MICRO-SPECIALIZATION IN MULTIDIMENSIONAL CONNECTED-COMPONENT LABELING

By

CHRISTOPHER JAMES LAROSE

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

MAY 2014

Approved By:

Dr. Saumya Debray
Department of Computer Science

Abstract

Connected-component labeling is a graph algorithm where unique identifiers are assigned to connected subgraphs based on a given heuristic. The algorithm is used in computer vision to identify connected regions of pixels in binary images. In these applications, the connectivity heuristic is specified as an array called a *structuring element* that defines how individual pixels should be considered for adjacency. Micro-specialization is an optimization technique that involves specializing code by introducing runtime invariants into code generation. Targeting a specific class of routines for micro-specialization in a given program can have dramatic impacts on runtime performance.

An implementation of connection-component labeling is presented that utilizes micro-specialization in a number of procedures. The implementation is derived from a modification of the image labeling routines found in the popular scientific computing library, Scipy. The approach involves analyzing a number of routines for their candidacy for micro-specialization and producing a program that generates specialized versions of those routines. For a given structuring element, the code generator produces code that is highly specialized, and yields a runtime performance improvement of as much 8% as compared to the unspecialized version.

1 Introduction

Image labeling is the process of identifying disjoint regions of an input image. It is used in virtually all image processing applications [1]. Labeling is often integrated into image recognition systems, such as optical character recognition (OCR). In OCR, image labeling can be applied to digital images of handwritten language to discern individual letters from one another. To do this, image labeling is used to identify all connected groups of pixels in the image. Then, small connected components are discarded to reduce noise. Various properties of remaining components can be then analyzed to produce the sequence of recognized characters. Similar applications of image labeling exist throughout computer vision algorithms, making it an ideal candidate for performance research as it would have wide-ranging effects.

Image labeling can be described as a function of two inputs. The first is a binary image, an array composed of pixels that each take on one of two possible values. The second input is called a connectivity heuristic or structuring element. It is represented as a symmetric binary array of pixels such that its size in each dimension is three. These two input arrays can be of any dimension d such that $d \geq 2$, and they must be of the same dimension. The output of an application of image labeling is a partitioning of the set of pixels of the input image such that for two arbitrary pixels P and Q of the same equivalence class, there exists a sequence of pixels $p_0, p_1, \ldots, p_n$, where $p_0 = P$ and $p_n = Q$ such that each intermediate pairing of pixels is adjacent as defined by the structuring element[2].

Micro-specialization is an approach to optimization that involves identifying invariants in the code, and using the values of those invariants to generate specialized routines. Micro-specialization can be used, for example, to parameterize the generation of DBMS query code with information that can be re-introduced at compile time like tuple attribute byte-offsets [3]. Analogously, the entries of a structuring element in image labeling can be used to inform code generation to produce an image labeling program that will label an input image for that specific structuring element. The specialized program can take advantage of its knowledge of the structuring element in advance and perform less work.

Presented first is the classical approach for multi-dimensional image labeling for a general structuring element. What follows is a systematic way to identify opportunities for specialization and a description of the specialized code that can be produced at compile-time.

2 Background

An image labeling routine begins by first constructing a data structure to store the equivalence classes among pixels of the input image. An efficient solution[4] uses a single-dimensional array to annotate unions among equivalence classes as they are discovered on a first linear pass through the pixels of the input image. A second pass through the array is later used to join equivalence classes that are found later to describe the same class. The Scipy source code refers to this data structure as a *merge table* and the discussion henceforth will be consistent with this terminology.

A simple calculation is performed during this initialization phase to determine the number of vectors of the structuring element that need to be considered. The fact that every vector need not be analyzed is due to the symmetric property of the structuring element. Let N be this number of vectors.

$$N = \left\lfloor \frac{3^{d-1}}{2} \right\rfloor$$

where d refers to the number of dimensions of the structuring element. A structuring element of two dimensions is always a 3×3 matrix and the value of N is always just 1. This means that only the algorithm looks only at the first row of the structuring element. In the case that the structuring element is of 3 dimensions, N will be 4; for 4 dimensions, N is 13, and so on.

The algorithm considers each vector of the input image along a particular axis. (Reasonable implementations will read the vectors of the axis such that the elements in a vector are stored in contiguous memory). Figure 1 illustrates the LabelLine procedure which is invoked for each line of the input image. The procedure loops through each of the first N vectors $(\vec{s_0}, \vec{s_1}, \ldots, \vec{s_N})$ of the structuring element, performs some bounds checking, and calls another routine called LabelLineWithNeighbor. LabelLineWithNeighbor, illustrated in Figure 2, takes a particular vector of the structuring element, s_i, and iterates over the current input line while reading from the *previous* input line in order to identify

pairs of neighboring pixels. Upon finding such pairs, LabelWithWithNeighbor updates the global merge table to keep track of connected components.

