Architectural Mechanisms To Support Sparse Vector Processing

R.N. Ibbett, T.M. Hopkins and K.I.M. McKinnon

Departments of Computer Science and Mathematics
University of Edinburgh
James Clerk Maxwell Building, King's Buildings
Mayfield Road, Edinburgh, EH9 3JZ

Abstract

We discuss the algorithmic steps involved in common sparse matrix problems, with particular emphasis on linear programming by the revised simplex method. We then propose new architectural mechanisms which are being built into an experimental machine, the Edinburgh Sparse Processor, and which enable vector instructions to operate efficiently on sparse vectors stored in compressed form. Finally, we review the use of these new mechanisms on the linear programming problem.

1 Introduction

Sparse vectors are an important feature of a number of computer applications. Their distinguishing characteristic is the occurrence of large numbers of zero elements in vectors and arrays, and in mapping sparse applications on to existing computers a variety of software techniques have been employed to reduce the storage and processing required for the zero elements. Most sparse codes run on *scalar* machines, or the scalar processors of vector computers, and make no use of standard vector processing facilities. A few computers, notably the CDC CYBER 205, have provided architectural support for sparse vectors in the form of addressing modes and special orders [7], but these orders have proved difficult to use in practice, mainly because of the *fill-in* problem. Fill-in occurs when, for example, two sparse vectors are added and the positions of the non-zero elements in the two vectors do not match, so that the result vector contains more elements than either of the source vectors. Since the extent of fill-in cannot be predicted at compile time, the compiler cannot know how much space to allocate to sparse vectors which are created during the running of a program.

2 Sparse Matrix Computation

Computation with sparse matrices whose pattern of sparsity is regular (*eg* matrices arising from partial differential equation solution by finite difference or finite element methods) can often be carried out efficiently on standard vector processor machines. Computation on matrices with irregular sparsity pattern is not amenable to these techniques, and so it is problems of this nature in particular that the proposed new architectural mechanisms address.

Irregular sparsity patterns arise from irregularity in the real-world problem being modelled, typical examples being found in engineering design simulations of physical structures or electrical circuits, and in Linear Programming (LP) problems. As examples of the type of vector and matrix calculation steps a sparse vector processor must support, we examine the steps of the *Product Form of Inverse (PFI) simplex* algorithm for Linear Programming, which include (in the re-invert step), the more general problem of the direct solution of a system of sparse linear equations.

2.1 Linear Programming

LP problems are typically very sparse. A large LP problem might involve a matrix of several thousand rows and 3 times as many columns, but with only 6 or so non-zeroes in each column. The four most computationally intensive steps of the PFI simplex algorithm are the so-called **BTRAN**, **pricing**, **FTRAN**, and **re-invert** steps, which typically take 30%, 35%, 20%, and 7% respectively of the total solution time. In the PFI method, the current inverse of the basis matrix is held as a series of *PFI matrices*, of special form, the product of which is the basis inverse. Each pivot step in the solution adds another PFI matrix to the series, so the series will usually contain many hundreds of matrices. Each of these has the same number of rows as the original problem matrix, and has the form of the unit matrix plus a single non-zero column, which may typically have a density between 1% and 20%. For each PFI matrix, only the non-zero column, plus a record of its position in the matrix, need be stored; these column vectors are generally known as the η (eta-) vectors.

2.1.1 BTRAN

This involves the post-multiplication of a vector by each of the PFI matrices in turn. Because of the special form of the PFI matrices, each multiplication step reduces to the replacement of one element of the vector being updated with the scalar product of that vector and the η vector of the relevant PFI matrix:

$$v_k \leftarrow \mathbf{v}.\eta$$

where k is the column position of the η vector in its PFI matrix.

2.1.2 Pricing

The result of the BTRAN operations is a *price vector*, and the pricing step involves the formation of the scalar product of this vector with each of the columns of the original problem matrix. The price vector will typically be 40% dense, while the original columns are very sparse - less than 0.1%.

2.1.3 FTRAN

This step involves the pre-multiplication of a vector (a column of the original problem matrix) by each of the PFI matrices in turn. Each multiplication step reduces to the summation of the vector with the relevant η vector, first scaled by a single element of the vector being updated:

$$\mathbf{v} \leftarrow \mathbf{v} + v_k * \eta$$

As above, k is the column position of η in the PFI matrix.

The vector being updated starts very sparse ($< 0.1\%$) and typically reaches 20% density at the end.

2.1.4 Re-invert

For reasons of numerical stability, it is necessary from time to time during the solution to replace the accumulated series of PFI matrices with an equivalent series of matrices of the same form derived by direct Gaussian Elimination on the current basis matrix. That matrix is a portion of the original problem matrix, and is therefore less than 0.1% dense. Each pivot step in the elimination will consist of the subtraction of a multiple of the pivot row from rows of the (updated) matrix with a non-zero in the pivot column:

$$B_{n*} \leftarrow B_{n*} - S * B_{p*}$$

where B_{n*} is a row of the matrix B with a non-zero in the pivot column, B_{p*} is the pivot row, and S is a scalar equal to the element of B_{n*} in the pivot column, divided by the pivot element.

