An Image Algebra Based SIMD Image Processing Environment

Joseph N. Wilson, E. Jason Riedy, Hongchi Shi, and Gerhard X. Ritter

December 3, 1997

Abstract

SIMD parallel computers have been employed for image related applications since their inception. They have been leading the way in improving processing speed for those applications. However, current parallel programming technologies have not kept pace with the performance growth and cost decline of parallel hardware. A highly usable parallel software development environment is needed. This chapter presents a computing environment that integrates a SIMD mesh architecture with image algebra for high-performance image processing applications. The environment describes parallel programs through a machine-independent, retargetable image algebra object library that supports SIMD execution on the Lockheed Martin, PAL-1 parallel computer. Program performance on this machine is improved through on-the-fly execution analysis and scheduling. We describe the relevant elements of the system structure, outline the scheme for execution analysis, and provide examples of the current cost model and scheduling system.

Keywords: SIMD, image algebra, image processing environment

*J. N. Wilson, E. J. Riedy, and G. X. Ritter are with Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32612. Email: {jnw, ejr, ritter}@cise.ufl.edu.

†H. Shi is with Department of Computer Engineering and Computer Science, University of Missouri-Columbia, Columbia, MO 65211. E-mail: shi@cecs.missouri.edu.
1 Introduction

The advent of VLSI technology led image processing researchers to use such SIMD, mesh-connected computers as the ILLIAC [2], the CLIP series [3, 4, 5], the DAP [6], the MPP [7], the GAPP [8, 9], and the Hughes 3D Computer [10] for improving their codes' performance. Both the performance and cost-effectiveness of SIMD machines have improved steadily. However, the current software development systems are still comparable to assembly language programming for traditional sequential systems [11, 12]. Each parallel computer has its own language which runs efficiently on only its architecture. Various approaches striving for architecture independence have been proposed [13], ranging from the use special-purpose languages [14] to special implementations of general-purpose languages [15]. A unified software development environment for these parallel systems has yet to appear.

Unified frameworks for image-related applications have interested many researchers. Many efforts have been devoted to searching such a unified framework that can serve as a model for algorithms dealing with image objects and fit well into the theory and practice of parallel computing [16]. Mathematical morphology provides a mathematical framework for expressing a large number of algorithms for image processing and analysis [17, 18, 19, 20] through image filtering and structuring elements. Morphology-based systems ignore important operations like transformations between different domains and between different value sets. The image algebra developed by Ritter and his colleagues [21, 22] provides a more general framework for image-related applications. Image algebra incorporates and extends mathematical morphology, providing more general image-template operations that support the elements missing from morphology. It defines images in the broadest sense and is widely applicable. Image algebra provides a common algebraic framework for algorithm development, optimization, comparison, coding, and evaluation.

In this chapter, we present a parallel environment based on image algebra suitable for SIMD machines. The environment keeps developers at a comfortable level of abstraction, specifying algorithms symbolically and algebraically, while automatically partitioning the image data and scheduling operations to achieve optimal performance. Specifically, we discuss the use of the retargetable Image Algebra C++ library [23] for image processing on the Lockheed Martin PAL-i SIMD computer, a fine-grained, SIMD-parallel computer. Modifying the iac++ library to provide efficient SIMD execution on the PAL system requires the development of a new image representation class, implementation of a client-server system, development of a strategy to reduce data transfers, and creation of a cost measure to control that strategy.

The next section briefly describes the properties of the PAL-i SIMD machine. The image algebra environment structure and the operations and operands provided by the iac++ library are described in section 3. Section 4 discusses retargeting the iac++ to the PAL. The cost function used to direct the evaluation of programs on the SIMD array to reduce the required processing time is presented in section 5. The heuristics applied to optimize the cost function are described in section 6. Section 7 contains possible improvements to the cost model and heuristics. Section 8 follows with a few examples of a simple cost model and evaluation heuristic. Finally, section 9 closes with a direction for future work and notes on the implementation in progress.

2 The PAL-I SIMD Processing System

The PAL-I SIMD processing system is a workstation-based image processing system. The PAL processor array itself is an attached image processing accelerator. The system's current workstation platform is a Sun SparcStation 4. The PAL processor array is attached to such a workstation with an edt scd-40 configurable dma interface [24]. This provides a nominal 40 megabyte per second (mb/s) connection between the host workstation and the PAL processor array. The PAL processor array is a multiple board uvm system with one controller board and one or more processor array boards. Each processor board contains 4,608 one-bit processing elements (pe's) arranged in a 72/64 grid. The typical configuration of a PAL-i system has two processor array boards...
3 Software Environment Based on Image Algebra

3.1 Operands and Operations of the iac++ Library
Figure 1: The image classes in the iac++ library are specialized for efficiency. class at the diamond end contains a pointer to the class at the box end. A class at the tail of a dashed, directed line derives from the class at the head of that line.

