Sourcery VSIPL++

User's Guide

Sourcery VSIPL++: User's Guide

CodeSourcery

Table of Contents

List of Tables

Part I. Tutorial

The sections in Part I form a tutorial for using Sourcery VSIPL++, covering serial programming, parallel programming, and performance analysis. You can follow along with the tutorial to learn how to use VSIPL++, and you can adapt the examples for use in your own programs.

Chapter 1
Fast Convolution

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.

1.1 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 1.3, “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 our 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 will multiply 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:

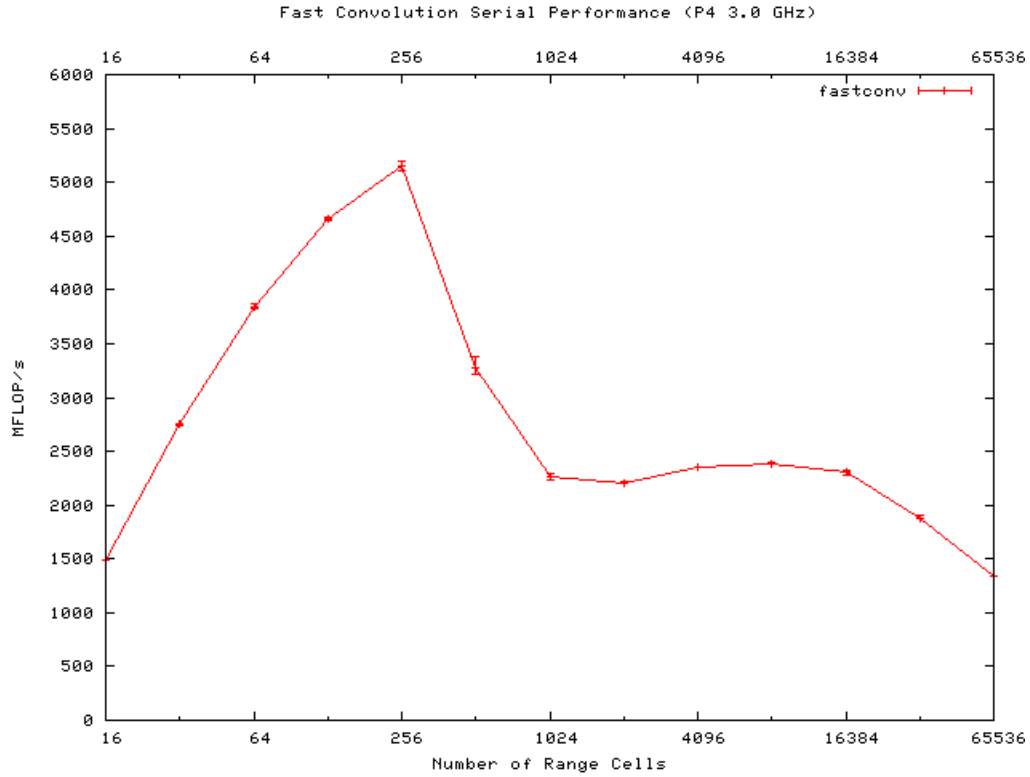
```
// Convert to the frequency domain.
for_fftm(data, tmp);

// 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.



1.2 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 1.1, “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.

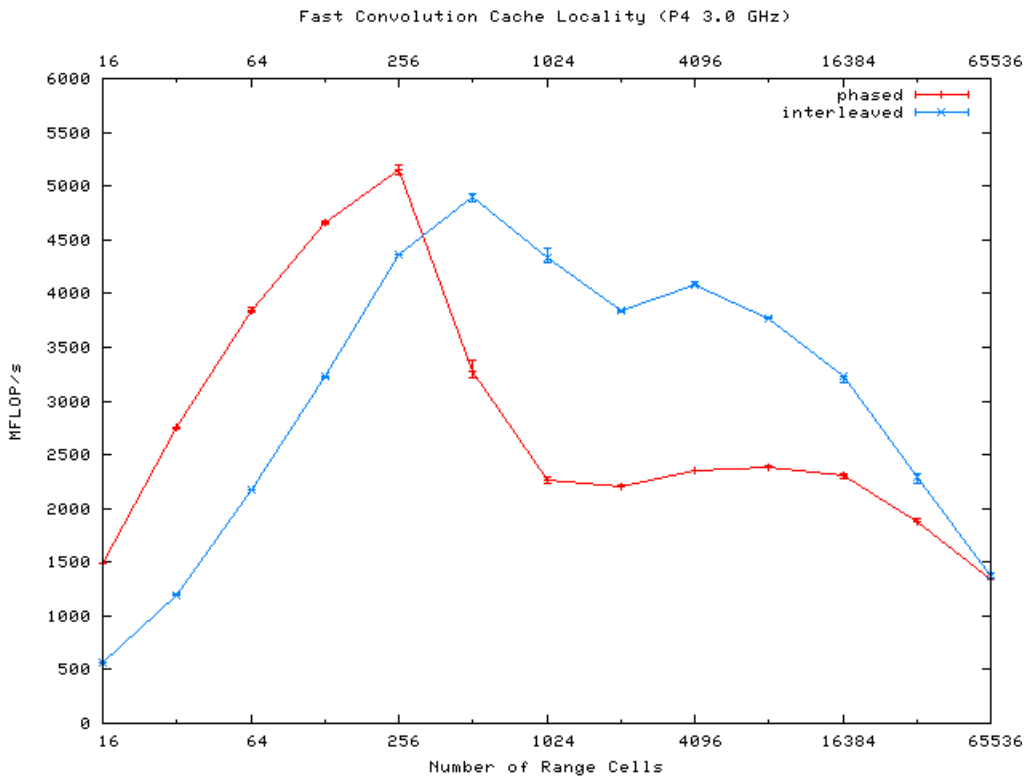
Explicitly iterating over the rows of the matrix (performing a forward FFT, elementwise multiplication, and an inverse FFT on each row before going on to the next one) will improve temporal locality. You can use this approach by using an explicit loop, rather than the implicit parallelism of `Fftm` and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



1.3 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 posix `read` and `write` it is necessary to obtain a pointer to data. Sourcery VSIPL++ provides multiple ways to do this: using *user-defined storage*, and using *external data access*. In this section you will use user-defined storage to perform I/O. Later, in ??? 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.

