycles	7261	19943	50731	80170

Table 2: Statistics for component labelling

In [23] Unnikrishnan *et al* presented comparative figures for connected component labelling algorithms. Their new algorithm required only 90 searches, whereas two other algorithms required 869 and 938 searches respectively. For the same data the CAM algorithm requires 249 searches, but searching would be quicker in a CAM. This performance comparison may seem inconclusive, but there are two points worth noting: the CAM algorithm is very simple, and more importantly, the CAM method has needed no special adaptation to optimise its use for connected component labelling.

7. Wider Aspects and Further Work

The methods described have assumed that the complete image can be stored in the CAM, which is unrealistic for some applications. A geographic information system (GIS) must be capable of handling large volumes of data. Therefore, the present algorithms need modification before a GIS which utilised CAM could be built. A paging scheme, which

divides the CAM into logical sections, is currently being investigated. One section would store an index formed from the higher levels of the quadtree. Other sections would be used for storing the lower levels as they are paged in, and some sections would be reserved for storing results that "carry over" from one page to another.

A much simpler modification to the present algorithms, which would be relevant to GIS applications, is to allow several colours (i.e. several values for one attribute) to be stored at each leaf node. This can be achieved by using an extra word for each additional colour, with the other two fields kept the same. Alternatively, it may be possible to store "don't cares" in the colour field to indicate multiple attributes. Recording variable length colour information can be cumbersome for many conventional quadtree storage structures, so this trivial extension to the CAM methods will make them even more appealing for some applications.

An interesting use for multiple attributes is in storing truncated quadtrees. Instead of storing "average" values, and losing colour information, each sibling colour could be stored. Because the truncated quadtree requires two less trits for each pruned level, an exact pixel count can be maintained for each colour, so only positional information is lost. This technique requires further research, but it may be suitable for reasonably static databases.

The locational code methods are equally applicable to 3D data, and similar algorithms could be developed for octrees. One application area likely to benefit from CAM is ray-tracing. Here, two of the most costly operations are refining the object space *in situ* and following a ray into adjacent voxels. The replacement and neighbour finding techniques already described would make a significant improvement to these operations.

The simulation studies referred to here have confirmed the appropriateness of the CAM architecture proposed. We plan further simulation studies including performance estimation prior to completing prototype chip layouts for fabrication by the USC/ISI Fast Turnaround Fabrication Service (MOSIS).

8. Conclusion

Storing quadtrees in Content-Addressable Memory with ternary-coded locational codes is attractive in several respects. CAMs can be searched rapidly for either single or multiple responders, and selected responder fields can be changed to the same pattern simultaneously.

Our survey of algorithms for quadtree applications in computer graphics and image processing shows that in many significant cases CAM improves the performance substantially, and at worst leaves the performance unchanged. Simulation studies have been performed on several single-processor computers and one massively-parallel one, the Connection Machine (CM1). They demonstrate the appropriateness of the trit storage scheme along with its related algorithms, and can be applied effectively with general-

purpose SIMD computers as well as the novel architecture proposed here. It remains to confirm the simulation results by the development of experimental CAM chips and CAM-based display systems.

2. Acknowledgements

The ternary coding scheme was first proposed by C.D. Stormon for binary trees arising in logic programming. One of us (JVO) was supported by a Visiting Fellowship from the U.K. Science and Engineering Research Council while on sabbatical leave. RDW is supported by the ICL Research Studentship and the ORS Award Scheme. JVO and MRB thank the New York State Center for Advanced Technology in Computer Applications and Software Engineering, and the Northeast Parallel Architectures Center, both at Syracuse University, for continuing support.

3. References

1 Atkinson, H.H., Gargantini, I., and Walsh, T.R.S. "Counting Regions, Holes, and their Nesting Level in Time Proportional to the Border", *Computer Vision, Graphics, and Image Processing*, vol. 29, no. 2, (February 1985), pp 196-215.

2 Brûlé, M.R., Oldfield, J.V., Ribeiro, J.C.D., and Stormon, C.D. "An Architecture based on Content-Addressable Memory for the Parallel Execution of Logic Programs", *CASE Tech. Rpt.* no. 8801, Syracuse University, January 1988.