The operands of iac++ are drawn from the image algebra developed at the University of Florida by Ritter and Wilson [27] and fall into the following categories:

1. points and sets of points,
2. values and sets of values,
3. images (functions from points to values),
4. neighborhoods (images with values that are sets of points), and
5. templates (images with values that are images).

The iac++ class library provides these operands via a collection of C++ template classes [28]. The groups of operands listed above are represented as follows:

1. IA_Point<int> and IA_Point<double> represent points with integral and floating-point coordinates (respectively) and IA_Set<IA_Point<int>> and IA_Set<IA_Point<double>> represent sets containing these points.
2. Values are represented by the built-in C++ types bool, unsigned char, int, and float and the additional types IA_RGB and IA_Complex. Sets containing elements of type T are provided by the type IA_Set<T>.
3. The image classes have C++ template arguments specifying the kind of points mapped and the type, T, of elements. Most images are discretely sampled and of the form IA_Image<IA_Point<int>, T>.
4. The neighborhood classes have C++ template arguments specifying the coordinate type of the domain points and the range points. The only presently implemented class is IA_Neighborhood<int, int>.
5. The presently implemented templates map discrete point sets to discrete images. The C++ template argument, T, to the class IA_DDTemplate<T> tells the type of image to which the template maps integral-coordinate points. Templates on images with integer points and value types bool, unsigned char, int, float, and IA_Complex are supported in the library.
3.2 The Image Algebra Environment Structure

4 Retargeting the iac++ Library to the PAL
5 Cost Model

5.1 Cost of Transfers
Visual Communication and Image Processing

Figure 3: Blindly applying templates leads to boundary effects (lightly shaded) along both image (solid) and array-induced (dashed) borders.

The straightforward method of breaking images into exactly array-sized blocks produces erroneous results. Template and neighborhood operations along image borders produce boundary effects by sampling outside the image. The boundary effects also occur along the induced, internal borders. Figure 3 shows these effects.

The common solution for external boundary effects is to extend the image with a known value. This solution works for many operations, but the padding around the image must still be transferred to the array. For the current discussion assume the image’s padding is loaded as a part of the image. In practice, initializing a transfer buffer with the padding value and assembling the image into the buffer also solves problems associated with images having non-rectangular point sets. Some platforms may have an operation that extends a subframe on the \textit{simd} array more efficiently. The padding term in equation 2 below is still necessary to calculate the number of blocks.

Some operations, such as an image-template product which combines through addition and reduces by minimization, do not have a single, convenient padding value. For additive minimum operations, the reduction identity is not preserved by the combination operator. For example, combining an 8-bit, unsigned padding value of 255 with a template value of one results in an 8-bit value of zero. That zero will be the minimum value and the result of the template. Fixing the padding values requires finding the correct minimum. That circularity implies that another method is needed.

The min operator uses the sum’s result only if that point is in the original image, otherwise it uses its identity. The correct value from either the sum or the identity is chosen according to a mask image. The mask has a value of one at each point in the original image’s point set and zero elsewhere. This mask image must be transferred to the array as well. The mask should be sent only if necessary on a per-block basis. This is dependent upon the point set of the original image; a sparse point set such as \( f(i/2; j/0; i/0; j/256) \) would require a mask on every subframe. Again, some architectures may provide more efficient mechanisms for creating masks on the processor array. The results below assume a mask is transferred with every block and provide an upper bound on the number of transfers.

The internal boundary effects also need variable-valued padding. The necessary padding values, however, are already available in the image and are loaded with the block. Avoiding internal boundary effects reduces the region of the array containing valid results as shown in figure 4. This valid region tiles the image and determines the number of subframes to be loaded.

5.1.1 Number of Subframes

\( \{ i, j \mid i \leq i < , j \leq j < \} \)
The functions \( l_d \) and \( r_d \) denote the extent of a template's bounding box on either side of its origin. Each image axis is the length of the padded image divided by the length of the valid region. The total number of subframes is the product of these across all the image's dimensions. If the dimensionality of the image does not match the dimensionality of the processor array, more complicated subframing methods must be applied.

Let \( t \) be a sequence of templates and neighborhoods applied to an \( L_x \times L_y \) image \( A \). The SIMD array contains \( L_x \times L_y \) processing elements. Each template can be fitted inside a bounding box. The sequence \( t \) has a corresponding sequence of bounding boxes, \( a \). On these boxes, define the functions \( l_d(b) \) and \( r_d(b) \) as shown in figure 5. The functions determine the extent of the bounding box on either side of the template's origin along dimension \( d \).

