Fast Binary and Gray-scale Mathematical Morphology on VLIW Mediaprocessors

George York, Ravi Managuli and Yongmin Kim

Image Computing Systems Laboratory
Department of Electrical Engineering, Box 352500
University of Washington, Seattle, WA 98195-2500

ABSTRACT

Mathematical morphology has proven to be useful for solving a variety of image processing problems (e.g., shape and size extraction, noise removal, skeletonization) and plays a key role in certain time-critical machine vision applications (e.g., parts inspection, robotics, military systems). The large computation requirement for morphology poses a challenge for microprocessors to support in real time, and often hardwired solutions such as ASICs and EPLDs have been necessary. This paper presents a method to implement binary and gray-scale morphology algorithms efficiently on programmable VLIW mediaprocessors. Efficiency is gained by (1) mapping the algorithms to the mediaprocessor's parallel processing units, (2) avoiding redundant computations by converting the structuring element into a unique lookup table, and (3) minimizing the I/O overhead by using an on-chip programmable DMA controller. Using our approach, "C" implementation of gray-scale dilation takes 7.0 ms and binary dilation takes 1.2 ms on a 200 MHz MAP1000 mediaprocessor for a 512x512 image with a 3x3 structuring element. This is 4 to 8 times faster than assembly code implementations on the TMS320C80 mediaprocessor, and more than 35 times faster than that reported for general-purpose microprocessors. With comparable performance to ASIC implementations and the flexibility of a programmable processor, this real-time image computing with mediaprocessors will be more widely used in machine vision and other imaging applications in the future.

Keywords: real-time imaging, mathematical morphology, binary morphology, gray-scale morphology, mediaprocessor, VLIW, TMS320C80, MAP1000, bit-mapped binary images.

1. INTRODUCTION

Mathematical morphology has become a fundamental image processing methodology since its introduction by Serra[1]. Binary morphology is derived from Minkowski set algebra and involves modifying and analyzing the shapes of the objects in an image by a reference shape known as the structuring element. The basic operations are dilation and erosion, which respectively involve expanding or contracting the border pixels of each object by the structuring element. Binary morphology has been applied to many image processing tasks, such as shape and size extraction, noise removal, skeletonization, and thickening images. Morphology has since been extended to gray-scale images, where it has found use in smoothing images, enhancing the contrast of bright and/or dark objects, edge detection, and texture segmentation[2]. Today morphology plays a key role in certain time-critical machine vision applications, such as parts inspection, robotics, and military systems. More information about morphology fundamentals can be found elsewhere[3].

Since morphology operations are used in the early stages of many image processing applications, it is essential these operations be completed as quickly as possible. In this paper, we focus on a fast implementation of dilation and erosion on mediaprocessors for both binary and gray-scale images. While speed is a priority, we also have a goal of supporting any arbitrary structuring element size and shape, as real applications are usually not constrained to certain classes of structuring elements. No effort has been made to decompose the structuring element into a series of smaller structuring elements, which can improve performance for certain convex structuring elements[4]. These techniques can easily be incorporated to further improve the performance. For binary morphology, we assume the images are bit-mapped binary images, which is a more efficient format than using binary 8-bit images, as pointed out by Boomgaard and Balen[4]. In the appendix, we present a unique method to convert gray-scale images to bit-mapped binary images using mediaprocessors.

Binary and gray-scale dilation require the following respective computations:

$$(X \oplus S)_{(m,n)} = OR_{(i,j)}[X_{(m-i,n-j)} AND\, S_{(i,j)}] \quad (1)$$

$$(X \oplus S)_{(m,n)} = MAX_{(i,j)}[X_{(m-i,n-j)} + S_{(i,j)}] \quad (2)$$

where X is the input image, S is the structuring element, and m, n, i, j are the spatial coordinates. Given the duality of dilation and erosion, binary and gray-scale erosion are performed by respectively replacing *OR, AND, MAX,* and "+" in the above equations with *AND, OR, MIN,* and "-", and by transposing S. The binary dilation computation can be simplified further by only selecting the input pixels $X_{(m-i,n-j)}$ that correspond to the active structuring element pixels (e.g., when $S_{(i,j)} = 1$), simplifying equation (1) to:

$$(X \oplus S)_{(m,n)} = OR_{((i,j)\in S_{(i,j)}=1)}[X_{(m-i,n-j)}] \quad (3)$$

The computation requirement for the dilation and erosion morphology algorithms is *O(MxNxIxJ)*, which poses a challenge to implement on microprocessors in real time, especially for large structuring elements. Therefore to achieve real-time performance, hardwired approaches ranging from ASICs (application specific integrated circuits)[5] to specialized image processing machines (e.g., CAACP[6] and CLIP[7] array processors) have been developed. Both ASICs and specialized systems with large 2-D arrays tend to be an expensive solution and are inflexible and difficult to modify for changing system requirements[9]. Hardwired approaches tend to impose restrictions, such as limiting the structuring element shape or size to 3x3 windows[5]. In addition, morphology is usually used in combination with several other image processing functions (e.g., connected components, Boolean image operations, histograms, etc.)[2], which a programmable system can easily be modified to handle. On the other hand, a hardwired system would need several specialized components for each algorithm, creating a fast but inflexible system. Thus, we are pursuing a programmable solution that can adapt to changing system requirements, yet still have the computation power to meet the real-time requirements. These requirements can be met with today's mediaprocessors.

Mediaprocessors are a new class of DSPs that have recently evolved to handle the high computation requirements of multimedia applications and offer fine-grained parallelism, typically with a VLIW (Very Long Instruction Word) architecture. A VLIW mediaprocessor has several arithmetic units and load/store units that operate concurrently. The arithmetic units also support partitioned operations, e.g., allowing a 64-bit computing engine to be partitioned into eight 8-bit units and then performing operations on eight 8-bit pixels in parallel. Figure 1 shows an example of the *partitioned_add* instruction. Several of the mediaprocessors also have an on-chip programmable DMA controller that can move the data on and off chip in parallel with the core processor's computations. In compute-bound applications like morphology, the DMA controller effectively eliminates the I/O overhead experienced by cache-based microprocessors. To fully take advantage of the features of mediaprocessors requires rethinking the implementation of algorithms and how they map to the underlying architecture[8,10,11], which we present for morphology in this paper.

