Efficient SIMD evaluation of image processing programs

J. N. Wilson and E. J. Riedy

University of Florida CISE Department
Gainesville, FL 32612-6120 USA
jnw@cise.ufl.edu ejr@cise.ufl.edu

ABSTRACT

SIMD parallel systems have been employed for image processing and computer vision applications since their inception. This paper describes a system in which parallel programs are implemented using a machine-independent, retargetable object library that provides SIMD execution on the Lockheed Martin PAL-I SIMD parallel processor. Programs' performance on this machine is improved through on-the-fly execution analysis and scheduling. We describe the relevant elements of the system structure, the general scheme for execution analysis, and the current cost model for scheduling.

Keywords: SIMD, image algebra

1. INTRODUCTION

Single instruction stream, multiple data stream (SIMD) parallelism has been employed for computer-vision related activities since the development of the Goddard Massively Parallel Processor (MPP).[1] Various approaches have been used to implement application programs on SIMD machines, ranging from the use of microcode to special-purpose languages[2] to special implementations of general-purpose languages.[3] In this paper, we discuss the use of the retargetable Image Algebra C++ object library `iac++`[4] designed to support image-processing and computer-vision applications. This library has been targeted to the Lockheed Martin PAL-I processor, a fine-grained, SIMD-parallel processor.

To modify the `iac++` library to provide efficient SIMD execution on the PAL system required 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 operations and operands provided by the `iac++` library and the way its software architecture supports retargeting to the PAL is described in section 3. Section 4 describes the cost function used to direct the evaluation of programs on the SIMD array to reduce the required processing time. The heuristics applied to optimize the cost function are described in section 5. We conclude with a discussion of limitations of our current cost function and evaluation strategy.

2. THE PAL-I SIMD PROCESSING SYSTEM

The PAL-I, IOC/SA system is a workstation-based image-processing development environment. The PAL processor itself is an attached image processing accelerator. The system's current workstation platform is a Sun SparcStation 4. The PAL processor is attached to such a workstation with an EDT SCD-40 configurable DMA interface.[5] This provides a nominal 40 megabyte per second (MB/s) connection between the host workstation and the PAL processor. The PAL processor itself 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 64×72 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. And because the processors are connected in a two-dimensional mesh network, performance of this system on local neighborhood operations can far outstrip any sequential processor.

Efficiently programming such a SIMD processor system presents challenges. The sustained throughput of the SCD-40 card is, in fact somewhat less that 10 MB/s. Thus, transfer of a 9,216 pixel image with 8-bit element 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 system. 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, however, 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.

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 machine as necessary to carry out the specified operations.

3. RETARGETING THE IMAGE ALGEBRA C++ 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.

3.1. Operands and operations of the `iac++` Library

The operands of image algebra are drawn from the image algebra developed at the University of Florida by Ritter *et al.*[6] 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.[7] 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 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 is provided, `IA_Image<IA_Point<double>, float>`.

4. 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. 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 the value types `bool`, `unsigned char`, `int`, `float`, and `IA_Complex` are supported by the library.

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. Retargeting the `iac++` library to the PAL

The retargeting of the library is supported by the separation of user interface from representation employed in the class hierarchy. The image classes are all instances of a single C++ template class, `IA_Image<P,T>`. These classes provides 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 ensurea that the behavior of any image in response to user-issued operations is dependent upon its representation class. To retarget 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.

This class structure is shown in figure 1. In this figure, we use the notation of the Unified Method of Booch and Rumbaugh.[8] 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 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 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. The full client-server arrangement is depicted in figure 2.

3.3. Evaluation of image operations on a small SIMD array

If we are to use a SIMD processor 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

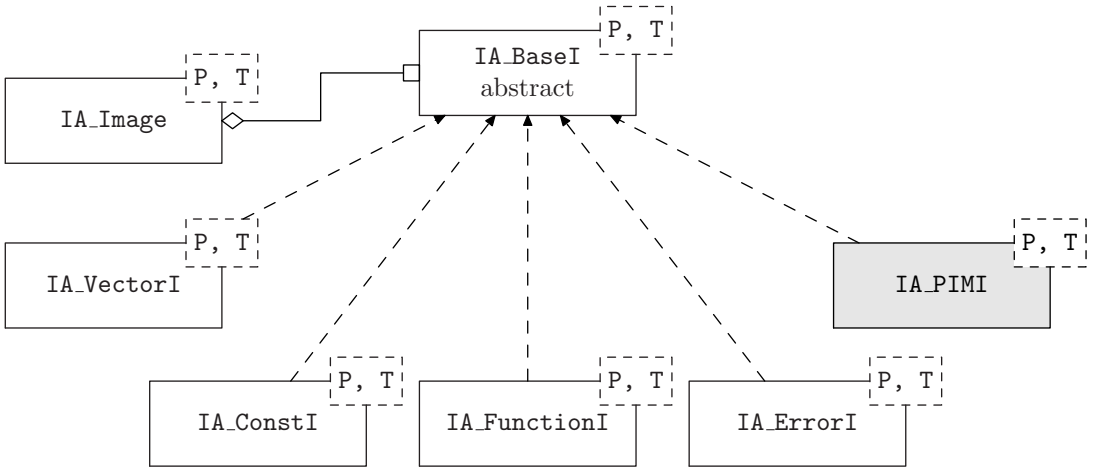

Figure 1. The image classes in the `iac++` library are specialized for efficiency.

