CVL: A C Vector Library Manual Version 2.1

Guy E. Blelloch   Siddhartha Chatterjee[1]
Jonathan C. Hardwick   Margaret Reid-Miller
Jay Sipelstein   Marco Zagha

June 1995
CMU-CS-93-114 (REVISED)

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

CVL is a library of low-level vector routines callable from C. This library includes a wide variety of vector operations such as elementwise function applications, scans, reduces and permutations. Most CVL routines are defined for segmented and unsegmented vectors. This paper is intended for CVL users and implementors, and assumes familiarity with vector operations and the scan-vector model of parallel computation.

[1]Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. Guy Blelloch was partially supported by a Finmeccanica chair and an NSF Young Investigator award (Grant CCR-9258525).

Keywords: data parallelism, vector library, nested parallelism, segmented vectors, portability, NESL, VCODE, CVL

1 Introduction

CVL is a library of low-level vector routines callable from C. This library presents an abstract model of a vector machine suitable either for stand-alone use or as the backend of a high-level language system. CVL includes a rich set of vector operations including both elementwise computations, and more global operations such as scans, reductions, and permutations. The library also includes segmented versions of these global operations; segmented operations are crucial for the implementation of nested data-parallel languages [1, 7, 4].

The vector machine model provided by CVL is very low-level and was designed so that efficient versions of the library could be developed for a wide variety of parallel architectures. Currently, optimized versions of the library are available for the Connection Machine CM-2 and CM-5, and the Cray Y-MP and Y-MP/C90. We and others are developing versions for the MASPAR MP-1 [10] and for a network of workstations communicating with PVM [11]. There is also a portable serial version of the library. Many of the primitives provided by the library (in particular the scan operations) are much faster than could be easily achieved with Fortran or C implementations on these machines. This allows CVL to be used as an efficient stand-alone library.

CVL was designed to provide a vector abstraction that can be used for the implementation of higher-level data-parallel languages. The authors have designed and implemented the nested data-parallel language NESL [2]. This language compiles into an intermediate language, VCODE [3, 6], which is then interpreted. The interpreter uses CVL to implement the required vector routines. In addition to our work, a research group at Linköping University in Sweden has targeted VCODE and CVL as compiler backends for the Predula parallel language [9], and researchers at the University of North Carolina, Chapel Hill, have targeted CVL for the data-parallel portion of the Proteus [12, 13] language.

This paper is intended for CVL users and implementors. We also assume familiarity with vector operations and the scan-vector model of parallel computation [1]. Because of its intended audience and the low-level nature of the CVL abstractions, this paper has lots of grungy details that would probably not be interesting to a casual reader. We strongly urge the reader to read the NESL implementation paper [4], in order to understand the context in which CVL was designed. A high-level description of the vector operations provided by CVL can be found either in Blelloch's thesis [1] or in the published descriptions of VCODE [3, 6]. Low-level details on how scans and segmented operations can be efficiently implemented on vector machines are also available [8].

2 CVL

CVL is a C library implementing a variety of vector operations on elements of a homogeneous vector memory. This vector memory should be viewed as distinct from the standard C heap or stack, and should only be accessed and modified through the CVL routines.

The CVL memory model distinguishes between *vector length* and *vector size*. Vector length is the number of elements in a vector; *vector size* is the number of "units of the vector memory" occupied by a vector. The vector memory unit is an implementation-dependent abstract quantity and may indicate nothing about the number of bytes or words taken up by a vector. In most implementations it refers to the amount of memory used per processor to hold the vector. An integer vector of length 1000, for example, might only require 4 units of vector memory in one CVL implementation and 500 in another.

CVL provides functions that map the length of a vector to its size; there is one such function for each element type. For example, `siz_foz(len)` returns the size of a vector of integers, given the vector's length. The inverse mapping is not supplied and need not be unique—many vector lengths might map to a single vector size. Both vector length and vector size have C type integer.

This distinction between vector length and size, and their dependence on type, give added flexibility to the CVL implementor: they allow different mappings of data on multiprocessor machines, allow boolean vectors to be packed into bit vectors, and allow an implementation to pad vectors to larger sizes.

CVL defines the C type `vec_p` as an abstract handle for accessing vector memory. For each vector in memory, there is a `vec_p`, and all references to that vector must be made though that `vec_p`. A `vec_p` should be thought of as a pointer into vector memory, but its realization may be more complicated. CVL defines an interface for manipulating and performing the equivalent of pointer arithmetic on a `vec_p`. For example, given a `vec_p int_vec` corresponding to a integer vector of length `len`, the call

```
vec_p new_vec = add_fov(int_vec, siz_foz(len));
```

gets a new `vec_p`, `vec_new`, that corresponds to a block of memory guaranteed not to overlap `int_vec` in vector memory. This is intended to be reminiscent of adding an offset to a pointer. Note that the offset argument to the `vec_p` arithmetic functions must be based on *vector size*, not vector length. Similarly, there is a function `sub_fov(vec_p1, vec_p2)` that corresponds to pointer subtraction and returns the size of the largest possible vector that has `vec_p1` as a handle and does not overlap a vector with handle `vec_p2`.

