# Optimizing Sparse Matrix Vector Product Computations Using Unroll and Jam*

John Mellor-Crummey and John Garvin
Department of Computer Science
Rice University
Houston, TX 77005-1892

August 30, 2002

**Abstract**

Large-scale scientific applications frequently compute sparse matrix vector products in their computational core. For this reason, techniques for computing sparse matrix vector products efficiently on modern architectures are important. This paper describes a strategy for improving the performance of sparse matrix vector product computations using a loop transformation known as unroll-and-jam. We describe a novel sparse matrix representation that enables us to apply this transformation. Our approach is best suited for sparse matrices that have rows with a small number of predictable lengths. This work was motivated by sparse matrices that arise in SAGE, an ASCI application from Los Alamos National Laboratory. We evaluate the performance benefits of our approach using sparse matrices produced by SAGE for a pair of sample inputs. We show that our strategy is effective for improving sparse matrix vector product performance using these matrices on MIPS R12000, Alpha Ev67, IBM Power 3 and Itanium processors. Our measurements show that for this class of sparse matrices, our strategy improves sparse matrix vector product performance from a low of 11% on MIPS to well over a factor of two on Itanium.

## 1 Introduction

Sparse matrix vector product computations are at the heart of many modern scientific applications; however, they often perform poorly on modern computer systems. Poor performance results from several factors. First, unlike dense matrices, sparse matrices require an explicit representation of the coordinates of nonzero elements. Manipulating this coordinate representation when computing a sparse matrix vector product consumes memory bandwidth, which is a scarce commodity in computer systems based on modern microprocessors. Second, sparse matrix computations have less temporal reuse of values than dense matrix computations; this makes it hard to use registers and caches effectively. Third, the loop structure of a matrix vector product computation using a sparse matrix is significantly less regular than one using a dense matrix; this leads to less efficient compiler-generated code. Im and Yelick [11] report a factor of two performance penalty for computing a matrix vector product on dense matrices using a sparse matrix representation rather than a dense matrix representation.

The inefficiency of computing sparse matrix vector products has prompted research into strategies for improving their performance. Techniques that have been explored include development of a variety of sparse matrix representations, matrix reordering, employing multiple representations in concert, cache and register blocking, and combinations of these approaches. Section 5 presents a summary of these strategies.

*This research was supported in part by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23 as part of the prime contract (W-7405-ENG-36) between the DOE and the Regents of the University of California and the Department of Energy's Accelerated Strategic Computing Initiative under research subcontract B347884.

SAIC's Adaptive Grid Eulerian hydrodynamics code (SAGE) is a computationally-intensive, multi-dimensional, multi-material code that employs cell-by-cell adaptive refinement on a hexahedral mesh of cells [12]. In working with this code, we found that a significant fraction of its execution time was spent computing sparse matrix vector products. Furthermore, we found that SAGE's adaptive mesh refinement scheme yields sparse matrices with properties that make it difficult to multiply them efficiently with a vector on modern microprocessors. SAGE's sparse matrices typically have only a few elements per row and contain no patterns of local density, even after row and column reordering. For this reason, neither register or cache blocking algorithms were effective for improving sparse matrix vector product performance with SAGE's sparse matrices.

In this paper, we explore a new strategy for improving the performance of sparse matrix vector product computations for a class of matrices that includes those used by SAGE. We describe a new sparse matrix organization that enables us to optimize sparse matrix vector product computations by using a loop transformation known as unroll-and-jam [2]. We found that this data structure and computation reorganization improves performance from 11% to well over a factor of two on the architectures we studied.

The next section briefly describes SAGE, its representation for sparse matrices and the performance results that motivated our investigation of this problem. Section 3 describes our data and computation transformation for accelerating sparse matrix vector product computations for SAGE on modern microprocessor-based systems. Section 4 presents an experimental evaluation of our approach. Section 5 describes related work. Section 6 summarizes our findings and conclusions.

## 2 Background

### 2.1 SAGE

SAGE is a parallel, multi-dimensional, multi-material, Eulerian hydrodynamics code that employs cell-by-cell adaptive refinement of a hexahedral mesh of cells [12]. SAGE was developed by SAIC (Scientific Applications International Corporation) and Los Alamos National Laboratory (LANL) as part of the Department of Energy's Accelerated Strategic Computing Initiative (ASCI). In LANL's Crestone project, SAGE is being used to explore the utility of continuous adaptive Eulerian techniques for simulation of the nation's nuclear stockpile. SAGE represents a large class of ASCI applications at LANL that run for months at a time on 2000–4000 processors [12]. In addition to stockpile stewardship, SAGE is used for simulating a variety of phenomena that arise in scientific and engineering problem domains.

SAGE's computational core consists of a conjugant gradient solver that requires each processor to compute a sparse matrix vector product in each iteration. Each time the solver is invoked, a sparse matrix representation on each node is built from data structures that represent the arrangement of mesh cells. The cost of building the sparse matrix representation on each processor is insignificant with respect to the amount of time spent performing sparse matrix vector multiplication in the solver. On the problems of interest at LANL, the solver typically runs hundreds to thousands of iterations before its convergence criteria are satisfied. To guide our investigation of techniques for improving the node performance of sparse matrix vector product computations that arise in SAGE, we studied sparse matrices formed by SAGE in single-processor simulations of two test problems. We used the `timing_c` and `timing_b` test problems provided to us by scientists at LANL as the basis for our study. Both inputs trigger three-dimensional simulations of two materials using adaptation and heat conduction. The `timing_c` input triggers a simulation in a long, skinny domain and yields a symmetric sparse matrix of with 72,000 rows containing $467,568$ non-zero elements. The `timing_b` input triggers a smaller simulation in a cubical domain and yields a symmetric sparse matrix with 38,464 rows containing $246,634$ non-zero elements.

