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 [1]. 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-I 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++ object library `iac++` [23] for image processing on the Lockheed Martin PAL-I 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 IOC/SA 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 IOC/SA 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 6U VME system with one controller board and one or more processor array boards. Each processor board contains 4,608 one-bit processing elements (PEs) arranged in a 72×64 grid. The typical configuration of a PAL-I system has two processor array boards and contains 9,216 PEs. Clocked at 40 MHz, this system can execute over 368 billion bit operations per second. For 32-bit integer operations, this translates into execution speeds on the order of up to billions of operations per second. Because the processors are connected in a two-dimensional mesh network, performance of this system on local neighborhood operations can far outstrip any sequential computer.

Efficiently programming such a SIMD processor system presents challenges. The sustained throughput of the SCD-40 card in real situations is somewhat less that 10 MB/s. Thus, transfer of a 9,216 pixel image with 8-bit pixel values will typically consume over 30,000 SIMD clock cycles. Even with its single-bit architecture, the PAL system can implement most 32-bit floating-point operations in just hundreds of clock cycles. The system is capable of executing between dozens and hundreds of operations in the time it takes to transfer an operand from host memory onto the processor array.

Two primary mechanisms are used to overcome such problems: designing the system to simultaneously transfer data and execute operations, or employing a cache memory hierarchy to place data near the processors. Both these activities can improve performance of a SIMD system such as the PAL-I, but they cannot completely overcome the severe imbalance between transfer rate and computation speed presented by the IOC/SA. Although it can halve the processing time spent on any computation, simultaneous transfer fails because the processor speed in our setting is still much faster than the doubled transfer speed. Providing a cache memory hierarchy will improve performance, but will still face the problem that image operands frequently occupy megabytes of storage. Even if one can store a few operands of this sort directly in the processing elements, the next level of the cache will incur some transfer penalty. A large frame buffer can dramatically improve many algorithms' performance, however.

The user interface provided by Lockheed Martin for programming the PAL system is the PAL Workstation Development Library (`PAL_WS`). This library supports parallel computation on SIMD-array-sized chunks of images and provides rudimentary support for creating larger images and directing operations to be carried out upon these images. Images are stored on the host workstation and transferred to the PAL processor array as necessary to carry out the specified operations.

3 Software Environment Based on Image Algebra

The ideal environment for application software development would rise from combining a simple, reasonable model with compilers that bridge the gaps between the model and specific architecture. Sequential computing has such systems, but parallel computing does not. Although there are more parallel computer models than parallel computer vendors, no suitable, all-encompassing model exists [25]. Neither do compilers capable of exploiting every extant parallel architecture. Indeed, difficulties in formulating a single, useful model make it appear unlikely that we will every have a single, useful environment for all parallel development. Thus, a practical methodology for application software development on parallel computers is to write algorithms using abstract libraries of fundamental data operations, implementations of which are optimized for specific computers. We adopt this approach and develop an image algebra based, parallel environment that augments the C++ language with image algebra operations.

3.1 Operands and Operations of the `iac++` Library

The `iac++` library provides classes of objects and related operations that are well suited to specifying image processing and computer vision algorithms. It was developed at a high level of abstraction, thus it is independent of any specific computer architecture. Furthermore, its software architecture allows it to be retargeted to exploit the capabilities of special-purpose computer architectures and devices. This class structure is shown in figure 1. In this figure, we use the notation of the Unified Method of Booch and Rumbaugh [26]. Boxes represent classes. The word abstract indicates that there are no direct object instances of a given class. The dashed boxes denote any parameters of the class. Solid lines imply that the

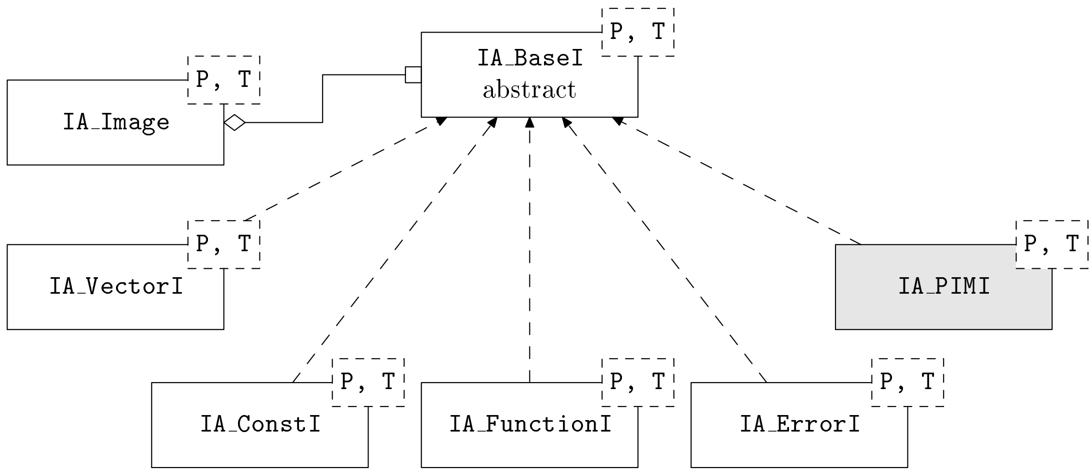


Figure 1: The image classes in the iac++ library are specialized for efficiency.

