Yes, An SIMD Machine Can Be Used For AI

Ruven Brooks
Rosalyn Lum
Systems Research Division
ITT - Advanced Technology Center
1 Research Drive
Shelton, CT. 06484
(203) 929-7341

Abstract

Nearly all of the proposed parallel architectures for artificial intelligence applications use a multiple instruction, multiple data stream (MIMD) approach. While this approach offers the greatest opportunity for ultimate exploitation of parallelism, it has been difficult to actually achieve a high level of parallelism in practice because most of the existing algorithms require a higher interprocessor communications bandwidth than the hardware can achieve.

An alternate approach to parallelism is the use of single instruction, multiple data stream parallelism (SIMD) machines. While SIMD machines do not offer the same ultimate exploitation of parallelism as MIMD architectures, they may, in fact, provide more useable parallelism. This is because they can be treated as serial machines with very long data words, so that existing algorithms may be more readily adapted in ways which better use available parallelism. We illustrate this concept with the Cellular Array Processor (CAP) being developed at the ITT Advanced Technology Center. This SIMD architecture is based on a rectangular processing array which accesses a single memory and which has very high speed data paths among the processing elements. We discuss the implementation and manipulation of data for two applications on the CAP; the OPS5 production system interpreter and a representation of an associative network. The expected performance of the CAP in these applications will also be presented.

Introduction

Recently, work on more powerful hardware and architectures for artificial intelligence applications has focused on parallelism as a solution to the problem of the looming limits to increased circuit speed in monoprocessor architectures (Deering, 1984). While there are a great many ways in which parallelism as a concept can be applied to architectures, the variety of parallelism that has attracted the predominant attention of the artificial intelligence community has been one in which multiple processors are each executing their own instruction stream on their own set of data. Among the designs based on this approach are C.MMP (Wulf and Bell, 1972) and the DADO architecture (Stolfo, 1984).

The major problem that these designs present is one of memory access. Since memory speed is as big a bottleneck as processor speed, designs in which all processors share a common memory offer little opportunity for parallelism. Consequently, current work on parallelism is focusing on distributing the computation among processing elements in such a way that each processor operates on its own set of data. Work on parallel inference systems, for example, has attempted to develop algorithms such that all the information needed to infer each, single clause is on one processor. (This may involve copies of some of the information). A conjunction or disjunction is then computed by having each processor work separately on one of the clauses and combining the results. Such "divide and combine" approaches are also the basis for parallelism in other paradigms, such as the one used by Stolfo et. al. for OPS5 (Stolfo, 1984).

While such algorithms can achieve a high degree of parallelism when they succeed, they do not work universally. Frequently, there is a trade-off between processor utilisation and communication time. Solutions that maximize processor utilization often result in slower overall computation time because of the time taken to communicate information - both instructions and data - among processors. In the worst case, the performance is inferior to that of single processor systems. Increasing the communication bandwidth is rarely cost effective and, ultimately, will run up against the same hardware factors which limit memory bandwidth in single processor designs.

While high performance multiple instruction, multiple data stream (MIMD) solutions may ultimately be found for many significant problems, equally high performance may be obtainable from architectures which trade reduced information communication requirements against a reduced degree of parallelism. We have been working with one such architecture, an SIMD (single instruction, multiple data

