An Apply Compiler for the CAAPP: Release 2

Glen Weaver and Michael Scudder
Computer and Information Science Department
University of Massachusetts *
Amherst, MA 01003
Phone: (413)545-1519
EMail: weaver@cs.umass.edu

September 23, 1994

Abstract

Apply is a partial language tailored for image processing. Apply defines image operations as pure functions that operate on a window of pixels. Because Apply is not a complete language, a host language is needed to actually control the execution of Apply functions.

Originally defined for systolic machines, Apply has been ported to the SIMD array (CAAPP) level of the IUA. This document describes the second release of Apply for the IUA. Besides supporting the full Apply language definition, release 2 now translates Apply code the into IUA C++ Class Library (ICL). ICL provides for debugging and serves as a natural host language.

This document serves as the user's guide for release 2. It discusses implementation decisions made in release 2 as well as how to use it to translate and compile Apply applications. This document also details the changes from release 1.

*This work was supported in part by the Advanced Research Projects Agency under contract DAAL02-91-K-0047, monitored by the Harry Diamond Laboratories; by the Office of Naval Research, under contract N00014-94-1-0742; and by the National Science Foundation, under grant number CDA-8922572.

Contents

1 What is Apply

Apply is an application specific programming language for image processing. It is a functional language that describes the operation to be applied to an image by describing the local computation required to produce a single pixel.

1.1 The History of Apply

Apply was developed at Carnegie Mellon University (CMU) as a tool for writing image processing programs in CMU's C/UNIX based standard vision programming environment. It now runs on UNIX systems, Warp systems, the Hughes HBA system, the IUA, and others. Apply was developed by Leonard G. C. Hamey, Jon A. Webb, and I-Chen Wu with contributions by Steve Shafer and others [2]. Extensions to Apply were considered as a result of experience from the 2nd Darpa Image Understanding benchmark exercise [7], but instead a new language, Adapt [8], was created. Though similar to Apply, Adapt exposes more of the architectural details of the Warp by specifying that functions are applied to pixels in raster order.

A new back end was added to the Apply compiler by Mike Scudder during 1989 [3] as a master's project. This back end produces code for the CAAPP level of the IUA architecture [9].

Mike Scudder's back end was modified during 1992 by Glen Weaver. The main enhancement was to translate the Apply procedures to C++ and IUA Class Library(ICL)[1] code instead of compiling to primitive forth instructions. Furthermore, the Apply implementation now matches fully the description in [2]. However, this implementation does not include the raster-order extensions suggested in [7].

1.2 Fundamental Concepts

Apply is intended to be an application specific but machine independent programming language. It provides a way to specify a pixel in an output image or images based on a window around the corresponding pixel in the input image or images. It thus provides for parallelism based on input partitioning.

Each operation is written as a procedure for a single pixel position. The Apply compiler translates this into a program which executes the procedure over the entire set of pixels. In

the original Apply no constraint on the order in which the pixels are processed is allowed by the language; thus, the system software has complete freedom in partitioning the pixels among processors.

1.3 The Language Specification

The language is described in [2]. The information in this document supersedes, for release 2 of the UMass implementation, the information in [3].

1.3.1 Syntax

The syntax of Apply corresponds to the syntax of a subset of Ada, with extensions to the procedure call syntax. This syntax is described in [2], with the exception that the type REAL is used instead of FLOAT. Apply is by no means an Ada subset, having keywords not in Ada and different semantics. Nor will it evolve into an Ada subset.

The acceptable syntax is modified by the m4 option. When the m4 option is used, the *Unix System V m4* macro preprocessor is called and processes the file according to its syntax rules before the Apply compiler's syntax analysis occurs. The option can be specified in the Apply code with a comment of the form:

```
-- Options: -m4
```

as the first line of the file.

1.3.2 Programming Model

When using the Apply language, the programmer writes a procedure which defines the operation to be applied to the current pixel location. The procedure conforms to the following programming model:

1. It accepts a rectangular window from each input image.
2. It performs arbitrary computations without the possibility of side-effects.
3. It returns a rectangular array of pixels in each output image.