Define the function \( \text{pad}_d \) on sequences of bounding boxes to be

\[
\text{pad}_d(a) = \max \left( \begin{array}{c} l_d(b) \\ r_d(b) \end{array} \right) \quad \text{for} \quad d \in \{x, y\}.
\]

This is the total length of padding needed along dimension \( d \).

The total number of subframes of \( A \) to be transferred is

\[
B = \prod_{d \in \{x,y\}} \left[ \frac{L_d - \sum_{b \in a} l_d(b) - r_d(b)}{l_d(b) - r_d(b)} \right].
\]
Figure 6: The valid region of image-image operations is the intersection of the operands' valid regions.

\begin{verbatim}
IA_Set<IA_Point<int>> domain = IA_boxy_pset(IA_Point<int>(0, 0), IA_Point<int>(255, 255));

// Images A and B are to use the SIMD array, C and D will inherit that property...
IA_Image<IA_Set<IA_Point<int>>, int> A = IA_PIMImage(IA_Image<IA_Point<int>, int>(domain, 1)),
B = IA_PIMImage(IA_Image<IA_Point<int>, int>(domain, 2)),
C, D;

C = A + B;
D = C * A;
cout << sum(D);
\end{verbatim}

Figure 7: Because C is still in scope, evaluation of D also evaluates and stores C.

Root's state variables. The number of blocks to be transferred is compositional with these operations. In
general, B is a function of the immediate history of the computation.

Equation 2 only gives the number of subframes necessary for a sequence of template and neighborhood
operations. Unary and image-scalar operations do not affect the valid region's size, so they can be introduced
freely. Operations between two images do affect the valid region as shown in Figure 6. The result's valid
region is the intersection of the arguments' valid regions. The new state variables are the maximum values
of the arguments' state variables. Note that the regions must be aligned consistently. In general, results of
template operations should be shifted back to their original points. This adds to the computational cost.

The merges from image-image operations also add dependencies on the order of operations and introduce a
limited form of shared subexpressions.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{example.png}
\caption{Example of image operations.}
\end{figure}

5.1.2 Transfer Time

Not all computations return a single result. Figure 7 shows one such case. The global reduction in line 15
triggers evaluation of D, which in turn evaluates C. The results for C may be used again later, so they
must be transferred back to the host. Had line 15 been written as D = (A + B) * A, the temporary result
would be out of scope by line 16, and the subresult of A + B would not be returned. The example is trivial, but
real code often contains unused temporaries for readability. Conservative dependency assumptions are a
Figure 8: This convolution path involves nine shift operations, eight for calculations and one for returning to the origin.

A significant limitation of systems that do not modify the original source.

For a general sequence of operations, the total transfer time into the array, \( T_{\text{in}} \), and the total transfer time out of the array, \( T_{\text{out}} \), are

\[
T_{\text{in}} = \frac{B}{0} \sum_{\text{operands and masks}} T - A,
\]

\[
T_{\text{out}} = \frac{B}{0} \sum_{\text{results}} T - A.
\]

where \( \text{rep(size}(A)) \) is the size of the machine representation of \( A \)'s values, and \( T(s) \) is the time needed to load a block of representation size \( s \) onto the processor array. The setup time is the small time required to initiate the transfer.

Many architectures are designed for streaming data and instructions. The general, transparent method attempted here must deal with the staccato bursts caused by intermediate return values and by delays in the controlling program. These bursts often will produce nonlinear times from low-level setup costs. Hence, the values of \( T \) should be determined experimentally to counter possibly nonlinear transfer times.

Note that the set of temporary results may change during execution. Results are not returned to host-side variables which have passed out of scope. At any given point in the tree, the \( T_{\text{out}} \) term determined during creation is an upper bound. Updating the \( T_{\text{out}} \) values and propagating the new values up the tree is similar to the shared-subexpression problem mentioned later.

5.2 Cost of Computations

5.2.1 Image-Template Operations

\( A \odot t \)
The cost $C$ for computing an image-template product $A \@ t$ with $B$ subframes is:

$$C = B \left( \gamma | \circ \circ \gamma \right) - 1 \left| \right|.$$  

5.2.2 Neighborhood Operations

$$C \quad A \@ n \quad B$$

$$C \quad B \left( \left| n \right| - \gamma - \right) \left| \right|.$$  

5.2.3 Unary, Image-Scalar, Image-Image operations

$$s \quad A \quad A_1 \quad A_2$$

$$C \quad B \quad .$$  

5.2.4 Global Reductions

5.2.5 Functional Composition

6 Scheduling Evaluation of Trees
7 Possible Improvements

7.1 Weak Template Decomposition

\[ A \otimes t \Rightarrow A \otimes t_1 \gamma \quad A \otimes t'_2 . \]

\[ A \otimes t_1 \quad A \otimes t'_2 \]
// Say A, B, C, and D are defined as before, and \( t_1 \) and \( t_2 \) are simple, small templates.