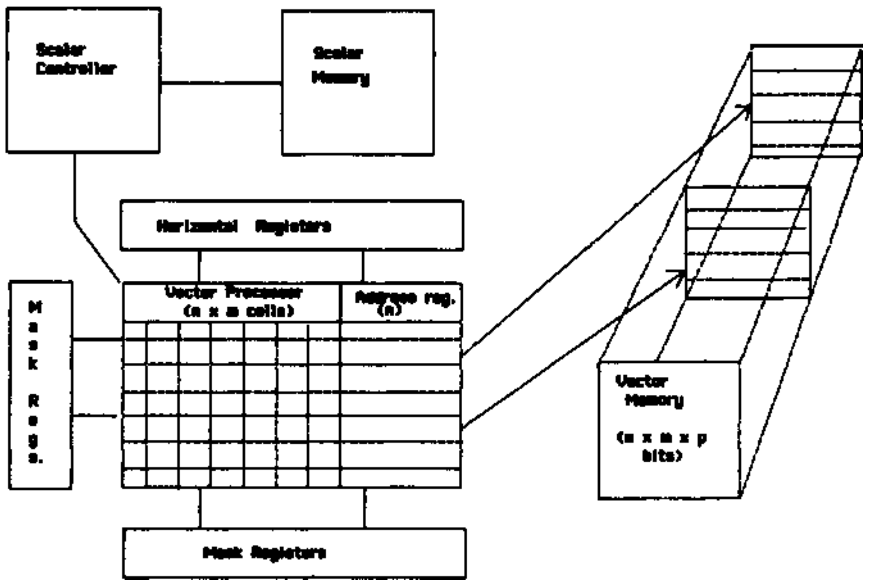


Figure 1.

stream) machine, which shows considerable potential for running some of the algorithms important to work in artificial intelligence.

The Cellular Array Processor

Architecture

The Cellular Array Processor (CAP) is a general purpose SIMD computer. The overall architecture of the CAP is shown in Figure 1. Originally, this architecture was designed for large scale numerical operations, particularly those involving floating point and vector operations. Adapting it to artificial intelligence algorithms was facilitated by a change to the memory access structure.

The heart of the CAP is the associative array, composed of a square array of processing elements. Each element is a one-bit "bit-slice* processor. The number of elements in the array can be optimized for particular applications in multiples of 16x16; thus, arrays can be built which are 16x32, 32x32 or larger.

Within the array, the processing elements are conceptually organized into rows; each row behaves like a conventional processor with three exceptions: First, all rows execute the same instruction at the same time; this is accomplished by keeping all of the instructions in a single program memory and sending the instruction to all processors at once. A masking mechanism can be used to disable some of the rows or parts of the rows.

Second, there are instructions for data transfer between processors; for example, it is possible to instruct all processors to transfer the contents of one of their registers into a register of the row next to them. It is also possible to broadcast the contents of a register in one row to all of the other rows.

Third, operations can be performed in parallel on parts of a row. For example, a 32 bit row can perform two, simultaneous 16 bit integer adds on each half of a row. Moreover, by setting most-significant and least-significant bits at different points in the row, it is possible to work with arbitrary row partitions. Thus, a 32 bit row could be partitioned into two 10 bit pieces, and a 12 bit piece. (There are some restrictions on partitioning across rows and on minimum partition sizes.)

As with conventional processors, each row has its own memory. Thus, there is effectively a stack of memory planes beneath the plane of processors. The amount of memory addressable by each plane is implementation dependent; the initial version, with 32 bit rows, will allow for 256K bytes per row, for an overall memory of 8M bytes.

Comparision to Other SIMD Architectures

Other SIMD machines (Kuhn & Padua, 1081) have been typically composed of complete processor units with small amounts of primary memory. Communication among processors is via a common bus of limited bandwidth. The problem is partitioned so that each processor executes the same instruction stream but on different parts of the problem (or in the case of production systems, different rules).

If the problem is not partitioned properly or if only part of the problem is executable at any one time, then some of the processors will have to wait until other processors have finished execution. This frequently prevents fully utilizing the speed gained in the parallel implementation of instructions.

The CAP, on the other hand, has a large primary memory for each processing row. Broadcast of data to the rows takes place at instruction execution speed. Since the data is organized along rows rather than into the depth of the memory, it is easier to allocate the data more evenly among the processors. Further, since instruction decoding is done once for the entire array, there is no need to replicate instruction memories. This combination increases the likelihood that the processing array can be kept fully occupied more of the time than has been the case with past SIMD architectures.

Implementation

The current implementation of the CAP architecture is a VLSI chip set consisting of a processor chip, a controller chip, a barrel-shifter chip, a communications processor chip and an input/output processor chip. Each array chip contains 16 one-bit cells. The chip is a custom design currently fabricated in 3-micron, N-well, double metal CMOS. It contains approximately 117,000 transistors, and has an estimated die size of 500x650 mil. die, with 132 I/O pins.

