THESIS

A MODEL FOR PREDICTING THE PERFORMANCE OF SPARSE MATRIX VECTOR MULTIPLY (SPMV) USING MEMORY BANDWIDTH REQUIREMENTS AND DATA LOCALITY

Submitted by

Stephanie Dinkins

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2012

ABSTRACT

A MODEL FOR PREDICTING THE PERFORMANCE OF SPARSE MATRIX VECTOR MULTIPLY (SPMV) USING MEMORY BANDWIDTH REQUIREMENTS AND DATA LOCALITY

Sparse matrix vector multiply (SpMV) is an important computation that is used in many scientific and structural engineering applications. Sparse computations like SpMV require the use of special sparse data structures that efficiently store and access non-zeros. However, sparse data structures can tax the memory bandwidth of a machine and limit the performance of a computation, which for these computations is typically less than 10% of a processor's peak floating point performance. The goal of this thesis was to understand the trade-off between memory bandwidth needs and data locality for SpMV performance. We construct three performance models based on memory bandwidth requirements and the data locality that is associated with the non-zero orderings within a sparse data structure. Our approach uses metrics that can be applied to any sparse data structure to model SpMV performance. We use a data locality metric termed Manhattan distance to predict the data locality execution time of each non-zero ordering, which produced strong per matrix results. Based on the per matrix results for the Manhattan distance we constructed a model that computes a linear fit for multiple parameters, but those results were not promising. We found that the memory bandwidth requirement for a data structure is a key component to predicting performance. Our final model uses memory bandwidth pressure and an averaged predicted data locality time to accurately predict the fastest performing data structure 73-84% of the time, depending upon the machine. Our results indicate that often the sparse data structure with the least taxation on the memory bandwidth produces the fastest execution times.

TABLE OF CONTENTS

Chapter 1

Introduction

Sparse matrix vector multiply (SpMV) is an important computation that is used in many scientific and structural engineering applications. In his DARPA HPCS presentation titled, "Defining Software Requirements for Scientific Computing", Phillip Colella [1] identified seven numerical methods known as the Seven Dwarfs, which identify patterns of computation and communication in algorithms that have persisted over time and will continue as important numerical methods for science and engineering for at least the next 10 years. Sparse linear algebra is included in this list as one of the Seven Dwarfs. The SpMV computation is commonly used by iterative methods for solving sparse linear systems [2, 3].

SpMV uses sparse matrix data structures to avoid excessive computation with zeros. The index array accesses needed by these structures apply pressure to the memory bandwidth of the machine and limits the performance of the algorithm, which can be as slow as 10% of a processor's peak floating point performance [4].

One common technique that improves the performance of SpMV is to use special sparse data structures that efficiently store and access the non-zeros of the matrix in a compressed format during computation. Other optimization techniques like cache blocking, and register blocking compress part of the index space to improve memory overhead and to improve locality by reordering the non-zeros of a sparse matrix to improve cache behavior. We hypothesize that the memory bandwidth requirements and data locality of each sparse data structure determines the execution time of the computation.

1.1 Memory Bandwidth Requirement versus Data Locality

While experimenting with various sparse data structure representations (e.g., COO, CSR, and cache blocked), we found that in some cases a sparse matrix data structure with a greater memory bandwidth requirement resulted in better performance than a different sparse data structure with a smaller memory requirement. The faster performance of the data structure with the larger memory bandwidth demand was due to a better ordering of the non-zeros. In other cases we found that the memory bandwidth requirement was the dominant performance factor. The goal of this thesis was to understand these results.

1.2 Previous Work

In related work, Nishtala et al. [4] observed similar performance behavior when comparing SpMV performance using a Compressed Sparse Row (CSR) data structure and cache blocked matrices. Their work produced a performance model based on cache and TLB miss and hit estimates, which was used to select cache block sizes that produced the best performance. In related work, Nishtala, Vuduc, et al. [5], used this performance model coupled with other cache blocking techniques to improve the performance of cache blocked SpMV. Related research performed by the Berkeley Benchmarking and Optimization Group (BeBop) [6] produced an autotuner named, the Optimized Sparse Kernel Interface (OSKI), which specializes in improving the performance of sparse kernels like SpMV and sparse triangular solve.

OSKI's performance model discussed in Nishtala's work is specific to SpMV, in which non-zeros are accessed using a CSR ordering with cache blocking. OSKI's performance model for SpMV was not applicable when we found a sparse matrix stored in the Coordinate storage (COO) format performing better than a matrix stored in the CSR data structure [7]. The CSR data structure is a more compressed format than the COO structure and may have a smaller memory footprint of the two structures for the same matrix. Additionally, CSR imposes a by row ordering to the non-zeros that is not required by the COO.

The conflicting performance results that we observed for SpMV executed using data structures with different memory bandwidth demands and non-zero orderings indicates the need for a performance model based on memory bandwidth requirements and data locality that can guide the selection of the sparse data structure and ordering that will produce the best performance.

1.3 Performance Model Contribution

This thesis analyzes the effect of data locality and memory bandwidth requirements on execution time by isolating each performance factor and modeling its impact on performance. We present a model that uses memory bandwidth needs and a data locality metric termed Manhattan distance, which measures the access distance between array indices for different orderings of non-zeros. Manhattan distance was chosen as a data locality metric because this measurement changes as the ordering of the non-zeros change, which enabled us to isolate and estimate the impact of data locality on execution time for a given sparse matrix non-zero ordering without the necessity to execute SpMV on additional data structures. Our model uses a lower bound memory bandwidth prediction that assumes each data item only needs to be brought in from memory once.

We evaluate our performance model based on its ability to predict execution time, select the best performing data structure and reordering, and performance degredation when it does not. The model is developed using linear regression techniques to determine the strength and type of relationship that exists between memory bandwidth needs, data locality, and execution time. We found that a linear relationship exists between our data locality metric, Manhattan distance, and data locality execution time per matrix and incorporated this finding into the model. Specifically, Manhattan distance can be used to predict the non-zero ordering that will produce the fastest execution times for a particular matrix. However, the metric is only useful as a per matrix predictor.

Ideally, this model would predict the execution times of SpMV performed on any sparse matrix. However, the average execution time relative error was 2.5 times the model's predicted execution time. Instead, we hypothesized that our performance model is able to predict relative performance by predicting the data structure and ordering that will produce the fastest execution time, but the accuracy was an average of 13.5%. Even though we were unsuccessful in predicting data locality execution times, we found that we could predict the fastest performing data structure 78.6% of the time using memory bandwidth requirements with an average execution time error prediction of 7.2%.

