A Scalable Associative Processor with Applications in Database and Image Processing

Hong Wang, Lei Xie, Meiduo Wu, and Robert Walker
Computer Science Department
Kent State University
Kent, OH 44242
{hwang1, lxie} @ kent.edu, {mwu, walker} @ cs.kent.edu

Abstract

This paper describes the implementation and use of a dedicated associative SIMD co-processor ideally suited for many applications such as database processing, image processing, genome matching, or molecular similarity analysis. The concept of associative SIMD processing is introduced, and differentiated from other associative and SIMD techniques. Then our ASC (ASsociative Computing) processor is briefly described, along with its implementation of associative SIMD processing. Finally, we demonstrate the use of our ASC processor on relational database processing and on the image processing operation of edge detection.

1. Introduction

This paper takes a new look at *associative computing*, a variation of SIMD processing developed over 30 years ago. While associative processing, and SIMD processing in general, is currently out of vogue, recent advances in FPGA technology permit hundreds or thousands of processors on a single chip, in effect an easily realizable "processor in memory" configuration. These hundreds of SIMD processors, together with associative search techniques, can be used as a ***dedicated associative SIMD co-processor*** ideally suited for many applications such as database processing, image processing, genome matching, or molecular similarity analysis.

A number of variations on associative processing [1, 2] have been explored over the years. One variation [3, 4] used an associative memory to locate data records by content rather than address, and provides various reduction operations via a combinational network. Another variation [5, 6] used SIMD techniques to not only locate multiple data records but to also process those data records in parallel. Still other variations [7] have explored the use of MSIMD (multiple SIMD) techniques to permit processing each data record in a different manner. Usually oriented toward database or pattern matching of some kind, most early associative computers were implemented using very simple single-bit processors, necessitating bit-serial processing of multi-bit data words. However, some systems explored the use of wider processors (e.g., 4-bit or 8-bit processors) or more powerful processors.

Associative processing at Kent State University (KSU) [1, 8] has its roots in associative system development at Goodyear Aerospace Corporation, in particular Goodyear's STARAN [9] and ASPRO [10] computers. Those early associative systems used TTL-based single-bit processors, and were supported by programming languages specially designed to permit efficient SIMD-style associative processing. As various individuals moved from Goodyear to KSU, more recent work at KSU has continued to explore associative processing, in particular demonstrating the power of this computing paradigm as compared to traditional SIMD, and even MIMD, computing [11].

Complementing that work on associative model and algorithmic development, our research group is developing a new 8-bit associative RISC processor, called the ASC (ASsociative Computing) processor, using a modern FPGA implementation. Our early prototypes [12, 13] were limited to only 4 8-bit Processing Elements (PEs) as a proof of concept, but the current version supports 36 8-bit PEs on a million-gate Altera APEX FPGA, and can easily be scaled to several hundred 8-bit PEs on larger FPGAs. Alternatively, that same FPGA could support thousands of 1-bit PEs, or a multiple-FPGA board could support thousands of our 8-bit PEs, but these variations are the subject of future work.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to associative processing, and Section 3 describing the implementation of associative processing in our ASC co-processor. Section 4 then demonstrates the use of that processor in two real applications — relational database processing and image processing.

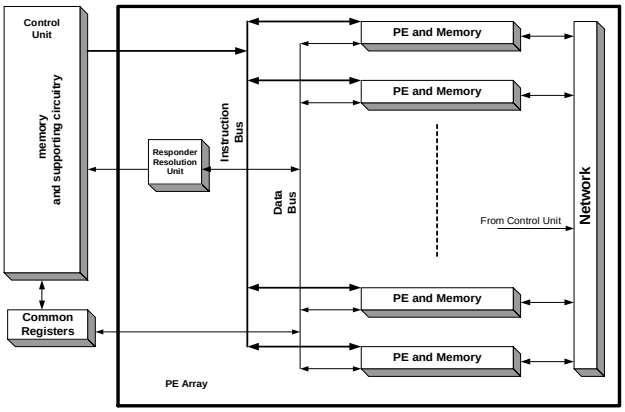

Figure 1. Scalable ASC (ASsociative Computing) processor

2. Associative Computing

The variation on associative computing being explored at Kent State University (KSU) over the past thirty years can most properly be described as associative SIMD computing (a multiple-SIMD, or MSIMD, variation called MASC is currently being explored, but will not be considered here). Associative computing is particularly well suited to processing records of data in a tabular format. As illustrated in Figure 1, each Processing Element (PE) of the SIMD associative computing array can store a record of this tabular data in its memory. If the number of PEs is insufficient compared to the number of data records, multiple virtual PEs can be mapped onto a smaller number of physical PEs. To use a database as an example, each record of the database can be stored in a separate PE as illustrated in Figure 2.

