Vlist: A Vectorized List
University of Florida CIS TR #92-007*

Theodore Johnson
Dept. of CIS, University of Florida
Gainesville, Fl 32611-2024
ted@cis.ufl.edu

April 3, 1992

Abstract

Many current supercomputers use vector or SIMD processors to exploit parallelism inexpensively and effectively. Many fast and efficient kernels have been written for vector and matrix computations, but little work has been done to vectorize data structures. As a result, maintaining the data structures can be the most time consuming part of a computation. In this paper, we introduce a vectorized list, the *Vlist*. The Vlist maintains same-length component lists to allow vectorized search operations. We show how inserts and deletes can be performed on the Vlist in constant time, and in a manner that preserves the same-length property of the component lists. Finally, we discuss applications to memory management, sparse matrix algorithms and object-oriented databases.

Keywords: SIMD, vector processor, data structure, list, memory management.

1 Introduction

Supercomputers have traditionally used vector processors to achieve high processing speeds. In addition, many of the new massively parallel supercomputers are either SIMD processors (such as the CM-2 and the Maspar), or are MIMD processors that have vector processing units attached to each node (such as the CM-5). Researchers have written efficient vectorized kernels for vector and matrix operations (such as BLAS [5]), which allow the vector processing supercomputers to achieve Gigaflop processing rates.

Little work has been performed on vectorizing data structures, so that symbolic processing is typically executed using only the scalar processor. Many algorithms, such those for as asymmetric sparse matrix factorization [3, 4, 6] require a significant amount of symbolic processing and use many data structures. Much of this code typically is executed using the scalar processor, and as a result can be relatively very time consuming.

*available via anonymous ftp at cis.ufl.edu:cis/tech-reports

In this paper, we present a new list data structure, the *Vlist*, which is designed to efficiently exploit the parallelism available from a vector or SIMD processor. We present vectorized insert and delete operations, and also several other vectorized Vlist operations. We show how the Vlist can be used to implement fast memory managers, and discuss its application to asymmetric sparse matrix algorithms and object-oriented databases.

1.1 Previous Work

Many search structures, such as balanced trees or hash tables, have logarithmic or even constant time complexity, so that using many processors to perform a single operation yields only a small speedup [11]. A more promising approach is to use the vector processor to perform many operations simultaneously. A problem that needs to be overcome is that several of the parallel operations might attempt to write into the same memory location. For example, if the vector processor is inserting keys into a hash table, then many keys might be hashed into the same bucket. If no concurrency control is performed, then only the final write will be observable after the vector insert operation.

Kanada has investigated the use of what he names the *filtering-overwritten-label* method, or FOL [7], to overcome the overwriting problem. The key idea is that before each write operation, each processing unit in the vector processor writes its tag to a memory location that is associated with the memory location that is to be modified. If several units of the vector processor need to write to the same memory location, they will write their tags to the same area. The unit that writes last wins the right to execute. So, if a processing unit writes its tag, and then reads it, the unit can modify the memory location. If the unit doesn't read its own tag, it remains inactive and allows the winning unit to execute. the operation is iterated until all units of the vector processor complete their operations.

Kanada has applied this technique to hashing, sorting, tree rewriting, and other types of symbolic processing [8, 9, 7]. Appel and Bendiksen [2] uses a similar technique for vectorizing a garbage collector.

The FOL method requires little modification of the existing data structure, so that it can be easily applied. However, it can require a significant amount of overhead, and performs poorly in the presence of 'hot spots', which one hopes are rare. We choose instead to modify the underlying data structure to support vector operations, and do not use the FOL method. A data structure that is designed to support vector operations will be more efficient than one constructed using the FOL method.

2 The Vlist

The data structure that we vectorize is the linked list, which is a set of records each linked to its immediate successor [10]. The usual scalar linked list isn't a good structure for vector operations, since the list must be followed one record at a time. In order to support vectorized operations on the linked list, we need to modify the structure of the list. Suppose that our vector processor contains n processing units. In the Vlist, each record points to its n^{th} successor. Correspondingly, we need n pointers to the first n elements on the list. Figure 1 shows a Vlist for a five unit vector processor. The Vlist is composed of five *component lists*, or *columns*, with head pointers A through E. For notational convenience, the records in the A list are denoted by Ax. The records in the list are in the order A1, B1, C1, D1, E1, A2, B2, etc. We call a set of records that are the same distance from the head of the list a *row*. for example, (A2, B2, C2, D2, E2) form a row.