We begin the discussion with some background information about the SpMV algorithm and the data structures used in the analysis in Chapter 2. Chapter 3 discusses related work followed by a discussion of our SpMV performance model in Chapter 4. Then, we evaluate our performance model in Chapter 5 and discuss conclusions and future work in Chapter 6.

Chapter 2

Background

Optimizing performance of Sparse Matrix Vector Multiply (SpMV) is complicated by machine architecture and the necessity to manipulate sparse data structures [4]. In general, performance optimizations for sparse computations tend to result in the use of complex data structures required to efficiently store and access sparse matrices. In order to understand the performance trade-offs between memory bandwidth pressure and data locality for SpMV, we must first understand the computation, and the sparse data structures that are used to store and access the non-zeros during SpMV. Sparse data structures are designed to store the non-zeros of sparse matrices in a compressed format that can be efficiently accessed. Some data structures impose a restriction to the ordering of the non-zeros that it stores and others do not. We begin with a discussion of the SpMV algorithm and each of the three data structures used in our analysis.

2.1 Sparse Matrix Vector Multiply (SpMV)

Figure 2.1 presents the general forumla for matrix vector multiply where A is the matrix, X is the source vector, and Y is the solution vector. Figures 2.1 and 2.2 demonstrate how the elements of a matrix, the source vector X and the solution vector Y are accessed during matrix vector multiplication. The dot product is computed between each element A_{ij} in a row of matrix A and a corresponding source vector element X_j. The dot product for a row is then stored in one element of Y_i. The Sparse Matrix Vector (SpMV) algorithm discussed in this thesis is expressed as

$$\vec{y} = A\vec{x}.$$

$$y_i = \sum_{j=1}^{n} A_{ij} x_j$$

Figure 2.1: Matrix Vector Multiply Formula

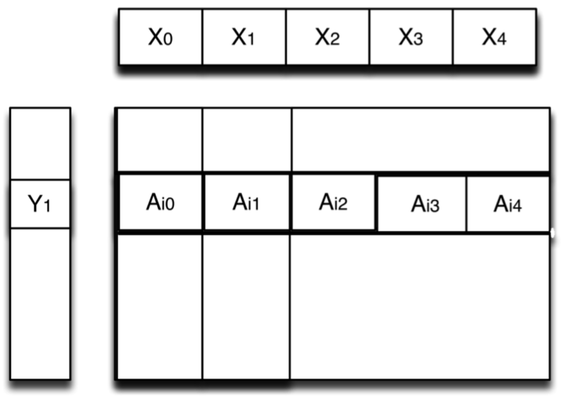


Figure 2.2: Matrix A, Source Vector X, and Solution Vector Y

2.2 Sparse Data Structures

This section presents a brief explanation for each of the data structures and non-zero orderings that were used in our analysis, which includes Coordinate Storage (COO), Compressed Sparse Row (CSR), and cache blocked data structures [8].

2.2.1 Coordinate Storage

The simplest implementation of SpMV uses a flat data structure known as Coordinate Storage to store the non-zero elements of the sparse matrix. The algorithm is implemented using a single loop to iterate over the non-zeros and has an algorithm complexity of order $\Theta(n)$, where n refers to the number of non-zeros in the matrix. Figure 2.3 shows the pseudocode for SpMV performed using the COO data structure.

```
for (i = 0; i < nnz; i++) {
    Y[row[i]]+=X[col[i]]*Val[i];
}
```

Figure 2.3: SpMV on Coordinate Storage Data Structure Pseudocode

Figure 2.4 shows a diagram of the COO data structure, which consists of three arrays of equal length: the column array, row array, and value array. Each row, column pair maps to a single non-zero. COO does not impose an ordering on the non-zeros like other sparse storage structures, which enabled us to benchmark SpMV with various non-zero orderings including a percent randomized reordering of the non-zeros when designing our performance model. In particular, we were able to use the COO data structure to isolate the memory footprint in our experiments by keeping the non-zero orderings the same as the non-zero orderings in other data structures and comparing the performance between sparse matrix data structures with different memory bandwidth demands. Additionally, COO enabled us to isolate the data locality by keeping the memory footprint the same and varying the non-zero orderings within the COO data structure.

2.2.2 Compressed Sparse Row

One of the most common sparse data structures used in sparse computations is Compressed Sparse Row (CSR), which restricts the ordering of the non-zero elements that it stores to a by row ordering. SpMV implemented using the CSR data structure has an algorithm complexity of order $\Theta(n + m)$, where n is the number of rows and m is the number of non-zeros, but performance is dominated the n term due to the indexing overhead of the row pointer array. This data structure consists of three arrays and is illustrated in Figure 2.5. The row pointer array points to the first non-zero stored in a row and is equal to the length of the number of rows + 1. The Column index array stores the column indices associated