```
  auto_ptr<value_type> data_ptr(new value_type[npulse*nrange]);
```

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

```
  Dense<value_type, 2> data_block(Domain<2>(nrange, npulse), data_ptr.get());
```

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(..., data_ptr, 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 outout 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>

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.
  auto_ptr<value_type> data_ptr(new value_type[npulse*nrange]);

  // Blocks.
  Dense<2, value_type> block(Domain<2>(npulse, nrange), data_ptr.get());

  // 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);


  // Perform input I/O.
  view.block().release(false);
  size_t size = read(0, data_ptr.get(), 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);

  // Perform output I/O.
  view.block().release(true);
  size_t size = read(0, data_ptr.get(), 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.

1.4 Performing I/O with External Data Access

In this section, you will use *External Data Access* to get pointer to a block's data. External data access allows a pointer to any block's data to be taken, even if the block was not created with user-specified storage (or if the block is not a `Dense` block at all!) This capability is useful in context where you

cannot control how a block is created. To illustrate this, you will create a utility routine for I/O that works with any view passed as a parameter.

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 syncing necessary.

The `layout_type` parameter is an 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 it is necessary to access the data in a certain way, such as dense or row-major.

`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 imaginar values if complex. This will allow you to read data sequentially from a file.

The sync 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 object. Moreover, the block being accessed should not be used during that time.

Putting this 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 to put the data into. 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* 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;

  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_policy>
    ext(view.block(), SYNC_OUT);

  ifstream ifs(filename);

  ifs.read(reinterpret_cast<char*>(ext.data()),
```

```
            view.size() * sizeof(value_type));
}
```

Chapter 2
Parallel Fast Convolution

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.

