VSIPL++ User's Guide

Working Draft

Version 0.5

January 8, 2007

©2005 Georgia Tech Research Corporation, all rights reserved.

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies: THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

The US Government has a license under these copyrights, and this material may be reproduced by or for the U.S. Government.

Authors

The primary authors of this document were:

Name	Affiliation
Jules Bergmann	Code Sourcery, LLC.
Susan Emeny	ITT Industries, AES
Randall Judd	
Rick Pancoast	Lockheed Martin
David Leimbach	Verari, Inc.
Sharon Sacco	
Brian Sroka	MITRE Corporation

The primary editors of this document are:

Name	Affiliation
Dan Campbell	Georgia Tech Research Institute
Jeremy Kepner	Massachusetts Institute of Technology Lincoln Laboratory

Table of Contents

Authors i
Table of Contents ii
Table of Acronyms iii
Introduction 4
Section 1. VSIPL++ Examples 5
1.1 Vector add 6
1.2 No-Copy vector reference to matrix 7
1.3 Using User Defined Blocks 8
1.4 Simple FFT Example 9
1.5 Comparing VSIPL to VSIPL++ Simple Pulse Compression Case Studies 10
1.6 Importing and Exporting User Allocated Memory to VSIPL++ View Objects 20
1.7 Synthetic Aperture Radar VSIPL++ Example 24
1.8 VSIPL++ Glossary 33

Table of Acronyms

API	Application Programming Interface
VSIPL	Vector, Signal, and Image Processing Library
SLOC	Software Lines of Code
FFT	Fast Fourier Transform
VSIPL++	The C++ version of the VSIPL API
HPEC-SI	High Performance Embeddded Computing Software Initiative
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
SAR	Synthetic Aperature Radar
RASSP	Rapid prototyping of Application Specific Signal Processors
PRI	Pulse Repetition Interval
FIR	Finite Impulse Response
RCS	Radar Cross Section
IO	Input/Output

Introduction

The purpose of this User's Guide is to serve as a complement to the VSIPL++ Specification for application developers. The VSIPL++ specification fully defines the behavior of VSIPL++ implementations, and is focused on compactness and formal correctness rather than ease of reading. This guide seeks to clarify important aspects of VSIPL++ application development.

This guide is presented in the form of several illustrative examples that have been developed by VSIPL Forum members during the development and demonstration of the VSIPL++ Specification. Each example is in the form of VSIPL++ C++ source code, along with descriptions, and in some cases equivalent VSIPL C source code for contrast. Each illustrates an element of VSIPL++ application development that users have found needing clarification.

Section 1. VSIPL++ Examples

1.1 Vector add

This is a simple example of adding two vectors element-wise. The example also demonstrates:

1) Creating vectors
2) Using the ramp function
3) Putting values in a vector using the put functionality
4) Finding the length of a vector;
5) Getting values from vectors using the `get` functionality

```
/* Author Randall Judd (rrjudd@mac.com)*/

#include<cstdlib>
#include<iostream>
#include<vsip/initfin.hpp>
#include<vsip/selgen.hpp>
#include<vsip/vector.hpp>
#include<vsip/math.hpp>

int main(){
  // initialize library
  vsip::vsipl myvsip;

  // create a float vector "a" of length 10 filled with 1.0
  vsip::Vector<vsip::scalar_f> a(10,1.0F);

  /* demonstrate "put" and "length"
   * replace the last element of "a" with a 2.0
   * note that vectors are indexed starting a zero so the
   * index of the last element is the length minus 1.
   * a.put(index,value)
  */
  vsip::index_type last = a.length() - 1;
  a.put(last, 2.0F);

  // create a float vector "b" using the ramp function
  // The vector length is 10; the starting value is 0; the increment is 1
  vsip::Vector<vsip::scalar_f> b = vsip::ramp(0.0F,1.0F,10);

  // output the "a" and "b" vectors
  std::cout << "input data" << std::endl;
  std::cout << "a, b" << std::endl;
  for(vsip::index_type i=0; i<10; i++)
     std::cout << a.get(i) << ", " << b.get(i) << std::endl;

  // add the "a" vector to the "b" vector elementwise
  // replace the values in "a" with the results of a+b
  a = a+b;

  // output the a vector
  std::cout << "result of a + b" << std::endl;
  for(vsip::index_type i=0; i<10; i++)
     std::cout << a.get(i) << std::endl;
}
```

1.2 No-Copy vector reference to matrix

This example illustrates that the expected behavior of row_type for a Matrix to create a vector representing a row of a matrix without copying the contents. Updates to that row are reflected in the original matrix.

```
#include <vsip/initfin.hpp>
#include <vsip/matrix.hpp>
#include <vsip/vector.hpp>
#include <vsip/domain.hpp>
#include <iostream>

// display a row of the matrix with tabs.
std::ostream & operator << (std::ostream & os, vsip::Vector <> v) {
  int idx = 0;
  int size = v.size();
  for ( ; idx < size; ++idx)
    os << v.get(idx) << '\t';

  return os;
}

// Prints out a matrix row-wise
std::ostream & operator << (std::ostream & os, vsip::Matrix <> m) {
  int idx = 0;
  int size = m.size(1); /* m.size(1) is the size of a dimension of the matrix */
  for( ; idx < m.size(1); ++idx)
    os << m.row(idx) << std::endl;

  return os;
}

int main () {
  vsip::vsipl

  // defaults to scalar_f.
  vsip::Matrix<> m0 (4, 4, 0.0f); //4x4 Matrix with 0s

  // Show the contents of the matrix.
  std::cout << m0 << std::endl;

  // Each row_type is a reference to a row in the Matrix
  vsip::Matrix<>::row_type v00(m0.row(0));
  vsip::Matrix<>::row_type v01(m0.row(1));
  vsip::Matrix<>::row_type v02(m0.row(2));
  vsip::Matrix<>::row_type v03(m0.row(3));

  // Throw in some values diagonally.
  v01.put(1, 1.0f);
  v02.put(2, 2.0f);
  v03.put(3, 3.0f);

  // Show the original matrix
  std::cout << std::endl << m0 << std::endl;
}
```

1.3 Using User Defined Blocks

This section illustrates the use of a User Defined Block. A User Defined Block allows the application to access the data either directly or via a View. These are often needed for special memories, (ex. DMA, FPGA). Functions requiring direct access to the memory can use block pointers, while application code can use View accessors.