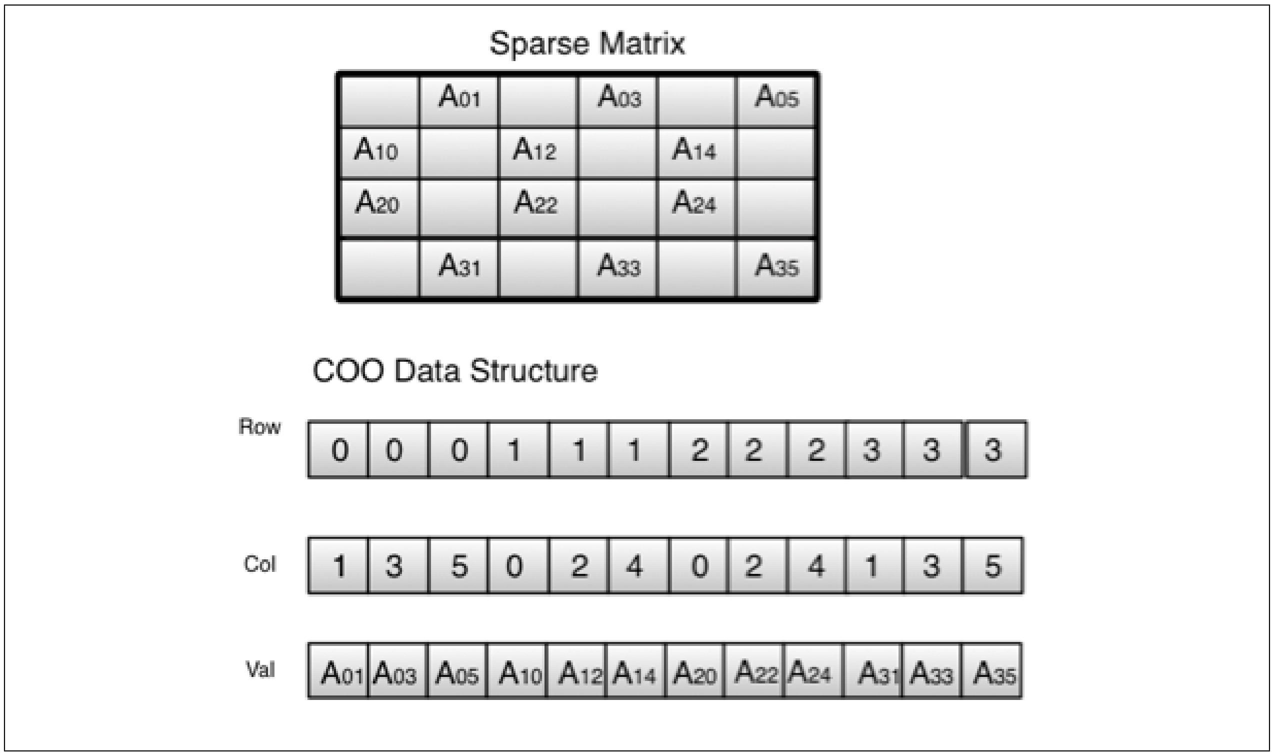


Figure 2.4: Coordinate Storage Data Structure (COO)

with each non-zero, and the value array stores the non-zeros. A pseudocode example of SpMV performed using the CSR data structure is shown in Figure 2.6.

2.2.3 Cache Blocked

Cacheblocking is a technique that is commonly used to improve the performance of SpMV by improving the temporal and spatial locality in the source vector X and solution vector Y. Figures 2.7(a), and 2.7(b) demonstrate how the temporal and spatial locality changes before and after cache blocking is applied. Figure 2.7(a) shows the order of accesses to the X vector during SpMV without cache blocking, which is indicated by each access to an index of X at time (t). In Figure 2.7(a) we see that the first index accessed at t(1) is index 2. Then, at t(2) index 4 is accessed, followed by indices 6,1,5, and so forth. The spatial locality is very poor in that the accesses to X are not to contiguous indices and therefore, the data elements are not likely to be fetched together from the cache. Also, the accesses to the same indices of X are probably not occuring close enough together in time so that data would still reside in the cache between accesses in X and Y, which results in a lack of

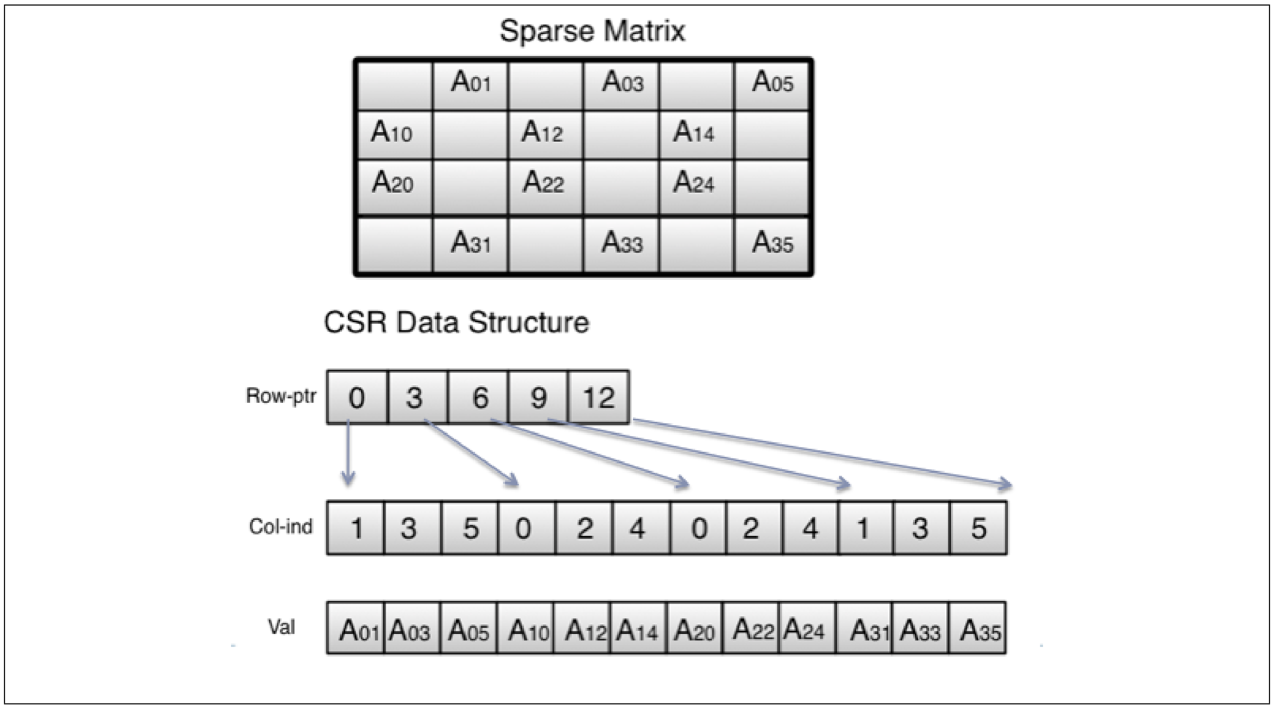


Figure 2.5: Compressed Sparse Row Data Structure (CSR)