The array chip has been designed to have the following attributes:

- Highly regular structure
- Fixed and floating point arithmetic
- Logic, branching and masking instructions
- 10 MHz sixteen bit addition
- 20 MHz input/output shift rate

In addition, each cell has it own control logic, ALU, a set of single bit registers, and simple I/O processor for external device DMA to off-chip RAM. The amount of off-chip RAM is abo variable with typical sizes being 16K, 64K, and 256K. The total available primary memory for the array is then n x 16 X m; where n is the number of chips times 16 (the number of cells on each chip) times m, the amount of memory attached to each cell.

Applications

To illustrate how the CAP architecture can be used for artificial intelligence applications, we show how the CAP can be used to implement some typical algorithms used in artificial intelligence work: the OPS5/RETE algorithm and association network algorithms. These will be discussed in terms of implementation and performance on the CAP.

OPS5/RETE Algorithm

The RETE algorithm for matching the left-hand sides of rules in the OPS5 forward-chaining production system language is one of the few, well analyzed computations in artificial intelligence (Forgy, Gupta, Newell & Wedig, 1984). Forgy et. al. describe this algorithm as follows:

■The Rete interpreter processes the left-hand sides of the productions prior to executing the system. It compiles the left-hand sides into a network that specifies the computations that the matcher has to perform in order to affect the mapping from changes in working memory to changes in the conflict set. The network is a dataflow graph. The input network consists of changes to working memory encoded in data structures called tokens. Other tokens output from the network specify changes that must be made to the conflict set. As the tokens flow through the network, they activate the nodes, causing them to perform the necessary operations, creating new tokens that pass on to subsequent nodes in the network. The network contains essentially four kinds of nodes:

- Constant-test nodes: These nodes test constant features of working memory elements. They effectively implement a sorting network and process each element added to or deleted from working memory to determine which conditions the element matches.

- Memory nodes: These nodes maintain the matcher's state. They store lists of tokens that match individual conditions or groups of conditions.

- Two-input nodes: These nodes access the information stored by the memory nodes to determine whether groups of conditions are satisfied. For example, a two input node might access the lists of tokens that have been determined to match two conditions of some production individually and determine whether there are any pairs of tokens that match the two conditions together. In general, not all pairs will match because the left-hand side may specify constraints such as consistancy of variable bindings that have to hold between the two conditions. When a two-input node finds two tokens that match simultaneously, it builds a larger token that indicates that fact and passes it to subsequent nodes in the network.

• Terminal nodes. Terminal nodes are concerned with changes to the conflict set. When one of these nodes is activated, it adds a production to or removes a production from the conflict set. The processing performed by the other nodes insures that these nodes are activated only when conflict set changes are required. ■

In their analysis of several, large OPS5 systems, Forgy and his colleagues conclude that the match process is the biggest component of OPS5 run times, since it occurs every time a rule action takes place. Further, processing the memory nodes and the two-input nodes takes the largest portion of match time. They also point out that most changes are localized, with only a small proportion of nodes being activated for any rule action.

CAP Implementation of the Rete Algorithm.

The RETE algorithm can be implemented on the CAP in a form which utilizes the CAP parallelism effectively. For purposes of describing this implementation, we will assume a 32x32 CAP array, although the approach will work the same way for other array sizes.

Constant-test nodes.

These can be handled in two different ways, depending on the characteristics of the tests that are needed.

Many values for a single attribute.

Here, the situation is one in which many rules test the same attribute of a particular working memory element, but where each rule matches a different value, or, alternately, there are many different classes of working memory element and different rules match different classes. In the CAP, this is handled by testing the incoming token against many possible alternatives simultaneously. Assuming that all of the alternative values are stored in one or more planes of the CAP memory, the operations are as follows:

1. Broadcast the value from the incoming token to all rows of the CAP and store it in a register.

2. Load the plane of constants into a CAP register in each row.

3. Compare the two registers in each row.

If there are 32 or fewer alternatives to be tested, then all of the tests can be made in one pass through these steps. If there are more, then steps 2 and 3 can be repeated for additional planes of alternatives.

Many constant values in a single LHS pattern.

Sometimes there will be many constant tests to be made on a single working memory element; this would occur if the working memory elements were very long with many attributes. In this situation, there is an alternative set of CAP operations. To use them, all of the attribute values that belong to one left-hand side element are placed in the same planes of CAP memory. The operations are then the same as those given above.