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> represents points with integral and floating-point coordinates (respectively) and IA_Set<IA_Point<int> > and IA_Set<IA_Point<double> > represents 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>. One continuous image type, IA_Image<IA_Point<double>, float>, is provided.
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.

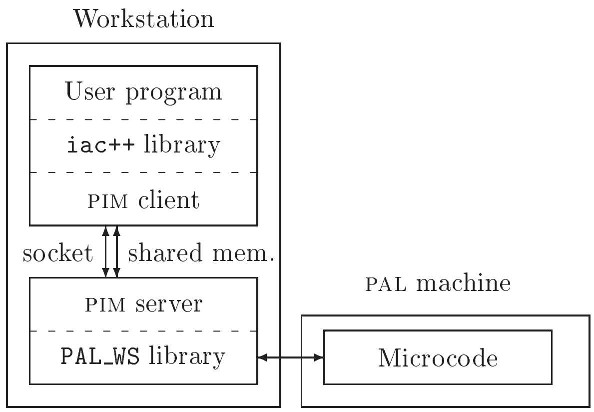


Figure 2: The user's program uses the PIM client through the `iac++` library to talk to the PIM server. The server in turn uses the `PAL_WS` library to communicate with the PAL device.

Operations upon images fall into several general categories:

1. binary and unary pointwise operations,
2. global reductions,
3. neighborhood and template reductions, and
4. composition with point-to-point or value-to-value functions.

These operations are provided, as appropriate, by overloaded operations on the image classes, by class member functions, and by overloaded functions.

3.2 The Image Algebra Environment Structure

The structure of the image algebra environment on the PAL-I system is depicted in figure 2. The current PAL image representation class, `IA_PIMI<P,T>`, implements its operations through a client-server system. The `iac++` client application directs the PAL Image Manager (PIM) through socket-sent operations. Image data is placed in shared memory segments, eliminating costly, needless copying. This client-server architecture allows us to mediate PAL access between multiple clients.

To effectively support image algebra based programming on the PAL-I, we needed to re-target the `iac++` library to the PAL-I and to implement data partitioning and operation scheduling strategies to efficiently schedule image algebra operations on the PAL-I under a cost model.

4 Retargeting the `iac++` Library to the PAL

The retargeting of the library is supported by the separation of user interface from the representation employed in the class hierarchy. As shown in figure 1, the image classes are all instances of a single C++ template class, `IA_Image<P,T>`. These classes provide the user interface to such images. Each of these user interface classes contains a reference (or handle) to an object of type `IA_BaseI<P,T>`, the base class for image representations. A variety of specific representation classes are derived from `IA_BaseI<P,T>` to effect efficient sequential implementation of the library, such as `IA_VectorI<P,T>`, `IA_ConstI<P,T>`, and `IA_FunctionI<P,T>`, which represent an image as a vector of values, a constant value, and a function mapping points to values, respectively. This arrangement of classes ensures that the behavior of any image in response to user-issued operations is dependent upon its representation class. To re-target the library to support the PAL IOC/SA system, we have developed an image representation class, `IA_PIMI<P,T>`, whose operations are implemented on the PAL.

If we are to use a SIMD computer to operate upon images having many more pixels than there are SIMD processing elements, then we must serialize our computation to some extent. One straightforward way is to break image operands up into smaller image subframes or blocks. This breaks the operation into a sequence of array-sized, parallel computations.

Suppose we wish to evaluate the expression $R = (A + B) \cdot C$. The value of each pixel in R is the sum of the corresponding pixels in A and B times the value of corresponding pixel in C. Assume that A, B, and C are so large that they occupy more memory than the entire PE array contains. Any single image operation cannot be computed on the PE array without being serialized. (This assumption is in fact warranted for many applications of the PAL and other SIMD systems.) Suppose we divide the images into k corresponding blocks $A_1, \ldots, A_k$, $B_1, \ldots, B_k$, and $C_1, \ldots, C_k$. There are two fundamentally different, yet correct strategies to evaluate these operations.

In the first approach, we carry out operations on images in the order specified by the expression, fully evaluating each image operation and yielding an entire image result. In our example, for each i in 1 to k, we calculate $T_i = A_i + B_i$, then for each i in 1 to k, we calculate $R_i = T_i \cdot C_i$. This approach produces temporary results that must be stored on the host.

In the second approach, we carry out operations on the entire expression for each subframe block, generating an image composite after completing the work on each of the subframes. Thus, for each i in 1 to k, we calculate $R_i = (A_i + B_i) \cdot C_i$.