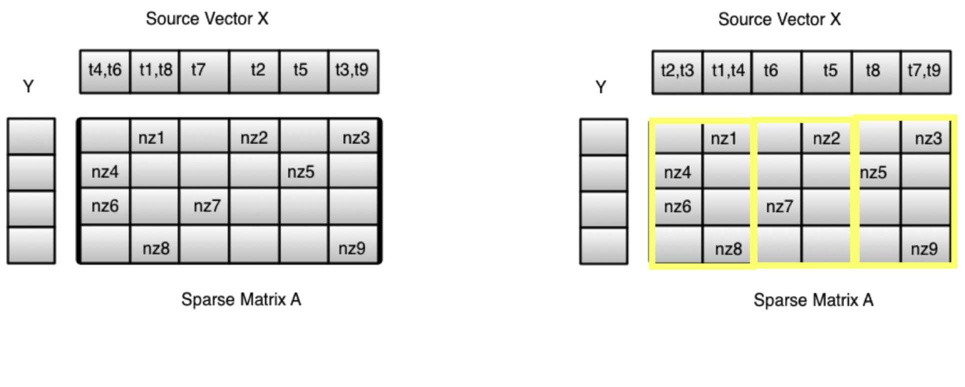
```
for (row = 0; row <= nrows-1; row++) {
        temp = 0.0;
        for (j = rowptr[row]; j < rowptr[row+1]; j++ ) {
               // j is index into the column array and value array
               temp += Val[j] * X[ColInd[j]];
        }
        Y[row] = temp;
}
```

Figure 2.6: SpMV on Compressed Sparse Row Data Structure Pseudocode

temporal locality.

Figure 2.7(b) shows that cache blocking changes the access pattern in the source vector such that contiguous indices of X are accessed multiple times during the iteration over the same cache block, which are outlined in yellow. As an example, let us observe the access pattern for a single cache block as shown in Figure 2.7(b). Looking at the first cache block, we now see at time t(1) that the first index of X accessed is index 2 followed by index 1 at t(2) and t(3). Then, index 2 is accessed again at t(4). This example explains how cache blocking increases the likelihood that contiguous indices of X that are associated with a cache block are accessed in short succession, thereby improving temporal and spatial locality on

X. Temporal locality occurs as reuse on Y if the successive accesses occur close enough in time as each dot product is computed between A_{ij} and X_j and written to Y_i. Thus, spatial and temporal locality are improved via the cache blocking technique.



(a) Before Cache Blocking (b) After Cache Blocking

Figure 2.7: Cache Blocked Data Structure

Our implementation of cache blocking uses a row pointer array into each cache block and row within a cache block, which results in an algorithm complexity of $\Theta(n * m) + nnz$, where n is the number of cache blocks, m is the number of rows in the cache block, and nnz is the number of non-zeros in the matrix. Each cache block is accessed in a by row order, similar to the CSR data structure shown in Figure 2.5. Example pseudocode of iterating over a cache blocked data structure is shown in Figure 2.8.

```
//nrows is number of rows in a cache block
//ncb is number of cache blocks

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

  //for each row r in cache block b
    for (r = 0, r < nrows; r++)
        y0 = 0.0;

        // for each non-zero value in row r of cache block b:
        for (p = pptr[b*nrows+r]; p < pptr[b*nrows+r+1]; p++) {
            c = col[p];    //c is column number
            x0 = X[c];
            y0 += val[p]*x0;  //dot product

      }
        Y[r] += y0   //write to Y
    }

}
```

Figure 2.8: SpMV on Cache Blocked Data Structure Pseudocode

Chapter 3

Related Work

The SpMV performance problem is well studied and high performance has been achieved using performance optimization tools like the Optimized Sparse Kernel Interface (OSKI) [3,9] autotuner developed by the Berkeley Benchmarking and Optimization Group (BeBop) [6]. The OSKI autotuner uses performance models relevant to CSR-like data structures (e.g., BCSR, GCSR, and CSR), where as our performance model considers metrics that may be used for any sparse data structure. Specifically, our model is based on the memory bandwith needs and the data locality that is associcated with the various non-zero orderings.

This section also discusses the roofline model developed by Williams et al., [10]. The roofline model was developed as a means to analyze performance of parallel programs executed on a particular architecture based on computation, memory bandwidth, and locality. Our performance model predicts relative performance within the memory bandwidth bounds for a particular data structure and architecture and for the data locality that occurs within those performance bounds for a serial implementation of SpMV. Our model is specific to SpMV and therefore incorporates some performance characteristics that are specific to SpMV.

3.1 The Optimized Sparse Kernel Interface (OSKI)

One of the sparse matrix optimizations available in OSKI is cache blocking. 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 structure that is generated for each submatrix [11].

Given an autotuned *rxc* cache block size, the conversion algorithm uses an inspector to step through the matrix along rows partitioning data into cache blocks. The start of each cache block begins with the first non-zero in a column. Empty columns are not included in the beginning of a cache block so, the start of each cache block begins with the first non-zero in a column. Once the matrix is partitioned into a list of cache blocks, a sparse submatrix is generated for each cache block [11]. Figure 3.1 illustrates this linked list of CSRs data structure.

OSKI further optimizes this cache blocked data structure by using either a general CSR (GCSR) or CSR storage. We found that the GCSR data structure was used to store cache blocks ranging in size from 8-256 columns. We suspect this decision eliminates storing empty rows and keeps the memory footprint small. Through inspecting the OSKI code base we determined that the GCSR differs from the CSR data structure in that the GCSR does not store information about rows that do not contain non-zeros. Thus, the GCSR data structure is slightly more compressed than the traditional CSR and would have a slightly smaller memory bandwidth need. Pseudocode for the OSKI 1.0.1h cache blocked SpMV algorithm is shown below in Figure 3.2.

