

Extracting an Explicitly Data-Parallel Representation of Image-Processing Programs

Lewis Baumstark
lewisb@ece.gatech.edu
School of Electrical and Computer Engineering, Georgia Institute of Technology

Murat Guler
gt1452c@prism.gatech.edu

Linda Wills
linda.wills@ece.gatech.edu
School of Electrical and Computer Engineering, Georgia Institute of Technology

Abstract

Our research goal is to retarget image processing programs written in sequential languages (e.g., C) to architectures with data-parallel processing capabilities. Image processing algorithms are often inherently data-parallel, but the artifacts imposed by the sequential programming language (e.g., loops, pointer variables, linear address spaces) can obscure the parallelism and prohibit generation of efficient parallel code. This paper proposes a program representation and pattern-recognition approach for generating a data-parallel program specification from sequential source code. The representation is based on an extension of the multi-dimensional synchronous dataflow (MDSDF) model of computation.

Central to extracting this representation from code is understanding the mapping between iterations and array variables in the source code and the operations over array regions (e.g., rows, columns, tiled blocks) that they implement. Examples are presented to illustrate this mapping, and a set of patterns for recognizing these regions are proposed. The correctness of the retargeted MDSDF specifications are validated and the potential speedup from parallel execution shown.

Keywords: program transformation, parallelization, program representation, SIMD, pattern-recognition

1. Introduction

Our research is focused on development of a system to retarget image and video processing applications written in sequential languages such as C to processors that support single-instruction, multiple-data (SIMD) execution, such as Intel's multimedia extensions (MMX) [1] and streaming SIMD extensions (SSE) [2], the Maspar MP-1 [3], the CM-2 Connection Machine [4], or the SIMPil processor [5]. These types of applications, including image convolutions and discrete cosine transforms, are known to be highly data-parallel in that the same operation is applied to all the data elements

(pixels or pixel regions) in a two-dimensional array (image). Since these operations are typically independent of each other, there is a huge potential for concurrent execution on a SIMD architecture. Sequential code, however, may obscure the data-level parallelism (DLP) with low-level syntactic and implementation details, for example, an operation on an array element within a loop body may not be readily recognizable as that operation mapped across all the elements of the array. The system under development transforms such low-level constructs to the high-level explicitly data-parallel concepts they implement; the results of this transformation can then drive code generation for SIMD architectures.

1.1. MDSDF representation and recognition

Given sequential C source code, our goal is to extract a high-level specification with explicit data-parallel semantics. Consider the source code of Figure 1, implementing a 3x3 convolution routine. Here, each pixel in the output image is a weighted sum of each of its eight nearest neighbors and itself; this is a common filtering operation that can be used to perform, for example, edge detection or noise removal, depending on the filter weights used.

Figure 2 shows an explicitly data-parallel specification for this algorithm, which we would like to extract from the code. This specification is captured in an extension of the multi-synchronous dataflow (MDSDF) model of computation (MOC) [6]. MDSDF is an extension of the synchronous dataflow (SDF) MOC; both are incorporated in the Ptolemy project [7]. In SDF models, production and consumption rates of (scalar) tokens are constant integers, leading to statically determinable execution rates for each node (process).

MDSDF augments SDF with multi-element, multi-dimensional tokens. These tokens provide a useful representation for image processing algorithms, which often operate on regular sub-regions of an image, such as rows, columns, and tiled blocks. Our work extends MDSDF in three ways (described in section 2.2): by explicitly specifying the bounds (as ranges of values) of the multi-dimensional tokens, by providing a relative

```

/* 3x3 convolution from the ESO Eclipse image processing
suite. Has minor changes to replace structure references
in the loop bodies to scalar variables */
01:image_t * image_filter3x3(image_t *image_in,
    double *filter) {
02: image_t *image_out ;
03: int i, j, curr_pos, im_width, im_height ;
04: double sum_pix, filter_norm ;
05: pixelvalue *in_data, *out_data;
06: im_width = image_in->lx ; im_height = image_in->ly;
07: in_data = image_in->data; out_data = image_out->data;
08: for (j=1 ; j<im_height-1 ; j++) {
09:     for (i=1 ; i<im_width-1 ; i++) {
10:         curr_pos = i + j*im_width; sum_pix = 0.0 ;
11:         sum_pix += filter[0] *
            (double)in_data[curr_pos-1-im_width] ;
12:         sum_pix += filter[1] *
            (double)in_data[curr_pos-im_width] ;
13:         sum_pix += filter[2] *
            (double)in_data[curr_pos+1-im_width] ;
14:         sum_pix += filter[3] *
            (double)in_data[curr_pos-1] ;
15:         sum_pix += filter[4] *
            (double)in_data[curr_pos] ;
16:         sum_pix += filter[5] *
            (double)in_data[curr_pos+1] ;

```

Figure 1. 3x3 convolution source code.

orientation of tokens when this is not implicit in the bounds, and by allowing symbolic expressions in dimensions. This extended MDSDF model provides several benefits for representing image processing algorithms:

1. The dimension attributes on the edges indicate the array regions operated upon and, when edge source and edge sink dimensions differ, the interactions between regions (e.g., partitioning into smaller regions, accumulation of scalar values into a vector, or transposing an $N \times M$ region into an $M \times N$ region).
2. Data-parallel operations can be easily identified. In Figure 2, consider the addition node highlighted by the gray rectangle. Its input and output edge dimensions are all equal, thus it represents a parallel, element-by-element addition over the input arrays. (Similarly, the other addition nodes have the same dimensions.)
3. A complete recognition of the program (here, as a convolution algorithm) is not necessary; a recognizer only needs to identify the data access patterns at the inputs and outputs of the graph and, in some cases, certain computational patterns on the interior (such as a sum over array elements). In particular, a distinction is made between those portions of the program that access and store data (in Figure 2, everything outside the dashed box, e.g., the *in_data* array coupled with the matrix-shift operators) and those that perform computations on the data (everything inside the dashed box).