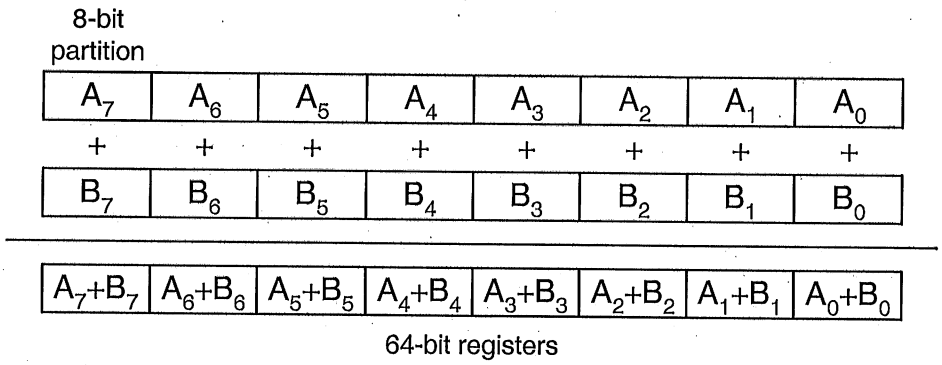

Figure 1. Example partitioned operation: *partitioned_add*.

The first generation of mediaprocessors, such as the Texas Instruments TMS320C80, offered much computing power (e.g., 4 parallel VLIW DSPs plus a RISC processor). However, to achieve an acceptable level of performance, we found assembly language programming was required as the C compiler technology had difficulty exploiting the parallelism of the C80[10]. For ease of maintenance, portability, and reduced development time, it would be preferred to use a high-level language such as C. The next generation of mediaprocessors are being developed with the C compiler in mind, for example, the MAP1000[9].

The MAP1000 is a new VLIW processor being developed for multimedia applications. The MAP1000 issues instructions to two clusters each clock cycle. Each cluster has two execution units, an IALU and an IFGALU, which share a large register bank (128 32-bit general registers; 16 predicate registers; two 128-bit registers). Each IALU can perform either 32-bit fixed-point arithmetic operations or 64-bit loads and stores. Each IFGALU can perform either 64-bit partitioned arithmetic operations (i.e., eight 8-bit adds), 64-bit fixed-point partitioned multiplies (i.e., multiple 16-bit multiplies), floating-point

operations, square roots, and other powerful operations like *inner-product*. The *inner-product* instruction is capable of multiplying eight 16-bit coefficients by eight 16-bit input pixels and sum the multiplication results, all in one instruction. Each cluster can issue these partitioned instructions each cycle. In addition, the MAP1000 has a 16-kbyte instruction cache and a 16-kbyte data cache, a programmable DMA controller called the data streamer, two PCI ports, and a video graphics coprocessor (VGC) on-chip. The VGC has its own 9 kbytes of on-chip memory and is designed to do bit-serial processing (such as that required in MPEG variable run-length encoding). A possible use for the VGC would be to automatically convert a gray-scale image into a binary bit-mapped image as the data is streamed from external memory to the core processor during binary morphology operations.

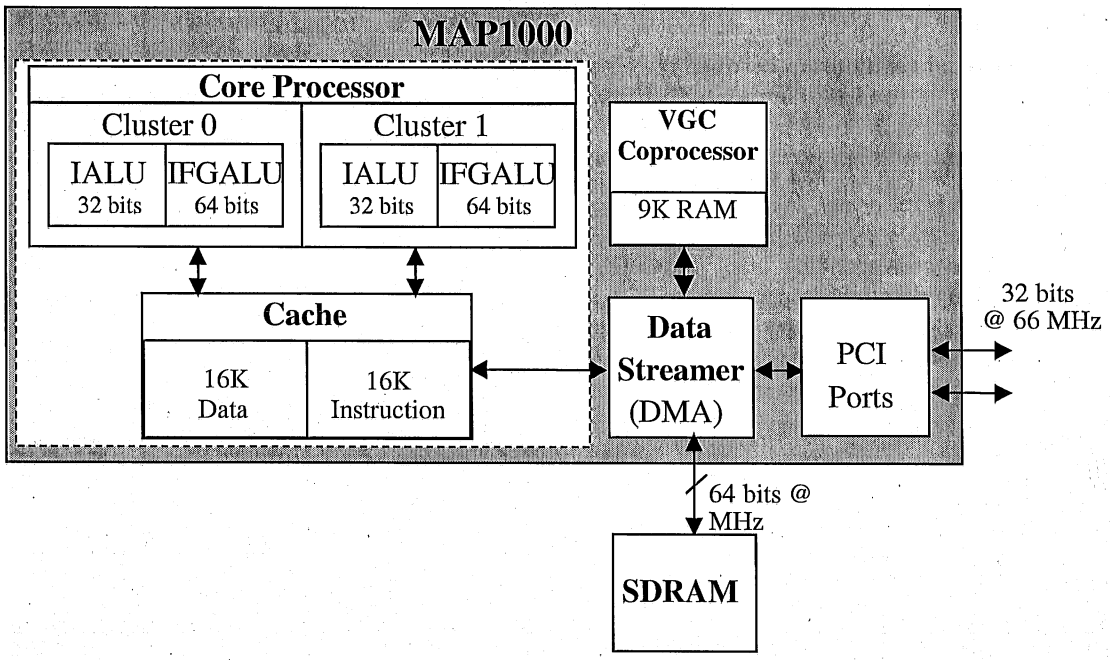

Figure 2. MAP1000 mediaprocessor.

In this paper, we present an approach to efficiently map morphology algorithms to mediaprocessors, along with the performance results on the TMS320C80 and MAP1000. Efficiency is gained by (1) mapping the algorithms to the mediaprocessor's multiple processing units, (2) avoiding redundant computations by converting the structuring element image into a unique lookup table (LUT) and only including the active structuring elements, (3) utilizing the full capacity of the multiple execution units using software pipelining, and (4) carefully managing the data flow using an on-chip programmable DMA controller, minimizing the I/O overhead. Using these techniques can reduce the computation time from $O(MxNxIxJ)$ to $O(MxNxK/W)$, where K is the number of active structuring element pixels and W is the number of pixels that can be processed in parallel across the multiple processing units. By efficiently mapping these algorithms to a mediaprocessor's architecture, our results indicate an improvement by a factor of 35 can be achieved compared to other software-based implementations reported[4].

