

Exposing Data-Level Parallelism in Sequential Image Processing Algorithms

Lewis Baumstark, Jr. and Linda Wills

School of Electrical and Computer Engineering, Georgia Institute of Technology
{ lewisb, linda.wills }@ece.gatech.edu

Abstract

As new computer architectures are developed to exploit large-scale data-level parallelism, techniques are needed to retarget legacy sequential code to these platforms. Sequential programming languages force programmers to include sequential artifacts in their code, particularly with respect to how the source code expresses data references (generally assuming a linear address space). In contrast, data-parallel programs apply many operations in parallel to elements in two-dimensional data sets, and a given data parallel operation can access other spatially local elements along either dimension. Of key importance in exposing data parallelism is determining these two-dimensional data dependencies among elements of a matrix. This paper presents a reverse engineering technique for identifying such dependencies in sequential image processing code, using pattern matching on an attributed dataflow representation of the program. The technique is applied to common image filtering algorithms. The technique is validated by retargeting to a Matlab program and matching the results against those of the original source.

1. Introduction

An enormous body of image and video processing software has been written for conventional (sequential) desktop computers and embedded digital signal processors (DSPs). These implement a wide range of operations, such as convolution, discrete cosine transform (DCT), and motion estimation [1]. These applications usually have a tremendous potential for parallelism in that they include a large percentage of independent operations that are applied to each pixel or region of a large, two-dimensional image array. This type of parallelism is called “data-level parallelism” (DLP) because the same operation can be applied simultaneously to multiple pieces of data (e.g., applying a thresholding function to each pixel in an image to convert it from gray scale to black and white). However,

the sequential processors for which these applications are written are poorly equipped for exploiting DLP on a large scale. At best, they exploit parallelism at the instruction level (ILP) by applying multiple (usually different) operations to different pieces of data, for example, using Very-Long Instruction Word (VLIW) execution or superscalar pipelined execution of overlapping independent instructions [2].

Support of data parallelism has motivated the definition and implementation of multimedia instruction set extensions (e.g., Intel’s MMX and SSE [21]). Efficient exploitation of data parallelism on a larger scale has also been demonstrated architecturally in Single-Instruction, Multiple-Data (SIMD) massively parallel architectures (e.g. Maspar MP2 and Thinking Machine’s CM-2).

Unfortunately, existing compilation techniques are not adequate for compiling sequential multimedia programs to such parallel architectures. The key obstacle lies in the programming specification; most image and video processing programs are written in sequential languages, such as C, which force programmers to introduce sequential artifacts into their code. For example, a programmer writing in C for a superscalar architecture may make assumptions about the linear memory layout of an architecture and write code to accommodate that layout. Such code may not easily translate to the distributed, 2D mesh-based layout of data inherent in a SIMD processor. Also, since these languages have no mechanism for specifying explicit parallelism (e.g., vector variables), these applications have been implemented using nested loops. While modern compilers are capable of exposing limited loop parallelism, they target instruction level parallelism (ILP) rather than data parallelism. However, automatic detection of data parallelism offers the potential for a much greater speedup at lower cost in chip area and power than can be gained from ILP alone.

This research focuses on retargeting sequential image and video processing programs to SIMD architectures by extracting the essential computations and ensuring that each computation has the data it requires to

complete. The overall parallelization strategy is to apply multiple partial recognition techniques at multiple levels of abstraction, without requiring a complete understanding of the program as a single well-known algorithm or image processing operation. This paper focuses on a subclass of image processing algorithms, known as filtering operations, because they raise important problems in the retargeting of data reference patterns. Filtering operations are common image processing operations which compute a value for each pixel based on some function of a small neighborhood of surrounding pixels.

A particularly difficult challenge faced in retargeting filtering algorithms is that typically, image data is represented in linear memory organizations (e.g., row-major or column-major form) so that neighboring pixels in the image are not necessarily stored as neighboring elements in linear memory. The sequential code that processes these pixels contains complex memory address computations to allow access to the pixels. These obscure the spatial relationships between the pixels so that opportunities to efficiently store and access the 2D image data are hidden. If the spatial data dependencies are exposed, it is easier to reimplement them to use uniform two-dimensional data access patterns that are needed for data parallel execution.

A related challenge is that the neighborhoods processed by filtering algorithms usually overlap (i.e., each pixel's neighborhood overlaps with the neighborhoods of the pixels surrounding it). This means that there is a great deal of sharing of intermediate values, making it difficult to create simple vectors of data elements that can be processed by traditional vectorization techniques.

This paper presents a technique for extracting the two-dimensional spatial data dependencies from C image filtering source code. A key insight gained by looking at the image filtering programs is that extracting these spatial data dependencies is the critical and most difficult step; often, the core filtering computation that is applied to each neighborhood of pixels can be directly transferred over to the data parallel code unchanged. Based on this insight, our strategy is to first focus on identifying two-dimensional data reference patterns in the source code (the subject of this paper) and later apply different analysis techniques (as needed) to the core filtering computation.

This paper applies this reverse engineering technique to image filtering code from a commercial library originally written for the Texas Instruments TMS320C62xx family of digital signal processors [19]. To validate the results, it retargets to a Matlab script and confirms that the results match those of the original C source code.

The rest of this introductory section discusses related work. Section 2 outlines the class of algorithms (image filtering operations) under analysis, as well as describing the data parallel architectures that are targeted. The analysis process is described in Section 3, along with an example image filtering program to illustrate the process. Section 4 presents experimental validation, and Section 5 describes future research in this area.

