

# Efficient Image Processing with the Apply Language

Leonard G. C. Hamey  
*Macquarie University, Sydney, Australia*

## Abstract

*Apply is a Domain-Specific Language for image processing and low-level computer vision. Apply allows programmers to write kernel operations that focus on the computation for a single pixel location. The compiler generates code to perform the kernel computation over entire images. The original Apply implementation was developed 20 years ago for efficient processing on parallel architectures. The current-generation Apply compiler targets efficient code generation for general-purpose computers, typically outperforming handwritten code, while maintaining the simplicity of the original language. The use of modern compiler writing tools, specifically Stratego/XT, has facilitated improvements in the language design and made it easy to target the compiler to different environments. A large number of computer vision and image processing operations can be expressed in Apply. However, some algorithms require additional features. To motivate future language development, we analyse the requirements of the algorithms provided in a commercial machine vision library.*

## 1. Introduction

This paper discusses the development of a Domain-Specific Language for image processing and low-level computer vision called Apply. Apply simplifies the task of programming kernel-based algorithms – a common class of image-based computation where each output pixel is computed from a small neighbourhood of the corresponding input. Many image processing and low-level computer vision operations can be expressed as kernel-based algorithms. The Apply language allows programmers to focus on the computation surrounding a single pixel while the compiler generates efficient code to execute the computation over the entire image. Apply modules are thus easier to write and to understand than efficient

handwritten code, while executing as fast as the best handwritten code. The end-user calls the Apply module from their application program just as they would call other image processing library operations. The Apply compiler can be targeted to different environments, making Apply code more portable than modules written in a conventional high-level programming language such as C.

Apply originated in the 1980's at Carnegie Mellon University [5]. Apply allowed easy and efficient programming of image-to-image computations, which are implicitly parallel, by writing a procedure to be applied to a window around a single pixel of the image. The Apply compiler converted the programmer's code into specific implementations for a range of hardware platforms including both special purpose multiprocessor parallel architectures and general-purpose (uniprocessor) computers [5, 11, 12]. Each platform offered different programming languages and different models of parallel computation, which were targeted by modifying the Apply compiler.

The benefits of using Apply for targeting parallel architectures have always been clear. The implicitly parallel specification of the computation allows the compiler to efficiently target the parallel capabilities of the particular architecture, while the language itself simplifies programming and provides portability between architectures. However, the justification for using Apply to program general-purpose computers has not been as strong because the compiled code for general-purpose computers was considerably less efficient than the best hand-written implementation of the same algorithm. Apply code could be tested on general-purpose architectures, but programmers seeking efficient implementations would not choose the Apply language. The compiler technologies used to implement the Apply compiler (Yacc and Lex) were insufficient to support the optimisation of the generated code that would make Apply-generated code execute as fast as hand-coded modules.

```

procedure sobel(from: in window (-1..1, -1..1) of byte border 0,
               to: out window of byte)
is
  x, y : integer;
begin
  x := from(-1,-1) + 2 * from(-1,0) + from(-1,1) - from(1,-1) -
        2 * from(1,0) - from(1,1);
  y := from(-1,-1) + 2 * from(0,-1) + from(1,-1) - from(-1,1) -
        2 * from(0,1) - from(1,1);
  if x < 0 then x := -x; end if;
  if y < 0 then y := -y; end if;
  x := x + y;
  if x > 255 then x := 255; end if;
  to := x;
end sobel;

```

**Figure 1. Apply code for computing Sobel edge detection.**

With the current availability of program transformation tools such as Stratego/XT [10], the opportunity exists to further develop Apply into a simple but powerful programming language that generates highly efficient code for image processing and low-level computer vision on current and future general-purpose computers. Using Stratego/XT, we have completely re-implemented the Apply compiler. The new compiler is designed to be easily targeted to different application program interfaces (APIs) so that Apply-generated code can be integrated with existing libraries without compromising execution efficiency [2]. This makes the Apply language a good choice for programming image processing and low-level computer vision operations as the programmer obtains ease of programming, portability, efficiency and the ability to extend the capabilities of libraries that they already use.

The primary design goals for Apply have remained largely unchanged throughout its development history.