### 2.2 Compressed Sparse Row Format

SAGE uses a Compressed Sparse Row (CSR) representation for sparse matrices. Figure 1 illustrates the organization of a conventional CSR matrix representation. The matrix is represented by three vectors: `cols`, which contains all column indices of non-zeroes in the matrix (ordered by row), `nz`, which contains the non-zero value corresponding to each entry in `cols`, and `rowstart`, which indicates the position in `cols` and `nz`

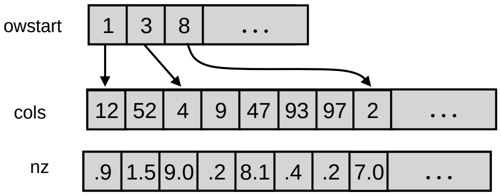

Figure 1: Compressed Sparse Row (CSR) format.

```
subroutine csr_mvmult(csr_nrows, csr_rowstart, csr_cols, csr_nz, x, result)
use define_kind
implicit none

! --------- arguments ----------
integer :: csr_nrows, csr_rowstart(*), csr_cols(*)
real(REAL8) :: csr_nz(*), x(*), result(*)

! ------- local variables ------
real(REAL8) :: tmp
integer :: row, col, firstcol, nextfirstcol

firstcol = csr_rowstart(1)
do row = 1, csr_nrows
   tmp = 0.0
   nextfirstcol = csr_rowstart(row+1)
   do col = firstcol, nextfirstcol-1
      tmp = tmp + csr_nz(col) * x(csr_cols(col))
   enddo
   result(row) = tmp
   firstcol = nextfirstcol
enddo

return
end subroutine csr_mvmult
```

Figure 2: Sparse matrix vector multiplication using CSR format.

of the first element of each row. On microprocessor-based systems, CSR format is commonly used for sparse matrix vector product computations. Since each row of the matrix in CSR format is contiguous, a register can be used to accumulate the result of the dot product of the row's non-zero entries with a vector.

Figure 2 shows a canonical procedure for performing sparse matrix vector multiplication using a matrix in CSR format. The first argument to the procedure `csr_mvmult` specifies the number of matrix rows. The next three arguments specify the sparse matrix data structure components `rowstart`, `cols`, and `nz` shown in Figure 1. The final two arguments specify an input vector `x` and a result vector `result`. The outer loop in `csr_mvmult` computes the length of the current row as the difference between the start of the next row and the start of the current row. The inner loop accumulates a dot product of the non-zeroes in the row and their corresponding entries in the vector `x`. In this procedure, the `rowstart`, `cols`, `nz` and `result` vectors are accessed in sequential order with spatial but not temporal reuse. The values of `x` are accessed in the order dictated by the sequence of values in `cols`; this order is potentially arbitrary and may result in spatial and/or temporal reuse of values in `x`.

|  | *Timing-C matrix* |  |  |  | *Timing-B matrix* |  |  |  |
|---|---|---|---|---|---|---|---|---|
| Row Length | Rows | % Rows | Non-zeroes | % Non-zero | Rows | % Rows | Non-zeroes | % Non-zero |
| 1 | 3,600 | 5 | 3,600 | 1 | 4,296 | 11 | 4,296 | 2 |
| 4 | 8 | < 1 | 32 | < 1 | 8 | < 1 | 32 | < 1 |
| 5 | 1,096 | 2 | 5,480 | 1 | 189 | < 1 | 945 | < 1 |
| 6 | 17,584 | 24 | 105,504 | 23 | 3,024 | 8 | 18,144 | 7 |
| 7 | 47,912 | 67 | 335,384 | 72 | 28,861 | 75 | 202,027 | 82 |
| 8 | 16 | < 1 | 128 | < 1 | 18 | < 1 | 144 | < 1 |
| 9 | 400 | < 1 | 3600 | < 1 | 276 | < 1 | 2,484 | 1 |
| 10 | 1,384 | 2 | 13,480 | 3 | 1,566 | 4 | 15,660 | 6 |
| 11 | 0 | 0 | 0 | 0 | 3 | < 1 | 33 | < 1 |
| 12 | 0 | 0 | 0 | 0 | 42 | < 1 | 504 | < 1 |
| 13 | 0 | 0 | 0 | 0 | 177 | < 1 | 2,301 | 1 |
| 16 | 0 | 0 | 0 | 0 | 4 | < 1 | 64 | < 1 |
| totals | 72,000 | 100 | 467,568 | 100 | 38,464 | 100 | 246,634 | 100 |

Table 1: Distribution of row lengths and non-zeroes in SAGE `timing_c` and `timing_b` examples.

## 2.3 Motivation

A preliminary investigation of the performance of sparse matrix vector product computations in SAGE found them to be quite slow, but suggested several promising directions for further investigation.

Version 20010624 of the SAGE code computed sparse matrix vector products in a curious way. Rather than indirectly accessing elements of the input vector `x` using column indices from `csr_cols` when performing the multiplication, the values of `x` were replicated into `x_nz`, a vector with length equal to the number of non-zeroes in the matrix. The Fortran 90 code fragment below shows this replication strategy.

```
allocate (x_nz(size(csr_cols)))
do i = 1, size(csr_cols)
  x_nz(i) = x(csr_cols(i))
enddo
```

Unlike `x`, which must be indexed indirectly during CSR sparse matrix vector multiplication, `x_nz` can be indexed directly in stride one order. Performance analysis of this approach on a MIPS R12000 and Alpha EV6.7 indicated that when the cost of replicating `x` into `x_nz` is combined with the cost of the sparse matrix vector product using the longer `x_nz`, the overall performance of this approach is a factor of two slower than simply performing CSR matrix vector multiplication as shown in Figure 2. After this result was communicated to the Crestone application team, they adjusted SAGE to perform the multiplication in the conventional way using CSR format, which roughly halved the cost computation and local copies in SAGE's conjugant gradient solver[8].