2.1. Associative Search and Responder Processing

One of the central features of associative computing is *associative searching*, which can be implemented in constant time (constant with respect to the number of processors) using an array of SIMD processing elements (PEs). In associative searching, a search key is broadcast and each SIMD PE looks for that key in its local memory at the same time. If the search key is found, the PE is designated a *responder* and a Responder bit is set to '1' in that PE. To allow for nested searches, the Responder bit is also recorded on a Mask Stack.

Once those PEs with a successful search — the *responders* — have been identified, they can be processed in a variety of ways. Masked instructions can be used to process all responders in parallel, as Masked instructions are executed SIMD-fashion only by those PEs that have a '1' on the top of their Mask Stack (in contrast, "normal" instructions are Unmasked, meaning they are executed by all PEs). Masked instructions are particularly useful in building complex associative searches, with successive searches restricted to only those processors matching earlier searches.

				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

Figure 2. Sample student database

In other situations, it may be appropriate to process the responders sequentially, or to process only a single responder. To process all responders sequentially in For-loop fashion, a STEP instruction is used. This instruction sets the top of the Mask Stack of one of the responders to '1', sets the top of the Mask Stack of the other responders to '0', and thus limits further processing to only that one responder. Additionally, the STEP instruction clears the responder's Responder bit, so that further STEP instructions will ignore the processed responder and select only one of the others.

Other Masked instructions can process the responders in While-loop fashion or process only a single responder. In the former case, a FIND instruction is used. Similar to a STEP instruction, the FIND instruction sets the top of the Mask Stack of one of the responders to '1' and the others to '0', limiting processing for now to that one responder. However, the FIND instruction does not clear the responder's Responder bit, so that responder is considered later by subsequent associative processing. In contrast, the RESOLVE_FIRST instruction (occasionally called a PICK_ONE instruction) also sets the top of the Mask Stack of one of the responders to '1' and the others to '0', but it also clears all responders so that no responders are considered by further processing.

As a simple example of associative searching and responder processing, consider the sample student database shown in Figure 2, which contains 12 Records and 3 Attributes (Student Name, ID, and Grade). An associative search for those students who have a Grade over 90 finds two responders — PE1 and PE4. In both of those PEs, the Responder bit is set to '1' and a '1' is pushed onto the top of the Mask Stack (labeled "RSPD" and "Mask" in the figure).

Now suppose that we want to process those two students one-by-one using the STEP instruction— perhaps printing the contents of their records. Since there is no responder before PE1, PE1 is selected to be processed first. In STEP1, its top of Mask Stack is set to

'1' and its Responder bit is cleared. At the same time, all the other PE's top of Mask Stacks are cleared, but their Responder bits are left unchanged for further processing. After processing the record in PE1, the program loops back to start STEP2. Since there is no responder before PE4, PE4 is selected, and similar to STEP1, the top of the Mask Stack and Responder bit are updated. After the second STEP, there are no more responders and the program continues executing the next instructions.

2.2. Associative Reduction for Maximum / Minimum Value

Another central feature of associative computing is its ability to perform certain reduction operations, such as searching for a maximum or minimum value, in constant time (with respect to the number of PEs). Falkoff's algorithm [14] is used to identify those PEs that contain the maximum value in a certain field (the minimum value can be found by complementing the data before processing). Returning to the student database example in Figure 2, Falkoff's algorithm could be used to find the student with the highest grade.

Falkoff's algorithm processes the data field from most significant bit to least significant bit, using a Mask bit to identify PEs that are candidates for holding the maximum value. This bit, initially '1', is ANDed with the most significant bit of the field simultaneously in all PEs to produce a new value for the Mask. If the Mask result is '1', the PE remains a candidate for holding the maximum value and has its Responder bit and top of Mask Stack bit set; otherwise those bits are cleared. If it happens that the result of the AND produces '0' for all candidates, the Mask is left unchanged (i.e., bits of lesser significance may still be used to refine the set of candidates). This processing proceeds from most significant bit to least significant bits, refining the set of candidates for maximum value.