4. Dataflow-based representations are familiar to systems engineers as these models are a natural way of expressing DSP and image-processing algorithms (e.g., they are used extensively in Cadence's Signal Processing Worksystem). Hence, in addition to facilitating the retargeting of sequential code to SIMD processors, this representation will aid human understanding of sequential DSP and image-processing programs. This will allow these programs to be imported into common DSP tools for less error-prone evaluation.
5. By building on the MDSDF formalism, our work is compatible with the Ptolemy heterogeneous modeling and simulation project [7]. By using MDSDF, which is an available MOC within the Ptolemy Classic tool, we will be able to integrate our recognized program specifications with other MOCs that are available in Ptolemy. For instance, they may be part of a larger distributed embedded system such as a Kahn Process Network, where locally-synchronous components globally communicate with each other asynchronously. Ptolemy provides simulation capabilities to validate the models extracted and to guide further retargeting.

This paper presents a dataflow representation for sequential programs written in C, an extended MDSDF representation for representing explicit data-parallelism, and a pattern-matching system for abstracting the former into the latter. We extend MDSDF with an array bounds notation, an attribute for ensuring proper orientation of arrays, and by allowing symbolic expressions in array dimensions.

Our earlier work [19] was targeted at image-filtering programs, which apply the same operations to every pixel in an image. The program representation described in this paper allows a larger class of programs that operate on sub-regions of pixels to be recognized and retargeted to parallel architectures, while still including image-filtering algorithms.

The remainder of this section discusses compiler and reverse engineering methods for extracting data-level parallelism from source code. Section 2 describes the representation and several useful patterns and illustrates the recognition method with an example. Section 3 presents an experiment that validates correctness of the retargeted programs and, additionally, provides a runtime comparison between sequential execution of the original programs and parallel execution of the retargeted programs. Section 4 concludes with discussion and issues to be addressed in future work.

1.2. Related work

Traditional data-level parallelization work (e.g., [8]-[10]) has focused on vectorization of loops and arrays in scientific applications. The primary goal has been to

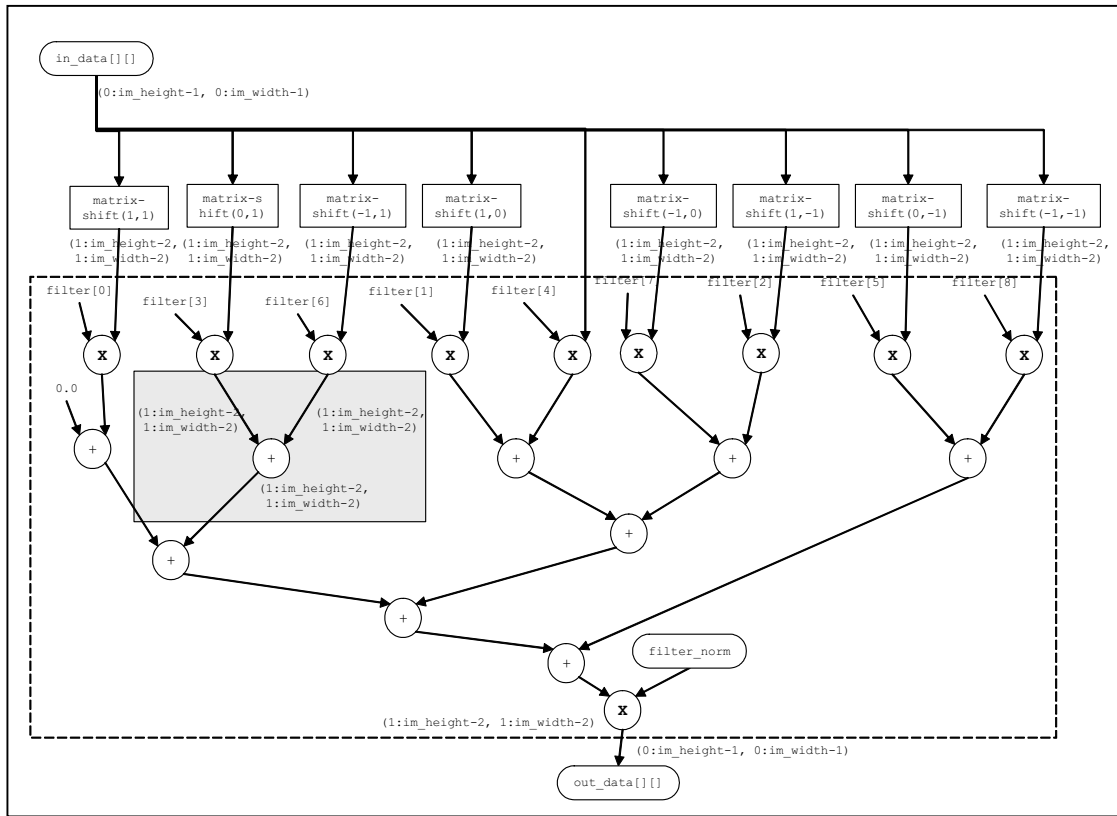


Figure 2. 3x3 Convolution, extended-MDSDF specification

achieve performance increases by identifying what data can be piped through composed operations in long one-dimensional vectors. These techniques have also been adapted for exploiting one-dimensional data-level parallelism in SIMD instruction set extensions ([11]-[13]). Vectorization attempts only to schedule program statements for parallel execution; instead of simply finding parallel statements, our representation extracts the parallel algorithm as a computational structure with parallel semantics. Furthermore, most vectorization techniques only recognize parallelism in one dimension, whereas our techniques recognize both one- and two-dimensional operations, attaining more efficient exploitation of parallelism for region-based operations such as convolution.

Waters [14] describes a loop as a “temporal sequence of values,” i.e., a stream of values produced by the computations in the loop. He defines several *plan-building methods* (PBMs) which decompose the loop control and computations into abstractions over these streams of data. An example is the *augmentation* PBM which consumes, element-by-element, a sequence of values and produces an equal-length sequence of values as some function of the original (e.g., multiplying the elements of an array – the sequence – by a scalar value). This work is similar; we also abstract over the temporal loop components inherent in sequential programming

languages (e.g., iteration, loop body calculations), but instead of linear sequences, we create temporal abstractions over two-dimensional array regions.

More recently, projects such as PAP ([16]-[18]) and PARAMAT ([15][18]) have employed program recognition to extract 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. Both PAP and PARAMAT attempt to replace identified sequential algorithms with well-known parallel implementations of those algorithms. Like these projects, our work employs pattern matching and transformations over a graph-based program representation. The main difference between these projects and our work is that their goal is a complete recognition of the program as a single well-known sequential algorithm that can be replaced with a parallel implementation, whereas our strategy operates at lower levels of abstraction, without requiring a complete recognition of the program. This efficiently accommodates variability in the overall algorithm while

benefiting from the recognition of common building blocks and data reference patterns used in these algorithms.