Here, both vectors are of roughly equal sparsity (although if the Markowitz criterion *(see below)* is used to choose the pivot, the pivot row will tend to be somewhat sparser than the other). When the matrix is very sparse, the subtraction is fast, and it is crucial that one can rapidly determine which rows have a non-zero in the pivot column, and access the value of that non-zero, or this operation will dominate.

As the elimination proceeds, the part of the matrix remaining to be eliminated becomes more dense. This can lead to a drastic increase in the number of floating point operations required, both to complete the elimination, and in the remainder of the LP solution. It is therefore vital to minimize fill-in by careful choice of pivot at each step. However, except in the special case of a matrix which is symmetric positive definite, it is also necessary to consider numerical stability in choosing the pivots. The usual compromise is known as *threshold pivoting* [2]. A potential pivot is chosen by selecting an element whose row and column are both very sparse (the *Markowitz* criterion [9]), but the potential pivot is rejected if it is too much smaller than the largest element in its column. Operation of this method requires knowledge at each pivot step of the number of non-zeroes in each row and column of the updated matrix. It also requires that the each non-zero in the proposed pivot column be accessible in reasonable time.

3 ESP Sparse Vector Mechanisms

ESP is a vector processing system consisting of a scalar control processor and a separate vector processing pipeline, in a manner similar to the CYBER 205. However, ESP is not designed as a stand-alone computer, but rather as a high-performance back-end co-processor which executes routines to complete critical core sections of numerical programs, under control of a main program running on the host computer. These main programs may be compiled on the host, and a range of optimised ESP routines will be available as a library. The prototype is being designed as a co-processor for an ORION [6] minicomputer, but it is intended that the design be easily adaptable to operate with commonly available workstations as host machines.

The routines executed by the ESP co-processor consist of a mix of vector instructions, and more conventional scalar and control instructions. A typical vector instruction involves two source vectors and one destination vector. As in the CYBER 205 [7], each vector is accessed via a descriptor, but in ESP the descriptors contain more information about the vector, and are more reminiscent of those used in MU5 [10]. In order to support the kinds of operation required for efficient execution of LP and other sparse problems, two different storage mechanisms for vectors are provided in ESP and many instructions do not specify the storage mechanism used for their operands (this information is in the descriptor) but will work on vectors stored in either form. The descriptor also contains the necessary information for the storage controller to find the vector elements, and information on the type of the vector elements (single or double precision etc.).

The first storage mechanism is the **full** vector. Here a vector is stored in standard form as a sequence of values in store, so that an n-element vector occupies n storage locations, and the element of a vector with a specified index is found by offsetting from the vector base address specified in the descriptor. While efficient for access to any element from its index, this mechanism does not support fast access to the non-zero elements only, and of course wastes enormous amounts of store if vectors are very sparse. For sparse vectors, the **list** vector storage mechanism is used.

3.1 The List Vector Mechanism

3.1.1 Linked List Methods - General Strategy

Storing a sparse vector as a list of non-zero values, plus a corresponding list of the indices of the non-zeroes within the vector is attractive, as it incurs *no* storage or access overhead for the zeroes, and thus remains efficient even for very sparse vectors. Standard vector/vector operations, such as add, subtract and scalar product, can be performed directly on vectors stored in this form if *both* the index list and the value list of each operand are simultaneously accessible by the processor, with the lists maintained in order of ascending index.

This ordering of the lists is necessary for efficient implementation of vector/vector operations such as $C \leftarrow A + B$. These are implemented by streaming the lists A and B into the vector processor's arithmetic unit (AU), using additional hardware in the AU input stage to align A and B elements with equal indices. Thus in the example of the add instruction, illustrated in figure 1, if the first index in the A list is n, and that in the B list is m, then if $n < m$ the first element output is the first index/value pair from A. This element is discarded from the list A, while the list B is unchanged. Similarly for $m < n$, while if $n = m$, the output element is the sum of the elements at the front of the A and B queues, both of which are then discarded. Input vector index matching is also required for vector/vector multiplication

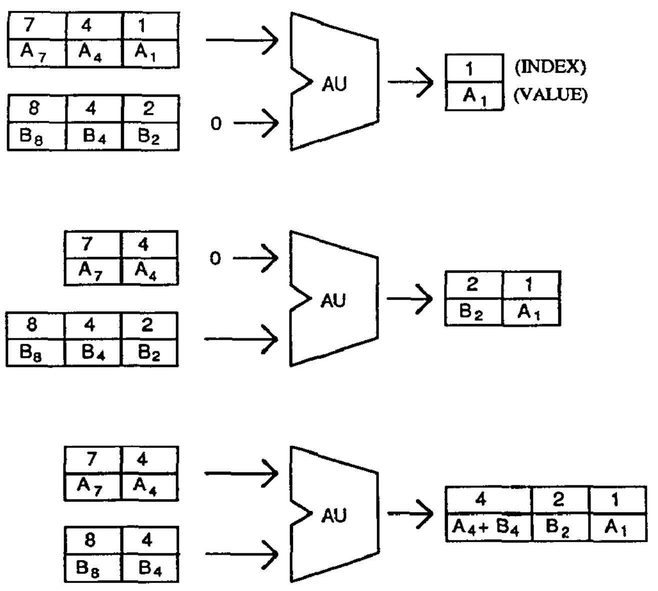

Figure 1: An ADD operation on list vectors

operations, but in this case a multiply step is only required if non-zeroes appear at the same index position in *both* input streams. If either or both of the vector lists were not in ascending index order, vector/vector arithmetic would involve searching for matching indices, and would be very inefficient.

