Sourcery VSIPL++

User's Guide

Version 2.2-9

Sourcery VSIPL++: User's Guide: Version 2.2-9

CodeSourcery, Inc.

Table of Contents

I. Working with Sourcery VSIPL++ 1
1. API overview 2
1.1. Views 3
1.2. Blocks 5
1.3. Matlab IO 7
2. Direct Data Access 13
2.1. Introduction 14
2.2. Basic usage 14
2.3. Non-dense blocks 14
2.4. Requesting a specific data layout 15
3. Using the Dispatch Framework 16
3.1. Introduction 17
3.2. Compile-time dispatch 17
3.3. Runtime dispatch 19
4. Custom Expression Evaluation 21
4.1. The problem 22
4.2. Expression templates 22
4.3. Expression templates in Sourcery VSIPL++ 23
4.4. Creating custom expressions 23
4.5. Creating custom evaluators 26
5. Profiling 29
5.1. Enabling Profiling 30
5.2. Using the Profiler 31
5.3. Profiler Output 32
6. Benchmarking 35
6.1. Overview 36
6.2. Obtaining the Benchmarks 37
6.3. Running Performance Tests 38
6.4. Benchmark Output 39
Glossary 42
II. Example Application 44
7. Fast Convolution 46
7.1. Fast Convolution 47
7.2. Serial Optimization: Temporal Locality 51
7.3. Performing I/O with User-Specified Storage 53
7.4. Performing I/O with External Data Access 56
8. Parallel Fast Convolution 58
8.1. Parallel Fast Convolution 59
8.2. Improving Parallel Temporal Locality 62
8.3. Performing I/O 68
A. Benchmark Options 74
B. Benchmark Usage 78
B.1. CELL Benchmark Usage 79
B.2. conv 79
B.3. conv2d 79
B.4. copy 80
B.5. corr 80
B.6. CUDA Benchmark Usage 81
B.7. CVSIP Benchmark Usage 83
B.8. dist_vmul 85
B.9. dot 85

B.10. fastconv 85
B.11. fft 86
B.12. fftm 87
B.13. FFTW3 Benchmark Usage 87
B.14. fir 88
B.15. HPEC_KERNEL Benchmark Usage 89
B.16. IPP Benchmark Usage 91
B.17. LAPACK Benchmark Usage 92
B.18. maxval 92
B.19. mcopy 93
B.20. meansqval 93
B.21. meanval 93
B.22. memwrite 94
B.23. memwrite_simd 94
B.24. MPI Benchmark Usage 94
B.25. prod 96
B.26. prod_var 96
B.27. pwarp 96
B.28. SAL Benchmark Usage 97
B.29. sfilt 98
B.30. sort 99
B.31. stream 99
B.32. sumsqval 99
B.33. sumval 100
B.34. sumval_simd 100
B.35. svmul 100
B.36. task_switch 100
B.37. vdiv 101
B.38. vma 101
B.39. vmagsq 101
B.40. vmmul 102
B.41. vmul 102
B.42. vmul_c 103
B.43. vmul_par 103

Part I. Working with Sourcery VSIPL++

Basic idioms and programming model

The chapters in this part provide information about fundamentals of Sourcery VSIPL++. You should also refer to the VSIPL++ API Specification and Sourcery VSIPL++ Reference Manual, both of which are available at http://www.codesourcery.com/vsiplplusplus.

Chapter 1
API overview

1.1. Views

VSIPL++ defines a number of mathematical types for linear algebra: vectors, matrices, and (3D) tensors. They provide a high-level interface suitable for solving linear algebra equations. All these types give an intuitive access to their elements. They are collectively referred to as views as the actual data they provide access to is sharable among views.

```
// create an uninitialized vector of 10 elements
Vector<float> vector1(10);

// create a zero-initialized vector of 10 elements
Vector<float> vector2(10, 0.f);

// assign vector2 to vector1
vector1 = vector2;

// set the first element to 1.f
vector1(0) = 1.f;

// access the last element
float value = vector1(9);
```

Every view has an associated `Block`, which is responsible for storing or computing the data in the view. More than one view may be associated with the same block.

Depending on how a view is constructed it may allocate the block, or refer to a block from another view. All views created via copy-construction will share the blocks with the views they were constructed with.

```
// copy-construct a new vector from an existing one
Vector<float> vector3(vector1);

// modify the original vector
vector1.put(1, 1.f);

// the new vector reflects the new value
assert(vector3(1) == 1.f);
```

1.1.1. Domains

A domain represents a logical set of indices. Constructing a one-dimensional domain requires a start index, a stride, and a length. For convenience an additional constructor is provided that only takes a length argument, setting the starting index to 0 and the stride to 1.

```
// [0...9]
vsip::Domain<1> all(10);

// [0, 2, 4, 6, 8]
vsip::Domain<1> pair(0, 2, 5);

// [1, 3, 5, 7, 9]
vsip::Domain<1> impair(1, 2, 5);
```

Two- and three-dimensional domains are composed out of one-dimensional ones.

```
// [(0,0), (0,2), (0,4),...,(1,0),...]
vsip::Domain<2> dom(Domain<1>(10), Domain<1>(0, 2, 5));
```

Views provide convenient access to subviews in terms of subdomains. For example, to assign new values to every second element of a vector, simply write:

```
// assign 1.f to all elements in [0, 2, 4, 6, 8]
vector1(pair) = 1.f;
```

All complex views provide real and imaginary subviews:

```
// a function manipulating a float vector in-place
void filter(Vector<float>);

// create a complex vector
Vector<complex> vector(10);

// filter the real part of the vector
filter(vector.real());
```

1.1.2. Elementwise Operations

VSIPL++ provides elementwise functions and operations that are defined in terms of their scalar counterpart.

```
Vector<float> vector1(10, 1.f);

Vector<complex<float> > vector2(10, complex<float>(2.f, 1.f));

// apply operator+ elementwise
Vector<complex<float> > sum = vector1 + vector2;

// apply conj(complex<float>) elementwise
Vector<complex<float> > result = conj(sum);
```

For binary and ternary functions VSIPL++ provides overloaded versions with mixed view / scalar parameter types:

```
// delegates to operator*=(complex<float>, complex<float>)
result *= complex<float>(2.f, 0.f);

// error: no operator*=(complex<float>, complex<double>)
result *= complex<double>(5., 0.);
```

1.1.3. Vectors

1.1.4. Matrices

Matrices provide a number of additional subviews. All of them

```
Matrix<float> matrix(10, 10);
//...

// return the first column vector
```

```
Matrix<float>::col_type column = matrix.col(0);

// return the first row vector
Matrix<float>::row_type row = matrix.row(0);

// return the diagonal vector
Matrix<float>::diag_type diag = matrix.diag();

// return the transpose of the matrix
Matrix<float>::transpose_type trans = matrix.trans();
```

1.1.5. Tensors

Tensors are three-dimensional views. In addition to the types, methods, and operations defined for all view types, they provide additional methods to access specific subviews:

```
// a 5x6x3 cube initialized to 0.f
Tensor<float> tensor(5, 6, 3, 0.f);

// a subvector
Vector<float> vector1 = tensor(0, 0, whole_domain);
```

The symbolic constant `whole_domain` is used to indicate that the whole domain the target view holds in a particular dimension should be used. In the example above that not only provides a more compact syntax compared to explicitly writing Domain<1>(6) but it also enables better optimization opportunities.

```
// a submatrix
Matrix<float> plane = tensor(whole_domain, 0, whole_domain);

Tensor<float> upper_half = tensor(whole_domain, Domain<1>(3), \
whole_domain);
```

1.2. Blocks

The data accessed and manipulated through the View API is actually stored in blocks. Blocks are reference-countable, allowing multiple views to share a single block. However, blocks may themselves be proxies that access their data from other blocks (possibly computing the actual values only when these values are accessed). These blocks are thus not modifiable. They aren't allocated directly by users, but rather internally during the creation of subviews, for example.

1.2.1. Dense Blocks

The default block type used by all views is `Dense`, meaning that `Vector<float>` is actually a shorthand notation for `Vector<float, Dense<1, float> >`. As such `Dense` is the most common block type directly used by users. Dense blocks are modifiable and allocatable. They explicitly store one value for each index in the supported domain:

```
// create uninitialized array of size 3
Dense<1, float> array1(Domain<1>(3));

// create array of size 3 with initial values 0.f
Dense<1, float> array2(Domain<1>(3), 0.f);
```

```
// assign array2 to array1
array1 = array2;

// access first item
float value = array1.get(0);

// modify first item
array1.put(0, 1.f);
```

1.2.1.1. Layout

Beside the two template parameters already discussed above, `Dense` provides an optional third parameter to specify its dimension ordering. Using this parameter you can explicitly control whether a 2-dimensional array should be stored in row-major or column-major format:

```
// array using row-major ordering
Dense<2, float, tuple<0, 1> > rm_array;

// array using column-major ordering
Dense<2, float, tuple<1, 0> > cm_array;
```

Row-major arrays store rows as contiguous chunks of memory. Iterating over its columns will thus access close-by memory regions, reducing cache misses and thus enhancing performance:

```
length_type size = rm_array.size(0);
for (index_type i = 0; i != size; ++i)
  rm_array.put(i, 1.f);
```

1.2.1.2. User Storage

They also allow user-storage to be provided, either at construction time, or later via a call to `rebind`:

```
float *storage = ...;

// create array operating on user storage
Dense<1, float> array3(Domain<1>(3), storage);

// create uninitialized array...
Dense<1, float> array4(Domain<1>(3));

// ...and rebind it to user-storage
array4.rebind(storage);
```

However, special care has to be taken in these cases to synchronize the user storage with the block using it. While the storage is being used via the block it was rebound to, it has to be *admitted*, and *released* in order to be accessed directly, i.e. outside the block.

```
// grant exclusive access to the block
array3.admit();

// modify it
array3.put(0, 1.f);

// force synchronization with storage
array3.release();
```

```
// access storage directly
assert(storage == 1.f);
```

1.3. Matlab IO

Sourcery VSIPL++'s `vsip_csl` library has routines that can read and write views from Matlab formated text and binary files.

1.3.1. Matlab Text (.m) Files

The `Matlab_text_formatter` object writes a view to an output stream in Matlab text file format (Matlab text files commonly have an `.m` suffix).

The following example illustrates using the `Matlab_text_formatter` to write a matrix and a vector to the same file.

The first part of the example shows the necessary `include`s and declarations. In addition to any VSIPL++ headers necessary for your program (the example includes headers for vectors, matrices, and generation functions), it is also necessary to include the `vsip_csl/matlab_text_formatter.hpp` header file. The example also uses the `vsip` and `vsip_csl` namespaces for convenience.

```
#include <iostream>
#include <fstream>

#include <vsip/initfin.hpp>
#include <vsip/vector.hpp>
#include <vsip/matrix.hpp>
#include <vsip/selgen.hpp>
#include <vsip/map.hpp>

#include <vsip_csl/matlab_text_formatter.hpp>

using namespace vsip;
using namespace vsip_csl;
```

The second part of the example shows writing a file. First matrix `m` and view `v` are created and filled with ramp data. Then an output file stream `out` is created. Finally, `Matlab_text_formatter` is used to write the views.

```
  // Initialize matrix 'm'.
  Matrix<float> m(3, 3);
  for(index_type i=0;i<3;i++)
    m.row(i) = ramp<float>(3*i, 1, 3);

  // Initialize vector 'v'.
  Vector<float> v(3);
  v = ramp<float>(0, 1, 3);

  // Open output stream to file 'temp.m'.
  std::ofstream out("text.m");

  // Write 'm' and 'a' to output stream
```

```
  out << Matlab_text_formatter<Matrix<float> >(m, "m");
  out << Matlab_text_formatter<Vector<float> >(v, "v");
```

The output file `temp.m` contains the following

```
m =
[
  [ 0 1 2 ]
  [ 3 4 5 ]
  [ 6 7 8 ]
];
v =
[ 0 1 2 ];
```

This text file can be run inside of a Matlab console window to load matrix `a` and vector `v`.

1.3.2. Matlab Binary Files (.mat)

The `Matlab_bin_formatter` object can read and write views to a streams in Matlab binary file format (Matlab binary files commonly have a `.mat` suffix). For reading matlab binary files, the iterator interface described in the next section may be more convenient.

1.3.2.1. Writing a Matlab Binary Format File

Writing matlab binary format files is similar to writing text format files, except that a header must be written to the file with `Matlab_bin_header` before writing each view with `Matlab_binary_formatter`. The following example shows how to write a matrix and a vector to a `.mat` file.

The first part of the example shows the necessary `include`s and declarations. In addition to any VSIPL++ headers necessary for your program (the example includes headers for vectors, matrices, and generation functions), it is also necessary to include the `vsip_csl/matlab_bin_formatter.hpp` header file. The example also uses the `vsip` and `vsip_csl` namespaces for convenience.

```
#include <iostream>
#include <fstream>

#include <vsip/initfin.hpp>
#include <vsip/vector.hpp>
#include <vsip/matrix.hpp>
#include <vsip/selgen.hpp>
#include <vsip/map.hpp>

#include <vsip_csl/matlab_bin_formatter.hpp>

using namespace vsip;
using namespace vsip_csl;
```

The second part of the example shows writing a file. First matrix `m` and view `v` are created and filled with ramp data. Then an output file stream `out` is created. Finally, `Matlab_text_formatter` is used to write the views.

```
  // Initialize matrix 'm'.
  Matrix<float> m(3, 3);
  for(index_type i=0;i<3;i++)
```

```
    m.row(i) = ramp<float>(3*i, 1, 3);

  // Initialize vector 'v'.
  Vector<float> v(3);
  v = ramp<float>(0, 1, 3);

  // Open output stream to file 'sample.mat'.
  std::ofstream out("sample.mat");

  // Write matlab binary format header.  This must is done once at \
the
  // beginning of the file before any views can be written.
  out << Matlab_bin_hdr("example");

  // Write 'm' and 'v' to output stream
  out << Matlab_bin_formatter<Matrix<float> >(m, "m");
  out << Matlab_bin_formatter<Vector<float> >(v, "v");
```

This result file `sample.mat` can be read by Matlab, or other programs capable of reading matlab binary format files, such as Octave and Sourcery VSIPL++ applications.

1.3.2.2. Reading a Matlab Binary Format File

Reading a matlab binary format file is similar to writing one. After the file is opened, it is necessary to read the file header. This header is used by each of the subsequent reads. The following example shows how to read the views back from the `sample.mat` file written in the previous example.

The same includes are used for this example.

First matrix `m` and view `v` are created. Their size must match the size of the views in the matlab binary file. Next an input stream is created to read the binary data. The header is read first into a `Matlab_bin_header`. Finally each view is read, using `Matlab_bin_formatter` objects.

```
  // Create matrix and vector views of correct size.
  Matrix<float> m(3, 3);
  Vector<float> v(3);

  // Open an input stream to read sample.mat.
  std::ifstream in("sample.mat");

  // Read matlab binary format file header.  This must be done once
  // after the file is opened before reading any views.  The header \
is
  // then used by Matlab_bin_formatter to determine global file
  // parameters such as endianness, etc.
  Matlab_bin_hdr h;
  in >> h;

  // Read the views.
  in >> Matlab_bin_formatter<Matrix<float> >(m, "m", h);
  in >> Matlab_bin_formatter<Vector<float> >(v, "v", h);
```

Note that when using `Matlab_bin_formatter` the size and types of the VSIPL++ views `m` and `v` must match the size and type of the views stored in the binary file. If they do not match, an exception

will be thrown. In situations were the size and type are not known in advance, it may be more convenient to use the Matlab iterator interface, described in the next section.

1.3.3. Matlab_file iterator interface

In situations where the size, type, and order of views written in a matlab file is not known in advance, the `Matlab_file` interface should be used to read the file. `Matlab_file` provides an iterator interface to step through each view in a file. The size, type, and name of each view can be queried before it is read. This allows an appropriate VSIPL++ view to be constructed dynamically.

The `Matlab_file` object handles opening and reading a file. It provides a standard iterator interface with `begin` and `end` functions.

`Matlab_file::iterator`s correspond to views in the file. Dereferencing the iterator returns a `Matlab_view_header` object. This contains information about the view, including its name, type, dimensionality, and size.

The `read_view` function reads the view referred to by the iterator.

The following example shows how to read the `m` matrix from the `sample.mat` binary file used in the previous examples.

The first part of the example shows the necessary `include`s and declarations. The `Matlab_file` interface is contained in the `vsip_csl/matlab_file.hpp` header file.

```
        #include <iostream>
#include <fstream>

#include <vsip/initfin.hpp>
#include <vsip/vector.hpp>
#include <vsip/matrix.hpp>
#include <vsip/selgen.hpp>
#include <vsip/map.hpp>

#include <vsip_csl/matlab_file.hpp>

using namespace vsip;
using namespace vsip_csl;

      \
```

The second part of the example shows how to read the file. First the `Matlab_file` object `mf` is created. Then iterators `begin` and `end` are created to iterate over the views stored in the file. For each view, the name and size are checked to determine if it should be read. Finally, `read_view` is used to read the selected view.

```
         // Create Matlab_file object for 'sample.mat' file.
  Matlab_file mf("sample.mat");
  Matlab_file::iterator cur = mf.begin();
  Matlab_file::iterator end = mf.end();
  Matlab_view_header* vhdr;

  // Block pointer to hold the matrix.  The block will be allocated
```

```
  // once it's size is known.
  Dense<2, float>* m_block = NULL;

  // Iterate through views in file.
  while (cur != end)
  {
    vhdr = *cur;

    // Check if view is the one we're looking for.
    if(!strcmp(vhdr->array_name, "m") && vhdr->num_dims == 2)
    {
      // Check for multiple views named "m" in file.
      assert(m_block == NULL);

      // Create block and view.
      // At this point we can make the block size match size in \
the file.
      m_block = new Dense<2, float>(Domain<2>(vhdr->dims[0], \
vhdr->dims[1]));
      Matrix<float> tmp(*m_block);

      // Read view from file.
      mf.read_view(tmp, cur);
    }

    ++cur; // Move to next view stored in the file.
  }

  // Check that we found a view named "a" in file.
  assert(m_block != NULL);

  // Create a view to process "m".
  Matrix<float> m(*m_block);

      \
```

The handling of vectors in Matlab files requires special consideration. Matlab stores vectors as matrices with one dimensions of size 1. VSIPL++ can read matlab vectors as either VSIPL++ vectors or matrices.

The following example shows how to read the `v` vector from the `sample.mat` binary file used in the previous examples.

```
         // Create Matlab_file object for 'sample.mat' file.
  Matlab_file mf("sample.mat");
  Matlab_file::iterator cur = mf.begin();
  Matlab_file::iterator end = mf.end();
  Matlab_view_header* vhdr;

  // Block pointer to hold the vector.  The block will be allocated
  // once it's size is known.
  Dense<1, float>* v_block = NULL;
```

```
  // Iterate through views in file.
  while (cur != end)
  {
    vhdr = *cur;

    // Check if view is the one we're looking for.
    //
   // Note: even though 'v' is a vector, it will be 2D because of \
how
    //       matlab stores vectors.
    if(!strcmp(vhdr->array_name, "v") && vhdr->num_dims == 2)
    {
      // Check for multiple views named "m" in file.
      assert(v_block == NULL);

      // Determine the vector's size:
      length_type size = std::max(vhdr->dims[0], vhdr->dims[1]);

      // Create block and view.
      // At this point we can make the block size match size in \
the file.
      v_block = new Dense<1, float>(Domain<1>(size));
      Vector<float> tmp(*v_block);

      // Read view from file.
      mf.read_view(tmp, cur);
    }

    ++cur; // Move to next view stored in the file.
  }

  // Check that we found a view named "a" in file.
  assert(v_block != NULL);

  // Create a view to process "m".
  Vector<float> v(*v_block);

      \
```

Chapter 2
Direct Data Access

2.1. Introduction

While views and blocks provide a powerful abstraction to handle data inside the Sourcery VSIPL++ API, it may be necessary to access the raw data, for example to pass them to third-party libraries or legacy code that requires data to be cast into different types.

To be able to cast Blocks into third-party types, without being required to copy the data, Sourcery VSIPL++ provides a *Direct Data Access* API.

Unless stated otherwise, all of the types discussed here are part of the *vsip_csl::dda* namespace.