Vector memory is allocated using the `alo_foz(size)` CVL function. This returns a `vec_p` for a block of vector memory of the requested size. If there is an error, `NULL` is returned. Vector memory is deallocated using `fre_fov(vec_p)`; the argument to the free function must have been the result of a call to `alo_foz`. The current CVL specification allows only a single block of memory to be active during program execution; a program using CVL can only have a single call to `alo_foz`. Multiple calls to `alo_foz` have undefined, implementation-dependent semantics.

CVL provides no memory management facilities other than the allocate/free functions just described and the `vec_p` arithmetic functions. It is the responsibility of programs using CVL to break up the block returned by `alo_foz` into pieces for storing individual vectors. One way of doing this, used in an interpreter for VCODE, is described in [5].

CVL instructions generally take handles to all their source and destination vectors. In addition, there is a length argument for each vector (except when several arguments are required to have the same length), and a segment descriptor argument for each segmented operand. Almost all CVL functions take as a final argument a `vec_p` for a scratch space which the function may use for storage of intermediate vectors (see below for an example of this). Thus, the function `add_wuz()` (which adds corresponding elements of two integer vectors) has the prototype:

```
void add_wuz(vec_p dest, vec_p src1, vec_p src2, int len, vec_p scratch)
```

For each CVL vector function `foo_bar`, there is a function `foo_bar_scratch(len)` (or, for segmented CVL instructions, `foo_bar_scratch(len, num_segs)`) which returns the amount of scratch space needed by that function on vectors of length `len` (with `num_segs` segments). This scratch area is required because, for some instructions on some architectures, the amount of extra space required may depend on vector length or size. For example, an implementation of the pack function may build an internal index vector; this vector would be put in the scratch space.

Let us take a look at a CVL fragment that puts all this together. Suppose that we have two integer vectors, `a` and `b` of length `len`, allocated at the start of vector memory. We wish to put the elementwise sum of these vectors into a new vector, allocated after `b` in memory:

```
vec_p add_vectors(vec_p a, vec_p b, int len) {
    vec_p sum = add_fov(b, siz_foz(len));
    vec_p scratch = add_fov(sum, siz_foz(len));
    add_wuz(sum, a, b, len, scratch);
    return sum;
}
```

First we use the `vec_p` addition function to get handles to two distinct regions of memory after `b` that can contain the sum and the scratch area. Then we use these as arguments to the `add_wuz` function which does all the work. More complete CVL code is given in Appendix B.

This function assumed that there was enough memory available to store the result and scratch vectors. There is no requirement that `add_fov` do any error checking: the result of the call may be an illegal handle. In general, CVL does *no* error checking. It is the responsibility of the calling program to manage vector memory and make sure that enough space is available.

3 CVL Data Types

All CVL instructions operate on homogeneous vector arguments whose elements are of a specific type. For example, there are separate elementwise addition functions for vectors of doubles and for vectors of integers. The allowed element types are `int`, `double`, and `cvl_bool`. Integers and doubles are the standard (machine-dependent) size and precision; booleans might be stored as words, chars, bits, etc., depending on time and space tradeoffs. It is the responsibility of the calling program to use the CVL function that corresponds to the type of the elements of a vector. CVL is low-level and has (at best) minimal error detection; failure to use the correct function may lead to unpredictable results or compile- or run-time failures.

There are two varieties of CVL vectors: segmented and unsegmented [1]. An unsegmented vector is a standard vector: a one-dimensional data structure containing elements of the same type. A segmented vector is a data structure consisting of a group of vectors of the same element type. Segmented vectors have the property that a function applied to them applies to each of its segments independently. For example, a plus reduction of a segmented vector will sum *each* segment:

segmented-plus-reduce [[7 2 9] [8 4 5 6 3]] = [18 26].

The CVL implementation of a segmented vector is an unsegmented vector together with a segment descriptor that describes how to partition the vector into sub-vectors. Most CVL functions on unsegmented vectors have a counterpart for segmented vectors. These functions perform the unsegmented function on each sub-vector and package the result into a new segmented vector.