Original 'C' Declaration

```
// Allocate memory for 32K complex single precision floating point values
complex_type *Samples = NULL;
int numSamples = 32 * 1024

Samples = (complex_type *) MALLOC(numSamples * sizeof(complex_type));
if (Samples == NULL)  printf("Memory Error\n");

// Do some work here
…
free (nodeSamples);
return;
```

VSIPL++ Declaration

```
// First Create a block of memory for 32K complex single precision
// floating point values

int numSamples = 32 * 1024;
Dense<1, cscalar_f> Samples_block (numSamples, new scalar_f [numSamples*2]);

// Then, create a vector that references the block for View access
Vector < cscalar_f > Samples(Samples_block);

// Get a pointer to the data for direct access
scalar_f *Samples_blockPtr = Samples_block.find(Samples_blockPtr);

// Do some Direct Access work here, (ex. Read from disk, using blockPtr reference)
…
// Done with direct access, now Admit the block for View Access
Samples_block.admit();

// Do some work, using any valid VSIPL++ functionality (FFT, View Accessors)
Samples.put(i, cscalar_f(1.5, 2.3))
…
// Release the block, do any remaining direct access here (i.e. write to disk)
Samples_block.release();

return;
```

1.4 Simple FFT Example

This section shows a simple example of using the FFT functionality of VSIPL++

Example Using FFTs

```
#include vsip_headers;

// Some variables
int numSamples = 32 *1024;
int invFftLen = numSamples / 2;

// Create Vectors
Vector < cscalar_f > signal(numSamples);
Vector < cscalar_f > Result(invFftLen);

// set up Forward complex to complex fft
Fft< const_Vector, cscalar_f, cscalar_f, fft_fwd, 0, SINGLE, by_reference >
             fft32K_fwd_c2c ((Domain<1>(fftLen)), static_cast<scalar_f>(1.0));

// set up Inverse complex to complex fft
Fft< const_Vector, cscalar_f, cscalar_f, fft_inv, 0, SINGLE, by_reference >
             fft16K_inv_c2c ((Domain<1>(invFftLen)),
             static_cast<scalar_f>(1.0/invFftLen));

// Take forward FFT
fft32K_fwd_c2c(Samples, signal);

// Take inverse FFT
fft16K_inv_c2c(signal, Result)
```

VSIPL++ Headers

```
#ifndef __VSIP_HEADERS__
    #define __VSIP_HEADERS__
    #include <vsip/vector.hpp>
    #include <vsip/dense.hpp>
    #include <vsip/domain.hpp>
    #include <vsip/complex.hpp>
    #include <vsip/initfin.hpp>
    #include <vsip/support.hpp>
    #include <vsip/signal.hpp>
    #include <vsip/maxmin.hpp>
    #include <vsip/math.hpp>

namespace vsip
{
    typedef vsip_scalar_d scalar_d;
    typedef std::complex<vsip_scalar_d> cscalar_d;
}

using namespace std;
using namespace vsip;

typedef cscalar_f complex_type ;
typedef cscalar_d dbl_complx ;
typedef scalar_f  real_type;
```

```
#endif
```

1.5 Comparing VSIPL to VSIPL++ Simple Pulse Compression Case Studies

This section illustrates the differences between using VSIPL and VSIPL++ application source code for a simple pulse compression algorithm with several different additional constraints. The main algorithm is:

```
output = ifft( fft(input) * ref );
```

Each subsection illustrates a variant of this algorithm as implemented in VSIPL and VSIPL++. The “Notes” subsections summarizes the highlights of the differences between the implementations.

1.5.1 Baseline

VSIPL

```
void pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    vsip_length size = vsip_cvgetlength_f(in);

    vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1,
             VSIP_ALG_SPACE);
    vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1,
             VSIP_ALG_SPACE);

    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);

    vsip_cvalldestroy_f(tmpView1);
    vsip_cvalldestroy_f(tmpView2);
    vsip_fft_destroy_f(forwardFft);
    vsip_fft_destroy_f(inverseFft);
}
```

VSIPL++

```
void pulseCompress(const vsip::Vector< std::complex<float> > &in,
                                  const vsip::Vector< std::complex<float> > &ref,
                                  const vsip::Vector< std::complex<float> > &out) {
  int size = in.size();

  vsip::Fft<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::fft_fwd> forwardFft
      ((vsip::Domain<1>(size)), 1.0);
  vsip::Fft<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::fft_inv, 0,
      vsip::SINGLE, vsip::by_reference> inverseFft ((vsip::Domain<1>(size)),
      1.0/size);
```

```
  inverseFft( ref * forwardFft(in), out );
}
```

NOTES

- VSIPL++ code has fewer SLOCS than VSIPL code
  (5 VSIPL++ SLOCS vs. 13 VSIPL SLOCS)
- VSIPL++ syntax is more complex than VSIPL syntax
  - Syntax for FFT object creation
  - Extra set of parenthesis needed in defining Domain argument for FFT objects
- VSIPL code includes more management SLOCS
  - VSIPL code must explicitly manage temporaries
  - Must remember to free temporary objects and FFT operators in VSIPL code
- VSIPL++ code expresses core algorithm in fewer SLOCS
  - VSIPL++ code expresses algorithm in one line, VSIPL code in three lines
  - Performance of VSIPL++ code may be better than VSIPL code

1.5.2 Catch any errors and propagate error status

VSIPL

```
int pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  int valid = 0;
  vsip_length size = vsip_cvgetlength_f(in);

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1,
       VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1,
       VSIP_ALG_SPACE);

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

  if (forwardFft && inverseFft && tmpView1 && tmpView2)  {
    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);
    valid=1;
  }

  if (tmpView1)  vsip_cvalldestroy_f(tmpView1);
  if (tmpView2)  vsip_cvalldestroy_f(tmpView2);
  if (forwardFft) vsip_fft_destroy_f(forwardFft);
  if (inverseFft) vsip_fft_destroy_f(inverseFft);
  return valid;
}
```

VSIPL++

```
void pulseCompress(const vsip::Vector< std::complex<float> > &in,
                                  const vsip::Vector< std::complex<float> > &ref,
                                  const vsip::Vector< std::complex<float> > &out) {
  int size = in.size();

  vsip::Fft<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::fft_fwd>  forwardFft
       ((vsip::Domain<1>(size)), 1.0);
  vsip::Fft<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::fft_inv, 0,
       vsip::SINGLE, vsip::by_reference>  inverseFft ((vsip::Domain<1>(size)),
       1.0/size);

  inverseFft( ref * forwardFft(in), out );
}
```