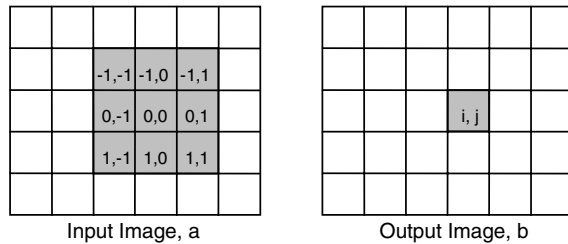
1.1 Related Work

Traditional parallelization work has focused on loop transformations usually targeted at vectorization of arrays in scientific applications (e.g., [3]-[10]). The primary goal has been to achieve performance increases by identifying what data can be piped through composed operations in long one-dimensional vectors. In contrast, the data parallelism inherent in image and video processing applications manifests itself in operations that apply concurrently to all pixels or overlapping regions of pixels in an image. To exploit this parallelism efficiently (not just in performance, but also in area- and power-efficiency), it is essential that the spatial and temporal adjacency of data in the image be preserved.

More recently, projects such as PAP and PARAMAT ([11]-[14]) have employed program recognition to extract program parallelism. The PAP system, targeted at distributed memory architectures, creates a program dependence graph (with additional annotations) representing the program, and then performs pattern-matching to identify groups of structures as concepts. Groups of concepts can then be grouped into higher-level concepts and, eventually, transformed into parallelized code. PARAMAT also performs program recognition, but uses an abstract-syntax tree as its basic program representation. PARAMAT attempts to replace identified sequential algorithms with well-known parallel implementations of those algorithms.

Metzger [15] describes a program understanding system employing a control flow graph (CFG) program representation, with data-flow graphs embedded at each CFG node. It performs hierarchical recognition similar to PAP and PARAMAT.

Like these projects, our work employs pattern matching and transformations over a graph-based program representation as a parallelization technique. The main difference between these projects and our work is that their goal is complete recognition of the program as a single well-known sequential algorithm that can be replaced with a parallel implementation. Our strategy operates at lower levels of abstraction, without requiring a complete recognition of the program as a single well-known operation. This accommodates variability in the overall algorithm while benefiting from the recognition of common building blocks and data reference patterns used in these algorithms.



```

for all pixels do:
b[i][j] = m1*a[i-1][j-1] + m2*a[i-1][j] + m3*a[i-1][j+1]
+ m4*a[i][j-1] + m5*a[i][j] + m6*a[i][j+1]
+ m7*a[i+1][j-1] + m8*a[i+1][j] + m9*a[i+1][j+1]

```

Figure 1. Filtering Example

Further, these projects deal with Fortran programs that have easily identifiable array references, due to the lack of pointers. The techniques described here can be applied to applications written in C (the language most frequently used for DSP and image-processing programs), where the references are often written using pointers. As will be explained, pointer notation can complicate the process of identifying array references.

At least one other project [16] exists that is targeted specifically at finding SIMD/DLP aspects in source code. It uses a syntactic analysis to find parallelizable statements in C/C++ code that can be replaced with macros implementing Intel MMX/SSE [21] instructions. In contrast, the techniques described in this paper are applicable to a broader target architectural space, including focal-plane SIMD architectures [18] and other packed-word SIMD ISA extensions, such as AltiVec [22] and 3DNow! [23].

2. Background

2.1 Filtering Operations

Filtering operations (convolution, median filtering, averaging filter) cover a large body of applications such as image enhancement (e.g., smoothing and noise reduction), image correlation, and edge detection [1][17]. All of these algorithms have a similar structure - each pixel in the output image is some function of a window of pixels (called a *neighborhood*) centered about the corresponding pixel in the input image, as shown in Figure 1.

Figure 2 contrasts a simple sequential convolution algorithm with a data-parallel/SIMD implementation of the same algorithm. (For the SIMD implementation, '@' specifies a data reference in the given direction, e.g., 'b@se' denotes, for each pixel, access the pixel 1 row below and 1 column to the right.) For each pixel in an image, 3x3 convolution multiplies each of the pixel's nine neighbors (which includes itself) with a corresponding element (coefficient) of the 3x3 mask (m1...m9), and sums the products to create the resultant

```

(a) //SEQUENTIAL IMPLEMENTATION – pointer arithmetic
convolution(int * in, int * out, int rows, int cols) {
int * a; int * b;
for (i=1;i<rows-1;i) {
a = in+(i*cols+1); // a points to the next row to process
b = in+(i*cols+1); // b points to the next row to output
for (j=1;j<cols-1;j++) {
nw = *(b-cols-1); n = *(b-cols); ne = *(b-cols+1);
w = *(b-1); h = *(b) ; e = *(b+1);
sw = *(b+cols-1); s = *(b+cols); se = *(b+cols+1);
*a++ = (m1*nw + m2*n + m3*ne + m4*w + m5*h + m6*e
+ m7*sw + m8*s + m9*se) / 16; }
b++;
}
}

```

```

(b) //SEQUENTIAL IMPLEMENTATION – doubly-subscripted
for(i=1;i<rows-1;i++) {
for(j=1;j<cols-1;j++) {
a[i][j] =(m1*b[i-1][j-1] + m2*b[i-1][j] + m3*b[i-1][j+1]
+ m4*b[i][j-1] + m5*b[i][j] + m6*b[i][j+1]
+ m7*b[i+1][j-1] + m8*b[i+1][j] + m9*b[i+1][j+1])/16;
}
}

```

```

(c) //SIMD IMPLEMENTATION
array a[rows][cols];
array b[rows][cols];
// all instances of the following statement are performed in
parallel
a = ( m1*b@nw + m2*b@n + m3*b@ne + m4*b@w + m5*b@
m6*b@e + m7*b@sw + m8*b@s + m9*b@se) / 16;

```

Figure 2. Sequential vs. SIMD Implementations.

pixel. It is used, for example, to perform edge detection by using a cross-gradient mask whose coefficients sum to 0: areas of constant pixel values sum to 0, but changes in intensities at edges result in nonzero values [16].