2. IMPLEMENTATION ON MEDIAPROCESSORS

2.1. Mapping Algorithms to the Parallel Processing Units

Mediaprocessors can efficiently execute all the operations of equations (1) and (2) (e.g., *OR, AND, MAX, MIN,* "+", "-") in a partitioned manner similar to Figure 1. Further parallelism is gained by using the multiple processing units, executing multiple sets of partitioned operations and loads/stores each cycle. For the MAP1000 with 2 IFGALUs, each with 64-bit registers partitioned into eight 8-bit pixel values, gray-scale dilation can be performed on 16 gray-scale pixels in parallel. Likewise, binary dilation can be performed on 128 binary pixels in parallel, following a technique proposed for 32-bit processors on binary bit-mapped images[4]. Similarly for the TMS32C80, its four parallel processing units, each 32-bit wide, combine for a total of 16 gray-scale pixels processed in parallel or 128 binary pixels in parallel.

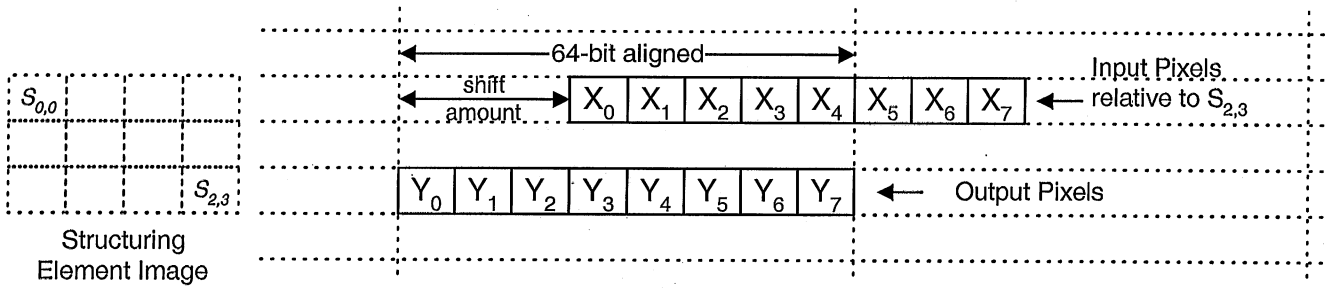

Figure 3. Example spatial relationship between the output pixels (Y_k), a single active structuring element pixel ($S_{2,3}$), and the corresponding input pixels (X_k). (Gray-scale case)

Directly implementing equations (2) and (3) implies the structuring element image $S_{(i,j)}$ must be parsed for each pixel to determine if it is an active pixel or not. If it is active, then the respective input image pixels $X_{(m-i,n-j)}$ must be fetched, which requires calculating the *relative_address* based on *(m-i,n-j)*. Figure 3 shows the spatial relationship between a group of output pixels (Y_k) and the corresponding input pixels (X_k) for one active structuring element pixel ($S_{2,3}$) for gray-scale dilation. Since we are using partitioned operations and processing several sequential pixels at a time, we must fetch multiple input pixels (8 for gray-scale; 64 for binary that are not necessarily aligned to a 64-bit boundary as illustrated by Figure 3. Therefore, two neighboring 64-bit words must be fetched and the proper bits aligned (or extracted) based on a *shift_amount,* which must be calculated. The calculations are:

$$relative\_address = \frac{(i - i\_origin) \cdot image\_width + (j - j\_origin)}{W} \tag{4}$$

$$shift\_amount = (j - j\_origin) \bmod W \tag{5}$$

where *i_origin* and *j_origin* are the spatial coordinates of the origin of the structuring element (represented by $S_{0,0}$ in Figure 3), *image_width* is the width of the image (in units of pixels), and W is the number of pixels loaded per word (W=8 for gray-scale and W=32 for binary).

Special mediaprocessor instructions are used to align the input data based on the *shift_amount*. For the MAP1000, we use the *align* instruction for gray-scale data as illustrated in Figure 4 and we use the *bit_extract* instruction for binary bit-mapped data shown in Figure 5.

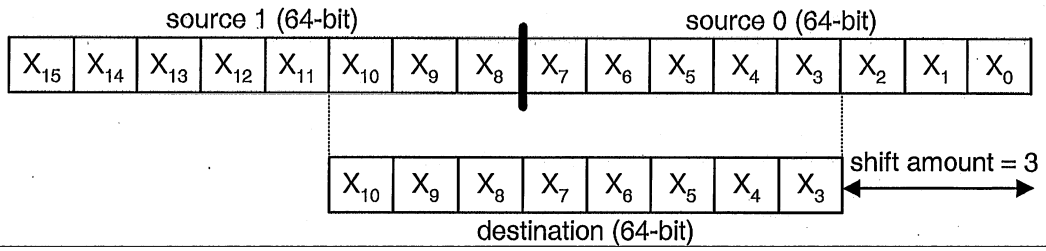

Figure 4. The *align* instruction.

An additional step for gray-scale dilation involves replicating each active $S_{(i,j)}$ pixel across an eight 8-bit partitioned register, which is needed to sum each $S_{(i,j)}$ with each respective set of 8 input pixels $X_{(m-i,n-j)}$ in equation (2).

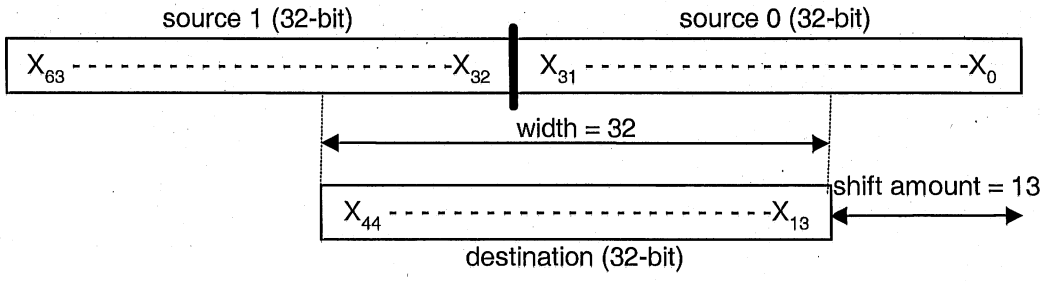

Figure 5. The *bit_extract* instruction.