As has already been noted, the amount of space needed to store a sparse vector in compressed form is not usually known at compile time. Nor is it in general known at run-time, even at the start of the operation producing the vector. In the add example above, the output list C may be as short as the longer input list or as long as the sum of the input list lengths, depending on the extent to which the postions of non-zeroes in the input vectors coincide. In some operations (*eg* the run-time compression of a vector from full storage format to list format) the resulting list length may be anywhere within much wider bounds. For the list storage format to be useful therefore, the amount of memory space allocated to a vector must be dynamically and automatically variable. The hardware must maintain a pool of free memory space, allocating extra space from the pool to vectors as required. Because it is impossible tell at the start of a vector operation how long the result list will be, it is not feasible to allocate at the start a single free block of store guaranteed to be large enough to hold the result. The solution adopted is to allow the index/value list comprising a single vector to reside in one or more *linked blocks* of memory locations. If the block allocated to hold the result at the start of an operation turns out to be too small, another block can be linked onto it. The unused portion of the last block allocated may be returned to the free pool.

Space may be freed by explicit de-allocation (under program control) of the memory used by temporary vectors which are no longer required. However, more often, it will become free automatically. To see why this is so, consider an operation of the type $A \leftarrow A + B$, and suppose vector A has non-zeroes at index positions 100 - 199, while B has non-zeroes at index positions 0 - 99. If the result vector is written into the memory blocks already occupied by A, then the first 100 output elements will overwrite the 100 non-zeroes of the original A. The first few of these will be in the arithmetic unit input pipe, but most will not yet have been read from memory, as the AU must deal with all 100 non-zeroes of B before using up any from A. As a result, most of the required input elements will have been corrupted before they are read. To avoid this, when a vector appears as both an input and result of an operation, the result vector must be allocated new space, and the original space used by that vector automatically returned to the free pool at the end of the operation.

3.1.2 Implementing Linked Lists in a Single-level Memory Environment

These list structures can be implemented in a simple way by treating memory as a pool of fixed size (small) blocks, each with a single link field. The free space is a linked list of unused blocks. As a vector operation produces its result, that result is written into the first locations on the free list, following links as required. The result vector's descriptor is updated to hold a pointer to the start of the vector list. The unused part of the final block in the result vector is wasted; this is the reason for small blocks. Reclaimed space is linked onto the start or end of the free list. The hardware required to support these operations is simple, and the operations of allocating new space and reclaiming old space are both very fast, each requiring only the updating of processor pointer registers, and the alteration of two links in memory.

Matrix codes tend to use many operations of the type $A \leftarrow B$ **op** A. For example, every elimination step in Gaussian Elimination causes a vector to be re-written, usually with a small increase in density, and for reasons in explained section 3.1.1 above, these re-writing steps involve the allocation of new space for the updated vector, and the reclaiming of space previously used. As space from vectors is reclaimed and later used again by vectors of different length, the blocks on the free list will become thoroughly mixed. As a result, the blocks used to store any vector will be randomly distributed throughout the whole memory space. In a single-level memory environment this may not matter, but in a hierarchical memory environment it is very likely to lead to thrashing of the paging/cacheing system. We believe that to restrict ESP to problems which will fit into a restricted single-level memory space (or which can be explicitly partitioned into smaller problems which fit) would be a mistake. We have therefore rejected this simple implementation in favour of mechanisms which retain more locality of reference.

3.1.3 Implementing Linked Lists in a Hierarchical Memory Environment

In this implementation, the free list remains a linked list of blocks of free memory, and allocation of new space for a result vector proceeds as above, except that blocks may now be of any length. Any space in the last block allocated (to a result vector) which remains unused at the end of the operation is left, as a smaller block, on the front of the free list. The key to maximising locality of reference lies in ensuring that the blocks on the free list remain as large as possible, so that the space allocated to a new vector consists of a small number of large blocks; this requires a more complex de-allocation algorithm. In general, a vector to be de-allocated itself consists of a list of blocks, and the de-allocation algorithm must check, for each of these blocks, whether is is *adjacent in memory* to a block (or blocks) already on the free list, and if it is, must merge the blocks. A simple way of achieving this merging de-allocation is to maintain the blocks in the vector lists and in the free list in order of ascending memory address (*ie* links from block to block are always *forward* through the memory address space). The de-allocation algorithm may then merge the two sorted lists of blocks in a straightforward way.