Notice in Figure 2 that the same core computation is being performed in each implementation. The differences are in where the data is located (and thus how it is accessed), and in that the sequential implementation is iterative and the SIMD implementation is not - because each instance of the "a = ..." statement is independent, a potential for a large decrease in running time exists when data parallel execution is performed.

In practice, syntactical variations and programmer ingenuity can make the actual source code references much more obscure than those shown in Figure 2, particularly with respect to the pointer-arithmetic style.

As shown in Figure 2, if an image is abstractly modeled as a two-dimensional array of values, each pixel in the resultant image is dependent on some set of pixels in the input image. Since our focus here is on filtering operations, which involve neighborhood data-dependencies, the information of immediate interest is the size and shape of the neighborhood, how those values are input into the algorithm, and how each computation is mapped back to its corresponding resultant pixel. In sequential programs, where two-dimensional image data

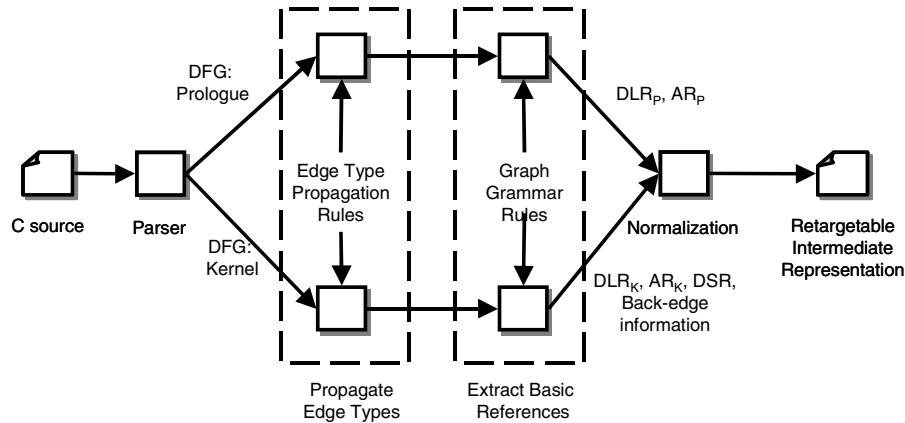


Figure 3. High-level program analysis flow

is usually stored in row-major or column-major form, these 2D data references often take place as relative references to other pixels in the array, either as stride-based offsets from the current (base) position, or as offsets within the array subscripts. For retargeting of the code to be successful, these references must be identified, generalized for all pixels in the image, and then mapped, as inter-processor communication, to each SIMD processor.

2.2 Texas Instruments ‘C6000 Convolution

To avoid dependence on C code contrived to fit the methodology described, sample code was taken from outside sources. Used as a running example throughout this paper is the Texas Instruments C implementation of a 3x3 convolution provided for its ‘C6000 line of DSPs [19]. The source code is provided in the appendix. Of particular interest, this code employs pointer-based (not doubly-subscripted, array-based) data references and, in the inner loop, some address computations are carried over into the next iteration.

As the retargeting approach is described, this example will be used to illustrate the steps. The original example code was changed to undo a loop unrolling optimization, since handling this type of optimization is currently an open issue.

3. Approach

Our high-level analysis approach is shown in Figure 3. It consists of parsing the source code into a dataflow representation (consisting of prologue and kernel parts, described below) which undergoes a pattern-matching process via attribute graph parsing and attribute synthesis to extract two-dimensional array references, and then a combination step to adjust the values returned by the pattern-matcher. Prototype tools have been developed to automate all stages of the process.

3.1 Preconditions and assumptions

For this step of the analysis, the following assumptions are made about common characteristics of image processing algorithms:

- The algorithm is implemented in C using doubly-nested for-loops of the form “for(i=lb;i<=ub-1;i++) { ... for(j=...) { ... } },” where lb and ub are the lower and upper iteration bounds, respectively.
- The outer loop has code before the inner loop (if at all), but not after the inner loop. The outer loop code will be referred to the *prologue*; the inner loop code will be referred to as the *kernel*.
- Each full (i,j) iteration of the algorithm outputs one pixel value.

In addition, the following information is assumed to be known or computed from external sources:

- Knowledge of which variables are pointers is available (from variable declarations and/or function prototypes).
- All loop input and output variables have been identified, including those that form a back-edge from the loop’s output to its input. These can be found from basic block analysis.
- The values of constant variables are known, via standard constant propagation techniques.

Furthermore, separate dataflow graphs are initially created for the prologue and kernel portions of the code. Finding the basic, unadjusted spatial references in the separate portions greatly simplifies the overall analysis, and the results can be combined later.

3.2 Attributed Dataflow Representation

The basic program representation used is a dataflow graph in which each node represents a functional relationship between its inputs and outputs. Fragments from the convolution example are shown in Figures 4

and 5 ("cols" is the width of the input image). With respect to the goal of extracting data references, such graphs expose the arithmetic expressions used to calculate the references' addresses. Furthermore, these graphs represent both the core and address computations in a language-independent manner.

Since these graphs represent iterative structures, they would technically be cyclic graphs with back-edges from loop outputs to loop inputs, hence, a back-edge node is used. The back-edge node is a single-input, single-output node that represents a data flow from one iteration to the next. During source code parsing, it is created based on variable uses and definitions. If a variable definition is visible beyond the end of a loop body (i.e., is a loop output), and that same variable is used in the loop body before being defined (i.e., is a loop input), then a back-edge node is inserted from the corresponding output edge to the appropriate input edge.

