VSIPL Fundamentals

Naming
Data Types
Blocks
&
Views

Randall Judd
SSC-SD
619 553 3086
judd@spawar.navy.mil
• VSIPL is an ANSI C API for signal processing.
• ANSI C does not support function overloading.
• There are a lot of functions in VSIPL multiplied by numerous data types and complicated by support functions.
• To prevent name collisions the Forum adopted guidelines to support the namespace.
The library namespace is reserved using `vsip_`

in front of all VSIPL functions
data types
to reduce the chance of name collisions with other libraries.
The Root Name

Nothing complicated here. These are the standard names like add, div, mul, fft etc.
The Depth and Shape

DEPTH (d) is
real(r) or complex (c)
and the
SHAPE (s) is
scalar (s), vector (v),
matrix (m), or tensor (t)
These go after the name space and before the root name.
Finally the precision ($p$)

There are lots of possible precision’s
For this tutorial we are interested in

- `_f` float
- `_i` integer
- `_bl` boolean
- `_vi`, `_mi`, `_ti` vector, matrix, tensor index

The precision is placed after the root name.
For Example - vsip_dsadd_p

• Real scalar complex vector add; float precision.
  - vsip_rscvadd_f

• If there can be no confusion the “real (r)” is understood
Real scalar vector add in float precision
  - vsip_svadd_f

Other add examples
  - vsip_vadd_f Add two real vectors
  - vsip_rcvadd_f Add a real to a complex vector
  - vsip_cvadd_f Add two complex vectors

• Not every possible expansion is defined by the specification. Only functionality explicitly defined is part of VSIPL.

QUESTIONS?
Data Types

• In the VSIPL header file are many type definitions specific for VSIPL functionality.
• Although VSIPL could have used ANSI data types for some precision's such as float and int this did not fit well with the naming guide and VSIPL supports many types not defined in ANSI.
• We will not cover every data type defined in the specification - just the ones we need for this tutorial.
• Data types not covered follow the same conventions and are all defined in the “Summary of VSIPL Types” chapter in the API document.
 Scalars

- Scalars are
  - `vsip_scalar_p` for real or `vsip_cscalar_p` for complex

- Scalars are defined in `vsip.h` so they are easy to manipulate.

- There are functions to manipulate scalars, for instance

```
vsip_cscalar_f a_complex;
vsip_scalar_f a_real=1, a_imaginary=2;
a_complex = vsip_cmplx_f(a_real,a_imaginary);
a_complex = vsip_cadd_f(a_complex,a_complex);
```
Index types

- A vector index (**vsip_scalar_vi**) is an unsigned type sufficient to index any vector in VSIPL.
  - The exact size is implementation dependent.
- A matrix index (**vsip_scalar_mi**) is defined by
  - typedef struct vsip_scalar_mi(vsip_scalar_vi r, c);
- A tensor index (**vsip_scalar_ti**) is a defined by
  - typedef struct vsip_scalar_ti(vsip_scalar_vi x, y, z);
- There is a convenience type
  - typedef vsip_index vsip_scalar_vi
Length, Stride and Offset

• These are covered in more detail when we talk about blocks and views

• For now
  - `vsip_length` is synonymous with `vsip_scalar_vi`
  - `vsip_offset` is synonymous with `vsip_scalar_vi`
  - `vsip_stride` is the same precision as `vsip_scalar_vi` but is signed.
    • Remember, `vsip_scalar_vi` is an unsigned integer.
Boolean

- VSIPL defines a boolean.
  - `vsip_scalar_bl`

- Zero is false, anything else is true.

- A function that returns `vsip_scalar_bl` does not necessarily return a one for true.
  - It is implementation dependent.
  - Always test against `VSIP_FALSE` since it has a standard value.