2.2. Basic usage

Direct data access is provided by the `Ext_data<>` class. In its simplest form, it can be instantiated with a block, and provides access to the data as a raw pointer by means of the `data()` accessor.

```
void process(float *data, size_t size);
...
Vector<float> v(8);
vsip_csl::dda::Ext_data<Vector<float>::block_type> ext(v.block());
process(ext.data(), ext.size());
```

This works, as long as the underlying block type is known to have unit stride. If this assumption can not be made, the access needs to take the stride into account.

2.3. Non-dense blocks

In the following example, we construct a subview of a dense view, aliasing every second value from a dense view, yielding a view with stride `2`:

```
void process(float *data, ptrdiff_t stride, size_t size);
...
Vector<float> v(8);
Vector<float>::subview_type subview = view.get(Domain<1>(0, 2, 4));
vsip_csl::dda::Ext_data<Vector<float>::block_type> ext(v.block());
process(ext.data(), ext.stride(), ext.size());
```

However, some functions may require unit-stride input. In that case, it is possible to force unit-stride access. The `Ext_data<>` object will copy the data into temporary storage, which the user then operates on, and synchronize back with the block it was constructed from. This synchronization may not always be necessary, and so it is possible to express whether to synchronize only from the block to the `Ext_data<>` object, the inverse, or both.

```
typedef Vector<float>::subview_type::block_type block_type;
typedef dda::Layout<1, row1_type, dda::Stride_unit> layout_type;

dda::Ext_data<block_type, layout_type> ext(subview.block(), \
dda::SYNC_OUT);
ramp(ext.data(), ext.size());
```

As this access type may involve a performance penalty (temporary data allocation, as well as one or two copy operations), it is desirable to be able to query whether the direct data access comes with

an extra cost. A user may decide to prefer unit-stride, as long as no copies are involved, but fall back to non-unit stride access otherwise:

```
typedef Vector<float>::subview_type::block_type block_type;
typedef dda::Layout<1, row1_type, dda::Stride_unit> layout_type;
if (dda::Ext_data<block_type, layout_type>::CT_Cost != 0)
{
  // If unit-stride access would require a copy,
  // choose non-unit stride access
  dda::Ext_data<block_type> ext(subview.block());
  ramp(ext.data(), ext.stride(0), ext.size());
}
else
{
  dda::Ext_data<block_type, layout_type> ext(subview.block(), \
dda::SYNC_OUT);
  ramp(ext.data(), ext.size());
}
```

Note that the `Ext_data<>::CT_cost` value is a compile-time constant, and thus can be used in compile-time expressions. Therefore, the above conditional may be done via compile-time decisions (e.g. template specializations).

In the above example we have considered unit- versus non-unit stride direct data access. However, there are other cases where data can never be accessed directly without copies, for example if the block represents an expression.

2.4. Requesting a specific data layout

The DDA API allows to express a very rich set of layout requirements. An aligned unit-stride (but not necessarily dense) row-major matrix can be requested like this:

```
typedef dda::Layout<2, row2_type, dda::Stride_unit_align<32> > \
layout_type;
dda::Ext_data<block_type, layout_type> ext(view.block(), \
dda::SYNC_IN);
```

And a dense column-major matrix may be requested like this:

```
typedef dda::Layout<2, col2_type, dda::Stride_unit_dense> \
layout_type;
dda::Ext_data<block_type, layout_type> ext(view.block(), \
dda::SYNC_IN);
```

For details on the Layout class template, see Section 5.2, “The Layout template”

Chapter 3
Using the Dispatch Framework

3.1. Introduction

Writing High-Performance code for a wide range of hardware is very challenging. Typically, the software is targetted at particular hardware or optimized for a specific set of parameters. This article describes a mechanism to interface a set of functions covering the same functionality but for different hardware or types of input with a single API, using a mechanism to dispatch to the most appropriate backend.

Sourcery VSIPL++ is configurable to target a wide range of backend implementations for most of its functions. it achives portability by hiding these backends behind common interfaces, yet strives to minimize the calling overhead by doing as much as possible at compile-time.

When the user performs a particular operation (e.g., adding two vectors) the library must select an appropriate implementation. For example, if the vectors are single-precision floating-point types, then a special SIMD routine might be used to perform the additional efficiently. Or, if the vectors are distributed across processors, multi-processor communication might be required.

When determining how to implement a given operation, Sourcery VSIPL++ performs a two-step process. One step is performed at compile-time; the other at run-time. Conceptually, the process is as follows:

1. Sourcery VSIPL++ forms a list of all possible implementations of the operation.

2. At compile-time, those implementations which do not accept arguments of appropriate types, or which are otherwise inappropriate for reasons which can be determined statically, are eliminated.

3. At run-time, each implementation not yet eliminated at compile-time is queried to see whether it can perform the operation. The first implementation that is able to perform the operation is used.

Each implementation is provided as a (possibly partial) specialization of the Evaluator class template. The library checks the `ct_valid` static data member to determine compile-time suitability and calls the `rt_valid()` static member function at run-time to determine run-time suitability. The actual implementation of the operation is performed by the `exec()` static member function.

3.2. Compile-time dispatch

Let us assume we want to implement an operation `Operation` by means of three backend classes `A_impl`, `B_impl`, and `C_impl`, all modeling the same concept:

```
class A_impl
{
  static void process(int*, size_t);
};

template <typename T>
class B_impl
{
  static void process(T*, size_t);
};

template <typename T>
class C_impl
{
```

```
  static void process(T*, size_t);
};
```

For each backend, we define an Evaluator, that is, a meta-function that will be used to evaluate whether for a given type this backend is usable.

Then we expose these backends by declaring a list of backends for the operation in question, so the dispatcher can iterate over it to find the first match. (Note that typically this list will also depend on configuration parameters, so the selection of the appropriate backend is in fact done in part at configure time and in part at compile time.

```
#include <vsip/opt/dispatch.hpp>

struct Operation;
struct A;
struct B;
struct C;

namespace vsip_csl
{
namespace dispatcher
{
template <>
struct Evaluator<Operation, A, int>
{
  static bool const ct_valid = true;
  typedef A_impl backend;
};

template <typename T>
struct Evaluator<Operation, B, T>
{
  static bool const ct_valid = has_feature<T>::value;
  typedef B_impl<T> backend;
};

template <typename T>
struct Evaluator<Operation, C, T>
{
  static bool const ct_valid = true;
  typedef C_impl<T> backend;
};

template <>
struct List<Operation>
{
  typedef Make_type_list<A, B, C>::type type;
};

}
}
```

Here, the Evaluator declares `A` to be available for `int` types, `B` is defined in a way that allows some external meta-function has_feature to evaluate the availability of this backend for a given type, while

Evaluator C declares C to be available for all types. (Typically, a catch-all backend is made available that is guaranteed to work for all types, but as it is slow, it is only available if no other backend matches.)

With that, writing the wrapper function that will do the actual dispatch is very simple:

```
template <typename T>
void process(T *input, size_t size)
{
  // Evaluate the dispatch (at compile-time):
  using vsip_csl::dispatcher::Dispatcher<Operation, T>;
  typedef Dispatcher<Operation, T>::backend backend_type;

  // Now use it.
  backend_type::process(input, size);
}
```

3.2.1. Implementation details

The `List<Operation>` type above creates a type-list of backend tags. The `Dispatcher<Operation>` class template constructs a type-list of Evaluators from that, which the compiler iterates over to select the first item for which the `ct_valid` member is `true`.

In other words, the `Dispatcher<Operation>` acts as a meta-function that takes a typename `T` as input, and returns a backend.

3.3. Runtime dispatch

Now let us make modifications to the above by stipulating that the function operates on an array of type `T` and size `size`, and backends have restrictions both on the type as well as the size of the array: Backend `A` only accepts `int`s, backend `B` accepts any type, but only buffers whose size is a power of 2, while backend C accepts any input:

```
namespace vsip_csl
{
namespace dispatcher
{
template <>
struct Evaluator<Operation, A, void(int*, size_t)>
{
  static bool const ct_valid = true;
  static bool rt_valid(int*, size_t) { return true;}
  static void exec(int *input, size_t size) { A_impl::process(input, \
size);}
};

template <typename T>
struct Evaluator<Operation, B, void(T*, size_t)>
{
  static bool const ct_valid = has_feature<T>::value;
  static bool rt_valid(T*, size_t size) { return size^(size-1);}
  static void exec(T *input, size_t size) { \
B_impl<T>::process(input, size);}
};
```

```
template <typename T>
struct Evaluator<Operation, C, void(T*, size_t)>
{
  static bool const ct_valid = true;
  static bool rt_valid(T*, size_t) { return true;}
  static void exec(T *input, size_t size) { \
C_impl<T>::process(input, size);}
};


template <>
struct List<Operation>
{
  typedef Make_type_list<A, B, C>::type type;
};

}
}
```

The operation wrapper for this now simply becomes:

```
template <typename T>
void process(T *input, size_t size)
{
  vsip_csl::dispatch<Operation, void(T*, size_t)>(input, size);
}
```

3.3.1. Implementation details

The runtime-dispatch works conceptually similar to the compile-time dispatch. However, in this case the dispatch is actually a two-phase process. The first phase is the same as in the compile-time dispatch. It reduces the type-list of Evaluators to those elements that match the given input type(s).

The second phase, then, is carried out at runtime, when this reduced type-list is traversed to evaluate all rt_valid() member functions until a match is found, based on runtime characteristics. In the case presented here this is a size parameter, though it could be anything else, as the signature of the operation to be carried out is a template parameter, too.

Chapter 4
Custom Expression Evaluation

4.1. The problem

The VSIPL++ API allows operations to be expressed in a concise mathematical way. Using operator overloading, it is possible to write expressions involving vectors and other VSIPL++ types, such as:

```
Vector<float> y(100), a(100), b(100), c(100), d(100);
y = (a+b)/(c-d);
```

However, this abstraction normally comes with a cost. Each binary operation is evaluated individually. This is equivalent to the following:

```
Vector<float> y(100), a(100), b(100), c(100), d(100);
Vector<float> tmp1 = a+b;
Vector<float> tmp2 = c-d;
Vector<float> tmp3 = tmp1/tmp2;
y = tmp3;
```

Thus, if the construction of those temporary objects is expensive, this adds considerable overhead. Further, for an element-wise definition of the above binary operations, each line performs a loop over all elements:

```
for (unsigned int i = 0; i != size; ++i) tmp1[i] = a[i] + b[i];
```

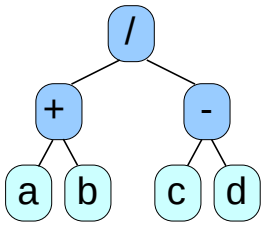
It would be desirable to avoid the temporary objects, and be able to fuse the different loops into one.

4.2. Expression templates

In Sourcery VSIPL++, the above operators are defined such that they don't evaluate the operation directly. Instead, they return a representation of the operation that can then be evaluated later. Using this technique, an entire expression will map to a single *expression object*, which then can be evaluated in a single pass, during the assignment.

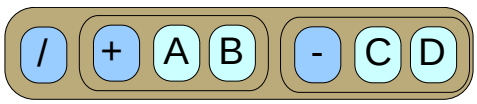
In other words, all such operators don't return a *value*, but a *parse tree* containing its operands, as well as operation

Figure 4.1. Parse tree representation of the expression `(a+b)/(c-d)`



With a suitable definition of those binary operators, this results into a result type like:

Figure 4.2. Expression type generated from the above expression `(a+b)/(c-d)`



If an assignment operator exists that takes such types as input, it may be able to perform a much faster assignment, without any temporaries.

4.3. Expression templates in Sourcery VSIPL++

4.3.1. Generating expression templates

Most functions and operators provided by Sourcery VSIPL++ are written to support such lazy evaluation. Thus, instead of calculating the result value directly, they return objects of such expression types. Moreover, they are written in such a way that their input may be expression type objects, too, such that expressions can be nested without being evaluated.

```
template <typename T, typename LeftBlock, typename RightBlock>
Vector<T, Binary_expr_block<1, Plus, LeftBlock, T, RightBlock, T> >
operator+(Vector<LeftBlock> left, Vector<RightBlock> right);
```

This version of `operator+` expects Vectors of arbitrary block types as input, and generates a Vector whose block type encodes the binary operation. (If the input arguments are themselves expressions, this will result in a composite expression tree.

Sourcery VSIPL++ provides a range of types to represent such *non-terminal* expression nodes, as well as the means to traverse them, to make it possible to generate expression (parse) trees for a wide range of expressions.

4.3.2. Evaluating expression templates

Expressions are ideally evaluated only once all the relevant information has been gathered. That point is typically reached once the whole assignment-expression has been seen. That is the case during assignment-operator evaluation.

Here is a conventional implementation of a vector-assignment, doing element-wise assignment:

```
template <typename T, typename LeftBlock, typename RightBlock>
void assign(Vector<T, LeftBlock> left, Vector<T, RightBlock> right)
{
  for (length_type i = Vector<T, LeftBlock>::size(); i; --i)
    left.put(i - 1, right.get(i - 1));
}
```

If `RightBlock` is an expression-block with an elementwise operation, its implementation of `get` will perform the (elementwise) evaluation. Thus, in this case, at least one temporary has already been eliminated, and multiple loops have been fused into one.

For non-elementwise operations, this is not quite as simple.

4.4. Creating custom expressions

Sourcery VSIPL++ allows you to write custom functions that participate in expression template dispatch and evaluation. This optimizes handling of the functions return value and allows custom evaluators to recognize fused expressions containing the expression.

Let us work through an example, starting with function scale():

```
template <typename T, typename BlockType>
Vector<T>
scale(Vector<T, BlockType> a, T value);
{
  Vector<T> r = a * value;
  return r;
}
```

This function takes a vector, scales it by a scalar value, and returns the result. As the result is returned by-value, it gets copied during assignment. In other words, the return value is a temporary object, which we may want to avoid.

To do that, we use a variant of a technique known as *return value optimization*. We rewrite `scale()` to return an expression type, which is only evaluated once the result object is available, so the computed value can be stored in-place. To do that, we capture the function logic into a functor, and rewrite the `scale()` function to return an *expression block* vector:

```
using vsip_csl::expr::Unary;
using vsip_csl::expr::Unary_functor;

// Scale implements a call operator that scales its input
// argument, and returns it by reference.
template <typename ArgumentBlockType>
struct Scale : Unary_functor<ArgumentBlockType>
{
  Scale(ArgumentBlockType const &a, typename \
ArgumentBlockType::value_type s)
    : Unary_functor<ArgumentBlockType>(a), value(s) {}

  template <typename ResultBlockType>
  void apply(ResultBlockType &r) const
  {
    ArgumentBlockType const &a = this->arg();
    for (index_type i = 0; i != r.size(); ++i)
      r.put(i, a.get(i) * value);
  }

  typename ArgumentBlockType::value_type value;
};

// scale is a return-block optimised function returning an \
expression.
template <typename T, typename BlockType>
lazy_Vector<T, Unary<Scale, BlockType> const>
scale(const_Vector<T, BlockType> input, T value)
{
  Scale<BlockType> s(input.block(), value);
  Unary<Scale, BlockType> block(s);
  return lazy_Vector<T, Unary<Scale, BlockType> const>(block);
}
```

With that improvement, the `scale()` function in

```
Vector<> a(8);
Vector<> r = scale(a, 2.f);
```

is entirely evaluated during the assignment.

The `Unary_functor` above poses certain requirements on its function parameter. If they can't be met, we need to write a different functor. For example:

```
using vsip_csl::View_block_storage;

template <typename ArgumentBlockType>
struct Interpolator
{
public:
  typedef typename ArgumentBlockType::value_type value_type;
  typedef typename ArgumentBlockType::value_type result_type;
  typedef typename ArgumentBlockType::map_type map_type;
  static vsip::dimension_type const dim = ArgumentBlockType::dim;

  Interpolator(ArgumentBlockType const &a, \
Domain<ArgumentBlockType::dim> const &s)
    : argument_(a), size_(s) {}

  // Report the size of the new interpolated block
  length_type size() const { return size_.size();}
  length_type size(dimension_type b, dimension_type d) const
  {
    assert(b == ArgumentBlockType::dim);
    return size_[d].size();
  }
  map_type const &map() const { return argument_.map();}

  ArgumentBlockType const &arg() const { return argument_;}

  template <typename ResultBlockType>
  void apply(ResultBlockType &) const
  {
    std::cout << "apply interpolation !" << std::endl;
    // interpolate 'argument' into 'result'
  }

private:
  typename View_block_storage<ArgumentBlockType>::expr_type \
argument_;
  Domain<ArgumentBlockType::dim> size_;
};
```

creates a new vector of different shape than the input Vector. To see the full requirements for the `UnaryFunctor`, see Section 6.6.3, “The Unary_functor class template”.

Here again, to write an `interpolate()` function that evaluates lazily, we need to return an *expression block* vector:

```
template <typename T, typename BlockType>
lazy_Vector<T, Unary<Interpolator, BlockType> const>
interpolate(lazy_Vector<T, BlockType> arg, Domain<1> const &size)
{
  typedef Unary<Interpolator, BlockType> expr_block_type;
  Interpolator<BlockType> interpolator(arg.block(), size);
  expr_block_type block(interpolator);
  return lazy_Vector<T, expr_block_type const>(block);
}
```

Now we can combine the above functions into a single expression:

```
Vector<float> a(8, 2.);
Vector<float> b = interpolate(scale(a, 2.f), 32);
```

The above demonstrates how to improve performance of an expression evaluation by using a technique that is a variant of the well-known *return value optimization*, where a copy operation (and a temporary object) may in certain cases be elided, if the result can be evaluated in-place.

4.5. Creating custom evaluators

In the previous section we have seen how to improve the expression evaluation by using the return-value optimization technique to avoid certain temporaries. However, there may be more that can be done to improve performance.

It may, for example, be possible to fuse multiple operations into one. Some platforms provide a fused "multiply-add" instruction that may be used, some algorithms are optimized for combined evaluation such as an FFT with a scalar multiplication, etc.

To be able to take advantage of those opportunities, we need to 'see' the whole expression at once, so we can dispatch the relevant sub-expression to such 'backends'.

For common cases, the library already performs this internally. However, sometimes users have their own optimized code that needs to be hooked into expression evaluation

In this section, we will develop an expression evaluator that matches the expression `interpolate(scale(a, 2.), 32)` from the last section.

Assignments are evaluated using the dispatch mechanism described in Chapter 3, “Using the Dispatch Framework”. To provide a custom evaluator for a particular expression assignment, it is thus necessary to specialize an `Evaluator`, using `op::assign<D>` as operation tag, and `be::user` as backend tag:

4.5.1. Specializing an evaluators for a particular expression type

To make Sourcery VSIPL++ use a custom evaluator, we need to specialize the `vsip_csl::dispatcher::Evaluator` template for the particular expression type we are interested in. Further, we need to model the evaluator concept.

The type of the expression can be discovered using `type_name()`:

```
std::cout << type_name(interpolate(scale(a, 2.), 32)) << std::endl;
```

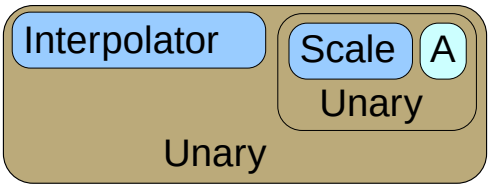
This yields (approximately):

```
vsip::Vector<float,
  vsip_csl::expr::Unary<example::Interpolator,
    vsip_csl::expr::Unary<example::Scale,
      vsip::Dense<1u, float, tuple<0u, 1u, 2u>, vsip::Local_map>,
        false> const>, false> const>
```

The block type is thus (with some details removed for clarity):

```
Unary<Interpolator, Unary<Scale, Dense<1>, false> const, false, \
const>
```

This can be visualized like this:



i.e., it is a `Unary` whose functor is an `Interpolator`. Its argument block, in turn, is a `Unary` whose functor is a `Scale`, and its argument block is a `Dense<1>`. This allows us to write a matching evaluator:

```
namespace vsip_csl
{
namespace dispatcher
{
template <typename ResultBlockType, typename ArgumentBlockType>
struct Evaluator<op::assign<1>, be::user
  void(ResultBlockType &,
       expr::Unary<Interpolator, expr::Unary<Scale, \
ArgumentBlockType> const> const &)>
{
  typedef typename ArgumentBlockType::value_type value_type;

  typedef ResultBlockType LHS;
  typedef expr::Unary<Interpolator, expr::Unary<Scale, \
ArgumentBlockType> const> RHS;

  static bool const ct_valid = true;
  static bool rt_valid(LHS &, RHS const &) { return true;}
  static void exec(LHS &lhs, RHS const &rhs) {} // TBD
};
```

This evaluator will match the desired expression. As usual, the `ct_valid` and `rt_valid()` members can be used to refine the selection process.