If a data structure is to be useful, it needs a set of efficient operations. The first Vlist operation that we specify is the search operation. To search a Vlist, each unit in the vector processor is assigned a unique component list. Each unit traverses its component list until some unit finds a match. In the following code, which is written in a C-like pseudo-code, a test for equality on a vector succeeds only if the test succeeds on all of the components. We assume that `op` returns `FALSE` and `next` returns `NIL` if passed a `NIL` pointer.

Vsearch Operation

```
Vsearch( Vlist,op() )
   point=Vlist.pointers
   while(!(point==NIL))
      (* perform a vector test on the records *)
      flag=op(point→record)
      (* stop if you match on some record, otherwise move to the next row *)
      if(flag!=FALSE)
         break;
      else
         point=next(point);
   (* return the first record matched, or NIL if none found *)
   if(flag==FALSE)
      return(NIL);
   else
      i=index of the first TRUE component of flag;
      return(point_i→record);
```

In a Vlist, all rows are complete except possibly for the last row in the list. This balance allows efficient vector processing of the records on the Vlist, since the time required to process the list is proportional to the longest list. Other vector list processing algorithms can be written using the template of the Vsearch operation.

We can also insert an element into a Vlist in unit time. The key observation is that the records in a component list aren't tied to the component list, but can be moved into other component lists. Figure 1 shows the insertion of record X between B2 and C2. In order to make room for the new record, the component lists must be cyclicly shifted, and only the current and previous records need to be modified. The movement of records to accommodate the inserted record can be vectorized, because there are only four cases, corresponding to whether a column is the first column, the column that receives the the insert, a column that follows the insert, or a column between the first column and the column that receives the insert. The first two cases are handled sequentially, and the last two cases can be executed with the vector processor.

In the following code, we assume that the position to insert the new record has already been found, via a modification to the `search` procedure. The parameter `point` is a vector pointer that with `pos` specifies the position to insert the new elements. The vector `prev` points to the preceding records, if there are any, and to the Vlist vector otherwise. We assume that an assignment to a `NIL` location has no effect, for clarity.

Vinsert Operation

```
Vinsert( point,pos,next,X )
   nextptr=point→next;

   (* modify the list receiving the new record *)
   prev_pos→next=X
   if(pos>0)
      X→next=nextptr_pos-1

   (* modify the first list *)
   point_0 → next = point_n-1

   (* modify lists that come before pos *)
   on vector units 1 through pos-1
      point_i→next=nextptr_i-1

   (* modify lists that come after pos *)
   on vector units pos+1 through n-1
```

$\texttt{prev}_i \rightarrow \texttt{next=point}_{i-1}$

The Vinsert operation preserves the order of the elements on the list. This feature is useful for maintaining sorted lists, and also for maintaining the implicit order of a queue or stack, for example. As records are inserted into the Vlist, the last record in the Vlist moves to increasingly greater column in the same row, starting a new row only after it is moved from the last column. So, the Vinsert operation preserves the balance between columns.

Many other Vlist operations which preserve order and balance can be implemented in a similar manner:

Single record delete: The delete procedure is the opposite of the insert procedure, and can also be implemented in unit time with vector operations. The are again four cases, but the last vector unit is a special case instead of the first.

Single item push (insert to head): The push operation is easily constructed from the insert operation.

Single item pop (delete from head): Likewise, the pop operation is easily constructed from delete operation.

Single item enqueue (insert to tail): If the tail of the Vlist can be found, then adding a record to the tail of the list requires a single scalar operation. In order to efficiently implement the enqueue operation, we need a pointer to the tail of the list. The tail pointer should point to the last complete row in the Vlist. The first record whose `next` pointer is `NIL` is the position to insert the record. This definition of a tail pointer is convenient because it is easy to detect if the tail pointer should be modified due to an insert or a delete: if any record pointed to by `point` has a `NIL next` pointer, then the tail pointer is affected. One additional case occurs if a new last row is completed.

List append: If tail pointers are kept in the Vlist, then appending a Vlist to another requires only a constant number of vector operations. If L2 is to be appended to L1, and the first k columns of the last

row are filled, then link the i^{th} column of L2 to the $(i+k)$mod n-st column of L1. An example of a Vlist append operation is shown in figure 2.

List split: This operation breaks Vlist L1 into 2 pieces, all records before and all records after (and including) a specified record. This operation is the opposite of the list append operation and requires a constant number of scalar and vector operations.

Block insert: The block insert operation inserts k consecutive records into the list. We assume that the starting insert position and the previous row is given, and that the block of records, `X` to be inserted forms a Vlist (constructing a Vlist from a contiguous set of records can be vectorized).

