Efficient Associative SIMD Processing for Non-Tabular Structured Data

Jalpesh K. Chitalia and Robert A. Walker
Computer Science Department
Kent State University
Kent, OH 44242
{jchitali, walker} @ cs.kent.edu

***Abstract* - With recent advances in VLSI technology, it is easy to develop chips containing hundreds or thousands of processors – in effect, SIMD processing on a chip. Associative SIMD processing goes further, providing efficient parallel methods for searching tabular data. The techniques described here show that associative processing can also be extended to efficiently process non-tabular structured data (e.g., trees), as we demonstrate by supporting *structure codes* in our associative ASC Processor.**

I. INTRODUCTION

At Kent State University, research has long focused on algorithms, software, and hardware for *associative processing* [5] – a form of SIMD processing that accesses data by content rather than address. Associative processing is particularly suited for applications that represent data in tabular fashion, such as database processing [3], image processing [7], genome matching, molecular similarity analysis, and air traffic control [4].

Over the past few years, our research group has been developing a new 8-bit associative RISC processor [6], called the ASC (ASsociative Computing) processor, using a modern FPGA implementation (see Fig. 1). Our current ASC processor [6] supports 36 8-bit Processing Elements (PEs) as a proof of concept on a small million-gate Altera FPGA. However, larger FPGAs or multiple boards could easily support hundreds or thousands of PEs.

II. ASSOCIATIVE COMPUTING

Associative computing primarily differs from SIMD computing in its emphasis on efficient searching. As illustrated in the diagram of our ASC processor in Fig. 1, each Processing Element (PE) of a SIMD associative computing array operates under the control of an Instruction Stream Control Unit (IS).

The broadcast and reduction network between the Instruction Stream and the PEs is shown in the figure as a simple data bus (the actual reduction network is more complicated), connecting the Common Registers with the PEs. Decoded instructions are sent to the PEs in the form of microcode, as if to an ALU, over the instruction bus. Under the control of the Mask Stack, PEs can accept or ignore the instruction or data broadcast by the Instruction Stream.

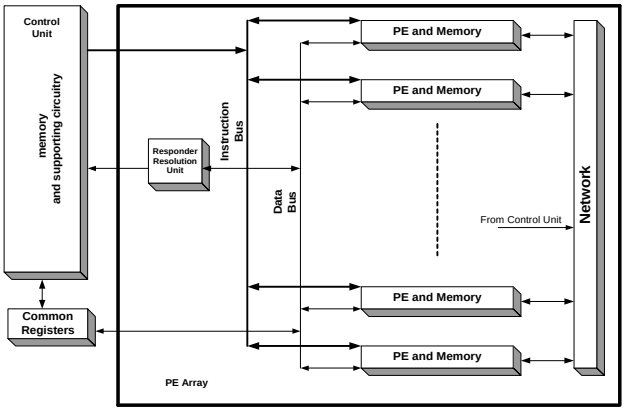

Fig. 1. ASC Associative Computing Processor

As in most SIMD arrays, each PE in our ASC processor can access only its own local memory. While the PEs could theoretically communicate using the broadcast and reduction network, that method may be too inefficient for some algorithms. For example, edge detection for image processing [7], or matrix multiplication[10], require data to be exchanged in a fixed pattern. To facilitate such an exchange our processor uses a PE network that can be configured as either a linear array or a 2D mesh [7].

Now suppose each PE stores a record of tabular data in its memory. In an *associative search* the Instruction Stream Control Unit broadcasts a search key to each PE, and then directs each PE to look for that key in its local memory. If the search key is found, that PE is designated a *responder*, and subsequent processing can be limited to only those responders. This selective processing is implemented using Masked instructions and a Mask Stack in each PE.

Consider associative processing applied to the sample student database shown in Fig. 2, where each row of data can be stored in the memory of a separate PE. If the teacher wants to know how many students have a grade above 90, the following ASC code fragment could be used:

```
0.  setscope all[$]
1.          for xx in grade[$] > 90 do
2.                  count = count + 1;
3.          endfor xx;
4.      endsetscope;
```

				Search		STEP1		STEP2	
	Student Name	ID	Grade	Mask	RSPD	Mask	RSPD	Mask	RSPD
PE0	John Smith	07	66	0	0	0	0	0	0
PE1	Gary Heath	05	95	1	1	1	0	0	0
PE2	Peter Smith	11	87	0	0	0	0	0	0
PE3	John Smith	04	78	0	0	0	0	0	0
PE4	Tarry Stanley	02	100	1	1	0	1	1	0
PE5	Will Hanson	01	84	0	0	0	0	0	0
PE6	Jane Antony	06	64	0	0	0	0	0	0
PE7	Mark Bloggs	13	88	0	0	0	0	0	0
PE8	Gill Pister	09	75	0	0	0	0	0	0
PE9	Min Lee	10	83	0	0	0	0	0	0
PE10	Goby Carmen	03	83	0	0	0	0	0	0
PE11	Gillian Roger	08	26	0	0	0	0	0	0