2.2. Convert Structuring Element to a Lookup Table

When calculating each set of output pixels $(X \oplus S)_{(m,n)}$, substantial overhead is incurred if the structuring element $S_{(i,j)}$ must be parsed (i.e., searched for active pixels and the *relative_address* and *shift_amount* computed for each active structuring element pixel) in real time for each group of output pixels. Therefore, we avoid these redundant calculations by converting the structuring element image into an on-chip lookup table (LUT). The LUT contains the *relative_address* and *shift_amount* (plus the replicated $S_{(i,j)}$ values for the gray-scale case) for each active structuring element pixel. The LUT is then used to automatically fetch the needed input data words from on-chip memory. The following pseudo-code illustrates how the LUT is used and the dilation implemented to compute one set of 8 gray-scale output pixels. A block diagram version of the gray_scale dilation is also shown in Figure 6.

```
accumulated_result = 0;                                   /* Initialize to 0 since we are looking for maximum */
for ( k = 0; k < number_of_active_elements; k++ ) {       /* For all active structuring elements */
    relative_addr     = LUT[ADDRESS][k];                  /* Read structuring element values for the active pixel */
    shift_amount      = LUT[SHIFT][k];
    element_value     = LUT[VALUE][k];
    input_data0       = *(input_image_ptr + relative_addr);        /* Load the respective input image data */
    input_data1       = *(input_image_ptr + relative_addr + 1);
    input_X           = align ( input_data1, input_data0, shift_amount );   /* Extract the input data pixels */
    sum_X_S           = partitioned_add( input_X, element_value);           /* Add X(m-i,n-j) and S(i,j) */
    accumulate_result = partitioned_max( sum_X_S, accumulated_result );     /* Find the maximum */
}
*output_image_ptr++ = accumulate_result;                  /* Store the results */
```

To modify the above pseudo-code for binary dilation, the *element_value* and the *partitioned_add* are not needed. The *partitioned_max* should be replaced by *band* (bitwise and) and the *align* instruction should be replaced by two *bit_extract* instructions (to extract a total of 64 bits).

Besides using this lookup table approach to avoid redundant computations, we also avoid any branching (i.e., *if, then, else* statements or *loops*) in the inner loop, which some approaches to morphology require[4]. Branches in the inner loop can severely degrade the performance of a VLIW processor, as the multiple paths of the branch make software pipelining difficult (discussed in section 2.3), resulting in many idle cycles on the parallel processing units. An experiment with a version of binary dilation requiring a branch in the inner loop was 50% slower on the MAP1000.

In the above pseudo-code, the partitioned operations (e.g., *partitioned_add, partitioned_max, align*) may appear to be function calls, but are actually *C intrinsics*. These intrinsics direct the compiler to replace the function call with the corresponding assembly instruction. Current compilers (e.g., MAP1000, Pentium II/MMX, TMS320C60) have difficulty recognizing the inherent parallelism in image processing code and automatically using the proper partitioned operations. Therefore, intrinsics are currently used, offering the programming advantage of high-level languages, while achieving the performance of assembly language[9].

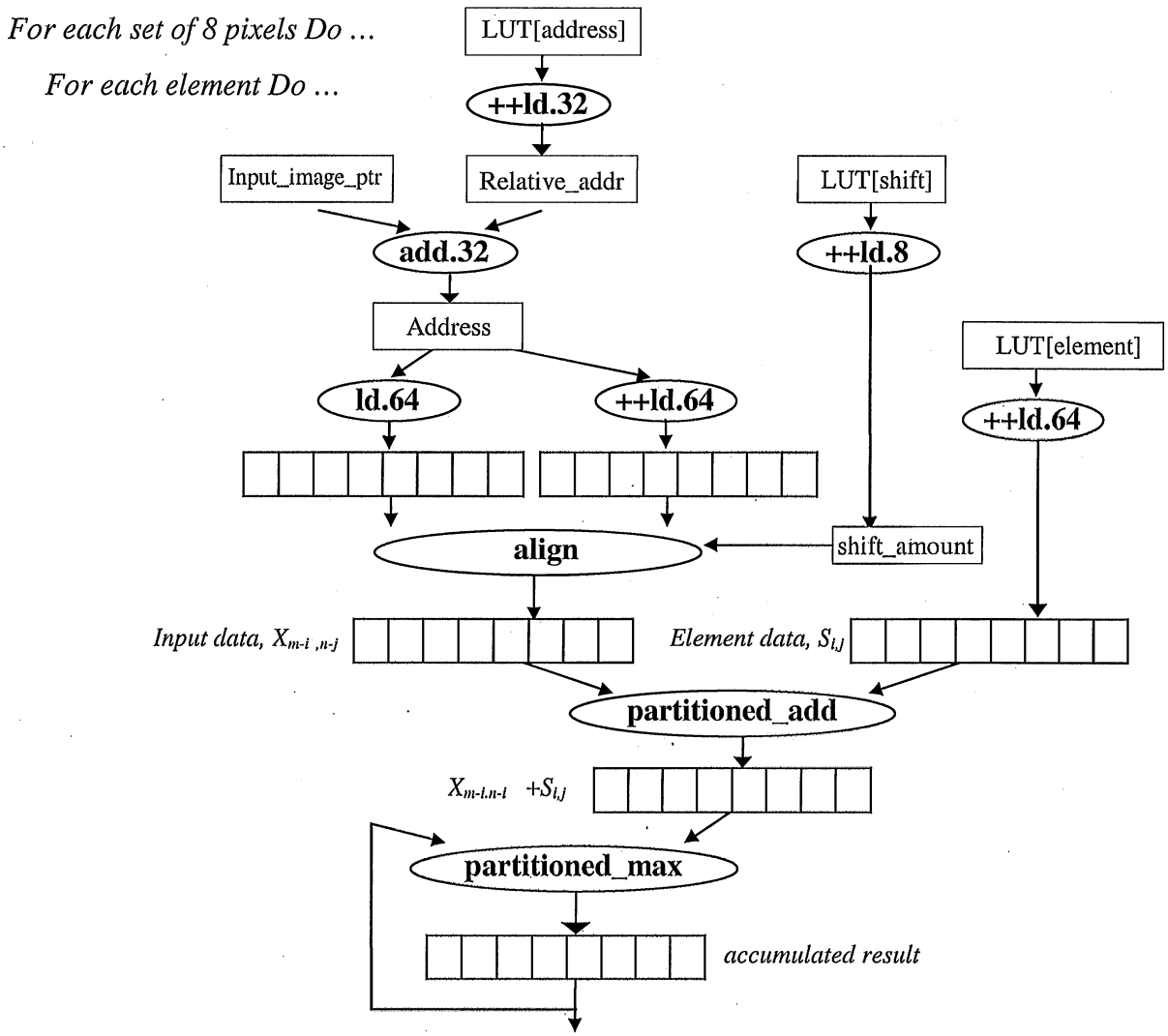

Figure 6. Gray-scale dilation computation implemented on VLIW mediaprocessors.