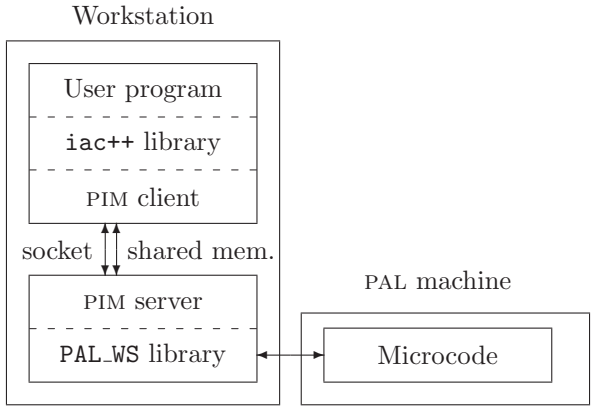

Figure 2. The user's program uses the PIM client through the `iac++` library to talk to the PIM server. The server in turn uses the `PAL_WS` library to communicate with the PAL device.

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 image 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 numerous 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[9] of image operations. When a client program operates upon images, the server construct an expression tree that represents the computation to be performed. When the client program attempts to use 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

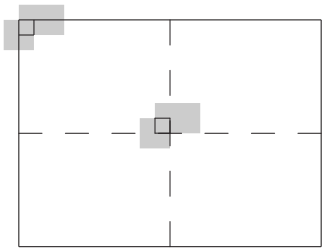

Figure 3. Blindly applying templates leads to boundary effects along both image (solid) and array-induced (dashed) borders.

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 achieved 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.

4. COST MODEL

We discuss 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 execution time is to be minimized, the cost is expressed in terms of time.

4.1. 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 cost of transferring a block.

4.1.1. 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. Figure 3 shows how these effects also occur along the induced, internal borders.

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 which 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

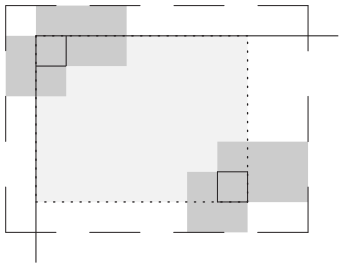

Figure 4. Each subframe (dashed) yields a smaller result (dotted).

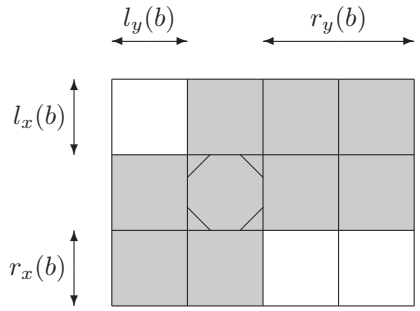

Figure 5. The functions l_d and r_d denote the extent of a template's bounding box on either side of the its origin.

preserved by the combination operator. For example, combining a padding value of 255, the 8-bit, unsigned identity for min, 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 transfered 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 processing array. The results below assume a mask is transfered 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 processing array, the number of block boundaries along 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 processing 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 $d \in \{x, y\}$.

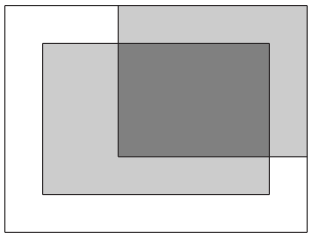

Figure 6. The valid region of image-image operations is the intersection of the operands' valid regions.

Define the function pad_d on sequences of bounding boxes to be

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

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

The total number of subframes of A to be transferred is

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

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 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.