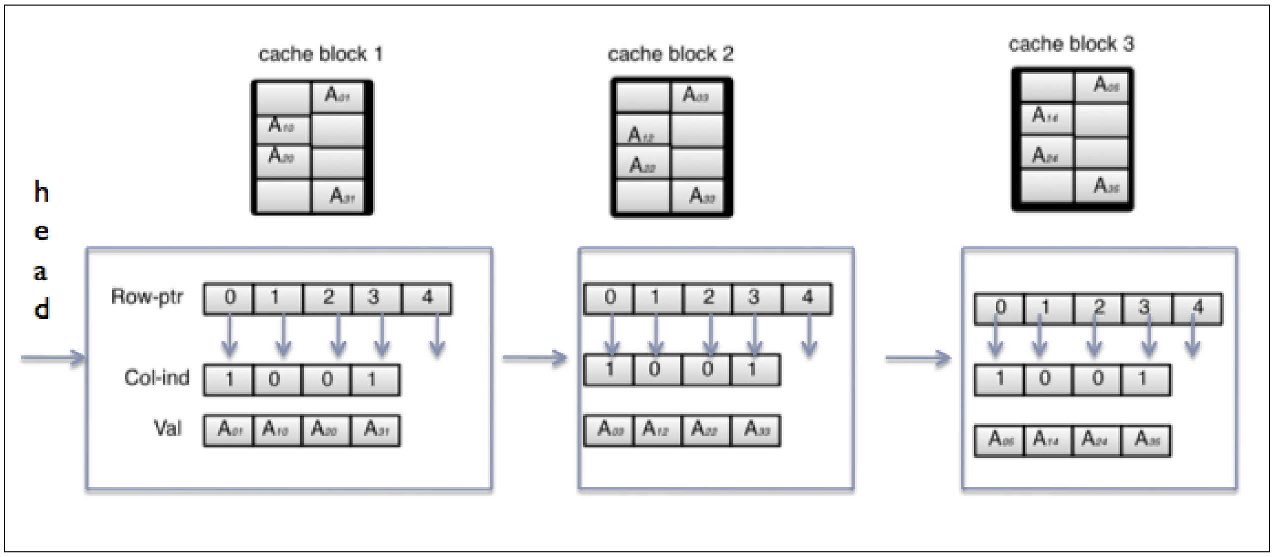


Figure 3.1: OSKI's Cache Blocked Linked List of Submatrix CSRs Data Structure

```
//for each cache block submatrix stored as a CSR

//for each row in a cache block
for (r = 0; r < nrows; r++) {
    y0=0;

    //for each non-zero value in row r of a cache block
    for (p = ptr[r]; p< ptr[r+1]; p++) {
        c = ind[p];      // column number
        x0 = X[c];
        y0 += Val[p]*x0;
    }
    Y[r] += y0;
}
```

Figure 3.2: Pseudocode for OSKI 1.0.1h Cache Blocked SpMV

During benchmarking experiments in which we compared the performance of our cache blocking algorithm to OSKI's we observed that OSKI executed faster than our implementation on the smaller cache block sizes. Our cache blocked implementation of SpMV accesses the non-zeros in the same order as OSKI so, we suspected that the performance difference was due to smaller memory bandwidth needs by the OSKI data structure.

Nishtala et al., [4] created a performance model for cache blocked SpMV based on a cache and TLB miss model that is used to predict the best cache block size for a matrix. In addition, the authors presented several matrix characteristics, which qualify a matrix as favorable for cache blocked SpMV: when X is large, Y is small, random distribution of non-zeros, and a sufficiently high density. Our performance model is used to predict relative performance of SpMV for a given data structure and ordering, which includes the cache blocked SpMV data structure and ordering.

3.2 Roofline Model

The roofline model [12] was developed as tool for program analysis and optimization for code executing on a parallel architecture. This model indicates that three primary components determine performance: computation, communication, and locality. Computation is measured in Gflops/s. Communication i.e., bandwidth is measured in GB/s with the

STREAM benchmark [13]. Locality is optimized by minimizing communication and improving data locality through software optimizations. The goal of the roofline model is to create a visual representation of the performance, memory bandwidth, and locality that is unique to each architecture.

The model bounds the best possible performance in terms of the product between machine bandwidth and the flop:byte ratio, and the peak floating point operations per second (Gflops/s) [10]. By comparision our model bounds performance with memory access time and attempts to find where within those bounds lies the attribution of data locality to performance. More specifically, we attempt to model the impact of the ordering of non-zeros, where as, the roofline model does not consider the impact of different non-zero orderings on its performance bounds.

Chapter 4

SpMV Performance Model

Ideally, a performance model would predict execution time, however, a model that is able to predict a data structure and non-zero reordering that provides the fastest execution times for a given data set and machine is also useful. Additionally, once training is complete for various non-zero orderings, we hypothesized that the model could predict performance for multiple data structures. Information collected during a single training session could then be used to predict performance for non-zero orderings and the COO, CSR, and cache blocked data structures without running any more benchmarks. After training is complete for a set of matrices, all that would be required for the model to predict performance for COO, CSR, and cache blocked data structures and machines are some sparse matrix and machine input parameters. The sparse matrix and input parameters include the number of rows, columns, and non-zeros in the matrix, the size of the cache line on the machine, and the machine bandwidth. This chapter describes the various performance models that we developed.

4.1 Components of the SpMV Execution Time

Our models are composed of three components: memory bandwidth needs, the time needed to refetch data for the SpMV computation, and the indexing overhead of a data structure.

$$Execution\ time\ = (Time\ to\ fetch\ data\ from\ memory\ once)+$$

$$(Time\ to\ refetch\ data) + (Indexing\ overhead\ for\ cache\ blocked\ data\ structure)$$

The memory bandwidth component in our models addresses the portion of execution time that is attributed to the constraints on communication that exist within the memory hierarchy. Specifically, memory bandwidth limits the rate at which data is transferred. We use the STREAM benchmark [13] to measure memory bandwidth of each architecture to determine the maximum rate at which data may be transferred within the memory hierarchy. Machine bandwidth is measured in GB/s and used to predict the execution time that may be attributed to the memory bandwidth needs of a data structure.

The data locality component of our model depends upon the ordering of access to the non-zeros of the sparse matrices. Maximizing temporal and spatial locality on the source and solution vectors improves execution time for SpMV by minimizing the number of cache fetches required during the computation. In essence, the data locality component refers to the time required to refetch the data. The third component of our model addresses any indexing overhead that is associated with each sparse matrix data structure based on how it stores the non-zero value indices.

4.2 Bounding Performance with Memory Access Time

SpMV is a memory bandwidth bound computation. In this section we discuss how we can model the memory bandwidth bounds for the execution time performance that is attributed to memory bandwidth and for now, ignore the indexing overhead that is due to the number of non-zeros and indexing structures used by the COO, CSR, and cache blocked data structures. Our training set used to develop the model consists of 32 matrices, which are shown in Table A in the Appendix.The machine architectures including bandwidth and cache configuration information is shown in Table 4.1.

The average percent of the execution time used to bring in all of the data from memory once is 53% and ranges between 34-71% of the total execution time. We compute this average as follows. Let AMT refer to the average execution time attributed to memory bandwidth, $MFlb$ refers to the best possible execution time, ET refer to the total execution time, and

Table 4.1: Various machine architectures and Cache Parameter.