3. Implementing Associative Computing in the ASC Processor

The initial prototype of our ASC (Associative Computing) processor is a byte-serial associative processor, illustrated earlier in Figure 1. This version of our ASC processor has a single Instruction Stream Control Unit (occasionally called the IS Control Unit, the Control Unit, or simply the IS), though an MSIMD-style MASC (Multiple ASC) processor is also under development. This Control Unit works in conjunction with the Processing Element (PE) Array; while the first prototype [12] was limited to only 4 PEs as a proof of concept the current version [15] can be scaled to hundreds of PEs on one FPGA. Additional circuitry [13] in the PE array supports associative search, responder resolution, and maximum / minimum associative reduction.

The Control Unit fetches and decodes instructions, executes scalar instructions, and sends control signals to the PE array to perform associative operations. The Control Unit contains an Instruction Memory and a Data Memory, an 8-bit ALU for scalar arithmetic, and 16 8-bit General Purpose Registers in which the 16th Register is dedicated to holding PE_ID number (0 to N-1). It communicates data (for example, a broadcast associative search key) to and from the PE array through 16 8-bit Common Registers, which are readable by both the Control Unit and the PEs in the PE Array.

Each Processing Element (PE) cell in the PE array consists of a custom-designed 8-bit RISC based PE and a local memory which typically stores one or more data records. Each PE contains an 8-bit ALU and 16 8-bit General-Purpose Registers for data processing, and a 1-bit ALU, 16 1-bit Logical Registers, a Responder bit, and a Mask Stack holding at most 16 1-bit values for associative processing. The PE Array is also supported by Responder Resolution and Maximum/Minimum circuitry. Since the PEs are only 8 bits wide, larger data fields must be processed in byte-serial fashion under the direction of the Control Unit. Finally, the PEs are connected by a 1-D and 2-D interconnection network.

3.1. Implementing Associative Search and Responder Processing

As described in the previous section, a central concept in associative processing is the associative search. To perform associative search in our ASC processor, the Control Unit broadcasts the search key to all PEs through a Common Register and then directs all PEs (SIMD-fashion) to look for that search key in the specified field of their local memory. Those PEs for which the search is successful are designated responders, and they set their Responder bit and the top of their Mask Stack to '1'.

Most parallel instructions in the ASC processor's instruction set come in both Masked and Unmasked versions. Unmasked instructions are executed by all PEs, while Masked instructions are executed only by those PEs with a '1' on the top of the Mask Stack (permitting further processing by the responders from a particular associative search). For complex associative searches, intermediate results can be stored in the PE's Logical Registers.

To implement the STEP, FIND, and RESOLVE_FIRST instructions, a dedicated Responder Resolution Unit in the PE Array works in conjunction with a STEP/FIND/RESOLVE_FIRST (SFR) Unit in each PE. The Responder Resolution Unit generates a signal for each PE to tell it whether or not there is a PE responder with a lower ID number (lower ID numbers have higher priority); it also tells the Control Unit and PE

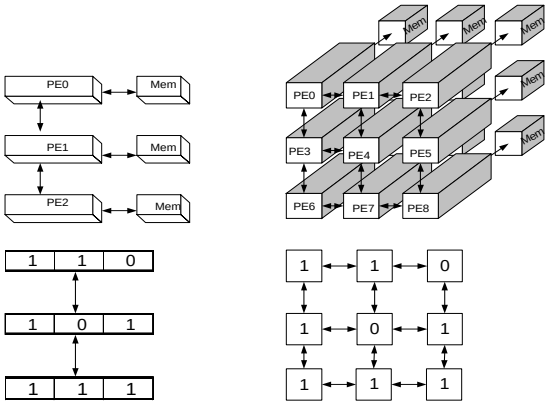

Figure 3. 1-D and 2-D PE interconnection network in the ASC processor

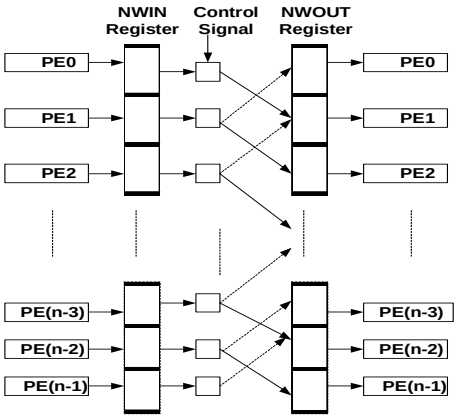

Figure 4. 1-D network operations

Array whether or not any responders exist. The SFR Unit in each PE receives a signal from Responder Resolution Unit and manipulates the Responder bit as well as top of Mask Stack in that PE as appropriate