Segment descriptors are stored in the vector memory along with the vectors. There are two length quantities associated with each segment descriptor: the number of segments and the total number of elements in the vector. For example, the segmented vector

```
s = [[7 2 9] [8 4 5 6 3]]
```

has 2 segments and 8 elements. The segmented vector `s` would be represented as an unsegmented vector of eight elements

```
v = [7 2 9 8 4 5 6 3]
```

and a segment descriptor that describes the partitioning. One concrete representation of a segment descriptor is a vector of segment lengths:

```
d = [3 5],
```

another is a vector of the indices of the initial element of each segment:

```
d' = [0 3].
```

The internal representation of a segment descriptor is implementation-dependent; the only restriction is that vectors and segment descriptors are both handled by objects of C type `vec_p`.

Since not all CVL functions (the elementwise ones, for example) need segmentation information, unsegmented CVL functions (an elementwise operation, for example) may take a segmented vector as an argument, and operate on that vector without regard to segmentation.

All segmented functions must be supplied with segment descriptor and both length quantities as arguments. For example, the prototype for the segmented plus scan operation is:

```
void add_nez (vec_p dest, vec_p src, vec_p segd, int vec_len,
              int seg_count, vec_p scratch)
```

A segment descriptor can only be created with the

```
mke_fos(vec_p out_segd, vec_p lengths, int vec_len, int seg_count,
        vec_p scratch)
```

function: `lengths` is a vector of length `seg_count` containing segment lengths and `vec_len` is the number of elements in the vector; the segment descriptor is returned in `out_segd`. The size of a segment descriptor is determined with the function

```
siz_fos(int vec_len, int seg_count).
```

4 Instruction Classes

This section classifies the instructions supplied by CVL. Full descriptions of each routine and its arguments are given in Appendix A.

CVL functions are named according to a strict naming convention for easy recall. The name of each CVL function has four components. The first is a three letter mnemonic for the root function being applied: e.g., `add`, `sub`, `lsh` (left shift), `sel` (select). The second field is a consonant denoting a modifier for the function that explains how it is used: e.g., `r` (reduce), `s` (scan), `p` (permute). The next field is a vowel indicating the kind of vector to which the function is to be applied: `u` (unsegmented), `e` (segmented), and `o` (none or scalar). The final field is a consonant specifying the type of the elements of the vector: e.g., `b` (boolean), `z` (integer), `s` (segment-descriptor). The first field is separated from the next three by an underscore. Table 1 gives the complete list of modifiers. Mix and match name fields for the function you need.[1]

For each CVL function (with the exception of a few of the facility functions) there are two auxiliary functions with names ending in the extensions `_scratch` and `_inplace`. The scratch functions were discussed earlier and return the amount of scratch space required by a function, for input of a given size. To provide better memory reuse and locality, CVL provides inplace auxiliary functions for most CVL functions. These functions indicate which arguments of a CVL function may be used destructively; in other words, which source vectors can be overwritten to form the destination vector. These inplace functions return the bitwise `or` of the appropriate subset of the values

```
INPLACE_NONE, INPLACE_1, INPLACE_2, ....
```

[1]The consonant-vowel-consonant scheme for suffixes leads to pronounceable function names. You too will soon have "add_wuz" and "len_fos" rolling off your tongue like a pro!

function name = fun_[fun_type][vec_type][element_type]					
fun_type		vec_type		element_type	
`w`	elementwise	`u`	unsegmented	`b`	boolean
`s`	scan	`e`	segmented	`z`	integer
`r`	reduce	`o`	none (scalar)	`d`	double
`p`	permute			`s`	segment-descriptor
`v`	vector-scalar			`v`	vec_p
`f`	facilities				
`l`	library				

Table 1: Fields for CVL function names.

which must be defined in `cvl.h`. These values are of type `unsigned int`. For example, the second source vector of an elementwise integer add can be overwritten only if

`add_wuz_inplace() & INPLACE_2`

returns a nonzero value. The `INPLACE` labels have a direct one-to-one correspondence with the arguments to a function, with numbering beginning at 0 (but since the first argument to a function is always the destination vector, there is no `INPLACE_0`). It is the responsibility of the caller to ensure that a source vector which an inplace function indicates may be overwritten is large enough to hold the destination.

The CVL library functions are divided into a number of different classes:

elementwise The elementwise operations perform a function either to every element of a vector or to corresponding elements of several different vectors of the same length. Examples of elementwise operations include negating each element of a vector and adding the corresponding elements of two vectors.

reduce The reduce functions combine all elements of a vector together under the operation of some other function such as addition or minimum. Examples of reduce functions include summing the elements of a vector and finding the smallest element of a vector. For zero length vectors, the result is the identity element for the operation.

scan The scan instructions generalize the reduce functions by creating a vector whose ith element is the reduction of the first $i-1$ elements of the argument vector. The first element returned is the identity element under the operation.

permute The permute functions rearrange the elements of a vector of values according to a vector of indices. The simple permute is a scatter/send operation, while the back-permute is a gather/get operation.

vector-scalar The vector-scalar operations convert vectors to scalars and back again. The extract function returns a specified element of a vector; the replace function replaces a specified element of a vector with a given value. The distribute function creates a vector of given length, all of whose elements have the same specified value.

facilities The facilities operations perform needed system functions and deal with the representation of vectors and segment descriptors. Facility functions include vector memory allocation and freeing, creation of segment descriptors, and computing the size of a vector whose elements are of some type.

library The library functions consist of a number of routines that could be expressed in terms of other CVL functions, but which, for reasons of efficiency or consistency of the CVL model, were implemented specially.

