VSIPL: AN OBJECT-BASED OPEN STANDARD API FOR VECTOR, SIGNAL, AND IMAGE PROCESSING

Randall Janka
Cadence
Design Systems
Atlanta, GA

Randall Judd
U.S. Navy
SPAWAR Systems Center
San Diego, CA

James Lebak
MIT Lincoln
Laboratory
Lexington, MA

Mark Richards
Dan Campbell
Georgia Tech
Research Institute
Atlanta, GA

ABSTRACT

VSIPL, the Vector, Signal, and Image Processing Library, is an open standard application programmer's interface (API) for signal and image processing. Defined by a consortium of industry, government, and academic representatives, VSIPL is gaining widespread acceptance as a *de facto* standard in the embedded signal processing world. The primary goal of the API is to increase the portability of vector signal processing, matrix signal processing, and image processing applications. In this paper, we present an overview of the design, features, and availability of the VSIPL API.

1. INTRODUCTION

In the application domain of real-time embedded signal processing systems based on commercial off-the-shelf (COTS) hardware and software, an increasingly important goal is portability and vendor-neutrality of application software. The Vector, Signal, and Image Processing Library (VSIPL) Forum is a consortium comprised of industry, government, and academia that is developing the VSIPL API. The API is intended to serve as an open, vendor-neutral, industry-standard interface to vector arithmetic, signal and image processing operations for users of COTS workstations and embedded signal processing products.

Since the release of the completed VSIPL 1.0 API specification in April 2000 [1], the standard is becoming more widely adopted by commercial vendors. In addition, extensions to the standard are planned that correspond to current growth areas in military embedded signal processing. In this paper, we will review the VSIPL API, paying special attention to supported data types, functionality, and the logic behind the interface design. Commercial implementations and future extensions will also be discussed.

The authors were sponsored by the U.S. Navy under a series of U.S. Air Force and General Services Administration contracts. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the Unites States Air Force, Navy, or General Services Administration.

2. THE VSIPL FORUM

VSIPL began under sponsorship from the U.S. Defense Advanced Research Projects Agency's Information Technology Office. David Schwartz of Hughes Research Laboratory (now HRL, LLC) was the initial principal investigator and Chair of the VSIPL Forum. The Navy's Tactical Advanced Signal Processor (TASP) Common Operating Environment (COE) program then joined the effort and has continued as the sponsor in recent years. DARPA and the Navy shared a common goal to develop computational and communication middleware that freed application software from being "hard-wired" to a specific vendor.

Vendor neutrality was obtained by forming the VSIPL Forum to bring in stakeholders from industry, universities, laboratories, and user organizations to define the VSIPL API. The primary forum participants from the user community were interested in radar and sonar applications. The primary vendors were high-end signal processing board vendors, although both chip vendors and workstation vendors also participated in the library design. The most notable contributions to the API were made by representatives from HRL, SPAWAR, Georgia Tech Research Institute, CSPI, Mercury, SKY, SGI Cray, Lockheed Martin Naval Electronics and Surveillance Systems, Mississippi State University, MIT Lincoln Laboratory, and the Naval Air Warfare Center.

3. DESCRIPTION OF THE VSIPL API

The primary focus of the VSIPL effort was to provide portability for signal processing applications between different embedded computing vendors. Portability to workstations and to other architectures such as field-programmable gate arrays was also sought. This portability has several important benefits. It allows signal processing systems an easier upgrade path to future hardware. Development cost is reduced as code is developed and debugged on a workstation and expensive embedded hardware is only necessary for the final phases of application development. Training costs are reduced since there is no need to learn a new environment for each product. A more familiar debugging environment will reduce application development time.

3.1. General characteristics

Memory management was seen by the forum as a primary obstacle to portability. To solve this problem, the forum decided to use an *object-based* API. The object-based nature of the API provides two important benefits. First, the user is freed from details of managing the memory alignment issues for a particular architecture, resulting in more portable code. Second, vendors are able to optimize the implementation of VSIPL objects and the algorithms used for their particular systems. Such optimizations may include data alignment or the use of special memory areas such as on-chip data memory in digital signal processing chips. Overall, the decision to use an object-based API greatly reduces the burden on the application programmer.