The block insert procedure can be constructed from the Vlist split and append operations. To insert X into L, break L into L1 and L2, append X to L1, then append L2 to the result.

Block delete: The block delete operation can also be implemented with the list split and append operations. Split L into L1 and L2. split L2 into D and L3, where D is the block to be deleted. Append L3 to L1.

Block push and pop: These operations are constructed by executing a block insert or delete at the head of the Vlist.

Block enqueue: This operation is a special case of the list append instruction.

List Merge: In order to merge Vlist L1 into Vlist L2, repeat the following procedure until L1 is empty: Remove the first record on L1, search forward for its position in L2, then insert it. On the next search, start from the current position in L2. This procedure is fast if L1 is much smaller than L2, but it loses all parallelism is L1 and L2 are about the same size.

List Select: This operation searches for records in Vlist L1 on which `op()` returns `TRUE`. These records are removed from L1 and added to L2. This procedure can be implemented by repeated applications of the search and the block delete operations. However, the selected records in a row aren't necessarily contiguous, which prevents effective use of the vector operations.

A more efficient implementation repeatedly removes a row from L1, appends the selected records to L2, and appends the unselected records to a temporary Vlist, L'. When L1 is empty, L1 is assigned L'. To use the block enqueue operation, we need to gather the selected blocks contiguously into one vector, and do the same for the unselected blocks. We can perform this gathering if we compute a prefix sum on the vector returned by the selection operation. If a record is selected, it should be moved to the vector unit whose index is the prefix sum minus one. If the record is unselected, it should be moved to the vector unit whose index is the index of the record's unit minus the prefix sum. The prefix sum is simple enough to be implemented in hardware. If no such operation is available, the $O(\log n)$ parallel prefix sum algorithm [1] can be used.

VSelect

```
VSelect( L1,L2,op() )
   (* initialize a temporary list *)
   L'=NIL;
   while L1 not empty
      selected=unselected=NIL;
      (* remove the first row and test it *)
      row=pop(L1,n);
      flag=op(row);
      (* gather the selected and unselected records into separate lists *)
      PreSum=prefixsum(flag);
      on the units for which flag = 1
         selected_{PreSum_{i-1}} = rowi
      on the units for which flag=0
         unselected_{i-PreSum_i} = row_i
      (* add the selected and unselected records to the
         selected and unselected lists, respectively *)
      append(L2,selected);
      append(L',unselected);
   L2=L'
```

As an example, consider the processing of an eight unit row. Suppose that the records in the columns two, three, five and six are selected. The `flag` vector will be $(0,0,1,1,0,1,1,0)$, so the `PreSum` vector is $(0,0,1,2,2,3,4,4)$. The key observation is that every time a record is selected, the prefix sum increments by one, and every time the record isn't selected, the position minus the prefix sum increments by one.

The Vselect procedure can be modified to add copies of, or pointers to, the selected records to L2 and leave L1 unmodified.

2.1 Vlist Extensions

2.1.1 Doubly Linked Vlist

A Vlist can be doubly linked, to support backwards as well as forwards searching. The implementations of the discussed operations translate in a straightforward manner to the doubly linked Vlist, since the predecessors and successors on the inserted and deleted blocks are known.

One reason for using a doubly linked list is to efficiently delete or insert after a known record on the list. A pointer in a Vlist consists of a vector pointer and an index into that vector. Consequently, an insertion or a deletion of a record near the record pointed to can require that the record pointer be modified. If there are a number of pointers into a Vlist, they must be examined when the Vlist is modified.

2.2 Single-level Indices

If the Vlist is very large, then access to the list can be made faster by a sparse single-level index into the list. The sparse index consists of a fixed number of vector pointers that point to the first row that contains a key which is larger than a specified value. For example, a list of names might have an alphabetical sparse index.

Whenever the list is modified, several of the index pointers might need to be updated. Fortunately, it is easy to determine which pointers are affected, since a check on the range of the modified row can be made. If the list is small compared to the number of indices, though, maintaining the indices is expensive.

3 Applications

The Vlist is a versatile data structure and has many applications. In this section, we discuss a few of them.

3.1 Memory Management

We can use the Vlist to implement fixed-size and variable-size buffer management. Fixed-size buffer management can be implemented with the block push and pop (or the block enqueue and pop) operations. The Vlist structure is valuable if the number of requested blocks varies, because between 1 and n blocks

can be allocated or released with a fixed number of scalar and vector operations. By comparison, a single buffer stack will require n operations to allocate n buffers, and n independent buffer stacks will become unbalanced if the request size varies.