Fig. 2. Sample Student Database

Our ASC processor executes this code as follows. Line 0 directs all PEs to listen to the Instruction Stream (IS), which broadcasts instructions (implementing Line 1) telling them to compare the value of grade in their local memories with the constant 90. In the PEs where this comparison is successful, the PE is designated a responder, meaning the Responder bit is set, and the top of the Mask Stack is also set. The Mask Stack can then be used in conjunction with Masked instructions to limit further processing to only those responders.

Running this code on the sample student database in Fig. 2 finds two responders, shown shaded in the figure. Due to the for-loop in Lines 1-3, these responders now have to be visited *sequentially* under the control of a Step instruction. This Step instruction chooses one of these responders arbitrarily. This arbitrary responder pushes 1 onto the top of its Mask Stack, while all other PEs push 0, and turns off its responder bit, preventing this PE from being revisited later. Line 2 is then executed for this masked PE, updating `count` (a shared variable stored in the Instruction Stream register or main memory). When Line 3 is reached, another responder is selected arbitrarily, continuing until all responders are visited. The columns labeled "STEP 1" and "STEP 2" in Fig. 2 show this process.

Our ASC processor can also use associative processing to search for a maximum or a minimum value in a specified field across all PEs. For example, in the sample student database it could search for a student with the highest grade. Instead of sorting or comparing all the values, two constant time operations, Max and Min, are implemented using Falkoff's algorithm [1, 5]. Using these operations, the PE with the extreme value is "masked" for further processing.

III. (Data) Structure Codes

The previous section demonstrates the advantage of associative SIMD processing over pure SIMD processing in applications that emphasize searching. However, associative processing, like SIMD processing in general, represents data in tabular fashion, which can be problematic for some algorithms.

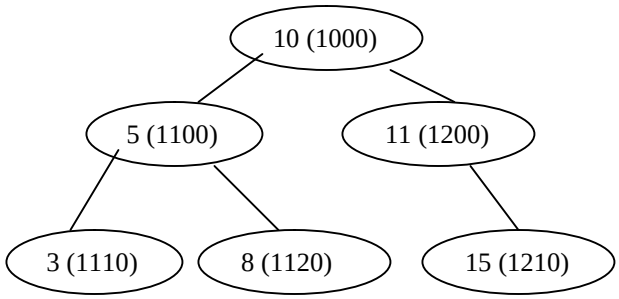

Fig. 3. An Example Linked List

More specifically, data parallel algorithms generally necessitate the division of data amongst the PEs. Furthermore, they rely on the premise that data is distributed evenly. While this is true for some types of data, it is not true for data structures such as trees. While it can be argued that trees can be avoided in many situations using associative processing (e.g., trees are not needed for efficient searching), there are algorithms and applications that inherently use trees, graphs, or generic linked lists as a data structure. Supporting these data structures increases the utility of associative processing.

Given a reference node in a tree, suppose it is necessary to find the left sibling of that node. In a conventional programming language or architecture without parent pointers, the tree must be traversed from the root until the parent of the reference node in question is found; and then the left to reference-node pointer can be selected (if one exists). This or similar searching is a necessary operation in most algorithms using generic linked lists.

In associative processing, each node of the linked list can be stored on a separate PE. However, some additional data must be added for each of the nodes to capture the structure of the graph. We refer to this additional data as a *data structure code*, or simply a structure code. Using these structure codes, any complex data structure could be either represented in a table or a nested linked list.

A structure code is defined in [5] as an address function representing positions of objects in the data structure space on a content addressable machine. For example, Fig. 3 shows an example data structure space (a nested linked list), with each node annotated with its corresponding structure code. Each digit in the structure code represents its level in the tree (i.e., the most significant digit represents the root, the next digit the root's children, etc.), and the value of the digit represents its position from the left (i.e., 1 represents the leftmost sibling, 2 the second leftmost, etc.). Structure codes constructed in this way are unique and efficiently represent the position of their respective nodes in the data structure.

IV. Operations Using Structure Codes

The ASC programming language [5], defined to support associative computing, includes some high level instructions that use structure codes. We have implemented most of

those instructions on our ASC processor, and are adding the remainder now. This section will define those instructions.