Processor	L1(KB)	L2(MB)	TLB(KB)	BW (GB/s)
Intel Core2 Duo E8300	32	6	128 entries x 8KB pages	5.34
Intel Core2 Duo 6400	32	2	256 entries x 4KB pages	8.53
Intel Xeon E5405	32	2 x 6	256 entries x 4KB pages	5.34

NM refer to the number of matrices.

$$AMT = (\Sigma(MFlb/ET))/NM$$

We use machine bandwidth, the memory footprint for each of our sparse data structures, and assumptions about the number of cacheline fetches needed to fetch the data to model the best and worst possible execution times attributed to memory bandwidth needs. Figure 4.1 illustrates the concept by modeling the execution time boundaries for a given data structure's memory bandwidth demands. The best possible execution time boundary, shown as a dashed blue line, assumes that the matrix, source, and solution vectors are each only read in once. The worst possible execution time boundary, shown in black, assumes that one cache line for X and one cache line for Y must be fetched for every non-zero that is accessed.

4.2.1 Computing Memory Bandwidth Requirements for SpMV

The smallest and largest memory bandwidth requirements for computing SpMV on a given matrix are dependent upon the sparse matrix memory footprint. The equation for the smallest memory bandwidth requirements assumes that the source and destination vectors, X and Y, are only read into memory once and that they are stored sequentially in memory.

We compute the X and Y requirements by multiplying the number of rows and columns by 8 because X and Y, both hold the doubles data type, which are 8 bytes each. The number of non-zeros, nnz, is multiplied by 16 due to the construction of the COO data structure as shown in Figure 2.4. We have 8 bytes for each double stored in the value array, and 4 bytes for each of the row and column index array entries, which are equal in length to the

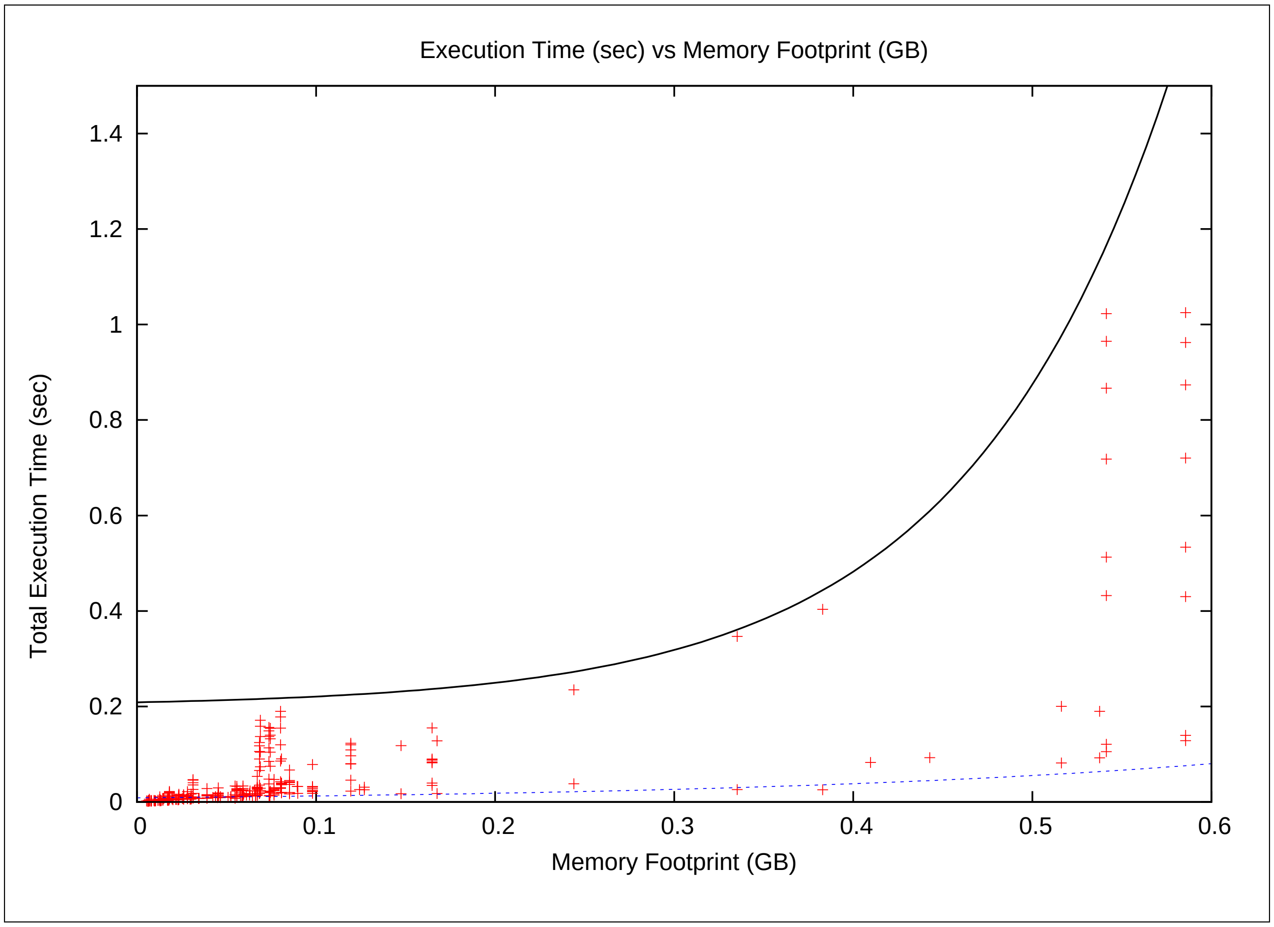


Figure 4.1: This model bounds the actual execution times between the best and worst possible execution time performance. Execution time attributed to data locality also lies within these bounds.

nnz in the matrix. The resulting equation for the smallest bandwidth requirement for the COO data structure is as follows, where $minbW$ refers to the smallest possible bandwidth requirements for the COO data structure.

$$minBW = (16 * nnz + (8 * ncols) + (8 * nrows))$$

Similarly, the smallest memory bandwidth required by the CSR data structure is computed as follows. Again, the requirements for X and Y are computed as $8 * ncols$ and $8 * nrows$, respectively. The row pointer array is equal in length to the number of rows in the matrix + 1. So, $4 * (nrows + 1)$ accounts for the integers stored in the row pointer array. Next, the column index and value arrays are equal in length to the nnz in the matrix. The value array holds doubles and the column index array holds integer values, which translates to $12 * nnz$. Putting the components of the equation together, we have the minimum bandwidth requirements for CSR, indicated as $minBW$:

$$minBW = (((12 * nnz) + (4 * (nrows + 1)) + (8 * ncols) + (8 * nrows)).$$

Next, we compute the largest memory bandwidth requirements needed by the COO and CSR data structures to perform SpMV. The worst possible requirements for bandwidth assumes that every access to a non-zero in the matrix requires a line of the cache to be fetched. The resulting equation needed to compute the maximum memory bandwidth, $maxBW$ needed by a matrix stored as COO follows:

$$maxBW = (16 * nnz + (cacheline\ size * 2 * nnz)),$$

where cacheline size refers to the size in bytes used to store a line of data in the cache. The size of a line of cache is 64 bytes for the architectures used in our experiments.