Performance measurements of the CSR-based sparse matrix vector product code shown in Figure 2 on several hardware platforms showed that it was less efficient than expected. An analysis of the assembly code generated by SGI's MIPSPro Fortran 90 compiler (version 7.3.1.3m, -O3 optimization) for this CSR-based sparse matrix vector product code showed that while memory references issue at 100% of the peak rate in the inner loop, in the outer loop they issue at only 41% of the peak rate. Using SAGE's `timing_c` sparse matrix, we found that on average a floating point operation completes every 7.2 cycles with this code on a MIPS processor. On an Itanium, performance is even worse, with a floating-point operation completing only every 13.6 cycles.

In an effort to understand the relative contribution of the outer loop to overall performance, we generated histograms of the matrix row lengths for the SAGE `timing_c` and `timing_b` test cases, as shown in Table 1. These measurements show that SAGE's sparse matrices have very short row lengths, implying that a significant fraction of time in the vector multiplication routine will be spent in the outer loop or in fill and drain code surrounding a software-pipelined inner loop. In the next section, we describe an approach that exploits the structure of these matrices to address this problem.

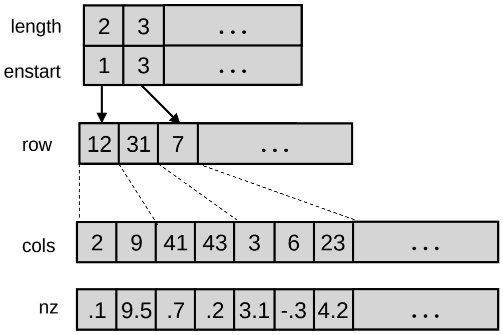

Figure 3: Length-grouped Compressed Sparse Row (L-CSR) format.

# 3 Approach

Besides the problem of short rows, the performance of CSR-based sparse matrix vector product code, as shown in Figure 2, suffers from a more fundamental problem: each multiply-add in the inner loop depends on the value computed by the previous iteration. Since floating-point operations on modern microprocessors are multi-stage pipelined operations, this dependence between iterations prevents the floating-point pipeline from filling, even with unrolling. To improve pipeline utilization in dense matrix codes that have a recurrence in the inner loop, the unroll-and-jam transformation [2] is often used.

Applying unroll-and-jam, also known as loop jamming, to a loop nest involves unrolling an outer loop and fusing the multiple inner loop bodies that result. This transformation has three potential benefits: (1) it increases the size of inner loop bodies to aid in optimization and reduce loop overhead, (2) it can increase floating point pipeline utilization by interleaving the computation of multiple independent recurrences [4], and (3) in dense matrix computations, it can increase close temporal reuse, some of which can then be converted into register reuse by applying scalar replacement [3, 5].

Unroll-and-jam, of course, cannot be applied if the inner loop bodies that result from unrolling cannot legally be fused. Thus, it cannot be employed directly to improve the efficiency of our CSR-based matrix vector product computation shown in Figure 2. If we unrolled the outer loop over rows, it would not be legal to fuse the multiple inner loop bodies that result because their rows may be of different lengths. However, the histograms of SAGE's sparse matrices shown in Table 1 show that besides having short rows, the rows generally have one of a small number of frequently occuring lengths. If we reorganized the matrix so that rows with the same length were together, then we could unroll and jam the sparse matrix vector product loop nest to interleave the product computations for a group of rows of the same length.

We introduce Length-grouped CSR (L-CSR), a new sparse matrix representation that enables us to apply unroll-and-jam to a CSR-style sparse matrix vector product. Figure 3 shows the structure of this representation. Like CSR format, L-CSR contains a pair of vectors `cols` and `nz` that contain, respectively, the column index and value of each non-zero value in the matrix. As in CSR, a matrix row is represented by a contiguous sequence of entries in `cols` and `nz`. However, in L-CSR, rows are grouped by length, whereas in CSR rows appear in normal matrix order. In L-CSR, the `row` vector and a vector of `(length,lenstart)` pairs substitute for the vector `rowstart` in CSR. Each entry in `row` indicates the matrix row that corresponds to a contiguous sequence of elements in `cols` and `nz`. A `(length,lenstart)` pair represents a group of rows of a particular length; `length` is the row length and `lenstart` is the index in `row` where rows of that length begin. The offset of the first row in a length group in `cols` and `nz` can be determined from the length and number of rows of in length groups to the left of the current group in the pair vector.