2.1 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 implicity 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 above 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 instance, 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);
```

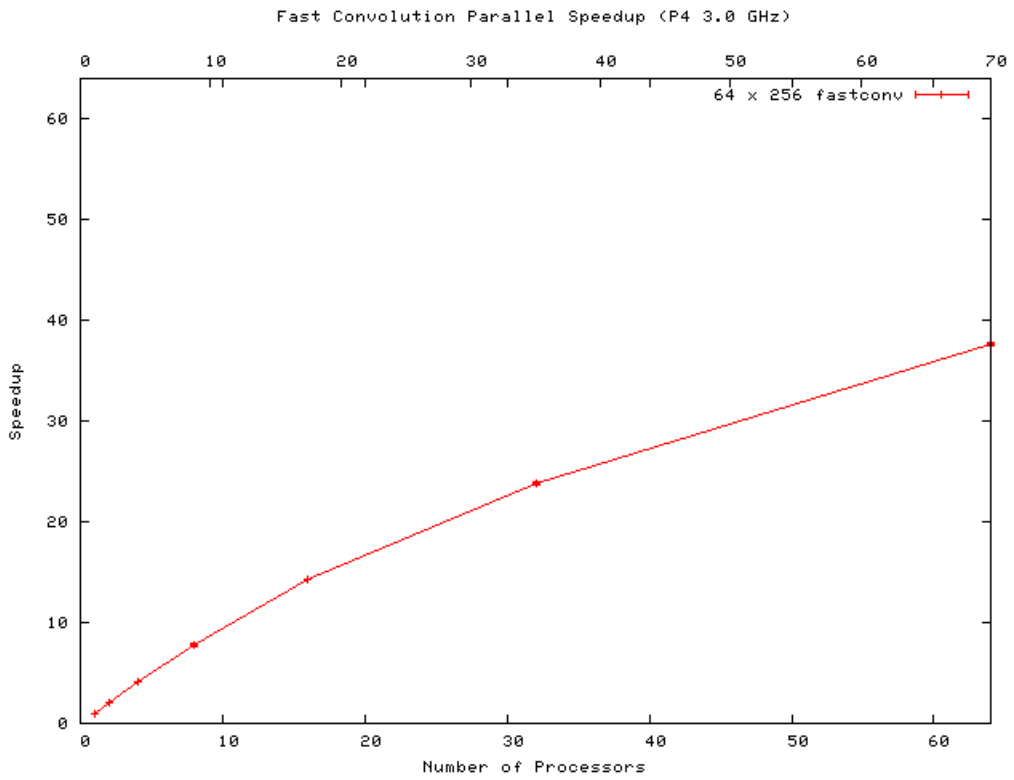
Because the application already uses implicitly parallel operators, no further changes are required. The entire algorithm (i.e., the part of the code that performs FFTs and vector-matrix multiplication) remains unchanged.

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.



2.2 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 2.1, “Parallel Fast Convolution”), but do not adjust the algorithm in use, the code in Section 1.2, “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 1.2, “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 2.1, “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));
  }
}
```

2.2.1 Implicit Parallelism: Parallel Foreach

You may feel that the original formulation 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 the temporal locality obtained with the style shown in the previous two sections.

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);
```

```
}
```

2.3 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++. Each processor must allocate the memory for its local portion of `data_in_block`. (All processors except the actual input processor will allocate zero bytes, since the input data is located on a single processor.) The code required to set up the views is:

```
block_type data_in_block(npulse, nrange, NULL, map);
view_type data_in(data_in_block);
view_type data   (npulse, nrange, map);
size_t size = subblock_domain(data_in).size();
auto_ptr<value_type> buffer(new value_type[size]);
data_in.block()->rebind(buffer);
```

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)  = ...;
```

Chapter 3
Performance

3.1 Library Profiling

Sourcery VSIPL++ provides library profiling features that speed application development by locating and quantifying the expensive computations in your algorithm. These profiling capabilities provide timing data for signal processing functions (such as FFTs), linear algebra computations (such as matrix multiply), and elementwise expressions (such as vector addition). When not required, profiling can be disabled at configure time, resulting in no application overhead. A full listing of functions covered is shown in the table below.