.	.	$221,0$	$221,1$	$221,2$	$221,3$
.	$-1,-3$ B	$-1,-2$ $222,0$	$-1,-1$ $222,1$	$-1,0$ $222,2$	$222,3$
.	$0,-3$ B	$0,-2$ $223,0$	$0,-1$ $223,1$	$0,0$ $223,2$	$223,3$
.	$1,-3$ B	$1,-2$ $224,0$	$1,-1$ $224,1$	$1,0$ $224,2$	$224,3$
.	.	$225,0$	$225,1$	$225,2$	$225,3$

Table 1: An Example Input Window

1.3.3 Semantics

Each procedure has a parameter list containing parameters of any of the following types: *in, out,* or *constant.* Input parameters are either scalar variables or two-dimensional arrays. A scalar input variable represents the value of an input image at the current processing coordinates. A two-dimensional array input variable represents a window of an input image. Element (0,0) of the array corresponds to the current processing coordinates. These parameters have an associated border value which is used to fill in windows which extend outside an input image. This border value defaults to zero if not specified.

Table 1 shows the input window $(-1..1,-3..0)$ around pixel $223,2$. The top entry in each cell is the index within the procedure's window; the bottom entry is the index within the image array. The symbol B stands for the specified border value. The dots stand for pixels beyond the procedure's window and outside the image array.

As with input parameters, output parameters may be either scalar or two-dimensional values. A scalar output parameter represents a single pixel value of an output image. A two-dimensional output parameter represents several pixel values in the *same* output image. Thus, the output image would be larger than the number of times the Apply function is applied (which is normally equivalent to the size of an input image). The final value of an output variable is stored in the output image at the current processing coordinates.

Constant parameters may be scalars or arrays with an arbitrary number of dimensions. They represent precomputed constants, such as a convolution mask, made available for use

Apply Types	Type on ACU	Type on CAAPP
byte	char	CharPlane
unsigned byte	unsigned char	UCharPlane
integer	short	ShortPlane
unsigned integer	unsigned short	UShortPlane
real	float	FloatPlane

Table 2: Translation of Apply to C++ data types.

by the procedure.

Each procedure may also have local variables which may be scalars or arrays. Each processing location has, conceptually, its own set of these variables. The Apply compiler may choose, when permissible, to store some variables on the array controller. Variables used as FOR loop indices must have the same value at each processing location and are actually stored only once. Direct assignment to these variables is not allowed. Note that Apply FOR loop indices must be declared at the beginning of a procedure and have scope global to the whole procedure, unlike FOR loop indices in Ada.

Apply does not provide subprocedures or user defined functions. Apply does not provide complex data types other than the windows described above and statically defined arrays.

The reserved variables ROW and COL are defined to contain the image coordinates of the current processing location. This is useful for algorithms which are dependent in a limited way on the image coordinates. The coordinates start at (0,0), which corresponds to the upper left hand corner of the image array.

Apply does not allow assignment of real expressions to fixed variables, or assignment of fixed expressions to real variables. Apply automatically converts expressions of mixed fixed and real types to real expressions.

Variable names are alpha-numeric strings of arbitrary length, commencing with an alphabetic character. Case is not significant, except in the optional preprocessing stage.

The Apply compiler converts data types according to Table 2. Check the IUA Class Library documentation for further information on data representations.

2 The Apply Compiler

The Apply compiler translates programs written in Apply to C++ code with calls to the IUA Class Library[1]. The translated Apply program is compiled again by the C++ compiler and linked with the IUA Class Library. To distinguish between the two separate compilation steps, the conversion of Apply to C++ is referred to as translation.

The Apply compiler runs under Unix [5], making use of Lisp [4], Gnu C++, and various Unix utilities [6].

The Apply compiler consists of a front-end and a back-end, and both have multiple phases. A main program sets up the file input/output and calls each of the phases in turn. The front-end consists of the phases: option processing, macro preprocessing, syntax analysis, parsing, and semantic checking. The back-end consists of two phases: parser tree massaging and code generation.

Though the top level control is written in Lisp, the phases are composed of various Unix utilities and Lisp code. The option processing phase is written in LEX and YACC, and the compiler calls the M4 Unix utility to handle macro preprocessing. The syntax analysis phase is written in LEX; the parser is written in YACC. The semantic checking phase and the entire back-end (i.e., the parser tree massaging phase and the code generation phase) are written in Lisp.

3 How to use and maintain Apply at UMass

3.1 Using Apply, release 2

3.1.1 Setting up an account to use apply

- Apply is currently available on slotnick@cs.umass.edu, a Sun Sparc2.
- Set up an alias to point to "~apply/release2/apply".

  Aliases are usually set up in either the .login or .cshrc file. The line should look something like the following:

  ```
  alias apply ~apply/release2/apply
  ```

  This is all that is needed for basic Apply use. Executing 'apply' invokes the Apply compiler. In reality, 'apply' is just an executable shell script which invokes a make file

which eventually runs the Apply compiler.