With this architectural support, the STEP instruction is implemented as follows (the FIND and RESOLVE_FIRST instructions are implemented similarly). First, a STEP instruction is sent to all PEs following an associative search. Since the STEP instruction follows an associative search, it will be ignored by all PEs except the successful responders. In that group of responders, each PE will examine the signal sent to it by the Responder Resolution Unit. If a PE sees that there are no responders with a lower ID number than it, it will replace the top of its Mask Stack with a '1' and clear its Responder bit (recall the responder processing in Figure 2). The other responders will see that a responder with a lower ID number exists, will replace the top of their Mask Stacks with '0's, but will leave their Responder bits untouched. Subsequent Masked instructions will now be executed only by the one PE with the lowest ID number, though later STEP instructions can process the other responders sequentially.

3.2. Implementing Associative Reduction for Maximum / Minimum Value

As described earlier in Section 2.2, another key concept in associative computing is the ability to perform reduction operations such as a maximum value search across all PEs in constant time. Our ASC processor implements Falkoff's algorithm using a dedicated shift register in each PE in conjunction with the PE's Mask Stack and the PE Array's Responder Resolution Unit. Before the search is performed, a '1' is pushed onto the top of every PE's Mask Stack; then a MAX instruction performs the actual search. When the MAX instruction finishes, the PE(s) that have the maximum value in the specified field will have their Responder bit and the top of their Mask Stack set to '1'. If the data is more than one byte wide, the Control Unit must process all bytes in turn.

The MAX instruction searches as follows. First, it copies the specified field to a dedicated shift register in each PE. Then each PE processes the data in the shift register from most the significant bit to the least significant bit, ANDing each bit in turn with the top of the Mask Stack. The result is sent to the Responder Resolution Unit, and if responders exist a signal is sent back to the PEs telling them to update their Responder bits and the top of their Mask Stack with that result; if there were no responders (i.e., all AND results were '0') the Mask Stacks are not updated.

3.3. Implementing 1-D and 2-D PE Interconnection Network

Although many applications can be implemented using SIMD associative computing with no interconnection network, our ASC processor supports both a 1-D and 2-D PE interconnection network for those applications that do require a network, as shown in Figure 3. Using image processing as an example (see the lower half of Figure 3), either an entire row of an image can be stored in each PE's memory and the cells can communicate using the 1-D network, or one pixel of the image can be stored per PE and the cells can communicate using the 2-D network.

The network is implemented as a large 8xN bit wide NWIN register (where N is the number of PEs), an 8xN bit NWOUT register, and routing circuitry as shown in Figure 4. Data enters the network through the NWIN register, which stores data for PE j in bits from 8j to 8j+7, and then that data is routed to the proper place in the NWOUT register.

Data can be moved over either network under program control. Using the 1-D network, data can be moved up or down one PE, and using the 2-D network data can be moved up, down, left, or right one PE. Wrap-around can be turned on or off as desired.

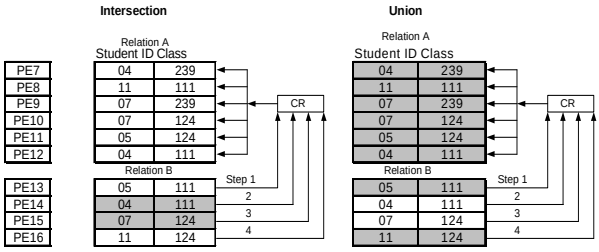

Figure 5. Intersection and Union

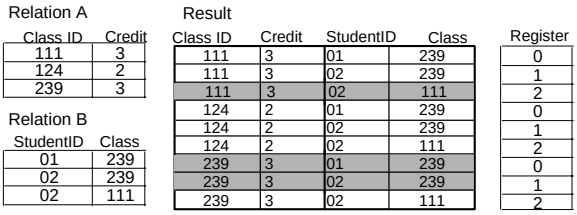

Figure 6. Cartesian Product and Join

4. Results Using the ASC Processor

Associative computing has been demonstrated as effective on a wide range of applications, ranging from database processing [3, 5], image processing [5, 16, 17], string processing [18], computational geometry [19], and even air traffic control [20], and holds great potential in the future for bioinformatics, computational chemistry, etc. In this section, we will demonstrate the use of our ASC processor on two simple problems in database processing and image processing.