4.5.2. Accessing the expression nodes.

The terminals in this expression are the block, the interpolator's target size, as well as the scale value. Once these three are available, the entire expression may be evaluated in a fused `scaled_interpolate()`, as shown here:

```
static void exec(LHS &lhs, RHS const &rhs)
{
  // rhs.arg() yields Unary<Scale, ArgumentBlockType>,
  // rhs.arg().arg() thus returns the terminal ArgumentBlockType \
block...
  ArgumentBlockType &block = rhs.arg().arg();
  // ...and rhs.arg().operation() the Scale<ArgumentBlockType> \
functor.
  value_type scale = rhs.functor().argument.functor().func.value;

  // rhs.operation() yields the Interpolator<Unary<Scale, ...> \
functor.
  length_type new_size(rhs.operation().size(1, 0));

  // wrap terminal blocks in views for convenience, and evaluate.
  Vector<value_type, LHS> result(lhs);
  const_Vector<value_type, ArgumentBlockType const> argument(block);
  scaled_interpolate(result, argument, size, scale, new_size);
} \
```

Chapter 5
Profiling

This reference explains how to compile a program with profiling statements enabled, how to use the profiling functions in order to investigate the execution timing of a program and finally how to interprete the resulting profile data.

5.1. Enabling Profiling

5.1.1. Configure and Compile Options

There are no configure options for profiling, instead it is enabled via compile-time options. However, to use profiling it is necessary to configure the library with a suitable high-resolution timer (refer to the Quickstart for details on this and other configuration options). For example,

```
--enable-timer=x86_64_tsc
```

Pre-built versions of the library enable a suitable timer for your system.

You may enable profiling per category by defining the corresponding macros for all the categories you want to enable. On many systems, these macros may be added to the `CXXFLAGS` variable in the project makefile, e.g.

```
CXXFLAGS="-DVSIP_PROFILE_USER -DVSIP_PROFILE_MEMORY"
```

.

Table 5.1. Profiling Configuration Mask

Section	Description	Macro
memory	Memory management and block copying	VSIP_PROFILE_MEMORY
dispatch	Operation dispatch (incl. expression evaluation)	VSIP_PROFILE_DISPATCH
parallel	Parallel data I/O	VSIP_PROFILE_PARALLEL
func	Elementwise Functions	VSIP_PROFILE_FUNC
signal	Signal Processing	VSIP_PROFILE_SIGNAL
matvec	Matrix - Vector operations	VSIP_PROFILE_MATVEC
solver	Linear Algebra solvers	VISP_PROFILE_SOLVER
user	User-defined Operations	VSIP_PROFILE_USER
all	Everything	VSIP_PROFILE_ALL

The above macros may be defined on the command line, or in a header, as long as that is included before any Sourcery VSIPL++ header.

5.1.2. Command Line Options

For programs that have been compiled with profiling enabled, the profiling mode and output file can be controlled from the command line. You may profile programs without modifying your source files using this method. Use this to choose the profiler mode:

```
--vsip-profile-mode=mode
```

where *`mode`* is either `accum` or `trace`.

Specify the path to the log file for profile output using:

```
--vsip-profile-output=/path/to/logfile
```

The second option defaults to the standard output on most systems, so it may be omitted if that is desireable.

The profiling command line options control profiling for the entire program execution. For finer grain control, such as enabling profiling during a specific portion of the program, or to mix different profiling modes, explicit profiling objects can be created.

5.2. Using the Profiler

5.2.1. Profiling Objects

The `Profile` object is used to enable profiling during the lifetime of the object. When created, it takes arguments to indicate the output file and the profiling mode (trace or accumulate). When destroyed (i.e. goes out of scope or is explicitly deleted), the profile data is written to the specified output file. For example:

```
  profile::Profile profile("profile.txt", profile::accum)
```

During the lifetime of the Profile object, timing data is stored through a simple interface provided by the `Scope` object. These objects are used to profile library operations for the different areas mentioned in Table 5.1, “Profiling Configuration Mask” above. Any `Scope` objects defined in user programs fall into the 'user' category.

The declaration of an instance of this object starts a timer and when it is destroyed, the timer is stopped. The timing data is subsequently reported when the `Profile` object is destroyed. For example:

```
  profile::Scope<profile::user> scope("Scope Name", op_count);
```

The `profile::user` template argument indicates that this scope falls into the *user* category (and thus can be enabled with `-DVSIP_PROFILE_USER`). The first constructor argument is the name that will be used to display the scope's performance data in the log file (Section 5.3.2, “Scope names” describes the names used internally by the library.) The second parameter, `op_count`, is an optional unsigned integer specifying an estimate of the total number of operations (floating point or otherwise) performed. This is used by the profiler to compute the rate of computation. Without it, the profiler will still yield useful timing data, but the average rate of computation will be shown as zero in the log.

Creating a `Scope` object on the stack is the easiest way to control the region it will profile. For example, from within the body of a function (or as the entire function), use this to define a region of interest:

```
  {
    profile::Scope<profile::user> scope("Main computation:");

    // perform main computation
    //
    ...
  }
```

The closing brace causes `scope` to go out of scope, logging the amount of time spent doing the computation.

5.2.2. Profiler Modes

In `trace` mode, the start and stop times where scopes begin and end are stored as profile data. The log will present these events in chronological order. This mode is preferred when a highly detailed view of program execution is desired.

In `accum` (accumlate) mode, the start and stop times are subtracted to compute the time spent in a scope and the cumulative sum of these durations are stored as profile data. The log will indicate the total amount of time spent in each scope. This mode is desirable when investigating a specific function's average performance.

5.3. Profiler Output

5.3.1. Log File Format

The profiler outputs a small header at the beginning of each log file which is the same accumulate and trace modes. The data that follows the header is different depending on the mode. The header describes the profiling mode used, the low-level timer used to measure clock ticks and the number of clock ticks per second.

5.3.1.1. Accumulate mode

```
# mode: pm_accum
# timer: x86_64_tsc_time
# clocks_per_sec: 3591375104
#
# tag : total ticks : num calls : op count : mops
```

The respective columns that follow the header are:

tag	A descriptive name of the operation. This is either a name used internally or specified by the user.
total ticks	The time spent in the scope in processor ticks.
num calls	The number of times the scope was entered.
op count	The number of operations performed per scope.
mops	The calculated performance figure in millions of operations per second. (num_calls * op_count * 10^{-6}) / (total_ticks / clocks_per_sec)

5.3.1.2. Trace mode

```
# mode: pm_trace
# timer: x86_64_tsc_time
# clocks_per_sec: 3591375104
#
# index : tag : ticks : open id : op count
```

The respective columns that follow the header are:

index	The entry number, beginning at one.

tag — A descriptive name of the operation. This is either a name used internally or specified by the user.

ticks — The current reading from the processor clock.

open id — If zero, indicates the start of a scope. If non-zero, this indicates the end of an scope and refers to the index of corresponding start of the scope.

op count — The number of operations performed per scope, or zero to indicate the end of an scope.

Note that the timings expressed in 'ticks' may be converted to seconds by dividing by the 'clocks_per_second' constant in the header.

5.3.2. Scope names

Sourcery VSIPL++ uses the following names for profiling objects and functions within the library. These names are readable text containing information that varies depending on the operation being profiled.

5.3.2.1. Signal Processing and Matrix Vector Operations

These operations follow this general format:

```
OPERATION [DIM] DATATYPE SIZE
```

OPERATION gives the object or function name, including direction for FFTs.

DIM is the number of dimensions (when needed).

DATATYPE describes the data types involved in the operation. FFTs have two listed, describing both the input type as well as the output type, which may be different. See Table 5.2, “Data Type Names” below.

SIZE is expressed by giving the number of elements in each dimension.

The specific operations profiled at this time are:

```
Convolution [1D|2D] T SIZE
Correlation [1D|2D] T SIZE
Fft 1D [Inv|Fwd] I-O [by_ref|by_val] SIZE
Fftm 2D [Inv|Fwd] I-O [by_ref|by_val] SIZE
Fir T SIZE
Iir T SIZE
dot T SIZE
cvjdot T SIZE
trans T SIZE
herm T SIZE
kron T SIZE_A SIZE_B
outer T SIZE
gemp T SIZE
gems T SIZE
cumsum T SIZE
modulate T SIZE
```

In all cases, data types `T`, `I` and `O` above are expressed using a notation similar to the BLAS/LAPACK convention as in the following table:

Table 5.2. Data Type Names

	Views	Scalars
single precision real	S	s
single precision complex	C	c
double precision real	D	d
double precision complex	Z	z

5.3.2.2. Elementwise Functions

Element-wise expression tags use a slightly different format:

```
EVALUATOR DIM EXPR SIZE
```

The EVALUATOR indicates which VSIPL++ evaluator was dispatched to compute the expression.

DIM indicates the dimensionality of the expression.

EXPR is mnemonic of the expression shown using prefix notation, i.e.

```
operator(operand, ...)
```

Each operand may be the result of another computation, so expressions are nested, the parenthesis determining the order of evaluation.

SIZE is expressed by giving the number of elements in each dimension.

At this time, these evaluators are profiled:

`Expr_Loop` - generic loop-fusion evaluator.
`Expr_SIMD_Loop` - SIMD loop-fusion evaluator.
`Expr_Copy` - optimized data-copy evaluator.
`Expr_Trans` - optimized matrix transpose evaluator.
`Expr_Dense` - evaluator for dense, multi- dimensional expressions. Converts them into corresponding 1-dim expressions that are re-dispatched.
`Expr_SAL_*` - evaluators for dispatch to the SAL vendor math library.
`Expr_IPP_*` - evaluators for dispatch to the SAL vendor math library.
`Expr_SIMD_*` - evaluators for dispatch to the builtin SIMD routines (with the exception of Expr_SIMD_Loop, see above).
For SAL, IPP and SIMD, the asterisk (*) denotes the specific function invoked.

Chapter 6
Benchmarking

Abstract

This chapter describes how to build and run the Sourcery VSIPL++ benchmark suite in order to determine how the library performs on a given platform.

This chapter explains how to build and run the performance benchmarks supplied with Sourcery VSIPL++. It gives an overview of the benchmarks in the top-level directory, then goes into some detail about the platform-specific and problem-specific benchmarks found in the subdirectories therein.

6.1. Overview

The following tables describe the different benchmarks available currently. They are organized by type of operation, to allow more easy cross-referencing with the specification.

Table 6.1. Sourcery VSIPL++ Benchmark Descriptions

Operation	Source File
math, functions, elementwise - fused multiply add	vma.cpp
math, functions, elementwise - vector copy	copy.cpp
math, functions, elementwise - vector division	vdiv.cpp
math, functions, elementwise - vector magnitude squared	vmagsq.cpp
math, functions, elementwise - vector multiply	vmul.cpp
math, functions, elementwise - vector multiply, distributed	dist_vmul.cpp
math, functions, elementwise - vector multiply, parallel	vmul_par.cpp
math, functions, elementwise - vector multiply, using straight C code	vmul_c.cpp
math, functions, elementwise - vector-matrix multiply	vmmul.cpp
math, functions, reductions - maximum value	maxval.cpp
math, functions, reductions - sum of values	sumval.cpp
math, functions, reductions - sum of values, using SIMD	sumval_simd.cpp
math, matvec - matrix-matrix products	prod.cpp
math, matvec - matrix-matrix products, variations	prod_var.cpp
math, matvec - matrix copy, transpose	mcopy.cpp
math, matvec - vector dot product	dot.cpp
signal - convolution	conv.cpp
signal - correlation	corr.cpp
signal - fast convolution	fastconv.cpp
signal - Fast Fourier Transform	fft.cpp
signal - Fast Fourier Transform, multiple	fftm.cpp
signal - Finite Impulse Response filter	fir.cpp
view, vector, assign - memory write bandwidth	memwrite.cpp
view, vector, assign - memory write bandwidth, using SIMD	memwrite_simd.cpp

All of the above source files are located in `share/benchmarks/` in the top-level install directory, or simply `benchmarks/` if using the full source distribution. Other system-specific or library-specific benchmarks are contained in various subdirectories of the main benchmarks directory. Please refer to the `README` files in those subdirectories for more information.

6.2. Obtaining the Benchmarks

The performance benchmarks are built and are ready to run as soon as Sourcery VSIPL++ is installed. However, they may be rebuilt from source if desired. This section explains what to do in either case.

6.2.1. Binary Packages

If you installed Sourcery VSIPL++ from a binary package (the most common case), then no additional steps are necessary -- the executable images may be found in the `benchmarks/` subdirectory in the top-level install directory.

6.2.2. Source Packages

If you are using the full source package for Sourcery VSIPL++, the benchmarks may be built simply by typing

```
> make benchmarks
```

from the top-level source directory (the one containing GNUMakefile and the `benchmarks` subdirectory).

The makefile contains instructions to build only those benchmarks which are appropriate for a given system. For example, on a system with Mercury SAL installed, it will build the ones under `benchmarks/sal/` as well as the ones in `benchmarks/`.

If you are making changes to one particular benchmark and would like to rebuild just it, then provide the executable name as the make target. For example, to rebuild the elementwise vector multiply benchmark `vmul`, enter

```
> make benchmarks/vmul
```

6.2.3. Standalone

If you wish to build only the benchmarks themselves, without rebuilding the entire library, then you may use a stand-alone makefile provided for this purpose. First, make a copy of the benchmark sources from the `share/benchmarks/` subdirectory in the top-level install directory, or simply copy the `benchmarks/` directory from the source tree.

To rebuild all benchmarks, enter the following command:

```
> make -f make.standalone
```

To rebuild a certain benchmark, enter the target name as well. For example:

```
> make -f make.standalone vmul
```

6.2.3.1. Optmization Settings

In order to experiment with different optimization settings for the compiler when building the benchmarks, it may be helpful to know that the makefiles, both in the source tree and stand-alone, use pkg-config to extract the appropriate build flags for a given architecture and operating system. As with most GNU Make projects, these are stored in a variable named `CXXFLAGS`. You may alter those values by editing the makefile and specifying options after the ones extracted using pkg-config, or you may replace its value entirely with your own. Before making changes, it may be helpful to

capture the output from a clean rebuild for later comparison. Use this command or one similar to accomplish this:

```
> make -f make.standalone clean
> make -f make.standalone &> build.log
```

6.3. Running Performance Tests

Benchmarks are invoked as follows:

```
> benchmark -test-number [-option[ -option[ ...]]]
```

Test numbers are defined individually for each benchmark and represent various combinations of algorithms and parameters used for a given performance measurement. Valid test numbers begin at one. Most benchmarks utilize zero to display a list of valid tests numbers. For example:

```
> vmul -0
vmul -- vector multiplication
single-precision:
Vector-Vector:
-1 -- Vector<        float > * Vector<        float >
-2 -- Vector<complex<float>> * Vector<complex<float>>
-3 -- Vector<complex<float>> * Vector<complex<float>> (SPLIT)
-4 -- Vector<complex<float>> * Vector<complex<float>> (INTER)
-5 -- Vector<        float > * Vector<complex<float>>
Scalar-Vector:
-11 --                float   * Vector<        float >
-12 --                float   * Vector<complex<float>>
-13 --        complex<float>  * Vector<complex<float>>
-14 -- t_svmul2
-15 -- t_svmul2
-15 -- t_svmul3
-15 -- t_svmul4
-21 -- t_vmul_dom1
-22 -- t_vmul_dom1
-31 -- t_vmul_ip1
-32 -- t_vmul_ip1

double-precision:
(101-113)
(131-132)
>
```

While performance tests may be run simply by specifiying the test number, several options are provided making it possible to control a variety of useful parameters. These parameters affect either the way the test is run or the type of output provided in the results. A summary of the most useful options is provided below:

Controlling the range of problem sizes

`-start` *M* Sets the starting problem size to 2^*M*. Defaults to 2 (4 points).

`-stop` *M* Sets the stopping problem size to 2^*M*. Defaults to 21 (2097152 points).

Controlling the samples taken

`-samples S` Sets the number of samples taken to *S*. Defaults to one. When S > 2, the median value is reported.

`-ms time` Sets the goal time that each measurement should take, in hundredths-of-a-second. Defaults to 25 (250 ms).

Benchmark specific parameters

`-param value` Sets the user-specified parameter, where *value* is used in some instances to override a default value of a parameter, such as the number of rows or columns in an input matrix for example. The effect, if any, is dependent on each individual benchmark.

Reporting

`-pts` Report millions of points per second (MPT/s). This is the default.

`-ops` Reports millions of operations per second (MOP/s).

`-iob` Reports millions of input/output operations per second (MB/s) (by summing read and write iob_per_point).

`-all` Reports all three statistics: MPT/s, MOP/s, MB/s

6.4. Benchmark Output

The benchmark output depends on the command line options, but typically includes some meta information on the benchmark (name, ops/point, etc) and individual measurements for each problem size.

The header information, denoted by lines begging with "#", contains three important factors that are used to convert timing data into other meaningful units. The number of floating point operations is shown as `ops_per_point` and the number of reads or writes to and from memory are shown as `riob_per_point` and `wiob_per_point` respectively.

Following the header information are performance results. Each line contains data for a certain problem size (number of points), which is given in the first column.

The second column contains the measured (or median) values calculated from the timing measurements. The default is in points-per-second as indicated in the header under "metric". Alternatively, the values are in units as requested with the -pts, -ops, -iob option.

In other cases, three columns of measurements follow the size given in the first column. The values listed vary depending on the options specified, as outlined below:

`-all` Displays points per second, operations per second and the sum of the memory reads and writes per second (MPT/s, MOP/s, MB/s).

`-pts`, `-ops`, `-iob` Displays one of points per second, operations per second and the sum of the memory reads and writes per second, as requested.

`-samples S` With `-all`, three columns will be displayed, each containing the median value of the respective measurement. Without `-all`, the second column will contain the

median value and columns three and four will contain the minimum and maximum value for the selected measurement. Note: *S* must be greater than two in order to display the minimum and maximum values for `-pts`, `-ops` or `-iob`.

6.4.1. Examples

This example shows a very simple benchmark for vector-vector multiplication using complex values, defaulting to units of “points-per-second”:

```
> vmul.exe -2
# what             : t_vmul1
# ops_per_point(1) : 6
# riob_per_point(1): 16
# wiob_per_point(1): 8
# metric           : pts_per_sec
# start_loop       : 2981969
4 60.606903
8 123.195221
16 173.855408
32 207.837997
64 232.163071
...
```

The output is truncated, but continues on up until 2^21 points per vector.

To measure operations per second instead, use:

```
> vmul.exe -2 -ops
# what             : t_vmul1
# ops_per_point(1) : 6
# riob_per_point(1): 16
# wiob_per_point(1): 8
# metric           : ops_per_sec
# start_loop       : 2973904
4 377.566650
8 765.744446
16 1055.679321
32 1261.269653
64 1425.231567
...
```

To measure ops/sec, with the median of 3 samples of 0.5 seconds in duration each:

```
> vmul.exe -2 -ops -samples 3 -ms 50
# what             : t_vmul1
# ops_per_point(1) : 6
# riob_per_point(1): 16
# wiob_per_point(1): 8
# metric           : ops_per_sec
# start_loop       : 5934442
4 409.272583 398.208191 413.359711
8 854.137939 811.087402 854.964539
16 1132.262939 1087.544800 1137.489502
32 1317.902710 1297.148560 1342.707886
64 1483.941650 1453.872192 1501.823242
```

Note that this option is most often used when error bars are desired for plotting the performance data.