A constant-time parallel operation, **prvdex,** can be defined to find the index of the left sibling (actually, to set the node as a responder). If each node of the linked list is stored in a separate PE, **prvdex** can be implemented as follows:

- Find all nodes with lesser structure codes than the reference node (at the same level in heirarchy)
- Find maximum of this result
- Mark this node

Other data structure operations can also be defined. Similar to **prvdex**, **nxtdex** finds the index of the right sibling, and **sibdex** finds the index of both siblings. Complementary to these, the parallel operations **prvval** and **nxtval** return the actual structure code for previous and next nodes, respectively, instead of finding their index and designating the resultant PEs as responders.

Of the instructions defined above, **prvdex**, **nxtdex**, and **sibdex** are the most useful for associative computing, as the responders that represent the result can be masked and then processed further. However, **prvval** and **nxtval** are occasionally useful, as are the scalar instructions to manipulate the structure codes listed in Fig. 4 (where the input node assumed is 1220).

In these scalar structure code manipulation instructions, "cd" refers to "child" and "tr" refers to "truncate". Thus **fstcd** finds the structure code of the reference node's first child, **nxtcd** and **prvcd** find the structure code of the reference node's siblings (next child and previous child, respectively), and **trncd** and **trnacd** finds the structure code of the reference node's parent and root of the tree. Note that these instructions are not parallel operations, but simply scalar instructions that manipulate the appropriate digit in the input structure code to produce a new result structure code.

Although our ASC processor supports the basic structure code instructions defined above, it does not have efficient support for variable-length structure codes or large fixed-length structure codes. One way to implement variable-length structure code is using one byte per digit of code.

V. Implementation

Our current ASC processor successfully implements most of the (scalar) instructions in Fig. 4 and all the parallel instructions noted above. The structure codes for each of the nodes in the nested linked list are sequentially input to (or output from) a PE one by one, using the broadcast and reduction network (for parallel instructions).

The parallel instructions are of two types: one type outputs a parallel-vector of flags (e.g., **sibdex**), and the other outputs a resultant structure code (e.g., **nxtval**). In either case, the processing is same with a difference of 'storing the code' or 'marking the flag' for a responder.

The scalar instructions (e.g., **nxtcd**) are executed on the Instruction Stream and do not make any use of parallel resources. These instructions are simple mathematical manipulations on an input structure code (this is not necessarily a reference node) that output a resultant structure code. However, they can be useful when assigning structure codes to an incoming data structure element.

Operation	Result
fstcd	1221
nxtcd	1230
prvcd	1210
trncd	1200
trnacd	1000

Fig. 4. Structure Codes Operations
(Result for Input Node 1220)

Our current implementation of structure codes also supports a variable-length structure code so as to accommodate up to 255 children, and unlimited levels of nesting (bounded by memory). Notwithstanding the increase in the space complexity of the algorithm, a simple representation of one byte per digit of structure code is used. This representation requires more space, but compensates for adding a multiplier-divider unit to each of the PEs.

Using *inorder* and *preorder* traversals, a binary tree can be constructed in O(log n) time [10] for SIMD computers, assuming an ideal PRAM model. This method produces a complete view of the tree structure, using different strings for left-child, right-child, and parent, but works only for binary trees. In contrast, structure codes and the operations defined above on an associative SIMD processor permit constant time operations on generic trees.

Furthermore, the PRAM model takes constant time in finding the parent of a reference node, and O (log d) time where d is the maximum depth of any tree in the forest [10]. The ASC model augmented with 'structure code' requires constant time for both these operations.

VI. Applications

It is commonly believed that SIMD processing is most appropriate for applications that involve massive parallelism, such as weather forecasting, and that associative SIMD processing is only suitable for relational database processing. However, associative SIMD processing's constant-time (irrespective of the number of input elements) searching and min/max search is very powerful, and when implemented on modern high-density FPGAs can provide dedicated co-processors or embedded processors for a wide variety of applications. A commercially available supercomputer has also been built using multiple FPGA boards.

Structure codes expand the use of associative processing beyond traditional tabular data structures, providing support for ubiquitous data structures such as trees. Compilers require parse trees and directed acyclic graphs, and computer games require complex search trees.