Temporary results should be avoided when they must be transferred over a slow path. In the second approach, these temporary operands need never be transferred, and hence computations are completed more quickly. The PIM server exploits this fact, using lazy evaluation [29] of image operations. When a client program operates upon images, the server constructs an expression tree that represents the computation to be performed. When the client program attempts to use the pixel values of an expression's result, we evaluate the associated tree and carry out as many operations as possible on each subframe.

It might appear that the second evaluation approach is the best and should be used as the only serialization rule. This ignores neighborhood and template operations' requirements of data from neighboring pixels. Longer sequences of neighborhood or template operations require larger neighboring regions in order to calculate a pixel's value. We cannot calculate valid results for neighborhood and template operations at every processing element, but only those whose neighbors have valid values. This leads us to divide image operands into overlapping subframes. As the number of such subframes increases, the time to communicate the original image grows, eventually overcoming the savings we achieve by avoiding temporary results. If we can express the cost incurred during a sequence of operations, we can attempt to analyze and minimize the cost dynamically.

5 Cost Model

We assume a two-dimensional system. Most restrictions to one dimension and extensions to higher dimensions are straightforward. The total cost breaks into two sub-costs, the cost of transferring data and the cost of computing results. Since the goal is to minimize execution time, the cost is expressed in units of time.

5.1 Cost of Transfers

Assume the image of interest is larger than the SIMD array. Then any computation must be broken into array-sized blocks or subframes. Each input subframe needs transferred to the array, and each result subframe needs shipped back out. The total transfer cost is the number of array-sized subframes transferred times the

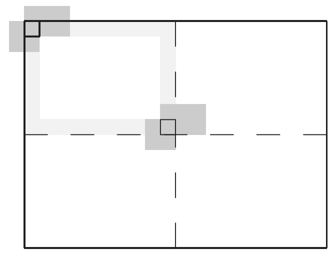


Figure 3: Blindly applying templates leads to boundary effects (lightly shaded) along both image (solid) and array-induced (dashed) borders.

cost of transferring a block.

5.1.1 Number of Subframes

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 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 $\{(i^2, j) | 0 \leq i < 16, 0 \leq j < 256\}$ 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.

For two-dimensional images on a two-dimensional processor array, the number of block boundaries along

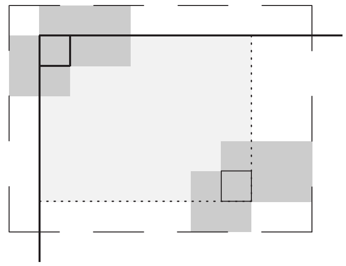


Figure 4: Each subframe (dashed) yields a smaller result (dotted).

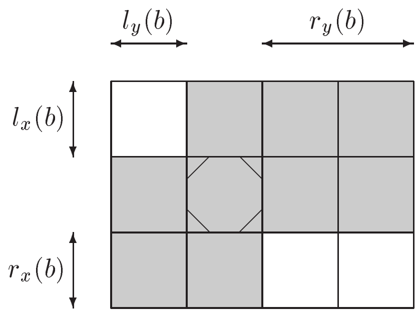


Figure 5: The functions l_d and r_d denote the extent of a template's bounding box on either side of the its origin.

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 $\mathcal{L}_x \times \mathcal{L}_y$ image A. The SIMD array contains $L_x \times L_y$ processing elements. Each template can be fit inside a bounding box. The sequence t has a corresponding sequence of bounding boxes, $\mathbf{a}$. 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 $d \in \{x, y\}$.

Define the function pad_d on sequences of bounding boxes to be

$$\mathrm{pad}_d(\mathbf{a}) = \max\{l_d(b) | b \in \mathbf{a}\} + \max\{r_d(b) | b \in \mathbf{a}\}. \tag{1}$$

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

The total number of subframes of A to be transferred is

$$B = \prod_{d \in \{x,y\}} \left\lceil \frac{\mathcal{L}_d + \mathrm{pad}_d(\mathbf{a})}{L_d - \sum_{b \in \mathbf{a}} (l_d(b) + r_d(b))} \right\rceil . \tag{2}$$

The ceiling operator takes care of image sizes which are not exact multiples of the array's size. The resultant image has the same point set as A. Any extra computed values are ignored. Notice that each neighborhood or template operation decreases the size of the subframe's valid region.

Building an expression tree for a sequence of template and neighborhood operations requires four state variables per dimension, the sum and maximum of the l_d and r_d values. The method for keeping track of the valid region's size is clear. Adding another operation to an existing expression tree needs only the current

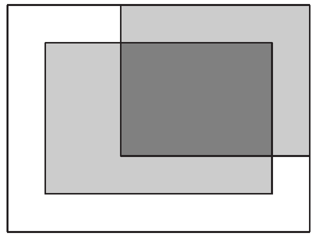


Figure 6: The valid region of image-image operations is the intersection of the operands' valid regions.

```
1       IA_Set< IA_Point<int> > domain =
2         IA_boxy_pset (IA_Point<int> (0, 0),
3                       IA_Point<int> (255, 255));
4
5       // Images A and B are to use the SIMD array, C and D
6       // will inherit that property...
7       IA_Image< IA_Set< IA_Point<int> >, int >
8         A = IA_PIMImage
9               (IA_Image< IA_Point<int>, int > (domain, 1)),
10        B = IA_PIMImage
11              (IA_Image< IA_Point<int>, int > (domain, 2)),
12        C, D;
13
14      C = A + B;
15      D = C * A;
16      cout << sum (D);
```

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.