Glossary

Block	A block is an interface to a logically contiguous array of data. Blocks provide a means to organize the access to the data. They may store the data themselves, or access the data through other blocks. This abstraction provides important latitude for optimizations such as expression templates, or parallelism. Block types have to fulfill the requirements outlined in table 6.1 of the specification.
Dense Block	Dense blocks are modifiable, allocatable blocks that explicitely store one value for each index in its domain. The data layout is specified in terms of a template parameter, allowing storage to be optimized for particular operations (see dimension ordering). Dense blocks allow users to supply data storage, either at construction time, or later, in which case the block is 'rebound' to an alternate user storage.
Dimension Ordering	Dimension ordering refers to the layout of data in a multi-dimensional block, such as row-major or column-major. Dimension ordering has an impact on performance in operations involving loops over the data, as adjacent reads / writes may require a new cache-line to be fetched first.
Domain	A domain represents a logical set of indices for which views provide data. It may be a contiguous set of indices for dense matrices, or a non-contiguous set of indices for subviews.
Expression Block	Expression blocks are used to store mathematical expressions, allowing optimized evaluation. Conventionally, in an equation 'View A = B + C * D' the computation of A would require at least two temporaries, representing the results of the two binary operations. Additionally, the evaluation of each of these subexpressions implies a loop, resuling in suboptimal performance.

With expression blocks, the above expression will generate a block representing 'B + C * D', which is evaluated when assigned to 'A'. Specializations of expression blocks may use highly optimized functions to be called, depending on the specific types and subexpressions involved.

Map

A map specifies how a block can be devided into subblocks for the purpose of parallel execution. It defines how subblocks are to be assigned to processors.

Map types have to fulfill the requirements outlined in table 3.1 of the parallel specification.

View

A view represents the base for mathematical linear algebra operations, such as vectors, matrices, tensors. It has a dimension, a value_type, and a number of accessors to access and manipulate its values. The actual data are stored in blocks, to which views hold references internally.

Multiple views may share the same data, making copy operations for those views an inexpensive operation. All views are parametrized for two types: the view's value_type, as well as the underlaying block type.

View types have to fulfill the requirements outlined in table 6.3 of the specification.

Part II. Example Application

Table of Contents

7. Fast Convolution 46
    7.1. Fast Convolution 47
    7.2. Serial Optimization: Temporal Locality 51
    7.3. Performing I/O with User-Specified Storage 53
    7.4. Performing I/O with External Data Access 56
8. Parallel Fast Convolution 58
    8.1. Parallel Fast Convolution 59
    8.2. Improving Parallel Temporal Locality 62
    8.3. Performing I/O 68

Chapter 7
Fast Convolution

Abstract

This chapter describes how to create and run a serial VSIPL++ program with Sourcery VSIPL++ that performs fast convolution. You can modify this program to develop your own serial applications.

This chapter explains how to use Sourcery VSIPL++ to perform *fast convolution* (a common signal-processing kernel). First, you will see how to compute fast convolution using VSIPL++'s multiple FFT (Fftm) and vector-matrix multiply operations. Then, you will learn how to optimize the performance of the implementation.

7.1. Fast Convolution

Fast convolution is the technique of performing convolution in the frequency domain. In particular, the time-domain convolution f * g can be computed as F . G, where F and G are the frequency-domain representations of the signals f and g. A time-domain signal consisting of n samples can be converted to a frequency-domain signal in O(n log n) operations by using a Fast Fourier Transform (FFT). Substantially fewer operations are required to perform the frequency-domain operation F . G than are required to perform the time-domain operation f * g. Therefore, performing convolutions in the frequency domain can be substantially faster than performing the equivalent computations in the time domain, even taking into account the cost of converting from the time domain to the frequency domain.

One practical use of fast convolution is to perform the pulse compression step in radar signal processing. To increase the effective bandwidth of a system, radars will transmit a frequency modulated "chirp". By convolving the received signal with the time-inverse of the chirp (called the "replica"), the total energy returned from an object can be collapsed into a single range cell. Fast convolution is also useful in many other contexts including sonar processing and software radio.

In this section, you will construct a program that performs fast convolution on a set of time-domain signals stored in a matrix. Each row of the matrix corresponds to a single signal, or "pulse". The columns correspond to points in time. So, the entry at position (i, j) in the matrix indicates the amplitude and phase of the signal received at time j for the ith pulse.

The first step is to declare the data matrix, the vector that will contain the replica signal, and a temporary matrix that will hold the results of the computation:

```
// Parameters.
length_type npulse = 64; // number of pulses
length_type nrange = 256; // number of range cells

// Views.
typedef complex<float> value_type;
Vector<value_type> replica(nrange);
Matrix<value_type> data(npulse, nrange);
Matrix<value_type> tmp (npulse, nrange);
```

For now, it is most convenient to initialize the input data to zero. (In Section 7.3, “Performing I/O with User-Specified Storage”, you will learn how to perform I/O operations so that you can populate the matrix with real data.)

In C++, you can use the constructor syntax `T()` to perform "default initialization" of a type `T()`. The default value for any numeric type (including complex numbers) is zero. Therefore, the expression `value_type()` indicates the complex number with zero as both its real and imaginary components. In the VSIPL++ API, when you assign a scalar value to a view (a vector, matrix, or tensor), all elements of the view are assigned the scalar value. So, the code below sets the contents of both the data matrix and replica vector to zero:

```
data    = value_type();
replica = value_type();
```

The next step is to define the FFTs that will be performed. Typically (as in this example) an application performs multiple FFTs on inputs with the same size. Since performing an FFT requires that some set-up be performed before performing the actual FFT computation, it is more efficient to set up the FFT just once. Therefore, in the VSIPL++ API, FFTs are objects, rather than operators. Constructing the FFT performs the necessary set-up operations.

Because VSIPL++ supports a variety of different kinds of FFT, FFTs are themselves template classes. The parameters to the template allow you to indicate whether to perform a forward (time-domain to frequency-domain) or inverse (frequency-domain to time-domain) FFT, the type of the input and output data (i.e., whether complex or real data is in use), and so forth. Then, when constructing the FFT objects, you indicate the size of the FFT. In this case, you will need both an ordinary FFT (to convert the replica data from the time domain to the frequency domain) and a "multiple FFT" to perform the FFTs on the rows of the matrix. (A multiple FFT performs the same FFT on each row or column of a matrix.) So, the FFTs required are:

```
  // A forward Fft for computing the frequency-domain version of
  // the replica.
  typedef Fft<const_Vector, value_type, value_type, fft_fwd, \
by_reference>
  for_fft_type;
  for_fft_type  for_fft (Domain<1>(nrange), 1.0);

  // A forward Fftm for converting the time-domain data matrix to the
  // frequency domain.
  typedef Fftm<value_type, value_type, row, fft_fwd, by_reference>
    for_fftm_type;
  for_fftm_type for_fftm(Domain<2>(npulse, nrange), 1.0);

  // An inverse Fftm for converting the frequency-domain data back to
  // the time-domain.
  typedef Fftm<value_type, value_type, row, fft_inv, by_reference>
    inv_fftm_type;
  inv_fftm_type inv_fftm(Domain<2>(npulse, nrange), 1.0/(nrange));
```

Before performing the actual convolution, you must convert the replica to the frequency domain using the FFT created above. Because the replica data is a property of the chirp, we only need to do this once; even if the radar system runs for a long time, the converted replica will always be the same. VSIPL++ FFT objects behave like functions, so you can just "call" the FFT object:

```
  for_fft(replica);
```

Now, you are ready to perform the actual fast convolution operation! You will use the forward and inverse multiple-FFT objects you've already created to go into and out of the frequency domain. While in the frequency domain, you will use the `vmmul` operator to perform a vector-matrix multiply. This operator multiplies each row (dimension zero) of the frequency-domain matrix by the replica. The `vmmul` operator is a template taking a single parameter which indicates whether the multiplication should be performed on rows or on columns. So, the heart of the fast convolution algorithm is just:

```
  // Convert to the frequency domain.
  for_fftm(data, tmp);

  // Perform element-wise multiply for each pulse.
  tmp = vmmul<0>(replica, tmp);
```

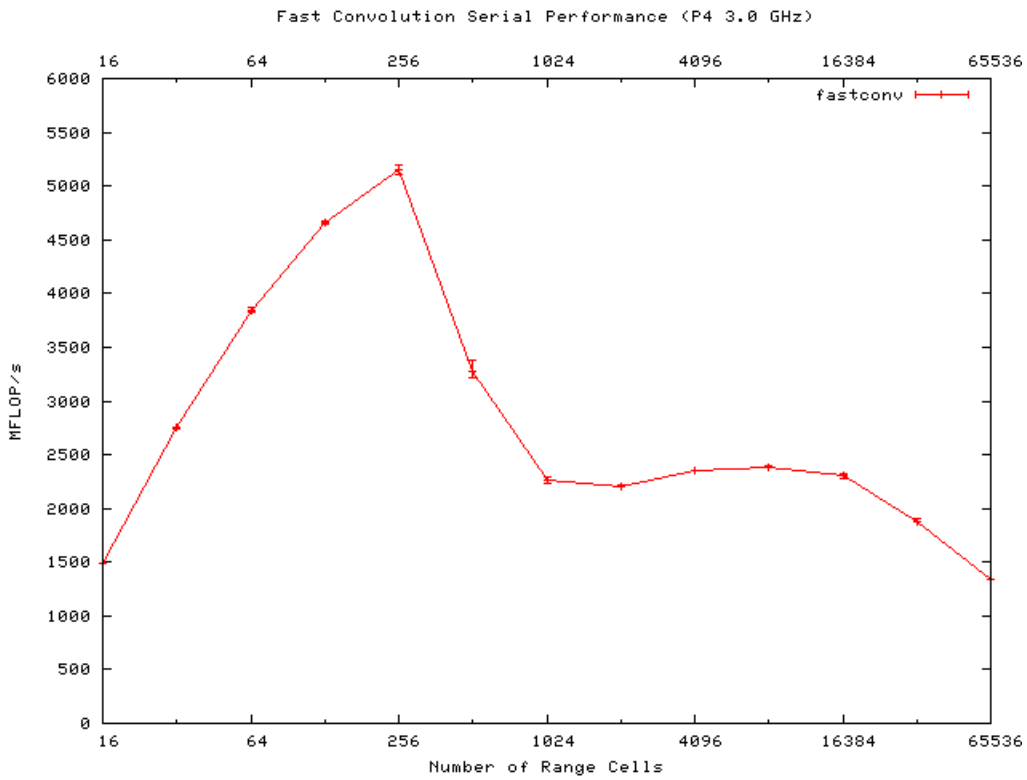
```
  // Convert back to the time domain.
  inv_fftm(tmp, data);
```

A complete program listing is show below. You can copy this program directly into your editor and build it. (You may notice that there are a few things in the complete listing not discussed above, including in particular, initialization of the library.)

```
/***************************************************************************
  Included Files
***************************************************************************/

#include <vsip/initfin.hpp>
#include <vsip/support.hpp>
#include <vsip/signal.hpp>
#include <vsip/math.hpp>

using namespace vsip;



/***************************************************************************
  Main Program
***************************************************************************/

int
main(int argc, char** argv)
{
  // Initialize the library.
  vsipl vpp(argc, argv);

  typedef complex<float> value_type;

  // Parameters.
  length_type npulse = 64; // number of pulses
  length_type nrange = 256; // number of range cells

  // Views.
  Vector<value_type> replica(nrange);
  Matrix<value_type> data(npulse, nrange);
  Matrix<value_type> tmp(npulse, nrange);

  // A forward Fft for computing the frequency-domain version of
  // the replica.
  typedef Fft<const_Vector, value_type, value_type, fft_fwd, \
by_reference>
  for_fft_type;
  for_fft_type  for_fft (Domain<1>(nrange), 1.0);

  // A forward Fftm for converting the time-domain data matrix to the
  // frequency domain.
  typedef Fftm<value_type, value_type, row, fft_fwd, by_reference>
    for_fftm_type;
  for_fftm_type for_fftm(Domain<2>(npulse, nrange), 1.0);
```

```
  // An inverse Fftm for converting the frequency-domain data back to
  // the time-domain.
  typedef Fftm<value_type, value_type, row, fft_inv, by_reference>
    inv_fftm_type;
  inv_fftm_type inv_fftm(Domain<2>(npulse, nrange), 1.0/(nrange));

  // Initialize data to zero.
  data    = value_type();
  replica = value_type();

  // Before fast convolution, convert the replica to the the
  // frequency domain
  for_fft(replica);


  // Perform fast convolution.

  // Convert to the frequency domain.
  for_fftm(data, tmp);

  // Perform element-wise multiply for each pulse.
  tmp = vmmul<0>(replica, tmp);

  // Convert back to the time domain.
  inv_fftm(tmp, data);
}
 \
```

The following figure shows the performance in MFLOP/s of fast convolution on a 3.06 GHz Pentium Xeon processor as the number of range cells varies from 16 to 65536.



7.2. Serial Optimization: Temporal Locality

In this section, you will learn how to improve the performance of fast convolution by improving *temporal locality*, i.e., by making accesses to the same memory locations occur near the same time.

The code in Section 7.1, "Fast Convolution" performs a FFT on each row of the matrix. Then, after all the rows have been processed, it multiplies each row of the matrix by the `replica`. Suppose that there are a large number of rows, so that `data` is too large to fit in cache. In that case, while the results of the first FFT will be in cache immediately after the FFT is complete, that data will likey have been purged from the cache by the time the vector-matrix multiply needs the data.

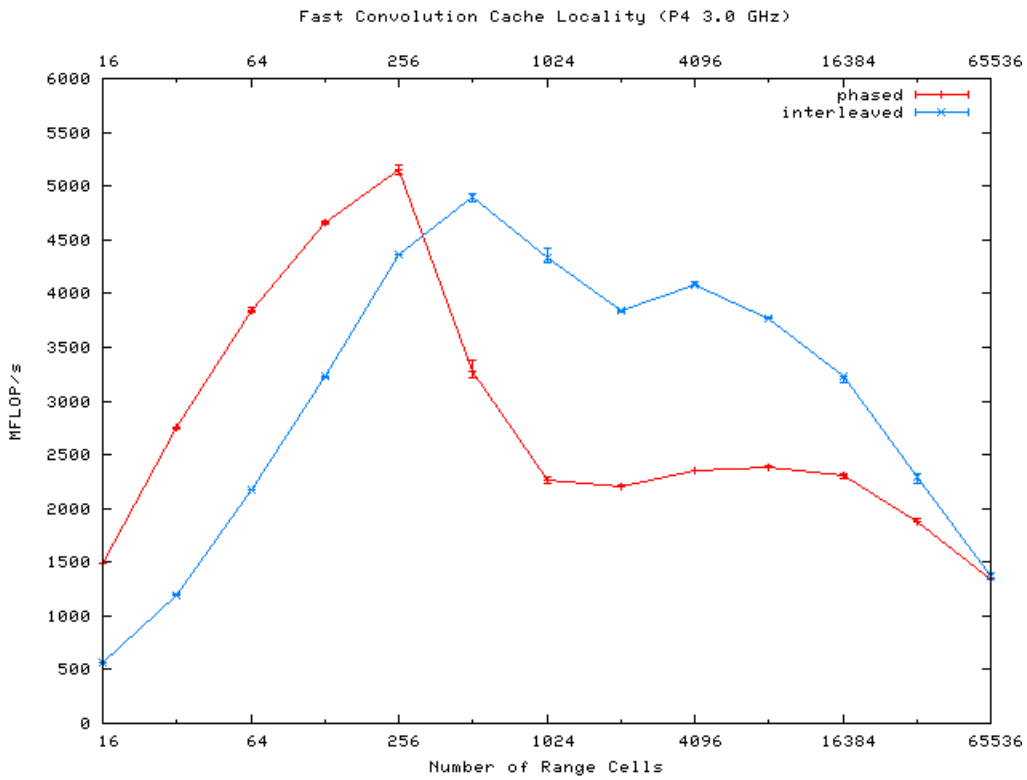
Explicitly iterating over the rows of the matrix (performing a forward FFT, elementwise multiplication, and an inverse FFT on each row before going on to the next one) will improve temporal locality. You can use this approach by using an explicit loop, rather than the implicit parallelism of `Fftm` and `vmmul`, to take better advantage of the cache.

You must make a few changes to the application in order to implement this approach. Because the application will be operating on only a single row at a time, `Fftm` must be replaced with the simpler `Fft`. Similarly, `vmmul` must be replaced with `*`, which performs element-wise multiplication of its operands. Finally, `tmp` can now be a vector, rather than a matrix. (As a consequence, in addition to being faster, this new version of the application will require less memory.) Here is the revised program:

```
// Create the data cube.
Matrix<value_type> data(npulse, nrange);
Vector<value_type> tmp(nrange);           // tmp is now a vector

// Create the pulse replica
Vector<value_type> replica(nrange);
```

```
  // Define the FFT typedefs.
  typedef Fft<const_Vector, value_type, value_type, fft_fwd, \
by_reference>
  for_fft_type;
  typedef Fft<const_Vector, value_type, value_type, fft_inv, \
by_reference>
  inv_fft_type;

  // Create the FFT objects.
  for_fft_type  for_fft(Domain<1>(nrange), 1.0);
  inv_fft_type  inv_fft(Domain<1>(nrange), 1.0/(nrange));

  // Initialize data to zero
  data    = value_type();
  replica = value_type();

  // Before fast convolution, convert the replica into the
  // frequency domain
  for_fft(replica);

  // Perform fast convolution:
  for (index_type r=0; r < nrange; ++r)
  {
    for_fft(data.row(r), tmp);
    tmp *= replica;
    inv_fft(tmp, data.row(r));
  }
```

The following graph shows that the new "interleaved" formulation is faster than the original "phased" approach for large data sets. For smaller data sets (where all of the data fits in the cache anyhow), the original method is faster because performing all of the FFTs at once is faster than performing them one by one.



7.3. Performing I/O with User-Specified Storage

The previous sections have ignored the acquisition of actual sensor data by setting the input data to zero. This section shows how to initialize `data` before performing the fast convolution.

To perform I/O with external routines (such as the POSIX `read` and `write` functions) it is necessary to obtain a pointer to the raw data used by Sourcery VSIPL++. Sourcery VSIPL++ provides two ways to do this: you may use either *user-defined storage* or *external data access*. In this section you will use user-defined storage to perform I/O. Later, in Section 7.4, “Performing I/O with External Data Access” you will see how to use external data access for I/O.

VSIPL++ allows you to create a block with user-specified storage by giving VSIPL++ a pointer to previously allocated data when the block is created. This block is just like a normal block, except that it now has two states: "admitted" and "released". When the block is admitted, the data is owned by VSIPL++ and the block can be used with any VSIPL++ functions. When the block is released, the data is owned by you allowing you to perform operations directly on the data. The states allow VSIPL++ to potentially reorganize data for higher performance while it is admitted. (Attempting to use the pointer while the block is admitted, or use the block while it is released will result in unspecified behavior!)

The first step is to allocate the data manually.

```
  std::vector<value_type> buffer(npulse*nrange);
```

Next, you create a VSIPL++ `Dense` block, providing it with the pointer.

```
  Dense<2, value_type> block(Domain<2>(nrange, npulse), \
&buffer.front());
```

Since the pointer to data does not encode the data dimensions, it is necessary to create the block with explicit dimensions.

Finally, you create a VSIPL++ view that uses this block.

```
  Matrix<value_type> data(block);
```

The view determines its size from the block, so there is no need to specify the dimensions again.

Now you're ready to perform I/O. When a user-specifed storage block is first created, it is released.

```
  ... setup IO ...
  read(..., &buffer.front(), sizeof(value_type)*nrange*npulse);
  ... check for errors (of course!) ...
```

Finally, you need to admit the block so that it and the view can be used by VSIPL++.

```
  data.block().admit(true);
```

The `true` argument indicates that the data values sould be preserved by the admit. In cases where the values do not need to preserved (such as admitting a block after output I/O has been performed and before the block will be overwritten by new values in VSIPL++) you can use `false` instead.