1. Provide for simple expression of kernel-based algorithms so that the programs are easy to write and easy to understand.
2. Retain the implicit parallelism of kernel-based processing so that the compiler can most efficiently map the computation onto the capabilities of the target environment.
3. Separate the expression of the kernel computation from the specific application of the computation so that Apply modules can easily be reused for new applications. For example, Apply modules do not specify the image dimensions.

4. Handle computations that overlap the borders of the image appropriately without unnecessary programming effort.
5. Provide efficient compiled implementations of Apply modules, removing the incentive for a programmer to hand-code the computations to obtain better performance.

While there are other DSL's that share many of these design goals (e.g. [1]), we are aware of none that share all the design goals of Apply.

In the sections that follow we discuss the Apply language as it is currently implemented, and provide a roadmap for future language features based upon an analysis of a commercial machine vision library.

## 2. The Apply Language

Syntactically, the Apply language is a subset of Ada [7] with some added special features. Apply provides the usual arithmetic and Boolean expressions and control-flow constructs including `if else` blocks, and `while` and `for` loops. The language provides `byte`, `real` and `integer` as primitive data-types, and allows the programmer to construct multidimensional arrays with index ranges. Apply procedures are interpreted as kernel operators. For example, figure 1 shows an Apply module that implements the well-known Sobel edge detector with 3x3 convolution masks [9]. For comparison, figure 4 is a hand-optimised C implementation of the same operator.

```

procedure roberts(from: in window (0..1, 0..1) of byte,
                 to: out window of byte)
is
  x, y : integer;
begin
  if defined(from(1,1)) then
    x := from(0,0) - from(1,1);
  else
    x := 0;
  end if;
  if defined(from(0,1)) and defined(from(1,0)) then
    y := from(0,1) - from(1,0);
  else
    y := 0;
  end if;
  if x < 0 then x := -x; end if;
  if y < 0 then y := -y; end if;
  x := x + y;
  if x > 255 then x := 255; end if;
  to := x;
end roberts;

```

**Figure 2. Using defined in Apply code for computing Roberts edge detection.**

Special features of Apply support kernel-based programming. The keyword `window` is used to declare a procedure parameter that represents a window of image pixels. A window can either hold a single pixel, such as the window `to` in figure 1, or a two dimensional array of pixels such as the window `from`. The index range syntax allows two-dimensional windows to have negative subscripts. By convention, element `(0,0)` of the window is the current pixel position, so the Sobel operator in figure 1 processes a 3x3 window around each pixel as expected.

Apply internally defines two read-only integer variables `row` and `column` which the programmer can use to determine the current pixel coordinates within the kernel operation. These variables are useful for simple image generation programs and also for low-level feature detectors that report the coordinates of image features.

Figure 1 demonstrates a simple form of border handling. The `border 0` modifier in the declaration of `from` succinctly describes how windows should be handled at the borders of the image. At the borders of an image, certain window locations may fall outside the image bounds. When this happens, Apply uses the constant value specified in the border modifier instead of taking a pixel value from the image. The constant value is often zero, as it is in this example.

The current Apply language defines several new language extensions including `defined` expressions and `assert` statements. The `defined` expression provides a more powerful way for programmers to express border handling. The `assert` statement is used by the compiler for optimisation and is key to the generation of efficient code. These and other extensions are discussed in [3].

The `assert expr` statement allows the programmer and the compiler to specify conditions that are known to be true concerning variables and expressions e.g. `assert x>=2;` indicates that the variable `x` is known at this point in the program to always have a value greater than or equal to two. Rarely, a programmer may use this statement to inform the compiler of known constraints on variables or expressions. More commonly, the `assert` statements are generated by the compiler to aid its own optimisations. However they arise, the compiler exploits the constraints expressed in `assert` statements to optimise code generation wherever possible. It is this optimisation capability that enables the Apply compiler to efficiently handle image borders, as discussed below.

The `defined(id(expr,expr))` expression tests whether a particular window pixel currently falls inside the image. If the window pixel falls outside the image, the `defined` expression evaluates to `false`. The programmer can use this expression to modify the computation performed near the borders of an image.

