Steps Toward Simplifying Sparse Matrix Data Structures

Stephanie Dinkins
Colorado State University, Fort Collins
Fort Collins, CO, USA
(970) 491-6275
sdinkins@cs.colostate.edu

Barbara Kreaseck
La Sierra University
Riverside, CA, USA
(951) 785-2549
kreaseck@lasierra.edu

Michelle Mills Strout
Colorado State University, Fort Collins
Fort Collins, CO, USA
(970) 491-4193
mstrout@cs.colostate.edu

Abstract

Sparse matrix computations occur in many scientific applications. In general, performance optimizations for sparse computations tend to result in the use of complex and/or new sparse matrix data structures. One common sparse matrix computation is Sparse Matrix Vector multiply (SpMV). The Optimized Sparse Kernel Interface (OSKI) auto-tuner developed by the Berkeley Benchmarking and Optimization Group (Bebop) performs a number of run-time reordering transformations like cache blocking and register tiling for given sparse matrices. Although OSKI has ostensibly solved the performance problem for SpMV, the solution is specific to SpMV and introduces some complex sparse matrix data structures. In this paper, we present a more general and simpler approach that enables the specification of sparse matrix computations using the easiest sparse matrix data structure coordinate storage and then uses a new optimization called the sparse loop optimization to introduce a compressed sparse block row data structure into the innermost loop. We show that a prototype of this more general and simpler approach results in the same performance as OSKI's cache blocking.

Keywords sparse matrix vector multiply, coordinate storage (COO), compressed sparse row (CSR), polyhedral framework, run-time reordering transformations

1. Introduction

Sparse Matrix Vector multiply (SpMV) is a common computation found in many scientific applications. Optimizing performance of this sparse computation is complicated by machine architecture and the necessity to manipulate sparse data structures [4]. The performance of sparse computations depends greatly upon data locality and the sparsity and non-zero structure of the matrices, which are not known until runtime. In this paper we explore optimizing SpMV with cache blocking and a sparse loop optimization.

The SpMV performance problem is well studied in academia and high performance has been achieved using performance optimization tools like the Optimized Sparse Kernel Interface (OSKI) [1] auto-tuner developed by the Berkeley Benchmarking and Optimization Group (Bebop). One of the sparse matrix optimizations available in OSKI is cache blocking. As with the other optimizations performed by OSKI, the cache blocking optimization is specific to SpMV and requires special sparse matrix data structures to implement. When OSKI tunes a sparse matrix for cache blocking, it splits the original Compressed Sparse Row (CSR) matrix into a list of cache blocked submatrices. Each submatrix is accessed using a CSR that is generated for each submatrix structure.

The objective of our research is to show that general sparse computations can be specified using a simple sparse matrix data structure called coordinate storage (COO), transformed using the Sparse Polyhedral Framework (SPF) [5], and then automatically optimized with various data reorderings, and a sparse loop optimization. The SPF represents SpMV as follows:

```
for (p=0; p<nz; p++) {
  y[row[p]] += val[p] * x[col[p]];
}
```

where `y` represents the output vector, `val[p]`, `row[p]`, and `col[p]` represent the `p`th non-zero in a sparse matrix, and `x` represents the input vector. Cache blocking on SpMV can be expressed with the following run-time reordering transformation:

$$\{[p] \rightarrow [b, r, p] \mid b = cacheblock(p) \wedge r = row(p)\}.$$

The resulting unoptimized code is as follows:

```
for (b=0; b<ncb; b++) {
  for (r=0; r<n; r++) {
    for (p=0; p<nz; p++) {
      if (b==cacheblock[p] && r==row[p]) {
        y[row[p]] += val[p] * x[col[p]];
      }
    }
  }
}
```

The code after the `val` and `col` arrays have been reordered to match the cache block order and sparse loop optimization will be as in Figure 2 where `pptr`, `val`, and `col` represent a Compressed Sparse Block Row (CSBR) data structure. This differs from a CSR format in that the `pptr` array indexes by block and row to produce an index into the `val` and `col` arrays.