2. Representation and recognition

2.1. Source program representation

To facilitate recognition, the C source must be parsed and translated into a suitable representation. The initial program representation is a flowgraph with the following node types to represent the program operations:

- *Basic computational operators*: addition, multiplication, logical operations, comparison operations, etc.
- *Memory operators*: load and store, with appropriate variants depending on whether the access is through a pointer, a singly-subscripted array, or a doubly-subscripted array.
- *Interfaces*: input and output nodes, representing data flows into or out of the flowgraph.
- *Control blocks*: abstractions over embedded loops and conditionals.

Edges connect node outputs to node inputs, however, cycles are not allowed. Instead, to indicate feedback from an output node to an input node of the same graph, an attribute is added to the output-input node pair; this is illustrated with a '*' on interface nodes having the same label.

The *memory* operators are required to represent accesses through pointer and array types. The simplest load, "LD", takes a pointer address as an input and produces the value stored at the address. The analogous store takes a pointer address edge and the data edge to be stored as its inputs, and produces no output. More complex load and store operators include the singly- and doubly-subscripted versions. For instance, the one-dimensional array-access load "LD[]" takes an array (or pointer) and an indexing expression as its inputs, producing the array element as its output; the corresponding one-dimensional store has a data input in addition to the array and index inputs and produces an array as its output.

Control blocks encapsulate loops and conditionals. Control- and data-flow information are difficult to represent on the same flow graph, so a control block allows, for example, an embedded loop to be represented in a dataflow graph by treating the entire loop (including its body) as a single node. Each contains its own dataflow graph (i.e., the loop body), and could itself contain other control blocks. The control blocks are annotated with information from the source code, including the block type (loop or conditional), loop exit or branching condition, and any loop initializations (e.g., the "i=0;" in "for(i=0;i<N;i++)").

2.2. Explicitly parallel target representation

Our goal is to abstract the initial source representation into a higher-level explicitly data-parallel representation. Our target is an extended form of the MDSDF model [6]. In MDSDF edges may carry multi-element tokens as multi-dimensional arrays. (The simpler case of MDSDF, defined for rectangular array tokens, will be used here, although the formalism allows for more general structures.) Further, MDSDF allows for edge sources and their connected edge sinks to produce and consume, respectively, tokens of differing dimensions. These unmatched token dimensions implicitly specify the relative rates at which the nodes in the graph must execute. Figure 3 gives an example. Since the multiply node consumes two 1x8 arrays (performing eight parallel multiplies each time it executes), it must execute eight times for every 8x8 token produced by the input A.

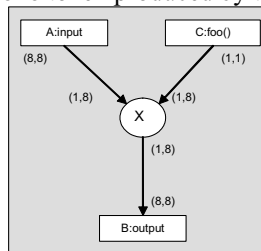


Figure 3. MDSDF example.

For our goal of retargeting sequential code to SIMD architectures, the MDSDF edge annotations encapsulate the loop control flow of the original source without requiring the embedded control blocks of the initial program representation. Additional reasoning can be applied to the MDSDF representation to generate SIMD code. In the example of Figure 3, a code generator could schedule 64 concurrent multiply operations by distributing the results of C and the corresponding elements of A to 64 different processors.

Our representation includes three extensions to the MDSDF representation. The first is that edge annotations also include the bounds of the dimensions, e.g., (0:7,0:7) specifies a two-dimensional token with bounds in each direction of zero to seven. This preserves information from the original source concerning loop iteration ranges and array boundary conditions. The second extension is the addition of an orientation attribute to the dimensions, e.g., (0:7):V specifies an eight-element vertically-oriented vector. This is needed in the cases where corresponding dimensions are vague, for example, given a multiply node with inputs (0:3,0:3) and (0:3,0:3) it is not clear whether the rows of the first input match the rows or columns of the second. Specifying (0:3,0:3):V and (0:3,0:3):H, respectively, indicates that the columns of the first input are multiplied by the rows of the second. Finally, our representation allows for symbolic expressions – necessary in an analysis of source code – in dimension

annotations, while MDSDF does not. (The MDSDF design goal of static schedulability of graph nodes as communicating processes requires exact knowledge of array dimensions, thus symbolic expressions were not part of the initial MDSDF representation.)

The examples in this paper include some shorthand. Here, “(1)” or no label indicates an edge with scalar dimensions. Additionally, if an edge dimension only appears at the producer, the consumer has the same dimensions. Finally, dimensions may, at some stages during recognition, be indeterminate or, for purposes of pattern-matching, not be relevant to the pattern. In these cases, the dimensions are expressed as an underscore.

2.3. Patterns

This section presents several common patterns useful for transforming the initial program representation to the MDSDF specification. These patterns are concerned primarily with understanding data accesses to arrays both indirectly through pointers and directly through explicit array references.

2.3.1. Count pattern

The count pattern represents an induction variable for the loop. Figure 4 shows the common case where the stride is one, abbreviated as “count(1).” Several array access patterns have the count pattern as a component. For simplicity of illustration, only the patterns using count(1) will be discussed in this paper; each load/store pattern presented here that involves count(1) will also have a count(N) analog to handle strided array accesses.

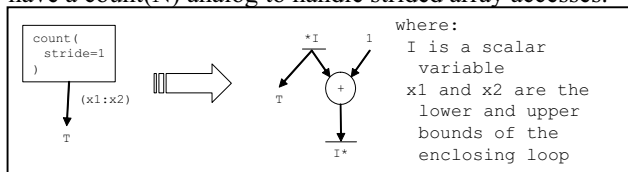


Figure 4. Count pattern with unit stride.