Although VSIPL supports many data types, including many integer types, it is primarily a floating-point library. No fixed-point functionality is currently included, and integer support is limited to elementwise functions.

The VSIPL specification defines two modes of operation. In *development mode*, extensive error checking is required on VSIPL objects. In *production mode*, error checking is not required. This bimodal development model allows the vendor to provide maximum aid to application developers without sacrificing performance in deployed systems.

VSIPL is a large API including support for many different data types. VSIPL function names are strongly typed and functionality tends to be specific. This allows small code size for each function, but results in an API with many defined functions. To allow the vendor to match the library memory footprint to system memory size, the API is divided into *profiles* targeted at different application areas. Vendors claiming compatibility with a given VSIPL profile must implement all the functions listed in that profile.

The *Core Lite* profile, targeted at vector signal processing applications, is a minimal implementation and includes only a small subset of vector arithmetic and basic vector signal processing functions. The much larger *Core* profile includes most linear algebra functionality, such as matrix products, decompositions and solvers, and a richer set of signal processing functionality. In each profile, the vendor must provide at least one integer and one floating-point type. The precision that is provided is the choice of the vendor.

To allow portability of code which needs specific data precision, VSIPL provides several *portable precision* types. These types allow the user to specify the minimum number of decimal digits of accuracy for floating-point types. For integer types, the user can specify the exact number of bits, the minimum number of bits, or can allow the implementation to choose the *fastest* architecture-specific type with a minimum number of bits.

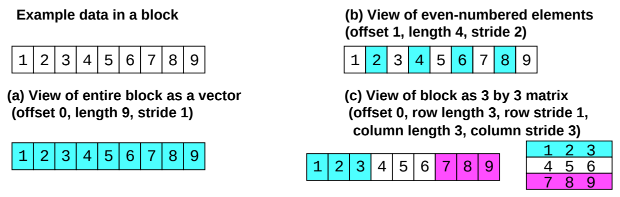
Example data in a block
1 2 3 4 5 6 7 8 9
(a) View of entire block as a vector
(offset 0, length 9, stride 1)
1 2 3 4 5 6 7 8 9
(b) View of even-numbered elements
(offset 1, length 4, stride 2)
1 2 3 4 5 6 7 8 9
(c) View of block as 3 by 3 matrix
(offset 0, row length 3, row stride 1,
column length 3, column stride 3)
1 2 3 4 5 6 7 8 9
1 2 3
4 5 6
7 8 9

Fig. 1. Different views of data in a block. The data may be viewed as a single vector of length 9 (a), or the even elements can be viewed as a vector of length 4 (b), or the data may be viewed as a 3 $\times$ 3 matrix (c).

3.2. Memory model

The basic objects in the VSIPL API are *blocks*, which represent contiguous areas of memory where data is stored, and *views*, which represent different virtual arrangements of data as one-, two-, or three-dimensional arrays. Views do not, themselves, contain data, but refer to data stored in a block. Views contain the structural information necessary to process data; an *offset* in the block of the first view element, and a *stride* and a *length* for each dimension. Figure 1 shows how data in a particular block can be viewed as a single vector of length 9 or a matrix of size 3 rows by 3 columns, and how the even-numbered elements can be viewed as a single vector of length 4. Through proper selection of lengths and strides, VSIPL supports matrices stored in both row-major and column-major order within a block.

In order to allow for vendor optimizations while preserving portability, VSIPL insists that data that it operates on not be accessed directly by the user or by other libraries. To allow data to be imported to and exported from VSIPL, the library supports the concept of two virtual spaces, *user space* and *VSIPL space*. VSIPL considers that it owns data that is in VSIPL space, and the user agrees to access the data only through VSIPL functions. Conversely, the library promises not to reference data while it is in user space. An area of memory may be associated with a block, and user data stored in the memory may move back and forth between the two spaces through use of the functions `admit` (which brings the data into VSIPL space) and `release` (which brings the data into user space). Depending on the machine's memory architecture and whether the user is interested in preserving the values in the block during the transfer, admit and release operations need not require a memory copy.