Figure 4 shows machine-generated code for performing a sparse matrix vector multiply using L-CSR format. It exploits the L-CSR representation by applying unroll-and-jam. The code shown was generated using an unroll-and-jam factor of 2. Lines 32–43 show unroll-and-jammed code for handling pairs of rows at a time. Interleaving the computation for multiple rows leads to better floating point pipeline utilization. Lines 47–54 show code for any leftover rows that can't be handled in bundles in the unroll-and-jammed loop. When rows of some particular short length occur with high frequency, it may be more efficient to entirely

```
subroutine csrl_mvmult(csrl_nlengths, csrl_lengths, csrl_rowindex, csrl_cols, csrl_nz, x, r)
use define_kind
implicit none
! --------- arguments ----------
integer :: csrl_nlengths, csrl_lengths(2,*), csrl_rowindex(*), csrl_cols(*)
real(REAL8) :: csrl_nz(*), x(*), r(*)
! ------- local variables ------
integer, parameter :: ROWLEN=1, GROUPSTART=2
integer :: col
integer :: len, row, rowlen, rowofnextlen
integer :: firstrow, lastrow, i
real(REAL8) :: val0, val1
col = 1
do len = 1, csrl_nlengths
  firstrow = csrl_lengths(GROUP_START,len)
  rowlen = csrl_lengths(ROW_LENGTH,len)
  lastrow = csrl_lengths(GROUP_START,len+1) - 1
  if (lastrow .ge. firstrow + 1) then
    if (rowlen .eq. 2) then ! unrolled code for 2 rows of length 2
      do row = firstrow, lastrow - 1, 2 ! compute dot product for 2 rows of length 2
        val0 = csrl_nz(col) * x(csrl_cols(col))
        val1 = csrl_nz(col + 2) * x(csrl_cols(col + 2))
        col = col + 1
        val0 = val0 + csrl_nz(col) * x(csrl_cols(col))
        val1 = val1 + csrl_nz(col + 2) * x(csrl_cols(col + 2))
        col = col + 1
        r(csrl_rowindex(row)) = val0
        r(csrl_rowindex(row + 1)) = val1
        col = col + 2
      enddo
    else
      do row = firstrow, lastrow - 1, 2 ! handle 2 rows at a time
        val0 = 0.0
        val1 = 0.0
        do i = 1, rowlen ! compute dot product for 2 rows of any length
          val0 = val0 + csrl_nz(col) * x(csrl_cols(col))
          val1 = val1 + csrl_nz(col + rowlen) * x(csrl_cols(col + rowlen))
          col = col + 1
        enddo
        r(csrl_rowindex(row)) = val0
        r(csrl_rowindex(row + 1)) = val1
        col = col + 1 * rowlen
      enddo
    endif
    firstrow = row
  endif
  do row = firstrow, lastrow ! for any number of rows
    val0 = 0.0
    do i = 1, rowlen ! compute dot product for a row of any length
      val0 = val0 + csrl_nz(col) * x(csrl_cols(col))
      col = col + 1
    enddo
    r(csrl_rowindex(row)) = val0
  enddo
enddo
end subroutine csrl_mvmult
```

Figure 4: Machine-generated sparse matrix vector multiplication using LCSR format with unroll-and-jam factor of 2 and special case fully-unrolled code for rows of length 2.

unroll the inner loop over elements in a row. Lines 20–30 show special-case code for handling rows of length 2. In general, special-case, fully-unrolled code can be generated for any small set of short row lengths.

To generate a tailored code for L-CSR matrix vector product for SAGE's sparse matrices, we generated special-case code for rows of length 1, 6, 7 and 10. Table 1 shows us that in SAGE's sparse matrices, these lengths account for over 97% of the rows and also contain 97% of the non-zeroes. The high frequency of these lengths in our two sample matrices is not an accident. In SAGE's hexahedral mesh, a typical cell depends upon itself and a pair of adjacent cells along each coordinate dimension; this results in a sparse matrix row of length 7. Rows of length less than 7 correspond to cells in various configurations along the domain boundary. Rows of length ten correspond to a typical cell that has an extra three neighbors on one side because a neighboring cell has been cut into octants by adaptation. In the next section, we compare the performance of CSR matrix vector product with code for an L-CSR matrix vector product using an unroll-and-jam factor of 4 (it handles bundles of 5 rows at a time) with special case code for rows of length 1, 6, 7 and 10.

There are three disadvantages of using an L-CSR representation. First, L-CSR format is not a random-access representation: the index in `cols` and `nz` where each row starts is computed incrementally from the start position and length of the prior row. Second, since rows are reordered by length in L-CSR format, the reordering could reduce access locality of `x` since it will tend to move non-zeroes away from the diagonal of the sparse matrix (assuming that the matrix was ordered so they were near the diagonal originally). Third, writes into `result` vector using L-CSR format may no longer be stride one. Our experiments in the next section show that for SAGE's sparse matrices, the benefits of unroll-and-jam and special-case full unrolling using L-CSR outweigh these potential costs.

# 4 Experimental Results

In this section, we describe an experimental evaluation of sparse matrix vector multiplication using CSR and L-CSR formats of sparse matrices from SAGE's `timing_b` and `timing_c` sample problems. In Section 4.1, we describe our measurement methodology. In Section 4.2, we briefly describe the hardware platforms used for our experiments. Our findings are presented in Sections 4.3 and Section 4.4, which, respectively, describe experiments on a collection of hardware platforms and results of detailed simulations.

## 4.1 Methodology

On each hardware platform, we measured 100 iterations of sparse matrix vector multiplication using each strategy. We took certain precautions to ensure that our measurements were not contaminated by confounding effects. First, cache conflicts can be exacerbated when arrays used in the same loop are placed in non-consecutive memory locations. To avoid this problem, for each experiment we allocated all components of a sparse matrix data structure, along with the input and output vectors that would be used with them, at the same point and placed them consecutively in memory. Second, the first iteration of a loop performing a memory-intensive computation may suffer performance penalties for faulting data into caches from memory and priming the translation lookaside buffer (TLB) with necessary page translations. To ensure that such effects didn't skew our results, we performed one iteration of a sparse matrix vector product computation to warm the memory system immediately before performing a timing test on multiple iterations of the computation.