3 Frei, E.H., and Goldberg, J. "A Method for Resolving Multiple Responses in a Parallel Search File", *IRE Transactions on Electronic Computers*, vol. EC-10, no. 4, (December 1961), pp 718-722.

4 Gargantini, I. "An Effective Way to Represent Quadtrees", *Communications of the ACM*, vol. 25, no. 12, (December 1982), pp 905-910.

5 Hillis, W. D. "The Connection Machine", MIT Press, Cambridge, MA, 1985.

6 Kedem, G. "A Data Structure for Hierarchical On-Line Algorithms", In *ACM IEEE Nineteenth Design Automation Conference Proceedings* (Las Vegas, Nevada, June 1982). pp 352-357.

7 Kohonen, T. "Content-Addressable Memories", Springer, New York 1980.

8 Mead, C., and Conway, L. "Introduction to VLSI Systems", Addison-Wesley, 1980 pp 152.

9 Oldfield, J.V. "Logic Programs and an Experimental Architecture for their Execution", *Proceedings IEE* (London), vol. 133, part E, no. 3, (May 1986), pp 163-167.

10 Oldfield, J.V. Williams, R.D., and Wiseman, N.E. "Content-addressable memories for storing and processing recursively subdivided images and trees", *Electronics Letters*, vol. 23, no. 6, (March 1987), pp 262-263.

11 Oliver, M.A., and Wiseman, N.E. "Operations on Quadtree Encoded Images", *The Computer Journal*, vol. 26, no. 1, (March 1983), pp 83-91.

12 Oliver, M.A., and Wiseman, N.E. "Operations on Quadtree Leaves and Related Image Areas", *The Computer Journal*, vol. 26, no. 4, (December 1983), pp 375-380.

13 Rogers, D.F. "Procedural elements for computer graphics", McGraw-Hill, New York, 1985.

14 Samet, H. "Connected Component Labelling Using Quadtrees", *Journal of the ACM, vol. 28, no. 3, (July 1981), pp 487-501.*

15 Samet, H. "Algorithms for the Conversion of Quadtrees to Rasters", *Computer Vision, Graphics, and Image Processing*, vol. 26, no. 1, (April 1984), pp 1-16.

16 Samet, H. "The Quadtree and Related Hierarchical Data Structures", *ACM Computing Surveys*, vol. 16, no. 2, (June 1984), pp 187-260.

17 Samet, H., and Tamminen, M. "Computing Geometric Properties of Images Represented by Linear Quadtrees", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 7, no. 2, (March 1985), pp 229-239.

18 Seeber, R.R., and Lindquist, A.B. "Associative Memory with Ordered Retrieval", *IBM Journal of Research and Development*, vol. 6, no. 1, (January 1962), pp 126-136.

19 Shaffer, C.A., and Samet, H. "Optimal Quadtree Construction Algorithms", *Computer Vision, Graphics, and Image Processing*, vol. 37, no. 3, (March 1987), pp 402-419.

20 Shneier, M. "Calculations of Geometric Properties Using Quadtrees", *Computer Graphics and Image Processing*, vol. 16, no. 3, (July 1981), pp 296-302.

21 Snyder, W.E., and Savage, C.D. "Content-Addressable Read/Write Memories for Image Analysis", *IEEE Transactions on Computers*, vol. C-31, no. 10, (October 1982), pp 963-968.

22 Troullinos, N.B. "A Static Content Addressable Memory Cell with Don't Care Capability", *CSE 664 Project Report*, Syracuse University, December 1986.

23 Unnikrishnan, A., Venkatesh, Y.V., and Shankar, P. "Connected Component Labelling using Quadtrees — A Bottom-up Approach", *The Computer Journal*, vol. 30, no. 2, (April 1987), pp 176-182.

24 Wade, J.P. et al. "The MIT Database Accelerator: 2K-trit Circuit Design", In *Proceedings, Symposium on VLSI Circuits* (Karuizawa, Japan, May 1987) pp 39-40.

25 Williams, R.D. "Organisation and Analysis of Spatial Data", *Ph.D. Thesis* (in preparation), Computer Laboratory, University of Cambridge, (1988).