- The "~apply/release2/make_apply_environment" shell procedure may be invoked to copy the 'apply' make file from the Apply account to your own.

  A copy of the make file is only needed for sophisticated applications or as a model for linking apply routines into a larger application.

3.1.2 Compiling an Apply Program

- An Apply procedure is written using a text editor.

  File names should end in ".app".

- Use the alias defined in section 3.1.1 to compile the Apply application.

  The compilation of an Apply application involves several steps. The original Apply application is translated to C++ code, which is compiled to object code which is finally linked.

  The shell script, 'apply', (and the make file which it calls) takes several parameters:

  Application Name (*Required*) This is the file name of the source file without the suffix. The shell script automatically prefixes its first parameter with "APP=", but this prefix would be required if the make file were used directly.

  Function (*Optional*) This tells the make file what function to perform. The following are the valid values:

  - `build`: This is the default. This will translate, compile, and link the application.
  - `translate_and_compile`
  - `compile_and_link`
  - `translate`
  - `compile`
  - `link`

  Configuration (*Optional*) The hardware configuration for which the Apply compiler should produce code. Currently this can be either `Sequential` or `Iua`. The default

is `Sequential`. Either of these values may be used as the second or subsequent parameters to 'apply', but it must be prefixed with "CONFIG=".

Debugging (*Optional*) The Apply compiler will insert debugging code into the C++ source, but the execution of this code depends on the definition of the DEBUG macro. Providing "DBG=U" will turn off the debugging code; whereas, "DBG=D" will compile in the debugging code, see section 3.1.3. "DBG=U" is the default, and this parameter may be specified as the second or following parameter to 'apply'.

3.1.3 Customizing the .cc file

The Apply compiler produces a C++ source file as its output. This source file contains standard C++ code, Apply specific function calls, and IUA Class Library function calls. This file may be modified with any text editor, which is useful for debugging, see section 3.1.5.

Appendix A shows an example of translated Apply code. All translated routines have a similar structure.

1. Beginning of translated Apply Procedure
2. Declaration of debugging display update macro.
3. Declaration of local variables
4. Buffering of input plane windows into each PE's local memory. This also implements the Sample option.
5. Initial display planes for debugging.
6. Translated procedure body.
7. Copying of resulting planes to the output planes. This also implements the magnification capability for output planes.
8. End of translated Apply procedure
9. The **main()** function which is provided for debugging.

3.1.4 Interfacing with an Apply procedure

Apply itself is not a complete computer language; it is not possible to describe an entire program in Apply. Instead, Apply just describes subroutines which must be called from a driver program written in some other language. In the UMass release 2 implementation, the natural language choice is C++, but any language which can call Gnu C++ functions and work with the IUA Class Library may be used.

For each Apply procedure, the Apply compiler produces a separate C++ function. Thus when the driver program wants to invoke an Apply procedure, it simply calls a C++ function of the same name as the Apply procedure. Constant parameters are regular C++ variables (or constants); whereas, images are IUA Class Library planes.

3.1.5 Debugging Apply Procedures

The Apply compiler catches syntax and some semantic errors, and it does not generate any programs the C++ compiler will reject. However, no array bounds checking is provided in the language or by the C++ compiler. Apply reports some run-time warnings and errors. But because Apply procedures are translated into IUA Class Library code, they may be debugged using the version of 'gdb' provided with the IUA Class Library.

The Apply compiler embeds debugging code in the C++ source code. The compilation of this code is controlled by a C++ compiler switch. The debug code does two basic things. First, it provides a macro to display various planes and calls that macro at various points during execution of the procedure. Secondly, a C++ **main()** function is provided which declares appropriate plane variables, accepts arguments from the command line, and displays the input and output of the procedure. This is especially useful when debugging, because Apply cannot generate an entire program of its own. However, the actual call to the Apply procedure is not included in the **main()** function. The actual function call must be hand coded. Also if the Sample option or output arrays are used, the plane sizes declared in **main()** may have to be adjusted.

3.2 Maintaining the Apply system

3.2.1 The directory structure

Apply currrently has its own user account. All of the release 2 files have been collected under a single subdirectory, ~apply/release2. Under this directory, the files are divided as follows:

Compiler ~apply/release2/compiler

This directory contains all the files which form the Apply compiler.

Documents ~apply/release2/docs

Any release 2 documents, such as this one, are located here.

Environment ~apply/release2/environment

This directory contains files needed by an Apply application, but which are not inherently part of the translation process.

Examples ~apply/release2/examples

This directory contains example apply applications, see 3.3.