Both of these methods can be combined in a single search tree. Thus, it may be desirable to test the classes of all of the elements at the top of the tree but then switch to testing multiple attributes within an element further down in the tree.

Memory and two-way nodes.

In these nodes, the problem is one of testing an incoming token against the set of tokens that have arrived at the node earlier and are being stored there. In the CAP implementation, the node memory will be set up in the following manner:

1. If the tokens are smaller than the row length of the CAP array, they will be placed contiguously within a plane of CAP memory. If the memory is large, then multiple planes of memory may be used, but the planes need not be contiguous.

2. If the tokens are larger than a row of CAP array, the tokens will be split across multiple planes of memory, but each part of a token will occupy the same relative position within its plane.

With this arrangement, the following sequence of CAP operations can be used to test an incoming token against the tokens which are already in a node memory:

1. Load the first part of the incoming token into the vertical data register.

2. Broadcast this value to all rows of the CAP and store it in a register.

3. Load the plane containing the first part of the tokens stored at the node into a CAP register in each row.

4. Compare the two registers in each row.

5. If there are no more parts, the matching rows are the memory tokens which match.

6. If there are more parts, mask out the rows which did not match in the previous steps, and repeat the load and compare operations with successive parts.

7. Repeat steps 2 through 5 for all of the planes containing additional stored tokens.

Expected Performance

The CAP implementation will affect the performance of the RETE match algorithm in two ways: First, it will effectively reduce the number of constant nodes in the tree. Currently, the number of constant nodes in the tree is proportionate to the product of the number of different attributes and the number of different values those attributes can take on. In the CAP implementation, it will be proportionate only to the number of different attributes, provided the number of attributes is less than the row size of the CAP.

Second, and, perhaps, most importantly, there will be a speed up in the performance of two-input nodes. Since the time in processing these nodes is spent comparing an incoming token to the tokens already stored at the node, the ability of the CAP to do multiple comparisons simultaneously could significantly improve performance. The amount of performance improvement will depend on the average number of tokens being stored in the node memories and will increase as the number increases. A factor equal to the number of rows of the CAP will be the upper bound; thus, a 32 row CAP will yield a maximum of a 32 fold improvement.

Overall, maximum parallelism will be achieved when the number of memory nodes is large and these nodes are filled much of the time and when the number of one-input (constant) nodes is large. Worst case performance will occur when the number of one-input nodes is small and there are many two-input nodes with no memories. This would be the situation if no variables were used on the left-hand sides of rules, but the left-hand sides had a large number of conditions.

Comparision to Other OPS5 Architectures

At least two other architectures have been proposed for forward production systems, particularly OPS5. (The PSM group (Forgy et. al., 1984) architecture is still in the the design analysis stage, so that no figures on the implementation are yet available.) The DADO architecture has been well described and several different algorithms have been proposed for it, so that a preliminary comparision is possible.

The DADO architecture is based on a relatively large number of processors, such as 1023, connected in a binary tree architecture (Stolfo, 1984). Each processor is currently implemented using a commercially available processor, a modest amount of local memory (16K bytes) and a semi-custom I/O switch. Since each of the processors operates in a conventional, serial manner, the system relies for speed improvement on distributing the computation out among the processors.

If each of the 1023 processors has an 8 bit word width, and if maximum parallelism is achieved, then the maximum processing capacity will be 1024 bytes at the instruction execution rate of the individual processors. A 64 row by 64 bit CAP array, running at higher clock rate, offers equivalent maximum processing capability with many fewer components. One important issue, then, is the extent to which the parallelism is useable.

As was noted earlier, the CAP will achieve best performance with large average node memory sizes. Forgy et. al. (1984) point out that this is the situation in the large OPS5 systems they analyzed. The DADO algorithm that most closely corresponds to this situation is algorithm 4, in which leaves of the tree contain one input tests, and the results of the test are broadcast up the tree. This means that, in general, fewer than 256 processors will be available for simultaneous work on two input nodes, that is, on any instruction cycle, only 256 bytes of data bandwidth will be available. In contrast, the full 512 bytes of CAP bandwidth will be useable on these tests. Hence, the CAP should achieve roughly twice the performance of the DADO machine.