The merit of using a framework such as SPF to transform SpMV is that the framework is applicable to any sparse computation and does not require the users to be exposed to a lot of specialized sparse matrix data structures. The key is that the computations are originally expressed using iteration over non-zeros [3], then reordering transformations are specified and applied, and finally various cleanup optimizations like sparse loop optimization are applied to the innermost loop to ensure that the non-zeros in the innermost loop are stored sequentially and in a generalized CSR data structure that instead of just being indexed by row can be indexed by any number of outer loop variables. Our CSBR data structure is an example of a generalized CSR data structure that can be indexed by block and row.

CCWIC '10 5 November 2010, Denver, Colorado, USA

In this paper, we evaluate our approach based on SpMV performance benchmarks that compare the execution times of hand-prototypes of SpMV with the simple cache blocking specification and cleanup optimizations to those from OSKI's cache block tuned implementation of SpMV. We show that we are able to match the performance, while exposing the programmer to less complexity.

Section 2 describes the OSKI cache blocked SpMV, and our cache blocked with sparse loop optimized SpMV algorithms. Section 3 describes our methodology and presents our results, and Section 4 concludes.

2. Sparse Matrix Vector Multiply

Sparse Matrix Vector multiply (SpMV) is a computation commonly found in many scientific applications. The SpMV function of interest is expressed as

$$\vec{y} = \beta\vec{y} + \alpha A\vec{x}$$

where $\alpha = \beta = 1$ for the purpose of simplifying our discussion. The performance of sparse computations depends greatly upon data locality and the sparsity of the matrices that are not known until runtime.

2.1 OSKI Cache Blocked SpMV

When OSKI tunes a sparse matrix to be cache blocked, it generates a linked list of r x c sparse, cache block submatrices that collectively represent the original sparse matrix. The conversion algorithm proceeds as follows. Given a specified r x c block size, the inspector walks through the matrix partitioning the data into cache blocks. The start of each cache block begins with the first non-zero in a column. Empty columns are passed over by the inspector and not included in the beginning of a cache block. Once the matrix is partitioned into a list of cache blocks, a sparse submatrix is generated for each cache block portion of the original matrix.

The OSKI cache blocked SpMV algorithm is given in pseudocode in Figure 1. For each sparse, cache block submatrix in the list, a starting location within X and Y are determined and an SpMV routine specific to the format of the submatrix is called to perform the calculations on the submatrix. When the submatrix format is a CSR data structure, it uses a *ptr* array to map to the relative location, p, of the first non-zero of each row. The column of the non-zero is stored at location p in the *ind* array and the value of the non-zero is stored at location p in the *val* array.

2.2 Cache Blocked and Sparse Loop Optimized SpMV

In contrast to the OSKI approach where cache blocking is implemented with a library routine that is specific to SpMV and then the OSKI user specifies to use cache blocking with a high-level interface, we propose a more general approach where sparse computations like SpMV are specified using a flat sparse data structure like COO. Transformations like cache blocking are specified in the more general Sparse Polyhedral Framework, and then a code generator will be responsible for automating optimizations. Instead of generating a CSR for each cache block we use a CSBR data structure, which stores the re-ordered non-zero and column entries and a single, two-dimensional pointer array to access each row of each cache block. The pointer array works similar to the row-pointer array in the CSR data structure used in OSKI's cache blocked SpMV described in the pseudocode of Figure 1. The Cache Blocked and Sparse Loop Optimized SpMV implementation uses the same algorithm for partitioning the matrix into cache blocks beginning with the first non-zero column. However, instead of storing each cache block into an individual CSR, each non-zero is mapped to a particular row within a specific cache block. A description of the algorithm follows.

```
algorithm CB_matmult (CB_matrix, alpha, X, beta, Y)
{
1   Scale Y by beta
2   If alpha is 0, return
3   for each cache block (b) in CB_matrix:
4     mx = b.cols    // num cols in block
5     dx = b.col+1   // starting col number
6     my = b.rows    // num rows in block
7     dy = b.row+1   // starting row number
8     X_b = pointer into X at dx //subVecView
9     Y_b = pointer into Y at dy //subVecView
10    call MatMult(b.mat, alpha, X_b, 1, Y_b)
        // when b.mat is in CSR format, & alpha is 1
        // the following call will ultimately be made:
        // CSR_MM_1_1(mx,my,b.ptr,b.ind,b.val, X_b, Y_b)
11  end for
}

algorithm CSR_MM_1_1(m, n, ptr, ind, val, X, Y)
{
12  for (r = 0; r < m; r++) {
13    register y0;
14    y0 = 0;
15    for (p = ptr[r]; p < ptr[r+1]; p++) {
16      c = ind[p];   // column number
17      register x0;
18      x0 = x[c];
19      y0 += val[p]*x0;
20    }
21    Y[r] = Y[r] + y0;
22  }
}
```