The above pseudo-code can be optimized further by processing four neighboring sets of output pixels together in the same loop, sharing the *relative_address* and *shift_amount* (plus the replicated $S_{(i,j)}$ values for the gray-scale case) values for each structuring element pixel. This minimizes the cost of loading two 64-bit words to extract each 64-bit input word as shown in Figure 7. Extracting the 4 input words individually would require eight loads, while taking advantage of sharing the words between each neighbor requires only five loads, reducing this overhead by about 40%. Processing four sets of output pixels together in the same loop also aids in software pipelining.

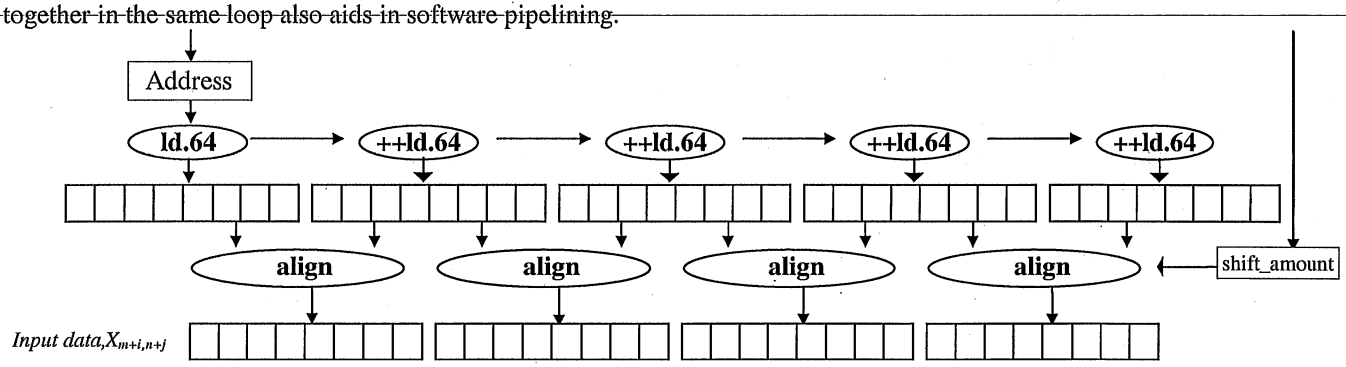

Figure 7. Technique to reduce overhead of multiple loads.

2.3. Loop Unrolling and Software Pipelining

For VLIW mediaprocessors to achieve their peak performance, all the processing units (i.e., IALU and IFGALU) need to be kept busy starting a new instruction each cycle. However, different classes of instructions have different latencies (cycles to complete), sometimes making it difficult to achieve peak performance. For example, loads have a 5-cycle latency and partitioned operations have a 3-cycle latency for the MAP1000. Figure 8(a) illustrates a simplified version of gray-scale morphology before pipelining (for simplicity, the LUT *loads*, *align*, and loop branching are ignored and only one cluster is shown). In this loop, *LD_X* and *LD_S* represent loading the input pixels and structuring element pixels. After a 5-cycle latency, the *partitioned_add* (ADD) is issued, and then after 3 cycles, the *partitioned_max* (MAX) is issued. Finally, after iterating for each active structuring element pixel, the results are stored (ST) after another latency of 3. This results in only 4 instruction slots used out of 20 possible slots for the IALU and IFGALU in the loop.