Another difference will be in the memory available for large working memories or numbers of rules. The current DADO design partitions the memory into relatively small units of 16K per processor. If a single node memory becomes very large, it may exceed the storage capacity of an individual processor, while still using only a small fraction of the overall memory of the machine. In the CAP, in contrast, the memory behind each row is relatively large - 256K for the initial implementation • and it can be dynamically allocated among the different node memories. Thus, the CAP is less likely to run into memory allocation problems on large OPS5 systems.

Association Networks

Association networks, also called semantic networks, are one of the most widely investigated approaches to knowledge representation in artificial intelligence. A wide diversity of approaches and systems exist (Findler, 1979); in some, for example, links between nodes indicate relationships among nodes; in others, links indicate positions in logical predicates. At the implementation level, though, nearly all of the formalisms use the following operations:

1. Find all of the node(s) reacheable from a given node using a specified set of links or link types.

2. Find a path between a pair of nodes.

There may be restrictions on the length of the paths and/or on the types of links used.

In general, the links are stored as lists of pointers such as LISP property lists. Consequently, the time to perform the operations tends to be heavily dependent of the total number of links emanating from a node since it may be necessary to traverse the entire list to find a link of a particular type.

The CAP can also be used to process association networks much faster than can be done with conventional serial processors. The approach that is used is to take advantage of the CAP's ability to simultaneously perform operations on individual bits. Instead of representing the links in the network as pointers, they are stored as bit matrices, one matrix for each type of link. Each node in the network is represented by a row and a column in each matrix, and a 1 in a row-column intersection indicates that a link of the given type exists for the two corresponding objects.

Figure 2 gives an example of such a matrix. The relationships are predefined in the vertical and horizontal directions and the axis in the x and y directions represent the nodes of the list. In this example, the vertical or columnwise relationship represent the children of the member in the horizontal axis. A 1 in the searched column will yield all children of a certain parent. This axis can also be discriminated further by male/female categories whereby son/daughter information can be ascertained. The horizontal or row-wise relationship was defined as the parents of the searched person whose identity belongs to that row.

From this simple example, it can be seen that this type of representation also commends itself to depth or iterative searches. By recursively tracing links of previous links, ancestral information for *x* number of generations can be found. Family trees or paths may be traced and on either parent's lineage. In addition,

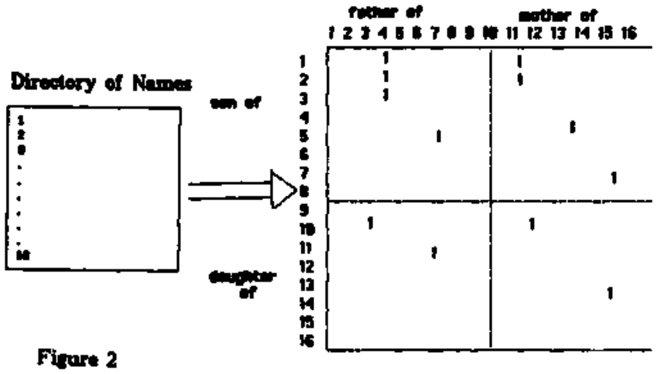


Figure 2

the search space for any row or column is restricted to only the pages which fall in the row or column of the searched individual. For example, if the query is to find all the children of *namex* and *namex* is on the fourth page, 10th column (Figure 3), then only the pages in the shaded area need be searched. Because the matrices are sparse, it is important to be able to determine quickly if a column is blank.

Expected Performance.

Assuming a 32x32 CAP configuration, the CAP will be able to simultaneously search 32 elements of a row or column in a single instruction sequence. Thus, it will offer a substantial performance improvement over uniprocessor architectures that test one element per instruction sequence.

Comparision to Other Architectures.

At least two other architectures have been proposed for similar tasks, the Connection Machine (Hillis, 1981) and SNAP (Dixit & Moldovan, 1984). Both of these are VLSI implementations of associative memory architectures in which there is a very large array of processing elements with nearest neighbor interconnections. In both designs, the array is driven by an external processor which uses the array for data storage and retrieval.

In these designs, there is a mapping from nodes in the association network to individual processing elements. In the Connection Machine, in which processing elements have 14 connections to their neighbors, a node is represented by a tree of processing elements in which the links in the tree are implemented as addresses stored in state vector of the processing element. In SNAP, the cells have a four-way connection and a content addressable memory that is used to store links; each processing element represents one node.