Traditional processors operate on complex data stored in an interleaved format, where the real part of each complex data element is followed by the corresponding imaginary part. On processors with a vector unit, such as the Motorola G4 with AltiVec, it is frequently more efficient

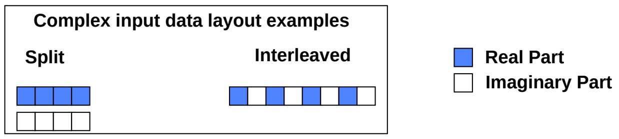

Fig. 2. VSIPL supports two formats for complex data, interleaved and split.

to store data in a *split* format, where all the real data for a given vector is stored contiguously and all the imaginary data is stored contiguously. These two formats are illustrated in Figure 2. VSIPL allows the implementor to choose the appropriate format internally. Regardless of the internal format, a complex block appears to the user as consecutive complex elements. VSIPL can support import or export of complex data in either format, allowing the user to make use of non-VSIPL routines that require either format.

3.3. Functionality

The expected arithmetic operations on vector and matrix views are supported, including unary, binary, and ternary operations, and specialized operations such as inner and outer products and matrix multiply. Boolean and index types are included to allow logical operations on vectors and special operations such as determining the indices of all elements that satisfy given criteria.

VSIPL supports a wide variety of operations important to signal and image processing. FFTs of one, two, and three dimensions, along with convolutions and correlations of one or two dimensions, are all supported. FIR and IIR filters can be easily described and implemented. A random number generator is provided that is portable between different implementations of VSIPL. Linear algebra functions that are provided include Gaussian elimination, QR factorization, Cholesky factorization, and singular value decomposition. Special solver functions allow the solution of common problems such as the linear least squares or covariance problems. Such solvers are provided to allow the implementation to optimize these problems in ways that may not be supportable in the individual steps.

One of the guiding principles in the design of the VSIPL API was that the user should look at each function in terms of its mathematical behavior rather than the details of its implementation. Thus, for example, the particular algorithm for an FFT is not specified; the standard specifies the required mathematical behavior of the FFT function. VSIPL supports encapsulation of implementation-specific variables such as twiddle tables inside an FFT object. Similarly, the QR factorization may be performed internally by either a Householder or Gram-Schmidt algorithm and the orthogonal factor can be stored in any convenient format. The user is given the resulting triangular factor and functions to multiply the orthogonal factor by other matrices.

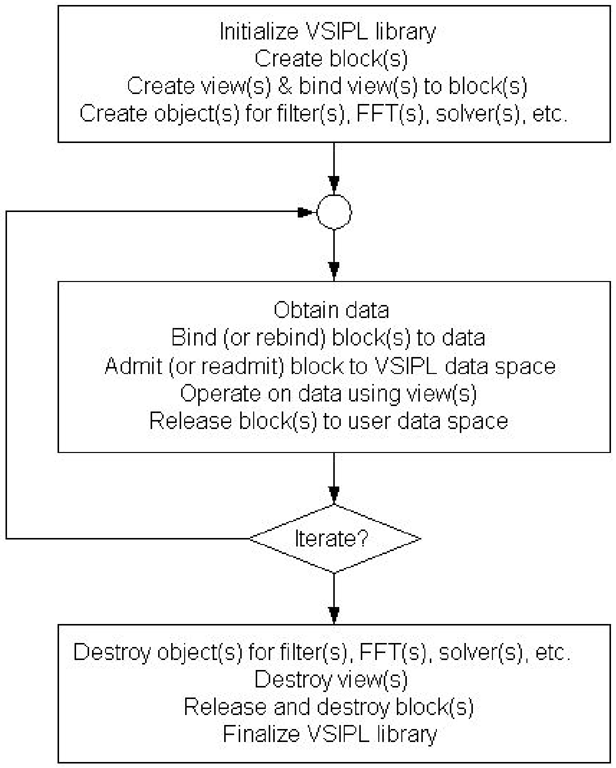
Initialize VSIPL library
Create block(s)
Create view(s) & bind view(s) to block(s)
Create object(s) for filter(s), FFT(s), solver(s), etc.
Obtain data
Bind (or rebind) block(s) to data
Admit (or readmit) block to VSIPL data space
Operate on data using view(s)
Release block(s) to user data space
Iterate?
Destroy object(s) for filter(s), FFT(s), solver(s), etc.
Destroy view(s)
Release and destroy block(s)
Finalize VSIPL library

Fig. 3. VSIPL Application Flow.