4.1. Relational Database Processing

Although an earlier section of this paper has motivated the use of associative computing for searching a one-table database, associative computing is also effective in processing more complex relational databases. A relational database is a set of relations (or tables) where each of the tables is a set of tuples (or records). Relational algebra is a set of operations that has been defined to operate on one or more of these tables. One of the primary advantages of associative computing for implementing relational algebra is that it does not require tabular data to be sorted for efficiency. Using associative computing, single table operations such as Insert, Delete, and Select (Search) can be performed in constant time on an unsorted table. Moreover, more complex database management operations such as Cartesian Product, Union, Intersection, Difference, and Join can also be performed much more efficiently using associative computing compared with execution on a von Neumann machine, as can aggregate functions such as Maximum, Minimum, Sum, and Count. This section will examine how these relational operations are implemented in the ASC processor.

In ASC, each PE stores one tuple. A specific register in each PE contains a unique relation ID so that in a multiple-relation database, the group of PEs that contain the same relation can be identified by the ID in the register. Using this storage scheme, a select (search) in one relation can be done in constant time as described earlier, and an update operation can follow a search whenever necessary. Insert and Delete on the unsorted database is implemented in the ASC processor as follows. To Insert data into a specific relation, we choose any idle PE in the PE array and write the data into this PE's memory, and we also write this table's ID number into the register. To Delete, we only need to store a '0' as the ID to indicate that this PE does not belong to any relation. Since no sorting is necessary, these operations take constant time with respect to the number of PEs.

Most database operations take much more than a single selection. More complicated database algebra operations such as Intersection, Union, and Join require many more steps for their implementation. However, ASC is also effective at speeding up these relational algebra operations, as shown below.

The Intersection and Union of two relations can be realized similarly to the Select operation (see Figure 5, which shows operation results shaded). To compute the Intersection of Relation A and B, first sequentially STEP through Relation B and broadcast the attributes in each tuple in Relation B to all the data in Relation A for comparison in parallel. Multiple field comparisons can be ANDed together to produce a match. These matching PEs contain the result of the intersection. Depending on what we want to do next, we could either push the result onto the Mask Stack for further processing or set a flag in a register to generate a View.

For Union, we need to do the reverse. First we set all Mask bits in Table A and B. During broadcast, once a match is found, we set this tuple's Mask bit (or register) back to '0' in Table B. To facilitate explanation, we are changing the original tables in Figure 5. If Table A and B have N_A and N_B tuples respectively, a new table could be generated for the Union operation with a worst case complexity of $O(N_A + N_B)$ instead of $O(N_A * N_B)$ time on a sequential processor. Further, we could always choose the smaller table to step through.

The Cartesian Product of Relation A and Relation B returns a relation containing all combinations of tuples from two relations. This operation can be performed in the ASC in time complexity $O(N_A + N_B)$ using the following hardware operations. To perform the Cartesian Product for the two relations in Figure 6, we first STEP through Relation A to produce N_B copies for each tuple in relation A in a group of new PEs. To distinguish the N_B

copies of a tuple from relation A, we need to set a different ID number (from 0 to N_B -1) in a register for each copy. The ID number could be generated through subtracting the ID number of each PE from the ID number of the first PE in the N_B copies. For example, in Figure 6, after broadcasting the first line in Relation A (Class ID 111), we broadcast the PE ID of the first line in the resulting Relation to the three PEs on line 1, 2, 3. Each of these three PEs subtracts this number from its own PE ID number to produce the ID number to be stored in the register. Then we broadcast each tuple in Relation B to corresponding positions in the result table where each tuple in Relation B aligns with each tuple from Relation A as shown in the figure. For example, we want to broadcast the first line in Relation B to lines 1, 4, and 7 in the result table in Figure 6. We first mask PEs who have the same number in the register and then broadcast each tuple in Relation B. Broadcasting relation A takes $O(N_A)$ time, and broadcasting relation B takes $O(N_B)$ time, so the whole operation takes $O(N_A+ N_B)$ time. The result of this Cartesian Product is shown in Figure 6.