5.1.2 Transfer Time

The final cost of transferring the images depends on both the number of subframes and the time to transfer each subframe. Blocks are transferred for both operand and result images.

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

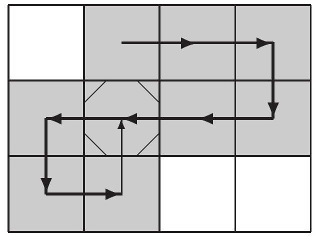


Figure 8: This convolution path involves nine shift operations, eight for calculations and one for returning to the origin.

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_{in}, and the total transfer time out of the array, T_{out}, are

$$T_{\mathrm{in}} = B\left(\mathrm{setup} + \sum_{\substack{A\in\{\mathrm{operands}\\ \mathrm{and\ masks}\}}} T(\mathrm{rep\_size}(A))\right), \text{and } T_{\mathrm{out}} = B\left(\mathrm{setup} + \sum_{A\in\{\mathrm{results}\}} T(\mathrm{rep\_size}(A))\right), \quad (3)$$

where rep_size(A) is the size of the machine representation of A's values, and $T(s)$ is the time needed to load 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_{out} term determined during creation is an upper bound. Updating the T_{out} values and propagating the new values up the tree is similar to the shared-subexpression problem mentioned later.

5.2 Cost of Computations

The cost of computation may be small compared to the cost of transferring the images, yet it must be considered as a portion of the total cost. With future PAL or other SIMD systems, these costs may be much closer. Each operation has costs associated with each step of the computation. In this context, consider the same mathematical operation on different representation sizes to be different operations.

5.2.1 Image-Template Operations

For general image-template products, $A \textcircled{\gamma} t$, a template is treated as a sequence of points in the support and associated values. Assume the support of all templates and neighborhoods fit on the array. The ordering of the sequence is given by the convolution paths. The first operation also initializes the accumulator image. The computation proceeds by shifting a pixel value to the target point, combining the entire subframe with each template point's value, and reducing the intermediate result into the result subframe.

The per-subframe cost of computing the general template product is the sum of the time to shift each pixel value to the target point and the time to combine and reduce at the target point. This process is illustrated in figure 8. Other convolution paths are possible. If there is an extra area of processor memory available, the path can be split into two separate paths starting from different template locations. The cost of moving processing element memory into a buffer negates the savings from a few shifts, so this optimization is only useful when temporary space already exists.

The cost C for computing an image-template product $A \textcircled{\gamma} t$ with B subframes is

$$C = B\left(|t|\left(\mathrm{op\_time}(\circ) + \mathrm{op\_time}(\gamma)\right) + \mathrm{op\_time}(\mathrm{shift}) \cdot |\mathrm{path}|\right). \tag{4}$$

Section 5.1.1 mentions the need for a masking image with certain image-template operations. The masking adds to the per-template-point operation time.

5.2.2 Neighborhood Operations

Neighborhood operations are simply local reductions. They are computed as templates without the combination operation. The cost C for a neighborhood product $A \textcircled{\Gamma} n$ with B subframes is

$$C = B\left(|n|\,\mathrm{op\_time}(\gamma) + \mathrm{op\_time}(\mathrm{shift}) \cdot |\mathrm{path}|\right). \tag{5}$$

5.2.3 Unary, Image-Scalar, Image-Image operations

Once the necessary data resides on the processor array, unary, image-scalar, and image-image operations are calculated through a single operation per block. The cost C for unary operations $(\mathrm{op}\, A)$, image-scalar operations $(s \,\mathrm{op}\, A)$, and image-image operations $(A_1 \,\mathrm{op}\, A_2)$ is

$$C = B \cdot \mathrm{op\_time}(\mathrm{op}) \tag{6}$$

5.2.4 Global Reductions

Global reductions produce a scalar result. Most languages, including C++, have no capacity for delaying the computation and storage of their basic types. Thus, all global reductions in the `iac++` system must be evaluated immediately. The cost is immaterial for that purpose. Note that some common, important operations such as equality testing are essentially global reductions.

If there were a method for delaying scalar results, global reductions could be considered neighborhood reductions over an image's entire support. This breaks the assumption that supports are smaller than the processor array, requiring decomposition of the support.

5.2.5 Functional Composition

Compositions between functions and images are currently not supported on the PAL machine. Extending support to general functions would place more restrictions on the source code. General functions will contain many sequential operations on types native to the client. With a stock C++ system, these operations cannot be supported transparently.