Experiments that attempted to exploit the symmetry of symmetric sparse matrices to improve performance yielded lower performance than L-CSR. We explored a sparse symmetric representation that explicitly represented only the lower or upper triangular portion of a sparse matrix in CSR format. Sparse matrix vector multiply with such a representation incurs additional overhead because when processing each `(row, col)` pair, it must load, update, and store `result(col)` as well as `result(row)`. This has the overall effect that each entry in the result vector is updated multiple times rather than computed in its entirety and stored only once, yielding lower performance.

Sparse matrix vector multiplication involves computing the dot product of each row of the matrix and the input vector. In the most straightforward case, this operation involves one floating-point multiplication and one addition per row element. We computed "useful floating point operations per second" as twice the number of non-zeroes in the matrix times the number of iterations, divided by the CPU time taken by the matrix multiplication. This counts the rate of essential floating-point instructions while disregarding non-productive operations such as register to register copies of floating point values. The higher the useful floating-point operations per second, the more efficient the matrix vector product computation.

For comparison with CSR and L-CSR strategies, we also measured sparse matrix vector product computations using Im and Yelick's Sparsity package for register and cache blocking [11, 10]. Our study of Sparsity using SAGE's `timing_c` matrix showed that Sparsity's exhaustive two-dimensional search of tile sizes (register blocking from 1x1 to 12x12 and cache blocking in powers of two from 1x1 to 65536x65536) found no cache or register blocking size that improved performance. We believe that cache and register blocking were ineffective for SAGE's sparse matrices because they lack patterns of local density; the majority of non-zero values in the matrices are clustered near the diagonal, with a few rows along outlying diagonals.

We also compared our results with that of IBM's sparse matrix vector multiplication routine DSMMX from its Engineering and Scientific Subroutine Library (ESSL). In DSMMX, the sparse matrix is stored in a form known as the Ellpack-Itpack format [14]. In this form, the matrix is stored in a pair of two-dimensional panels whose major dimension size is the number of rows and whose minor dimension size is the maximum number of non-zeroes per row. One panel holds the column indices; the second holds the non-zero values. The fixed width of the panels is an advantage, but if most rows are not near the maximum length, the large number of zero values in the panel leads to inefficient memory usage and computation. Results obtained with DSMMX are presented in Section 4.3.

## 4.2 Hardware Platforms

Below we enumerate architectures used in our experiments. On each platform, we performed experiments on a single processor.

- *Compaq ES40 containing four 667MHz Compaq Alpha EV6.7 (21264A) processors.* An EV6.7 processor can issue up to four instructions per cycle, which may include at most two memory accesses and one floating-point multiply-accumulate. Floating-point arithmetic operations have a 4-cycle latency. The ES40's memory hierarchy consists of a 64KB 2-way set associative primary cache and an 8-MB Dual Data Rate cache. Primary cache latency is three cycles for integer loads and four cycles for floating-point loads, while the latencies to the secondary cache and memory are 12 and 80 cycles, respectively. Code was compiled Compaq C V6.3 with optimizations "-O3 -arch ev67". Performance results were measured using `uprofile`.

- *IBM SP 2 containing 16 375MHz IBM Power3-II processors.* Each Power3-II can issue 8 instructions per cycle including at most 2 memory accesses and 2 floating-point instructions. Each processor has a 64KB L1 data cache and an 8MB L2 cache. Code was compiled IBM's xlf90 and xlc compilers using optimization level -O5. Performance results were measured using the PAPI interface to hardware performance counters.

- *SGI Origin 2000 containing 16 300MHz MIPS R12000 processors.* Each node in the system contains a pair of processors, which share an 8MB unified secondary cache. Each R12000 can issue four instructions per cycle, which may include at most one memory access and one floating-point multiply-accumulate. Each processor has a 32KB 2-way set associative primary cache. Latency penalties for L1 and L2 misses are approximately 8 and 100 cycles each. Code was compiled with MIPSpro Fortran and C V7.3.1.3m using optimizations "-O3 -mips4 -r12000". Performance results were measured with SpeedShop.

- *A single-processor Itanium workstation.* A first-generation Itanium can execute one three-instruction VLIW bundle per cycle. It has a 32Kb 4-way set-associative L1 cache and a 96K 6-way set-associative L2 cache. Code was compiled with Intel's efc and ecc with optimization level "-O3". Performance results were measured using the PAPI interface to hardware performance counters.

|  | *Cycles* |  |  | *Graduated Instructions* |  |  | *Graduated FP Instructions* |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  | *CSR* | *L-CSR* | *% decrease* | *CSR* | *L-CSR* | *% decrease* | *CSR* | *L-CSR* | *% decrease* |
| *Itanium* | 1.36e+9 | 5.97e+8 | 56.0% | 1.27e+9 | 5.40e+8 | 57.6% | 1.01e+8 | 8.65e+7 | 14.1% |
| *MIPS* | 3.35e+8 | 2.99e+8 | 10.9% | 5.40e+8 | 3.17e+8 | 41.3% | 4.67e+7 | 4.67e+7 | 0.154% |
| *Alpha* | 4.96e+8 | 2.97e+8 | 40.2% | 5.71e+8 | 3.20e+8 | 40.0% | - | - | - |
| *Power3-II* | 3.37e+8 | 1.96e+8 | 41.9% | 5.53e+8 | 2.59e+8 | 53.1% | 1.03e+8 | 4.75e+7 | 53.9% |

Table 2: Performance of sparse matrix vector multiplication using SAGE's `timing_c` sparse matrix in Compressed Sparse Row format and Length-grouped Compressed Sparse Row format. Results are for 100 iterations.

|  | *Timing-B matrix* |  | *Timing-C matrix* |  |
|---|---|---|---|---|
|  | *CSR* | *L-CSR* | *CSR* | *L-CSR* |
| *Itanium* | 69 | 160 | 55 | 123 |
| *MIPS* | 94 | 109 | 86 | 96 |
| *Alpha* | 160 | 254 | 140 | 208 |
| *Power3* | 118 | 235 | 103 | 177 |

Table 3: Millions of useful floating-point operations per second achieved for SAGE `timing_b` and `timing_c` test cases.