\[
C = \text{linear_product}(A, t_1) + B; \\
D = C \times \text{linear_product}(A, t_2);
\]

\[\text{cout} \ll \text{sum}(C)\]

\[\text{Figure 9: The global reduction of } C \text{ will invalidate the state information for the tree rooted at } D.\]

7.2 Shared Subexpressions

7.3 Finer Optimizations

8 Examples
8.1 Roberts Edge Detector

\[ A \quad e \quad \sqrt{A \oplus s^2} \quad A \oplus t^2 \quad E \]

\[ Q \]

\[ s \quad t \]

\[ c_i \quad n_i \]

\[ n_6 \quad n_5 \quad n_3 \]

\[ i \quad Q_i \quad B_i \quad \tau_i \quad c_i / n_i \]

\[ \times -5 \]

\[ \times -3 \]

10C/FB

8.2 Three-Level Wavelet Transform

\[ W_6 \quad \times \quad A \quad h_i \quad g_k \]

\[ x \]
Figure 11: The cost $Q$ associated with the expression tree for $p(A/s) + (A/t)$ strictly decreases. The times for $/b$ and $c$ are in units of 10$^{-3}$ seconds, and the time for $Q$ is in seconds.

$h_0 = g_0$
$h_1 = g_1$
$h_2 = g_2$
$h_3 = g_3$
$h_4 = g_4$

Figure 12: The high-pass ($h_i$) and low-pass ($g_i$) filters sample from successively farther locations. The shaded boxes form the non-zero support of each template.

$\sqrt{A \oplus s^2} = A \oplus f^2$

$B \quad n \quad \tau \quad c \quad Q$
\begin{align*}
\times & \quad . \\
\times & \quad \tau_1 \quad c_1 \quad . \\
\tau_2 & \quad c_2 \quad . \\
\times & \quad \tau_1 \quad c_1 \quad . \\
\tau_5 & \quad c_4 \quad . \\
\times & \quad \tau_5 \quad \tau_3 \quad c_5 \quad c_3 \quad . \\
\tau_6 & \quad c_6 \quad . \\
\end{align*}
Figure 14: Assuming the transfer cost overpowers the computation cost, this expression tree's cost per operation will increase after step 6, forcing evaluation of $A/h_0/h_1$.

Figure 13 holds the basic wavelet decomposition code. The arrays hold the similarly-named filters for each level. As the server builds the tree in figure 14, it maintains the maximum and sum of $l_x$ and $r_x$ over the templates to be applied to the image. The total number of blocks $B$ is calculated as in equation 2. When the server adds the $/g_2$ operation to the tree, the total number of blocks transferred per operation increases. The simple heuristic we use indicates that the delayed expression $A/h_0/h_1$ should be evaluated.

As figure 14 demonstrates, this evaluation decreases the number of blocks, but it also decreases the number of operations. The average number of blocks transferred actually increases more when $A/h_0/h_1$ is fully evaluated. This suggests another possible heuristic: Evaluate when evaluation does not increase the cost more than another delay does.

In the previous example, we counted the transfer time for both loading and returning the image data. The transfer cost out is substantial in the current example. A frame buffer can hold the results and delay returning them to the host. The potential savings for such algorithms as wavelet analysis, where the images will be filtered and recombined before being returned, are great. A limited frame buffer presents significant challenges for the cost model. The cost of managing the memory allocation should be included, especially with access from multiple clients. Currently, we ignore the return time and assume the result remains in the frame buffer. This should be a fair approximation for algorithms that produce few temporary results and for environments where few clients use the same ioC/fb unit.

9 Summary

We have presented a general, cost-based model for optimizing image processing operations for the pali computer. We have shown how the operations of image algebra affect both the cost of computation on the pali and the cost of communication of data to the pali. The model balances these costs with a set of heuristics, scheduling expression evaluation on-the-fly.

Clearly, there are many possible choices for heuristics and even for parameters within the heuristics. A systematic study of image processing algorithms should reveal which parameters work well for sets of...
Acknowledgements

References


<table>
<thead>
<tr>
<th>J</th>
<th>J</th>
<th>D</th>
<th>WSI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>K</td>
<td>MA</td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J</th>
<th>J</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J</th>
<th>NY</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>J</td>
<td>SIMD</td>
</tr>
<tr>
<td></td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
3 \ 83) \\
\]

\[
f_i \quad - \quad w \\
\]

<table>
<thead>
<tr>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CA</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>f_i</td>
</tr>
<tr>
<td>G</td>
<td>#</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J</th>
<th>f_i</th>
<th>O j  - O</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FL</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CRC</th>
</tr>
</thead>
</table>
++

J

X:

&

CBMS-NSF

SIA M PA