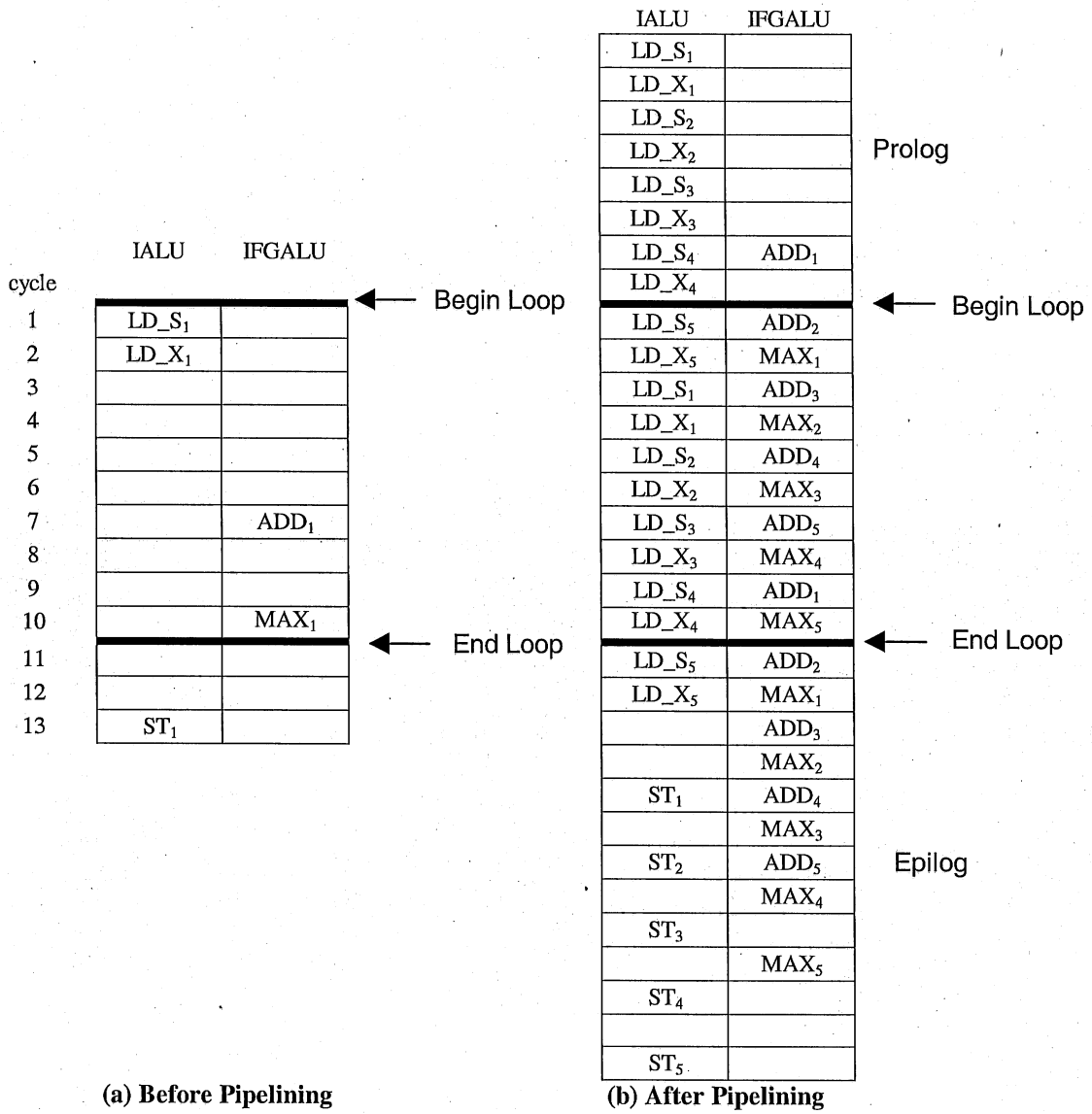

Figure 8. Example of pipelining.

For better performance, loop unrolling and software pipelining can be used to more efficiently utilize the processing units by working on multiple sets of data each loop and overlapping their execution[12]. Unrolling a loop means executing multiple sets of data inside the loop, illustrated by Figure 8(b), in which we dilate five sets of data (indexed 1 through 5). We then software pipeline the five sets of operations, overlapping their execution wherever possible to keep the IALU and IFGALU processing units starting a new instruction each cycle. Using these techniques results in all possible slots used in the inner loop in Figure 8(b), processing five times more data in approximately the same number of cycles. This example is an ideal

example in that there are equal number of IALU and IFGALU instructions, allowing all the slots to be filled. For the actual dilate algorithm, the number of IALU instructions slightly exceeds the IFGALU instructions, resulting in a few unused IFGALU instruction slots in the inner loop.

Pipelining code in assembly language is a tedious process and creates code that is difficult to modify and maintain. The MAP1000 C compiler has the capability to automatically unroll and software pipeline the code for the programmer. We implemented binary dilation in both assembly and C language to compare the C compiler's ability to perform software pipelining. As the results in Table 1 show, the compiler is only 50% slower than the assembly code implementation.

2.4. Manage Dataflow with Programmable DMA Controller

Since the on-chip memory is limited (16 kbytes for the MAP1000) and cannot hold entire images, we process 2D image blocks at a time. To keep the processor from waiting on data I/O from external memory to on-chip memory, an on-chip programmable DMA controller is used to move the data on and off chip concurrent with the processor's computations. This technique is commonly known as *double buffering*, illustrated in Figure 9. To double buffer, we allocate four buffers in on-chip memory, two for input blocks (*ping_in_buffer, pong_in_buffer)* and two for output blocks (*ping_out_buffer, pong_out_buffer).* While the core processor dilates a current output image block (e.g., block #2) from the *pong_in_buffer* and stores the result in *pong_out_buffer*, the DMA controller stores the previously calculated output block (e.g., block #1) in *ping_out_buffer* to external memory and brings the next input block (e.g., block #3) from external memory into *ping_in_buffer.* Then the core processor and DMA controller switch buffers, with the core processor working on the *ping* buffers and the DMA controller working on the *pong* buffers.

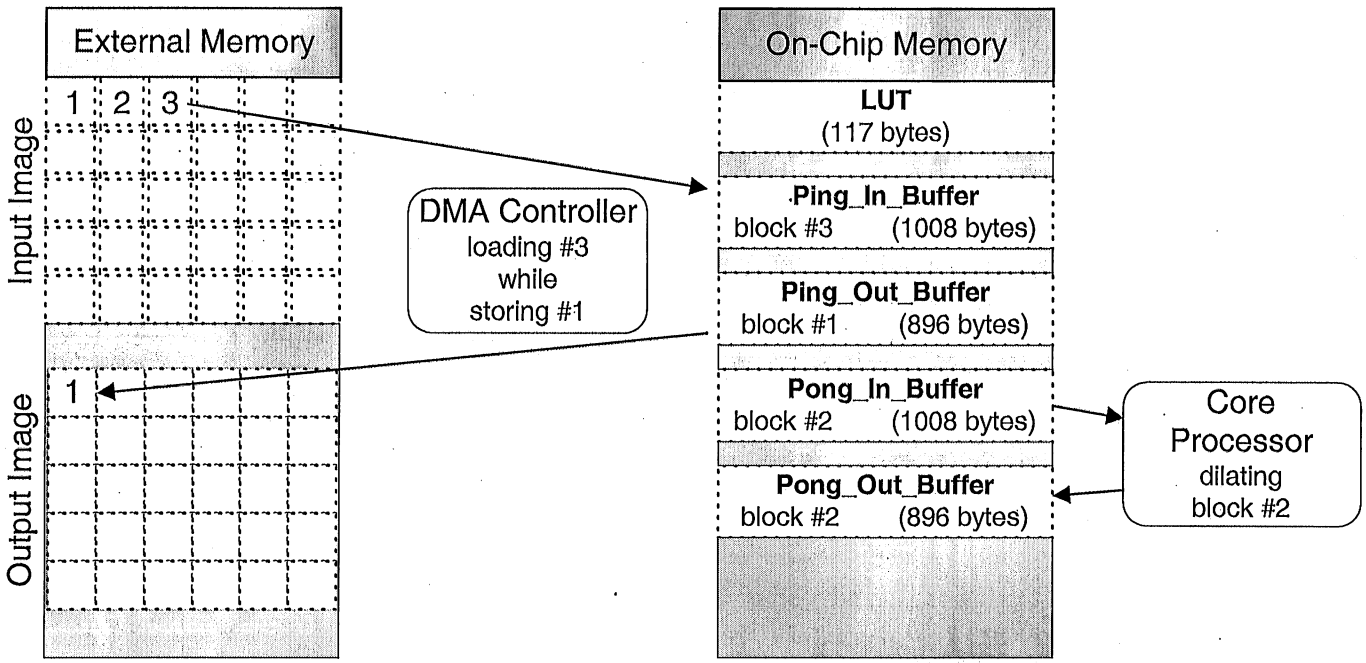

Figure 9. Double buffering the dataflow using a programmable DMA controller.