The Join operation is used to combine related tuples from two relations into a single tuple. If a pair of tuples (one from each relation) satisfies a given condition, a new tuple is created containing the combined attributes from both original tuples. All possible pairs of tuples are considered. While the condition can be any Boolean operation among attributes of the tuples, we will only illustrate EquiJoin here. For EquiJoin, the condition is that one or more attributes of the tuples are identical. This operation then concatenates tuples from the two tables whenever they meet the condition.

For example, suppose we perform EquiJoin for the two relations in Figure 6. ClassID and Class have some identical ID numbers for classes. To do EquiJoin, first Cartesian Product is performed on the two relations to generate a new set of combined tuples as shown in Figure 6. Once Cartesian Product is done, it takes constant time to compare values in Attributes Class ID and Class to find the Join result (results shown in grey). Because the Cartesian Product takes $O(N_A + N_B)$ time, the EquiJoin operation also is in time complexity $O(N_A + N_B)$.

Many aggregation functions can be performed efficiently in ASC. ASC provides built-in hardware to support constant time maximum and minimum searching through the data in a relation as described in earlier. To Sum data from tuples throughout multiple PEs, we could issue a Search to find all the tuples that are to be summed, and then STEP through all the PEs adding to the sum in each step. Though the summation is sequential, it is not necessary to process all PEs in a relation. The Count function is implemented by Summing all the responder bits across all PEs in a relation. We do not yet use the existing network in these operations, but expect to do so in the future to improve the efficiency of these operations.

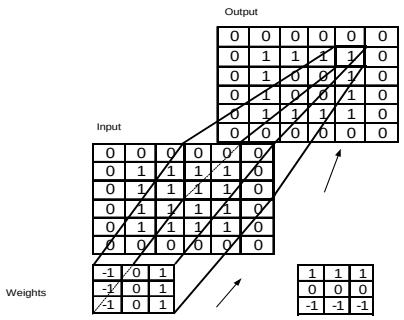

Figure 7. Convolution to detect a vertical edge

4.2. Image Processing (Edge Detection Using Convolution)

Often organized as a large number of simple processors arranged in a 2-D mesh, SIMD processors have long been applied to image processing [21]. Since our ASC processor supports both SIMD operations and 1-D and 2-D interconnection networks, augmented with associative processing, it is even more suited to image processing. This section demonstrates the use of our ASC processor for *edge detection* — an image pre-processing operation for identifying edges in an image, where edges are defined as areas where the brightness changes abruptly. Edge detection is typically followed by other image processing operations such as segmentation and object recognition.

We have implemented edge detection on our ASC processor using convolution on an array of 36 PEs. More specifically, we detect vertical or horizontal edges in the image by convolving each interior pixel in the image with a 3x3 weight matrix called the Prewitt operator. Convolution is defined as the sum of the products of each cell in the weight matrix with its corresponding cell in the input image, so a single convolution operation reads 9 input image cells, performs 9 multiplications, adds those 9 products, and produces a result. The absolute value of that result is then thresholded to produce an image cell in the output image.

Figure 7 shows a cell in a 1-bit (i.e., black and white) input image convolved with the Prewitt vertical edge mask to produce a cell in the output image. The Prewitt vertical mask, shown at the lower left of the figure, finds the vertical edges in the input image, while the horizontal mask, shown at the lower right of the figure, finds the horizontal edges. To find both edges as shown in the figure, the absolute values of the two results are added together and thresholded (values above '1' replaced with '1'), producing the square boundary of the solid input square. Compared to the $O(N^2)$ algorithm on a sequential machine (assuming a NxN image), our ASC machine can

perform this operation in O(N) time using N PEs connected with a 1-D network or in constant time using N^2 PEs connected with 2-D network.

In 1-D mode, each row of the image is stored in the memory of a different PE, with each memory location holding one pixel. As shown at the lower left of Figure 3, this means that each column of the image is stored at corresponding locations in the memory of different PEs. Pseudocode for finding the vertical edges in an NxN image using 1-D mode is as follows (finding both edges is a simple extension):

```
1. Loop: For N pixels in a row
2.         For all PEs in parallel
3.         Loop: For each row (three cells) in the weight
           matrix
4.                 Multiply the three weight cells with
                   corresponding three cells in input image
5.                 Add the three products and move the sum
                   to the output cell
6.         End Loop
7.         Add the three sums (one per row in the weight
           matrix) to produce the output image cell
8. End Loop
```

The innermost loop iterates 3 times and the outermost loop iterates N times, so overall the algorithm runs in O(N) time using a 1-D network.