A	B	C
D	E	F
G	H	I

**Figure 3. Nine image regions.**

For example, figure 2 shows an implementation of the Roberts edge detector [9] where `defined` is used to check that the necessary pixel values of `from` are individually accessible. Compared to the simpler program using the `border 0` modifier, this program eliminates an artefact that produces edge responses at the bottom and right borders of the image. The current Apply compiler implements the `border` modifier using the `defined` expression.

The implementation of the `defined` expression is a Boolean expression that checks whether the pixel coordinates lie inside the image. This would be computationally expensive if the bounds checks were performed for every pixel in the entire image. The Apply compiler uses sophisticated looping constructs to process pixels close to the border of the image separately from pixels in the central portion of the image. There are nine regions of the image which each have different bounds check requirements, as shown in figure 3. For example, region A requires checks on pixels above and to the left of the centre of the window, while region B only requires checks on pixels above the centre of the window. The compiler generates separate loops for the different bounds-check requirements – up to nine loops are generated depending upon compile-time options. Compiler-generated `assert` statements are then used to optimise the generated code, eliminating unnecessary bounds checks. Examples of generated code can be found in [3].

The original compiler also provided border handling, but the capabilities were more limited. The original compiler used an interface library to access the image data. The interface library implemented border handling similar to the `border expr` modifier in the current compiler. The interface library dynamically extended the input image as it fetched data for the compiled Apply module to process. This approach simplified the compiler design, but reduced the efficiency of the code particularly when running on a uniprocessor machine since the generated code could not directly access image data stored in memory. The current compiler directly accesses image data in

**Table 1. Execution times for Sobel operator**

	Core gcc	PC gcc	SPARC gcc	Core MSVC	PC MSVC
Hand1	15.7	12.2	24.5	14.6	11.0
Hand2	5.23	3.61	4.67	5.19	4.07
Old	4.43	3.44	6.37	4.45	4.60
Speedup	15%	5%	-36%	14%	-13%
New	3.28	2.46	5.60	3.94	3.97
Speedup	37%	32%	2%	24%	2%

memory, resulting in a significant performance improvement. A performance comparison between the two Apply compilers is included in the next section.

The current Apply compiler also provides command-line options to allow the end-user to advise the compiler how to refine the generated code to target the intended application. One particularly useful option allows the programmer to specify the preferred image width(s). For images stored as arrays in memory, the compiler generates code that can handle arbitrary image dimensions but this code requires some address calculations for each pixel access. When the image width is known, the calculation can be simplified to a constant offset from the current pixel location, improving the execution performance significantly. Using the preferred image width(s), the compiler generates code to handle images of the specified width(s) with additional efficiency while also generating the usual code to handle other image widths.

### 3. Performance

The Apply compiler optimises the generated code to make it more efficient in the target environment. It is worth considering whether the compiler’s optimisation improve the execution speed of the final code, or whether the C compiler’s own optimisations may be sufficient. Our experiments with both hand-written code and generated code, reported below, demonstrate that some of the optimisations performed by the Apply compiler are not performed by the current-generation C compilers. Thus, the Apply compiler’s optimisations provide a significant performance benefit.