2.3.2. Pointer-based load and store patterns

Patterns for reasoning about loads and stores via pointers are shown in Figure 5. These illustrate the common (C language) pattern of “*A++,” i.e., where a pointer is dereferenced to load or store a value and then the pointer address is updated. The complete pattern library includes patterns for more general pointer-arithmetic expressions, similar to the graph-grammar described in our earlier work of [19]. The patterns listed in [19], and those here, derive from the common row-major memory layout of arrays, which lead to pointer expressions of the form “ $A+(i+r)*N+(j+c)$ ” for accessing elements in a two-dimensional array, where A is a pointer variable, i and j are loop index variables, r and c are constant expressions, and N is the column width (a.k.a., the row-stride). For those patterns, a distinction is made between edges that carry the result of an address calculation expression (pointer edges), those that are some

offset from a known index variable (index edges) and those that carry any other kind of data computation (data edges).

2.3.3. Array-based load and store patterns

Figure 6 shows two basic patterns for load and store accesses through singly-indexed arrays. These are the cases where an induction variable is used to index an array within a loop. Figure 7 shows the more complex situation where doubly-subscripted loads occur (the analogous stores are omitted for brevity). Here, the cumulative effects of applying subsequent loop nests to a load are illustrated; the row-generator node specifies that rows, with horizontal bounds $x1$ to $x2$, are being created from the input array (whose bounds may or may not be known at this point). An analogous set of patterns exist for the case where the columns are created first (i.e., iteration over the vertical component occurs first).

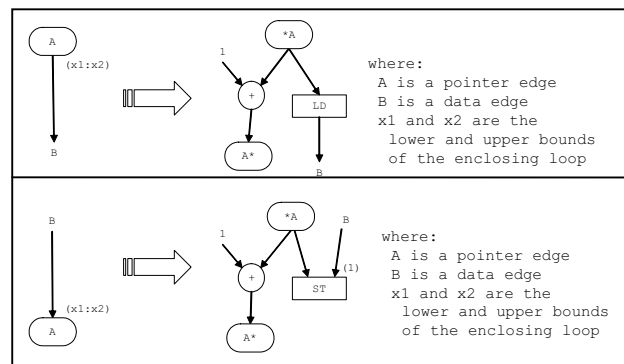


Figure 5. Example pointer-based load and store patterns.

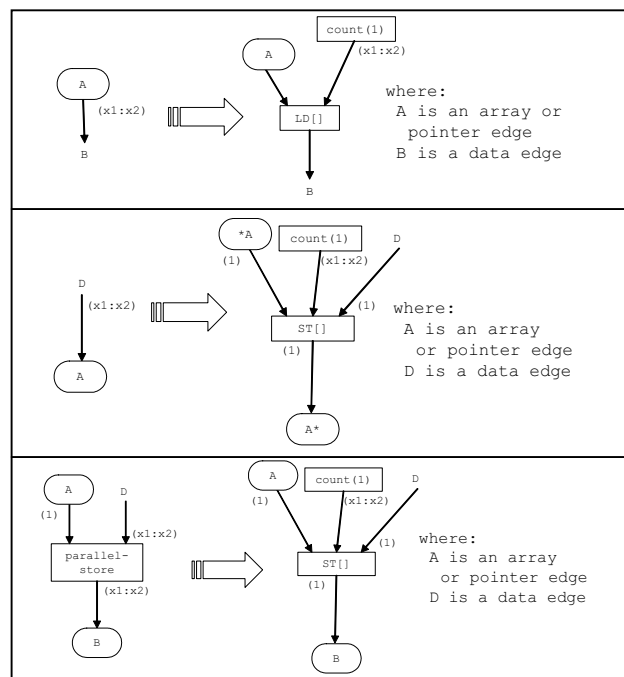


Figure 6. Example singly-indexed load and store patterns.

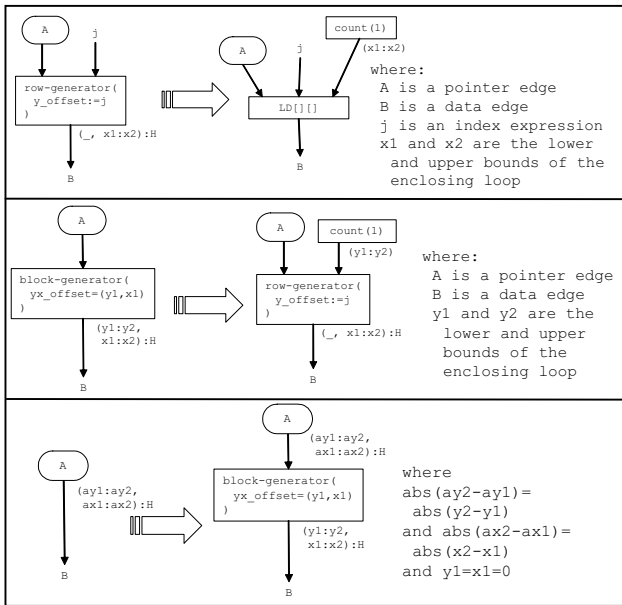


Figure 7. Doubly-indexed load patterns.

2.4. Example: 8x8 DCT

As a more detailed example, consider the source code of the Appendix, which implements a two-dimensional 8x8 discrete cosine transform (DCT). This implementation performs a one-dimensional DCT, first on every row in each 8x8 sub-block of the image and then on the columns of the row-transformed block. The one-dimensional DCT is defined as:

$$A_m = c_m \sum_{p=0}^{n-1} B_p * \cos\left(\frac{\pi(2p+1)m}{2*n}\right); m = 0,1,\dots,n-1; c_m = \begin{cases} \sqrt{1/n}, & m = 0 \\ \sqrt{2/n}, & \text{otherwise} \end{cases}$$

where B is the input vector with indices zero to (n-1) and A is the output vector with the same dimensions. The DCT is among the transformations applied in JPEG image compression.

To illustrate application of the patterns described in section 2.3, Figure 8 and Figure 9 show several recognition steps for the loop of lines 06 through 10 in the Appendix. A description of each step follows:

- This is the initial representation as derived from the source code. In terms of the entire program representation, this flow graph is contained inside a control block of type loop, with bounds (0 to cnt-1), and with loop exit condition ($x \geq cnt$). The gray curves indicate recognition of a vector-sum node and a count(1) node.
- The vector-sum operation replaces the feedback loop/addition node construct. Thus, the feedback loop over the variable z no longer exists, i.e., the graph input interface labeled z is now the initial loop value of z (carried from outside the original loop), and the graph output interface labeled z is the final loop value of z. The gray curve indicates recognition of an array input interface.