Figure 1. Pseudocode for OSKI 1.0.1h Cache-Blocked SpMV.

```
algorithm CB_sparseLoop_MatMult(pptr, col, val, ncb, n
                                alpha, X, beta, Y)
{
    // n is number of rows in each cache-block
    // ncb is number of cache-blocks
    // pptr is a ncb * n + 1 array of row-start indices
    //      into col and val
    // together: the pptr, col and val arrays comprise
    //      a Compressed Sparse Block Row (CSBR)
    //      data structure

1   Scale Y by beta
2   if alpha is 0, return
3   Scale X by alpha giving aX

    // for each cache block b:
4   for (b = 0; b < ncb; b++) {

      // for each row r in cache block b:
5     for (r = 0; r < n; r++) {
6       register y0;
7       y0 = 0.0

        // for each non-zero in row r of bucket b:
8       for (p = pptr[b*n+r]; p < pptr[b*n+r+1]; p++) {
9         register x0;
10        c = col[p];     // c is column number
11        x0 = aX[c];
12        y0 += val[p]*x0;
13      }
14      Y[r] = Y[r] + y0;
15    }
16  }
}
```

Figure 2. Pseudocode for Cache-Blocked and Sparse Loop Optimized SpMV.

Table 1. Rectangular matrices selected from the University of Florida Sparse Matrix Collection [2].

Matrix	Rows	Cols	Non-zeros	Density	Footprint	UF#	Group	Name	Type
fit2d	25	10524	129042	49.05%	1595 KB	626	LPnetlib	lp_fit2d	linear programming
Trec14	3159	15905	2872265	5.72%	33 MB	2148	JGD_Kocay	Trec14	combinatorial problem
spal	10203	321696	46168124	1.41%	531 MB	1674	Mittelmann	spal_004	linear programming
osa07	1118	25067	144812	0.52%	1905 KB	645	LPnetlib	lp_osa_07	linear programming