Shell Scripts ~apply/release2

This directory contains the shell scripts which an Apply user will need to use.

3.2.2 Compiling the Apply Compiler

Compiling the Apply compiler is managed by a make file in ~apply/release2/compiler.

3.3 A complete example

Appendix A contains a complete example of the steps required to compile and execute an Apply procedure to smooth an image. The file ~apply/examples/smooth.app exists on the system, and the appendix can be used as a tutorial.

4 New for Release 2

This section describes the changes from release 1 to release 2.

4.1 Location of Variables

Apply was originally created to run on the Warp, which is a systolic processor. It was not designed for a SIMD mesh, such as the CAAPP. Thus, Apply syntax does not distinguish between local variables which need to be stored on the SIMD array and those which can be stored on the array controller.

Release 1 of Apply at UMass placed only for-loop variables and constant procedure parameters on the array controller. All other variables were planes. Release 2 has been enhanced to determine when a variable is assigned only ACU values. If the variable is never assigned a plane value, then the variable is moved to the ACU. Otherwise, it is kept on the CAAPP.

4.2 IUA Class Library Code Generation

The most significant change from release 1 to release 2 is that the Apply compiler now produces C++ code with IUA Class Library subroutines. Thus, the IUA Class Library is required in order to run a translated Apply procedure.

Any debugging or display capabilities needed are provided through the IUA Class Library. Also, if necessary a translated Apply procedure can be augmented with IUA Class Library subroutines by editing the C++ source output by the Apply compiler.

4.3 Image Reduction and Image Buffering

Release 2 implements the Sample function which is part of Apply. Sample causes the Apply procedure to be applied to only a subset of image pixels. The subset is a regular pattern across the image, such as every third pixel. However, the nearest neighbor pixels are still considered to be the nearest neighbors from the original image.

Sample is handled when the input image windows are buffered. The UMass implementation reads all the pixel values for an input image window into each PE before beginning the body of the procedure. This uses more memory, but gives better performance. Thus, Apply uses the whole image as input when building window buffers, but window buffers are built only at sampled pixels.

4.4 Image Magnification

Release 2 implements the output array capability of Apply. This implies that the Apply procedure is applied to fewer pixels than are present in the output image. Release 2 handles this by using an array of planes to hold the output results until the end of the procedure. Then the arrays are expanded to fit into a single output image (of a larger size).

4.5 Planes as Indices

Release 2 allows plane variables to be used as array indices. However, they may only be used to index constant parameters. Constant parameters are maintained on the array controller. When a constant array is indexed by a plane variable, each element of the constant array is broadcast down to the processor array. Each processing element in the array uses its local index values to determine which broadcast value is appropriate for it.

4.6 Debugging Support

Release 2 of the Apply compiler inserts debugging code into every C++ source file. See section 3.1.5 for more information.

4.7 Unsigned Integer Planes

Release 2 permits planes of type *unsigned integer*.

4.8 Virtualization

Release 2 supports virtualization. Virtualization is the ability of the physical machine to handle image sizes larger than the array.

References

[1] James Burrill. The Class Library for the IUA Tutorial. Amerinex Aritifical Intelligence, INC. *Unpublished*, 1992.

[2] Leonard G. C. Hamey, Jon A. Webb, and I-Chen Wu. Low-Level Vision on Warp and the Apply Programming Model. *Carnegie-Mellon University technical report* CMU-RI-TR-87-17, 1987.

[3] Michael Scudder. An Apply Compiler for the CAAPP. *UMass Technical Report 90-60*, 1990.

[4] Sun Microsystems. Sun Common Lisp 3.0 User's Guide. *Sun Microsystems Part No.* 800-3046-10, 1988

[5] Sun Microsystems. Getting Started *with* UNIX: Beginner's Guide. *Sun Microsystems Part No.* 800-1284-03, 1986.

[6] Sun Microsystems. Programming Utilities for the Sun Workstation. *Sun Microsystems Part No.* 800-1301-03, 1986.

[7] Jon A. Webb and Mike B. MacPherson. The Second DARPA Image Understanding Benchmark on WARP and extending Apply to Include Global Operations. *Proceedings of the May 1989 Image Understanding Workshop*, pp. 597–616, 1989.

[8] Jon A. Webb. Steps Toward Architecture-Independent Image Processing. *Computer*, Vol 25, No. 2, Feb. 1992, pp. 21-31.

[9] Charles C. Weems, Steven P. Levitan, Allen R. Hanson, Edward M. Riseman, J. Gregory Nash, and David B. Shu. The Image Understanding Architecture. *University of Massachusetts COINS Technical Report* 87-76, 1987.