After admitting the block, you can use `data` as before to perform fast convolution. Here is the complete program, including I/O to output the result after the computation.

```
/***********************************************************************
  Included Files
***********************************************************************/

#include <vsip/initfin.hpp>
#include <vsip/support.hpp>
#include <vsip/signal.hpp>
#include <vsip/math.hpp>
#include <vector>

using namespace vsip;



/***********************************************************************
  Main Program
***********************************************************************/

int
main(int argc, char** argv)
{
  // Initialize the library.
  vsipl vpp(argc, argv);

  typedef complex<float> value_type;

  // Parameters.
  length_type npulse = 64; // number of pulses
  length_type nrange = 256; // number of range cells
```

```
  // Allocate data.
  std::vector<value_type> data(npulse*nrange);

  // Blocks.
  Dense<2, value_type> block(Domain<2>(npulse, nrange), \
&data.front());

  // Views.
  Vector<value_type> replica(nrange);
  Matrix<value_type> data(block);
  Matrix<value_type> tmp(npulse, nrange);

  // A forward Fft for computing the frequency-domain version of
  // the replica.
  typedef Fft<const_Vector, value_type, value_type, fft_fwd, \
by_reference>
  for_fft_type;
  for_fft_type  for_fft (Domain<1>(nrange), 1.0);

  // A forward Fftm for converting the time-domain data matrix to the
  // frequency domain.
  typedef Fftm<value_type, value_type, row, fft_fwd, by_reference>
    for_fftm_type;
  for_fftm_type for_fftm(Domain<2>(npulse, nrange), 1.0);

  // An inverse Fftm for converting the frequency-domain data back to
  // the time-domain.
  typedef Fftm<value_type, value_type, row, fft_inv, by_reference>
    inv_fftm_type;
  inv_fftm_type inv_fftm(Domain<2>(npulse, nrange), 1.0/(nrange));

  // Initialize data to zero.
  data    = value_type();
  replica = value_type();

  // Before fast convolution, convert the replica to the the
  // frequency domain
  for_fft(replica);


  // Read input.
  view.block().release(false);
  size_t size = read(0, &data.front(), \
sizeof(value_type)*nrange*npulse);
  assert(size == sizeof(value_type)*nrange*npulse));
  view.block().admit(true);

  // Perform fast convolution.

  // Convert to the frequency domain.
  for_fftm(data, tmp);

  // Perform element-wise multiply for each pulse.
```

```
  tmp = vmmul<0>(replica, tmp);

  // Convert back to the time domain.
  inv_fftm(tmp, data);

  // Write output.
  view.block().release(true);
  size_t size = write(0, &data.front(), \
sizeof(value_type)*nrange*npulse);
  assert(size == sizeof(value_type)*nrange*npulse));
  view.block().admit(false);
}
 \
```

The program also includes extra `release()` and `admit()` calls before and after the input and output I/O sections. For this example, they are not strictly necessary. However they are good practice because they make it clear in the program where the block is admitted and released. They also make it easier to modify the program to process data repeatedly in a loop, and to use separate buffers for input and output data. Because the extra calls have a `false` update argument, they incur no overhead.

7.4. Performing I/O with External Data Access

In this section, you will use *External Data Access* to get a pointer to a block's data. You can use this method with any block, even if the block does not use user-specified storage. The external data access method is useful in contexts where you cannot control how the block is allocate. For example, in this section, you will create a utility routine for I/O that works with any matrix or vector, even if it was not created with user-defined storage.

To access a block's data with external data access, you create an `Ext_data` object.

```
  Ext_data<block_type, layout_type> ext(block, SYNC_INOUT);
```

`Ext_data` is a class template that takes template parameters to indicate the block type `block_type` and the requested layout `layout_type`. The constructor takes two parameters: the block being accessed, and the type of synchronization necessary.

The `layout_type` parameter is a specialized `Layout` class template that determines the layout of data that `Ext_data` provides. If no type is given, the natural layout of the block is used. However, in some cases you may wish to specify row-major or column-major layout.

The `Layout` class template takes 4 parameters to indicate dimensionality, dimension-ordering, packing format, and complex storage format (if complex). In the example below you will use the layout_type to request the data access to be dense, row-major, with interleaved real and imaginary values. This layout corresponds to a common storage format used for binary files storing complex data.

The synchronization type is analgous to the update flags for `admit()` and `release()`. `SYNC_IN` indicates that the block and pointer should be synchronized when the `Ext_data` object is created (like `admit(true)`) `SYNC_OUT` indicates that the block and pointer should be synchronized when the `Ext_data` object is destroyed (like `release(true)`) `SYNC_INOUT` indicates that the block and pointer should be syncrhonized at both points.

Once the object has been created, the pointer can be accessed with the `data` method.

```
  value_type* ptr = ext.data();
```

The pointer provided is valid only during the life of the `Ext_data` object. Moreover, the block referred to by the `Ext_data` object must not be used during this period.

Using these capabilities together, you can create a routine to perform I/O into a block. This routine will take two arguments: a filename to read, and a view in which to store the data. The amount of data read from the file will be determined by the view's size.

```
template <typename ViewT>
void
read_file(ViewT view, char const* filename)
{
  using vsip::impl::Ext_data;
  using vsip::impl::Layout;
  using vsip::impl::Stride_unit_dense;
  using vsip::impl::Cmplx_inter_fmt;
  using vsip::impl::Row_major;
  using vsip::impl::SYNC_OUT;

  dimension_type const dim = ViewT::dim;
  typedef typename ViewT::block_type block_type;
  typedef typename ViewT::value_type value_type;

  typedef Layout<dim, typename Row_major<dim>::type,
                 Stride_unit_dense, Cmplx_inter_fmt>
  layout_type;

  Ext_data<block_type, layout_type>
  ext(view.block(), SYNC_OUT);

  std::ifstream ifs(filename);

  ifs.read(reinterpret_cast<char*>(ext.data()),
    view.size() * sizeof(value_type));
}
```

Chapter 8
Parallel Fast Convolution

Abstract

This chapter describes how to create and run parallel VSIPL++ programs with Sourcery VSIPL++. You can modify the programs to develop your own parallel applications.

This chapter explains how to use Sourcery VSIPL++ to perform parallel computations. You will see how to transform the fast convolution program from the previous chapter to run in parallel. First you will convert the `Fftm` based version. Then you will convert the improved cache locality version. Finally, you will learn how to handle input and output when working in parallel.

8.1. Parallel Fast Convolution

The first fast convolution program in the previous chapter makes use of two implicitly parallel operators: `Fftm` and `vmmul`. These operators are implicitly parallel in the sense that they process each row of the matrix independently. If you had enough processors, you could put each row on a separate processor and then perform the entire computation in parallel.

In the VSIPL++ API, you have explicit control of the number of processors used for a computation. Since the default is to use just a single processor, the program in Section 7.1, “Fast Convolution” will not run in parallel, even on a multi-processor system. This section will show you how to use *maps* to take advantage of multiple processors. Using a map tells Sourcery VSIPL++ to distribute a single block of data across multiple processors. Then, Sourcery VSIPL++ will automatically move data between processors as necessary.

The VSIPL++ API uses the Single-Program Multiple-Data (SPMD) model for parallelism. In this model, every processor runs the same program, but operates on different sets of data. For example, in the fast convolution example, multiple processors perform FFTs at the same time, but each processor handles different rows in the matrix.

Every map has both compile-time and run-time properties. At compile-time, you specify the *distribution* that will be applied to each dimension. In this example, you will use a *block distribution* to distribute the rows of the matrix. A block distribution divides a view into continguous chunks. For example, suppose that you have a 4-processor system. Since there are 64 rows in the matrix `data`, there will be 16 rows on each processor. The block distribution will place the first 16 rows (rows 0 through 15) on processor 0, the next 16 rows (rows 16 through 31) on processor 1, and so forth. You do not want to distribute the columns of the matrix at all, so you will use a *whole distribution* for the columns.

Although the distributions are selected at compile-time, the number of processors to use in each dimension is not specified until run-time. By specifying the number of processors at run-time, you can adapt your program to the configuration of the machine on which your application is running. The VSIPL++ API provides a `num_processors` function to tell you the total number of processors available. Of course, since each row should be kept on a single processor, the number of processors used in the column dimension is just one. So, here is the code required to create the map:

```
typedef Map<Block_dist, Whole_dist>              map_type;
map_type map = map_type(/*rows=*/num_processors(),
                        /*columns=*/1);
```

Next, you have to tell Sourcery VSIPL++ to use this map for the relevant views. Every view has an underlying *block*. The block indicates how the view's data is stored. Until this point, you have been using the default `Dense` block, which stores data in a continguous array on a single processor. Now, you want to use a continguous array on *multiple* processors, so you must explicitly distribute the block. Then, when declaring views, you must explicitly indicate that the view should use the distributed block:

```
typedef Dense<2, value_type, row2_type, map_type> block_type;
typedef Matrix<value_type, block_type>            view_type;
```

```
  view_type data(npulse, nrange, map);
  view_type tmp(npulse, nrange, map);
```

Performing the vector-matrix multiply requires a complete copy of `replica` on each processor. An ordinary map divides data among processors, but, here, the goal is to copy the same data to multiple processors. Sourcery VSIPL++ provides a special `Replicated_map` class to use in this situation. So, you should declare `replica` as follows:

```
  Replicated_map<1> replica_map;
  typedef Dense<1, value_type, row1_type, Replicated_map<1> >
                                                        \
replica_block_type;
  typedef Vector<value_type, replica_block_type>    \
replica_view_type;
  replica_view_type replica(nrange, replica_map);
```

Because the application already uses implicitly parallel operators, no further changes are required. The entire algorithm (i.e., the part of the code that performs FFTs and vector-matrix multiplication) remains unchanged.

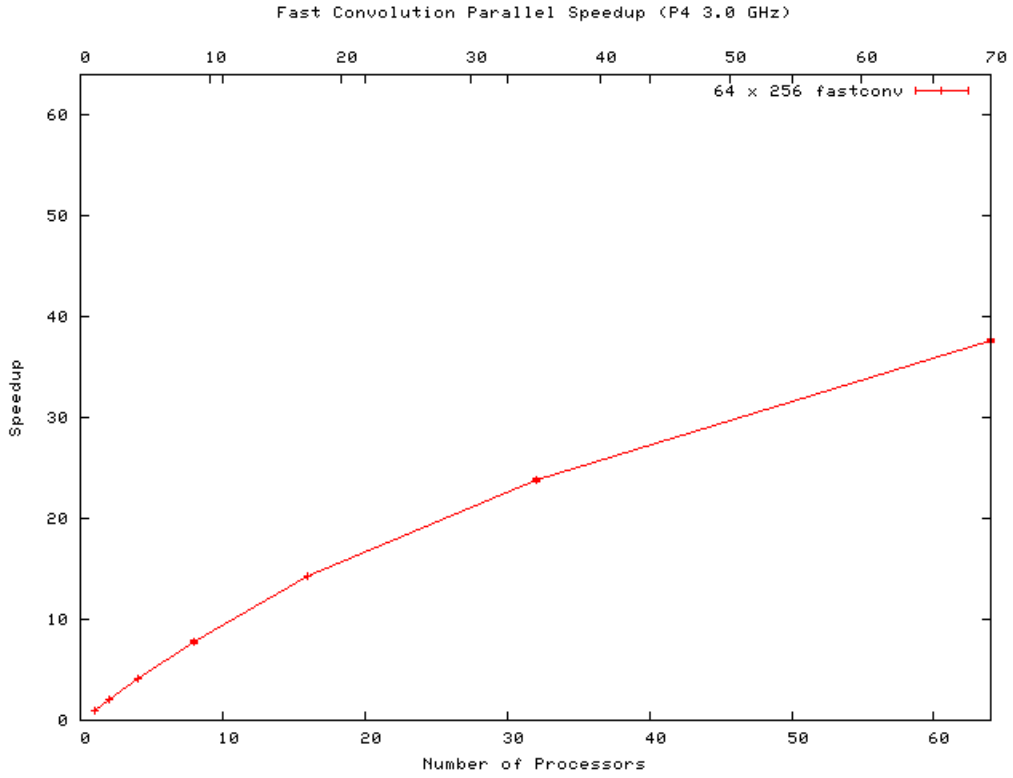
The complete parallel program is:

```
/***********************************************************************
  Included Files
***********************************************************************/

#include <vsip/initfin.hpp>
#include <vsip/support.hpp>
#include <vsip/signal.hpp>
#include <vsip/math.hpp>
#include <vsip/map.hpp>

using namespace vsip;



/***********************************************************************
  Main Program
***********************************************************************/

int
main(int argc, char** argv)
{
  // Initialize the library.
  vsipl vpp(argc, argv);

  typedef complex<float> value_type;

  typedef Map<Block_dist, Whole_dist>               map_type;
  typedef Dense<2, value_type, row2_type, map_type> block_type;
  typedef Matrix<value_type, block_type>            view_type;

  typedef Dense<1, value_type, row1_type, Replicated_map<1> >
                                                        \
```

```
replica_block_type;
  typedef Vector<value_type, replica_block_type>    \
replica_view_type;

  // Parameters.
  length_type npulse = 64; // number of pulses
  length_type nrange = 256; // number of range cells

  // Maps.
  map_type          map = map_type(num_processors(), 1);
  Replicated_map<1> replica_map;

  // Views.
  replica_view_type replica(nrange, replica_map);
  view_type         data(npulse, nrange, map);
  view_type         tmp (npulse, nrange, map);

  // A forward Fft for computing the frequency-domain version of
  // the replica.
  typedef Fft<const_Vector, value_type, value_type, fft_fwd, \
by_reference>
  for_fft_type;
  for_fft_type  for_fft (Domain<1>(nrange), 1.0);

  // A forward Fftm for converting the time-domain data matrix to the
  // frequency domain.
  typedef Fftm<value_type, value_type, row, fft_fwd, by_reference>
    for_fftm_type;
  for_fftm_type for_fftm(Domain<2>(npulse, nrange), 1.0);

  // An inverse Fftm for converting the frequency-domain data back to
  // the time-domain.
  typedef Fftm<value_type, value_type, row, fft_inv, by_reference>
    inv_fftm_type;
  inv_fftm_type inv_fftm(Domain<2>(npulse, nrange), 1.0/(nrange));

  // Initialize data to zero.
  data    = value_type();
  replica = value_type();

  // Before fast convolution, convert the replica to the the
  // frequency domain
  for_fft(replica);


  // Perform fast convolution:

  // Convert to the frequency domain.
  for_fftm(data, tmp);

  // Perform element-wise multiply for each pulse.
  tmp = vmmul<0>(replica, tmp);

  // Convert back to the time domain.
```

```
  inv_fftm(tmp, data);
}
 \
```

The following graph shows the parallel speedup of the fast convolution program from 1 to 32 processors using a 3.0 GHz Pentium cluster system. As you can see, increasing the number of processors also increases the performance of the program.



8.2. Improving Parallel Temporal Locality

In the previous chapter, you improved the performance of the fast convolution program by exploiting temporary cache locality to process data while it was "hot" in the cache. In this section, you will convert that program to run efficiently in parallel.

If we apply maps (as in Section 8.1, “Parallel Fast Convolution”), but do not adjust the algorithm in use, the code in Section 7.2, “Serial Optimization: Temporal Locality” will not run faster when deployed on multiple processors. In particular, every processor will want to update `tmp` for every row. Therefore, all processors will perform the forward FFT and vector-multiply for each row of the matrix.

VSIPL++ provides *local subviews* to solve this problem. For a given processor and view, the local subview is that portion of the view located on the processor. You can obtain the local subview of any view by invoking its `local` member function:

```
  view_type::local_type     l_data    = data.local();)
```

Every view class defines a type (`local_type`) which is the type of a local subview. The `local_type` is the same kind of view as the view containing it, so, in this case, `l_data` is a matrix. There is virtually no overhead in creating a local subview like `l_data`. In particular, no data is copied;

instead, `l_data` just refers to the local portion of `data`. We can now use the same cache-friendly algorithm from Section 7.2, “Serial Optimization: Temporal Locality”on the local subview:

```
rep_view_type::local_type l_replica = replica.local();

for (index_type l_r=0; l_r < l_data.size(0); ++l_r)
{
  for_fft(l_data.row(l_r), tmp);
  tmp *= l_replica;
  inv_fft(tmp, l_data.row(l_r));
}
```

Because each processor now iterates over only the rows of the matrix that are local, there is no longer any duplicated effort. Applying maps, as in Section 8.1, “Parallel Fast Convolution” above, results in the following complete program:

```
/***********************************************************************
  Included Files
***********************************************************************/

#include <vsip/initfin.hpp>
#include <vsip/support.hpp>
#include <vsip/signal.hpp>
#include <vsip/math.hpp>
#include <vsip/map.hpp>

using namespace vsip;



/***********************************************************************
  Main Program
***********************************************************************/

int
main(int argc, char** argv)
{
  // Initialize the library.
  vsipl vpp(argc, argv);

  typedef complex<float> value_type;

  typedef Map<Block_dist, Whole_dist>               map_type;
  typedef Dense<2, value_type, row2_type, map_type> block_type;
  typedef Matrix<value_type, block_type>            view_type;

  typedef Dense<1, value_type, row1_type, Replicated_map<1> >
                                                    \
replica_block_type;
  typedef Vector<value_type, replica_block_type>    \
replica_view_type;

  // Parameters.
```

```
  length_type npulse = 64; // number of pulses
  length_type nrange = 256; // number of range cells

  // Maps.
  map_type          map = map_type(num_processors(), 1);
  Replicated_map<1> replica_map;

  // Views.
  replica_view_type  replica(nrange, replica_map);
  view_type          data(npulse, nrange, map);
  Vector<value_type> tmp(nrange);

  // A forward Fft for converting the time-domain data to the
  // frequency domain.
  typedef Fft<const_Vector, value_type, value_type, fft_fwd, \
by_reference>
  for_fft_type;
  for_fft_type  for_fft(Domain<1>(nrange), 1.0);

  // An inverse Fft for converting the frequency-domain data back to
  // the time-domain.
  typedef Fft<const_Vector, value_type, value_type, fft_inv, \
by_reference>
    inv_fft_type;
  inv_fft_type  inv_fft(Domain<1>(nrange), 1.0/nrange);

  // Initialize data to zero.
  data    = value_type();
  replica = value_type();

  // Before fast convolution, convert the replica into the
  // frequency domain
  for_fft(replica.local());

  view_type::local_type         l_data    = data.local();
  replica_view_type::local_type l_replica = replica.local();

  for (index_type l_r=0; l_r < l_data.size(0); ++l_r)
  {
    for_fft(l_data.row(l_r), tmp);
    tmp *= l_replica;
    inv_fft(tmp, l_data.row(l_r));
  }
}
 \
```

8.2.1. Implicit Parallelism: Parallel Foreach

You may feel that the original formulation using implicitly parallel operators was simpler and more intuitive than the more-efficient variant using explicit loops. Sourcery VSIPL++ provides an extension to the VSIPL++ API that allows you to retain the elegance of that formulation while still obtaining good temporal locality.

In particular, Sourcery VSIPL++ provides a "parallel foreach" operator. This operator applies an arbitrary user-defined function (or an object that behaves like a function) to each of the rows or columns of a matrix. In this section, you will see how to use this approach.