Database operations such as multi-valued attributes in object-relational databases can be implemented efficiently using linked lists. For example, consider an XML [9] representation of a database, as shown in Fig. 5. Two nodes

```
<person>
    <name>
        <last=”Wright”/>
         <first=”John”/>
     </name>
      <email=”jwright@mail.com/>
      <phone=”13301234567”/>
      <phone=”14401234567”/>
</person>
```

Fig. 5. Example XML Mark-up for “address-book” Record

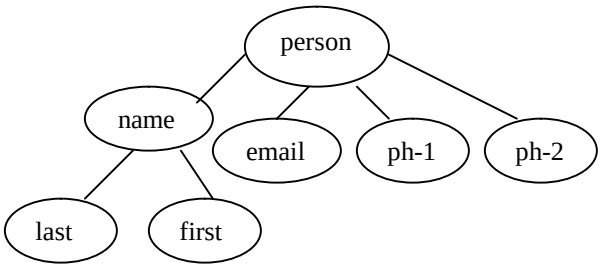

Fig. 6. Equivalent XML Parse Tree for Fig. 5.

on the network could exchange data in the form of markup, which can easily be parsed by an XML parser to give an XML parse tree (shown in Fig. 6).

For the sake of simplicity the values of the fields in nodes are discarded in Fig. 6. This tree, like any other tree as shown in Fig, 3, can be represented using structure codes. With this representation, the email address can be found in constant time through a simple associative directory search for last-name=”Wright”, selection of a “parent” node (**trncd**), and selection of “right sibling” (**nxtcd**). In contrast, the PRAM model of computation would require computing an entire sequence of right-siblings. This takes O(log n) time [10], as compared to O(1) time on the ASC processor with as many PEs.

Without losing the possible object-relational context, the dataset could be represented using structure codes. Database support for multi-valued attributes and nested relations is shown in [2] for the ASC model, and another associative processor has previously demonstrated an efficient implementation for a relational database in business applications [3]. Finally, an advanced database application in the area of homeland security is shown in [2], with support for nested and flexible relations uses a coding scheme of structure codes.

As a database example, suppose a customer at a store purchases multiple items. In such a case, a relation can be defined with the purchase's receipt number and the purchased item's barcode (possibly also information identifying the customer) as a composite primary key in a relational database. Alternatively, and intuitively, the purchase's receipt number (and the customer's id information) as the primary key and a ‘set of purchased items' barcodes’, together, represents the same information in an object-relational database.

The use of structure code preserves the realistic, object-relational structure of the database, and still gives the same efficient relational view to the architecture. Since HTML is a simplified or a constrained version of XML, web pages can also be represented in this tabular format, and rendered as easily.

VII. Conclusions and Future Work

This paper has briefly described associative SIMD processing and our ASC associative processor. We have shown how associative SIMD processing can also be extended to efficiently process non-tabular structured data (e.g., linked lists or trees) using structure codes, and have described the support for structure codes in our ASC associative processor. We briefly showed how support for linked lists can be beneficial, citing their use in an object-relational databases.

We are currently cleaning up our ASC processor’s architecture [8, 10], making the instruction set more regular and improving the processor’s efficiency. As part of this process, we are planning to add a multiplier and divider for each PE. With the use of a divider, byte representation of each digit in the structure codes can be omitted. Other current work is focused on efficiently supporting larger and variable-length structure codes, and on developing applications that use those structure codes.

VIII. Acknowledgments

The authors would like to thank Dr. Jerry Potter, Kevin Schaffer, and Lei Xie, as well as the Parallel Processing Group in general, for many helpful discussions.

IX. References

[1] A. Falkoff, “Algorithms for Parallel Search Memories”, *Journal of Associative Computing*, March 1962.

[2] Eva Gustafsson, *An Associative Database on a Parallel Computer*, Master’s Thesis, Mathematical Sciences Department, Kent State University, 1986.

[3] J. Storrs Hall, Donald E. Smith, and Saul Y. Levy, “Database Mining and Matching in the Rutgers CAM”, in *Associative Processing and Processors*, IEEE CS Press, 1997.

[4] Will Meilander, Mingxian Jin, and Johnnie W. Baker, “Tractable Real-Time Air Traffic Control Automation”, in *Proc. of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2002.

[5] Jerry L. Potter, *Associative Computing: A Computing Paradigm*, Plenum Publishing, 1992.

[6] Hong Wang, Lei Xie, Meiduo Wu, and Robert Walker, “A Scalable Associative Processor with Applications in Database and Image Processing”, in *Proc. of the 18th International Parallel and Distributed Processing Symposium (Workshop in Massively Parallel Processing)*, April 2004.

[7] Lei Xie, *Implementing a PE Interconnection Network for a FPGA-based Associative Computer*, Master’s Thesis, Computer Science Department, Kent State University, April 2004.

[8] Kevin Schaffer, *Developing a Practical Instruction Set for a RISC-based Associative Processor*, Master’s Thesis, Computer Science Department, Kent State University, April 2003.

[9] A. Silberschatz, H. Korth, S. Sudarshan, *Database System Concepts*, Fourth Edition, McGraw-Hill, New York.

[10] S. Akl, *Parallel Computation Models and Methods*, Prentice-Hall. New Jersey. 1997.