In ESP, this mechanism is supported by hardware interposed

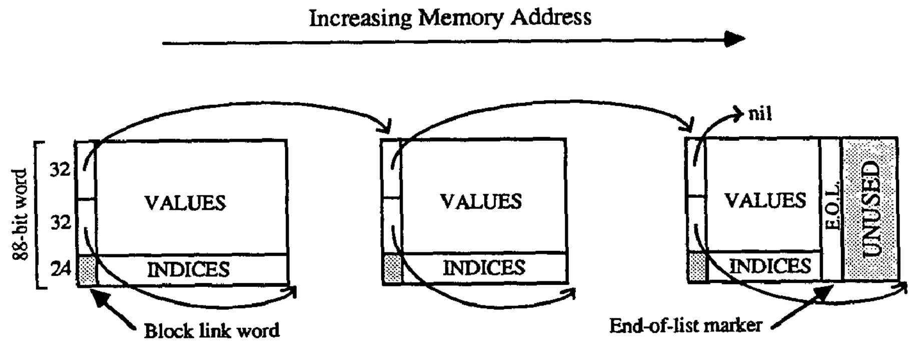

Figure 2: Structure of linked list vector

between the arithmetic unit and the memory. Vector elements are held in memory as an index/value pair, in a single memory word of 88 bits (64 bits for the value, and at 24 for the index). In addition to straight index/value pairs, memory words may hold an *end-of-list marker*, or they may hold *block link* words. A list vector is held in a series of blocks of consecutive memory words, the final word in the list being an end-of-list marker (see Fig. 2). The first word of each block is a block link word, and these words contain *two* pointers, each of which is a 32 bit (or larger) virtual memory address, known as the *external* pointer and the *internal* pointer. The external pointer holds the address of the first word (*ie* the word containing the block link) of the next block in the list, and is there to maintain the list linkage. The external pointer in the last block of a list holds the special value **nil**. The internal pointer holds the address of the first word *after* the end of the current block, and is there because the de-allocation algorithm needs to know the size of blocks to perform concatenation of adjacent blocks. Because the blocks are in order of increasing memory address, all pointers point forwards. Vector descriptors contain, in addition to other information about vector type and size, a pointer to the current position of the first word of the first block in the vector list. The free list is of identical structure, and a pointer to the start of it is maintained in a register.

As the AU produces index/value pairs as the results of a vector operation, these are written into the free list. The hardware for performing this contains a small number of pointer registers to keep track of the position in the free list currently being written to, and the block linkage. Eventually, the AU will signal the end of the result vector, by producing an element with the form of the special end-of-list marker. This is written out in the normal way, and a check is performed to determine how much free space remains in the block currently being written to. If this space is less than the minimum block size, it is left on the end of the result vector, otherwise it is reclaimed (by writing new block link words), and left at the front of the free list.

The de-allocation hardware, which reclaims vector space for the free list, requires two pointer registers (**A** and **B**), plus a small number of working registers. The algorithm merges two ordered lists of blocks - the free list, and the vector list being de-allocated. At the start, the free list pointer register is updated to point to the lower of the two list starting addresses. **A** also points to this address; **B** points to the other list. During the algorithm, the pointers **A** and **B** proceed along the two lists. The external pointers in the block link words are adjusted to merge the two lists, while the internal pointers are examined to check for contiguous blocks, and updated to merge such blocks. The algorithm terminates when one list is exhausted, and at this point, all blocks are linked, in order of increasing memory address, onto the list pointed to by the free list pointer. The maximum number of algorithm iterations will equal the total number of blocks on both lists which occupy memory locations between the lower starting address and the lower end address of the two lists.

3.1.4 Efficiency Considerations

The transfer of operand elements between memory and the arithmetic unit can be made as efficient for list structured vectors as it is for vectors stored in the usual full form. Because the link word is at the *start* of the block, if blocks are above a certain minimum size, there is time to emit the start address of the next block to the memory sufficiently far in advance to avoid a gap in the address generator→memory→AU pipeline. The writing of result elements back to memory may also be effectively pipelined.

Many vector processing systems use interleaved banks of memory to achieve the memory bandwidth required to run the arithmetic unit at full speed and in ESP, within a block of a list vector, interleaving will work effectively. However, even though the link address is known well in advance, if the first word of the next block falls into the wrong bank, there will be a hiatus in the interleaving. To avoid this, it is sensible to restrict all blocks to starting in a particular bank, and all pointers are thus multiples of the number of banks. For example, in an eight-way interleaved memory, to allow full use of interleaving and pipelining, pointers should be restricted to be a multiple of 16.

A potential performance limitation in the system so far described is the de-allocation operation (a de-allocate must be performed after most vector operations). How long this takes depends on the number of blocks on the free list and on the vector list being disposed, and on the start and end positions of both lists; in the worst case, the algorithm must examine every block on both lists to complete the de-allocation. However, there are several ways of mitigating the delays caused by this operation.