The profiler operates in two modes. In 'trace' mode, the time spent in each function is stored separately and presented in chronological order. This mode is preferred when a highly detailed view of program execution is desired. In 'accumulate' mode, the times and opcounts are summed so that an average runtime and MFLOP/s for the function can be computed. This is desirable when investigating a specific function's performance.

Section	Objects/Functions
signal	`Convolution, Correlation, Fft, Fir and Iir`
math.matvec	`dot, dotcvj, trans, herm, kron, outer and gemp`

See the file `profiling.txt` for a detailed explanation of the profiler output for each of the functions above.

3.1.1 Configuration Options

Before using profiling, you need to configure the library with profiling enabled. A high resolution and low overhead timer is also needed, such as the Pentium and x86_64 time-stamp counters. When building the library from source, you should enable a timer suitable for your particular platform along with the profiler itself. These may be subsequently disabled for the production version of the code without altering the source code. For 64-bit Intel and AMD processors, use:

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

```
--enable-profiler
```

If you are using a binary package on either of these platforms, then you need take no special steps, as the timer and profiler are already enabled for you.

3.1.2 Accumulating Profile Data

Using profiler's accumulate mode is easy. Simply construct a `Profile` object with the name of a log file as follows:

```
Profile profile("/dev/stdout", pm_accum);
```

Or, as `pm_accum` is the default mode:

```
Profile profile("/dev/stdout");
```

Profiled library functions will store timing data in memory while this object is in scope. The profile data is written to the log file when the object is destroyed. Note that for this reason, only one object of this type may be created at any given time.

The `examples/` subdirectory provided with the source distribution demonstrates this profiling mode using a 2048-point forward FFT followed by an inverse FFT scaled by the length. The profiler uses the timer to measure each FFT call and uses the size to compute an estimate of the performance. For each unique event, the profiler outputs an indentifying tag, the accumulated time spent 'in scope' (in

"ticks"), the number of times called, the total number of floating point operations performed per call and the computed performance in millions of flops per second. The time value may be converted to seconds by dividing it by the 'clocks_per_second' constant.

```
# mode: pm_accum
# timer: x86_64_tsc_time
# clocks_per_sec: 3591371008
#
# tag : total ticks : num calls : op count : mops
Fwd FFT C-C by_val 2048x1 : 208089 : 1 : 112640 : 1944.03
Inv FFT C-C by_val 2048x1 : 209736 : 1 : 112640 : 1928.77
```

This information is important in analyzing total processing requirements for an algorithm. However, care should be taken in interpreting the results to ensure that they are representative of the intended application. For example, in the above FFT the data will most likely not be resident in cache as it would be in some instances. With a well designed pipelined processing chain (typical of many embedded applications) the data will be in cache, yielding significantly better performance. To obtain a good estimate of the in-cache peformance, place the FFT in a loop so that it is called many times.

```
# mode: pm_accum
# timer: x86_64_tsc_time
# clocks_per_sec: 3591371008
#
# tag : total ticks : num calls : op count : mops
Fwd FFT C-C by_val 2048x1 : 6212808 : 100 : 112640 : 6511.26
```

This is only a portion of the analysis that would be necessary to predict the performance of a real-world application. Once you are able to accurately measure library performance, you may then extend that to profile your own application code, using the same features used internal to the library.

3.1.3 Trace Profile Data

This mode is used similarly to accumulate mode, except that an extra parameter is passed to the creation of the `Profile` object.

```
Profile profile("/dev/stdout", pm_trace);
```

This mode is useful for investigating the execution sequence of your program. The profiler simply records each library call as a pair of events, allowing you to see where it entered and exited scope in each case.