NOTES

- VSIPL code additions are highlighted
    - No changes to VSIPL++ function due to VSIPL++ support for C++ exceptions
    - 5 VSIPL++ SLOCS vs. 17 VSIPL SLOCS
- VSIPL behavior not defined by specification if there are errors in FFT and vector multiplication calls
    - For example, if lengths of vector arguments unequal, implementation may core dump, stop with error message, silently write past end of vector memory, etc
    - FFT and vector multiplication calls do not return error codes

1.5.3 Decimate input by N prior to first FFT

VSIPL

```
void pulseCompress( int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref,
vsip_cvview_f *out) {
  vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1,
       VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1,
       VSIP_ALG_SPACE);

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

  vsip_cvputstride_f(in, decimationFactor);
  vsip_cvputlength_f(in, size);

  vsip_ccfftop_f(forwardFft, in, tmpView1);
  vsip_cvmul_f(tmpView1, ref, tmpView2);
  vsip_ccfftop_f(inverseFft, tmpView2, out);

  vsip_cvalldestroy_f(tmpView1);
  vsip_cvalldestroy_f(tmpView2);
  vsip_fft_destroy_f(forwardFft);
  vsip_fft_destroy_f(inverseFft);
}
```

VSIPL++

```
void pulseCompress(int decimationFactor,  const vsip::Vector< std::complex<float> >
       &in, const vsip::Vector< std::complex<float> > &ref
const vsip::Vector< std::complex<float> > &out) {
  int size = in.size() / decimationFactor;
  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::Fft<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::fft_fwd> forwardFft
       ((vsip::Domain<1>(size)), 1.0);
  vsip::Fft<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::fft_inv, 0,
       vsip::SINGLE, vsip::by_reference> inverseFft ((vsip::Domain<1>(size)),
       1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```

NOTES

- SLOC count doesn't change much for VSIPL or VSIPL++ code
    - VSIPL: 2 lines changed, 2 additional lines
    - VSIPL++: 3 lines changed, 1 additional line
- VSIPL version of code has a side-effect: The input vector was modified and not restored to original state. This type of side-effect was the cause of many problems/bugs when we first started working with VSIPL

1.5.4 Decimate input by N prior to first FFT, no side-effects

VSIPL

```
void pulseCompress( int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref,
vsip_cvview_f *out) {
  vsip_length savedSize   = vsip_cvgetlength_f(in);
  vsip_length savedStride = vsip_cvgetstride_f(in);
  vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1,
       VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1,
       VSIP_ALG_SPACE);
  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvputlength_f(in, size);
  vsip_cvputstride_f(in, decimationFactor);
  vsip_ccfftop_f(forwardFft, in, tmpView1);
  vsip_cvmul_f(tmpView1, ref, tmpView2);
  vsip_ccfftop_f(inverseFft, tmpView2, out);
  vsip_cvputlength_f(in, savedSize);
  vsip_cvputstride_f(in, savedStride);

  vsip_cvalldestroy_f(tmpView1);
  vsip_cvalldestroy_f(tmpView2);
  vsip_fft_destroy_f(forwardFft);
  vsip_fft_destroy_f(inverseFft);
}
```

VSIPL++

```
void pulseCompress(int decimationFactor,  const vsip::Vector< std::complex<float> >
       &in, const vsip::Vector< std::complex<float> > &ref
const vsip::Vector< std::complex<float> > &out) {
  int size = in.size() / decimationFactor;
  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::Fft<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::fft_fwd> forwardFft
       ((vsip::Domain<1>(size)), 1.0);
  vsip::Fft<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::fft_inv, 0,
       vsip::SINGLE, vsip::by_reference> inverseFft ((vsip::Domain<1>(size)),
       1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```

NOTES

- VSIPL code must save the input vector state prior to use and restore it before returning
- Code size changes:
    - VSIPL code requires 4 additional lines
    - VSIPL++ code does not change from prior version

1.5.5 Support both single and double precision floating point

VSIPL – Single Precision

```
void pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    vsip_length size = vsip_cvgetlength_f(in);

    vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1,
       VSIP_ALG_SPACE);
    vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1,
       VSIP_ALG_SPACE);

    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);

    vsip_cvalldestroy_f(tmpView1);
    vsip_cvalldestroy_f(tmpView2);
    vsip_fft_destroy_f(forwardFft);
    vsip_fft_destroy_f(inverseFft);
}
```

VSIPL – Double Precision

```
void pulseCompress(vsip_cvview_d *in, vsip_cvview_d *ref, vsip_cvview_d *out) {
  vsip_length size = vsip_cvgetlength_d(in);

  vsip_fft_d *forwardFft = vsip_ccfftop_create_d(size, 1.0, VSIP_fft_fwd, 1,
       VSIP_ALG_SPACE);
  vsip_fft_d *inverseFft = vsip_ccfftop_create_d(size, 1.0/size, VSIP_fft_inv, 1,
       VSIP_ALG_SPACE);

  vsip_cvview_d *tmpView1 = vsip_cvcreate_d(size, VSIP_MEM_NONE);
  vsip_cvview_d *tmpView2 = vsip_cvcreate_d(size, VSIP_MEM_NONE);

  vsip_ccfftop_d(forwardFft, in, tmpView1);
  vsip_cvmul_d(tmpView1, ref, tmpView2);
  vsip_ccfftop_d(inverseFft, tmpView2, out);

  vsip_cvalldestroy_d(tmpView1);
  vsip_cvalldestroy_d(tmpView2);
  vsip_fft_destroy_d(forwardFft);
  vsip_fft_destroy_d(inverseFft);
}
```

VSIPL++ - Arbitrary Precision

```
template<class T, class U, class V> void pulseCompress(const T &in, const U &ref,
       const V &out) {
  int size = in.size();
  vsip::Fft<vsip::Vector, typename T::value_type, typename V::value_type,
       vsip::fft_fwd> forwardFft ((vsip::Domain<1>(size)), 1);
  vsip::Fft<vsip::Vector, typename T::value_type, typename V::value_type,
       vsip::fft_inv, 0, vsip::SINGLE, vsip::by_reference> inverseFft
       ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft(in), out );
}
```

NOTES

- VSIPL++ code has same SLOC count as original
    - Uses C++ templates (3 lines changed)
    - Syntax is slightly more complicated
- VSIPL code doubles in size
    - Function must first be duplicated
    - Small changes must then be made to code (*i.e.*, changing _f to _d)