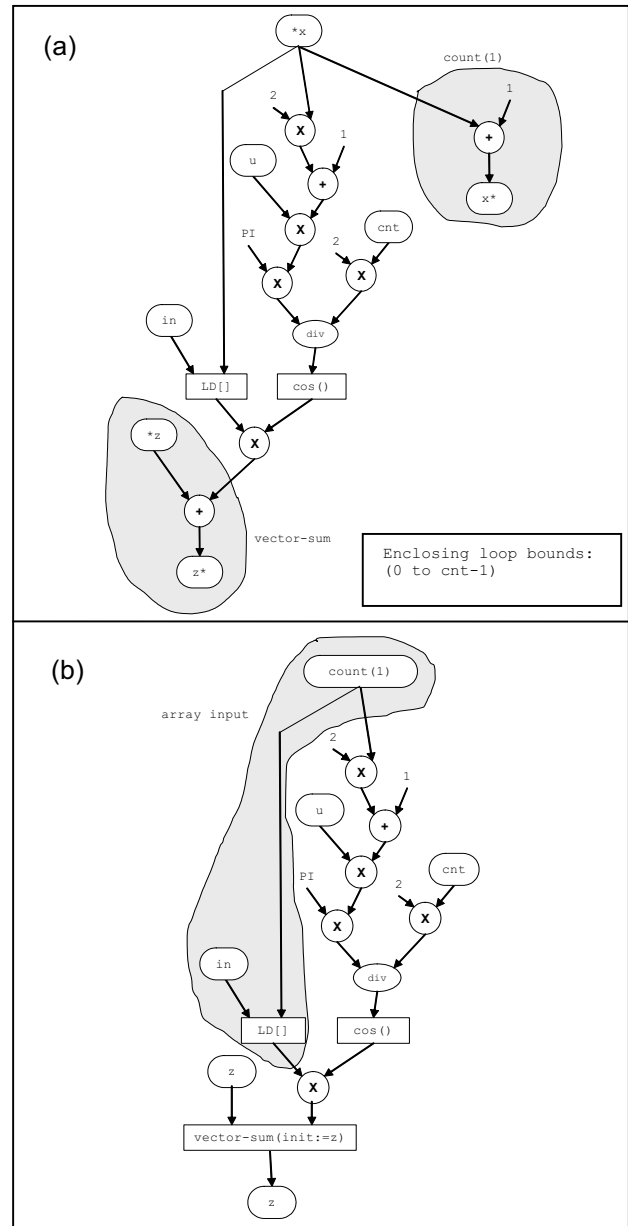


Figure 8. DCT recognition, part 1.

- Edge dimensions have been propagated from producers to consumers. Basic binary operations are interpreted as element-by-element array operations under our semantics, thus the scalar inputs to the multiply block can be expanded to vectors via propagation from the edge source (here, the in interface). Accordingly, the output of the multiply block and the input to the vector-sum block also receive the same dimensions.
- The gray region from (c) has been identified as the special pattern “index-mapped-vector-fn,” which encapsulates an arbitrary function over elements generated by a count(1) pattern, i.e., its elements are

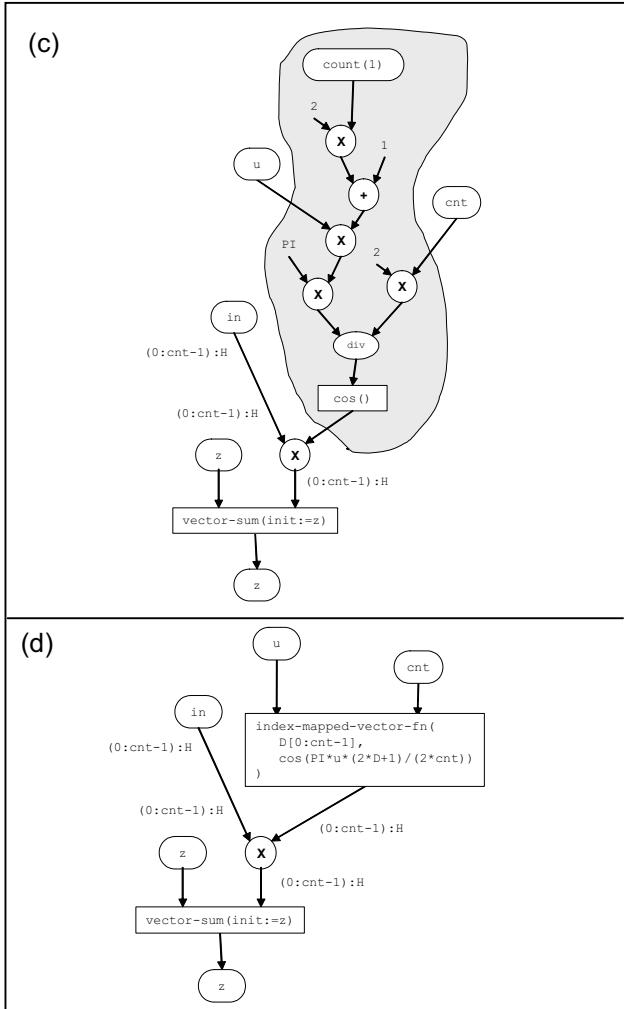


Figure 9. DCT recognition, part 2.

functions of the loop iteration. It is matched as a region containing only arithmetic nodes (or known functions, such as cosine) with a single count(1) input and an arbitrary number of input interface nodes.

When recognition has been applied to the entire DCT source code of the Appendix, the MDSDF-style specification of Figure 10 is produced. This specification was captured directly in an MDSDF model in Ptolemy Classic, as shown in Figure 11.

3. Validation

As a quantitative motivation for further development of this representation and its associated recognition system, we applied the recognition process by hand to several image processing algorithms, using the complete pattern library (including those described in section 2.3). The resulting specifications were used to generate code for SIMPIL, for Matlab, and for the Ptolemy Classic environment which can simulate MDSDF models.