The detailed execution of this code on our ASC processor is under the control of the compiler, but in our tests worked as follows. Basically, one row of the image (three cells) is processed at a time using the STEP operation, reading weight matrix cells and broadcasting them to all PEs. PEs read input image cells three at a time, and multiply them with the corresponding weight matrix cells. The three products in each PE are then added, and a MOVE instruction moves the sum down if it is the first line of the weight matrix, up if it is the third line of the weight matrix, or will not move the sum at all if it is the center row of the weight matrix. This sum is then stored temporarily in the destination PE until it can be added to the other sums. After the final sum is computed, it is thresholded to either '1' or '0' (in the simple example shown, the threshold is 0, so if the final sum is other than '0', it is set to '1', otherwise '0'), producing a new pixel in the output image. Other rows in the input image cells are processed similarly.

In 2-D mode, each PE holds a pixel in the image, as shown at the lower right of Figure 3. This mode requires NxN PEs to process the image as follows:

```
1. Loop: For each cell in the weight matrix
2.         For all PEs in parallel
3.                 Multiply weight cell with all input cells
                   and move to output cells
4.         Add the nine products (one per cell in the weight
           matrix) to produce the output image
5. End Loop
```

The outermost loop iterates 9 times, so overall the algorithm runs in constant time (with respect to the number of PEs) using a 2-D network.

The detailed execution of this code is similar to that using the 1-D network, but requires more network movement. Weight matrix cells must be read one by one, and each product must be moved differently through the network in order to reach the center PE. For example, after broadcasting the weight matrix to all PEs, the weight matrix cell at the upper left corner will be multiplied with the corresponding image cells and the product will first move down to the PEs below those image cells and then move right to the center PEs where all nine products will be added together.

Although these algorithms could be executed in pure SIMD fashion, the use of associative processing features such as the STEP operation does provide some additional power. However, advanced image processing will take more advantage of associative processing. For example, Pixel Planes [22] is a dedicated system for graphics and imaging. This fast processor uses one processor at one pixel, similar to our 2-D network arrangement. During scan conversion, first, a coefficient is broadcast to all pixels in parallel, then an Enable register is used to identify whether a pixel is inside or outside a polygon by comparing its own data with this coefficient. These methods can be performed by our ASC processor by searching and using the top of the Mask Stack to enable certain processors (pixels). Many of their other techniques can also be adapted to associative processing.

5. Conclusions and Future Work

This paper has described the implementation and use of a dedicated associative SIMD co-processor, which we feel is ideally suited for many applications such as database processing, image processing, genome matching, or molecular similarity. The implementation of our ASC processor has been briefly described, along with its implementation of associative SIMD processing. Although our current prototype is limited to 36 8-bit PEs on a single small FPGA, larger FPGAs or multiple-FPGA boards should easily support hundreds or thousands of PEs. We have also demonstrated the use of our ASC processor on relational database processing and on the image processing operation of edge detection.

Our future work will continue to target the applications mentioned above, building on associative algorithms developed at Kent State as well as local expertise in those application areas. In addition to further work on relational database processing and image processing, we are beginning to implement some string matching algorithms [18] with applications in genome matching, and some methods of representing complex data structures such as trees more efficiently [8]. With the

current second-generation prototype ASC processor as a base, we intend to streamline the architecture, as well as adding some functionality not currently present, such as floating-point operations and I/O operations. In addition, we intend to work with other researchers at Kent State to develop a more dynamic multiple-instruction-stream (MASC) version of the processor, along with the necessary runtime environment.

References

[1] Jerry L. Potter, *Associative Computing*, Plenum Publishing, New York,1992.

[2] Anargyros Krikelis and Charles C. Weems (Editors), *Associative Processing and Processors*, IEEE CS Press, 1997.

[3] Isaac D. Scherson and Sener Ilgen, "A Reconfigurable Fully Parallel Associative Processor", *Journal of Parallel and Distributed Computing*, 6, pp. 69-89 (1989).

[4] J. Storrs Hall, Donald E. Smith, and Saul Y. Levy, "Database Mining and Matching in the Rutgers CAM", in *Associative Processing and Processors*, Anargyros Krikelis and Charles C. Weems (Editors), IEEE CS Press, 1997, pp. 218-232.