The other two special nodes are the LOAD and STORE nodes. These represent the points in the graph where data must be read from or written to memory, respectively. A LOAD takes as its input a memory address and produces the data located at that address; similarly, a STORE takes as its inputs a memory address and data to be written, and has no outputs. Slight variations can tailor the LOAD and STORE to handle subscripted arrays (e.g., a LOAD might have inputs for each of its subscripts, in addition to the base address input).

It has been observed that the graph often has a natural partitioning into two sets of sub-graphs: those that deal with address calculation of the inputs and outputs, and those that deal with the actual algorithmic (or "core") computations. Such partitioning, and the interface (after graph parsing) between the partitions, is illustrated in Figure 5. Intuition suggests that the boundaries between the address subgraphs and the core computation subgraphs would occur at nodes describing load/store functionality. By separating these types of functionality from each other, the appropriate transformations can be applied to the data accesses to tailor them for the new architecture (e.g., neighborhood array accesses can be changed to inter-processor data movement), while the core computations can be independently transformed (e.g., if-conversion-style transformations for conditionals) and sent to a new architecture.

Of particular interest in this step of the analysis is the annotation of edges according to whether they are used to carry data information for the core computations or whether they carry information used to calculate addresses. For address calculation flows, the distinction is made between edges that carry a pointer expression, and those that carry some offset from a known loop-indexing variable (referred to as pointer and index edges, respectively). Since internal graph edges are not

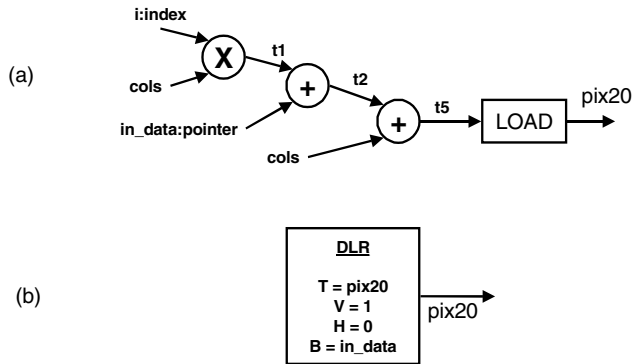


Figure 4. (a) Sample address calculation sub-graph from the prologue of the TI convolution algorithm. (b) Data-Load Reference (DLR) node as an abstraction over the sub-graph in (a).

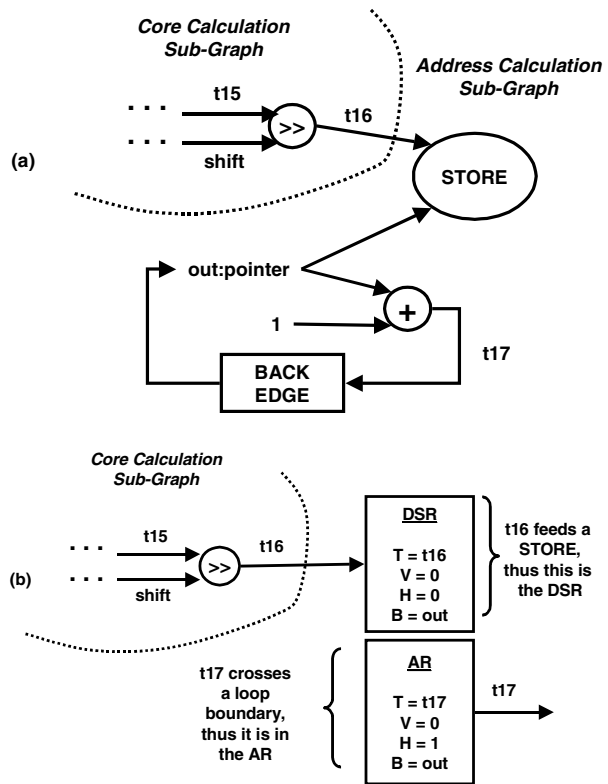


Figure 5. Interface between the core calculation and address calculation sub-graphs. (a) graph fragment prior to graph parsing. (b) fragment after spatial dependencies have been identified via graph parsing.

automatically annotated in this way, the rules of Figure 6 are needed to propagate these flows from the inputs. These rules are applied, beginning with loop inputs, and continuing toward outputs until a load or store node, or a loop output, is reached. Using these rules, it can be

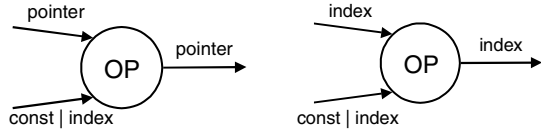


Figure 6. Edge-attribute propagation rules.
OP is either '+' or 'X'.

shown that edge t1 of Figure 4 is an index edge, and that edges t2 and t5 are pointer edges, as is edge t17 of Figure 5.