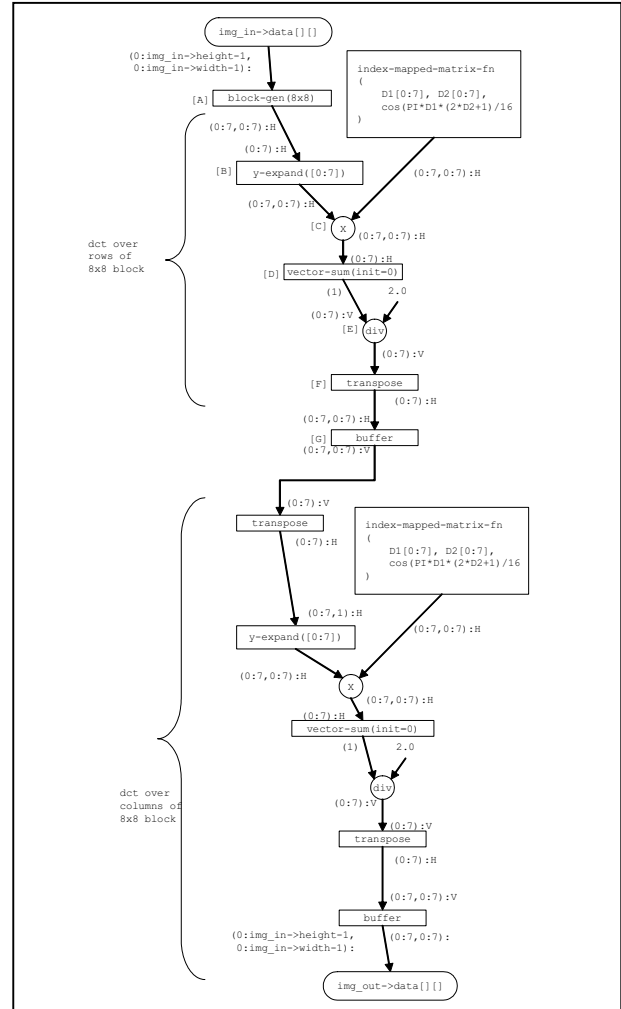


Figure 10. Final DCT specification in the extended MDSDF representation.

3.1. MDSDF/Ptolemy model creation

MDSDF models were constructed based on the graphical specifications (i.e., Figure 2 and Figure 10) using the graphical editor provided by the Ptolemy Classic tool. When needed nodes (such as vector-sum) were not available in Ptolemy Classic, they were built from existing primitives. Figure 11 shows a screen capture from Ptolemy Classic of the retargeted DCT. It includes annotations illustrating which collections of primitives implement certain nodes from the graph of Figure 10.

3.2. SIMD code generation from MDSDF

Strategies for generating SIMD (and, similarly, Matlab) code from MDSDF vary depending on the nature of the recognized program specification. Two classes of algorithm are described in this paper – pixel-level algorithms (convolution and thresholding) and region-level algorithms (DCT and quantization) – and each

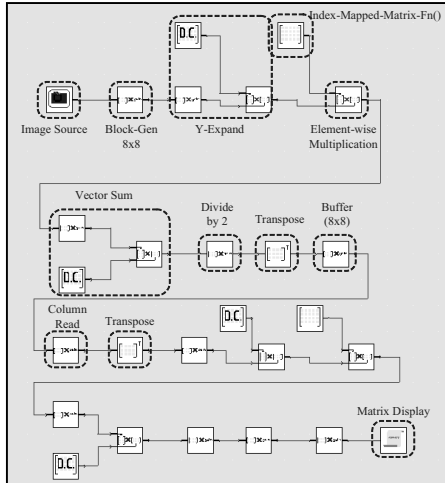


Figure 11. Retargeted DCT, specified from Figure 10, modeled in Ptolemy Classic.

requires a different SIMD mapping. The approach here is to map sequences of operations over a pixel in a pixel-level algorithm to a single processor in a SIMD array (all pixels operated upon in parallel). Similarly, sequences of operations over sub-regions of the image in a block-level algorithm are mapped to a single processor, i.e., all regions are operated upon in parallel.

Pixel-level code generation. In an MDSDF graph where all edge dimensions (but not necessarily bounds) are equal (or, in the relaxed case, off by one or two, to account for border conditions) are equal (e.g., Figure 2), the graph represents a pixel-level algorithm. Thus the graph can be viewed as a partial-ordering of scalar operations applied to a single pixel and executed on a single processor. In this situation, a depth-first traversal, beginning from the inputs, is sufficient to generate the proper sequence of program statements.

Region-level code generation. In the example of Figure 10, the graph performs a series of operations on 8x8 regions of the input image. Further, some of those operations in turn perform operations on sub-regions of the 8x8 blocks. If an MDSDF specification can be determined to be a region-level algorithm, the approach is to generate sequential code that performs the operations over the region, and then to map multiple copies of the sequential code to processors in the SIMD array such that all regions are operated upon in parallel.

If an MDSDF graph is suspected to be a region-level algorithm (i.e., it is not a pixel-level algorithm), a graph is constructed showing the number of times each operation node in the graph must execute, relative to the inputs. For part of the DCT graph of Figure 10, the sequence of Figure 12, using letter labels from the original graph, is constructed. Here, a directed edge specifies how many times its producer node must execute relative to its consumer node, e.g., $B \rightarrow A$ with a label of “8” indicates B executes eight times for every invocation of A. (The

index-mapped-matrix-fn node is not shown as its semantics dictate it only executes once and is constant for the remainder of the execution). To use region-level code generation, three conditions must hold: a sub-graph must be found with only one input node and one output node, they must have the same output dimensions, and they must only execute once. In other words, at the input to the sub-graph there must be only one region read in, and at the output, only one region of the same size written out. Inspection of Figures 10 and 12 show the first and second conditions hold. The third can be found by calculating the frequency of G with respect to A, which is the product of relative frequencies along a path from G to A. Inspection shows this frequency to be one, so the second condition is also satisfied.

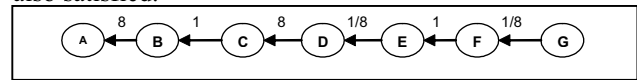


Figure 12. Relative execution frequencies of the DCT graph operation nodes.

Code generation proceeds from the graph of Figure 12. Beginning with A, traverse the graph toward G. If an edge label is an integer x greater than one, instantiate a new loop from 0 to $(x-1)$ and add the function defined by the next node to the body, matching array indices accordingly. If a label is a one, add the next node to the current loop body. If the label is a fraction, end the current loop, creating a vector of results if necessary, and place the next node after the loop. This produces the general structure of Figure 13 (a) which expands to the pseudo-code of Figure 13 (b).