For performance comparison, we used the Sobel operator written in Apply (see figure 1) and as hand-written C code (figure 4). In table 1, ‘Hand1’ and ‘Hand2’ represent hand-written C code, while ‘Old’ and ‘New’ represent the original Apply compiler and the current Apply compiler respectively. Speedup is reported relative to the best hand-written C code.

```

/* in_bounds returns true when window pixel coordinates (i,j) fall inside */
/* the bounds of the input image. The current pixel position is (row,column) */
#define in_bounds(i,j) \
    (row+i >= 0 && row+i < height && column + j >= 0 && column + j < width)
/* FROM(i,j) computes access to 1-dimensional array "from" for window */
/* pixel coordinates (i,j) and current pixel position (row,column) */
#define FROM(i,j) from[(row+i)*width + column + j]
/* FROM_R(i,j) returns input pixel at window coordinates (i,j) if the */
/* coordinates are in bounds, otherwise return zero */
#define FROM_R(i,j) (in_bounds(i,j) ? FROM(i,j) : 0)
/* TO computes access to 1-dimensional array "to" for current pixel */
/* position (row,column) */
#define TO to[row * width + column]
void sobel(unsigned char from[], unsigned char to[],
           int height, int width)
{
    int row, column, x, y;

    /* Process the window positions that overlap the edge/corners */
    /* of the input image. Bounds checks are performed on all input */
    /* window accesses */
    for (row = 0; row <= height - 1; row++) {
        for (column = 0; column <= width - 1; ) {
            x = FROM_R(-1,-1) + 2 * FROM_R(-1,0) + FROM_R(-1,1) -
                FROM_R(1,-1) - FROM_R(1,1) - 2 * FROM_R(1,0);
            y = FROM_R(-1,-1) + 2 * FROM_R(0,-1) + FROM_R(1,-1) -
                FROM_R(-1,1) - FROM_R(1,1) - 2 * FROM_R(0,1);
            if (x < 0) x = -x;
            if (y < 0) y = -y;
            TO = x + y;
            if (column == 0 && row > 0 && row < height-1) {
                column += width-1;
            } else {
                column++;
            }
        }
    }

    /* Process the window positions that do not overlap the */
    /* edges/corners of the input image. Accesses to input pixels */
    /* are sped up because there is no need for bounds checking. */
    for (row = 1; row < height - 1; row++) {
        for (column = 1; column < width - 1; column++) {
            x = FROM(-1,-1) + FROM(-1,1) + 2 * FROM(-1,0) -
                FROM(1,-1) - FROM(1,1) - 2 * FROM(1,0);
            y = FROM(-1,-1) + 2 * FROM(0,-1) + FROM(1,-1) -
                FROM(-1,1) - FROM(1,1) - 2 * FROM(0,1);
            if (x < 0) x = -x;
            if (y < 0) y = -y;
            TO = x + y;
        }
    }
}

```

**Figure 4. C code for computing Sobel edge detection.**

It should be noted that the hand-written code is strongly optimised for performance using the same techniques developed for the Apply compiler. In particular, the Sobel computation is included at two places in the C source code – the first pair of nested loops processes the pixels near the borders of the image while the second pair of nested loops processes the pixels in the central portion of the image where border considerations are irrelevant. However, to test the optimisation capabilities of current C compilers, Hand1 contains unoptimised C code while Hand2 has been manually optimised as discussed below.

Performance tests were run with five combinations of CPU and C compiler. For each C compiler a range of common optimisation options were tested, with the best option used for comparison for each compiler on each platform [3]. In table 1, each measurement is the median time over 7 runs, each of which involved 30 seconds of execution time with randomised image array locations to avoid the impact of the memory cache and paging. Performance measurements were run over 512x512 images. The code generated by the current Apply compiler was specialised with a constant image width – the compiler generated special code to process 512 pixel-wide images especially efficiently, based upon a compile-time option as described above.

The systems tested were: PC Pentium 4 2.8 GHz (PC), Dell dual core Intel T2400 1.83 GHz processor (Core) and Sun-Fire-V490 with 2 x Ultra-IV 1.8GHz SPARC processors (SPARC). We used the GNU C compiler 3.4.4 (gcc) and Microsoft Visual C .NET 2003 (MSVC).

The differences between the two hand-written code versions were used to test the optimisation capabilities of the C compilers. In Hand1, the C source code performs image bounds checks on all pixel accesses, even when processing the central region of the input image (region 'E' in figure 3). In contrast, Hand2 only performs the bounds checks when processing pixels near the image borders, as shown in figure 4. The large timing differences between Hand1 and Hand2 are evidence that the C compilers' optimisations did not remove the unnecessary bounds checks in Hand1. The optimisation of Hand2 was performed by the programmer manually modifying the computation applied to the central region of the image. The Apply compiler automatically performs similar optimisations of the computation during the generation of the C source code. The compiler generates `assert` conditions based upon the loop bounds and uses this information to determine that the `defined` expressions always evaluate to true when processing the central region of the input image. These results

demonstrate one specific performance advantage of source code optimisation in the Apply compiler.

The results in table 1 show that the Apply generated code ran up to 37% faster than the best hand-written code. The results vary greatly between hardware platforms and C compilers. We also tested the performance of operators that had simpler code, and found that the speed-up achieved by the Apply generated code depended on the complexity of the Apply program [2, 3].