5 Conclusion

We are interested in any comments or bug reports about CVL and the contents of this technical report. Source code for our implementations of CVL are available via anonymous ftp. For further details, please send email to `blelloch@cs.cmu.edu`. Future changes in CVL are likely; please send email to find out about the latest version of the library.

A CVL Instructions

This appendix gives a description and the interface definition for each CVL instruction. Most functions will be defined on all data types for which the operation makes sense; not all valid type signatures are given here.

As a general rule, the first argument to a CVL functions is a `vec_p` corresponding to the destination argument. The contents of this vector will be overwritten with the result of the operation. Each non-facility function takes as its final argument a `vec_p` that refers to a scratch space. The required minimum size of this scratch space is returned by `fun_scratch(len)`, where `len` is the length of the vector arguments to the function `fun`.

A.1 Facilities

1. The size functions return the number of vector memory units required by a vector of given type and length. In the case of segment descriptors, both the length of the segment descriptor and the length of the unsegmented vector must be supplied.
   ```
   int siz_foz (int length)
   int siz_fob (int length)
   int siz_fod (int length)
   int siz_fos (int vec_len, int seg_count)
   ```
   Note: Vectors and segments of a vector may have zero length. All CVL functions must work probably when the length argument is zero.

2. Segment descriptors are created from a vector of segment sizes using the `mke_fov` function. The inverse function is `len_fos`. There are scratch functions corresponding to both of these functions.
   ```
   void mke_fov (vec_p segd, vec_p lengths, int vec_len, int seg_count,
                 vec_p scratch)
   void len_fos (vec_p lengths, vec_p segd, int vec_len, int seg_count,
                 vec_p scratch)

   void mke_fov_scratch (int vec_len, int seg_count)
   void len_fos_scratch (int vec_len, int seg_count)
   ```

Here, `lengths` is an integer vector containing segment lengths, `segd` is a segment descriptor, `vec_len` is the length of the unsegmented vector (the sum of the values in `lengths`), and `seg_count` is the number of segments (number of elements in `lengths`).

3. CVL provides memory allocation and freeing instructions. As explained earlier, these functions may only be invoked once per program execution.

```
vec_p alo_foz (int size)
void fre_fov (vec_p vec)
```

   `alo_foz` tries to allocate `size` units of vector memory. The allocation function returns `(vec_p)NULL` if the allocation fails.

4. CVL provides a move vector instruction:

```
void mov_fov(vec_p dest, vec_p src, int size, vec_p scratch)
void mov_fov_scratch(int size)
```

   This instruction moves a block of vector memory from one location to another. The implementation of `mov_fov` must allow situations in which the source and destination vectors overlap.

5. CVL provides the following arithmetic functions on the `vec_p` type:

```
vec_p add_fov(vec_p v, int a)
int sub_fov(vec_p v1, vec_p v2)
cvl_bool eql_fov(vec_p v1, vec_p v2)
int cmp_fov(vec_p v1, vec_p v2)
```

   The add function takes a `vec_p v` and an integer `a` and returns a new `vec_p` corresponding to a region of vector memory that is guaranteed not to overlap a vector with `vec_p v` and size `a`. The subtract function takes two `vec_p` arguments and returns the size of the largest vector that may be stored at the first argument without overlapping a vector stored at the second. The equality function returns `1` if the two arguments refer to the same region of vector memory[2] and returns `0` otherwise. The compare function compares two `vec_p`s and returns a positive value if `v1` corresponds to a region of vector memory after that of `v2`, returns `0` if the `vec_p`s are equal (in the sense above), and returns a negative value otherwise.

   Implementation note: The actual pointer to vector memory (either the `vec_p` or one of its components) must be "maximally aligned." In other words, given a `vec_p`, a vector of any type must be storable in the corresponding memory block. Without this property, it might not be possible, for example, to move an integer vector into the location once held by a boolean vector. The implementations of `alo_foz` and `add_fov` must guarantee this property of the `vec_p` returned.