Variable-sized (or heap) memory allocation is often implemented by some variant of a free list algorithm [10], whether first fit, best fit, or worst fit. Compared to other heap memory managers, free list algorithms are slow, but have low fragmentation. We can use the Vlist to implement a fast and space efficient free list memory manager.

In a free list algorithm, free blocks are linked into a list. When an allocated block is released, the list is searched to find contiguous blocks. The contiguous blocks are removed from the free list and the combined block is inserted. To allocate a memory, the free list is searched for the most suitable block to allocate from. The free list algorithm is determined by the definition of suitability. For example, first fit chooses the first free block large enough to satisfy the request.

A vectorized first fit memory manager can be easily implemented with the Vlist. To allocate a block, search for the first free block that is as large as the request and allocate from it. To release a block, search for contiguous free blocks and delete them if they exist, then insert the combined block.

3.2 Sparse Matrix Algorithms

A sparse matrix is one in which most entries are zero, and there is a significant space and computation advantage to storing the matrix in a compressed form.

Much work has been performed on the LU-factorization of sparse matrices. Sparse matrix factorization algorithms work on the compressed matrix. As the computation proceeds, the matrix rows can shrink, as their entries are zeroed, or can grow due to fill-in (non-zeros created by the factorization). In Davis' D2 algorithm [3] rows are stored in linked blocks, and hence could benefit from a vectorized buffer allocator. Other algorithms, known as *multi-frontal* methods [4], build dense submatrices to factor. Allocation of the space for these submatrices can benefit from a vectorized heap memory allocator.

Asymmetric sparse matrix factorization algorithms which calculate the pivoting sequence while they factor the matrix typically make extensive use of linked list data structures to tie together partial results [3, 4, 6]. A typical sequence of operations is to scan the lists that hold the rows and columns to find an acceptable pivot, compute the update, and scan the lists that hold the affected rows and columns to record the effect of the pivot step.

3.3 Object Oriented Databases

Object-oriented Databases (OODBs) have been proposed as a better model of complex applications than more traditional database systems. Su et al. have parallelized the OSAM* OODB [13] by managing objects of different classes on different processors. Objects of different classes that are related are linked by *associations*. Physically, objects in a class store the name of all other objects that they are associated with. OSAM*'s query language allows operations on sets of objects that have a specified association.

Su, Chen and Lam [12] have examined parallel algorithms for determining which the objects that have a specified association. Their algorithms involve searching the list of objects in a class for the specified local associations, then passing this information to the processors that hold the associated objects. Their algorithms would benefit from the Vlist select and merge operations.

4 Conclusions

In this paper, we have presented a vectorized list data structure, the Vlist. The lengths of the component lists are balanced, so the Vlist is suitable for vectorized list processing. We show how many list operations, including insert, delete and append, can be implemented using a constant number of vector and scalar operations. We discuss several applications of the Vlist to memory management and sparse matrix algorithms.

Our approach to constructing vectorized data structures is different than Kanada's FOL technique in that we specialize the data structure for efficient vector processing. As a result, we can construct a highly efficient and vectorizable data structure. However, the FOL technique has the advantage that it requires only simple modifications to the existing data structure, and allows the processing of shared data. Our approach requires a new approach to the data structure, and requires that all vector writes are to different locations.

Our future work will concentrate on applications in numerical analysis, image processing, and database applications. We will search for additional vectorized data structures, and will integrate the FOL technique as required.

5 Acknowledgements

I'd like to thank Dave Bernhold, Mark Schmalz, Tim Davis, and Yaw-Huei Chen for reading and commenting on this paper.

References

[1] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[2] A.W. Appel and A. Bendiksen. Vectorized garbage collection. *Journal of Supercomputing*, 3:151–160, 1990.

[3] T. A. Davis and P. C. Yew. A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization. *SIAM J. Matrix Anal. Appl.*, 11(3):383–402, 1990.

[4] T.A. Davis and I.S. Duff. Unsymmetric-pattern multifrontal methods for parallel sparse LU factorization. Technical report, University of Florida, Dept. of CIS TR-91-23, 1991. Available at anonymous ftp site cis.ufl.edu:cis/tech-reports.

[5] J.J. Dongarra, J. Du Croz, I.S. Duff, and S. Hammarling. A set of three basic linear algrbra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[6] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, 1986.

[7] Y. Kanada. A method of vector processing for shared symbolic data. In *Supercomputing '91*, pages 722–731, 1991.

[8] Y. Kanada, K. Kojima, and M. Sugaya. Vectorization techniques for Prolog. In *ACM International Conference on Supercomputing*, pages 539–549, 1988.