Similarly, we can compute the maximum memory bandwidth requirement needed by CSR data structure, $maxBW$ as

$$maxBw = (((12 * nnz) + (4 * nrows) + 4) + (cacheline\ size * 2 * nnz)).$$

The smallest memory bandwidth needed by the cacheblocked data structure is computed as the summation of the memory required by each cache block, multiplied by the number of cache blocks. Each cache block requires $12 * (nnz/cache\_block)$, which breaks down as 8 bytes for the the cache block's portion of the value array and 4 bytes to store the cache block's portion of the column array. The summation of all of the non-zeros in all of the cache blocks is equal to all of the non-zeros in the matrix, which results in $12 * nnz$ for all of the cache blocks. In addition, there is a row pointer array that indexes into each row of a cache block, which results as $4 * nrows * ncb$, where ncb refers to the number of cache blocks. Again, we account for the 8 bytes each that are needed for X and Y. The final equation for the minimun bandwidth requirements for the cache blocked data structure is

$$minBW = (12 * nnz) + (4 * nrows * ncb) + (8 * ncols) + (8 * nrows)).$$

The largest amount of memory bandwidth needed to perform cache blocked SpMV for our implementation of the cache blocked data structure assumes that each access to a non-zero results in cache line fetch for both the associated X and Y elements. The final equation follows:

$$maxBW = (12 * nnz) + (4 * nrows * ncb) + (cacheline\ size * 2 * nnz)).$$

As described in previous chapters, our implementation of the cache blocked data structure has the overhead of an indexing array for every cache block that is generated, which can be significant enough to impact execution time for cache blocks ranging in size from 2K to 8K columns per cache block. OSKI's cache blocked implementation uses either a CSR or GCSR data structure to store the non-zeros in each cache block. The GCSR data structure referred to as a generalized CSR does not index into an empty row of a cache block. This optimization reduces the indexing overhead and as a result also reduces the memory bandwidth requirements of the data structure for OSKI's implementation of the cache blocked data structure, which improves performance over our implementation for the smaller cache block sizes.

4.2.2 Computing the Execution Time Boundaries for the Memory Bandwidth Model

Using the memory bandwidth requirements explained above, we computed the execution time boundaries for best and worst case execution times for every data structure and ordering described in Chapter 2 on a training set of 32 matrices. The modeled boundaries for the best and worst possible execution times are computed as

$$(Best\ Possible\ Execution\ time) = (least\ memory\ required\ /Machine\ bandwidth)$$

and

$$(Worst\ Possible\ Execution\ time) = (most\ memory\ required\ /$$
$$Machine\ bandwidth),$$

where the least or most memory required refers to the amount of memory required to perform SpMV for a given matrix and data structure. The amount of memory required for a data structure is computed for each matrix and data as described in the previous section.

Timing experiments shown in Figure 4.1 were conducted on a 2.83 GHz, Intel Core 2 Duo E8300, 64-bit machine running Linux kernel 2.6.32.21- 168.fc12.x86 64. The machine has 2 cores, a 32 KB L1 data cache, a 6MB L2 cache, and a 128 entry x 8KB page TLB. We used the GNU C compiler version RedHat 4.4.4-10 with compiler flags '-O3' for executables. The results shown are for the training set of 32 matrices executing SpMV on each data structure and non-zero ordering described above.

The x-axis in Figure 4.1 is the memory footprint measured in GB for each data structure and ordering using the equations presented in this section. The y-axis is the total execution time measured for the SpMV computation, which was averaged over 100 iterations of SpMV. From the graph we observe that execution time increases as the memory bandwidth requirements increase, which limits the rate at which data is fetched from within the memory hierarchy. The modeled performance bounds for SpMV as shown in Figure 4.1 mark the best and worst possible execution times based on the minimum and maximum memory bandwidth required to perform SpMV given the assumptions for best and worst performance described above, machine bandwidth, and cacheline size.

In the model for execution time bounds we can compute the amount of execution time that is attributed to the memory bandwidth and the portion of execution time that is attributed to data locality. We hypothesized that data locality is the other primary component needed by our performance model. The amount of execution time attributed to the memory bandwidth requirements in our model is the best possible execution time boundary ($MFlb$), which was computed at the beginning of this section and is appears as the red line in Figure 4.1. The amount of execution time attributed to data locality, DLT, is computed as the difference between the total execution time, ET, and the best possible execution time, $MFlb$ as shown in the equation below.

$$DLT = ET - MFlb$$

4.3 Impact of Data Locality on Performance

Much research has been invested into developing sparse data structures that efficiently store sparse matrices and enable the user to iterate over the non-zeros in a manner that produces good data locality [14,15], [16] for a particular computation. In SpMV, opportunities for both spatial and temporal data locality are present. Spatial locality for SpMV occurs on the source vector X when contiguous data elements are fetched together from the cache, which occurs when contiguous indices of X are accessed. Temporal locality occurs as reuse of an X or Y element if more than 2 non-zeros in a column or row are accessed close enough in time that the data is still in the cache.

Data locality can be measured directly by observing cache and TLB hit rates, which means that we can measure the lack of data locality via the miss rates. We can estimate the effect of data locality on performance with the Manhattan distance, which measures the distance between address accesses in both the source and solution vectors.