6. CVL includes two timing functions for benchmarking purposes. The get time function, `tgt_fos`, stores a time stamp in a structure of type `cvl_timer_t`. The time difference function, `tdf_fos`, takes two such structures and returns a double precision count of the number of seconds between the two events.

```
void tgt_fos (cvl_timer_t *time)
double tdf_fos (cvl_timer_t *t1, cvl_timer_t *t2)
```

[2]This does not necessarily mean that `v1 = v2`. One possible implementation of a `vec_p` is as a pointer to a structure, within which is a pointer into vector memory; `eql_fov` should return `1` if the pointer fields of the structure are equal.

7. To provide a convenient interface between normal C arrays and the CVL vectors, CVL provides translation functions that convert between the two:

```
void v2c_fuz(int *dest, vec_p src, int len, vec_p scratch)
void c2v_fuz(vec_p dest, int *src, int len, vec_p scratch)
```

   `v2c` writes the contents of a vector into a unit-stride C array; `c2v` performs the inverse operation. These instructions exist for boolean, integer and double precision types. They do not exist for segment descriptors; these must first be converted to/from length vectors.

8. CVL allows the user to seed the random number generator:

```
void rnd_foz(int seed)
```

A.2 Elementwise functions

All source and destination arguments to elementwise functions must be of the same length, which is supplied as a parameter. The same elementwise function works on either segmented or unsegmented vectors, with no segment descriptor required.

The logical functions `and`, `ior`, and `not` are defined on both integers and booleans. They act as bitwise operators on integers and as logical operators on booleans.

Table 2 gives a list of all the elementwise functions provided by CVL.

1. The unary elementwise functions include the type conversion routines (`int`, `dou`, and `boo`), various arithmetic functions (`flr`, `cei`, `trn`, `rou`, `log`, `exp`, `sqt`), negation, and copy. `cpy_wus` takes both segment count and vector length arguments.

```
void not_wuz (vec_p dest, vec_p src, int len, vec_p scratch)
void cpy_wuz (vec_p dest, vec_p src, int len, vec_p scratch)
void cpy_wus (vec_p dest, vec_p src, int vec_len, int seg_count,
              vec_p scratch)
```

2. The binary elementwise functions include all the standard arithmetic, logical, and comparison functions.

```
void add_wuz (vec_p dest, vec_p src1, vec_p src2, int len, vec_p scratch)
```

3. The only ternary function is `select`. The select operations take three vector source arguments: a boolean vector and two vectors of the same type.

```
void sel_wub (vec_p dest, vec_p bool_vec, vec_p s1, vec_p s2, int len,
              vec_p scratch)
```

   The result vector has value:

```
dest[i] = (bool_vec[i] ? s1[i] : s2[i])
```

A.3 Scan and Reduce

The reduce functions combine all the elements in the source vector, using the identity element as the initial combining element. The unsegmented reduce returns this value; the segmented version combines each segment independently and returns all these results in the destination vector argument.

```
int add_ruz (vec_p src, int len, vec_p scratch)
void add_rez (vec_p dest, vec_p src, vec_p segd, int vec_len,
              int seg_count, vec_p scratch)
```

function		element type				C equivalent
mnemonic	function	b	z	d	s	
add	addition		x	x		$d = a + b$
sub	subtraction		x	x		$d = a - b$
eql	equal	x	x	x		$d = a == b$
cpy	copy	x	x	x	x	$d = a$
min	minimum		x	x		$d = \min(a, b)$
max	maximum		x	x		$d = \max(a, b)$
mul	multiplication		x	x		$d = a * b$
div	division		x	x		$d = a/b$
mod	modulus		x			$d = a\%b$
rnd	random		x			$d = \mathrm{random}()\%a$
lsh	left shift		x			$d = a << b$
rsh	right shift		x			$d = a >> b$
grt	greater than		x	x		$d = a > b$
les	less than		x	x		$d = a < b$
geq	grt than or eq		x	x		$d = a >= b$
leq	less than or eq		x	x		$d = a <= b$
neq	not equal	x	x	x		$d = a\ != b$
not	not	x	x			$d = !a$, $d = \tilde{}a$
ior	inclusive or	x	x			$d = a\vert\vert b$, $d = a\vert b$
and	and	x	x			$d = a\&\&b$, $d = a\&b$
xor	exclusive or	x	x			$d = a\hat{}b$
log	natural log			x		$d = \log(a)$
exp	exp			x		$d = \exp(a)$
sqt	sqrt			x		$d = \mathrm{sqrt}(a)$
sin	sine			x		$d = \sin(a)$
cos	cosine			x		$d = \cos(a)$
tan	tangent			x		$d = \tan(a)$
asn	arcsin			x		$d = \mathrm{asin}(a)$
acs	arccosine			x		$d = \mathrm{acos}(a)$
atn	arctangent			x		$d = \mathrm{atan}(a)$
snh	sinh			x		$d = \sinh(a)$
csh	cosh			x		$d = \cosh(a)$
tnh	tanh			x		$d = \tanh(a)$
sel	select	x	x	x		$d = c?a : b$
int	integer	x		x		$d = (\mathrm{int})(a)$
dbl	double		x			$d = (\mathrm{double})(a)$
boo	bool		x			$d = !!(a)$
flr	floor			x		$d = (\mathrm{int})\mathrm{floor}(a)$
cei	ceiling			x		$d = (\mathrm{int})\mathrm{ceil}(a)$
trn	truncate			x		$d = (\mathrm{int})(a)$
rou	round			x		$d = (\mathrm{int})\mathrm{rint}(a)$