Performance results obtained with the original Apply compiler were not as good as the current Apply compiler. This reflects the simpler implementation of the original compiler. In particular, the original compiler accesses image data through an interface library, incurring a significant overhead that is most noticeable for simple computations such as the Sobel operator.

#### 4. Future development of Apply

The current Apply language supports efficient implementation of kernel-based computations. This encompasses a wide range of image processing and low-level computer vision computations. However, there are many well-known useful image computations that do not conform to the computation model currently supported. In order to evaluate the potential application of the existing Apply language, and to estimate the potential benefits of a range of possible extensions to the language, we analysed the requirements for implementing the image processing and low-level computer vision operations in a commercial machine vision library.

The analysis was conducted by examining the documentation provided with the library. Many operators in the library are well known in the literature while others include sufficient description in the documentation that it is possible to assess the language features that would be required to implement the operator in a language such as Apply.

The library we examined contains approximately 1200 distinct procedures. Of these, approximately 260 are image processing or low-level computer vision operations which we included in the analysis. The remaining 940 operations include algorithms to manipulate specialised data structures and system interface operations. Of the 260 procedures considered, 18 could not be classified and 25 were considered unsuitable for the Apply language because their computation model is very different than the Apply programming model. Table 2 presents the results of the analysis. It displays the portion of the approximately 260 operators in this machine vision

**Table 2. Coverage of Apply features.**

	<b>Without multi-pass</b>	<b>With multi-pass</b>
Apply current	40%	43%
Add reducers	48%	52%
Add variable kernel size	53%	57%
Add convolution mask calculation	56%	63%
Add image warping	63%	70%
Add raster recursive computation	67%	78%

library that could be implemented by an Apply compiler with different features. The display is cumulative, so that the introduction of each new feature increases the coverage up to a maximum of 78% for the features that we considered.

The current Apply compiler covers approximately 40% of the image processing and low-level computer vision operations. Other operations that cannot currently be programmed in Apply could be expressed if certain features were added to the language. Based upon the analysis of the existing machine vision library, we suggest the following enhancements to Apply.

**Multi-pass:** The current Apply language supports a single kernel operation in each module. A multi-pass version of Apply would allow multiple kernel operations to be written in a single module, and the programmer could execute the kernel operations sequentially to perform more complex image processing and computer vision algorithms. For example, the Harris corner detector [6] requires at least two passes of kernel operators – the first pass computes first derivatives and their products; the second pass applies a smoothing kernel to the results of the first pass and then computes the corner features. In addition to this simple type of multi-pass kernel processing, some operations require the ability to repeatedly apply the same kernel operations to an image – we have included this capability as a type of multi-pass operation.

In table 2, the multi-pass capability is shown as a separate column. This highlights the fact that multi-pass processing can combine passes that use different features of the Apply language. As the other features discussed below are added to the language, a multi-pass capability yields a greater benefit.

**Reducers:** Operations such as histogram computation require data to be accumulated over the entire image. This process can be efficiently parallelised, so it is suitable for the Apply compiler. Following Sawzall [8], we propose to add a data type mechanism for data reduction in Apply. A histogram can then be computed by emitting a data value of 1 to be added to the appropriate histogram bin. The compiler would implement this operation by incrementing the histogram bin directly. However, in a parallel implementation, multiple histograms could be computed for different parts of the image and then added together to compute the final result.

**Variable kernel size:** Most Apply operations involve a fixed kernel size. Indeed, some 41% of the operations considered in our analysis use only a single-pixel input kernel, e.g. gray level manipulation and histogram calculation. Despite this, a significant group of operations require a variable kernel size. This feature would mean that the end-user program invoking the Apply module would specify the desired kernel dimensions through a call parameter, and the Apply module would process the specified kernel size.

**Convolution mask calculation:** Many Apply operations involve convolution with small fixed masks, e.g. Sobel edge detection. Small fixed masks can be written as expressions in Apply (as in figure 1), but large masks and variable sized masks (e.g. an arbitrary sized discrete Gaussian filter implemented in the spatial domain) require calculation of the convolution mask within the Apply module. This computation is not parallel over an image – it is a type of precalculation that should precede the normal Apply computation over the image. Adding such precalculation capability to Apply could prove useful in other applications also.