- A VSIPL copy function from Boolean to float will copy true as 1.0 and false as 0.0. A VSIPL copy from float or integer to Boolean will copy 0 as false and anything else as true.
Non-Scalar Data Types

- There are many other *abstract data* types (ADT) defined in VSIPL to accommodate the object based methodology of the library.
- The type definitions in `vsip.h` for all the ADT’s are incomplete.
  - ADT’s are defined in VSIPL by specifying their API and characteristics.
  - Vendors are free to create any internal structure they want as long as the ADT API follows the specification.
- We call these ADT’s *objects*. The two most common are the *block* object, and the *view* object.

QUESTIONS?
Blocks

• A block is a method for a user to reference data the same on any architecture without any knowledge of the underlying memory structure.

• A block may be thought of as a list of consecutive VSIPL scalars (data items) with a zero offset and a length of N.

• Data in a VSIPL block must be set and retrieved using VSIPL accessor functions. Any other method which retrieves or sets data in a block is not portable.
Some Terminology

- To make sure we are all on the same page
  - DATA are the numbers we work with.
  - DATA ARRAY is the memory where we store our data.
  - A User Array is memory the user has a pointer to and can access directly.
  - A VSIPL Array is memory associated with a block and allocated by the VSIPL Library. The layout and location of this memory is not available to the user.
  - A VSIPL data array must be accessed via a block
User Block

• Functionality to create a block object and associate the block with memory allocated by the user is available.
  – \texttt{vsip\_blockbind\_p} or \texttt{vsip\_cblockbind\_p}

• This type block is termed a \textit{user} block.

• Every block type has a defined layout of data in user memory which associates element by element with a block.

• User data and blocks is not difficult but is involved and will be covered in more detail later in the talk.
Derived Block

- All blocks are strongly typecast. A complex block and a real block are not the same type.
- Functions exist within VSIPL to create real views of complex data. The data in the real view and the corresponding data in the complex view exist in the same data array.
- When this is done the function will create a real block for the real view to be associated with. This type block is termed a *derived* block.
- We will cover this in more detail later in the talk.
Normal Block Creation

• The basic way to create a real float block is
  - `vsip_block_f *block=
    vsip_blockcreate_f(N,VSIP_MEM_NONE);`

• The basic way to create a complex float block is
  - `vsip_cblock_f *cblock=
    vsip_cblockcreate_f(N,VSIP_MEM_NONE);`

• Or a block of any other type is
  - `vsip_block_p *block=
    vsip_blockcreate_p(N,VSIP_MEM_NONE);`
  - where `_p` is one of the the precisions like `_bl, _vi, etc.`
Block Create Functionality

• The block create
  – creates and initializes the block object.
    • It does not initialize the data referenced by the block.
    – allocates memory for the data referenced by the block.
• If the block create fails a NULL is returned.
Block Destruction

• When no longer needed you must destroy all created objects.
  – `vsip_dblockdestroy_p(vsip_dblock_p* block)`

• It is not a mistake to destroy a NULL block.

• It is a mistake to use a destroyed block
  – A development mode library will warn you.
  – A production mode library probably will not.

• Note: Using a destroyed block results in weird program behavior.
  – sometimes it core dumps, sometimes it gives the wrong answer, sometimes it runs fine (but only on some machines).
So - What can you do with a block?

Not Very Much, but you can bind them to views; our next topic.

QUESTION?)
Views

• The view is the object used to access data as a vector, matrix or tensor.
• Every view will have an associated block bound to it.
• The view may not change its block after it is created.
• Every view will have an attribute called the offset which is the index into the block of the first element the view will access.
  – For a block referencing N data element the offset may be set at zero for the first element to N-1 for the last element.
Vector

• The vector is the simplest view. Besides the offset it has
  – stride
  – length

• To map the data from a block onto a vector view we have the block offset for element $a_n$ for a particular element $i$ in the vector.
  – $n = \text{offset} + i \times \text{stride}$
Matrix

• A matrix is a little more complicated. It has an offset, two strides, and two lengths.
  – row_stride and row_length
  – col_stride and col_length

• A stride is the distance in block elements between two consecutive elements in a dimension.