Table 2: List of elementwise CVL functions. In each case, d refers to the destination vector, and a, b, and c are the argument vectors. For `ior`, `and`, and `not` the two versions are for booleans and integers, respectively. The semantics of each elementwise function is equivalent to the those defined by ANSI C for the C version in this table.

The scan functions compute a running reduction of the source vector and return this result in a destination argument. As with the reduce functions, the identity element is the initial combiner.

```
void add_suz (vec_p dest, vec_p src, int len, vec_p scratch)
void add_sez (vec_p dest, vec_p src, vec_p segd, int vec_len,
              int seg_count, vec_p scratch)
```

Scan and reduce functions are provided for addition, multiplication, or, and, xor, maximum, and minimum.

A.4 Permute

The permute functions take source and index vectors and write the result into a destination vector. Simple and backward permutes, both segmented and unsegmented, are part of the basic library. Also provided are backward and forward flag permutes, and forward default and default-flag permutes. For those permute operations with differing source and destination lengths, two sets of length or segment descriptor arguments must be provided. Any constraints on vector arguments given below must also be satisfied by each segment in the segmented version of the operation.

1. Simple (or forward) permute puts an element into the location in the destination given by the index vector: `dest[index[i]] = src[i]`. The elements of index must be a proper permutation (all indices present, and no repetitions) of the allowable range. All vector arguments must be the same length.

   ```
   void smp_puz (vec_p dest, vec_p src, vec_p index, int len,
                 vec_p scratch)
   void smp_pez (vec_p dest, vec_p src, vec_p index, vec_p segd,
                 int vec_len, int seg_count, vec_p scratch)
   ```

2. Backward permute gets an element from the indexed location:

   ```
   dest[i] = src[index[i]]
   ```

   The destination and index vectors must be of the same length, which can differ from that of the source.

   ```
   void bck_puz (vec_p dest, vec_p src, vec_p index,
                 int src_len, int dest_len, vec_p scratch)
   void bck_pez (vec_p dest, vec_p src, vec_p index,
                 vec_p src_segd, int src_vec_len, int src_seg_count,
                 vec_p dest_segd, int dest_vec_len, int dest_seg_count,
                 vec_p scratch)
   ```

3. The default permute operation is a simple permute that relaxes the restriction that the source and destination must be of the same length. Any unassigned position in the destination gets its value from the corresponding position of a default vector.

   ```
   void dpe_puz (vec_p dest, vec_p src, vec_p index, vec_p default,
                 int src_len, int dest_len, vec_p scratch)
   void dpe_pez (vec_p dest, vec_p src, vec_p index, vec_p default,
                 vec_p src_segd, int src_vec_len, int src_seg_count,
                 vec_p dest_segd, int dest_vec_len, int dest_seg_count,
                 vec_p scratch)
   ```

4. The flag permute is a combination of the select and permute operations: a flag vector determines whether or not each element is moved. The set of indices and values not masked are the arguments to the appropriate permute operation. For example, in the simple flag permute (`fpm`):

```
if (flags[i]) { dest[index[i]] = src[i]; }
```

In the backwards flag permute (`bfp`), unfilled positions in the destination vector are set to 0 or false. There are simple (`fpm`), backward (`bfp`), and default (`dfp`) flag permute operations.

```
void fpm_puz (vec_p dest, vec_p src, vec_p index, vec_p flags,
              int src_len, int dest_len, vec_p scratch)
void fpm_pez (vec_p dest, vec_p src, vec_p index, vec_p flags,
              vec_p src_segd, int src_vec_len, int src_seg_count,
              vec_p dest_segd, int dest_vec_len, int dest_seg_count,
              vec_p scratch)
void bfp_puz (vec_p dest, vec_p src, vec_p index, vec_p flags,
              int src_len, int dest_len, vec_p scratch)
void bfp_pez (vec_p dest, vec_p src, vec_p index, vec_p flags,
              vec_p src_segd, int src_vec_len, int src_seg_count,
              vec_p dest_segd, int dest_vec_len, int dest_seg_count,
              vec_p scratch)
void dfp_puz (vec_p dest, vec_p src, vec_p index, vec_p flags,
              vec_p default, int src_len, int dest_len,
              vec_p scratch)
void dfp_pez (vec_p dest, vec_p src, vec_p index, vec_p flags,
              vec_p default,
              vec_p src_segd, int src_vec_len, int src_seg_count,
              vec_p dest_segd, int dest_vec_len, int dest_seg_count,
              vec_p scratch)
```