Figure 1: Labeling a single line of input

```
1: function LABEL LINE(lineBuffer, neighborBuffer, (s⃗_0, s⃗_1, ..., s⃗_N))
2:     for all s⃗_i ∈ (s⃗_0, s⃗_1, ..., s⃗_N) do
3:         prev, adjacent, next ← s⃗_{i0}, s⃗_{i1}, s⃗_{i2}
4:         if not (prev or adjacent or next) then
5:             continue
6:         end if
7:         if not boundsChecksSatisfied() then
8:             continue
9:         end if
10:        labelLineWithNeighbor(lineBuffer, neighborBuffer, prev, adjacent,
    next, i == N)
11:    end for
12: end function
```

Figure 2: Labeling a single line of input for a specific vector of the structuring element

```
1: function LABEL LINE WITH NEIGHBOR(lineBuffer, neighborBuffer, prev,
    adjacent, next, last)
2:     for i ∈ (0, 1, ..., lineBuffer → length) do
3:         if lineBuffer[i] ≠ BACKGROUND then
4:             if previous or last then
5:                 lineBuffer[i] = merge(lineBuffer[i], neighbor[i - 1])
6:             end if
7:             if adjacent then
8:                 lineBuffer[i] = merge(lineBuffer[i], neighbor[i])
9:             end if
10:            if next then
11:                lineBuffer[i] = merge(lineBuffer[i], neighbor[i + 1])
12:            end if
13:        end if
14:    end for
15: end function
```

3 Approach

Traditional optimization approaches perform a sequence of transformations on a program to produce a semantically equivalent program that uses fewer resources. Micro-specialization is a dynamic code specialization technique that differs from

traditional techniques in that it is a form of specialization—programs produced by way of micro-specialization assume specific runtime conditions instead of applying an algorithm in the general case.

In the context of image labeling, a structuring element is provided as input to the procedure. Its properties, including its dimension, size, and values of individual entries are constant throughout execution. These properties are used throughout code in a tight loop, an especially performance-critical section of code. A program written to perform labeling given an *arbitrary* structuring element may incur costs associated with the fact that it cannot know these properties at compile time. It may, for example, iterate over elements of the structuring element that would ultimately have no effect on performing unions among input pixels.

A micro-specialized application of the same algorithm would instead perform image labeling on a *specific* structuring element. It can take advantage of information gathered about the structuring element at compile time to produce more efficient code. We apply micro-specialization in this fashion by first identifying variables whose values never change throughout the processing of every line of input. In Figure 1, we observe that the variables holding current and previous input vectors ($lineBuffer$ and $neighborBuffer$, respectively), change for every iteration, making them not suitable candidates for specialization. However, the rows of the structuring element over which we iterate, $(\vec{s_0}, \vec{s_1}, \ldots, \vec{s_N})$, never change. We can take advantage of this information at compile time.

To generate specialized code for a particular structuring element, it would be trivial to replace the variables $(\vec{s_0}, \vec{s_1}, \ldots, \vec{s_N})$ with their actual values as defined by the structuring element. This, however, would yield little value in our application, though, because our for loop in Figure 1 would still need to iterate over each of those vectors.

Consider that in addition to actualizing the values of the structuring element in the specialized code, we unroll the for loop of Figure 1. Every instance of the values $prev$, $adjacent$, and $next$ can be made known at compile-time. This means we can eliminate a number of statements entirely. Consider the conditional statement on line 4 of Figure 1. Because we now know the values of $prev$, $adjacent$, and $next$, we can move this check entirely to compile-time. That is, the unrolled loop does not need to include iterations where the entries of a particular vector are all 0.

For structuring elements of two dimensions, removing iterations of the loop will likely have no effect as N is just 1, and if the only row of the structuring element considered has all zeros, this has the meaning of "no connectivity" and would produce an output where each pixel formed its own equivalence class—an unlikely use case. For higher dimensions, however, removing these useless iterations can be beneficial. Consider the three-dimensional structuring element defining 6-connectivity in Figure 3. It specifies that pixels should be considered to be members of the same component if and only if they are orthogonally adjacent. For $d = 3$, $N = 4$, meaning an unspecialized loop would iterate over four vectors of the structuring element for every line of input. For the structuring element in Figure 3, two of those four vectors contain entries of

all zeros. Therefore, these iterations can be removed entirely in the specialized code.

Figure 3: A structuring element defining 6-connectivity in 3 dimensions.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Consider, too, that we inline the function call to $labelLineWithNeighbor$ on line 13 of Figure 1 in each unrolled iteration of the loop. As before, the values of *prev*, *adjacent*, and *next* are all known at compile time and we can remove the appropriate conditional statements beginning on lines 4, 7, and 10 of Figure 2.