### 4.3 Hardware Performance Results

Table 2 compares raw measurements from hardware performance counters on each architecture for 100 iterations of a sparse matrix vector product using SAGE's `timing_c` matrix in CSR or L-CSR format. The L-CSR code used in these experiments was tailored for SAGE's matrices as described in Section 3. Our results show that using an L-CSR sparse matrix vector multiply improves performance on all hardware platforms. The smallest improvement was roughly 11% on the MIPS R12000. On the Alpha and Power3-II, improvements of 40% were measured. On the Itanium performance improved by more than a factor of 2. Table 2 shows that the L-CSR code executed 40–58% fewer instructions than the CSR code. Some of this was a reduction in useless floating point register-to-register copies. In the next section, we present simulations that show that a significant part of the reduction comes from index arithmetic.

Table 3 reports useful MFLOP rates (excluding useless register-to-register copies) measured on each of the platforms. These results show that sparse matrix vector multiplication using the L-CSR format substantially improved performance on all the platforms we studied for both the `timing_b` and `timing_c` test cases. Although the previous table compared detailed performance counter measurements only for SAGE's `timing_c` sample input, this table shows that improvements with SAGE's `timing_b` matrix are even better, yielding a factor of two improvement on Power3 and a factor of 2.3 on Itanium. The improvement on Alpha is 58%. Only the MIPS fails to achieve a substantial boost, improving only 16%. Using IBM's ESSL sparse matrix vector product routine DSMMX, on the Power3 we achieved only 46 MFLOPS with `timing_c` and 40 MFLOPS with `timing_b`. Both CSR and L-CSR routines dominated this performance by a wide margin.

We performed a series of experiments to evaluate the impact of (1) loop unrolling to interleave computation of multiple rows and (2) fully-unrolled special case code for short, common row lengths. These results are shown in detail in Table 4. We evaluated unrolling factors of 1 (no unrolling), 2, 4, 6, and 8, with and without special case code for common row lengths. The results show that both optimizations yielded important performance improvements. Loop unrolling to interleave computation over multiple rows improved performance by a maximum of 25% on the Itanium and by a maximum of 17% on the Alpha. This unrolling enabled the recurrences of different rows to be interleaved, which reduced memory stalls. Special-case fully-unrolled code further improved performance. On the Alpha, performance with loop unrolling and special-case code improved by a maximum of 29% over plain unrolled code. On the Itanium, the imorovement was especially dramatic–a maximum of 116% over plain unrolled code. This full unrolling of short, common

| | *No Special-case Code* | | | | | *Special-case Code* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Loop body copies | 1 | 2 | 4 | 6 | 8 | 1 | 2 | 4 | 6 | 8 |
| *Itanium MFLOP/s* | 64 | 80 | 74 | 68 | 74 | 137 | 137 | 160 | 137 | 120 |
| *Alpha MFLOP/s* | 205 | 240 | 237 | 218 | 200 | 238 | 250 | 244 | 258 | 258 |

Table 4: Effect of special-case code (full loop unrolling for common row lengths) and interleaving computation on multiple rows for L-CSR format matrix vector multiplication with the `timing_c` matrix.

| | *CSR* | *L-CSR* |
|---|---|---|
| *ALU* | 3,634 | 849 |
| *Write* | 72 | 73 |
| *Read* | 1,475 | 1,479 |
| *Branch* | 684 | 16 |
| *FPU* | 1,007 | 866 |
| *Total* | 6,871 | 3,282 |

Table 5: Instructions counts of different types in the RSIM simulation for one iteration, in thousands of instructions.

row lengths eliminated loop overhead and provided larger blocks of code for the compiler to optimize. Full unrolling of small, short-trip loops was especially beneficial on the Itanium. On the Itanium, filling and draining of software pipelines are done in the main loop body, using conditional execution, rather than in shorter, separate blocks before and after the loop. Since filling and draining take as much time as several iterations of the main loop, fill and drain overhead cause major inefficiencies in software-pipelined loops with low trip counts.

## 4.4 Simulation Results

We used the Rice Simulator for ILP Multiprocessors (RSIM) [15] to measure the dynamic flow of instructions in the processor and to get more detailed data about memory system utilization. RSIM is an execution-driven simulator that uses executable files compiled for the SPARC architecture. It models both superscalar processor behavior and multi-level memory hierarchies to a high degree of detail. The RSIM processor model takes advantage of instruction-level parallelism (ILP) in a way similar to modern superscalar processors. RSIM is commonly used to model multiprocessor systems; we used it in single-processor mode. We used the default RSIM parameters for single-processor simulation. This configuration includes two ALUs, two FPUs, and two address generation units; four instruction retire slots; and eight register windows. The memory system includes a 16kB write-through L1 cache with 1-cycle latency, 64kB four-way set-associative L2 cache with 5-cycle latency, and DRAM with 18-cycle latency. These are generally consistent with a 300-MHz SPARC processor. RSIM enabled us to precisely measure dynamic instruction counts, functional unit usage, and memory performance.