1.5.6 Support all previously stated requirements

VSIPL – Single Precision

```
void pulseCompress(int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref,
vsip_cvview_f *out) {
  vsip_length savedSize   = vsip_cvgetlength_f(in);
  vsip_length savedStride = vsip_cvgetstride_f(in);

  vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1,
       VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1,
       VSIP_ALG_SPACE);

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

  if (forwardFft && inverseFft && tmpView1 && tmpView2)
  {
    vsip_cvputlength_f(in, size);
    vsip_cvputstride_f(in, decimationFactor);

    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);

    vsip_cvputlength_f(in, savedSize);
    vsip_cvputstride_f(in, savedStride);
  }

  if (tmpView1)   vsip_cvalldestroy_f(tmpView1);
  if (tmpView2)   vsip_cvalldestroy_f(tmpView2);
  if (forwardFft) vsip_fft_destroy_f(forwardFft);
  if (inverseFft) vsip_fft_destroy_f(inverseFft);
}
```

VSIPL – Double Precision

```
void pulseCompress(int decimationFactor, vsip_cvview_d *in, vsip_cvview_d *ref,
vsip_cvview_d *out) {
  vsip_length savedSize   = vsip_cvgetlength_d(in);
  vsip_length savedStride = vsip_cvgetstride_d(in);

  vsip_length size = vsip_cvgetlength_d(in) / decimationFactor;

  vsip_fft_d *forwardFft = vsip_ccfftop_create_d(size, 1.0, VSIP_FFT_FWD, 1,
       VSIP_ALG_SPACE);
  vsip_fft_d *inverseFft = vsip_ccfftop_create_d(size, 1.0/size, VSIP_FFT_INV, 1,
       VSIP_ALG_SPACE);

  vsip_cvview_d *tmpView1 = vsip_cvcreate_d(size, VSIP_MEM_NONE);
  vsip_cvview_d *tmpView2 = vsip_cvcreate_d(size, VSIP_MEM_NONE);

  if (forwardFft && inverseFft && tmpView1 && tmpView2)
  {
    vsip_cvputlength_d(in, size);
    vsip_cvputstride_d(in, decimationFactor);

    vsip_ccfftop_d(forwardFft, in, tmpView1);
    vsip_cvmul_d(tmpView1, ref, tmpView2);
    vsip_ccfftop_d(inverseFft, tmpView2, out);

    vsip_cvputlength_d(in, savedSize);
    vsip_cvputstride_d(in, savedStride);
  }

  if (tmpView1)   vsip_cvalldestroy_d(tmpView1);
  if (tmpView2)   vsip_cvalldestroy_d(tmpView2);
  if (forwardFft) vsip_fft_destroy_d(forwardFft);
  if (inverseFft) vsip_fft_destroy_d(inverseFft);
}
```

VSIPL++ - Arbitrary Precision

```
template<class T, class U, class V> void pulseCompress(int decimationFactor, const T
       &in, const U &ref, const V &out) {
  int size = in.size() / decimationFactor;

  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::Fft<vsip::Vector, typename T::value_type, typename V::value_type,
       vsip::fft_fwd> forwardFft ((vsip::Domain<1>(size)), 1);
  vsip::Fft<vsip::Vector, typename T::value_type, typename V::value_type,
       vsip::fft_inv, 0, vsip::SINGLE, vsip::by_reference> inverseFft
       ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```

NOTES

- Final SLOC count: VSIPL++ -- 6 lines VSIPL -- 40 lines (20 each for double and single precision versions).

1.6 Importing and Exporting User Allocated Memory to VSIPL++ View Objects

Following the philosophy of the `VSIPL` specification, `VSIPL++` objects manage their own memory allocation, alignment, and storage. To interface `VSIPL++` objects with external or user defined or allocated memory requires additional block manipulation. `VSIPL++ Dense<>` blocks provide the member functions listed in Table 1.1 to allow the import and export of externally allocated memory.

Member Functions	Description
`Dense<>::admit()`	Commits external memory for exclusive `VSIPL++` object manipulation
`Dense<>::release`	Release external memory from exclusive use by `VSIPL++` objects
`Dense<>::find()`	Access external memory pointers from `VSIPL++` objects
`Dense<>::rebind()`	Reset external memory pointer(s) within `VSIPL++` objects previously bound to external memory
`Dense<>::user_storage()`	Determine the state and layout of user/external memory from `VSIPL++` objects
`Dense<>::admitted()`	Determine whether a `VSIPL++` objects external memory is committed for exclusive `VSIPL++` object manipulation

Table 1.1 `Dense<>` Block Functions for the Import and Export of External Data

There are four general steps for importing and exporting external memory and data into and out of `VSIPL++` block objects. First a `Dense<>` block object is declared and bound to external memory pointer(s). Second, a `VSIPL++` view object is created using the `Dense<>` block so that operations may later be performed on the external data. Next, The `Dense<>` external/user block is admitted to the `VSIPL++` library, thereby entering an informal contract ensuring that only `VSIPL++` operations access the provided external memory pointer(s). Finally, after vector or mathematical operations, the eternal memory and modified data is released back for non-`VSIPL++` access and manipulation. The following code sample illustrates these steps.

```
#define N 5

scalar_f data[N] = {2, 3, 5, 8, 13};
scalar_f *ptr    = &data[0];

Dense<1, scalar_f> ExtBlock( Domain<1>(N), ptr );
Vector<scalar_f>   ExtVector( ExtBlock );
Vector<scalar_f>   LibVector( N, 10 );

ExtBlock.admit( true );
ExtVector = ExtVector + LibVector;
ExtBlock.release( true );
```

The result of this code fragment's execution increments the elements of the sequence pointed to by `ptr` and stored in the `data` array by 10. While this example illustrates the general methods, it is more often the case that the external memory pointer(s) will not be available at the time of `VSIPL++` object declarations. Should external data or memory come into scope after object declaration there are two possible means for importing the data into the `VSIPL++` objects. First, object pointers can be declared and later defined after the external data becomes available.

```
scalar_f *ptr;

Dense<1, scalar_f> *ExtBlock;
Vector<scalar_f>   *ExtVector;

ReceiveData( &ptr );

ExtBlock  = new Dense<1, scalar_f>( Domain<1>(5), ptr );
ExtVector = new Vector<scalar_f>( ExtBlock );

ExtBlock->admit( true );
```

While this method successfully imports external data, it may cause object memory allocation in time critical inner loops thereby degrading run-time performance. A second method to import late arriving data into `VSIPL++` objects is to declare and define `VSIPL` block objects with unset or wild pointers and later reset the pointers (via the `Dense<>::rebind` method) before any operations. While the initial objects will not be valid, this forces early object memory allocation at the time of declaration. This method is used in many high performance codes and is referred to as *early binding*.