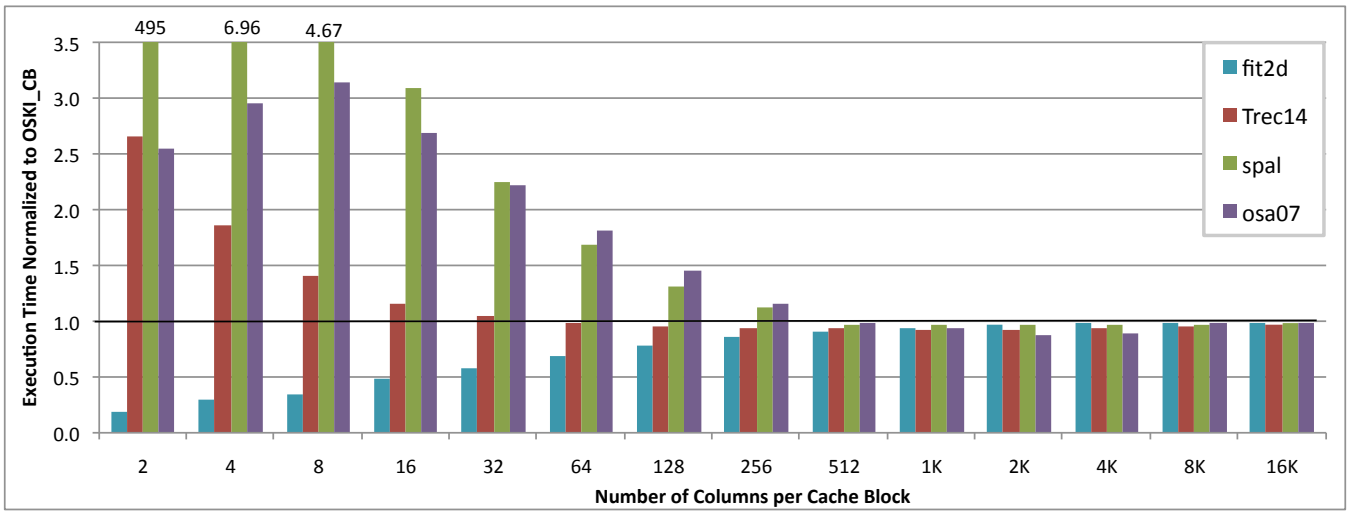

Figure 3. Execution time of Cache-Blocked And Sparse Loop Optimized SpMV normalized to OSKI Cache-Blocked.

The sparse loop optimization algorithm shown in Figure 2 assumes the sparse matrix is stored in a COO data structure that has been re-ordered by accesses into each cache block. Within each cache block, the non-zeros are sorted by row and then by column. The simplest implementation of SpMV uses a COO data structure as shown in the SPF representation of Section 1 where each `p` is an index to a non-zero. Next, we create better data accesses by reorganizing the non-zeros into the cache block. Finally, we access data in each row of each cache block based on the reordering.

In Figure 2 we use a schedule to iterate over each cache block and cache block row. This schedule (stored in `pptr`) specifies which indices, `p`, identify non-zeros in cache block, `b`, and row, `r`. This scheduling allows us to identify non-zeros for a particular cache block and row once and enables us to use a temporary register variable, `y0`, to store dot products inside the inner loop as shown in Figure 2. Each dot product performed within one iteration of the `r`-loop corresponds to the same `Y` value. After the `r`-loop finishes incrementing across a cache blocked row, the temporary, `y0`, is used to update the `Y[r]` location. This is one of the reasons why we observe slightly better performance on larger cache block sizes than OSKI's cache blocked implementation. Larger cache blocks contain a larger number of non-zeros per cache-block row and enable us to make greater use of the inner loop temporary and to do fewer writes to Y. The main reason our approach performs better with a larger cache block size is that larger cache blocks equate to fewer cache blocks overall, resulting in a smaller `pptr` structure, fewer loop iterations and less overhead in general.

3. Methodology and Results

In this section we first describe the methodology of our experiments, followed by a presentation and discussion of our results.

3.1 Methodology

We downloaded four rectangular matrices of varying densities from the University of Florida Sparse Matrix Collection [2]. Table 1 provides the characteristics of the matrices. We chose the spal_004 matrix specifically because it fit the criteria given in [4] to benefit from Cache-Blocking for our platform. The number of rows is small enough to allow the Y vector to fit in cache. The number of columns is large enough such that the X vector will not fit in cache. And the density of the matrix was such that there is sufficient re-use of the Y values per row. We chose the other three matrices to give us a variety of densities, while still keeping a significant rectangular shape. The *density* of a matrix is given as a percent and is calculated as (number of non-zeros)/[(number of rows)*(number of columns)]. When every location in the matrix contains a non-zero, the density would be 100%.