Figure 4 shows the specialized *labelLine* procedure produced for the 6-connectivity structuring element in Figure 3. The two halves of the specialized procedure refer to the only two non-zero vectors (of the first $N = 4$) in the structuring element, s_1 and s_3. The generated code for the two vectors is almost identical with the exception that the latter contains an additional merge due to the fact that s_3 is the last vector of structuring element. The same effect is achieved in the non-specialized version on line 5 of Figure 2.

Figure 4: Specialized labeling of a single line of input

```
1:  function LABEL LINE(lineBuffer, neighborBuffer)
2:      // s_1
3:      if not boundsChecksSatisfied() then
4:          GOTO s_3
5:      end if
6:      for i ∈ (0, 1, ..., lineBuffer → length) do
7:          if lineBuffer[i] ≠ BACKGROUND then
8:              lineBuffer[i] = merge(lineBuffer[i], neighbor[i])
9:          end if
10:     end for
11:     // s_3
12:     if not boundsChecksSatisfied() then
13:         GOTO end
14:     end if
15:     for i ∈ (0, 1, ..., lineBuffer → length) do
16:         if lineBuffer[i] ≠ BACKGROUND then
17:             lineBuffer[i] = merge(lineBuffer[i], neighbor[i - 1])
18:             lineBuffer[i] = merge(lineBuffer[i], neighbor[i])
19:         end if
20:     end for
21: end function
```

4 Results

Careful consideration is made in the selection of test cases for the purpose of performance comparison. As discussed earlier, there is no expectation for any significant performance difference between specialized and unspecialized versions for structuring elements in two dimensions. For images of larger dimensions, however, loop unrolling is expected to have some impact because the number of considered vectors, N is greater than one. Lastly, we expect code generated for structuring elements with some all-zero vectors to require significantly fewer instructions. With these considerations in mind, we compare the running time of the unspecialized and micro-specialized versions of the connected component labeling algorithm using the 6-connectivity structuring element in three dimensions as illustrated in Figure 3.

To produce a comparison of the unspecialized and micro-specialized versions, we generated a sequence of 3-dimensional images, with side lengths from 2^3 to 2^9 pixels. For every input image, we ran both the specialized and unspecialized programs ten times and collected their execution times. All tests were conducted on late 2013 Apple MacBook Pro with a 2.4 GHz Intel Core i5 processor. Figure 5 offers these results.

Figure 5: Execution time comparison

Number of Pixels	Unspecialized (s)	Micro-specialized (s)	% Change
512	0.000450 ± 0.000070	0.000487 ± 0.000083	+7.60%
4096	0.000569 ± 0.000020	0.000553 ± 0.000024	-2.89%
32768	0.001402 ± 0.000008	0.001322 ± 0.000013	-6.05%
262144	0.008513 ± 0.000460	0.008081 ± 0.000703	-5.35%
2097152	0.060633 ± 0.003685	0.055856 ± 0.002754	-8.55%
16777216	0.467795 ± 0.012717	0.433987 ± 0.008930	-7.79%
134217728	3.805973 ± 0.028138	3.552283 ± 0.030928	-7.14%

For small inputs, the specialized code performs rather poorly, but for sufficiently large inputs, performance improvements vary from 2% to 8%.

5 Future Research

Although the method of specializing the *labelLine* routine discussed here did not produce significant performance improvement, it is possible that alternative strategies might yield more dramatic results. For example, consider inverting the *labelLine* and $labelLineWithNeighbor$ routines. As they are presented here, every line of input is processed multiple times to find adjacent elements, once for each of N vectors of the structuring element. Consider instead processing each input vector exactly once while considering adjacency as defined by *all* N structuring element vectors simultaneously. While this solution may not be suitable for a generic implementation, a micro-specialized implementation might process the input more quickly.

The work presented here demonstrates the potential for micro-specialization in connected component labeling. For a specific class of parameters, small, but reproducible, performance gains were shown to be possible. Further research into alternative specialization methods may yield more dramatic results.

References

[1] Samet, H. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence* Volume 10, Issue 4. July 1988.

[2] A. Rosenfeld and J. L. Pflatz. "Sequential operations in digital image processing". *J. ACM* Volume 13, pp. 471-494. October 1966.

[3] R. Zhang, S. Debray, and R. T. Snodgrass. Micro-Specialization: Dynamic Code Specialization of Database Management Systems. *Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, March 2012.

[4] L. Di Stefano, A. Bulgarelli. A simple and efficient connected components labeling algorithm. *International Conference on Image Analysis and Processing, 1999. Proceedings.* September 1999.