For instance, composing an integer-valued image with a function that adds one would require parallelizing the application of the function. If the function worked with the standard `int` type, a stock compiler would produce standard sequential code. The library would need a way to execute that specific code on each PE. No such facility exists for the PAL, which has different opcodes and data sizes than the host processor.

6 Scheduling Evaluation of Trees

The costs discussed so far are accumulated as an expression tree is built. At some point, the delayed operations must be evaluated. The system only knows the history of the computation so far, not the entire span of computation. A new operation is potentially the optimum place to stop building the tree, so some heuristic must guide the system from the incomplete information available.

One simple heuristic is to examine the average cost per operation in a tree. As long as adding a new operation decreases the average cost, it can be delayed. If the new operation increases the average cost, the tree should be evaluated before the new operation is added.

Template and neighborhood operations decrease the valid region and initially increase the number of blocks to be transferred. Multiple operations can be performed per block, however, obviating the need for intermediate, temporary results. The two effects are balanced according to the time constants on a particular system. Unary and image-scalar operations will always decrease the average cost, so they will never trigger an evaluation.

An image-image operation can increase the average cost by reducing the valid region. If it does, only one of the two subtrees needs to be evaluated. One possible choice is to evaluate the subtree with the smallest valid region. The smaller region cannot be delayed for many more image-template products, so this is a good choice when those operations predominate. Evaluating the subtree with the lower average cost is also attractive. The higher cost is likely to be decreased further than the lower. The lower average cost might correspond to the larger valid region, however. A sampling of real examples will be necessary to determine which is more effective.

Other possible heuristics include hysteresis extensions on the average cost to support small bumps or waiting until a result is required or the valid region is completely destroyed before partitioning the computation. The average-cost heuristic has the advantages of being somewhat simple and relatively intuitive.

7 Possible Improvements

Some of the assumptions made so far are not always suitable for real systems. Templates and neighborhoods may be larger than the processor array. One delayed computation may be shared by multiple expression trees. A mostly transparent system cannot detect these and modify the source program; it must deal with them as they occur.

The processor array and the sequential host only have finite resources to devote to temporary results, as well. Most hosts have limited shared memory for transferring images, and most processor arrays have limited on-board memory. Some algorithms use very little space on the host but need many temporaries on the processor array. The resource needs of others are reversed. Using nodes from a pre-allocated node pool controls the resource use on both ends, but particular allocation strategies may be sub-optimal for certain algorithms.

7.1 Weak Template Decomposition

Not all templates and neighborhoods will fit within a single subframe. The supports need to be decomposed, and the decomposed pieces need to be calculated separately and reduced together [30]. Image algebra already requires the reduction operator to be associative, so no ordering problems arise.

If template t decomposes into templates t_1 and t_2, the template product becomes

$$A \circledgamma t \Rightarrow (A \circledgamma t_1) \, \gamma \, \mathrm{translate}(A \circledgamma t_2'). \tag{7}$$

The translate function is a general notation for lining up the appropriate subframes, and t_2' is t_2 with the origin relocated to lie within the same subframe.

If the subexpressions $A \circledgamma t_1$ and $A \circledgamma t_2'$ can be broken into the same blocks, the calculation can be pipelined up to the number of temporary stores available in the array. Subframes could be loaded as needed and used multiple times before being replaced.

```
// Say A, B, C, and D are defined as before,
// and t1 and t2 are simple, small templates
// of the appropriate type.

C = linear_product (A, t1) + B;
D = C * linear_product (A, t2);
cout << sum (C);
```

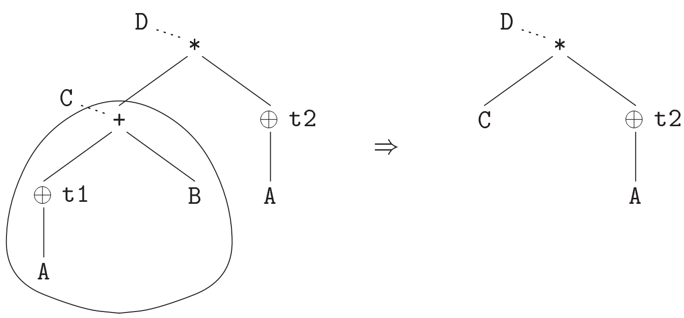


Figure 9: The global reduction of `C` will invalidate the state information for the tree rooted at `D`.

7.2 Shared Subexpressions

Shared subexpressions seem to offer potential for further optimizations. However, they destroy the compositional nature of the cost function. Expressions are no longer disjoint; they join to form a directed, acyclic graph. We will loosely refer to the expression graphs as trees for the remainder of the paper.

The evaluation of any subtree in an expression will invalidate the state information carried at the root of the tree. Because subtrees can be shared between trees, evaluation of one expression's tree can trigger evaluation of a subtree within another expression's tree. Figure 9 shows how even a simple sequence of operations can invalidate state information.