Firstly, note that the list restructuring remaining to be performed at any time during de-allocation takes place *beyond* the locations pointed to by **A** and **B**. Since the new free list pointer is set up at the *start* of the de-allocate, writing the result of the next vector operation into the free list can commence almost immediately after any pending de-allocate has started, and can continue concurrently with the de-allocate, subject to the condition that each free block used by the write must start at a memory address *less than* the value of pointer **A** (which in the particular algorithm used, is itself always less than **B**). If this condition fails, the write must be delayed until it is again satisfied. In this way, so long as

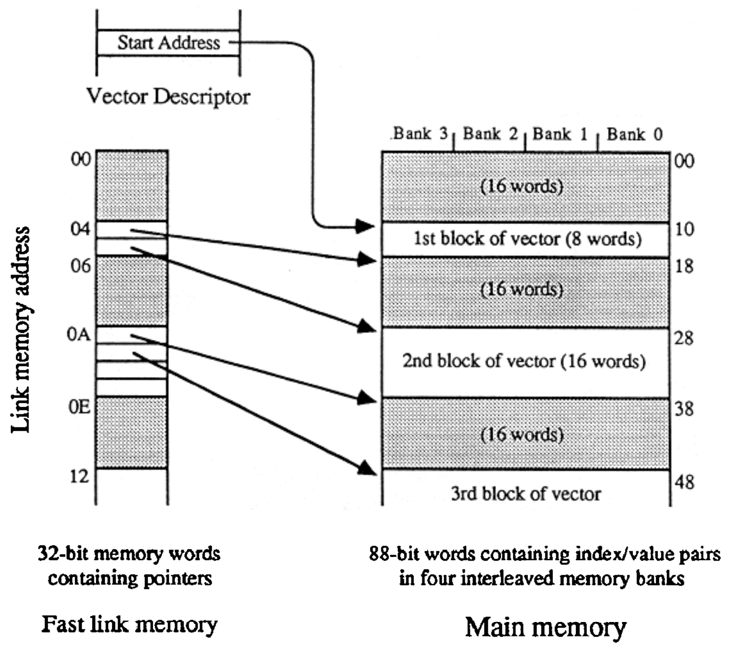

Figure 3: List vector storage with separate link memory

the de-allocation of vectors normally takes less time than writing them, de-allocation need not necessarily delay the processor.

For de-allocation to work concurrently with writing (which is also often concurrent with the reading of one or two streams of input operands *into* the arithmetic unit), there must be plenty of memory bandwidth, and since all pointers are restricted to one bank of memory (all blocks start in the same bank), the bandwidth requirement is considerably greater for that bank than for the other banks. One way of providing this extra bandwidth is to provide a completely separate memory to hold the block link words, as illustrated in figure 3. Here, the words in the link memory are 32 bits wide, which is large enough to hold one pointer only. There is one word of link memory per four words of main memory, and the internal and external pointers of a list vector block which starts at main memory address a are at link memory addresses $a/4$ and $a/4+1$ respectively. Since every block uses two link memory words, the minimum block size is 8 words. Virtual to real mappings must be maintained in parallel on both memories, but the link memory need not be accessible by the vector or scalar processors, only by the memory controller. Of course, if blocks are large, large amounts of the link memory will be unused, and so the dual memory system introduces an overhead of wasted memory area. However, this overhead is more than compensated for by the main memory bus bandwidth gained, and by the simpler bus arrangements which result from the separation of the two memory types.

Finally, alternative deallocation strategies have been considered. By linking lists in both directions, and tagging the block link words in free blocks, it is possible to deallocate a list of blocks in a time proportional to the number of blocks in the list being deallocated, independent of the number of free list blocks (see for example, memory management algorithms in [8]). Because, as described above, deallocation can be overlapped with vector writing, we do not think that the extra complexity of deallocation hardware would be justified. However, simulations of ESP's mechanisms, and we hope, the prototype hardware itself, will be flexible enough to experiment with alternatives in this respect.

3.2 The LP problem on ESP

List vectors do not waste store, and they provide immediate and implicit identification of the non-zero positions of the vector. An operation like the Gaussian Elimination step $B_{n*} \leftarrow B_{n*} - S * B_{p*}$, where both vectors are of roughly equal sparsity, will execute efficiently using list vector storage and a single ESP vector instruction, which operates by streaming operand elements into the arithmetic unit in the manner described in section 3.1.1 above. This is also true of other vector/vector operations on vectors in list form, where both vectors are of roughly equal sparsity. In the *pricing* step of LP (section 2.1 above), however, the price vector is much denser than the column vectors it is multiplied into. To execute the multiplication by streaming in two list vectors would be inefficient, as most of the elements of the price vector would be discarded because there is a zero in the corresponding position of the column vector.

If one vector is several times denser than the other, the scalar product operation is more efficient with the denser vector stored in *full* form. The sparser vector (stored in list form) moves into the vector unit element by element, and the index fields of the elements are used to offset into the denser vector, using normal indexed addressing (this is similar to the *gather* operation supported in hardware in several vector supercomputers [7]). Obviously, access to the elements of the denser vector is slower than streaming a vector out of memory, because the elements accessed are in non-consecutive locations, and memory bank interleaving will be interrupted, and so this method of access is only preferable where the sparsity of the two vectors differs by a factor of four or more.

Performance on list vector/full vector operations can be further increased, if the same full vector is to be operated on many times over, by providing a fast access vector register near the arithmetic unit, to hold the full vector operand. This reduces memory bandwidth use, and circumvents the problem of failure of interleaving.