Long traces can result when profiling in this mode, so be sure to avoid taking more data than you have memory to store (and have time to process later). The output is very similar to the output in accumulate mode.

```
# mode: pm_trace
# timer: x86_64_tsc_time
# clocks_per_sec: 3591371008
#
# index : tag : ticks : open id : op count
1 : FFT Fwd 1D C-C by_val    2048x1 : 4688163420488244 : 0 : 112640
2 : FFT Fwd 1D C-C by_val    2048x1 : 4688163420626385 : 1 : 0
3 : FFT Inv 1D C-C by_val    2048x1 : 4688163420643116 : 0 : 112640
4 : FFT Inv 1D C-C by_val    2048x1 : 4688163420830298 : 3 : 0
```

For each event, the profiler outputs an event number, an indentifying tag, and the current timestamp (in "ticks"). The next two fields differ depending on whether the event is coming into scope or out of scope. When coming into scope, a zero is shown followed by the estimated count of floating point operations for that function. When exiting scope, the profiler displays the event number being closed followed by a zero. In all cases, the timestamp (and intervals) may be converted to seconds by dividing by the 'clocks_per_second' constant in the log file header.

3.1.4 Performance API

An additional interface is provided for getting run-time profile data. This allows you to selectively monitor the performance of a particular instance of a VSIPL class such as Fft, Convolution or Correlation.

Classes with the Performance API provide a function called `impl_performance` that takes a string parameter and returns single-precision floating point number.

The following call shows how to obtain an estimate of the performance in number of operations per second:

```
float mops = fwd_fft.impl_performance("mops");
```

An "operation" will vary depending on the object and type of data being processed. For example, a single-precison Fft object will return the number of single-precison floating-point operations performed per second.

The table below lists the current types of information available.

Parameter	Description
`mops`	performance in millions of operations per second
`count`	number of times invoked
`time`	total time spent performing the operation, in seconds
`op_count`	number of floating point operations per invocation
`mbs`	data rate in millions of bytes per second (not applicable in for all operations)

3.2 Application Profiling

The profiling mode provides an API that allows you to instrument your own code. Here we introduce a new object, the `Scope_event` class, and show you how to use it in your application.

To create a `Scope_event`, simply call the constructor, passing it the string that will become the event tag and, optionally, an integer value expressing the number of floating point operations that will be performed by the time the `Scope_event` object is destroyed. For example, to measure the time taken to compute a simple running sum of squares over a C array:

```
#include <vsip/initfin.hpp>
#include <vsip/support.hpp>
#include <vsip/impl/profile.hpp>

using namespace vsip;
using namespace impl;

int
main()
{
  vsipl init;
```

```
  int data[1024];
  for (int i = 0; i < 1024; ++i)
    data[i] = i;

  profile::Scope_enable scope("/dev/stdout" );

  // This computation will be timed and included in the profiler output.
  {
    profile::Scope_event user_event("sum of squares", 2 * 1024);

    int sum = 0;
    for (int i = 0; i < 1024; ++i)
      sum += data[i] * data[i];
  }

  return 0;
}
```

This resulting profile data is identical in format to that used for profiling library functions.

```
# mode: pm_accum
# timer: x86_64_tsc_time
# clocks_per_sec: 3591371008
#
# tag : total ticks : num calls : op count : mops
sum of squares : 18153 : 1 : 2048 : 405.174
```

Combining both application and library profiling is possible in either trace or accumulate modes. Performance events can be nested to help identify points of interest in your program. Events can be used to label different regions, such as "range processing" and "azimuth processing" for SAR. When examining the trace output, profile events for library functions, such as FFTs, will be nested within profile events for application regions.

Part II. Reference

The sections in Part II form a reference manual for Sourcery VSIPL++.

Chapter 4
API overview

4.1 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);
```

4.1.1 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());
```

4.1.2 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.);
```

4.1.3 Vectors

4.1.4 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();
```

4.1.5 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 explicitely 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);
```

4.2 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.

4.2.1 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 explicitely 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.set(0, 1.f);
```

4.2.1.1 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 explicitely 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.set(i, 1.f);
```

4.2.1.2 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.set(0, 1.f);

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

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

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.