<pre> A Repeat 8 times { B C Repeat 8 times { C } E F } G </pre>	<pre> a[0:7][0:7] = A(); for(i=0 to 7) { b[0:7][0:7] = B(a[i][0:7]); c[0:7][0:7] = C(b[0:7][0:7]); h[0:7][0:7]; // cos coefficients for(j=0 to 7) { t1 = D(c[j][0:7]); d[j] = t1; } e[0:7][0] = E(d[0:7]); t2 [0][0:7][i] = F(e[0:7][0]); f[j][0:7] = t2[0][0:7][j]; } h[0:7][0:7] = G(f[0:7][0:7]; </pre>
(a)	(b)

Figure 13. Pseudo-code for the generated DCT region-level SIMD code.

3.3. Results

Recognition was applied to four programs: the DCT and 3x3 convolution already discussed, an image thresholding algorithm (pixel values above a certain threshold are assigned a “high” value while those below the threshold are assigned a “low” value), and a block quantization algorithm (sub-blocks of the image are multiplied, element-by-element, with an array of quantization coefficients).

Table 1 compares the results of each program retargeted to Matlab and SIMPIL (except DCT; SIMPIL can simulate this program, but cannot output the results in

an appropriate data type, thus it was retargeted to Ptolemy instead) with those produced by a baseline sequential execution (i.e., compiled C executable) as an average difference of the results, defined as:

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

where a and b are the elements of the matrix results, with dimensions $M \times N$, of the sequential execution and retargeted simulation, respectively. Here, the average distance measure has units of grayscale pixel values over the range of 0 to 255 (8-bit values). In all cases but one, explained below, the average distance is quite small, indicating the retargeted program produces correct results within a reasonable margin-of-error.

Table 1. Average distance results.

Program	Sequential Baseline	Retargeted to Matlab	Retargeted (SIMPil/Ptolemy)
threshold	0	1.31	0 (SIMPil)
quantization	0	0	0 (SIMPil)
3x3 convolution	0	2.33	4.88 (SIMPil)
DCT (with border condition)	0	0.00001599	0.007370 (Ptolemy)
DCT (no border condition)	(baseline with border condition only)	24.72	24.72 (Ptolemy)

As Table 1 shows, the DCT was retargeted two different ways, with one – the average distance of 24.71 – producing unacceptable results. This large error was due to the omission of an iteration-dependent conditional (line 10 of the source code in the Appendix), which acts as a border condition for the `dct_1d` function. Such conditionals can make representation and recognition more difficult, so a two-tier recognition strategy is under investigation. The first, less costly, tier ignores certain conditionals (e.g., border conditions) that may have a minimal effect on the overall calculation; in the image processing domain, results that pass a qualitative test (e.g., a visual inspection) are often acceptable, even if minor variations exist from the original non-retargeted results. Tests are performed to check that program results are within an acceptable margin of those produced by the original sequential code. In Table 1, this pass produced the unacceptable results of the “DCT (no border condition)” row. Failing those tests, a second, more costly recognition pass requiring a larger pattern library and graph transformations (to concatenate irregular sub-regions created by the conditional into regular ones) is applied; for the “DCT (with border condition)” row, this brings the results within an acceptable error. Note that the other test programs succeeded recognition with only the simpler pass.

Table 2 compares the execution times of the retargeted SIMPil programs with that of the original sequential code, provided by execution on the SimpleScalar [20] simulator (sim-outorder). The calculated speedups are, for DCT and convolution, approximately double that of the expected speedups, due to architectural factors other than parallelism (instruction

set differences, inlining of function calls, memory architecture, etc.) that could not be normalized with the available tools. The lower-than-expected speedup for quantization is due to a loop optimization not present in the retargeted code as well as limitations in the code generation algorithm; both could be improved by applying standard loop transformations to the generated code. Regardless, in all cases, the retargeted code provides a clear benefit over sequential execution.

As Table 1 and Table 2 suggest, this representation and recognition process can provide, from sequential source code, accurate code for execution on SIMD processors, resulting in large potential speedup (i.e., $sequential\ run-time / SIMD\ run-time$) in execution times (as measured in clock cycles).

Table 2. Execution time comparison.

Program	SimpleScalar execution time (cycles)	SIMPil execution time (cycles)	# SIMPil processors (ideal speedup)	Speedup
DCT (border conditions)	178617705	88042	1024	2029
3x3 Convolution	6341485	57	65536	111254
Quantization	1116274	288015	16	4
Threshold	1905421	36	65536	52928

4. Conclusions

In this paper, we have presented a dataflow representation for explicitly representing data-parallel algorithms and have described the recognition process for extracting those algorithms. We have identified two issues to be addressed in future work:

Exclusive regions. Some programs may perform different operations on non-intersecting regions of an array, for example, border pixels of an image may be processed differently from the interior pixels. Some analysis will be required to either separate these calculations at code-generation time, or to unify the regions under some higher-level abstraction. The two-tier recognition strategy discussed in section 3.3 is being investigated for handling these regions.

Program coverage. What parts of the input program should be targeted? Should recognition be applied to the entire program, or is there a fast way to limit the analysis to likely candidate code? An approach would be to dynamically profile the original sequential code to measure its potential for data-parallel execution, such as the system described in [21]. The profiling results would then be correlated with the source code to target candidate program regions for recognition.

Acknowledgements

We are grateful to Sam Sander, Cameron Craddock, Chris Lee, Nidhi Kejriwal, 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 NSF CAREER Grant CCR-0092552.

The DCT code of the Appendix was written by Emil Mikulic and is available from <http://members.optushome.com.au/darkmoon7/code/dct/>. The convolution source code of Figure 1 is part of the Eclipse image processing suite distributed by the European Southern Observatory [22] and the quantization and thresholding test programs are part of the IMGLIB suite of applications for the Texas Instruments 'C6200 line of DSPs [23].

