UnCvL:
The University of North Carolina
C Vector Library

Rickard E. Faith   Doug L. Hoffman   David G. Stahl

TR93-063
18 May 1993

Department of Computer Science
University of North Carolina at Chapel Hill
Campus Box 3175, Sitterson Hall
Chapel Hill, North Carolina 27599-3175

1 Introduction

This paper describes our efforts in implementing a version of CVL (C Vector Library) for the MASPAR MP-1 SIMD computer. CVL is a library of rudimentary vector routines, callable from C, as described by G. Blelloch, et. al. [2, 6]. UNCVL is an implementation of the CVL library routines written in `mpl`, MASPAR's parallel version of C. The main motivation for implementing CVL on the MP-1 is to provide support for NESL, a high level, portable parallel programming interface [3]. NESL compiles to an intermediate language called VCODE which, in turn, is implemented using CVL. Our CVL implementation on the MP-1 will also be used in the parallel execution of the PROTEUS language at the University of North Carolina [5].

Implemented functions include 195 elementwise operations, 180 scan and reduce operations, 108 permute operations, and 102 miscellaneous and support operations. Of the undocumented functions that we were able to elucidate, only 9 pack, 9 shuffle, and 18 split function have not yet been implemented.[1]

The current version of UNCVL supports execution of NESL and PROTEUS programs on the MASPAR MP-1.

2 Problem Description

The obvious goals of this project were to correctly implement all of the CVL functions and to have the coded routines run as efficiently as possible on the MP-1 architecture. Working from the rather vague descriptions of the CVL functions in [2, 6], and checking the results of our work against test programs known to run on the sequential version of CVL, we have written code for all the documented functions and for many of the undocumented functions.[2] The question of efficiency is a bit more complicated since the efficiency of the library routines is greatly impacted by the decomposition of large vectors.

[1] During our implementation, we noticed several deficiencies in the obtainable CVL documentation. Most notably, many functions appear in other CVL implementations, but are not documented. Most other functions are not documented with sufficient detail and precision. Often questions regarding the exact functionality of a required routine need to be resolved by reading source code for other CVL implementations. Of special concern is the "scan" operation: although typical "scan" operations in the literature (and in CVL) do not include the current element in the scan, the builtin MASPAR library scan does. Significant time was lost because this distinction was not well understood.

[2] The descriptions of the undocumented functions were obtained by reading code for other implementations of the CVL library, and by examining how these functions were used in the `test-prims` test program.

There are two basic approaches to large vector organization on the MP-1: hierarchical decomposition and "cut and stack" decomposition. The choice of decomposition method determines the way in which vector elements are distributed among the MasPar's processing elements. The element distribution, in turn, greatly impacts the communication cost for the scan and reduce vector operations. We chose the "cut and stack" decomposition after reviewing an earlier implementation effort by David Becker and Steve Tell [1] that used the hierarchical model.

The cut and stack decomposition simplifies index calculation in the permutation operations and tends to utilize the interconnection network more effectively. Its main drawback is increased cost of scan and segmented reduction operations.

3 Algorithm Description

The functions in Cvl can be broken down into four major categories: vector allocation and maintenance operations, element-wise operations, scan and reduce operations, and permute operations. The allocation and maintenance operations are used to dynamically allocate vector storage space and to perform space management type operations. The element-wise operations include all element to element mathematical and logical operations. The scan and reduce operations and the permute operations are much more complex and will be described in greater detail below.

4 Implementation Details

Since the Cvl `alo_foz` routine does not allow type specifications, all vector elements must be the same size, regardless of type. The data elements of a vector may be of type `int`, `double`, or `bool`. This implies that, on the the MP-1, the actual storage space for a single vector element must contain 64 bits, the size of the C type `double`. Because of the fixed size of each element and the "cut and stack" decomposition of the vectors, we decided to allocate vector elements with a granularity of the number of MasPar processing elements, using eight bytes per processing element per vector element. Each one of these allocation units is referred to by a pointer number, or handle, and the units themselves are referred to as "handle units."[3]

Any vector will occupy at least one element on each of the MP-1's processing elements. The default memory size on the MP-1 at UNC-CH has room for about 768 handle units, each using 8 bytes per processing element and providing 8192 vec-

[3]The "handle unit" nomenclature is ours.

tor elements.[4] Vectors that are larger than the number of processing elements are allocated multiple consecutive handle units to accommodate their data elements.

4.1 Elementwise Operations

This memory organization makes the element-wise operations extremely easy to implement. The memory allocation functions defined by CVL are also quite simple due to their rudimentary nature.

4.1.1 Future Work

The elementwise operations perform quite well and are, in general, faster than both the CM-2 and the Cray C90. Possible improvements include loop unrolling and optimization of the boolean AND functions. Some functions are capable of running with identical source and destination vectors—this should be noted in the `inplace` functions.