• To find matrix element \((i,j)\) in its block use
  – \(n = \text{offset} + i \times \text{col\_stride} + j \times \text{row\_stride}\)
The tensor in VSIPL is simply a three dimensional array. It has an offset, three strides, and three lengths.
- x_stride, x_length
- y_stride, y_length
- z_stride, z_length

To find the block element $n$ referenced by $(i,j,k)$ in a tensor use
- $n = \text{offset} + i \times z\_stride + j \times y\_stride + k \times x\_stride$
Example One

Lets create a 2 by 3 matrix that looks like

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>-3</td>
</tr>
</tbody>
</table>

```c
#include<vsip.h>
int main()
{
    int init=vsip_init((void*)0);
    vsip_block_f *a_block = vsip_blockcreate_f(6,VSIP_MEM_NONE);
    vsip_vview_f *a_vector = vsip_vbind_f(a_block,0,2,3);
    vsip_mview_f *a_matrix = vsip_mbind_f(a_block,0,3,2,1,3);
    vsip_vramp_f (1,1,a_vector);
    vsip_vputoffset_f(a_vector,1);
    vsip_vramp_f(-1,-1,a_vector);
    /* The matrix has all proper data in it so print*/
    { vsip_scalar_vi i,j;
        for(i=0;i<2;i++)
            for(j=0;j<3;j++) printf("%f ",vsip_mget_f(a_matrix,i,j));
        printf("\n");
    }
    vsip_vdestroy_f(a_vector);
    vsip_mdestroy_f(a_matrix);
    vsip_blockdestroy_f(a_block);
    vsip_finalize((void*)0);
    return 0;
}
```

QUESTIONS?
Detailed Example One

```c
#include<vsip.h> /* Always needed */

int main(){
    /* every VSIPL program starts with an initialization before any VSIPL function is called */
    int init=vsip_init((void*)0);
    /* VSIP_MEM_NONE is a hint to the library about memory allocation */
    vsip_block_f *a_block = vsip_blockcreate_f(6,VSIP_MEM_NONE);
    vsip_vview_f *a_vector = vsip_vbind_f(a_block,0,2,3); /* offset 0, stride 2, length 3 */
    /* below offset is 0, col_stride 3, col_length 2, row_stride 1, row_length 3 */
    vsip_mview_f *a_matrix = vsip_mbind_f(a_block,0,3,2,1,3);
    /* fill the vector with 1, 2, 3 */
    vsip_vramp_f (1,1,a_vector); /* start at 1, increment by 1, go to the end of the vector */
    vsip_vputoffset_f(a_vector,1); /* move the vector one element over in the block */
    vsip_vramp_f(-1,-1,a_vector); /* start at -1, increment by -1, go to the end of the vector */
    /* The matrix has all proper data in it so print*/
    /* Note the data is referenced by the block and the view indicates what part of the block */
    /* to access. So changing the data in the vector also changes the data in the matrix */
    for(i=0;i<2;i++){ /* index along column */
        for(j=0;j<3;j++) /* index along row */
            printf("%f ",vsip_mget_f(a_matrix,i,j));
        printf("\n");
    }
    vsip_vdestroy_f(a_vector); /* free the vector view objects memory */
    vsip_mdestroy_f(a_matrix); /* free the matrix view objects memory */
    vsip_blockdestroy_f(a_block); /* free the block objects memory and the memory associated with the data */
    vsip_finalize((void*)0); /* Must be called; All VSIPL objects must be destroyed first */
    return 0;
}
```
User Blocks

- VSIPL allows a user data array to be associated with a VSIPL block object.
  - For instance:
    ```c
    vsip_scalar_f *dta=(vsip_scalar_f*)malloc(N * sizeof(vsip_scalar_f));
    vsip_block_f *block=vsip_blockbind_f(dta,N,VSIP_MEM_NONE);
    
    vsip_block_f *block=vsip_blockbind_f((vsip_scalar_f*)NULL,N,VSIP_MEM_NONE);
    
    Before using (admitting) the block is rebound to data
    vsip_blockrebind_f(block,dta);
    