Table 5 shows the frequency of types of instructions comparing simulations of a single iteration of CSR and L-CSR on SAGE's `timing_c` matrix. Our simulation results show a dramatic reduction in the number of total instructions executed. Table 5 shows that ALU instructions account for more than 50% of the total instructions in the RSIM simulation of CSR multiplication. In matrix vector product computations, only floating-point and and memory operations are useful work. ALU instructions are overhead that can be eliminated. L-CSR multiplication graduated 72% fewer ALU instructions than CSR. Floating-point and branch instructions were also eliminated. The principal difference in floating point operations observed here was due to the lack of floating point register copies in L-CSR and slightly fewer useful floating point operations needed (as explained in Section 4.1).

Reductions in total instruction count do not necessarily improve running time. In a processor with instruction-level parallelism, code with many instructions that can be parallelized may actually execute

| | *CSR* | *L-CSR* |
|---|---:|---:|
| *ALU stall* | 39 | 1 |
| *Write stall* | 0 | 1 |
| *Read stall* | 2,345 | 2,476 |
| *Branch stall* | 31 | 0 |
| *FPU stall* | 457 | 84 |
| *Busy* | 1,718 | 820 |
| *Total stall* | 2,872 | 2,562 |
| *Total* | 4,590 | 3,384 |

Table 6: Breakdown of time spent in the RSIM simulation, in thousands of cycles, for one iteration.

faster than code with a few non-parallelizable instructions. To understand how our simulations differ in performance, we measured detailed cycle usage in addition to instruction counts. RSIM divides the processor time into parts: time spent handling different kinds of *stalls*, such as cache misses and interlocks, and *busy time* spent graduating instructions at the processor's maximum rate. Table 6 shows the breakdown of time spent during each simulation.

Our intuition was that the performance of sparse matrix vector multiplication should be limited by memory stalls since there is no temporal locality in loading elements of the sparse matrix. In Table 6, we see that read stalls are the dominant stall cost. The memory stall cost measured for L-CSR was 10.9% than that for CSR. A slight increase was expected. CSR achieves good spatial locality for the input vector as the multiplication progresses through the rows because most of the non-zero values are near the diagonal of the matrix. This locality property can be destroyed by the length-based row reordering performed by L-CSR. In measurements of memory hierarchy utilization on our hardware systems, we found that L-CSR decreases the rate of first level and second level cache misses but increases the rate of third level cache misses and TLB misses on the Itanium. On the MIPS, L-CSR increases the rate of translation lookaside buffer (TLB) misses and both primary and secondary cache misses relative to CSR.

Table 6 shows that "busy time" for CSR is more than double that for L-CSR. Busy time is a sign that the processor is bogged down with ALU overhead, causing the memory system to be used at less than full capacity. In CSR multiplication, the processor was busy 37.43% of the time. L-CSR reduced the busy time by 74% while decreasing total stall time by 7%. The reduction of instructions issued and the dramatic reduction in busy time seems to results from the fact that L-CSR code presents larger loop bodies to the compiler, revealing more opportunity to reduce arithmetic overhead.

L-CSR reduced the number of floating-point instructions as well as ALU instructions. In the SPARC/RSIM assembly code for CSR sparse matrix vector multiplication, the floating point operations include the following: (1) for each row, loading a zero into an accumulator register, (2) for each row element, multiplying an element of the matrix by an element of the input vector, (3) for each element, adding the product computed in step 2 to the accumulator register, and (4) performing register-to-register moves. In L-CSR, since the row length is known ahead of time, the accumulator register can be initialized with the first value of each row instead of zero. In this way, steps 1 and 3 can be completely eliminated for the first element of each row. For matrices with small average row lengths, the elimination of a few instructions per row can be quite significant. The unroll-and-jam optimization enabled by L-CSR can also eliminate many of the register-to-register move instructions in step 4, though this effect was negligible on the SPARC/RSIM platform. The details of eliminating floating-point operations differed greatly across different platforms. On the MIPS, almost no floating-point instructions could be eliminated. On the Power3, however, over 50% of the floating-point instructions could be eliminated; most were register-to-register moves.

Branch prediction in CSR was good enough to nearly eliminate branch stall time. Therefore, while L-CSR's unroll-and-jam in the outer loop and full unrolling of certain inner loops eliminated the vast majority of CSR's branch instructions, the impact of this improvement was negligible in processor time analysis.

In addition to reducing the number of instructions, L-CSR improves performance by overlapping instructions to fill the floating-point pipeline. With the unroll-and-jam optimization, floating-point operations in several consecutive rows can be interleaved, reducing floating-point stalls. In RSIM, floating-point stall time was reduced by 82%.

# 5 Related Work

A variety of different data and computation reorganization strategies have been explored in an effort to increase the performance of sparse matrix vector product computations.

For matrices where the maximum number of non-zeroes in any row is not large and the number of non-zeroes per row fairly uniform, the Ellpack-Itpack format [14], which collapses sparse matrices into a pair of dense matrices of size number of rows (`nrows`) by the maximum number of non-zeroes (`maxnz`) per row; one matrix contains coefficients and the other column indices. If the number of non-zeroes per row is not fairly uniform, then this representation can waste a lot of computation and memory bandwidth on zero-fill entries.

Paolini and Radicati di Brozolo [16] and Saad [17] proposed similar variants of the Ellpack-Itpack data structure known as jagged diagonals. This data structure reorders rows by length and then stores a dense vector with a column entry from each row. Each column goes from the first (longest) row down until the row just prior to the one in which the first non-zero appears in this column. While this representation facilitates vectorization along columns, for sparse matrix vector multiply it sacrifices reuse of the output vector, which it loads and updates repeatedly as each diagonal is processed.