Appendix: DCT source code

```

01: void dct_1d(double *in, double *out, const int cnt) {
02:     int x, u;
03:     for (u=0; u<cnt; u++) {
04:         double z = 0;
05:         for (x=0; x<cnt; x++) {
06:             z += in[x] * cos(PI *
07:                 (double)u * (double)(2*x+1)
08:                 / (double)(2*cnt));
09:         }
10:         if(!u) z *= 1.0 / sqrt(2.0);
11:         out[u] = z/2.0;
12:     }}
13: void dct(const img *im, double data[8][8],
14:         const int xpos, const int ypos) {
15:     int i,j; double in[8], out[8], rows[8][8];
16:     for (j=0; j<8; j++) { /* transform rows */
17:         for (i=0; i<8; i++){
18:             in[i] = im_data[((ypos+j) *
19:                 im_width) + (xpos+i)];
20:         }
21:         dct_1d(in, out, 8);
22:         for (i=0; i<8; i++)
23:             rows[j][i] = out[i];
24:     }
25:     for (j=0; j<8; j++) { /* transform columns */
26:         for (i=0; i<8; i++)
27:             in[i] = rows[i][j];
28:         dct_1d(in, out, 8);
29:         for (i=0; i<8; i++)
30:             data[i][j] = out[i];
31:     }}
32: int main() {
33:     ...
34:     for (j=0; j<img_in->height/8; j++)
35:         for (i=0; i<img_in->width/8; i++) {
36:             dct(img_in, dct_out, i*8, j*8);
37:             for(n=0;n<8;n++) // copy-back added by lbb
38:                 for(m=0;m<8;m++)
39:                     img_out.data[(i*8+n)*
40:                         img_out.width+(j*8+m)];
41:         } ... }

```

References

[1] Alex Peleg, Uri Weiser, "MMX Technology Extensions to the Intel Architecture," *IEEE Micro*, Vol. 16, No. 4, pp. 42-50, August 1996.

[2] Srinivas K. Raman, Vladimir Pentkovski, Jagannath Keshava, "Implementing Streaming SIMD Extensions on the Pentium III Processor," *IEEE Micro*, Vol. 20, No. 4, pp. 47-57, July/August 2000.

[3] Tom Blank, "The Maspar MP-1 Architecture," *In Proc. of the 35th IEEE Computer Society Int. Conf.*, San Francisco, CA, February 1990, pp. 20-24.

[4] Lewis W. Tucker and George G. Robertson, "Architecture and Applications of the Connection Machine," *IEEE Computer*, Vol. 21, No. 8, pp. 26-38, August 1988.

[5] Huy H. Cat, Antonio Gentile, John C. Eble, Myunghee Lee, Olivier Vendier, Young Joong Joo, D. Scott Wills, Martin Brooke, Nan Marie Jokerst, April S. Brown, "SIMPil: An OE Integrated SIMD Architecture for Focal Plan Processing Applications," *Proc. of the Third IEEE Int. Conf. on Massively Parallel Processing using Optical Interconnection (MPPOI-96)*, Maui, Hawaii, October 1996, pp. 44-52.

[6] Praveen K. Murthy and Edward A. Lee, "Multidimensional Synchronous Dataflow," *IEEE Trans. on Signal Processing*, Vol. 50, No. 8, pp. 2064-2079, August 2002.

[7] Edward A. Lee, "Overview of the Ptolemy Project," Dept. of Electrical and Computer Engineering and Computer Science, University of California, Berkeley, Tech. Memo. UCB/ERL M01/11, March 2001.

[8] J. R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *TOPLAS*, Vol. 9, No. 4, pp. 491-542, October 1987.

[9] Hans Zima and Barbara Chapman, *Supercompilers for Parallel and Vector Computers*, New York: ACM Press, 1991, pp. 112-172, 218-238.

[10] D. Callahan, et. al., "Vectorizing compilers: a test suite and results," *Proc. Supercomputing '88*, pp. 98-105, November 1988.

[11] N. Sreeraman, R. Govindarajan, "A Vectorizing Compiler for Multimedia Extensions," *Int. Journal of Parallel Programming*, Vol. 28, No. 4, pp. 363-400, August 2000.

[12] Gerald Cheong and Monica Lam, "An Optimizer for Multimedia Instruction Sets," *In Proc. of the Second SUIF Compiler Workshop*, Stanford University, August 1997.

[13] Aart Bik, Milind Girkar, Paul Grey, Xinmin Tian, "Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems", *Intel Technology Journal*, February 2001.

[14] Richard C. Waters, "A Method for Analyzing Loop Programs," *IEEE Trans. on Software Engineering*, Vol. SE-5, No. 3, pp. 237-247, May 1979.

[15] Christoph W. Keßler, "Pattern-Driven Automatic Parallelization," *Scientific Programming*, Vol. 5, No. 3, pp. 251-274, Fall 1996.

[16] Beniamino Di Martino, G. Ianello, "PAP Recognizer: a Tool for Automatic Recognition of Parallelizable Patterns," *In Proc. of the 4th Int. Workshop on Program Comprehension (WPC '96)*, Berlin, Germany, March 1996, pp. 164-173.

[17] Beniamino Di Martino, Hans P. Zima, "Support of automatic parallelization with concept comprehension," *Journal of Systems Architecture*, Vol. 45, No. 6-7, pp. 427-439, 1999.

[18] Beniamino Di Martino, Christoph W. Keßler, "Two Program Comprehension Tools for Automatic Parallelization," *IEEE Concurrency*, Vol. 8, No. 1, pp. 37-47, January-March 2000.

[19] Lewis Baumstark, Jr., and Linda Wills, "Exposing Data-Level Parallelism in Sequential Image Processing Algorithms," *In Proc. of the 9th Working Conference on Reverse Engineering (WCRE '02)*, Richmond, VA, November 2002, pp. 245-254.

- [20] Doug Burger and Todd Austin, "The SimpleScalar Tool Set, Version 2.0," Univ. of Wisconsin-Madison Computer Sciences Dept., Tech. Report TR #1342, June 1997.
- [21] Wills, L., Taha, T., Baumstark, L., and Wills, S., "Estimating Potential Parallelism for Platform Retargeting," In *Proc. of the 9th Working Conference on Reverse Engineering (WCRE '02)*, Richmond, VA, October 2002, pp. 55-64.
- [22] Nicolas Devillard, "ESO C Library for an Image Processing Software Environment (eclipse)," *Astronomical Data Analysis Software and Systems X ASP Conference Series*, Vol. 238, pp. 525-528, 2001.
- [23] *TMS320C62x Image/Video Processing Library Programmer's Reference*, Texas Instruments Literature Number SPRU400, March 2000.