First, declare a `Fast_convolution` template class. The template parameter `T` is used to indicate the value type of the fast convolution computation (such as `complex<float>`):

```
template <typename T>
 class Fast_convolution
 {
```

This class will perform the forward FFT and inverse FFTs on each row, so you must declare the FFTs:

```
  typedef Fft<const_Vector, T, T, fft_fwd, by_reference> \
for_fft_type;
   typedef Fft<const_Vector, T, T, fft_inv, by_reference> \
inv_fft_type;

   Vector<T>    replica_;
   Vector<T>    tmp_;
   for_fft_type for_fft_;
   inv_fft_type inv_fft_;
```

Next, define a constructor for `Fast_convolution`. The constructor stores a copy of the replica, and also uses the replica to determine the number of elements required for the FFTs and temporary vector.

```
template <typename Block>
   Fast_convolution(
     Vector<T, Block> replica)
     : replica_(replica.size()),
       tmp_    (replica.size()),
       for_fft_(Domain<1>(replica.size()), 1.0),
       inv_fft_(Domain<1>(replica.size()), 1.0/replica.size())
   {
     replica_ = replica;
   }
```

The most important part of the `Fast_convolution` class is the `operator()` function. This function performs a fast convolution for a single row of the matrix:

```
  template <typename       Block1,
      typename       Block2,
      dimension_type Dim>
   void operator()(
     Vector<T, Block1> in,
     Vector<T, Block2> out,
     Index<Dim>        /*idx*/)
   {
     for_fft_(in, tmp_);
     tmp_ *= replica_;
     inv_fft_(tmp_, out);
   }
```

The `foreach_vector` template will apply the new class you have just defined to the rows of the matrix:

```
  Fast_convolution<value_type> fconv(replica.local());
   foreach_vector<tuple<0, 1> >(fconv, data);
```

The resulting program contains no explicit loops, but still has good temporal locality. Here is the complete program, using the parallel foreach operator:

```
/***********************************************************************
  Included Files
***********************************************************************/

#include <vsip/initfin.hpp>
#include <vsip/support.hpp>
#include <vsip/signal.hpp>
#include <vsip/math.hpp>
#include <vsip/map.hpp>
#include <vsip/parallel.hpp>

using namespace vsip;



/***********************************************************************
  Main Program
***********************************************************************/

template <typename T>
class Fast_convolution
{
  typedef Fft<const_Vector, T, T, fft_fwd, by_reference> \
for_fft_type;
  typedef Fft<const_Vector, T, T, fft_inv, by_reference> \
inv_fft_type;

public:
  template <typename Block>
  Fast_convolution(
    Vector<T, Block> replica)
    : replica_(replica.size()),
      tmp_    (replica.size()),
      for_fft_(Domain<1>(replica.size()), 1.0),
      inv_fft_(Domain<1>(replica.size()), 1.0/replica.size())
  {
    replica_ = replica;
  }

  template <typename       Block1,
     typename       Block2,
     dimension_type Dim>
  void operator()(
    Vector<T, Block1> in,
    Vector<T, Block2> out,
```

```
    Index<Dim>        /*idx*/)
  {
    for_fft_(in, tmp_);
    tmp_ *= replica_;
    inv_fft_(tmp_, out);
  }

  // Member data.
private:
  Vector<T>    replica_;
  Vector<T>    tmp_;
  for_fft_type for_fft_;
  inv_fft_type inv_fft_;
};



int
main(int argc, char** argv)
{
  // Initialize the library.
  vsipl vpp(argc, argv);

  typedef complex<float> value_type;

  typedef Map<Block_dist, Whole_dist>               map_type;
  typedef Dense<2, value_type, row2_type, map_type> block_type;
  typedef Matrix<value_type, block_type>            view_type;

  typedef Dense<1, value_type, row1_type, Replicated_map<1> >
                                                    \
replica_block_type;
  typedef Vector<value_type, replica_block_type>    \
replica_view_type;

  // Parameters.
  length_type npulse = 64; // number of pulses
  length_type nrange = 256; // number of range cells

  // Maps.
  map_type          map = map_type(num_processors(), 1);
  Replicated_map<1> replica_map;

  // Views.
  replica_view_type replica(nrange, replica_map);
  view_type         data(npulse, nrange, map);
  view_type         tmp (npulse, nrange, map);

  // A forward Fft for computing the frequency-domain version of
  // the replica.
  typedef Fft<const_Vector, value_type, value_type, fft_fwd, \
by_reference>
  for_fft_type;
  for_fft_type  for_fft (Domain<1>(nrange), 1.0);
```

```
  Fast_convolution<value_type> fconv(replica.local());

  // Initialize data to zero.
  data    = value_type();
  replica = value_type();

  // Before fast convolution, convert the replica into the
  // frequency domain
  for_fft(replica.local());

  // Perform fast convolution.
  foreach_vector<tuple<0, 1> >(fconv, data);
}
 \
```

8.3. Performing I/O

The previous sections have ignored the acquisition of actual sensor data by setting the input data to zero. This section shows how to extend the I/O techniques introduced in the previous chapter to initialize `data` before performing the fast convolution.

Let's assume that all of the input data arrives at a single processor via DMA. This data must be distributed to the other processors to perform the fast convolution. So, the input processor is special, and is not involved in the computation proper.

To describe this situation in Sourcery VSIPL++, you need two maps: one for the input processor (`map_in`), and one for the compute processors (`map`). These two maps will be used to define views that can be used to move the data from the input processor to the compute processors. Let's assume that the input processor is processor zero. Then, create `map_in` as follows, mapping all data to the single input processor:

```
  typedef Map<> map_type;
  Vector<processor_type> pvec_in(1);  pvec_in(0)  = 0;
  map_type map_in (pvec_in,  1,  1);
```

In contrast, `map` distributes rows across all of the compute processors:

```
  // Distribute computation across all processors:
  map_type map    (num_processors(), 1);
```

Because the data will be arriving via DMA, you must explicitly manage the memory used by Sourcery VSIPL++. Because VSIPL++ uses the SPMD model, each processor must allocate the memory for its local portion the input block, even though all processors except the actual input processor will allocate zero bytes. The code required to set up the views is:

```
  block_type data_in_block(npulse, nrange, 0, map_in);
  view_type data_in(data_in_block);
  view_type data   (npulse, nrange, map);
  size_t size = subblock_domain(data_in).size();
  std::vector<value_type> buffer(size);
  data_in.block()->rebind(&buffer.front());
```

Now, you can perform the actual I/O. The I/O (including any calls to low-level DMA routines) should only be performed on the input processor. The `subblock` function is used to ensure that I/O is only performed on the appropriate processors:

```
  if (subblock(data_in) != no_subblock)
   {
     data_in.block().release(false);
     // ... perform IO into data_in ...
     data_in.block().admit(true);
   }
```

Once the I/O completes, you can move the data from `data_in` to `data` for processing. In the VSIPL++ API, ordinary assignment (using the = operator) will perform all communication necessary to distribute the data. So, performing the "scatter" operation is just:

```
  data = data_in;
```

The complete program is:

```
/***********************************************************************
  Included Files
***********************************************************************/

#include <vsip/initfin.hpp>
#include <vsip/support.hpp>
#include <vsip/signal.hpp>
#include <vsip/math.hpp>
#include <vsip/map.hpp>
#include <vsip/parallel.hpp>

using namespace vsip;



/***********************************************************************
  Main Program
***********************************************************************/

template <typename       ViewT,
   dimension_type Dim>
ViewT
create_view_wstorage(
  Domain<Dim> const&                          dom,
  typename ViewT::block_type::map_type const& map)
{
  typedef typename ViewT::block_type block_type;
  typedef typename ViewT::value_type value_type;

  block_type* block = new block_type(dom, (value_type*)0, map);
  ViewT view(*block);
  block->decrement_count();

  if (subblock(view) != no_subblock)
  {
```

```
    size_t size = subblock_domain(view).size();
    value_type* buffer = vsip::impl::alloc_align<value_type>(128, \
size);
    block->rebind(buffer);
  }

  block->admit(false);

  return view;
}



template <typename ViewT>
void
cleanup_view_wstorage(ViewT view)
{
  typedef typename ViewT::value_type value_type;
  value_type* ptr;

  view.block().release(false, ptr);
  view.block().rebind((value_type*)0);

  if (ptr) vsip::impl::free_align((void*)ptr);
}



template <typename ViewT>
ViewT
create_view_wstorage(
  length_type                                rows,
  length_type                                cols,
  typename ViewT::block_type::map_type const& map)
{
  return create_view_wstorage<ViewT>(Domain<2>(rows, cols), map);
}



template <typename ViewT>
ViewT
create_view_wstorage(
  length_type                                size,
  typename ViewT::block_type::map_type const& map)
{
  return create_view_wstorage<ViewT>(Domain<1>(size), map);
}




int
main(int argc, char** argv)
```

```
{
  // Initialize the library.
  vsipl vpp(argc, argv);

  typedef complex<float> value_type;

  typedef Map<Block_dist, Block_dist>               map_type;
  typedef Dense<2, value_type, row2_type, map_type> block_type;
  typedef Matrix<value_type, block_type>            view_type;

  typedef Dense<1, value_type, row1_type, Replicated_map<1> >
                                                    \
replica_block_type;
  typedef Vector<value_type, replica_block_type>    \
replica_view_type;

  typedef Dense<1, value_type, row1_type, Map<> >   \
replica_io_block_type;
  typedef Vector<value_type, replica_io_block_type> \
replica_io_view_type;

  // Parameters.
  length_type npulse = 64; // number of pulses
  length_type nrange = 256; // number of range cells

  length_type np = num_processors();

  // Processor sets.
  Vector<processor_type> pvec_in(1);  pvec_in(0)  = 0;
  Vector<processor_type> pvec_out(1); pvec_out(0) = np-1;

  // Maps.
  map_type          map_in (pvec_in,  1, 1);
  map_type          map_out(pvec_out, 1, 1);
  map_type          map_row(np, 1);
  Replicated_map<1> replica_map;

  // Views.
  view_type data(npulse, nrange, map_row);
  view_type tmp (npulse, nrange, map_row);
  view_type data_in (create_view_wstorage<view_type>(npulse, nrange, \
map_in));
  view_type data_out(create_view_wstorage<view_type>(npulse, nrange, \
map_out));
  replica_view_type    replica(nrange);
  replica_io_view_type replica_in(
    create_view_wstorage<replica_io_view_type>(nrange, map_in));

  // A forward Fft for computing the frequency-domain version of
  // the replica.
  typedef Fft<const_Vector, value_type, value_type, fft_fwd, \
by_reference>
  for_fft_type;
  for_fft_type  for_fft (Domain<1>(nrange), 1.0);
```

```
// A forward Fftm for converting the time-domain data matrix to the
// frequency domain.
typedef Fftm<value_type, value_type, row, fft_fwd, by_reference>
  for_fftm_type;
for_fftm_type for_fftm(Domain<2>(npulse, nrange), 1.0);

// An inverse Fftm for converting the frequency-domain data back to
// the time-domain.
typedef Fftm<value_type, value_type, row, fft_inv, by_reference>
  inv_fftm_type;
inv_fftm_type inv_fftm(Domain<2>(npulse, nrange), 1.0/(nrange));

// Perform input IO
if (subblock(data_in) != no_subblock)
{
  data_in.block().release(false);
  // ... perform IO ...
  data_in.block().admit(true);

  replica_in.block().release(false);
  // ... perform IO ...
  replica_in.block().admit(true);

  data_in    = value_type();
  replica_in = value_type();

  // Before fast convolution, convert the replica into the
  // frequency domain
  for_fft(replica_in.local());
}

// Scatter data
data    = data_in;
replica = replica_in;

// Perform fast convolution.
for_fftm(data, tmp);  // Convert to the frequency domain.
tmp = vmmul<0>(replica, tmp); // Perform element-wise multiply.
inv_fftm(tmp, data);  // Convert back to the time domain.

// Gather data
data_out = data;

// Perform output IO
if (subblock(data_out) != no_subblock)
{
  data_out.block().release(true);
  // ... perform IO ...
  data_out.block().admit(false);
}

// Cleanup
cleanup_view_wstorage(data_in);
```

```
  cleanup_view_wstorage(data_out);
  cleanup_view_wstorage(replica_in);
}
```

The technique demonstrated in this section extends easily to the situation in which the sensor data is arriving at multiple processors simultaneously. To distribute the I/O across multiple processors, just add them to `map_in`'s processor set `pvec_in`:

```
  Vector<processor_type> pvec_in(num_io_proc);
  pvec_in(0)              = 0;
  ...
  pvec_in(num_io_proc-1)  = ...;
```

Appendix A
Benchmark Options

This appendix lists all options used in the benchmark programs, along with a brief description of each one.

Option

`-all`	Reports all three statistics: MPT/s, MOP/s, MB/s
`-cal` *M*	Sets to 2^*M* the problem size for calibration. The default is four, that is a problem size of 16. Calibration attempts to determine the number of times to run a benchmark to ensure that the total measured time is within a threshold of the goal time, set by `-ms`.
`-nocal`	Do not calibrate. The default is to calibrate with a problem size of 2^4 or 16. The `-cal` option can change this size.
`-center` *M*	Sets the range type to *centered* with the center value *M*.
`-data`	Must be combined with `-samples` *M* where *M* >= 3. Displays many columns of data suitable for a comma-separated-values spreadsheet. The columns are: problem size median time minimum time maximum time memory per point operations per point input memory per point output memory per point number of loops time per loop
`-diag`	Run the benchmark in diagnostic mode. For many benchmarks, this displays information on what evaluators or backend libraries are used in the benchmark.
`-fix_loop`	Runs a rough set of executions with no calibration and one loop per problem size.
`-geom`	Uses a geometric progression of problem sizes. For start <= I <= stop, the problem size is 2^I. See also `-linear`.
`-iob`	Reports millions of input/output operations per second (MB/s) (by summing read and write iob_per_point).
`-lat`	Reports seconds per point.
`-lib_config`	Displays a summary of the configuration of the VSIPL++ library.
`-linear` *M*	Uses a linear progression of problem sizes. For start <= I <= stop, the problem size is *M**I.
`-loop_start` *M*	Initial number of times to run the problem during calibration. The default is one. Calibration may increase or decrease *M* to ensure that the total measured time is within a threshold of the goal time, set by `-ms`.

`-mem`	In the table of results, the left column usually is the number of points. This option switches that value to the memory footprint of the problem. It can be useful in illustrating how performance is affected by problem size.
`-mfile` *F*	Reads the progression values from file *F*. Implies `-start` *M* where *M* is the first value read from the file.
`-ms` *M*	Sets the goal time that each measurement should take, in hundredths-of-a-second. Defaults to 25 (250 ms).
`-ops`	Reports millions of operations per second (MOP/s).
`-p:`*KEY VALUE*	Sets the benchmark specific parameter *KEY* to the value *VALUE*. *KEY* and *VALUE* are string values. See the individual benchmarks for valid uses.
`-param` *M*	Sets the user-specified parameter, where value is used in some instances to override a default value of a parameter, such as the number of rows or columns in an input matrix for example. The effect, if any, is dependent on each individual benchmark. Using -param to set parameters is being phased out in favor of `-p:`*KEY VALUE*.
`-pause`	When using multiple processes, stop each one to allow a debugger to be attached. The process continues after the user types a single character.
`-pool def`	Uses the default memory pool, which allocates aligned memory with the `new` operator.
`-pool huge`	Uses the *huge* page memory pool, which allocates aligned memory in huge pages. Uses the default number of huge pages.
`-pool huge:`*N*	Uses the *huge* page memory pool, which allocates aligned memory in huge pages. Allocates *N* huge memory pages.
`-prof`	Enables profiling. An accumulate-mode profile output is generated for each problem size.
`-pts`	Reports millions of points per second (MPT/s). This is the default.
`-riob`	Reports millions of input operations per second (MB/s).
`-samples` *M*	Sets the number of samples taken to *M*. Defaults to one. When *M* > 2, the median value is reported.
`-show_loop`	Displays the number of times the algorithm is run for each problem size.
`-show_time`	Displays the total time the algorithm runs for each problem size.
`-single` *M*	Runs the benchmark once on the problem size 2^*M* only.
`-start` *M*	Sets the starting problem size to 2^*M*. Defaults to two (four points). Uses this size for calibration also.
`-steady` *M*	Runs the benchmark in a loop on the problem size 2^*M* only. The user must stop the program manually.

`-stop` *M*	Sets the stopping problem size to 2^*M*. Defaults to 21 (2097152 points).
`-verbose`	Sets a flag to enable printing of internal information. Used by authors during benchmark development.
`-wiob`	Reports millions of output operations per second (MB/s).

Appendix B
Benchmark Usage

This appendix presents the *usage* display from each benchmark program. A benchmark prints the display when invoked with no parameters or with the single parameter `-0`.

B.1. CELL Benchmark Usage

This appendix presents the displays from the CELL benchmarks.

B.1.1. fastconv

```
fastconv -- fast convolution benchmark for Cell BE
 Sweeping pulse size:
    -1 -- IP, native complex, distributed, single-expr
    -2 -- IP, native complex, distributed, Fastconv object

   -11 -- OP, inter complex,  non-dist
   -12 -- IP, inter complex,  non-dist, single FC
   -13 -- IP, inter complex,  non-dist, multi FC

   -21 -- OP, split complex,  non-dist
   -22 -- IP, split complex,  non-dist
   -23 -- IP, split complex,  non-dist, multi FC

   -32 -- Multiple coeff vectors in time domain, IP, native complex, \
non-dist, single FC
   -42 -- Multiple coeff vectors in freq domain, IP, native complex, \
non-dist, single FC
```

B.2. conv

```
conv -- 1D convolution
   -1 -- float, support=full
   -2 -- float, support=same
   -3 -- float, support=min
   -4 -- complex<float>, support=full
   -5 -- complex<float>, support=same
   -6 -- complex<float>, support=min

 Parameters:
  -param N      -- size of coefficient vector (default 16)
  -start N     -- starting problem size 2^N (default 4 or 16 points)
  -loop_start N -- initial number of calibration loops (default 5000)
```

B.3. conv2d

```
conv2d -- 2D convolution
   -1 -- float, support=full
   -2 -- float, support=same
   -3 -- float, support=min

 Parameters:
  -p:rows     ROWS -- rows in input matrix
```

```
                        (default 16)
  -p:m        M     -- rows in coefficient matrix
                        (default 3)
  -p:n        N     -- columns in coefficient matrix
                        (default 3)
  -p:mn       MN    -- if not zero, set both M and N
                        to MN (default 0)
  -start      N     -- starting problem size 2^N
                        (default 4, that is 16 points)
  -loop_start N     -- initial number of calibration
                        loops (default 5000)
```

B.4. copy

```
copy -- vector copy
   -1 -- local copy (A = B))
   -5 -- local copy (memcpy))
 Using assignment (A = B):
  -10 -- float root copy      (root -> root)
  -11 -- float scatter        (root -> all)
  -12 -- float gather         (all  -> root)
  -13 -- float dist copy      (all  -> all)
  -14 -- float point-to-point (p0   -> p1)
  -15 -- float scatter2       (root -> all non-root)
 Using Setup_assign:
  -20 -- float root copy      (root -> root)
  -21 -- float scatter        (root -> all)
  -22 -- float gather         (all  -> root)
  -23 -- float dist copy      (all  -> all)
  -24 -- float point-to-point (p0   -> p1)
  -25 -- float scatter2       (root -> all non-root)

 MPI low-level Par_assign directly:
  -100-105 -- Chained_assign
  -110-115 -- Blkvec_assign
  -150-155 -- Chained_assign (non-amortized setup)
  -160-165 -- Blkvec_assign (non-amortized setup)

 PAS low-level Par_assign directly:
  -200-205 -- Pas_assign
  -210-215 -- Pas_assign_eb
  -220-225 -- Direct_pas_assign
  -250-255 -- Pas_assign (non-amortized setup)
  -260-265 -- Pas_assign_eb (non-amortized setup)
  -270-275 -- Direct_pas_assign (non-amortized setup)
```

B.5. corr

```
corr -- correlation
  option dim backend support           type biased
  ------ --- ------- ------- -------------- ------
     -1   1    n/a     full           float   yes
     -2   1    n/a     same           float   yes
```

```
    -3   1    n/a     min            float   yes
    -4   1    n/a     full           float    no
    -5   1    n/a     same           float    no
    -6   1    n/a     min            float    no
    -7   1    n/a     full  complex<float>   yes
    -8   1    n/a     same  complex<float>   yes
    -9   1    n/a     min   complex<float>   yes
   -10   1    n/a     full  complex<float>    no
   -11   1    n/a     same  complex<float>    no
   -12   1    n/a     min   complex<float>    no
   -13   2    [O]     full           float   yes
   -14   2    [O]     same           float   yes
   -15   2    [O]     min            float   yes
   -16   2    [O]     full           float    no
   -17   2    [O]     same           float    no
   -18   2    [O]     min            float    no
   -19   2    [O]     full  complex<float>   yes
   -20   2    [O]     same  complex<float>   yes
   -21   2    [O]     min   complex<float>   yes
   -22   2    [O]     full  complex<float>    no
   -23   2    [O]     same  complex<float>    no
   -24   2    [O]     min   complex<float>    no
   -25   2    [G]     full           float   yes
   -26   2    [G]     same           float   yes
   -27   2    [G]     min            float   yes
   -28   2    [G]     full           float    no
   -29   2    [G]     same           float    no
   -30   2    [G]     min            float    no
   -31   2    [G]     full  complex<float>   yes
   -32   2    [G]     same  complex<float>   yes
   -33   2    [G]     min   complex<float>   yes
   -34   2    [G]     full  complex<float>    no
   -35   2    [G]     same  complex<float>    no
   -36   2    [G]     min   complex<float>    no

  Notes:
   [O] -- optimized generic backend
   [G] -- generic backend

  Parameters:
   -param N      -- size of coefficient vector
                    (default 16)
   -start N      -- starting problem size 2^N
                    (default 4, or 16 points)
   -loop_start N -- initial number of calibration
                    loops (default 5000)
```

B.6. CUDA Benchmark Usage

This appendix presents the displays from the CUDA benchmarks.