    Blocks associated with user data arrays are called user blocks.
Complex User Arrays

- A complex user data array is **NOT** an array of complex VSIPL scalars.
- A complex user data array is either an array of vsip_scalar_f in an interleaved format or two arrays of type vsip_scalar_f in split format.
  - For instance Interleaved:
    ```c
    vsip_scalar_f *dta=(vsip_scalar_f*)malloc(2 *N * sizeof(vsip_scalar_f));
    vsip_cblock_f *block=vsip_cblockbind_f(dta,(vsip_scalar_f*)NULL,N,VSIP_MEM_NONE);
    ```
  - Split
    ```c
    vsip_scalar_f *dta_r=(vsip_scalar_f*)malloc(N * sizeof(vsip_scalar_f));
    vsip_scalar_f *dta_i=(vsip_scalar_f*)malloc(*N * sizeof(vsip_scalar_f));
    vsip_cblock_f *block=vsip_cblockbind_f(dta_r,dta_i,N,VSIP_MEM_NONE);
    ```
Additional Info on User Data

• Matrix and Tensor index user arrays are not arrays of the corresponding scalar type; they are all interleaved and of type vsip_scalar_vi.
  – No split format exists for these types.

• Blocks are not split or interleaved; only user data is split or interleaved. Blocks are always consecutive elements of a particular type.
  – Blockbind always takes the number of elements in the block, not the number of words in the user data array.

• The preferred layout of user data for complex (for an implementation) is available both in the VSIPL header file and via the vsip_cstORAGE function.
  – Either layout is required to work on every implementation
Properties of User Blocks

• Blocks have state information of admitted or released.

• A user block, when created, is in the released state.

• A user block must be admitted before any data it references is changed.
  – You may, however, bind as many views as you want to the block in either state.

• When a block is admitted the program can require that the data in the block be made to agree with the data in the user data array.

• When a block is released the program can require that the data in the user array be made consistent with the data in the block.
**Blocks & Views**

### VSIPL Memory
- **block**
- **offset 0**
- **col_stride 1**
- **col_length 2**
- **row_stride 3**
- **row_length 3**
- **0 3 6**
- **1 4 7**

### User Data Array
- **blockbind**
- **offset 1**
- **stride 2**
- **length 3**
- **1 3 5**

### User Data Array
- **vector view**
- **0 1 2 3 4 5 6 7 8**

**Admit** block for VSIPL views to be valid and to update the data in the block.

**Release** block to update data in user array.

**QUESTIONS?**
Example two
User data

This is the same as Example One except we initialize the data using a user array.

```c
#include<vsip.h>
int main(){
    int init=vsip_init((void*)0);
    vsip_scalar_f dta[6]={1,-1,2,-2,3,-3};
    vsip_block_f *a_block = vsip_blockbind_f(dta,6,VSIP_MEM_NONE);
    vsip_mview_f *a_matrix = vsip_mbind_f(a_block,0,3,2,1,3);
    vsip_blockadmit_f(a_block,6,VSIP_TRUE);
    /* The matrix has all proper data in it so print*/
    { vsip_scalar_vi i,j;
      for(i=0;i<2;i++){
        for(j=0;j<3;j++) printf(“%f “,vsip_mget_f(a_matrix,i,j));
        printf(“\n“);
      }
    }
    vsip_blockrelease_f(a_block,VSIP_FALSE);
    vsip_malldestroy_f(a_matrix);
    vsip_finalize((void*)0);
    return 0;}
```