4.2 Scan and Reduce

The MasPar library defines a scan operation which includes the current element in the scan result (an inclusive scan). The CVL definition for a scan does not include the current element (an exclusive scan). Therefore, UnCVL scan and segmented reduce operations are implemented using the MasPar scan function followed by a right-shift. Although this is not as efficient as possible, this implementation route was selected to simplify debugging and verification.[5]

Non-segmented reductions were implemented in a much more efficient manner: elements were reduced within the PE, and then a single machine-size reduction was performed. This is possible because all CVL reductions involve commutative operations.

[4] We allow 2048 bytes for stack space and non-vector storage per PE. Empirical evidence suggests that this is sufficient. When `mplimit` is used to increase the PE memory to 64 kilobytes, over 7900 handle units are available.

[5] A more efficient scan can be implemented for functions which have an inverse by simply computing the inclusive scan and the "subtracting" the current value. However, many functions, such as AND and MAX do not possess inverses. Implementing a single "macro" that was generic across all functions greatly simplified debugging. Later, an optimized scan for the ADD operation can be implemented.

4.2.1 Future Work

A custom exclusive-scan routine (possibly modeled on the fast scan by Injong Rhee of UNC-CH) should be implemented. This scan would implement the CVL scan definition directly, and would be expected to increase the performance of all scan operations, segmented reduce operations, and the `dis_vex` function—all of which are currently based on the MasPar library scan functions.

Ultimately, a pipelined scan operation should be implemented to improve performance under the cut and stack model.

4.3 Permute

In order to use the CVL vectors to solve most non-trivial problems a mechanism to re-order (permute) vectors is required. CVL defines three distinct types of permutes: simple, default, and backward.

A CVL simple permute uses an index vector to indicate which destination element in the destination vector should receive a given element from a source vector. The pseudo-code for this operation is:

forall $0 \leq i < n$
  destination[index[i]] = source[i]

CVL requires that the source, index, and destination vectors all be of the same length.

The CVL default permute is a simple permute preceded by a copy to the destination vector of the values in a default vector. The pseudo-code for this operation is:

forall $0 \leq i < n$
  destination[i] = default[i]
forall $0 \leq i < n$
  destination[index[i]] = source[i]

Finally, in a CVL backward permute, the index indicates which source element should be copied to a given destination. The pseudo-code for the backward permute is:

forall $0 \leq i < n$
  destination[i] = source[index[i]]

We investigated two mechanisms on the MasPar for implementing permutes: using the router and using the parallel I/O memory. Since initial performance tests showed the router to be faster by an order of magnitude, we implemented the simple and backward permutes with the MasPar's router using the `pp_send` and `pp_fetch` functions. Our use of the cut and stack representation made it simple to convert an index value to a processor and handle unit offset since the processor holding index[i] is

$$\text{index}[i] \bmod p$$

where p in the number of processors. Note that this is equivalent to

$$\text{index}[i] \wedge (p-1)$$

if p is a power of 2. The handle offset is

$$\left\lceil \frac{\text{index}[i]}{p} \right\rceil$$

We implemented the segmented versions of these routines using a scratch vector in which we add the start of the segment position to the index value of the permute. We then use this modified index vector to perform the permute.

4.3.1 Future Work

The permute operations are the most expensive functions in terms of time, and an efficient permute is essential to allowing Nesl and Cvl to provide reasonable performance. Due to the time constraints of the project we implemented a naïve, straightforward version of the permute functions. The segmented versions of the permute functions can be improved by performing the calculation of the permute index in-line rather than using `dis_vez` and `add_wuz`. We were encouraged to see that even with a naïve implementation, our routines appeared to be competitive with the optimized CM-2 implementation of Cvl by Blelloch, et. al.

Some of the permute operations are dependent on the current implementation of the Cvl scan operations. When scan is improved, some permutes will be faster.

On a more theoretical level, we could calculate from the MasPar clock cycle, routing architecture and PE instruction set, the fastest possible permute on the MP-1. This would give us a clear benchmark to aim for in terms of optimizing the permute code. We did not have time, however, to further explore these calculations.

5 Performance Evaluation

The `test-prims` program was used to generate performance timings[6] For this evaluation, the UNCVL library was compiled with `mpl` using full optimization (`-Omax`) and was executed on a 8192 processor MP-1 using vector lengths of 1048576.

Tables 1 and 2 compares the current MASPAR implementation with seasoned, optimized CM-2 and Cray versions of CVL.[7] Speedup was calculated using the following formula [4, p. 10]:

$$\text{Speedup} = \frac{\text{Execution time}_{\text{MASPAR}}}{\text{Execution time}_{\text{other machine}}}$$

[6]The following command was used to generate the performance data: `test-prims -t6 -p4 -l1048576` (clock resolution was computed as 133 μseconds per tick).

[7]Timings for the CM-2 also used vecotors of length 1048576. The Cray C90 used vectors of length 100000. The number of processors on the two test machines is unknown. The data for the CM-2 are from the file `cm2.may-11-92`, and the data for the Cray are from the file `cray-feb-8-93-C90-VL128`.