B.6.1. copy

```
CUDA copy -- fixed rows\n"
   -1 -- host to device copy\n"
   -2 -- device to host copy\n"
   -3 -- host->device->host copy (A = B)\n"
   -4 -- device to shared copy\n"
   -5 -- device to device copy\n"
   -6 -- device fill with zeroes\n"
CUDA copy -- fixed columns\n"
  -11 -- host to device copy\n"
  -12 -- device to host copy\n"
  -13 -- host->device->host copy (A = B)\n"
  -14 -- device to shared copy\n"
  -15 -- device to device copy\n"
  -16 -- device fill with zeroes\n"

Parameters:
  -p:rows ROWS (default 64)
  -p:size SIZE (default 2048)
```

B.6.2. fastconv

```
fastconv -- fast convolution benchmark
 Sweeping pulse size:
   -1 -- Out-of-place, phased
   -2 -- On-device, phased
   -3 -- On-device, interleaved

 Parameters (for sweeping convolution size, cases 1 through 10)
  -p:rows ROWS -- set number of pulses (default 64)

 Sweeping number of pulses:
  -11 -- Out-of-place, phased
  -12 -- On-device, phased
  -13 -- On-device, interleaved

 Parameters (for sweeping number of convolutions, cases 11 through \
20)
  -p:size SIZE -- size of pulse (default 2048)
```

B.6.3. fftm

```
fftm -- FFT/FFTM benchmark using CUDA
 Fixed rows, sweeping FFT size:
   -1 -- op  : out-of-place CC fwd fft
   -2 -- ip  : In-place CC fwd fft
   -3 -- dev : On-device CC fwd fft

 Parameters (for sweeping FFT size, cases 1 through 6)
  -p:rows ROWS -- set number of pulses (default 64)

 Fixed FFT size, sweeping number of FFTs:
  -11 -- op  : out-of-place CC fwd fft
```

```
  -12 -- ip  : In-place CC fwd fft
  -13 -- dev : On-device CC fwd fft

 Parameters (for sweeping number of FFTs, cases 11 through 16)
  -p:size SIZE -- size of pulse (default 2048)
```

B.6.4. transpose

```
CUDA transpose (direct - memory moves not timed)
 Sweeping column size:
   -1 -- Out-of-place, complex
 Sweeping row size:
  -11 -- Out-of-place, complex

CUDA transpose (normal - memory moves are timed)
 Sweeping column size:
  -21 -- Out-of-place, complex
 Sweeping row size:
  -31 -- Out-of-place, complex

 Parameters (for sweeping number of columns, cases 1, 21)
  -p:rows ROWS -- set number of rows (default 64)

 Parameters (for sweeping number of columns, cases 11, 31)
  -p:cols COLS -- set number of columns (default 2048)
```

B.6.5. vmmul

```
CUDA vmmul -- vector-matrix multiply
 Sweeping column size, vmmul<row>:
   -1 -- Out-of-place, complex
 Sweeping row size, vmmul<row>:
  -11 -- Out-of-place, complex
 Sweeping column size, vmmul<col>:
  -21 -- Out-of-place, complex
 Sweeping row size, vmmul<col>:
  -31 -- Out-of-place, complex

 Parameters (for sweeping number of columns, cases 1, 21)
  -p:rows ROWS -- set number of rows (default 64)

 Parameters (for sweeping number of columns, cases 11, 31)
  -p:cols COLS -- set number of columns (default 2048)
```

B.7. CVSIP Benchmark Usage

This appendix presents the displays from the CVSIP benchmarks.

B.7.1. fft

```
fft -- Fft (fast fourier transform)

Single precision
```

```
 Planning effor: estimate (number of times = 1):
   -1 -- op: out-of-place CC fwd fft
   -2 -- ip: in-place     CC fwd fft
   -5 -- op: out-of-place CC inv fft (w/scaling)
   -6 -- ip: in-place     CC inv fft (w/scaling)

 Planning effor: measure (number of times = 15): 11-16
 Planning effor: pateint (number of times = 0): 21-26

         // When there is no single precision
         // support, the following line replaces
         // the above.

Single precision FFT support not provided by library

 Planning effor: estimate (number of times = 1): 101-106
 Planning effor: measure (number of times = 15): 111-116
 Planning effor: pateint (number of times = 0): 121-126

         // When there is no double precision
         // support, the following line replaces
         // the above.

Double precision FFT support not provided by library
```

B.7.2. fftm

```
fftm -- Fftm (multiple fast fourier transform) benchmark
Single precision
 Fixed rows, sweeping FFT size:
   -1 -- op  : out-of-place CC fwd fft
   -2 -- ip  : In-place CC fwd fft

 Parameters (for sweeping FFT size, cases 1 through 6)
  -p:rows ROWS -- set number of pulses (default 64)

 Fixed FFT size, sweeping number of FFTs:
  -11 -- op  : out-of-place CC fwd fft
  -12 -- ip  : In-place CC fwd fft

 Parameters (for sweeping number of FFTs, cases 11 through 16)
  -p:size SIZE -- size of pulse (default 2048)
```

B.7.3. fir

```
fir -- FIR signal processing object benchmark
   -1 -- No state save, float
   -2 -- No state save, complex<float>
  -11 -- State save, float
  -12 -- State save, complex<float>

Parameters
  -param:k <size>  Kernel size (default 16)
  -param:d <size>  Decimation (default 1);
```

B.7.4. vmul

```
vmul -- vector multiplication
single-precision:
 Vector-Vector:
   -1 -- Vector<        float > * Vector<        float >
   -2 -- Vector<complex<float>> * Vector<complex<float>>
   -5 -- Vector<        float > * Vector<complex<float>>

double-precision:
  (101-113)
  (131-132)
```

B.8. dist_vmul

```
dist_vmul -- distributed vector multiplication
 Using Assignment
  -11 -- Local vmul (non-parallel) - float
  -12 -- Local vmul (non-parallel) - complex
  -21 -- Clique vmul               - float
  -22 -- Clique vmul               - complex
  -31 -- Pipelined vmul            - float
  -32 -- Pipelined vmul            - complex
 Using Setup_assign
  -41 -- Local vmul (non-parallel) - float
  -42 -- Local vmul (non-parallel) - complex
  -51 -- Clique vmul               - float
  -52 -- Clique vmul               - complex
  -61 -- Pipelined vmul            - float
  -62 -- Pipelined vmul            - complex
```

B.9. dot

```
dot -- dot product
  -1 -- float
  -2 -- complex<float>
  -3 -- float          -- generic backend
  -4 -- complex<float> -- generic backend
  -5 -- float          -- BLAS backend
  -6 -- complex<float> -- BLAS backend
  -7 -- float          -- CUDA backend
  -8 -- complex<float> -- CUDA backend

 Parameters:
  -start N     -- starting problem size 2^N (default 4 or 16 points)
  -loop_start N -- initial number of calibration loops (default 5000)
```

B.10. fastconv

```
fastconv -- fast convolution benchmark
 Sweeping pulse size:
```

```
   -1 -- Out-of-place, phased
   -2 -- In-place, phased
   -3 -- Psuedo in-place Fftm (in-place Fft), phased
   -4 -- Psuedo in-place Fftm (out-of-place Fft), phased
   -5 -- Out-of-place, interleaved
   -6 -- In-place, interleaved
   -7 -- In-place (w/tmp), interleaved
   -8 -- Foreach_vector, interleaved (2fv)
   -9 -- Fused expression, vector of coefficients (4vc)
  -10 -- Fused expression, matrix of coefficients (4mc)

 Parameters (for sweeping convolution size, cases 1 through 10)
  -p:rows ROWS -- set number of pulses (default 64)

 Sweeping number of pulses:
  -11 -- Out-of-place, phased
  -12 -- In-place, phased
  -13 -- Psuedo in-place Fftm (in-place Fft), phased
  -14 -- Psuedo in-place Fftm (out-of-place Fft), phased
  -15 -- Out-of-place, interleaved
  -16 -- In-place, interleaved
  -17 -- In-place (w/tmp), interleaved
  -18 -- Foreach_vector, interleaved (2fv)
  -19 -- Fused expression, vector of coefficients (4vc)
  -20 -- Fused expression, matrix of coefficients (4mc)

 Parameters (for sweeping number of convolutions, cases 11 through \
20)
  -p:size SIZE -- size of pulse (default 2048)

 Common Parameters
  -p:check {0,n}|{1,y} -- check results (default 'y')
```

B.11. fft

```
fft -- Fft (fast fourier transform)
Single precision
 Planning effor: estimate (number of times = 1):
   -1 -- op: out-of-place CC fwd fft
   -2 -- ip: in-place     CC fwd fft
   -3 -- bv: by-value     CC fwd fft
   -5 -- op: out-of-place CC inv fft (w/scaling)
   -6 -- ip: in-place     CC inv fft (w/scaling)
   -7 -- bv: by-value     CC inv fft (w/scaling)
 Planning effor: measure (number of times = 15): 11-16
 Planning effor: pateint (number of times = 0): 21-26


Double precision
 Planning effor: estimate (number of times = 1): 101-106
 Planning effor: measure (number of times = 15): 111-116
 Planning effor: pateint (number of times = 0): 121-126
```

B.12. fftm

```
fftm -- Fftm (multiple fast fourier transform) benchmark
Single precision
 Fixed rows, sweeping FFT size:
   -1 -- op  : out-of-place CC fwd fft
   -2 -- ip  : In-place CC fwd fft
   -3 -- pop : Psuedo out-of-place CC fwd fft
   -4 -- pip1: Psuedo in-place v1 CC fwd fft
   -5 -- pip2: Psuedo in-place v2 CC fwd fft
   -6 -- bv  : By-value CC fwd fft

 Parameters (for sweeping FFT size, cases 1 through 6)
  -p:rows ROWS -- set number of pulses (default 64)

 Fixed FFT size, sweeping number of FFTs:
  -11 -- op  : out-of-place CC fwd fft
  -12 -- ip  : In-place CC fwd fft
  -13 -- pop : Psuedo out-of-place CC fwd fft
  -14 -- pip1: Psuedo in-place v1 CC fwd fft
  -15 -- pip2: Psuedo in-place v2 CC fwd fft
  -16 -- bv  : By-value CC fwd fft

 Parameters (for sweeping number of FFTs, cases 11 through 16)
  -p:size SIZE -- size of pulse (default 2048)
```

B.13. FFTW3 Benchmark Usage

This appendix presents the displays from the FFTW3 benchmarks.

B.13.1. fft

```
fftw3/fft -- FFTW3 Fft (fast fourier transform)
Single precision, Interleaved complex
 Planning effor: estimate:
   -1 -- op: out-of-place CC fwd fft
   -2 -- ip: in-place     CC fwd fft
   -4 -- op: out-of-place CC fwd fft + UNALIGNED
 Planning effor: measure:
  -11 -- op: out-of-place CC fwd fft
  -12 -- ip: in-place     CC fwd fft
  -14 -- op: out-of-place CC fwd fft + UNALIGNED
  -15 -- op: out-of-place CC fwd fft + PRESERVE_INPUT
  -16 -- op: out-of-place CC fwd fft + PRESERVE_INPUT + ND
 Planning effor: patient:
  -21 -- op: out-of-place CC fwd fft
  -22 -- ip: in-place     CC fwd fft
  -24 -- op: out-of-place CC fwd fft + UNALIGNED
 Planning effor: exhaustive:
  -31 -- op: out-of-place CC fwd fft
  -32 -- ip: in-place     CC fwd fft
  -34 -- op: out-of-place CC fwd fft + UNALIGNED
```

```
Single precision, Split complex
 Planning effor: estimate:
  -51 -- op: out-of-place CC fwd fft
  -52 -- ip: in-place     CC fwd fft
 Planning effor: measure:
  -61 -- op: out-of-place CC fwd fft
  -62 -- ip: in-place     CC fwd fft
 Planning effor: patient:
  -71 -- op: out-of-place CC fwd fft
  -72 -- ip: in-place     CC fwd fft
```

B.13.2. fftm

```
fftw3/fftm -- FFTW3 FFTM (multiple fast fourier transform) benchmark
Single precision, interlaved, ESTIMATE
   -1 -- out-of-place CC fwd fft, fixed rows, row-major
  -11 -- out-of-place CC fwd fft, fixed cols, row-major
  -21 -- out-of-place CC fwd fft, fixed rows, col-major
  -31 -- out-of-place CC fwd fft, fixed cols, col-major

Single precision, interlaved, MEASURE
 -1-1 -- out-of-place CC fwd fft, fixed rows, row-major
 -111 -- out-of-place CC fwd fft, fixed cols, row-major
 -121 -- out-of-place CC fwd fft, fixed rows, col-major
 -131 -- out-of-place CC fwd fft, fixed cols, col-major

Single precision, interlaved, PATIENT
 -201 -- out-of-place CC fwd fft, fixed rows, row-major
 -211 -- out-of-place CC fwd fft, fixed cols, row-major
 -221 -- out-of-place CC fwd fft, fixed rows, col-major
 -231 -- out-of-place CC fwd fft, fixed cols, col-major

 Parameters for all cases
  -p:sw [0|1] -- save wisdom (default 0)

 Parameters (for sweeping FFT size, cases 1 through 6)
  -p:rows ROWS -- set number of pulses (default 64)

 Parameters (for sweeping number of FFTs, cases 11 through 16)
  -p:size SIZE -- size of pulse (default 2048)
```

B.14. fir

```
fir -- FIR signal processing object benchmark
 Sweep block size == problem size
   -1 -- No state save, float
   -2 -- No state save, complex<float>
  -11 -- State save,    float
  -12 -- State save,    complex<float>

Parameters for cases 1, 2, 11, 12
  -p:k <size>  Kernel size (default 16)
  -p:d <size>  Decimation (default 1)
```

```
 Sweep block size, fixed problem size
  -21 -- No state save, float
  -22 -- No state save, complex<float>
  -31 -- State save,    float
  -32 -- State save,    complex<float>

Parameters for cases 22, 32
  -p:k <size>  Kernel size (default 16)
  -p:size <size> Problem size (default 1)
```

B.15. HPEC_KERNEL Benchmark Usage

This appendix presents the displays from the HPEC_KERNEL benchmarks.

B.15.1. cfar

```
cfar -- Constant False Alarm Rate Detection

  type
    F   float
    D   double

  storage
    S   slice
    V   vector
    H   hybrid
                                      CFAR  CFAR
                                doppler range guard
        type storage beams   bins  gates cells
        ---- ------- ----- ------- ----- -----
   -1:    F     S      16      24     5     4
   -2:    F     S      48     128    10     8
   -3:    F     S      48      64    10     8
   -4:    F     S      16      16    20    16

  -11:    D     S      16      24     5     4
  -12:    D     S      48     128    10     8
  -13:    D     S      48      64    10     8
  -14:    D     S      16      16    20    16

  -21:    F     V      16      24     5     4
  -22:    F     V      48     128    10     8
  -23:    F     V      48      64    10     8
  -24:    F     V      16      16    20    16

  -31:    D     V      16      24     5     4
  -32:    D     V      48     128    10     8
  -33:    D     V      48      64    10     8
  -34:    D     V      16      16    20    16

  -41:    F     H      16      24     5     4   SSE
  -42:    F     H      48     128    10     8   SSE
```

```
  -43:    F     H      48      64    10     8   SSE
  -44:    F     H      16      16    20    16   SSE
```

B.15.2. cfar_c

```
cfar_c -- Constant False Alarm Rate Detection (C version)

                      CFAR  CFAR
                doppler range guard
         beams   bins  gates cells
         ----- ------- ----- -----
  -21:    16      24     5     4
  -22:    48     128    10     8
  -23:    48      64    10     8
  -24:    16      16    20    16

  -41:    16      24     5     4   simd (SSE)
  -42:    48     128    10     8   simd (SSE)
  -43:    48      64    10     8   simd (SSE)
  -44:    16      16    20    16   simd (SSE)
```

B.15.3. firbank

```
firbank -- FIR Filter Bank
  #   Set    Method   Data
  -1   1      Time     generated
  -2   2      Time     generated
 -11   1    Freq/FFT   generated
 -12   2    Freq/FFT   generated
 -21   1    Freq/FFTM  generated
 -22   2    Freq/FFTM  generated
 ---
 -51   1      Time     external
 -52   2      Time     external
 -61   1    Freq/FFT   external
 -62   2    Freq/FFT   external
 -71   1    Freq/FFTM  external
 -72   2    Freq/FFTM  external
```

B.15.4. svd

```
svd -- Singular Value Decomposition
  Fixed number of rows.
   -1: float           rows=500
   -2: float           rows=180
   -3: float           rows=150
  -11: complex<float>  rows=500
  -12: complex<float>  rows=180
  -13: complex<float>  rows=150

  Fixed number of columns.
  -21: float           cols=100
  -22: float           cols= 60
  -23: float           cols=150
```

```
  -31: complex<float>  cols=100
  -32: complex<float>  cols= 60
  -33: complex<float>  cols=150

  Fixed ratio of rows to columns.
  -41: complex<float>  1 : 1  storage=default
  -42: complex<float>  3 : 1  storage=default
  -43: complex<float>  1 : 3  storage=default
  -51: complex<float>  1 : 1  storage=svd_uvnos
  -52: complex<float>  3 : 1  storage=svd_uvnos
  -53: complex<float>  1 : 3  storage=svd_uvnos
```

B.16. IPP Benchmark Usage

This appendix presents the displays from the IPP benchmarks.

B.16.1. conv

```
conv -- IPP Convolution

   -1: Perform convolution on float vector.

  Default Parameters:
    -loop_start 5000 // Suggest number of loops for calibration.
    -start 4         // Starting problem size is 2^4 or 16.
    -param 16        // Size of coefficient kernel.
```

B.16.2. fft

```
ipp/fft -- IPP Fft (fast fourier transform) benchmark
Single precision
   -1 -- inter-op: interleaved, out-of-place CC fwd fft
   -2 -- inter-ip: interleaved, in-place     CC fwd fft (vrsn >= 5)
  -11 -- split-op: interleaved, out-of-place CC fwd fft
  -12 -- split-ip: interleaved, out-of-place CC fwd fft (vrsn >= 5)
```

B.16.3. fft_ext

```
fft_ext -- FFT using Ext_data to call IPP

  -1: FFT on vector complex<float>
```

B.16.4. mcopy

```
mcopy -- float matrix copy using IPP

   -1: rows -> rows   select algorithm
   -2: rows -> cols   select algorithm
   -3: cols -> rows   select algorithm
   -4: cols -> cols   select algorithm

  -11: rows -> rows   copy algorithm
  -12: rows -> cols   copy algorithm
```

```
  -13: cols -> rows   copy algorithm
  -14: cols -> cols   copy algorithm

  -22: rows -> cols   transpose algorithm
  -23: cols -> rows   transpose algorithm

  Default parameters:
    -stop 12        // Largest problem size is 2^12 or 4096
```

B.16.5. vmul

```
vmul -- IPP vector multiply

    -1: float           VSIPL++ allocation
    -2: complex<float>  VSIPL++ allocation

   -12: complex<float>  IPP allocation (ippsMul_32fc)   (in place)
   -22: complex<float>  IPP allocation (ippsMul_32fc_I) (in place)

  -101: float           IPP allocation (ippsMul_32f)
  -102: complex<float>  IPP allocation (ippsMul_32fc)
```

B.17. LAPACK Benchmark Usage

This appendix presents the displays from the LAPACK benchmarks.