This is similar to the earlier problem of overestimating the number of intermediate results. The current solutions to both problems are to ignore them. The cost function composition method provides an upper bound. One extension which could tighten the bound would be to keep track of a limited number of parents per node and update their state information. Future tests will show how many updates will be both useful and feasible for practical algorithms.

Subexpressions shared only within one tree do not suffer from early evaluation. They could provide extra loading optimizations, as seen in the support decomposition discussion.

7.3 Finer Optimizations

The cost estimate is an upper bound. Many fine-tuning optimizations exist beyond the general framework presented here. Some, such as always loading array-sized blocks of the image and shifting unused pieces into the position for the next block, may save more time at considerable programming expense. Others, such as tracking the origin of result blocks and shifting them to agree rather than always returning the result to its origin, are of marginal savings and little programming expense. Any implementation must find an acceptable balance between the different aspects of efficiency [31].

8 Examples

Any cost model/heuristic pair must be evaluated in the context of numerous examples. Here we present two fairly simple ones, a Roberts edge detector and a 3-level wavelet transform. We are taking the simple cost

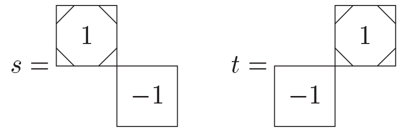


Figure 10: The templates s and t for the Roberts edge detector.

model proposed above and combining it with a heuristic that dictates evaluation of a stored tree when adding a new operation would increase the average cost per operation. We also assume no shared subexpressions, so the implementation cost of this heuristic is very low. These algorithms are to run on a single board PAL system, so the array dimensions are 72×64.

8.1 Roberts Edge Detector

The Roberts edge detector [27] consists of a simple, typical expression. The edge image E is derived from an input image A by $e = \sqrt{(A \oplus s)^2 + (A \oplus t)^2}$, where figure 10 displays the templates s and t. The table in figure 11 shows how the estimated cost per operation Q varies as the tree grows when applying the algorithm to a 1024×1024 image. Because the reduction identity, zero, is preserved by the combination operator, multiplication, there are no mask images in this example. The loading times, τ_i, and computation times, c_i, are per block. The number of operations, n_i, is tallied as if there are no shared subtrees. For example, n_6 is $n_5 + n_3 + 1$, ignoring the fact that the subtrees share their initial operation. The approximate average cost per operation at step i is then $Q_i = B_i(2\tau_i + c_i)/n_i$.

The costs are approximately correct for a PAL-I IOC/SA system. Two multiplications, two additions, and two shifts comprise the 2.37×10^{-5} seconds for each template. The extra 8.00×10^{-5} seconds included in the computation time of step 6 is the time required to move the result of step 3 into each PE's memory before continuing. The 2.40×10^{-3} seconds for loading a block assumes a throughput of around 8 MB/S to the PAL unit.

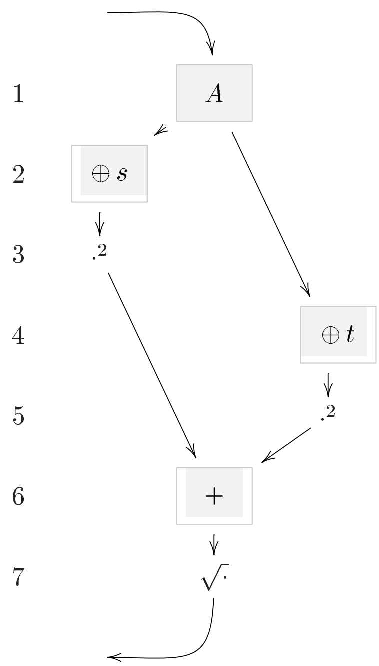
Note that both the time to load a block and the number of operations are overestimated in step 6 of figure 11. Each subtree is conservatively assumed not to share nodes. The actual average cost per operation is .210 seconds. Each node could maintain a dependency list and image-image operations could determine their costs more intelligently, but this imposes an extremely large overhead on computations that form long chains.

Another important observation is that the cost of communication outweighs the cost of computation for a small expression by nearly two orders of magnitude. If every expression in an application is fairly small, as when each tree consists of less than 100 operations, updating the computational cost is needless overhead.

One of the major factors attributing to the severe imbalance between computation and communication costs is the repeated transmission of data in the overlap regions. A simple cache hierarchy, like the 64 MB frame buffer in the forthcoming IOC/FB, will let the system transfer the whole image once. This saves time not only by sending each value once but also by better using the link's available bandwidth. The total time to load will then be a function of only the image size plus a per-block constant on the order of 5×10^{-5} seconds.

8.2 Three-Level Wavelet Transform

Next, examine a three-level wavelet transform on a 512×512 image A. Assume the base wavelet filters have six taps like Daubechies' $\mathcal{W}_6$ wavelet [32]. The supports of the high-pass filters h_i and low-pass filters g_i are shown in figure 12. Here we ignore the negligible cost of computation and focus on the number of blocks to be transmitted between the array and the host. Also, we only apply the transform along the x-dimension for simplicity.