A.5 Vector-scalar

Extract, replace, and distribute functions exist in both segmented and unsegmented versions. Extract is an indexing function that removes the requested element from a vector and returns a scalar (unsegmented version) or an unsegmented vector (segmented version). Replace is the inverse operation. It is a destructive operation, modifying the contents of the destination vector. The output of distribute is a vector of given length, all of whose elements have a given value.

```
void dis_vuz (vec_p dest, int value, int len, vec_p scratch)
void dis_vez (vec_p dest, vec_p value, vec_p dest_segd,
              int dest_vec_len, int dest_seg_count, vec_p scratch)

int ext_vuz (vec_p src, int index, int len, vec_p scratch)
void ext_vez (vec_p dest, vec_p src, vec_p index, vec_p src_segd,
              int src_vec_len, int src_seg_count, vec_p scratch)

void rep_vuz (vec_p src, int index, int value, int len, vec_p scratch)
void rep_vez (vec_p dest, vec_p src, vec_p value, vec_p segd,
              int vec_len, int seg_len, vec_p scratch)
```

A.6 Library

CVL also contains a set of library functions. These functions could be implemented in terms of the other primitives, but for efficiency, may be implemented directly.

1. The `pack` primitive has been divided into two parts, `pk1` and `pk2`.[3] The first part of pack, `pk1_lev` (this function is independent of element type), takes a flag vector, and returns a value (vector) giving the number of true elements (in each segment).

   ```
   int pk1_luv (vec_p flags, int vec_len, vec_p scratch)
   void pk1_lev (vec_p dest, vec_p flags, vec_p segd,
                 int vec_len, int seg_count, vec_p scratch)
   ```

   If `flags = [T F] [F F T T]`, then after calling `pk1_lev`, the value of `dest` is `[1 2]`. The `pk2_le*` functions fill a destination vector with elements corresponding to the true elements of the flag vector.

   ```
   void pk2_luz (vec_p dest, vec_p src, vec_p flags,
                 int src_len, int dest_len, vec_p scratch)
   void pk2_lez (vec_p dest, vec_p src, vec_p flags,
                 vec_p src_segd, int src_vec_len, int src_seg_count,
                 vec_p dest_segd, int dest_vec_len, int dest_seg_count,
                 vec_p scratch)
   ```

   It is the responsibility of the calling function to use the result of the first pack step to allocate the destination vector and destination segment descriptor.

2. The index function generates a vector of integer values, starting from a given initial value, with given stride, and generating a given number of values. CVL provides both segmented and unsegmented index functions.

   ```
   void ind_luz (vec_p dest, int init, int stride, int count, vec_p scratch)
   void ind_lez (vec_p dest, vec_p init, vec_p stride,
                 vec_p dest_segd, int dest_vec_len, int dest_seg_count,
                 vec_p scratch)
   ```

   The arguments to the index functions are similar to those of the distribute functions.

3. CVL provides rank functions for sorting vectors of doubles or integers. Rank returns a permutation indicating how the source elements are ordered. For segmented rank, the result restarts at 0 for each segment, i.e. each segment contains a permutation.

   ```
   rku_luz(vec_p rank, vec_p src, int len, vec_p scratch)
   rku_lez(vec_p rank, vec_p src, vec_p segd, int vec_len,
           int seg_count, vec_p scratch)
   ```

   The `rku` functions perform an upward rank (lowest element gets rank 0); the `rkd` functions perform a downward rank (highest element gets rank 0). The source is an integer or double vector. The result is always an integer vector. The rank is stable.

[3] This was done in order to facilitate memory management. The first part of the operation provides information required to obtain a `vec_p` for the results of the second part.

B Example: Dot Product

We give as an example of C code that uses CVL to calculate the dot product of two vectors. The code is shown in Figure 1. Two vectors are generated by function calls inside the dot product function.[4] We could have eliminated some of the scratch checking by precomputing how much memory is needed (since this only depends on vector length and which CVL operations are performed) before doing the allocation.

C Changes

This section lists some of the changes between this and older versions of the CVL documentation.