3.3 Finding basic spatial references

The next step in the analysis is to find the basic spatial references contained in each graph (prologue and kernel). These references denote points in the graph that access data values (or addresses of data) in the 2D input

image. The algorithm output (per pixel) is a function of these basic references. A basic reference can be described as a four-tuple (T, V, H, B), where T is the edge label describing the reference's position in the graph, and V and H are the vertical and horizontal offsets, respectively, from a base address B, where B is a pointer to some location in an image array. In other words, the edge T represents the two-dimensional offset (V, H) from the base pointer B. Figures 4 (b) and 5 (b) illustrate using a basic reference to abstract over address calculation sub-graphs. Further, they also illustrate the three variants on the basic reference:

1. Data Load Reference (DLR): for a data load reference, an actual data element has been accessed, via a load node, to be input to a calculation. Figure 4 (b) is a DLR.
2. Data Store Reference (DSR): for a data store

Left-Hand Side	Right-Hand Side	{Constraints} Attributes	Rule #	Left-Hand Side	Right-Hand Side	{Constraints} Attributes	Rule #
		DLR.H = P.H DLR.V = P.V DLR.Base = P.Base	(1)			{ Q crosses loop boundary edge } Q.H = Q.V = 0	(10)
		DLR.V = I.temp DLR.H = J.temp DLR.Base = P	(2)			if (C%num_cols)==0 Q.V = P.V + C/num_cols Q.H = P.H else Q.H = P.H + C Q.V = P.V Q.Base = P	(11)
		DLR.V = I.V DLR.H = I.H DLR.Base = P	(3)			Q.V = P.V + J.V Q.H = P.H + J.H Q.Base = P.Base	(12)
		DLR.V = 0 DLR.H = I.temp DLR.Base = P	(4)			Q.V = P.V Q.H = P.H + J.temp Q.Base = P.Base	(13)
		DSR.H = P.H DSR.V = P.V DSR.Base = P.Base	(5)			T.H = 0 if C == num_cols T.V = I.temp else fail	(14)
		DSR.V = I.temp DSR.H = J.temp DSR.Base = P	(6)			if (C % num_cols) == 0 T.V = I.V + C/num_cols T.H = I.H else T.H = I.H + C T.V = I.V	(15)
		DSR.V = I.V DSR.H = I.H DSR.Base = P	(7)			T.V = I.V T.H = I.H + J.temp	(16)
		DSR.V = 0 DSR.H = I.temp DSR.Base = P	(8)			{ T crosses loop boundary edge } T.H = T.V = 0	(17)
		{ P crosses loop boundary edge OR connects to back-edge block }	(9)			T.temp = I.temp + C T.V = 0 T.H = 0	(18)

Figure 7. Graph-grammar rules

Table 1.
TI Convolution
Prologue Results

	T	V	H	B (Base)
DLR _p	pix10	0	0	in_data
	pix11	0	1	in_data
	pix12	0	2	in_data
	pix20	1	0	in_data
	pix21	1	1	in_data
	pix22	1	2	in_data
	pix30	2	0	in_data
	pix31	2	1	in_data
	pix32	2	2	in_data
AR _p	in1	0	3	in_data
	in2	1	3	in_data
	in3	2	3	in_data
	out	1	1	out_data

Table 2.
TI Convolution
Kernel Results

	T	V	H	B (Base)
DLR _k	pix13	0	0	in1
	pix23	0	0	in2
	pix33	0	0	in3
DSR	t16	0	0	out
AR _k	in1_0	0	1	in1
	in2_0	0	1	in2
	in3_0	0	1	in3
	t17	0	1	out

reference, the result of some core computation has been written back to memory location. Note that since there is assumed to be only one store for the code under analysis (in the inner loop), there can be only one DSR. The DSR is noted on Figure 5 (b),

- Address Reference (AR): an address reference exists for an address edge that reaches the end of a loop (i.e., is a loop output). An AR propagates this offset information to the next iteration (or into the next nested loop body). An AR element is illustrated in Figure 5 (b).

The graph grammar rules used to parse the graph fragment given in Figure 4 (a) are shown in Figure 7. The DLR shown in figure 4 (b) can be found by applying the rules, from the left of 4 (a) towards the right, in the following order: 17 (for the “I:index” input), 14, 10 (for the “in_data:pointer” input), 12, 11, and 1. The grammar rules match expressions of the form “A+(I+N)*num_cols+(J+M)” where A is a base pointer, I and J are index variables, N and M are arbitrary constants, and num_cols is a known row width. This is the general expression form for accesses to two-dimensional arrays stored in row-major form. The grammar also includes rules to handle expressions for accessing singly- and doubly-indexed arrays.

For the reference extraction stage of the analysis, a set of DLR’s and AR’s are returned for both the prologue (DLR_p, AR_p) and the kernel (DLR_k, AR_k), as well as the single DSR from the kernel. For the convolution

example, the values returned are listed in Tables 1 and 2.

3.4 Normalizing reference values

This step combines prologue and kernel results into a single set of reference values, and then normalizes those values into the final results. Initially, the base addresses of the input and output arrays are taken as the zero-point (since they are of the same dimensions and can be visualized as having a one-to-one correspondence of pixel locations). However, not all raw values in Tables 1 and 2 have these points as their bases. Thus, some adjustment is required for each to find its absolute distance from the initial origin. For example, the distance from in_data to in2_o is (1,3) + (0,1) = (1,4) (i.e., the distance from in_data to in2 in the prologue plus the distance from in2 to in2_o in the kernel). Figure 8 (a) shows, on an x-y graph, the layout of all DLR/DSR/AR points (in the combined set taken from Tables 1 and 2), using the in_data/out_data point as the origin, after such an adjustment has been performed.

All references, whether they are data-load or address references, must then be normalized such that the data store reference (i.e., the output pixel location) is the zero-point. The reason for this is that image processing algorithms are typically thought of as an output pixel (i.e., the DSR) being a function of some set of input pixels, so it is useful to make the DSR the zero-point. Since all references must maintain the same relative distance from one another, when the DSR (denoted by the black triangle) -- here, initially at (1,1) -- is shifted to (0,0), all other references must shift accordingly. As stated earlier, the initial common point of reference is in_data/out_data, so some adjustment is necessary. Figure 8 (b) shows the results of the normalization shift, which are the final values of interest. In particular, it shows the points pix10, pix11, pix12, pix20, pix21, pix22, pix30, pix31, and pix32 as forming a neighborhood of DLR’s around the DSR -- these correspond directly to the 3x3 neighborhood of 2D references needed by a general 3x3 convolution. When compared to the convolution code in the Appendix (lines 30-32), one can see that these are, in fact, the neighborhood values expressed in the source.