4.1.2. 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 transfered back to the host. Had line 14 been written `D = (A + B) * A`, the temporary result would be out of scope by line 15, 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 significant limitation of the implemented system, one which does 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

```
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`.

of the array, T_{out}, are

$$T_{\text{in}} = B\left(\text{setup} + \sum_{\substack{A\in\{\text{operands}\\ \text{and masks}\}}} T(\text{rep-size}(A))\right), \text{and} \tag{3}$$

$$T_{\text{out}} = B\left(\text{setup} + \sum_{A\in\{\text{results}\}} T(\text{rep-size}(A))\right), \tag{4}$$

where rep-size(A) is the size of the machine representation of As values, and $T(s)$ is the time needed to load block of representation size s onto the processing 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.

4.2. 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, the 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.

4.2.1. Image-template operations

For general image-template products, $A\textcircled{\gamma}t$, treat a template 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 the convolution path. An accumulator image is initialized to the reduction operator γ's identity. The computation

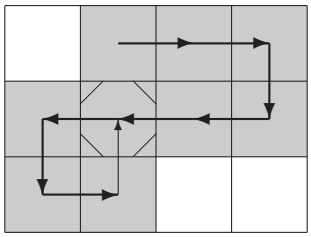

Figure 8. This convolution path involves nine shift operations, eight for calculations and one for returning to the origin.

proceeds by combining the entire subframe with each template point's value, reducing the intermediate result into the result subframe, and shifting the result to the next point.

The per-subframe cost of computing the general template product is the sum of the time to combine and reduce at each template point and the time to shift the result to the next. At present, the shifting not only positions the images for the current computation but also returns the result image to its original location, shown 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

$$C = B \left(|t| \left(\text{op-time}(\circ) + \text{op-time}(\gamma) \right) + \text{op-time}(\text{shift}) \cdot |\text{path}| \right). \tag{5}$$

Section 4.1.1 mentions the need for a masking image with certain image-template operations. The masking adds to the per-template-point operation time.

4.2.2. 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

$$C = B \left(|n| \, \text{op-time}(\gamma) + \text{op-time}(\text{shift}) \cdot |\text{path}| \right). \tag{6}$$

4.2.3. Unary, image-scalar, image-image operations

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

$$C = B \cdot \text{op-time}(\text{op}). \tag{7}$$

4.2.4. 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 processing array, requiring decomposition of the support.

4.2.5. Functional composition

Compositions between functions and images are not supported currently on the PAL. 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 processing element. No such facility exists for the PAL, which has different opcodes and data sizes than the host processor.

5. 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.

6. LIMITATIONS

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

The processing 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 processing arrays have limited on-board memory. Some algorithms use very little space on the host but need many temporaries on the processing array. The resource needs of others are reversed. Using nodes from a pre-allocated node pool controls the resource use on both ends.

6.1. 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.[10] The image algebra already requires the reduction operator to be associative, so no ordering problems arise.

```
1    // Say A, B, C, and D are defined as before,
2    // and t1 and t2 are simple, small templates
3    // of the appropriate type.
4
5    C = linear_product(A, t1) + B;
6    D = C * linear_product(A, t2);
7    cout << sum(C);
```

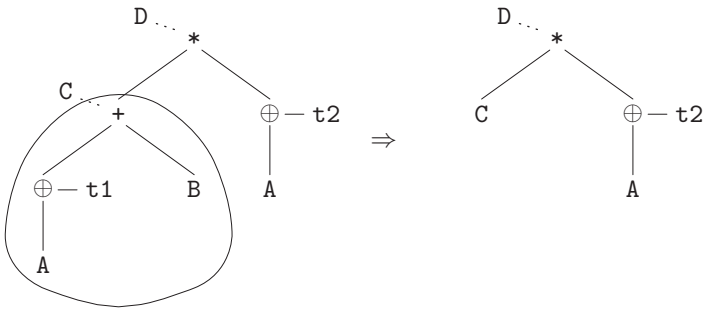

Figure 9. The global reduction of `C` will invalidate the state information for the tree rooted at `D`.

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

$$A \textcircled{\gamma} t \Rightarrow (A \textcircled{\gamma} t_1)\, \gamma\, \text{translate}(A \textcircled{\gamma} t_2'). \tag{8}$$

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

If the subexpressions $A \textcircled{\gamma} t_1$ and $A \textcircled{\gamma} t_2'$ 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.

6.2. Shared subexpressions

Shared subexpressions seem to offer potential for further optimizations. However, they can destroy the compositional nature of the cost function. Expressions are no longer disjoint; they join to form a directed, acyclic graph.

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.

6.3. 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 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.[11]

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, FL, and the University of Florida. We would like to thank the consortium members for their efforts in support of this and other, related work. Dr. Hongchi Shi of the University of Missouri-Columbia has assisted in editing.

references

1. K. E. Batcher, "Design of a massively parallel processor," *IEEE Transactions on Computers* **C-29**, pp. 836–840, Sept. 1980.
2. J. Rose and G. Steele, "C*: An extended C language for data parallel programming," tech. rep., Thinking Machines Corporation, 1987.
3. J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener, *Fortran 90 Handbook*, McGraw–Hill, New York, NY, 1992.
4. J. Wilson, "Supporting image algebra in the C++ language," in *Image Algebra and Morphological Image Processing IV*, *Proceedings of* SPIE **2030**, pp. 315–326, (San Diego, CA), July 1993.
5. Engineering Design Team, Inc., *SBus Configurable DMA Interface User's Guide*, document #00-00419-04 ed., 1997.
6. G. Ritter and J. Wilson, *Handbook of Computer Vision Algorithms in Image Algebra*, CRC Press, Boca Raton, FL, 1996.
7. M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison–Wesley, 1990, 1990.
8. G. Booch and J. Rumbaugh, *Unified Method for Object-Oriented Development Version 0.8*, Rational Software Corporation, 1995.
9. R. J. M. Hughes, "Why functional programming matters," *The Computer Journal* **32**, pp. 98–107, Apr. 1989.
10. G. 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.
11. L. Wall, T. Christiansen, and R. L. Schwartz, *Programming Perl, 2nd. Edition*, pp. 537–546. O'Reilly & Associates, Inc., 1996.