B.17.1. qrd

```
qrd -- QR factorization.
   -1: by    row, storage=qrd_nosaveq
   -2: by    row, storage=qrd_saveq1
   -3: by    row, storage=qrd_saveq

  -11: by column, storage=qrd_nosaveq
  -12: by column, storage=qrd_saveq1
  -13: by column, storage=qrd_saveq
```

B.18. maxval

```
maxval -- find maximum value, and its index, in a vector
  -1 -- local vector -- random numbers
  -2 -- local vector -- forward ramp
  -3 -- local vector -- reverse ramp
  -4 -- mapped vector, parallel backend -- random numbers
  -5 -- mapped vector, parallel backend -- forward ramp
  -6 -- mapped vector, parallel backend -- reverse ramp
  -7 -- mapped vector,  generic backend -- random numbers
  -8 -- mapped vector,  generic backend -- forward ramp
  -9 -- mapped vector,  generic backend -- reverse ramp
```

B.19. mcopy

```
mcopy -- matrix copy with and without transpose
    -1: local,         float,  rows <- rows, assignment
    -2: local,         float,  rows <- cols, assignment
    -3: local,         float,  cols <- rows, assignment
    -4: local,         float,  cols <- cols, assignment
    -5: local, complex<float>, rows <- rows, assignment
    -6: local, complex<float>, rows <- cols, assignment
    -7: local, complex<float>, cols <- rows, assignment
    -8: local, complex<float>, cols <- cols, assignment
   -11: local,         float,  rows <- rows, memcpy
   -14: local,         float,  cols <- cols, memcpy
   -21:    pb,         float,  rows <- rows, assignment
   -22:    pb,         float,  rows <- cols, assignment
   -23:    pb,         float,  cols <- rows, assignment
   -24:    pb,         float,  cols <- cols, assignment
   -31:   par,        float,  Map<>(np,1) <- Map<>(np,1), assignment
   -32:   par,        float,  Map<>(np,1) <- Map<>(1,np), assignment
   -41:   par,         float,  Map<>(np,1) <- Map<>(np,1), setup \
assignment
   -42:   par,         float,  Map<>(np,1) <- Map<>(1,np), setup \
assignment
  -102: local,           int,  rows <- cols, assignment

 Notes:
    pb -- plain blocks
   par -- distributed blocks
```

B.20. meansqval

```
meansqval -- mean of the squared values in a view
   -1: vector, float, random values
   -2: vector, float, index is 0
   -3: vector, float, index is size-1
  -11: vector,   int, random values
  -21: vector, float, random values, get/put
  -31: vector, unsigned short, random values
  -32: vector, unsigned short, random values, no index result
  -33: vector, unsigned short, random values, no index result, \
unsigned long result
 -101: matrix, float
```

B.21. meanval

```
meanval -- mean of the values in a view
   -1: vector, float, random values
   -2: vector, float, index is 0
   -3: vector, float, index is size-1
  -11: vector,   int, random values
  -21: vector, float, random values, get/put
  -31: vector, unsigned short, random values
```

```
  -32: vector, unsigned short, random values, no index result
  -33: vector, unsigned short, random values, no index result, \
unsigned long result
 -101: matrix, float
```

B.22. memwrite

```
memwrite -- memory write bandwidth
  -1 -- write a float scalar into all elements of a view
  -2 -- same using an explicit loop
```

B.23. memwrite_simd

```
memwrite_simd -- SIMD memory write bandwidth
  -1 -- write a float scalar into all elements of a view
        using an explicit SIMD loop
  -2 -- same using a loop unrolled 4 times
```

B.24. MPI Benchmark Usage

This appendix presents the displays from the MPI benchmarks.

B.24.1. alltoall

```
alltoall -- MPI alltoall.

  Method
  ------------
  alltoall     -- MPI_alltoall
  alltoallv    -- MPI_alltoall with local copy
  isend        -- MPI_isend / MPI_recv
  isend_x      -- MPI_isend / MPI_recv with local copy
  persistent   -- MPI 'persistant' communication
  persistent_x -- MPI 'persistant' communication with local copy

    -1: rows -> rows, alltoall
    -2: rows -> cols, alltoall

   -11: rows -> rows, isend
   -12: rows -> cols, isend

   -21: rows -> rows, isend_x
   -22: rows -> cols, isend_x

   -31: rows -> rows, persistent
   -32: rows -> cols, persistent

   -41: rows -> rows, persistent_x (copy, single row, phase=1)
   -42: rows -> cols, persistent_x (copy, single row, phase=1)

  -101: rows -> rows, alltoall
  -102: rows -> cols, alltoall
```

```
  -111: rows -> rows, isend
  -112: rows -> cols, isend

  -121: rows -> rows, isend_x
  -122: rows -> cols, isend_x

  -131: rows -> rows, persistent
  -132: rows -> cols, persistent

--- FIXED ROWS from here down ---

  -141: rows -> rows, persistent_x (nocopy, single row, phase=2)
  -142: rows -> cols, persistent_x (nocopy, single row, phase=2)
  -143: rows -> rows, persistent_x (nocopy, single row, phase=2) (np)
  -144: rows -> cols, persistent_x (nocopy, single row, phase=2) (np)

  -151: rows -> rows, persistent_x (nocopy, single row, phase=1)
  -152: rows -> cols, persistent_x (nocopy, single row, phase=1)
  -153: rows -> rows, persistent_x (nocopy, single row, phase=1) (np)
  -154: rows -> cols, persistent_x (nocopy, single row, phase=1) (np)

  -161: rows -> rows, persistent_x (nocopy, single row, phase=0)
  -162: rows -> cols, persistent_x (nocopy, single row, phase=0)
  -163: rows -> rows, persistent_x (nocopy, single row, phase=0) (np)
  -164: rows -> cols, persistent_x (nocopy, single row, phase=0) (np)

  -201: rows -> rows, persistent_x (copy, single row, phase=2)
  -202: rows -> cols, persistent_x (copy, single row, phase=2)

  -171: rows -> rows, alltoallv (nocopy)
  -172: rows -> cols, alltoallv (nocopy)
  -173: rows -> rows, alltoallv (nocopy) (np)
  -174: rows -> cols, alltoallv (nocopy) (np)

  -181: rows -> rows, alltoallv (copy)
  -182: rows -> cols, alltoallv (copy)
  -183: rows -> rows, alltoallv (copy) (np)
  -184: rows -> cols, alltoallv (copy) (np)

  Parameter:
               default
               ----------------
    -1..-99
      -param    1  Number of columns per row
      -stop    11  Stop at 2^11
    -100..
      -param   48  Number of rows
      -stop    15  Stop at 2^15
```

B.24.2. copy

```
copy -- MPI point-to-point messaging (aka "copy").
                   vector  struct
```

```
                    ------  ------
  -10: isend           no      no
  -11: isend        early      no
  -12: isend        early   early
  -13: isend        early    late
  -14: isend         late      no
  -15: isend         late    late
  -20: persistent      no      no
  -21: persistent   early      no
  -22: persistent   early   early
```

B.25. prod

```
prod -- matrix-matrix product
    -1 -- default implementation, float
    -2 -- default implementation, complex<float>
    -3 -- generic implementation, float
    -4 -- generic implementation, complex<float>
    -5 --    BLAS implementation, float
    -6 --    BLAS implementation, complex<float> {interleaved only}
   -11 -- default impl with transpose, float
   -12 -- default impl with transpose, complex<float>
   -13 -- default impl with hermetian, complex<float>
  -103 -- generic impl, plain blocks, float
  -104 -- generic impl, plain blocks, complex<float>
```

B.26. prod_var

```
prod_var -- Randy Judd's matrix-matrix product variations
   -1 -- direct matrix product using VSIPL function
   -2 -- prod_2:    Alg 1.1.8 Outer Product from G&VL
   -3 -- prod_3:    Alg 1.1.7 Saxpy Version G&VL
   -4 -- prod_3c:
   -5 -- prod_3sv:  Alg 1.1.7 Saxpy Version G&VL using subview
   -6 -- prod_4:    Alg 1.1.5 ijk variant G&VL
   -7 -- prod_4t:   Alg 1.1.5 ijk variant G&VL with tmp storage
   -8 -- prod_4trc:
   -9 -- prod_4tsv:
  -10 -- prod_4dot:
  Options  -1 through -10 use float matrices.
  Options -11 through -20 are the same as the above, but
                          use complex<float> matrices.
```

B.27. pwarp

```
pwarp -- Perspective_warp
 Object:
   -1 -- float
   -2 -- char
 Funcion:
  -11 -- float
  -12 -- char
```

```
Parameters:
   -p:rows ROWS -- set image rows (default 16)
```

B.28. SAL Benchmark Usage

This appendix presents the displays from the SAL benchmarks.

B.28.1. fastconv

```
fastconv -- Fast Convolution

    -2: pulse fixed, phased, interleaved complex
    -6: pulse fixed, interleaved, interleaved complex

   -12: range cells fixed, phased, interleaved complex
   -16: range cells fixed, interleaved, interleaved complex

  -102: pulse fixed, phased, split complex
  -106: pulse fixed, interleaved, split complex

  -112: range cells fixed, phased, split complex
  -116: range cells fixed, interleaved, split complex

  Parameters: default
    -cal         4  calibrate with size 2^4
    -start       4  start with size 2^4
    -stop       16  stop  with size 2^16
    -loop_start 10  run problem 10 times for calibration
    -param      64  number of pulses / size of range
```

B.28.2. fft

```
fft -- SAL FFT
        in-place complex scale
   -1:      no    split    no
   -2:     yes    split    no
   -5:      no    split   yes
   -6:     yes    split   yes

  -11:      no    inter    no
  -12:     yes    inter    no
  -15:      no    inter   yes
  -16:     yes    inter   yes
```

B.28.3. lvgt

```
SAL lvgt
  -1 -- SAL lvgtx      (float) Z(i) = A(i) > B(i) ? 1    : 0
  -2 -- SAL lvgtx/vmul (float) Z(i) = A(i) > B(i) ? A(i) : 0
 -11 -- C (float) Z(i) = A(i) > B(i) ? 1    : 0
 -12 -- C (float) Z(i) = A(i) > B(i) ? A(i) : 0
```

B.28.4. memwrite

```
memwrite -- SAL memory write
  -1: use vfillx to fill a vector with a scalar
```

B.28.5. vatan2

```
atan2 -- SAL benchmark;
   -1 : SAL vatan2x;
   -2 : SAL vatan2dx;
   -3 : SAL vatan2x, input strides 2;
  -11 : C = atan2(A,B) {float};
  -12 : C = atan2(A,B) {double};
  -13 : C = atan2(A,B) {float}, input strides 2;
  -21 : C = atan2(A,B) {float}, scalar;
```

B.28.6. vma

```
vma -- SAL vector multiply-add
    -1: vma_x
   -11: vsmax
   -12: cvsmax
   -13: zvsmax
```

B.28.7. vmul

```
vmul -- SAL vector multiply
   -1: vmulx
   -2: cvmulx
   -3: zvmulx

  -11: vsmulx
  -13: cvsmulx
  -14: zvsmulx

  -31: vmulx
  -32: cvmulx B = A * B
  -33: cvmulx A = A * B
```

B.28.8. vthresh

```
SAL vthres
  -1 -- SAL vthresx (float) Z(i) = A(i) > b ? A(i) : 0
  -2 -- SAL vthrx   (float) Z(i) = A(i) > b ? A(i) : b (optional)
 -11 -- C           (float) Z(i) = A(i) > b ? A(i) : 0
```

B.29. sfilt

```
sfilt -- Separable_filter
   -1 -- float (min_zeropad)

Parameters:
   -p:m M       -- set filter size M (default 3)
   -p:n N       -- set filter size N (default 3)
```

```
   -p:mn MN     -- set filter sizes M and N at once
   -p:rows ROWS -- set image rows (default 16)
```

B.30. sort

```
Sort benchmark
   -1: float,  ascending, std::sort
   -2:   int,  ascending, std::sort
   -3: float, descending, std::sort
   -4:   int, descending, std::sort
  -11: float,  ascending, sort_data in-place
  -12:   int,  ascending, sort_data in-place
  -13: float, descending, sort_data in-place
  -14:   int, descending, sort_data in-place
  -21: float,  ascending, sort_data out-of-place
  -22:   int,  ascending, sort_data out-of-place
  -23: float, descending, sort_data out-of-place
  -24:   int, descending, sort_data out-of-place
  -31: float,  ascending, sort_indices
  -32:   int,  ascending, sort_indices
  -33: float, descending, sort_indices
  -34:   int, descending, sort_indices
```

B.31. stream

```
stream -- VSIPL++ STREAM benchmark
STREAM_Composite:
   -1 -- composite (VSIPL++ float)
   -2 -- composite (VSIPL++ double)
   -3 -- composite (VSIPL++ int)
 -101 -- composite (C float)
 -102 -- composite (C double)
 -103 -- composite (C int)

 Parameters
  -p:check      {0,1,n,y} -- check benchmark result (default y)
  -p:streamprnt {0,1,n,y} -- print STREAM results   (default n)
```

B.32. sumsqval

```
sumsqval -- sum of the squared values in a view
   -1: vector, float, random values
   -2: vector, float, index is 0
   -3: vector, float, index is size-1
  -11: vector,   int, random values
  -21: vector, float, random values, get/put
  -31: vector, unsigned short, random values
  -32: vector, unsigned short, random values, no index result
  -33: vector, unsigned short, random values, no index result, \
unsigned long result
 -101: matrix, float
```

B.33. sumval

```
sumval -- sum of the values in a view
   -1: vector, float, random values
   -2: vector, float, index is 0
   -3: vector, float, index is size-1
  -11: vector,   int, random values
  -21: vector, float, random values, get/put
  -31: vector, unsigned short, random values
  -32: vector, unsigned short, random values, no index result
  -33: vector, unsigned short, random values, no index result, \
unsigned long result
 -101: matrix, float
```

B.34. sumval_simd

```
sumval_simd -- SIMD sumval
    -1 -- SIMD sumval (overhead included)
    -2 -- SIMD sumval (overhead not included)
  -101 -- SIMD sumval (overhead included, loop unrolled 4 times)
  -102 -- SIMD sumval (overhead not included, loop unrolled 4 times)
```

B.35. svmul

```
svmul -- scalar-vector multiplication
single-precision:
 Scalar-Vector:
   -1 --                float   * Vector<        float >
   -2 --                float   * Vector<complex<float>>
   -3 --        complex<float>  * Vector<complex<float>>
   -4 --                float   * Vector<       int32_t>
  -15 -- t_svmul3
  -15 -- t_svmul4

double-precision:
  (101-113)
  (131-132)
```

B.36. task_switch

```
task_switch -- Task switch overhead benchmark
 Recommended:
   task_switch -11 -p:size 2048 -steady 10 -lat -show_loop -ms 200

 Sweeping pulse size:
   -1 -- Alternate Fftm/vmmul kernels
   -2 -- Alternate */+ kernels

 Parameters (for sweeping convolution size, cases 1 through 10)
  -p:rows ROWS -- set number of pulses (default 64)
```

```
 Sweeping number of pulses:
  -11 -- Alternate Fftm/vmmul kernels
  -12 -- Alternate */+ kernels

 Parameters (for sweeping number of convolutions, cases 11 through \
20)
  -p:size SIZE -- size of pulse (default 2048)

 Common Parameters
  -p:check {0,n}|{1,y} -- check results (default 'y')
```

B.37. vdiv

```
vdiv -- vector divide
                F  - float
                CF - complex<float>
   -1 -- vector element-wise divide -- F/F
   -2 -- vector element-wise divide -- CF/CF
   -5 -- real / complex vector element-wise divide
  -11 -- scalar-view vector divide (Scalar / View) -- F/F
  -12 -- scalar-view vector divide (Scalar / View) -- F/CF
  -13 -- scalar-view vector divide (Scalar / View) -- CF/CF
  -14 -- scalar-view vector divide (View / Scalar) -- F/F
  -15 -- scalar-view vector divide (View / Scalar) -- CF/CF
  -21 -- vector element-wise divide (using domains) -- F/F
  -22 -- vector element-wise divide (using domains) -- CF/CF
  -31 -- vector element-wise divide (in place) -- F/F
  -32 -- vector element-wise divide (in place) -- CF/CF
```

B.38. vma

```
vma -- vector multiply-add
  -11 -- V = A * B + C [complex]
  -21 -- V += A * B [float]
  -22 -- V += a * B [float]
  -31 -- V += A * B [complex]
  -32 -- V += a * B [complex]
  -51 -- V = A * B + C [complex], nonfused
 -201 -- V = a * B + C [complex*float + complex]
```

B.39. vmagsq

```
vmagsq -- vector magnitude {squared}
    -1 -- vector element-wise magnitude squared --         float
    -2 -- vector element-wise magnitude squared -- complex<float >
    -3 -- vector element-wise magnitude squared --         double
    -4 -- vector element-wise magnitude squared -- complex<double>
   -11 -- vector element-wise magnitude         --         float
   -12 -- vector element-wise magnitude         -- complex<float >
   -13 -- vector element-wise magnitude         --         double
   -14 -- vector element-wise magnitude         -- complex<double>
```

```
  -111 --    dense matrix magnitude -- float
  -112 -- nondense matrix magnitude -- float
```

B.40. vmmul

```
vmmul -- vector-matrix multiply
 Sweeping column size, row major:
   -1 -- Out-of-place, complex
   -2 -- Out-of-place, complex, using vmul
   -3 -- Out-of-place, complex, scaled
   -4 -- Out-of-place, complex, scaled, using vmul
 Sweeping row size, row major:
  -11 -- Out-of-place, complex
  -12 -- Out-of-place, complex, using vmul
  -13 -- Out-of-place, complex, scaled
  -14 -- Out-of-place, complex, scaled, using vmul
 Sweeping column size, column major:
  -21 -- Out-of-place, complex
  -22 -- Out-of-place, complex, using vmul
  -23 -- Out-of-place, complex, scaled
  -24 -- Out-of-place, complex, scaled, using vmul
 Sweeping row size, column major:
  -31 -- Out-of-place, complex
  -32 -- Out-of-place, complex, using vmul
  -33 -- Out-of-place, complex, scaled
  -34 -- Out-of-place, complex, scaled, using vmul

 Parameters (for sweeping number of columns, cases 1-4, 21-24)
  -p:rows ROWS -- set number of rows (default 64)

 Parameters (for sweeping number of columns, cases 11-14, 31-34)
  -p:cols COLS -- set number of columns (default 2048)
```

B.41. vmul

```
vmul -- vector multiplication
single-precision:
 Vector-Vector:
   -1 -- Vector<        float > * Vector<        float >
   -2 -- Vector<complex<float>> * Vector<complex<float>>
   -3 -- Vector<complex<float>> * Vector<complex<float>> (INTER)
   -4 -- Vector<complex<float>> * Vector<complex<float>> (SPLIT)
   -5 -- Vector<        float > * Vector<complex<float>>

  -21 -- t_vmul_dom1
  -22 -- t_vmul_dom1
  -31 -- t_vmul_ip1
  -32 -- t_vmul_ip1
 Vector-Vector (using explicit get()/put()):
  -41 -- Vector<        float > * Vector<        float >
  -42 -- Vector<complex<float>> * Vector<complex<float>>
 Vector-Vector (using mul() function):
  -51 -- mul(Vector<        float >, Vector<        float >)
```

```
  -52 -- mul(Vector<complex<float>>, Vector<complex<float>>)

double-precision:
  (101-113)
  (131-132)
```

B.42. vmul_c

```
vmul_c -- vector multiply using C arrays
     -1 -- view-view elementwise multiply -- float
     -2 -- view-view elementwise multiply -- complex<float>
     -6 -- view-view vector multiply      -- float
     -7 -- view-view vector multiply      -- complex<float>
    -11 -- scalar-view vector multiply    -- float
    -12 -- scalar-view vector multiply    -- complex<float>
```

B.43. vmul_par

```
vmul -- vector multiplication
  -91 -- Vector<        float > * Vector<        float > NONGLOBAL
  -92 -- Vector<complex<float>> * Vector<complex<float>> NONGLOBAL
 -101 -- Vector<        float > * Vector<        float > PAR
 -102 -- Vector<complex<float>> * Vector<complex<float>> PAR
 -111 -- Vector<        float > * Vector<        float > PAR sync
 -112 -- Vector<complex<float>> * Vector<complex<float>> PAR sync
 -121 -- Vector<        float > * Vector<        float > PAR local
 -122 -- Vector<complex<float>> * Vector<complex<float>> PAR local
 -131 -- Vector<       float > * Vector<       float > PAR early \
local
 -132 -- Vector<complex<float>> * Vector<complex<float>> PAR early \
local
 -141 -- Vector<       float > * Vector<       float > PAR setup \
assign
 -142 -- Vector<complex<float>> * Vector<complex<float>> PAR setup \
assign
```