The DMA controller can also be programmed to automatically pad the image border with zeros to prevent edge artifacts from occurring. We have developed a unique method to perform this padding with little additional overhead. As Figure 10 shows, since the interior blocks are not on the boundary, they require no padding. Therefore, the processor dilates these blocks first while the DMA controller performs the padding of the boundary pixels around the exterior blocks. This padding is completed by the time the core processor finishes dilating the interior blocks, thus can automatically begin dilating the exterior blocks with no additional overhead. Some researchers handle the boundary pixels as a special case by the processor in the computation loop[4]. This can add extra overhead due to the branching and extra instructions required in the inner computation loop. Instead, by using the programmable DMA controller to pad the border of the image, the core processor can be allowed to continue computing without the overhead of checking for border pixels.

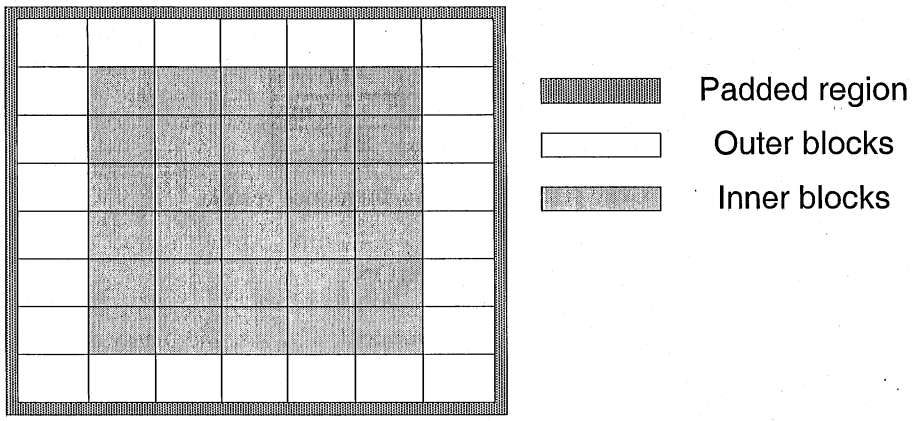

Figure 10. Example of a padded image.

The alternative to using the DMA controller to bring the data on and off chip is to rely on the natural caching mechanism of the MAP1000's data cache. Most microprocessors today use a caching mechanism, whereby simple loads and stores by the core processor stimulate the data cache to bring the external data on and off chip. If the data does not already reside in the cache, a cache miss penalty occurs, in which the cache must first store a cache line (32 bytes) to external memory for coherency reasons, then load the requested data into the cache line. The core processor must stall until the necessary data are ready. When we used the caching mechanism instead of the DMA controller, the overhead increased, ranging from 17% to 67% as shown in Table 1.

3. RESULTS AND DISCUSSION

Table 1 and 2 show the results of binary dilation and gray-scale dilation for the TMS320C80 and MAP1000 mediaprocessors for 512x512 images and various structuring element sizes. The TMS320C80 has 4 internal parallel processors operating at 50 MHz, while the MAP1000 has 2 clusters operating at 200 MHz. For both processors, the total number of gray-scale pixels processed in parallel is 16 and the number of binary pixels processed in parallel is 128. Thus, the MAP1000 with a four time faster clock should ideally have four times better performance*. To achieve efficient performance on the TMS320C80, we found it necessary to program in assembly language. On the other hand, the MAP1000 programmed in C is better than assembly code on the TMS320C80, resulting in 4.7 to 5.6 times faster execution in gray-scale dilation and 5.0 to 7.7 times faster in binary dilation. When comparing assembly code to assembly code implementations, the MAP1000 was 6.9 to 11.5 times faster in binary dilation.

The MAP1000 compiler's ability to software pipeline was only 50% slower than our hand-optimized software pipelining in assembly language for the binary dilation, indicating an efficient compiler. Our previous experience with C versus assembly coded functions on the TMS320C80 has typically resulted in 8:1 performance difference[13]. Table 1 also shows the impact of using the DMA controller versus the normal caching mechanism on the MAP1000. Using the DMA controller increased the speed by a factor ranging from 1.2 to 1.7.

Structuring Element Size	C80 DMA Asm code	MAP1000 No DMA C code	MAP1000 DMA C code	MAP1000 DMA Asm code
3x3	9.2 ms	2.0 ms	1.2 ms	0.8 ms
5x5	14.4 ms	3.4 ms	2.6 ms	1.8 ms
7x7	22.8 ms	5.4 ms	4.6 ms	3.3 ms

Table 1. Binary dilation performance results.

* This assumes all overhead scales accordingly, such as memory speed, which is not necessarily valid.

Structuring Element Size	C80 Asm code	MAP1000 C code
3x3	32.7 ms	7.0 ms
5x5	68.7 ms	14.2 ms
7x7	136.3 ms	24.5 ms

Table 2. Gray-scale dilation performance results.

4. CONCLUSION

For a 515x512 image and 3x3 structuring element, gray-scale dilation took 32.7 ms, and binary dilation took 9.2 ms on the TMS320C80 mediaprocessor. On the new MAP1000 mediaprocessor, a faster execution of 7.0 ms for gray-scale dilation and 0.8 ms for binary dilation was achieved. These results offer comparable performance to ASIC-based approaches proposed in the literature (e.g., a gray-scale dilation ASIC takes 8.8 ms* at 30 MHz[5]) and improved performance over previously reported programmable approaches (e.g., binary dilation took 43.2 ms* on Sun SparcStation[4]). Programmable mediaprocessors offer more flexibility than the ASIC-based approaches, which usually impose limitations on the size and shape of the structuring elements (e.g., 3x3 only) and require a fixed structuring element origin. Also, unlike hardwired ASICs, mediaprocessors can be programmed to implement other functions as well, such as further morphology functions, connected components, and higher-level vision tasks. While older mediaprocessors like the TMS320C80 could only achieve efficient performance by using assembly language, the C compilers for the new mediaprocessors are producing code that is closing the gap with assembly versions, as illustrated by our results for binary dilation on the MAP1000. However, to achieve this substantial improvement over general-purpose microprocessors, the algorithm must be carefully mapped to the architecture of the mediaprocessor by (1) utilizing the parallel processing units and partitioned operations, (2) converting the structuring element into a special LUT, avoiding redundant computations, (3) utilizing the full capacity of the multiple execution units using software pipelining, and (4) managing the dataflow with the programmable DMA controller, minimizing the I/O overhead and automatically handling details like image padding and block alignment.