Primitive	Milliseconds per Element			$L^{\dagger}_{1/2}$			Speedup	
	MP-1	CM-2	Cray	MP-1	CM-2‡	Cray	CM-2	Cray
cpy_wuz	0.0015	0.0035	0.0024	17621	1547	828	2.33	1.60
boo_wuz	0.0015	0.0028	0.0045	17916	762	458	1.87	3.00
add_wuz	0.0025	0.0055	0.0030	23328	905	663	2.20	1.20
mul_wuz	0.0046	0.0056	0.0072	19986	1081	286	1.22	1.57
div_wuz	0.0064	0.1432	0.0146	16818	59	148	22.37	2.28
mod_wuz	0.0068	0.1343	0.0226	16089	53	107	19.75	3.32
max_wuz	0.0037	0.0082	0.0047	23132	671	456	2.22	1.27
les_wuz	0.0026	0.0042	0.0046	23416	1310	457	1.62	1.77
sel_wuz	0.0029	0.0077	0.0049	15978	1118	427	2.66	1.69
not_wub	0.0018	0.0007	0.0024	27908	2709	791	0.39	1.33
and_wub	0.0025	0.0007	0.0030	23219	-3843	842	0.28	1.20
int_wub	0.0015	0.0028	0.0024	17568	2271	860	1.87	1.60
dis_vuz	0.0008	0.0015	0.0023	39281	3798	849	1.88	2.88
add_suz	0.0959	0.0246	0.0053	8578	29612	464	0.26	0.06
max_suz	0.0990	0.0339	0.0101	8627	28351	274	0.34	0.10
add_ruz	0.0013	0.0184	0.0023	158417	39337	855	14.15	1.77
max_ruz	0.0016	0.0250	0.0046	68384	37650	476	15.63	2.88
smp_puz	0.0654	0.1377	0.0033	90656	-7058	596	2.11	0.05
bck_puz	0.0631	0.2407	0.0034	91109	-5134	583	3.81	0.05
ind_luz	0.0016	0.0070	0.0027	20950	-12045	730	4.38	1.69
add_wud	0.0062	0.0096	0.0030	15888	537	667	1.55	0.48
mul_wud	0.0098	0.0096	0.0029	12953	546	685	0.98	0.30
div_wud	0.0141	0.0195	0.0103	11290	269	206	1.38	0.73
max_wud	0.0037	0.0113	0.0047	17863	460	460	3.05	1.27
les_wud	0.0048	0.0063	0.0048	19736	804	443	1.31	1.00
sel_wud	0.0044	0.0135	0.0049	16331	647	420	3.07	1.11
add_sud	0.1849	0.0439	0.0054	8332	77901	450	0.24	0.03
max_sud	0.1788	0.0808	0.0102	8443	20230	273	0.45	0.06
add_rud	0.0036	0.0073	0.0023	134062	42631	851	2.03	0.64
max_rud	0.0029	0.0349	0.0046	66191	39886	474	12.03	1.59

†$L_{1/2}$ is the vector length at which $\frac{1}{2}$ of the maximum performace is obtained

‡Some of these values are clearly erroneous; others are difficult to believe.

Table 1: CVL Execution Times for Unsegmented Operations

Primitive	Milliseconds per Element			Speedup	
	MP-1	CM-2	Cray	Over CM-2	Over Cray
add_sez	0.1100	0.0440	0.0090	0.40	0.08
max_sez	0.1140	0.0540	0.0170	0.47	0.15
add_nez	0.1060	0.0300	0.0070	0.28	0.07
max_nez	0.1090	0.0360	0.0140	0.33	0.13
add_rez	0.0960	0.0270	0.0040	0.28	0.04
max_rez	0.0990	0.0340	0.0070	0.34	0.07
add_sed	0.2060	0.0910	0.0090	0.44	0.04
max_sed	0.2030	0.1910	0.0170	0.94	0.08
add_red	0.1830	0.0510	0.0040	0.28	0.02
max_red	0.1770	0.0790	0.0070	0.45	0.04
dis_vez	0.1100	0.0450	0.0040	0.41	0.04

Table 2: CVL Execution Times for Segmented Operations

References

[1] D. Becker and S. Tell. C vector library (CiViL) for MasPar. University of North Carolina at Chapel Hill, Department of Computer Science, 1990.

[2] G. Blelloch, S. Chatterjee, J. Sipelstein, and M. Zagha. CVL: A C vector library. Carnegie Mellon University, School of Computer Science, 1991.

[3] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, Carnegie Mellon University, School of Computer Science, 1992.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. San Mateo, California: Morgan Kaufmann Publishers, 1990.

[5] 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*, May 1993.

[6] J. Sipelstein, G. Blelloch, S. Chatterjee, J. Hardwick, and M. Zagha. CVL: C vector library manual, version 2. Technical Report CMU-CS-93-114, Carnegie Mellon University, School of Computer Science, 1993.