	B	n	τ	c	Q
1	15×16	1	240	0	1.15
2	15×17	2	τ_1	$c_1 + 2.37$	.615
3		3	τ_2	$c_2 + 1.02$	.411
4	15×17	2	τ_1	$c_1 + 2.37$	.615
5		3	τ_5	$c_4 + 1.02$	.411
6	15×17	7	$\tau_5 + \tau_3$	$c_5 + c_3$ $+8.00$	.355
7		8	τ_6	$c_6 + 2.88$	.311

Figure 11: The cost Q associated with the expression tree for $\sqrt{(A \oplus s)^2 + (A \oplus t)^2}$ strictly decreases. The times for τ and c are in units of 10^{-5} seconds, and the time for Q is in seconds.

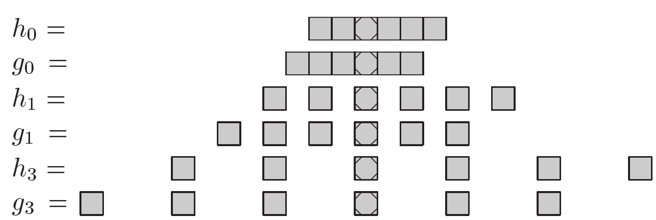
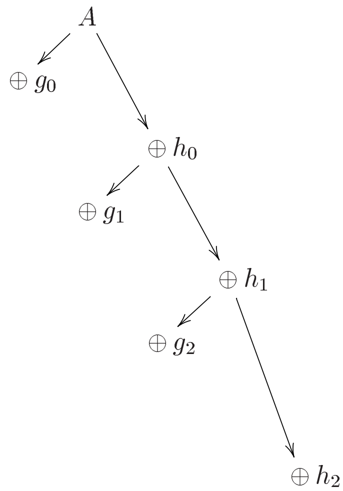


Figure 12: The high-pass (h_i) and low-pass (g_i) filters sample from successively farther locations. The shaded boxes form the non-zero support of each template.

```
S = IA_PIMImage (A);
for (i = 0; i < 4; i++)
  D[i] = linear_product (S, g[i]);
  S = linear_product (S, h[i]);

```

Figure 13: A short, simple wavelet decomposition along one dimension



	$\max l_x$	$\max r_x$	$\sum l_x$	$\sum r_x$	B	n	B/n
1	0	0	0	0	8×8	1	64
2	3	2	3	2	8×8	2	32
3	2	3	2	3	8×8	2	32
4	6	4	8	7	10×8	3	26.6
5	4	6	6	9	10×8	3	26.6
6	12	8	18	17	15×8	4	30
	12	8	12	8	11×8	2	44
7	8	12	8	12	11×8	2	44

Figure 14: Assuming the transfer cost overwhelms the computation cost, this expression tree's cost per operation will increase after step 6, forcing evaluation of $A \oplus h_0 \oplus h_1$.

Figure 13 holds the basic wavelet decomposition code. The arrays hold the similarly-named filters for each level i. 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 $\oplus g_2$ operation to the tree, the total number of blocks transferred per operation increases. The simple heuristic we use indicates that the delayed expression $A \oplus h_0 \oplus h_1$ 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 transfered actually increases more when $A \oplus h_0 \oplus h_1$ 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 challanges 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 PAL computer. We have shown how the operations of image algebra affect both the cost of computation of results on the PAL and the cost of communication of data to the PAL. 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 algorithms. Such a study requires specifications of algorithms in image algebra, `iac++` implementations of those algorithms, and a statistic-collecting PIM server. Catalogs like [27] provide a useful starting point for the former, and most of the algorithms in it have already been implemented. Only minor modifications are necessary to begin a study, but we are still determining which parameters to examine.

We are also building a real system with this general, cost-based model. In the implementation, expression node objects contain cost objects through an aggregation system. The cost object classes share a common interface to the node objects, but each class can maintain a different internal interface. The cost of a new node is determined by its cost object's examination of the childrens' cost objects. Thus different cost objects can be used within different trees. Initially, we assume that each client can select a single cost class provided by the server. We may find it useful to allow clients to have multiple contexts; expressions within one context may use a different cost class from expressions within another context. Moving an image to a different context may be implemented by fully evaluating the image on a context switch or by providing a common, inter-cost-class interface to allow the classes to determine if they are compatible.

Acknowledgements

This work was sponsored in part by the PAL Consortium, funded by the Wright Laboratory Armament Directorate at Eglin Air Force Base, Lockheed Martin Electronics and Missiles in Orlando, the University of Florida, and the University of Missouri-Columbia. We would like to thank the consortium members for their efforts in support of this and other related work.

References

[1] Behrooz Parhami. SIMD machines: Do they have a significant future? In *Report on a Panel Discussion at Frontiers'95: The Fifth Symposium on the Frontiers of Massively Parallel Computation, McLean, VA*. IEEE, Feburary 1995.