4. Validation

Our primary motivation for reverse engineering to an IR that exposes 2D spatial data relationships is to enable retargeting to a SIMD platform where uniform computations on 2D regions of pixels can be performed more efficiently. However, to validate the primary reverse engineering step, before composing it with SIMD code generation, we generated Matlab code from the extracted intermediate representation, as shown in Figure 9. This allows us to confirm that the information extracted produces the same results as the original C code. We also validated the technique on additional image filtering programs to demonstrate its applicability.

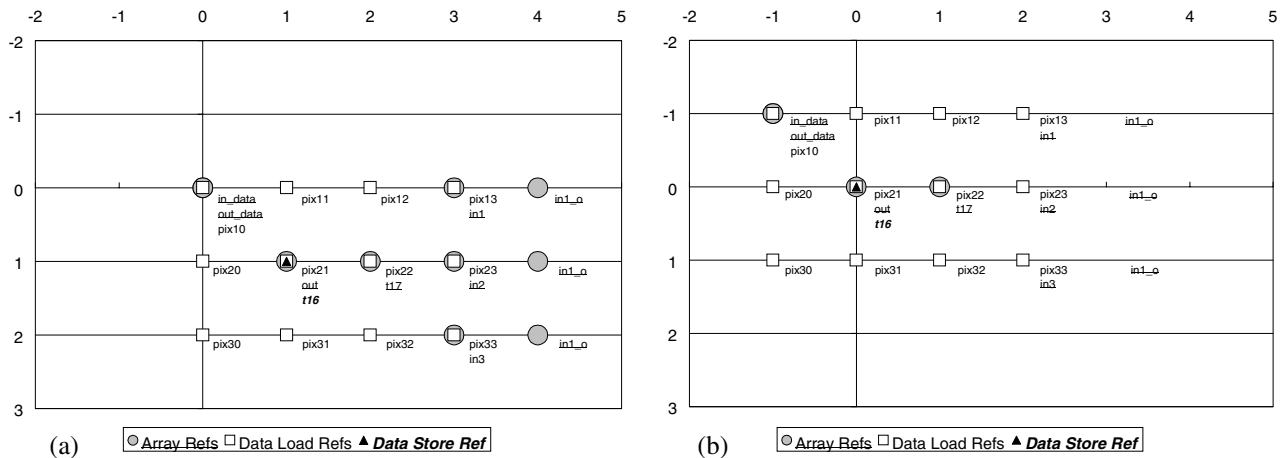


Figure 8. Relative positions of AR's/DLR's/DSR (a) before and (b) after normalization.

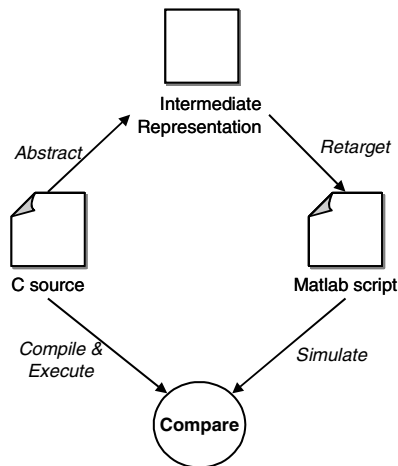
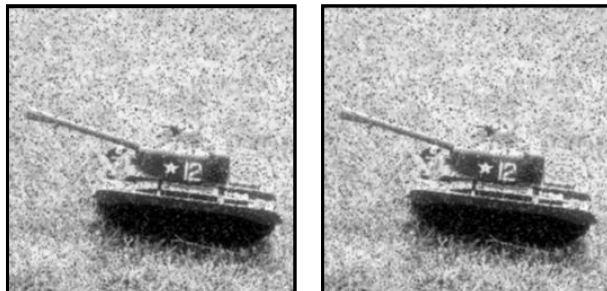


Figure 9. Validation experiment

```
# Begin convolution script
function t16 = conv(in_data,
mask10, mask11, mask12,
mask20, mask21, mask22,
mask30, mask31, mask32,
round, shift)
pix10=comm(in_data,-1,-1);
pix11=comm(in_data,-1,0);
pix12=comm(in_data,-1,1);
pix13=comm(in_data,-1,2);
pix20=comm(in_data,0,-1);
pix21=comm(in_data,0,0);
pix22=comm(in_data,0,1);
pix23=comm(in_data,0,2);
pix30=comm(in_data,1,-1);
pix31=comm(in_data,1,0);
pix32=comm(in_data,1,1);
pix33=comm(in_data,1,2);
t2=scalar_mult(pix11,mask11);
t3=scalar_mult(pix12,mask12);
t4=add(t2,t3);
t5=scalar_mult(pix20,mask20);
t6=scalar_mult(pix21,mask21);
t7=scalar_mult(pix22,mask22);
t8=add(t6,t7);
sum00=add(t1,t4);
sum11=add(t5,t8);
t9=scalar_mult(pix30,mask30);
t10=scalar_mult(pix31,mask31);
t11=scalar_mult(pix32,mask32);
t12=add(t10,t11);
sum22=add(t9,t12);
t13=add(sum11,sum22);
t14=add(sum00,t13);
t15=add(t14,round);
t16=scalar_shr(t15,shift);
# End Convolution script
```

Fig. 10. Generated Matlab script.