The usefulness of a vector register is even clearer in the case of the *BTRAN* and *FTRAN* steps. BTRAN also requires a scalar product of a sparse η vector with a vector which is (for most of the BTRAN steps) less sparse, and then requires that one element of that less sparse vector be replaced with the result of the product. This final step requires access to an element of the vector with specified index, and is clearly *very* inefficient on a list vector. However, it can be carried out with ease if the vector is stored in full form in a register. The result of the complete series of BTRAN steps is the price vector, which is thus conveniently in the register ready for the pricing step.

FTRAN requires a summation of a sparse η vector (scaled) with a vector which for most of the FTRAN steps is denser than the η vector. The scaling factor to be applied to the η vector is an element of the denser vector, specified by its index value. It is therefore useful to store the vector being updated in full form in the register, to allow indexed access to these scaling elements.

Finally, the *re-invert* step provides special problems. These cannot be overlooked, as it is intended that ESP should be generally useful on a wide range of sparse matrix problems, including the solution of large sparse linear systems of equations by Gaussian Elimination, which corresponds to the re-invert step of LP.

3.3 Sparse Gaussian Elimination on ESP

In some cases, in particular when the matrix to be factorised is symmetric positive definite, it is possible to decide which elements to use as pivots on sparsity grounds only, before the elimination starts [2]. The matrix may be permuted so that pivoting proceeds down the diagonal, and it is possible to work out in advance (ignoring cancellation during the subtraction steps) the positions of non-zeroes in each pivot column. Elimination will then be very efficient on ESP with the rows of the matrix stored as list vectors. Many Gaussian Elimination implementations on standard computers need to switch over from 'sparse code' to 'dense code' when the density of the filling matrix reaches a critical value. This is not necessary on ESP - operations on list vectors, such as that illustrated in figure 1, remain more efficient than equivalent operations on full vectors, however much the vectors fill in.

As described in section 2.1.4 above, Gaussian Elimination on matrices which are not symmetric positive definite requires knowledge of the number of non-zeroes in each row and column of the partially eliminated matrix, and also requires rapid access to the non-zeroes in each column. In this case, the number and position of non-zeroes in each row and column of the matrix as elimination proceeds cannot be determined in advance. If the matrix is stored within ESP as row vectors in list form, then the elimination steps themselves are efficient, but choosing the pivot is very inefficient. This is because the number and positions of the non-zeroes in each *row* are available (vector descriptors include a field specifying the number of non-zeroes in the vector), but not the corresponding information for each *column*. It is also not possible to access directly the non-zeroes in a specified column - one must search down the row vectors to find them. To support the pivot choosing algorithm, codes for Gaussian Elimination of sparse indefinite matrices which run on *scalar* machines normally store the matrix as a linked structure linked in two directions, along both rows and columns. However, to provide links to matrix elements by column is directly at variance with the dynamic nature of list vector storage in ESP - if a row of the matrix is operated on by a vector instruction, it will move in memory, invalidating any pointers to its elements. An extra facility has therefore been added to the vector processor in ESP, known as the *sideways list unit (SLU)*, which supports maintenance of lists of non-zero *indices* (but not values) by column as the elimination proceeds (ignoring cancellation during subtractions).

3.4 The Sideways List Unit

The SLU keeps updated counts of the *number* of non-zeroes per column of a matrix, and their positions, throughout Gaussian Elimination by rows. The list of non-zero positions in each column is kept in main memory as a linked list of single memory locations each holding a non-zero position (24 bits) and a link to the next word on the list (32 bits), as illustrated in figure 4. These 56-bit pairs are held in the 64-bit value field of locations of a vector which is itself stored in the ESP list format described in section 3.1.3 above. Many such lists of non-zero positions can be stored inside a single ESP list vector and, since the non-zero position lists do *not* have to be linked forwards in memory, a single non-zero position list can extend through several ESP list vectors. The reason for storing the non-zero position lists *inside* ESP list vectors is that memory space for extending the non-zero position vectors can then be allocated using the standard list vector allocation mechanism, and when elimination is complete, all the space can be de-allocated by de-allocating all the list vectors used for this purpose.

After the first pivot is chosen, but before the elimination steps using that pivot row are executed, a list vector is produced (using a vector instruction which generates a vector of specified length) with number of non-zeroes equal to the maximum number of new non-zeroes that the elimination with that pivot row can possibly produce. (That number is the multiple of the number of non-zeroes in the pivot row and the number of non-zeroes in the pivot column, and has already been calculated during pivot choice using the Markowitz criterion.) The descriptor of this list vector is passed to the SLU, so that the vector can be used as space into which to expand the lists of non-zeroes in the matrix columns, during the elimination steps with the first pivot row. The vector is known as the *SLU space vector*.