5. REFERENCES

1. J. Serra, *Image Analysis and Mathematical Morpholog, vol. 1.* Academic Press, London, 1982.
2. R. C. Gonzalez and R. E. Woods, *Digital Image Processing.* Addison-Wesley, New York, 1993.
3. R. M. Haralick, S. R. Sternberg, and X. Zhuang, "Image analysis using mathematical morphology," *IEEE Trans. Pattern Recognition Machine Intelligence* **9**, pp. 532-550, 1987.
4. R. V. Boomgaard and R. V. Balen, "Methods for fast morphological image transforms using bitmapped binary images," *Graphical Models and Image Processing* **54**, pp. 252-258, 1992.
5. I. Andreadis, A. Gasteratos, and Ph. Tsalides, "An ASIC for fast grey-scale dilation," *Microprocessors and Microsystems* **20**, pp. 89-95, 1996.
6. C. C . Weems, S. P. Levitan, A. R. Hanson, and E. M. Riseman, "The image understanding architecture," *International Journal of Computer Vision* **2**, pp. 251-282, 1989.
7. M. J. B. Duff and T. J. Fountain, *Cellular Logic Image Processing.* Academic Press, London, 1986.
8. G. York, C. Basoglu, and Y. Kim, "Real-time ultrasound scan conversion on programmable mediaprocessors", *SPIE Medical Imaging* **3335**, pp. 252-262, 1998.
9. C. Basoglu, R. Gove, K. Kojima, and J. O'Donnell, "A single-chip processor for media applications: The MAP1000", *International Journal of Imaging Systems and Technology*, in press, 1998.
10. C. Basoglu, W. Lee, and Y. Kim, "An efficient FFT algorithm for superscalar and VLIW processor architectures," *Real-Time Imaging*, in press, 1999.
11. R. A. Managuli, C. Basoglu, S. D. Pathak, and Y. Kim, "Fast convolution on a programmable mediaprocessor and application in unsharp masking," *SPIE Medical Imaging* **3335**, pp. 675-685, 1998.
12. M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," *SIGPLAN: Conference on Programming Language Design and Implementation*, pp. 318-328, 1988.
13. I. Stotland, D. Kim, and Y. Kim, "Image computing library for a VLIW multimedia processor," *SPIE Electronic Imaging,* **3655**, in press, 1999.

* Times scaled for equivalent image and structuring element sizes and assuming no I/O overhead.

APPENDIX: CONVERTING GRAY-SCALE TO BINARY BIT-MAPPED IMAGES

In the absence of a specific instruction to convert gray-scale images into binary bit-mapped images, we have developed a unique approach using the mediaprocessor's *inner-product* instruction to perform the conversion, as shown in Figure 11. First, a set of eight 8-bit input pixels are loaded with one 64-bit load. A partitioned compare instruction (*compare_less_than* if thresholding or *compare_equal_to* if selecting a particular value) is used to select those gray-scale values that are to become a binary "1" and that are to become a binary "0". This outputs an hexadecimal "FF" for each binary "1" and a "00" for each binary "0". Bitwise and (*band*) is then required to convert the "FF" values into the binary "1". At this point, the eight values are binary, but represented in 8 bits, not bit-mapped to one bit. Therefore, the *inner-product* instruction is used:

$$\sum_{i=0}^{7} X_i \cdot 2^i \qquad (6)$$

to multiply each pixel by its respective bitplane (2^i) and then sum these results to create the eight bit-mapped bits. For a 512x512 image, this conversion adds about 0.32 ms of execution time to dilate.

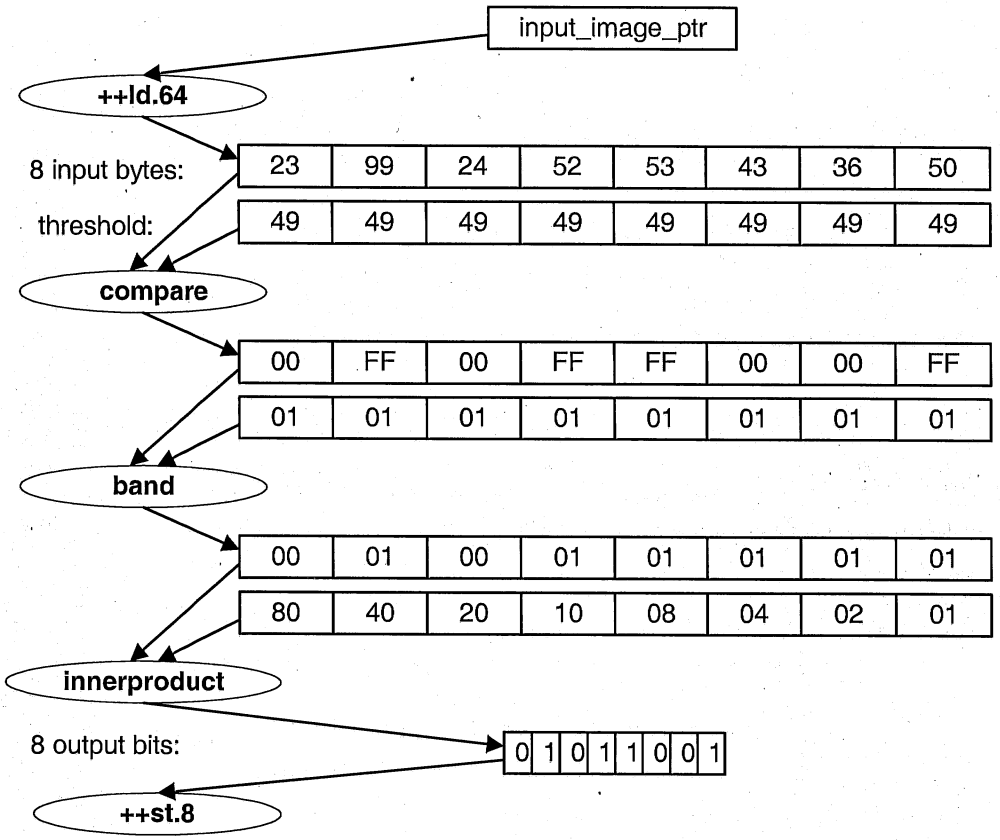

Figure 11. Converting gray-scale image to a bit-mapped binary image using the *inner-product* instruction.