C Version

Retargeted (Matlab) Version

Figure 11. Output images of compiled C and retargeted Matlab convolution algorithms.

```
// Array dimension.
#define DIMENSION 500
// Input array.
int in[DIMENSION][DIMENSION];
// Output array.
int out[DIMENSION][DIMENSION];
void CalculateAveragingFilter()
{
    int i, j;
    int sum;
    for (i=1; i<(DIMENSION-1); i++) {
        for (j=1; j<(DIMENSION-1); j++) {
            sum = in[i-1][j-1] + in[i-1][j] +
                in[i-1][j+1] + in[i][j-1] +
                in[i][j] + in[i][j+1] +
                in[i+1][j-1] + in[i+1][j] +
                in[i+1][j+1];
            out[i][j] = sum/9;
        } }
}
```

Fig. 12. Averaging Filter

4.1 Comparing results of sequential code versus retargeted Matlab code

As shown in Figure 9, the retargeting tool was applied to the convolution example to produce the Matlab script of Figure 10, which uses DLP primitives we define in Matlab, such as matrix-add (add) and communicate-with neighbor (comm). The lines `pix10=...` through `pix33=...` validate that the correct communication patterns of (-1,-1),(-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), and (1,1) have been identified. Additionally, the C version was compiled into a normal executable, and the results of the two versions (qualitatively shown in Figure 11) were compared for equivalence. As a quantitative comparison metric, the following average distance formula was used:

$$avg_dist = \frac{1}{MN} \sum_{n=1}^N \sum_{m=1}^M |a_{nm} - b_{nm}|$$

Here, *a* and *b* are the output images produced by the two programs, *N* and *M* are the dimensions of the images, and *avg_dist* is the average distance between the two. The resultant average distance for the comparison was 3.3520 (note that pixel values range from 0 to 255, as these are 8-bit images). This error is relatively small and is most likely due to datatype mismatching between the C (small-precision integers) and Matlab (larger-precision floating point numbers) versions.

4.2 Other examples

Two other filtering algorithm programs have been tested with this process to ensure the pattern-matching rules are sufficiently robust, and to demonstrate the process' applicability over a range of image filtering operations. The first is a 3x3 averaging filter (Figure 12), where each pixel is replaced by the average of itself and its eight neighbors, implemented using doubly-subscripted array notation. The second is a 3x3 median filtering algorithm, which creates a sorted list of values from a pixel and its eight nearest neighbors, and replaces that pixel with the median (fifth) value. Using the prototype analysis tool, the expected spatial dependency information was successfully extracted for both algorithms.

An averaging filter is algorithmically similar to a convolution (technically, an averaging filter is a convolution where all mask elements are 1/9). This version was chosen to test the robustness of the grammar rules to handle doubly-subscripted notation.

The median filter used here is from the same Texas Instruments code library as the 3x3 convolution described earlier. It uses address calculations similar to that of the convolution, but the core algorithm using that

data is quite different - a non-iterative sorting algorithm with many comparisons and conditionals, in contrast to the convolution's arithmetic core.

5. Further Research

This paper explores an important aspect of the DLP retargeting process, that of finding the spatial data references inherent in the source code. More support for this process is required, including inter-procedural analysis. Further, to retarget a larger body of algorithms, more complex reference patterns must be investigated. Additional challenges include:

- **Targeting variable-grain SIMD platforms:** We plan to compose the reverse engineering technique with a retargeting SIMD compiler that is under development [20]. This will allow specifications such as Figure 2(c) to be retargeted to a wide range of SIMD architectural configurations (from packed subword parallel to fine-grain massively parallel configurations).
- **Optimizations:** The programmer may employ optimizations (such as loop unrolling, or carrying the result of a common calculation across iterations of a loop) that severely complicate the data access patterns and obscure the regular, uniform data access needed for data parallel execution.
- **Complex control flow in core:** Core computations may require further retargeting techniques, e.g., to deal with control flow structures within the core. When retargeting to data parallel code, branches can often be eliminated by turning them into parallel predicates and using data masking.
- **Epilogue:** The technique must be generalized to deal with "epilogue" code, which is code within the outer loop but executed after the inner loop.

6. Conclusions

We have shown that, for a range of image filtering operations written in a sequential language such as C, we can automatically identify the high-level spatial data dependencies from the lower-level expressions in the source. Further, our research has shown that using such dependence information along with the "core" algorithmic body of the source, the filtering algorithms can be successfully retargeted to a platform implementing a data-parallel style of computation.

Acknowledgement

We are grateful to Antonio Gentile, Murat Guler, Sam Sander, Tarek Taha, and Scott Wills for their advice and technical assistance. We also appreciate the comments of the anonymous reviewers. This work was supported in part by the National Science Foundation under NSF CAREER Grant CCR-0092552.