A single elimination step consists of the operation $B_{n*} \leftarrow B_{n*} - S * B_{p*}$. Whenever the arithmetic unit produces a non-zero in an index position in the result vector B_{n*} which contained a zero in the left-hand input operand, that element is a new non-zero, and its index, i, (its column position in the matrix) is passed to the SLU. The SLU contains a register holding the row number n, and three vector registers, one (the *count register*) holding a count of the number of non-zeroes in each column of the matrix, the second (the *base register*) holding the address of the start of the list of non-zero positions for each column, and the third (the *address register*) holding, for each column, a pointer to the address of the next free location in the list of non-zeroes in that column. On receiving an index i from the AU, the SLU increments the non-zero count for column i, and adds the row number n of the new non-zero onto the non-zero list for column i, by writing it, togther with a link to the next free location in the SLU space vector, to the address pointed to by the ith entry in the address register. It then updates that entry in the address register, loading into it the address of the next free location in the space vector. Since the space vector is an ESP list vector, determining the next free location may involve following a link to the next block of the space vector.

When all the elimination steps with the first pivot row are complete, the SLU space vector may still contain some unused locations, as the real amount of fill-in may have been less than the possible maximum calculated at the start. The SLU maintains a count of the number of locations remaining in the space vector. After the second pivot is chosen, the maximum possible fill-in during elimination with *that* pivot may be calculated, and a new list vector generated and queued for use by the SLU, to replace the current SLU space vector when it is full. This is repeated for

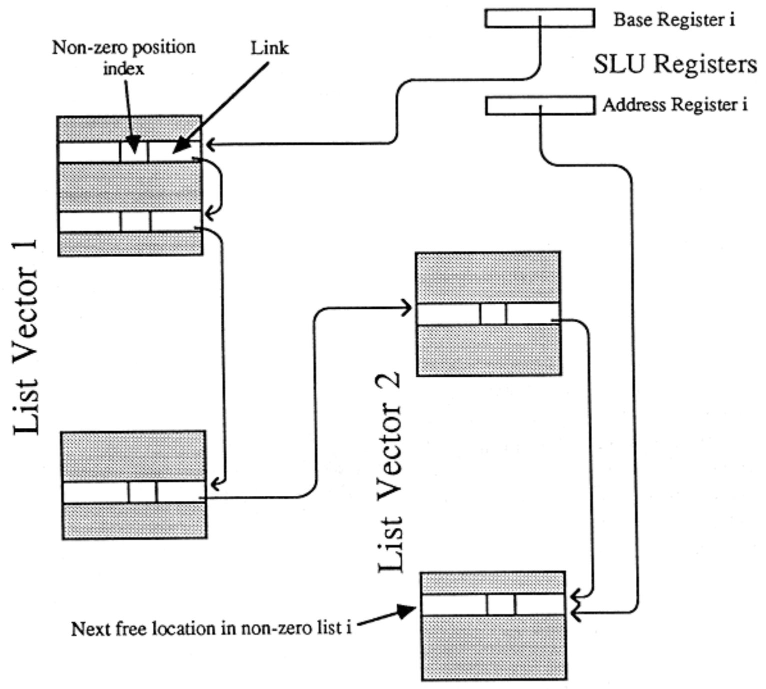

Figure 4: The structure of non-zero position lists

each new pivot, and ensures that the SLU will never run out of space.

The information maintained by the SLU is accessed by ESP's scalar/control processor during pivot choice. The new non-zero counts for all the columns in which there were non-zeroes in the previous pivot row (these are the only columns which can have had extra non-zeroes added during the elimination steps with that pivot row) are read from the SLU to enable pivot choice by the Markowitz criterion. The positions of the non-zeroes in the chosen pivot column are then read from the SLU, which itself reads them direct from main memory, following the links. The *values* of the non-zeroes must be found by using the vector pipeline to search down the relevant rows until the correct column index is reached (using a vector instruction which extracts from a vector the element with a specified index). Depending on sparsity, these operations are likely to involve an overhead of perhaps 100% on top of the time for the subtraction steps of the elimination, and this compares very favourably with the total time required for pivot choice (compared with elimination time) in scalar Gaussian Elimination codes for sparse indefinite matrices [2].

Before the elimination steps start, the SLU must be fed the positions of the non-zeroes in the original matrix, and this is achieved by first allocating a space vector large enough to contain all the non-zeroes in that matrix. The SLU registers are then initialized by loading zeroes into the count register, and loading the base and address registers with locations taken from the space vector. The operation $B_{n*} \leftarrow 0 + B_{n*}$ is then performed on each row of the matrix. Here, the left-hand operand is always zero, so every non-zero position in B_{n*} is passed to the SLU to be added to the relevant column non-zero position list.

4 Performance

Mechanisms similar to those described above for operating directly on compressed sparse vectors have been implemented in the *software* of sparse matrix programs for many years [1, 3, 11, 2]. Such programs run on scalar processors and do not use vector instructions. By providing hardware to implement the vector loops in these programs as single vector instructions, with separate hardware units operating in parallel on the subfunctions which scalar implementations control with separate instructions, we expect to achieve an order of magnitude improvement in the arithmetic rate of these codes.

More recently, there has been interest in the use of the hardware vector indirect addressing facilities (*scatter/gather*) provided in some vector machines (eg CYBER 205, IBM 3090VF), to support sparse vector operations. Because of the bank conflict problem (see section 3.2 above), such operations will always be several times slower than ESP's sparse vector instructions, if the vectors concerned are of roughly equal sparsity. Where the sparsity of the two operand vectors differs markedly, indirect addressing is relatively more efficient, and ESP's provision of a large vector register allows such operations to proceed at the full rate of the arithmetic pipeline.