**Image warping:** Kernel-based Apply programs operate on corresponding pixel locations in the input and output images. Operations such as image rotation and affine transformations cannot be expressed in this simple model. An image warping feature would allow the Apply programmer to explicitly calculate pixel coordinates in an input image and access the corresponding image pixel. Such an extension to the language should not be difficult to implement, and it is suitable for general-purpose computers, but a compiler targeting a parallel architecture would not be able to divide the image data across the processors as was originally done with the Warp Apply compiler [5] – the “warped” input image would require special handling of such architectures.

**Raster recursive:** In addition to strict kernel operations, there are a number of interesting image processing operations where the output at a particular pixel location is calculated on the basis of the output at one or more adjacent pixel locations, which must have been previously calculated. Examples of this type of algorithm include recursive filters, and image chamfering [9] for calculating distance transforms. Most of these operators can be implemented efficiently by processing the image in row or column raster order, either forward or reverse. Unfortunately, the pixel dependencies reduce the implicit parallelism from  $O(N^2)$ , where the entire  $N \times N$  image can be processed in parallel, to  $O(N)$ . It remains to be determined whether this feature can be supported while maintaining the portability and efficiency of Apply.

## 5. Conclusion

Apply is a DSL for image processing and low-level computer vision operations. Apply programs are portable across a variety of architectures and image library API's. We have re-implemented the Apply compiler using Stratego/XT, introducing new features for optimisation and for handling the borders of images. These features provide improved execution speed compared to the original compiler while increasing the expressive power of the language. Code generated by the current compiler outperforms the best hand-written code.

The current Apply language is sufficient for implementing approximately 40% of the image processing and low-level computer vision operations in a commercial machine vision library. Through analysis of such a library we have identified additional features that could be added to the language to increase the coverage to approximately 78%. This analysis provides a roadmap for the future development of Apply.

## 6. References

- [1] Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware", *ACM Transactions on Graphics*, **23**, 2004, pp. 777-786.
- [2] Hamey, L.G.C. and S.N. Goldrei, "Implementing a Domain-Specific Language Using Stratego/XT: an experience paper," in *Proceedings of the Seventh Workshop on Language Descriptions, Tools and Applications (LDTA 2007)* (A. Sloane and A. Johnstone, eds.), Braga, Portugal, Elsevier Science, 2007, pp. 32-47. To appear in *Electronic Notes in Theoretical Computer Science*.
- [3] Hamey, L.G.C. and S.N. Goldrei, *Implementing the Apply Compiler Using Stratego/XT*, Technical Report C/TR07-01, Department of Computing, Macquarie University, NSW, Australia, 2007.
- [4] Hamey, L., *Re-implementation of Apply: Stratego/XT experience notes*, unpublished, 2006.
- [5] Hamey, L.G.C., J.A. Webb and I.-C. Wu, "Low-Level Vision on Warp and the Apply Programming Model", in *Parallel Computation and Computers for Artificial Intelligence*, 1988, pp. 185-199.
- [6] Harris, C. and M. Stephens, "A Combined Corner and Edge Detector", in *Proceedings of the Fourth Alvey Vision Conference*, Manchester, 1988, pp. 147-151.
- [7] Ledgard, H., *Reference Manual for the ADA Programming Language*, Springer-Verlag, New York, 1983.
- [8] Pike, R., S. Dorward, R. Griesemer, S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall", *Scientific Programming*, **13**, 2005, pp. 277-298.
- [9] Sonka, M., V. Hlavac and R. Boyle, *Image Processing, Analysis, and Machine Vision*, Brooks/Cole, Pacific Grove, CA, 1999.
- [10] Visser, E., *Program Transformation with Stratego/XT*, Technical Report UU-CS-2004-011, Institute of ICS Utrecht University, 2004.
- [11] Wallace, R.S. and M.D. Howard, "HBA vision architecture: built and benchmarked", *IEEE PAMI*, **11**, 1989, pp. 227-232.
- [12] Weaver, G. and M. Scudder, *An Apply compiler for the CAAPP: Release 2*, Technical Report UM-CS-1994-065, University of Massachusetts, Amherst, MA, 1994.

## Acknowledgements

The author is grateful for helpful discussions with Shirley Goldrei and Tony Sloane.