We conducted our timing experiments on a 2.83 MHz, Intel Core 2 Duo E8300, 64-bit machine running Linux kernel 2.6.32.21-168.fc12.x86_64. It has 2 cores, a 32 KB L1 data cache, a 6MB L2 cache, and a 256 entry x 4KB page TLB. For all executables, we used the GNU C compiler version RedHat 4.4.4-10 with compiler flags '-O3 -g'.

Table 1 shows the average memory footprint for the total computations performed on each matrix. The smaller matrices, fit2d and osa07, fit within the 6MB L2 cache, but not within the 32KB L1 cache without cache blocking. The larger matrices, spal_004 and Trec14, certainly do not fit within the 6MB L2 cache without cache blocking.

For our OSKI results, we installed OSKI 1.0.1h [1] onto the machine described above using the default integer/double configuration. The OSKI matrix-vector multiply calculates $Y = \alpha * Y + \beta * A * X$, where A is the sparse matrix, X is the source vector, Y is the destination vector, and α and β are input coefficients. OSKI

implements a number of sparse matrix types to facilitate their optimizations. For each matrix type, they implement specific matrix multiply routines for situations when α and/or β are 0 or 1. For all of our experiments, we specified that α and β were 1.

3.2 Results

Figure 3 compares our Cache Blocked and Sparse Loop Optimized SpMV to OSKI's cache blocked SpMV. The vertical axis records the execution time of our algorithm normalized to that of OSKI's. The horizontal axis identifies the number of columns per cache block, where the number of rows in the cache block is the total number of rows in the matrix. We show the results for four matrices in Figure 3.

In general, for smaller cache block sizes ranging from 2-256 columns, OSKI's cache blocked SpMV executes much faster than our Cache Blocked and Sparse Loop Optimized SpMV. One exception on smaller cache block sizes occurs with the fit2d matrix, which has a high density of 49.05%. For this matrix, our Cache Blocked and Sparse Loop Optimized SpMV out-performs OSKI on all cache-block sizes.

For larger cache block sizes ranging from 512 to 16K columns, our Cache Blocked and Sparse Loop Optimized SpMV implementation performed slightly better than OSKI's cache blocked SpMV. As discussed in Section 2 and seen in the code of Figure 2, a larger cache block not only reduces the number of nested loop iterations, but also contains a larger number of non-zeros per row, enabling greater reuse of the inner loop temporary and fewer writes to Y in our sparse loop optimization. Overall, our results in Figure 3 demonstrate that the Cache Blocked and Sparse Loop Optimized SpMV algorithm performs better than OSKI's cache blocked SpMV for large cache block sizes and supports our hypothesis.

4. Conclusions

Complex sparse matrix data structures abound in current hand-optimized and auto-tuner optimized sparse matrix computations. We present a simpler approach where sparse matrix computations are specified using a coordinate storage format, transformed using the Sparse Polyhedral Framework (SPF), automatically optimized with various data reorderings, and finally an optimization called the sparse loop optimization is applied to introduce a generalized CSR data structure, the Compressed Sparse Block Row (CSBR), into the innermost loop. We show that this simpler approach matches the performance of the more complex sparse matrix data structure used by OSKI to implement cache blocking on the sparse matrix vector multiplication computation.

Acknowledgments

This work was supported by the CSCAPES Institute, which is supported by the U.S. Department of Energy's Office of Science through grant DE-FC-0206-ER-25774, as part of its SciDAC program. This work was also supported by a DOE Early Career Award.

References

[1] OSKI: Optimized Sparse Kernel Interface. URL `http://bebop.cs.berkeley.edu/oski/`.

[2] University of Florida Sparse Matrix Collection. URL `http://www.cise.ufl.edu/research/sparse/matrices/`.

[3] A. J. C. Bik and H. A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 416–424, Tokyo, Japan, July 20–22, 1993. ACM SIGARCH.

[4] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick. When cache blocking sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication, and Computing*, 18(3):297–311, March 2007.

[5] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, June 2003. ACM.