A An Example: Smoothing an Image

A.1 The Apply Code

```
procedure smooth( inimg: in array (-1..1, -1..1)
                          of byte border 128,
                          outimg: out byte)
is
  sum, i, j: integer;
begin
sum := 0;
  for i in -1..1 loop
      for j in -1..1 loop
          sum := sum + inimg(i,j);
      end loop;
  end loop;
  outimg := (sum + 4) / 9; -- 4 is added to round result
end smooth;
```

A.2 On the command line

```
alias apply ~apply/release2/apply
apply smooth translate
<need to compile smooth.cc and link in with a driver routine>
```

A.3 The smooth.cc file

```
#include <math.h>
#include <IuaC.h>
#include <IuaClassLib.h>
#include <IuaUMassLib.h>
#include <apply_lib.h>
#include <apply.h>
#ifdef DEBUG
#include <stdio.h>
#include <stdlib.h>

extern "C" void Pause(char *msg);
#endif

/* Functions to index acu array by plane. */
void smooth(UCharPlane &INIMG_plane, UCharPlane &OUTIMG_plane)
{
#undef APPLYDisplayUpdate
#ifdef DEBUG
```

```
#define APPLYDisplayUpdate() \
   { \
      int i,j,r,c; \
      OUTIMG.Update(); \
   }
#else
#define APPLYDisplayUpdate()
#endif

   PlaneSize APPLY_PLANE_SIZE(64,64);
   ShortPlane SUM(APPLY_PLANE_SIZE);
   int I;
   int J;
   UCharPlane INIMG(64,64)[9];
#define INIMG_offset 4
#define INIMG_aref(i,j) INIMG_offset+(3*(i))+(1*(j))
#define INIMG_border 128
   UCharPlane OUTIMG(APPLY_PLANE_SIZE);

   /* Collect input image windows into each PE. */
   bufferWindow(INIMG_plane, UCharPlane, INIMG, -1, 1, -1, 1, 1,
                1,INIMG_border);

#ifdef DEBUG
   {/* Display images. */
      int i,j,r,c;
      char buf[80];
      INIMG_plane.Display("INIMG",APPLY_DISP_ROW,APPLY_DISP_COL);
      for (i=0,r=-1; r<=1; i++,r++) {
         for (j=0,c=-1; c<=1; j++,c++) {
            sprintf(buf,"INIMG[%d][%d]",r,c);
            INIMG[(i*3)+j].Display(buf,APPLY_DISP_ROW,
                                   APPLY_DISP_COL);
         }
      }
      OUTIMG.Display("OUTIMG",APPLY_DISP_ROW,APPLY_DISP_COL);
   }
#endif

   APPLYfile = "smooth.app";
   APPLYname = "SMOOTH";

   /* Translation of apply procedure body */
   ln=8;APPLYDisplayUpdate(); /* assignment statement */
   SUM = 0;
```

```
   ln=9;APPLYDisplayUpdate(); /*for loop*/
   for (I = (-1); I <= 1; I++) {
     ln=10;APPLYDisplayUpdate(); /*for loop*/
     for (J = (-1); J <= 1; J++) {
       ln=11;APPLYDisplayUpdate(); /* assignment statement */
       SUM = (SUM + INIMG[INIMG_aref(I, J)]);
     } /* End for-loop */
   } /* End for-loop */
   ln=14;APPLYDisplayUpdate(); /* assignment statement */
   OUTIMG = ((SUM + 4) / 9);
   /* End translation of apply procedure body */


   /* Process the magnified planes. */
   APPLYDisplayUpdate();
   OUTIMG_plane = OUTIMG;

#ifdef DEBUG
   Pause("End of SMOOTH.");
#endif
}

#ifdef DEBUG
main(int argc, char *argv[])
{  Everywhere active;

   UCharPlane input_INIMG(64,64);
   UCharPlane output_OUTIMG(64,64);

   if (argc != 2) {
      fprintf(stderr,"usage: %s <1 planes>\n", argv[0]);
      exit(1);
   }

   input_INIMG.Read(argv[1]);
   input_INIMG.Display("INIMG",APPLY_DISP_ROW,APPLY_DISP_COL);

   Pause("About to execute body of program.");
   /*Put actual function calls here.*/
   output_OUTIMG.Display("OUTIMG",APPLY_DISP_ROW,APPLY_DISP_COL);

   Pause("About to end.");
} /* End main() */
#endif
```