3.4. Developing a VSIPL Application

Development of embedded signal processing applications using COTS multiprocessing hardware and software typically consists of partitioning the code into two portions. One portion is the "outer loop" where the setup and cleanup functions are executed, typically memory allocation and coefficient generation, such as FFT twiddle factors and window coefficients. The other portion is the "inner loop" where the time-critical repetitive streaming data transformation functions lie. An efficient VSIPL application will be built similarly, with the outer loop executing heavyweight system functions that allocate memory when creating blocks and views. The inner loop contains the computation functions, i.e., the scalar, elementwise, signal processing, and linear algebra functions. Assuming the application does terminate for a given mission, then the outer loop would conclude after the inner loop concludes, destroying views and blocks. This is illustrated in Figure 3.

4. IMPLEMENTATIONS OF VSIPL

The complete VSIPL specification is available on the VSIPL web site (http://www.vsipl.org). An implementation of the

VSIPL production mode, written by Randy Judd of SPAWAR, is also available there. Versions of the library that support the Core and Core Lite profiles can be downloaded. The library can be compiled on any platform with an ANSI C compiler.

VSIPL is also becoming widely available on workstations and military-class embedded computing hardware. Core Lite implementations of VSIPL are available on VME-class embedded COTS signal processors from Mercury Computer Systems, SKY Computers, and CSPI. MPI Software Technology has produced implementations of VSIPL Core Lite for use on commercial single-board VME computers and personal computers. This implementation is licensed by Ixthos, Cetia, and DNAENT for their G3 and G4 boards, and a version for Linux is in development. MPI Software Technology and SKY are implementing the full Core profile as well. DARPA has funded Annapolis Microsystems to develop a VSIPL API interface for use with its FPGA-based adaptive computing technology. Teachey has reported some preliminary experiments with VSIPL implementations which show that the overhead for the library is minimal compared with more traditional vendor-optimized signal processing libraries [2].

A test suite is provided to test compliance of an implementation with the Core Lite profile of the API. Developed by the Georgia Tech Research Institute, the test suite is also available from the VSIPL web site. While currently only available to those participating in the Test Suite Working Group, it will be openly available for download in the near future.

5. FUTURE DEVELOPMENT OF VSIPL

Future activities of the VSIPL Forum will concentrate in the near term on necessary revisions of the specification based on user feedback. These revisions will constitute version 1.01 of the API, for release early in 2001.

Areas under active development include the completion of the draft image processing functions for the API. GTRI has investigated the issues in creating an object-oriented version of VSIPL and outlined a possible C++ binding in a report available on the VSIPL web site [3]. Extensions for better performance in a cluster environment and for enhanced error handling are also under development.

6. CONCLUSIONS

VSIPL is a portable API for vector, signal and image processing defined by representatives of signal processing industry vendors, application developers, and research organizations. It provides embedded applications with an easier upgrade path and the potential for easier system development. Portability is achieved through the use of opaque objects, protecting the user from architecture-specific details of memory management. Version 1.0 of the API is defined and available from the VSIPL web site, http://www.vsipl.org, along with an ANSI C implementation. Vendor-optimized implementations are also available on many embedded platforms.

7. REFERENCES

[1] D. A. Schwartz, R. R. Judd, W. J. Harrod, and D. P. Manley, "Vector, signal, and image processing library (VSIPL) 1.0 application programmer's interface," tech. rep., Georgia Tech Research Corporation, Mar. 2000.

[2] R. Teachey, A. Donadeo, E. Pancoast, G. Faix, and B. Chin, "COTS software portability standards and VSIPL benchmarks," in *Proceedings of the Fourth Annual High-Performance Embedded Computing (HPEC) Workshop*, Sept. 2000.

[3] D. P. Campbell, "Prototype extension of VSIPL into C++ and object oriented design," tech. rep., Georgia Tech Research Institute, Sept. 2000.