References

- [1] R. Gonzalez and R. Woods, *Digital Image Processing*, Addison-Wesley, 1993.
- [2] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, Inc., 1997.
- [3] J. R. Allen and K. Kennedy, Automatic Translation of FORTRAN Programs to Vector Form, *TOPLAS*, 9:4, pp. 491-542, Oct.1987.
- [4] D. DeVries, A Vectorizing SUIF Compiler: Implementation and Performance. Master's thesis, University of Toronto, June 1997.
- [5] D. Kuck, et al., Analysis and transformation of programs for parallel computation, computation, In *COMPSAC 80*, pp. 709-715, Chicago, IL, October 1980.
- [6] R. Wilson, et al., SUIF: an infrastructure for research on parallelizing and optimizing compilers, *SIGPLAN Notices*, 29(12), pp. 31-37, Dec. 1994.
- [7] K. McKinley, "Automatic and Interactive Parallelization," Rice University, Houston, Texas, CRPC-TR92214-S, March 1994.
- [8] D. Kulkarni, M. Stumm, "Linear Loop Transformations in Optimizing Compilers for Parallel Machines," *The Australian Computer Journal*, May 1995, pp. 41-50.
- [9] C. Lee and D. DeVries, Initial Results on the Performance and Cost of Vector Microprocessors, *Proc. of 30th Annual Int. Symposium on Microarchitecture*, pp. 171-182, Research Triangle Park, NC, Dec. 1997.
- [10] D. Callahan, et al., "Vectorizing compilers: a test suite and results," *Proc. Supercomputing '88*, pp. 98 -105, Nov. 1988.
- [11] B. Di Martino, H. P. Zima, "Support of automatic parallelization with concept comprehension," *Journal of Systems Architecture*, 45(6-7):pp. 427-439, 1999.
- [12] Christoph W. Kessler, "The PARAMAT Project: Current Status and Plans for the Future," in *Proc. of AP'95 Workshop on Automatic Data Layout and Performance Prediction*, Rice University, Houston, April 1995.
- [13] B. Di Martino, C. W. Kessler, "Two Program Comprehension Tools for Automatic Parallelization," *IEEE Concurrency*, vol. 8, no. 1, January-March 2000, pp. 37-59.
- [14] Christoph W. Kessler, "Pattern-Driven Automatic Parallelization," *Scientific Programming*, vol. 5, no. 3, Fall 1996, pp. 251-274R.
- [15] R. Metzger, "Automated Recognition of Parallel Algorithms in Scientific Applications," in *Proc. of Workshop on Plan Recognition at IJCAI'95*, August 1995.
- [16] V. Gustin, T. M. Pinkston, "Extracting SIMD parallelism from 'for' loops," *ICPP*, Valencia, Spain, Sept. 2001, pp. 23 - 28.
- [17] G. A. Baxes, *Digital Image Processing: Principles And Applications*, NY: John Wiley & Sons, Inc., 1994, pp. 86-99.
- [18] H. H. Cat, et al., "SIMPil: An OE Integrated SIMD Architecture for Focal Plane Processing Applications," *Proc. 3rd International Conference on Massively Parallel Processing using Optical Interconnections*, Maui, 1996, pp. 44-52.
- [19] *TMS320C62x Image/Video Processing Library Programmer's Reference*, Texas Instruments Literature Number SPRU400, March 2000, pp. 5-36—5-38.
- [20] S. Sander, "Retargetable Compilation for Variable-Grain Data Parallel Execution in Image Processing," Georgia Tech Ph.D. thesis, August 2002.
- [21] A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, Vol. 16, No. 4, pp. 5159, Aug. 1996.
- [22] M. Phillip et al, "AltiVec Technology: Accelerating Media Processing Across the Spectrum," *Proc. of the HOTCHIPS-X Conference*, Aug. 1998.
- [23] S. Oberman et al, "AMD 3DNow! Technology and the K6-2 Microprocessor," *Proc. of HOTCHIPS-X Conference*, 1998.

Appendix: Convolution Code

```
//Modified, as previously described, from that described in [19]
void corr_3x3 ( const unsigned char *in_data, unsigned char
               *out_data, int rows, int cols, unsigned char
               *mask, short round )
{
int count,sum,j, i;
int sum00,sum11,sum22;
const unsigned char *IN1,*IN2,*IN3;
unsigned short pix10,pix11,pix12,pix13;
unsigned short pix20,pix21,pix22,pix23;
unsigned short pix30,pix31,pix32,pix33;
unsigned short mask10,mask11,mask12;
unsigned short mask20,mask21,mask22;
unsigned short mask30,mask31,mask32;
unsigned char *OUT;
unsigned char shift = 4;
mask10 = mask[0]; mask11 = mask[1]; mask12 = mask[2];
mask20 = mask[3]; mask21 = mask[4]; mask22 = mask[5];
mask30 = mask[6]; mask31 = mask[7]; mask32 = mask[8];
1: for(i = 0; i < rows-2; i++)
2: { IN1 = in_data + (i*cols);/* input pointer to first row */
3: IN2 = IN1 + cols; /* input pointer to 2nd row */
4: IN3 = IN2 + cols; /* input pointer to 3rd row */
5: OUT = out_data + ((i+1)*cols) + 1; /*output pointer */
6-8: pix11=*IN1++; pix12=*IN1++; pix13=*IN1++;
9-11: pix21=*IN2++; pix22=*IN2++; pix23=*IN2++;
12-14: pix31=*IN3++; pix32=*IN3++; pix33=*IN3++;
15,16: count = (cols - 2); sum = round;
17: for (j = count; j>0; j--)
18-20: { pix10 = pix11; pix11 = pix12; pix12 = pix13;
21: pix13 = *IN1++;
22-24: pix20 = pix21; pix21 = pix22; pix22 = pix23;
25: pix23 = *IN2++;
26-28: pix30 = pix31; pix31 = pix32; pix32 = pix33;
29: pix33 = *IN3++;
30: sum00 = ((pix10*mask10) + (pix11*mask11) +
(pix12*mask12));
31: sum11 = ((pix20*mask20) + (pix21*mask21) +
(pix22*mask22));
32: sum22 = ((pix30*mask30) + (pix31*mask31) +
(pix32*mask32));
33: sum += sum00+sum11+sum22;
34: sum >>= shift;
35: *OUT++ = sum;
36: sum = round; } }
```