Agarwal, Gustavson, and Zubair [1] proposed a feature extraction based algorithm to extract certain kinds of regular structure so it can be processed separately in a more efficient manner. They dissect the matrix into three separate representations: one for nearly dense blocks, one for nearly dense diagonals, and one for the remaining non-zeroes. Once the nearly dense blocks and diagonals are extracted, the remaining rows of non-zero elements are sorted by row length and stored in a "ladder scheme" matrix format similar to the Ellpack-Itpack format. Unlike Itpack, however, this format reduces zero padding by partitioning the rows with a constant number of rows per partition. The width of each partition is the maximum row length within the partition. Thus, the ladder scheme may still result in excessive zero padding when the difference between row lengths within a partition is large, though the amount of padding is almost always much less than that of Itpack.

Das et. al. [6] applied a breadth-first strategy known as Reverse Cuthill-McKee [7] to reorder elements in an unstructured mesh. Applying this reordering, originally devised to minimize the bandwidth and profile of sparse matrices, has the effect of improving the locality of indirect accesses to the input vector when computing its product with the sparse matrix. The reordering improves both cache and TLB locality.

Im and Yelick designed the Sparsity system, which tiles the matrix into uniform block sizes rather than searching for dense blocks within a matrix. An exhaustive search with speed profiling is used to find the optimal block size. This performs well for some patterns of non-zeroes, but tiling is less effective on matrices with irregular non-zero patterns.

Toledo used a combination of bandwidth reduction, blocking, and prefetching to improve performance of sparse matrix vector multiplication on RISC processors. A blocking strategy is used in which the matrix is scanned for completely dense 1x2 and 2x2 blocks. The intent was to find as many opportunities for blocking as possible, even with small blocks, rather than finding a small number of large semi-dense blocks. This strategy reduces the number of load instructions required, since some row and column index loads and input vector loads are made unnecessary. The number of arithmetic instructions, however, remains the same or increased with this system. For matrices with small average row length, the large number of arithmetic instructions prevents the memory system from being used to full capacity. It seems likely, therefore, that a blocking strategy such as Toledo's combined with length reordering could reduce both arithmetic and memory instructions, perhaps improving performance beyond either approach taken alone.

Keyes et. al. [9] describe a strategy for improving performance for a code that needs to multiply a sparse matrix by multiple independent vectors. By multiplying a sparse matrix by multiple independent vectors at a time, the sparse matrix data structure components can be reused in registers rather than streaming them through the memory hierarchy once per vector. Like our unroll-and-jam strategy, interleaving the computations on multiple independent vectors boosts floating point pipeline efficiency.

# 6 Conclusions

Previous research has shown that sparse matrix vector product computations can be accelerated by exploiting local patterns of density to enable register blocking [18, 10, 11]. Register blocking strategies for sparse matrix vector product computations reduce the need to load indirection vectors and provide temporal register reuse of both non-zero matrix elements and the corresponding vector elements by which they are multiplied. Memory hierarchy blocking for cache and TLB reuse has also been shown to be profitable [10].

In this paper, we described a new technique for accelerating sparse matrix vector product calculations for matrices in which rows occur in a small number of distinct lengths. Matrices with this property are characteristic of those that arise in SAGE, an important code that uses cell-by-cell adaptive refinement of a structured hexahedral mesh. Reorganizing the matrix representation to group together rows of identical length enabled us to apply unroll-and-jam to the sparse matrix vector product computation. This format yielded performance improvements from 11% to a factor of 2.3 for the platforms we tested.

Preliminary experiments on general sparse matrices from the NIST matrix market catalogue [13] showed that having a high frequency of rows with a small number of fixed lengths is not a common property. Thus, we expect that our strategy will be most useful for matrices that arise out of codes like SAGE that perform structured adaptive mesh refinement. Despite this somewhat narrow range of applicability, our technique is significant because of its benefits for long-running adaptive computations with SAGE and SAGE's applicability to a broad class of problems.

# References

[1] R. Agarwal, F. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix vector multiplication. In *Proceedings of Supercomputing '92*, Minneapolis, MN, Nov. 1992.

[2] F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[4] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, Aug. 1988.

[5] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[6] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562. In *Proceedings of the 30th Aerospace Sciences Meeting*. AIAA, Jan. 1992.

[7] A. George and G. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[8] M. Gittings. Personal communication, Mar. 2002.

[9] W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Improving the performance of sparse matrix-vector multiplication by blocking. Talk presented at SIAM Annual Meeting, July 2000, San Juan, Puerto Rico. Available as `http://www.icase.edu/~keyes/multivec.pdf`.

[10] E.-J. Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, University of California Berkeley, May 2000.

[11] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In V. N. Alexandrov, J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan, editors, *Proceedings of International Conference on Computational Science*, volume 2073 of *Lecture Notes in Computer Science*, pages 127–136. Springer, 2001.

[12] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Supercomputing 2001*, Denver, CO, November 2001.

[13] N. I. of Standards and Technology. Matrix market. `http://math.nist.gov/MatrixMarket`.

[14] T. Oppe, W. Joubert, and D. Kinkaid. NSPCG user's guide. Technical report, Center for Numerical Analysis, The University of Texas at Austin, Dec. 1988.

[15] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997. Also appears in IEEE TCCA Newsletter, October 1997.

[16] G. Paolini and G. R. di Brozolo. Data structures to vectorize CG algorithms for general sparsity patterns. *BIT*, 29:703–718, 1989.

[17] Y. Saad. Krylov subspace methods on supercomputers. Technical Report 88.40, Research Institute for Advanced Computer Science, NASA Ames Research Center, Dec. 1988.

[18] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–725, 1997.