The Connection Machine is partially an SIMD design, since all cells share the same rule table. Though it is not as clear from the published descriptions, it appears that all SNAP cells also execute the same instructions. In contrast to the CAP

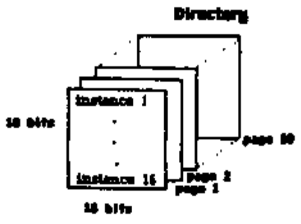
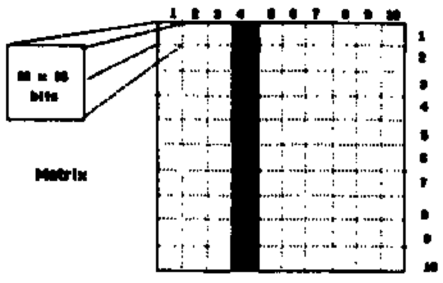


Figure 3



though, the individual cells may operate asynchonously in both designs. Thus, they may be simultaneously operating on different parts of an association network.

The performance of these designs relative to the CAP will depend on several factors: The first is overall array size. Since these machines are driven by external processors, loading the arrays will depend on the speed of these processors. If the size of the network exceeds the size of the array, the effective processing speed may be considerably reduced. The cost effectiveness of array size will, in turn, depend on the effectiveness of the VLSI implementation. Since the CAP architecture uses conventional RAM, the ability to directly access large networks may compensate for the reduced number of active processing elements.

A second important factor will be the structure of the particular network and its effect on array congestion. If the network is partitioned into largely independent graphs with only a few nodes connecting the areas, then these connecting nodes can effectively be a bottleneck to many processing array operations and greatly reduce the effective parallelism.

Third, the Connection Machine and SNAP designs are only intended to search and manipulate links among nodes. Any processing done on the nodes themselves are, again, handled by an external processor. In contrast, the CAP is capable of a wide variety of arithmetic and logical operations, so that no interaction with an external processor is needed.

Conclusions

The previous analyses have been intended to show that the CAP is, at least, a competitive architecture for two, important artificial intelligence algorithms. Other architectures for the same algorithms all have significantly more processing power when measured in terms of overall processing bandwidth, but potentially suffer from problems of bandwidth utilization caused by restrictions on common access to data. In contrast, the large memory of the CAP and the large bandwidth of access to this memory may more than compensate for the reduced opportunities for parallelism.
Oh, yes, it does do floating point.

Finally, in making these analyses, we have deliberately understressed the CAP's ability to do high speed arithmetic processing. For byte-wise integer operations, such as might be involved in picture processing, a 32x32 array could achieve a speed of over 1,200 MOPS. For floating point operations, the speed approaches 10 Mflops. The ability to perform these operations on the same processor with high speed rule firing or fast association network searching might be very valuable in intelligent signal processing tasks such as scene analysis or speech recognition.

References

Deering, Michael. Hardware and software architectures for efficient AI. *Proceedings, National Conference on Artificial Intelligence,* American Artificial Intelligence Association, 1084.

Dixit, V. and Moldovan, D. I. Discrete relaxation on SNAP. *Proceeding; The First Conference on Artificial Inteligence Applications* I.E.E.E. Computer Society, 1084.

Findler, N. V. (Ed.) *Associative Networks: The representation and use of knowledge by computers.* Academic Press, New York, 1979.

Forgy, C; Gupta, A.; Newell, A.; and Wedig, R. Initial assessment of architectures for production systems. *Proceedings, National Conference on Artificial Intelligence,* American Artificial Intelligence Association, 1984.

Hillis, W. Daniel. The Connection Machine. A.I. Memo 648. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1981.

Kuhn, R. H. and Padua, D. A. *Tutorial on Parallel Processing,* I.E.E.E. Computer Society, 1981.

Stolfo, S. Five parallel algorithms for production system execution on the DADO machine. *Proceedings, National Conference on Artificial Intelligence,* American Artificial Intelligence Association, 1984.

Wulf, W. A. & Bell, C. G. C.mmp - a multi-mini-processor. *AFIPS Conference Proceedings,* Vol. 41, part II, FJCC, 1972, 765-777.