```
scalar_f *ptr = 0xf;  // Wild pointer - must RESET

Dense<1, scalar_f> ExtBlock( Domain<1>(5), ptr );
Vector<scalar_f>   ExtVector( ExtBlock );

ReceiveData( &ptr );
```

```
ExtBlock.rebind( ptr );
ExtBlock.admit( true );
```

Adding complexity to the process of importing and exporting data to and from `VSIPL++` block objects is complex vector and matrix data layouts. Traditionally, complex vectors are stored in memory in an interleaved manner, alternating real and imaginary components linearly in contiguous memory. An alternative complex vector layout, referred to as split, is to store all real components followed by all imaginary components in contiguous memory. `VSIPL++` supports the import and export of both these complex data layouts from external memory by providing two `Dense<>` constructors, one accepting a single memory pointer (interleaved) and the other accepting two memory pointers (split).

```
Dense<2, cmplx_f>  SplitBlock( Domain<2>(R,C), iptr, qptr );
Dense<2, cmplx_f>  InterBlock( Domain<2>(R,C), iqptr );

Matrix<cmplx_f>    SplitMatrix( SplitBlock );
Matrix<cmplx_f>    InterMatrix( InterBlock );
```

The problem of having two methods to import/export complex data into Dense<> block objects is that most `VSIPL++` implementations will support computations in only one of the two complex data layouts. Therefore, if external data is provided in the format unused by the `VSIPL++` implementation the `Dense<complex<>::admit` invocation will cause a library internal data copy that will degrade run-time performance. Furthermore, the `VSIPL++` specification requires that complex data be released in the same format as it was admitted, therefore causing another internal data copy to restore external memory (should it be required for export). Providing external or user data in the `VSIPL++` implementation's preferred complex layout will increase the overall run-time performance of a `VSIPL++` application.

Careful use of the boolean argument to the `Dense<>::admit` and `Dense<>::release` methods can help avoid unneeded library internal data copies. If external memory is used only for importing data to `VSIPL++` objects, there is no need to require data updates (i.e. internal data copies) when releasing that memory from the library. Conversely, if external memory is used only for exporting data from `VSIPL++` objects, there is no need to require data updates when admitting than memory to the library.

The following code sample illustrates a strategy for providing the correct complex data layout to `Dense<>` constructors using preprocessor directives.

```
#ifdef SPLIT_CMPLX
Dense<2, cmplx_f>  InputBlock( Domain<2>(M,N), iptr, qptr );
Dense<2, cmplx_f>  OutputBlock( Domain<2>(R,C), iptr, qptr );
#else
Dense<2, cmplx_f>  InputBlock( Domain<2>(M,N), iqptr );
Dense<2, cmplx_f>  OutputBlock( Domain<2>(R,C), iqptr );
#endif

Matrix<cmplx_f>    InputMatrix( InputBlock );
Matrix<cmplx_f>    OutputMatrix( OutputBlock );

#ifdef SPLIT_CMPLX
ReceiveSplitInput( &iptr, &qptr );
InputBlock.rebind( iptr, qptr );
#else
ReceiveInterInput( &iqptr );
InputBlock.rebind( iqptr );
#endif

InputBlock.admit( true );
OutputBlock.admit( false );

// Perform VSIPL++ operations. Store final results in OutputMatrix

InputBlock.release( false );
OutputBlock.release( true );

#ifdef SPLIT_CMPLX
SendSplitInput( &iptr, &qptr );
#else
SendInterInput( &iqptr );
#endif
```

1.7 Synthetic Aperture Radar VSIPL++ Example

1.7.1 Introduction

This chapter describes the implementation of the Rapid Prototyping of Application Specific Signal Processors (RASSP) Synthetic Aperture Radar (SAR) benchmark algorithm using VSIPL++.

1.7.2 SAR Processing Description

Frame Processing and Parameters

Each SAR frame (or Pulse Repetition Interval [PRI]) produces a raw data cube:

(`npulse` pulses) x (4 polarizations) x (`nrange` range cells)

These raw pulses are first range processed. After range processing, the 4 x `npulse` x `nrange` current frame data cube is combined with the previous frame data cube for azimuth processing. Azimuth processing results for the first `npulse` pulses x `nframe` cells are written out to disk.

The key parameters of the SAR algorithm are:

`nframe`	- Number of PRI/frames.
`nrange`	- Number of range cells in a PRI/frame.
`npulse`	- Number of pulses in a PRI/frame.
`niq`	- Number of filter taps for sideband filter.

Range Processing

Range processing consists of the following operations:

1. Rearrange sample data into complex format.
2. Apply sideband filter to real and imaginary components.
3. Apply equalization weights.
4. Forward FFT.
5. Apply Radar Cross Section (RCS) weights.

Azimuth Processing

Azimuth processing consists of the following operations:

1. Convolution across current and previous range processed frames (using excpliti FFT fast convolution)
2. Write processed frame as output.

1.7.3 VSIPL++ Implementation

Frame Processing

In VSIPL++, we handle the frame processing in the function `process`. As inputs, it takes the following:

- Datacube parameters: nframe, nrange, npulse, ncscamples, niq
- Input and output file handles.
- Weights for equalization and RCS.
- Filter coefficients for sideband filter and azimuth processing.
- More details on how the main program creates these inputs is discussed below.

Process performs the following:

- create the top-level data structures to hold the data cubes
- foreach frame:
    - call IO functions to read in each pulse
    - call range processing to process each range sample
    - call azimuth processing after an entire frame has been range processed (azimuth processing handles the output IO)
    - shift the data cube

Additionally, process takes its processing precision as a template parameter T. This allows the same process function to be used for both single and double precision processing.

Data Structures

Prior to processing any frames, data structures must be created to hold the input data and data cubes for each polarization. Input data is read from C routines, C style arrays are used.

The 4 polarization data cubes are held in a tensor (a three-dimensional array) of complex values:

```
Tensor<complex<T> > buf(4, 2*npulse, nrange, 0.f);
```

The first three parameters indicate the size of the Tensor:

- 4 - the number of polarizations.
- 2*npulse - total number of pulses in two frames of data (two frames are necessary since azimuth processing operates on the current and previous frames).
- nrange - the number of range cells in each pulse.

The final parameter 0.f is the value that each element in the tensor will be initialized to. If no value is provided, the initial values of the tensor are undefined.

In addition to data storage, objects for range processing and azimuth processing are created and given their parameters.

Frame Processing

At the start of each frame, pulses are read one by one and given to the range processing object:

```
for (int p=0; p<npulse; ++p)
```

```
  for (int pol=HH; pol <= VV; ++pol)
    // read pulse data into 'rod' a C-array of shorts
    read_adts(fpin, rod, aux, &range, pol, ncscamples);
    // range process pulse data in rod
    rp.process(buf(pol, whole, whole), w_eq.row(pol), rod, p);
```

The RangeProcess objects the data cube for a single polarization. To extract this from buf, a matrix subview is used:

```
// previously in process, we defined full for convenience:
typename buf_view_type::whole_domain_type full =
                  buf_view_type::whole_domain;

// extract a submatrix from tensor buf for polarization pol:
buf(pol, full, full);
```

The resulting subview is an alias to the data stored in buf. Creating the subview does not require data to be copied, and changes made to the subview are also present in the original tensor. However, this is the behavior we want.

Since RangeProcess only updates the p'th pulse of buf, alternatively process and RangeProcess could have been implemented to pass just this vector subview:

```
buf(pol, p, full);
```

After range processing has been done on each input pulse, azimuth processing is performed for each polarization:

```
for (int pol=HH; pol<=VV; ++pol)
  ap.process(fpout[pol], buf(pol, whole, whole), kernel0);
  if (frame != nframe)
    ap.shift(buf(pol, whole, whole));
```

After azimuth processing, the data cubes are shifted to make space for the next frame's data.

1.7.4 Range Processing

Range processing is handled by a RangeProcess object. It encapsulates all the information necessary to process a single pulse.

The first step is organize the raw data (which is in a C array of shorts) into a VSIPL++ vector of complex (recall that T is a template parameter describing the precision):

```
for (index_t i=0; i<ncsamples_; i+=2) {
   vec_iq_.put(i+0, complex<T>( (float)rod[2*i+0],
                                (float)rod[2*i+1]) );
   vec_iq_.put(i+1, complex<T>(-(float)rod[2*i+2],
                               -(float)rod[2*i+3]) );
   }
```

Upper Sideband Filter

Next is to perform a sideband filter which is a FIR filter with niq taps. This is performed with a Convolution object. This filter was created when the RangeProcess object was initialized. Since

we need multiple filters (one for in-phase and one for quadrature), a typedef is created for convenience:

```
typedef vsip::Convolution<
      const_Vector,
      vsip::nonsym,
      vsip::support_min,
      T,
      0,
      vsip::alg_space> conv_t;
```

The Convolution type parameters are as follows:

`const_Vector`	- describes the dimension of the convolution. In our case it is a 1D convolution.
`nonsym`	- describes the symmetry of our coefficients. In our case there is no symmetry.
`T`	- describes the precision of the filter.
`0`	- Number of expected invocations of the filter. This is a hint to the implementation that may be used this to enable/disable optimizations. In our case we say “0”, which is equivalent to an infinite number of invocations.
`alg_space`	- Another hint to the implementation.

With the typedef, we can initialize our filters:

```
conv_t iconv_(i_coef, Domain<1>(ncsamples_), 1);
conv_t qconv_(q_coef, Domain<1>(ncsamples_), 1);
```

The first argument is a vector of coefficients (i_coef and q_coef are arguments given to the RangeProcess object when it is created). The second argument describes the input sizes that will be given to the filter. The last argument is the decimation factor.

- The first argument is a vsip::Vector holding the coefficients.
- The second argument is the size of views that this filter will be applied to. In our case will be convolving vecIQ, which contains ncscamples.
- The final argument is the decimation factor, which in our case is 1.

Finally, we can invoke the filters on the data in vec_iq_:

```
iconv(vec_iq_.real(), line_(conv_dom).real());
qconv(vec_iq_.imag(), line_(conv_dom).imag());
```

Several things are occurring here. The filters are invoked with an input view and an output view. Since we wish to apply the filters separately to the real and imaginary components of vec_iq_, the inputs to the filters are subviews. vec_iq_.real() indicates a subview of the real values in vec_iq_. As a subview, it aliases the data in vec_iq_ and does not create a copy. Since we would like to put our filter results back into complex format, our output views are also real and imaginary subviews. As an added twist, the output vector line_ is actually larger than required, so a domain subview is also applied.

The following code is functionally equivalent (but less efficient since it requires explicit data copies):

```
Vector<T> real_in = vec_iq_.real();
Vector<T> imag_in = vec_iq_.imag();
Vector<T> real_out(ncsamples_-niq-+1);
Vector<T> imag_out(ncsamples_-niq-+1);
iconv(real_in, real_out);
qconv(imag_in, imag_out);
line_(conv_dom).real() = real_out;
line_(conv_dom).imag() = imag_out;
```

(Note, we could also have used a FIR filter to perform the sideband filter. The primary difference between a FIR filter and a Convolution is that FIR filters can store state between invocations so that an input stream can be processed in multiple chunks. FIR filters are also limited to 1-dimension.)

Direct Approach

For comparison, we could also just convert the filter into an explicit convolution loop

```
for(index_type i=0; i<ncsamples_-niq_; i++){
  T isum = 0.;
  T qsum = 0.;

  for(index_type k=0; k<niq_; k++) {
    isum += vec_iq_.real().get(i+k) * i_coef_.get(niq_-k-1);
    qsum += vec_iq_.imag().get(i+k) * q_coef_.get(niq_-k-1);
    }
  line_.put(i, vsip::complex<T>(isum, qsum));
  }
```

If we're starting from existing code, this might be the easiest approach. However there are several advantages of using VSIPL++ higher level constructs:

- Higher level constructs give the library more information about the algorithm being performed, enabling greater optimizations. As a nested for loop, the library just sees get()'s and put()'s and its optimization potential is limited. As a convolution call, the library can plug in higher-performance algorithms, such as FFT based overlap-add algorithm.
- Higher level constructs make the program easier to understand. Once a programmer understands what a VSIPL++ convolution object does, they will easily understand each invocation. Explicit loops require the programmer to reverse engineer the higher-level mathematical intent.

Equalization

After applying the filter, we have to apply equalization weights to the first ncsamples-niq elements of line_ and zero-fill the rest. Ideally, we would say:

```
// create convenience domains
Domain keep_dom(ncscamples_-niq_);
Domain zero_dom(ncscamples_-niq_, 1, nrange_-(ncsamples_-niq_));

// apply weights to first ncsamples_-niq_ elements:
line_(keep_dom) *= weight(keep_dom);
// zero remaining elements:
line_(zero_dom)  = 0.f;
```

However, if we are processing in double precision (T = double), then line_ will have a different value type (complex<double>) than weight (complex<float>). Unfortunately, std::complex does not define an operator* for complex values with different precision.

To handle this, we use the function cast_view to cast each element of weight to the appropriate type:

```
line_(keep_dom) *= cast_view<complex<T> >(weight(keep_dom));
```

The implementation of cast_view is discussed later.

Fast Fourier Transform

Next, an FFT is applied to the data in line_. RangeProcess creates its FFT object as follows:

```
// First, a convenience typedef
typedef vsip::Fft<
      const_Vector,     // FFT is 1-dimensional
      complex<T>,       // Input is complex<T>
      complex<T>,       // Output is complex<T>
      fft_fwd,          // Direction is Forward
      by_reference,     // Invoke by reference
      0,                // Hint: FFT will be used a lot
      alg_space>        // Hint: Optimize for space
            fft_t;
// Now define the FFT:
fft_t fft_(
      Domain<1>(nrange_),      // Size of FFT
      1.f);                    // Scale: 1.0 -> no scaling
```

To apply the FFT, we use the in-place variant:

```
fft_(line_);
```

RCS Weighting

After the Fourier transform, we apply RCS weights. Again, we use cast_view to work with mixed-precision:

```
line_ *= cast_view<complex<T> >(rcs_);
```

Storing into Datacube

The last step of range processing is to store line_ into the data cube:

```
buf.row(j + npulse_) = line_;
```

1.7.5 Azimuth Processing

Azimuth processing is handled by an azimuth processing object, which like the range processing object, stores necessary parameters and data to azimuth process the data cube.

Fast Convolution

The core of azimuth processing is performing a fast convolution for each range line. This requires that forward and inverse FFT objects be created when the azimuth processing object is initialized:

```
// Convenience typedefs
typedef Fft<
      const_Vector,           // FFT is 1-dimensional
      complex<T>, complex<T>, // complex-to-complex
      fft_fwd,                // Direction is forward
      by_reference,           // Invoke by-reference
      0, alg_space>           // Usual hints
            for_fft_t;
typedef Fft<
      const_Vector,           // FFT is 1-dimensional
      complex<T>, complex<T>, // complex-to-complex
      fft_inv,                // Direction is inverse
      by_reference,           // Invoke by-refernce
      0, alg_space>           // Usual hints
            inv_fft_t;
for_fft_t for_fft_(
      Domain<1>(2*npulse_),   // Size = 2 * npulse_
      1.f);                   // Forward scaling = 1.0
inv_fft_t inv_fft_(
      Domain<1>(2*npulse_),   // Size = 2 * npulse_
      1.f / (2*npulse_));     // Inverse scaling 1/N
```

Now these fast convolution is performed for each range line:

```
for (index_type i=0; i<nrange_; ++i) {
  // copy range line into buffer:
  azbuf_ = buf.col(i);

  // Forward FFT (in-place)
  for_fft_(azbuf_);

  // Multiple by convolution kernel
  index_type index = kernel0 + (i*16)/nrange_;
  azbuf_ *= cast_view<complex<T> >(phase_.row(index));

  // Inverse FFT (in-place)
  inv_fft_(azbuf_)

  ...
  }
```

The convolution kernel is selected from a set of kernels provided to the azimuth processing object when it is created. cast_view is used to handle mixed-precision correctly.

Alternatives: Another VSIPL++ tool that could be used here is Convolution. It, however, requires its coefficients to be given at initialization. Since azimuth processing uses a separate kernel for each range line, this would require 2*npulse convolution objects.

Store Range Cell to File

After fast convolution, the second half of each range cell is written to disk. This requires setting a special user-storage block up in advance:

```
// allocate raw memory
float* io_buf_ = new float[2*npulse_];
// define user-storage block
Dense<1, complex<float> > io_block_(npulse_, io_buf_);
// wrap with vector
Vector<complex<float> > io_vec_(io_block_);
```

With this in place, we use admit/release to transfer data from VSIPL++ views to a regular array:

```
// admit the block so we write into it
//   (update = false since we're going to overwrite its values)
io_vec_.block().admit(false);

// write second half of range cell into io_vec_
io_vec_ = azbuf_(Domain<1>(npulse_, 1, npulse_));

// release block so we can use data directly
//   (update = true - we want to use the values)
io_vec_.block().release(true);

// perform IO
write(io_buf_, ...);
```

Shifting the data

Finally, in prepreation for the next frame, the data cube is shifted.

```
Domain<2> first_dom (npulse_, nrange_);
Domain<2> second_dom(Domain<1>(npulse_, 1, npulse_),
                     nrange_);

buf(first_dom)  = buf(second_dom);
buf(second_dom) = 0.f;
```

1.7.6 Utilities

Handling I/O

The general approach to importing data into VSIPL++ is to use user-storage. For example, to read a vector of size N stored in a file, the following pattern might be used:

```
// setup data structures:
float*          raw = new float[N];       // raw buffer
Dense<1, float> block(Domain<1>(N), raw);// user-storage block
Vector<float>   view(block);

// read data into buffer
FILE* fd = fopen("filename", "r");
fread(data, sizeof(float), N, fd);
fclose(fd);

// admit block with update flag = true
block.admit(true);

... use view ...
```

The LoadView class provides this functionality, and generalizes it to support matrices and tensors. Using LoadView, the

This functionality is generalized to support matrices and tensors in the LoadView class:

```
// setup data structures, read file, and admit.
LoadView<1, float> load_icoef("filename", Domain<1>(N));