QUESTIONS?
Detailed Example Two

```c
#include<vsip.h>

int main(){
    int init=vsip_init((void*)0);
    /* Any method can be used to allocate memory */
    /* Data layout in user memory is defined in the specification */
    vsip_scalar_f dta[6]={1,-1,2,-2,3,-3};
    /* Block Bind creates a block */
    /* Note: One should not assume the data in the block is stored in */
    /* same memory as the data in the user array. */
    vsip_block_f *a_block = vsip_blockbind_f(dta,6,VSIP_MEM_NONE);
    /* As many views of the block as necessary may be created */
    vsip_mview_f *a_matrix = vsip_mbind_f(a_block,0,3,2,1,3);
    /* Before we can operate on the data in the block we must */
    /* admit it. If we want to ensure the block data is the same */
    /* as the user array data then we assert a true. A false assertion */
    /* means we don’t know what the data is in the block */
    vsip_blockadmit_f(a_block,6,VSIP_TRUE);
    /* The matrix has all proper data in it so print*/
    { vsip_scalar_vi i,j;
      for(i=0;i<2;i++){
        for(j=0;j<3;j++) printf("%f ",vsip_mget_f(a_matrix,i,j));
        printf("\n");
      }
    }
    /* Since we don’t need the user data back here we don’t need to */
    /* release the block; for this example, however, false indicates we */
    /* don’t care about the data in the block and after release we */
    /* don’t know what the data in the user array is. */
    vsip_blockrelease_f(a_block,VSIP_FALSE);
    /* The alldestroy function is a convenience to destroy both the view and */
    /* the block it is bound to in one function call. No other views should */
    /* be bound to the block if this call is used. */
    vsip_malldestroy_f(a_matrix);
    vsip_finalize((void*)0);
    return 0;}
```
Derived Block

• A few motivating facts
  – A view of type `vsip_vview_f` must bind a block of type `vsip_block_f`.
  – A view of type `vsip_cvview_f` must bind a block of type `vsip_cblock_f`.

• Consider the following function
  – `vsip_cvview_f *a_cvview;
    vsip_vview_f  *a_vview;
    /* Create and fill a_cvview */
    a_vview=vsip_cvrealview_f(a_cvview);
  – This function creates a vector view of the real part of the complex vector view.
  – The real data in the complex view is located in the same memory as the data in the real view.

• When we create a view using this method the library also creates a derived block of the proper type for it to bind.
Derived Block Functionality
What can you do with them?

• A derived block acts just like any other VSIPL block. You can do (almost) anything with them.

• However, because of the way they are created, they must be handled carefully
  – The view created using `realview` (or `imagview`) must be queried for information on attributes. Only the lengths are defined to remain the same as the parent view.
  – One should not go outside the boundaries of the original views data set; in particular one should not try to access `imaginary/real data` of the parent using information gained from the `realview/imagview` function.
Derived Block Destruction

• The derived block is destroyed when the parent complex block is destroyed.
• Any view binding the derived block should be destroyed before the parent complex block is destroyed.

SO THATS IT EXCEPT FOR TWO SLIDES WITH SOME FINAL THOUGHTS

QUESTIONS?
VSIPL Supplies support functions which create blocks and views. Every created block and view uses memory.

/* If you do something like */

vsip_scalar_f beta0 = BETA0;
int i;
vsip_mview_f *A = vsip_mcreate_f(m,n,VSIP_ROW,VSIP_MEM_NONE)
for(i=0; i<m; i++){
    vsip_vcopy_f_f(vsip_vcreate_kaiser_f(n,i*beta0,VSIP_MEM_NONE),
    vsip_mrowview_f(A,i)); /* MEMORY LEAK */
}

/*You will probably be in trouble on some systems */
bind early and destroy late
example of method to support early binding

static void
VU_vmcopycolv_f(
    vsip_vview_f *x, /* input vector to copy out of */
    vsip_vview_f *v, /* work vector on block of matrix */
    vsip_mview_f *A, /* matrix to copy into */
    vsip_index r, /* row index of first element */
    vsip_index c) /* column index of first element */
{
    vsip_mattr_f attr;
    vsip_vattr_f v_attr;
    vsip_mgetattrib_f(A,&attr);
    v_attr.length = vsip_vgetlength_f(x);
    v_attr.offset = attr.offset + r * attr.col_stride + c * attr.row_stride;
    v_attr.stride = attr.col_stride;
    vsip_vputattrib_f(v,&v_attr);
    vsip_vcopy_f_f(x,v);
    return;
}