A further problem (on existing machines) with vector instructions involving indirection into one of the operand vectors is the long instruction startup time, due to the long effective pipeline length. Because LP problems involve sparse vectors with a very small number of non-zeroes, we have been concerned to keep vector startup times to a minimum, and have therefore decided to implement the vector pipeline as several short, independently controlled, sections. A queue is provided for vector instructions which have been issued by the control processor, but not yet executed, and there is a mechanism for the control processor to determine whether a particular vector instruction has completed.

Together, these provisions allow us to have several vector instructions flowing through the pipeline at once, substantially reducing effective start-up times. Simulations are underway to quantify the resulting speed-up on representative problems, and to determine the optimum number of pipeline sections.

The detailed design of the prototype ESP is now underway. This will be built in slow technology (clock speed 100-200ns), using standard VLSI arithmetic components, and will achieve peak speeds of up to 20 MFLOPs. The prototype will allow us to make a thorough investigation of the new mechanisms on real, large, sparse matrix problems.

5 Conclusions

Pipelined vector processors achieve their high performance in part by taking advantage of the storage of vector elements in sequential memory locations. The mechanisms developed for ESP allow this advantage to be maintained in the case of sparse vectors, by providing a new form of vector storage, the *list* vector. The list form wastes no space for the zeroes in a vector, and unlike compressed sparse vector storage mechanisms in other machines, solves the problem of fill-in.

ESP supports all the normally found vector/vector operations, including two operator functions such as scalar product and the Gaussian Elimination step $B_{n*} \leftarrow B_{n*} - S * B_{p*}$, as single vector instructions which will operate directly and efficiently on full form vectors, list form vectors, or a combination of the two. The operations work efficiently over the whole range of non-zero densities. This allows the advantages of vector processing to be extended to the case of sparse vectors.

However, some particular computations commonly performed on sparse matrices, such as the Gaussian Elimination of an indefinite matrix, require that information be maintained about the matrix as a two-dimensional object, rather than simply as a set of one-dimensional vectors, throughout the program's execution. Although it is possible to write algorithms which treat the matrix symmetrically, allowing it to be viewed both by row and by column, fully symmetrical treatment is not possible without sacrificing the fundamental advantage of sequential vector element storage. A compromise solution to this particular problem has therefore been developed, which retains the full advantages of the *one-dimensional* list vector system described above, but in addition allows information about the non-zero distribution in the *two-dimensional* matrix to be maintained in easily accessible form. The mechanism which supports this, the *sideways list unit*, has been developed with the specific requirements of Gaussian Elimination codes in mind, but it is expected to prove useful in other sparse matrix computations.

There has recently been interest in putting sparse problems on to *parallel* processors [4, 5]. We believe that, whilst coarse grain parallelism will clearly enable the solution of very much larger problems, the increasing scale of silicon circuit integration will mean that individual processors can cost-effectively incorporate extra hardware to exploit parallelism at the level of single vector or matrix operations. This is the level of parallelism at which ESP derives its advantages. Such individual processors may then be connected to work in parallel on larger problems.

Acknowledgements

The ESP project is a collaborative project between the Departments of Computer Science and Mathematics at the University of Edinburgh, and High Level Hardware Ltd., of Oxford. The authors would like to acknowledge the support of High Level Hardware, and of the UK Science and Engineering Research Council.

References

[1] James R. Bunch and Donald J. Rose, editors. *Sparse Matrix Computations.* Academic Press, 1976.

[2] Thomas F. Coleman. *Large Sparse Numerical Optimization.* Lecture Notes in Computer Science. Springer Verlag, 1984.

[3] Iain S. Duff and G.W. Stewart, editors. *Sparse Matrix Proceedings 1978.* SIAM, Philadelphia, 1979.

[4] H. Amano et al. $(SM)^2$: Sparse matrix solving machine. In *ACM Proc. 10th Symposium on Computer Architecture,* pages 213–220. ACM, 1983.

[5] H. Amano et al. $(SM)^2$-II : A new version of the sparse matrix solving machine. In *ACM Proc. 12th Symposium on Computer Architecture,* pages 100–107. ACM, 1985.

[6] High Level Hardware Ltd., Headington, Oxford, UK. *ORION Time Sharing Manual,* 1986.

[7] R.N. Ibbett and N.P. Topham. *Architecture of High Performance Computers,* volume 1. Macmillan, Basingstoke, Hampshire, UK, 1989.

[8] D.E. Knuth. *The Art of Computer Programming,* volume 1 - Fundamental Algorithms. Addison-Wesley, 2 edition, 1973.

[9] H.M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science,* 3:255–269, 1957.

[10] D. Morris and R.N. Ibbett. *The MU5 Computer System.* Macmillan, London, 1979.

[11] Ole Østerby and Zahari Zlatev. *Direct Methods for Sparse Matrices.* Lecture Notes in Computer Science. Springer-Verlag, 1983.