C.1 Changes in 2.0

change The general scan operation is no longer directly supported, and the `***_n**` instructions have all been renamed to `***_s**`.

new Scan and reduce instructions for xor and multiplication have been added.

new The `mov_fov` instruction has been added and the requirement that `cpy_wu*` handle overlapping vectors has been removed.

new The `v2c` and `c2v` instructions have been added.

new The index and rank library instructions have been added.

change The `inplace` mechanism has been changed.

change The `or_***` function has been renamed to `ior_***`; all functions now have a three letter root.

change The `pck_luz` function has been broken up into `pk1_luv` and `pk2_luz` components.

change The count argument to `ind_lez` has been removed.

C.2 Changes in 2.1

new The `rnd_foz` function (seeding the random number generator) has been added.

References

[1] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[2] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[3] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings Frontiers of Massively Parallel Computation*, pages 471–480, October 1990.

[4] These vectors would typically be passed as arguments, but we wanted to show how vector memory is allocated.

```
/* Complete CVL example: dot product of double precision vectors.
 *
 * This routine takes a vector length as input. It allocates vector memory, calls two (dummy) functions
 * to generate the operands for a dot product, computes the dot product, frees memory, performs (some)
 * error checking, and returns the value of the dot product.  This has all the gory details!
 *
 * We assume that the dummy functions take three arguments: a vector length, a vec_p in which to
 * store the result, and some scratch space.
 */
#include <cvl.h>                            /* This contains all CVL declarations */

double dotp(int len)
{
  vec_p vmem;                               /* vector memory: result of alo_foz */
  int vsize = siz_fod(len);                 /* size of a double vector of length len */
  int vmem_size = 10 * vsize;               /* amount of vector memory to allocate */
  vec_p a,b;                                /* two vector arguments */
  vec_p product;                            /* elementwise product of a and b */
  vec_p scratch;                            /* scratch storage for CVL */
  int scratch_needed;                       /* ammount of scratch needed by CVL function */
  double result;                            /* final result */

  vmem = alo_foz(vmem_size);                /* Allocate memory for vectors */
  if ( vmem == (vec_p) NULL ) {             /* check for failure */
    fprintf(stderr, "dotp: cvl could not allocate memory");
    exit(1);
  }
  a = vmem;                                 /* store a at beginning */
  b = add_fov(a, vsize);                    /* store b right after a */
  product = add_fov(b, vsize);              /* result of multiply goes here */
  scratch = add_fov(product, vsize);        /* rest is for scratch */

  get_a(len, a, scratch);                   /* create first vector */
  get_b(len, b, scratch);                   /* create second vector */

  /* check if there is enough scratch space for the multiply operation */
  scratch_needed = mul_fod_scratch(len);
  check_scratch(vmem, scratch, vmem_size, scratch_needed);
  mul_wud(product, a, b, len, scratch);     /* elementwise multiply */

  /* check for enough scratch to do add reduce */
  scratch_needed = add_rud_scratch(len);
  check_scratch(vmem, scratch, vmem_size, scratch_needed);
  result = add_ruz(tmp, len, scratch);      /* add reduce */

  fre_fov(vmem);                            /* free up memory */
  return result;
}

/* check_scratch is a useful utility function for verifying that enough scratch space has been reserved.
 * It assumes that the scratch vector is at the end of vector memory.
 * exit() is called if an error is encountered.
 */
void check_scratch(vec_p vmem, vec_p scratch, int vmem_size, int scratch_needed)
{
  if ((vmem_size - sub_fov(scratch, vmem)) < scratch_needed) {
    fprintf(stderr, "Not enough scratch space.");
    exit(1);
  }
}
```

Figure 1: CVL code for dot product. This example demonstrates memory allocation, scratch space checking and basic CVL function calls.

[4] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, May 1993.

[5] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. Technical Report CMU-CS-93-112, School of Computer Science, Carnegie Mellon University, February 1993.

[6] Guy E. Blelloch, Siddhartha Chatterjee, Fritz Knabe, Jay Sipelstein, and Marco Zagha. VCODE reference manual (version 1.1). Technical Report CMU-CS-90-146, School of Computer Science, Carnegie Mellon University, July 1990.

[7] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, February 1990.

[8] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, November 1990.

[9] Johan Fagerström, Peter Fritzson, Johan Ringström, and Mikael Pettersson. A data-parallel language and its compilation to a formally defined intermediate language. In *Proceedings Fourth International Conference on Computing and Information*, May 1992.

[10] Rickard E. Faith, Doug L. Hoffman, and David G. Stahl. UnCvl: The University of North Carolina C Vector Library. Version 1.1, May 1993.

[11] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3.0 User's Guide and Refernce Manual*, February 1993.

[12] Peter H. Mills, Lars S. Nyland, Jan F. Prins, John H. Reif, and Robert A. Wagner. Prototyping parallel and distributed programs in Proteus. Technical Report UNC-CH TR90-041, Computer Science Department, University of North Carolina, 1990.

[13] Jan F. Prins and Daniel W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, May 1993.