[5] Karl E. Grosspietsch and Ralf Reetz, "The Associative Processor System CAPRA – Architecture and Applications", *IEEE MICRO* 12 (6): 58-67, December 1992.

[6] Behrooz Parhami, "Search and Data Selection Algorithms for Associative Processors", in *Associative Processing and Processors*, Anargyros Krikelis and Charles C. Weems (Editors), IEEE CS Press, 1997, pp 10-25.

[7] Martin C. Herbordt and Charles C. Weems, "Associative, Multiassociative, and Hybrid Processing", in *Associative Processing and Processors*, Anargyros Krikelis and Charles C. Weems (Editors), IEEE CS Press, 1997, pp. 26-49.

[8] Jerry Potter, Johnnie Baker, Stephen Scott, Arvind Bansal, Chokchai Leangsuksun, and Chandra Asthagiri, "ASC: An Associative Computing Paradigm," *IEEE Computer*, November 1994, pp.19-26.

[9] K. E. Batcher, "STARAN Parallel Processor System Hardware", 1974 National Computer Conference, *AFIPS Conference Proceedings*, vol. 43, pp. 405-410, 1974.

[10] R. Reed, "The ASPRO Parallel Inference Engine (P.I.E.) – A Real Time Production Rule System," *Proc. Conf. Aerospace*, pp.85-89.

[11] Will Meilander, Johnnie Baker, and Mingxian Jin, "Importance of SIMD Reconsidered", *Proc. of the 17th International Parallel and Distributed Processing Symposium (Workshop in Massively Parallel Processing)*, abstract on p. 266, full text on accompanying CDROM. IEEE, Nice, France, April 2003.

[12] Robert Walker, Jerry Potter, Yanping Wang, and Meiduo Wu, "Implementing Associative Processing: Rethinking Earlier Architectural Decisions", *Proc. of the 15th International Parallel and Distributed Computing Symposium (Workshop on Massively Parallel Processing)*, abstract on p. 195, full text on accompanying CDROM. IEEE, San Francisco, California, April 2001.

[13] Meiduo Wu, Robert A. Walker, and Jerry Potter, "Implementing Associative Search and Responder Resolution", *Proc. of the 16th International Parallel and Distributed Computing Symposium (Workshop on Massively Parallel* Processing), abstract on p. 246, full text on accompanying CDROM. IEEE, Ft. Lauderdale, Florida, April 2002.

[14] A. Falkoff, "Algorithms for Parallel Search Memories", *Journal of Associative Computin*g. March 9 1962, pp. 488-511.

[15] Hong Wang and Robert A. Walker, "Implementing a Scalable ASC Processor", *Proc. of the 17th International Parallel and Distributed Processing Symposium (Workshop in Massively Parallel Processing)*, abstract on p. 267, full text on accompanying CDROM. IEEE, Nice, France, April 2003.

[16] Frederick P. Herrmann and Charles G. Sodini, "A Dynamic Associative Processor For Machine Vision Applications", *IEEE MICRO* 12 (3): 31-41, June 1992 .

[17] A.W.G. Duller, R.H. Storer, A.R. Thomson, et al. "An Associative Processor Array for Image-Processing", *Image Vision COMPUT* 7 (2): 151-158, May 1989

[18] Mary C. Esenwein and Johnnie W. Baker, "VLDC String Matching for Associative Computing and Multiple Broadcast Mesh", *Proc. of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 69-74. 1997.

[19] Maher Atwah, Johnnie Baker, and Selim Akl, "An Associative Implementation of Classical Convex Hull Algorithm", *Proc. of the Eighth IASTED International Conference on Parallel and Distributed Computing Systems*, pp. 435-438. October 1996.

[20] Will Meilander, Johnnie Baker, and Mingxian Jin, "Predictable Real-Time Scheduling for Air Traffic Control", *Proc. of the 15th International Conference of Systems Engineering*, pages 533-539, August 2002.

[21] Milan Sonka, Vaclav Hlavac, and Roger Boyle, *Image Processing, Analysis, and Machine Vision (Second Edition)*, Brooks / Cole Publishing, 1999.

[22] H. Fuchs, J. Goldfeather, J.P. Hultquist, S. Spach, J.D. Austin, F.P. Brooks, J.G. Eyles, and J. Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes", *Computer Graphics* (SIGGRAPH '85 Conference Proceedings), Vol. 19, No. 3, July 1985, pp. 111-120.