[2] B. H. McCormick. The Illinois pattern recognition computer ILLIAC III. *IEEE Transactions on Electronic Computers*, 12:791 813, 1963.

[3] M. J. B. Duff, D. M. Watson, T. J. Fountain, and G. K. Shaw. A cellular logic array for image processing,. *Pattern Recognition*, 5(3):229 247, September 1973.

[4] M. J. B. Duff. Clip4. In K. S. Fu and T. Ichikawa, editors, *Special Computer Architectures for Pattern Processing*, chapter 4, pages 65 86. CRC Press, Boca Raton, FL, 1982.

[5] T. J. Fountain, K. N. Matthews, and M. J. B. Duff. The CLIP7A image processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(3):310 319, 1988.

[6] S. Reddaway. The DAP approach. In C. Jesshope and R. Hockney, editors, *Infotech State of the Art Report: Supercomputers, vol. 1 & 2*, pages 309 329. Maidenhead: Infotech Int. Ltd., 1979.

[7] Kenneth E. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, C-29(9):836 840, September 1980.

[8] E. L. Cloud. Geometric arithmetic parallel processor: Architecture and implementation. In V. K. Prasanna, editor, *Parallel Architectures and Algorithms for Image Understanding*, pages 279 305. Academic Press, San Diego, CA, 1991.

[9] M. S. Tomassi and R. D. Jackson. An evolving SIMD architecture approach for a changing image processing environment. *DSP & Multimedia Technology*, pages 1 7, October 1994.

[10] M. J. Little and J. Grinberg. The 3-D computer: An integrated stack of WSI wafers. In E. E. Swartzlander, editor, *Wafer Scale Integration*. Kluwer Academic Publishers, Boston, MA, 1989.

[11] J. A. Webb. Steps toward architecture-independent image processing. *Computer*, pages 21 31, February 1992.

[12] R. Davoli, L. A. Giachini, O. Babaoglu, A. Amoroso, and L. Alvisi. Parallel computing in networks of workstations with Paralex. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):371 384, April 1996.

[13] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21:4 14, 1994.

[14] J. Rose and G. Steele. C*: An extended C language for data parallel programming. Technical report, Thinking Machines Corporation, 1987.

[15] J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener. *Fortran 90 Handbook*. McGraw Hill, New York, NY, 1992.

[16] H. Shi, G. X. Ritter, and J. N. Wilson. Parallel image processing with image algebra on SIMD mesh-connected computers. In P. W. Hawkes, editor, *Advances in Imaging and Electron Physics*, volume 90, chapter 4. Academic Press, San Diego, CA, 1995.

[17] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, London, 1982.

[18] S. R. Sternberg. An overview of image algebra and related architectures. In S. Levialdi, editor, *Integrated Technology for Parallel Image Processing, (Polignano, Italy, June 1 3, 1983)*, pages 79 100, London, 1985. Academic Press.

[19] P. Maragos. *A Unified Theory of Translation-Invariant Systems with Applications to Morphological Analysis and Coding of Images*. PhD thesis, Georgia Institute of Technology, 1985.

[20] E. R. Dougherty and C. R. Giardina. Image algebra - induced operators and induced subalgebras. In *SPIE Proceedings of Visual Communications and Image Processing II*, volume 845, pages 270 275, Cambridge, MA, October 1987.

[21] G.X. Ritter, J.N. Wilson, and J.L. Davidson. Image algebra: An overview. *Computer Vision, Graphics, and Image Processing*, 49(3):297 331, March 1990.

[22] G. X. Ritter. Recent developments in image algebra. In P. Hawkes, editor, *Advances in Electronics and Electron Physics*, volume 80. Academic Press, New York, NY, 1991.

[23] J.N. Wilson. Supporting image algebra in the C++ language. In *Image Algebra and Morphological Image Processing IV*, volume 2030, pages 315 326, San Diego, CA, July 1993.

[24] Engineering Design Team, Inc. *SBus Configurable DMA Interface User's Guide*, document #00-00419-04 edition, 1997.

[25] S. Sahni and V. Thanvantri. Parallel computing: Metrics and models. In preparation.

[26] Grady Booch and James Rumbaugh. *Unified Method for Object-Oriented Development Version 0.8*. Rational Software Corporation, 1995.

[27] G.X. Ritter and J.N. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, Boca Raton, FL, 1996.

[28] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990, 1990.

[29] R. J. M. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98 107, April 1989.

[30] G.X. Ritter. Image algebra with applications. Unpublished manuscript, available via anonymous ftp from `ftp://ftp.cise.ufl.edu/pub/src/ia/documents/ia.*.ps.gz`, 1994.

[31] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl, 2nd. Edition*, pages 537 546. O'Reilly & Associates, Inc., 1996.

[32] I. Daubechies. *Ten Lectures on Wavelets.* CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, Philadelphia, PA, 1992.