[9] Y. Kanada and M. Sugaya. Vectorization techniques for prolog without explosion. In *Int. Joint Conference on Artificial Intelligence*, pages 151–156, 1989.

[10] D. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley, 1968.

[11] M.J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, 19787.

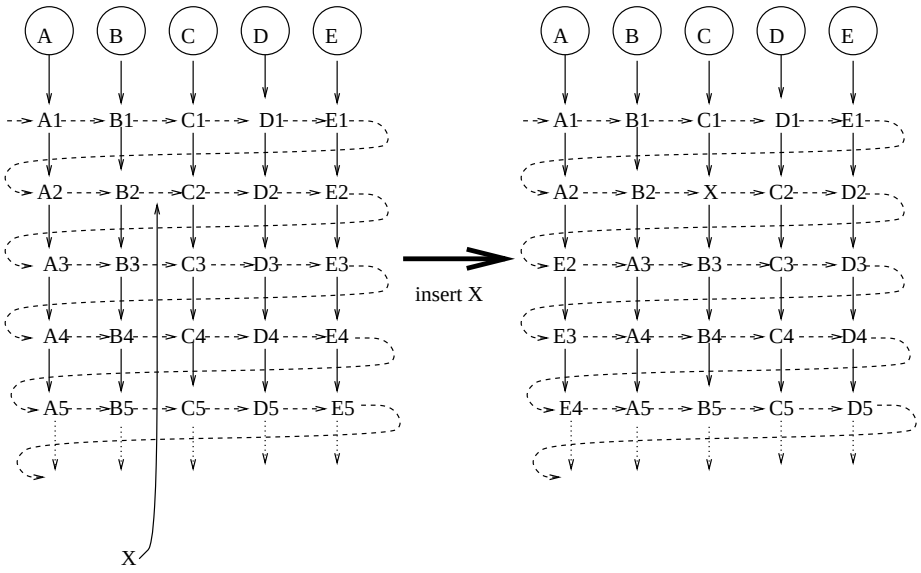

Figure 1: Insert into a Vlist

[12] S.Y.W Su, Y.H. Chen, and H. Lam. Multiple wavefromt algorithms for patern-based processing of object-oriented databases. In *Proc. of the First Int'l Conference on Parallel and Distributed Information Systems*, pages 46–55. ACM and IEEE, 1991.

[13] S.Y.W. Su, V. Krishnamurthy, and H. Lam. An object-oriented semantic association model (OSAM*). In S. Kumara, A.L. Soyster, and R.L. Kashyap, editors, *Artificial Intelligence: Manufacturing Theory and Practice*, pages 463–494. Institute of Industrial Engineers, Industrial Engineering, and Management Press, 1989.

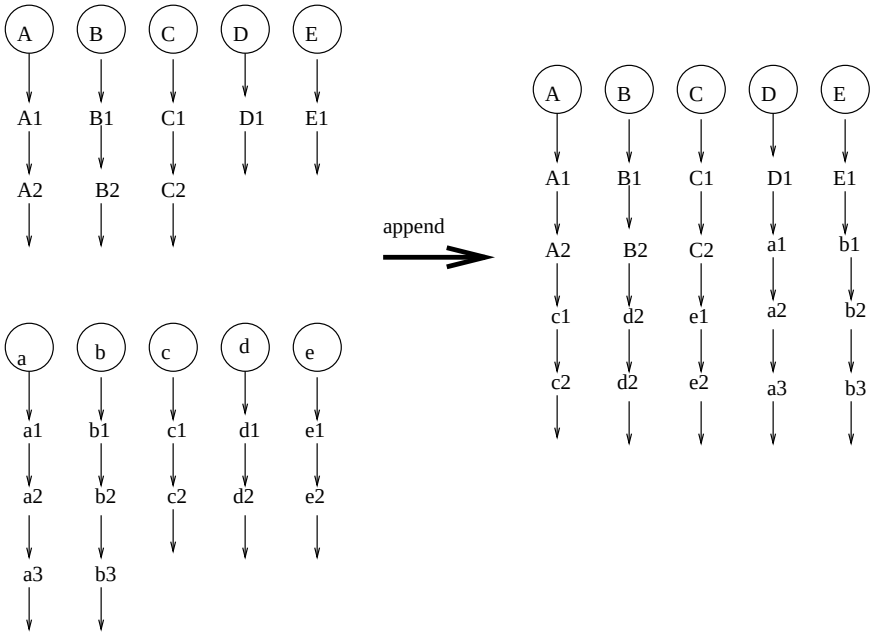

Figure 2: Appending a Vlist