// view of date can be accessed through .view() member function:
... load_view.view()
```

LoadView takes two template parameters:

- Dim – dimension of the view (1 = vector, 2 = matrix, ...)
- T – the element type of the view.

LoadView is used to load the following data into the SAR program:

- Coefficients for the I and Q upper sideband filters.
- Convolution kernels for azimuth processing
- EQ/Taylor weights for each polarization.

1.8 VSIPL++ Glossary

The following is a short list of terminology used within the VSIPL++ specification.

Block: Every block is logically a contiguous array of data. Blocks provide element-wise operations to access data. Blocks have dimensionality associated with them. A block is *x* dimensional if it supports *x* dimensions. A block is *x, y* dimensional if is supports both *x* and *y* dimensions. A block may store data itself, or it may access data through other means such as other blocks or optimization methods.

Cyclic_Contiguity: The number of contiguous values to consider as a unity when distributing in a round-robin fashion. Cyclic with a cyclic_contiguity greater than one is often called "block cyclic. A block distribution can be considered to be a degenerate cyclic distribution if the cyclic_contiguity is the same as the subblock size.

Data Distribution: Maps in VSIPL++ use three distributions, block_dist, cyclic_dist, and whole_dist. Fig. 1 contains examples of block_dist and cyclic_dist.

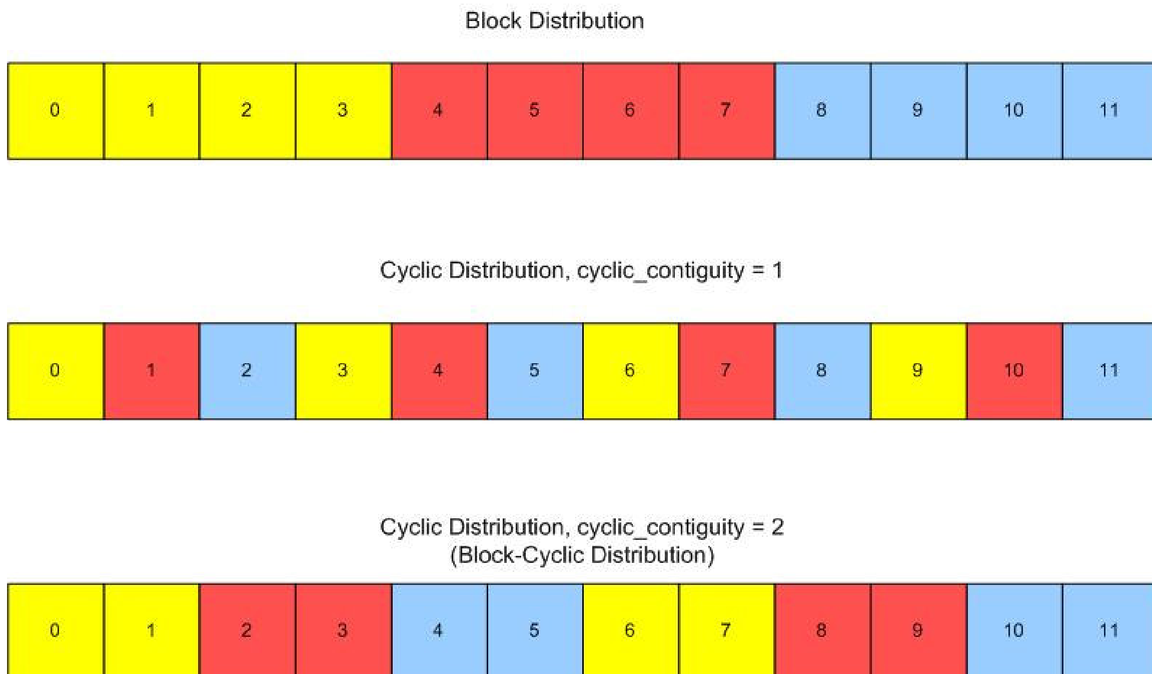

Fig. 1 Block and Cyclic Distributions

In this example, there are twelve elements in a 1D block to be distributed over three processors. For the block distribution, the elements are partitioned into subblocks by partitioning into three contiguous groups. The other two distributions illustrated, cyclic and block-cyclic are both variants on cyclic_dist. A cyclic distribution partitions the block by assigning indices in round-robin fashion. The difference between the cyclic and block-cyclic distributions is the cyclic_contiguity. The distribution is block-cyclic if the cyclic_contiguity is greater than one. In this example, the cyclic contiguity is 2.

whole_dist is not illustrated. A whole distribution subblock contains the entire block.

In the case of multiple dimensional blocks, each dimension can have its own distribution which is independent of the other dimensions distributions. In the following example, a two dimensional block has block_dist along the columns and whole_dist along the rows.

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)

Fig 2. 2D distribution with different distributions along the dimensions

Dense: The dense class is a block. This specialized class stores a value for each index in its block.

Domain: A domain has the following formal definition:

An Index<D> represents an element of N^D, where N is the set of nonnegative integers and D is a positive integral dimension. A Domain<D> represents a subset of N^D.

The purpose of the domain is to describe the data of interest in a block. A domain has one or more dimensions. Each dimension has a length and a stride which locates the indices in the dimension of the domain within the block. Note that there is no requirement that the stride be positive. Strides in a domain dimension can be negative or zero. The indices of a domain are not required to be contiguous.

In Fig. 3, a 1D domain starting at 1 with stride 2 and length 5 is chosen and highlighted in yellow. The odd indices of this 1D block are in the domain.

Fig. 3 A domain within a 1D block

Map: A map is an interface that describes how data stored in block can be distributed over processors. Applying a map to a block results in a set of subblocks that cover the indices of the original block.

Patch: A patch is a maximal subset of a subblock with contiguous indices. In Fig.1, the patch for the block distribution contains four indices. For the cyclic distribution example, the patch has one index, and in the case of the block cyclic example, the patch contains two indices.

Subblock: A subblock is the result of applying a map to a block. Applying a map to a VSIPL++ block yields a set of disjoint subblocks, whose union contains all the block's indices. Each subblock is an ordered set of indices. There is no requirement that the indices of a subblock be contiguous.

Subview: A subview of a view object refers to all or part of the original view's block. A subview uses reference semantics. When a subview modifies an element in its block at some index i in its domain, the element at index i in the domain of the original view is also modified.

View: A view is a logically a contiguous array with *D* dimensions. The domain associated with a view provides the indices that are used to access the elements of the view. The size of the view is determined at creation and cannot be modified.

Fig. 4 shows the view associated with the domain described in Fig. 3**.**

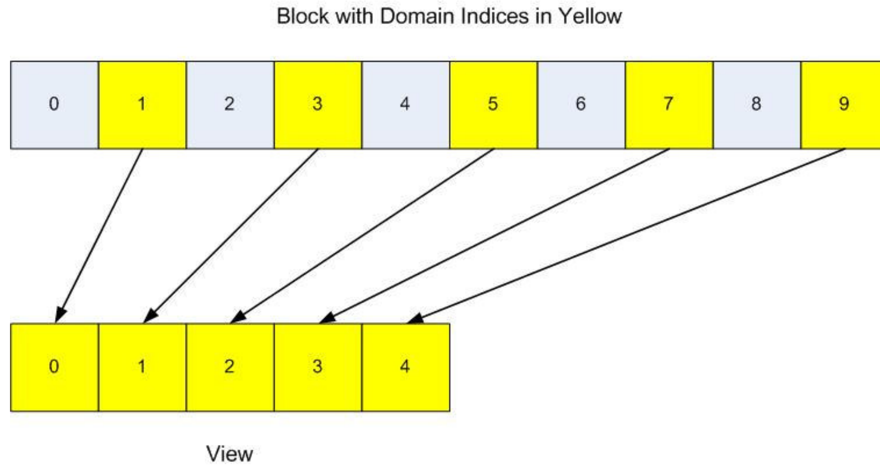

Fig 4. A view based on the domain in Fig. 3