On average, we observed that data locality accounts for 47% of the total execution time for our training set and ranges between 29-66% of the total execution time. The average portion of execution time attributed to data locality is computed as follows, where ADT refers to the average execution time attributed to data locality, and $MFlb$, ET, and NM refer to the best possible execution time, execution time, and the number of matrices.

$$ADT = (\Sigma((ET - MFlb)/ET)/NM$$

4.3.1 Measuring Data Locality Using Cache and TLB Misses

We measured the number of L1 and L2 cache and TLB misses in order to measure the amount of data locality present in a set of non-zero orderings of a sparse matrix in the data set. We used the performance API tool (PAPI) [17,18] in combination with machine hardware counters to observe the total number of cache and TLB misses that occur while executing SpMV on the training set. Cache and TLB miss experiments shown in Figure 4.5 were conducted on a 2.0 GHz, Intel Xeon E5405, 64-bit machine. The machine has 4 cores, a 32KB L1 datacache, a 2x6MB L2 cache, and a 256 entry x 4KB page TLB. We used the

GNUC compiler with compiler flags '-O3' for the executables and included the papi-3.7.0 library for the analysis.

We used the COO data structure to observe cache behavior for various orderings of the non-zeros. Some data structures such as CSR restrict the order in which non-zeros are stored to a by row ordering, but the Coordinate storage data structure (COO) does not impose a restriction on the ordering. The orderings used in the experiment include the original ordering of the sparse matrix file, cache blocked orderings (2K, 8K, and 32K columns per cache block), a range of percentage randomized reorderings on the original matrix from 10 - 100%, and the CSR by row ordering.

In the next section, we explain how Manhattan distance is used to measure data locality and then, we compare the directly measured data locality counts of each non-zero ordering against the computed Manhattan distance. The purpose of comparing the two measurements is to show that the Manhattan distance is an effective means to estimating data locality without the necessity of executing SpMV directly measure cache hits or misses.

4.3.2 Estimating Data Locality Using Manhattan Distance

We are able to use Manhattan distance to estimate the data locality per matrix. Application of this metric is limited to a per matrix measurement that cannot be computed across all of the data, but is applied as a per matrix data locality metric. One advantage of using Manhattan distance to predict data locality execution time is that we are able to do the prediction without executing SpMV on any other data structures than COO. Further, this approach is an improvement over measuring data locality using cache and TLB misses, which requires that we execute the code on each data structure, at which point, the code has already executed and we should measure execution time directly. Lastly, we hypothesized that this approach would allow our model to estimate total execution time for the test set by computing the memory footprint component using several sparse matrix and machine characteristics, which were described in Section 4.2.

Manhattan distance measures the access distance between the non-zeros as the order of

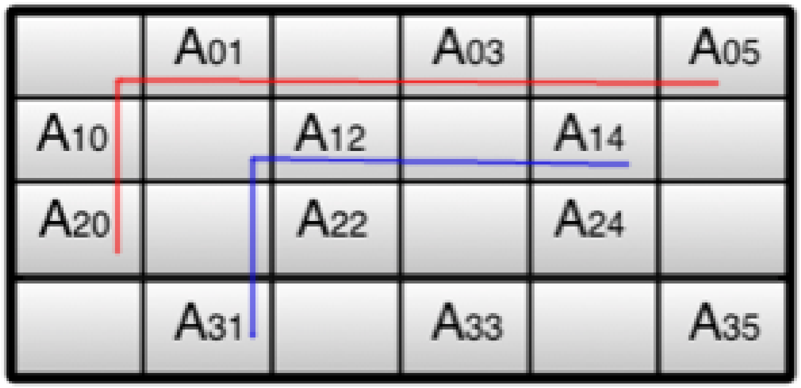


Figure 4.2: Illustration of How Manhattan Distance is Computed for a Sparse Matrix.

the non-zeros change along the rows and columns, which in effect measures the data locality present in the source and solution vectors due to the ordering of the non-zeros. Figure 4.2 shows how this metric is applied to an ordering of non-zeros in a sparse matrix to measure address access distances on the source and solution vectors. The distance measured along the path between non-zero values A_{20} an A_{05} is traced in red. The Manhattan distance traced along the length of the red path shown in Figure 4.2 is 19. Similarly, the blue line indicates the path along which a Manhattan distance between A_{31} and A_{14} is computed as 11. Using this metric, we are able to directly observe the change in execution time due to data locality per non-zero ordering. This address access distance measurement forms the basis for the data locality metric used in our initial model to predict execution time. The formula used to compute our data locality metric Manhattan distance (MD) follows:

$$MD = \sum((row\ distance) + (column\ distance)).$$

Since Manhattan distance changes with each new reordering of the non-zeros, we can relate the amount of data locality present in each ordering to execution time using this metric. The locality component of the model was developed using linear regression on the Manhattan distances and corresponding data locality execution times for the randomly selected training set. According to C. Annis [19], linear regression indicates the strength and the type of relationship that exists between two or more variables. In our case, we use linear regression to determine the strength and type of relationship that exists between Manhattan distance and data locality execution time. According to [20], linear regression is a method used to best fit a linear equation of the form:

$$Y(x) = a + bx$$

to a collection of data points (x_i, y_i) where $1 \leq i \leq N$, where (x_i, y_i) may refer to the Manhattan distance and DLT data points for a matrix or to all of the Manhattan distances and DLT values for all of the matrices. The slope is b and the y-intercept is a. First, we compute Sum_x over the independent variable x, which in our case is Manhattan distance.

$$Sum_x = \sum_{i=0}^{N-1} (x_i - \bar{x})^2$$

Then, we compute Sum_y over the dependent variable y, which is the data locality execution time (DLT).

$$Sum_y = \sum_{i=0}^{N-1} (y_i - \bar{y})^2$$

Then, we compute Sum_{xy} over the product of the differences as:

$$Sum_{xy} = \sum_{i=0}^{N-1} (x_i - \bar{x}) * (y_i - \bar{y}).$$

The slope of the linear regression line, b is computed as:

$$Slope_b = Sum_{xy}/Sum_x.$$

and the y-intercept, a, is computed as:

$$y - intercept_a = \bar{y} - b\bar{x}.$$

The formula for the correlation coefficient, R^2, is computed as:

$$R^2 = Sum_{xy}/\sqrt{Sum_x * Sum_{xy}}$$

A correlation coefficient (R^2), which is a measure of the strength of the linear fit, was computed both per matrix ordering and for a linear fit over the entire data set [21]. A correlation coefficient of 0.70 or greater indicates a strong linear relationship between components and the closer the value is to 1 indicates a stronger linear relationship [22]. The correlation coef-

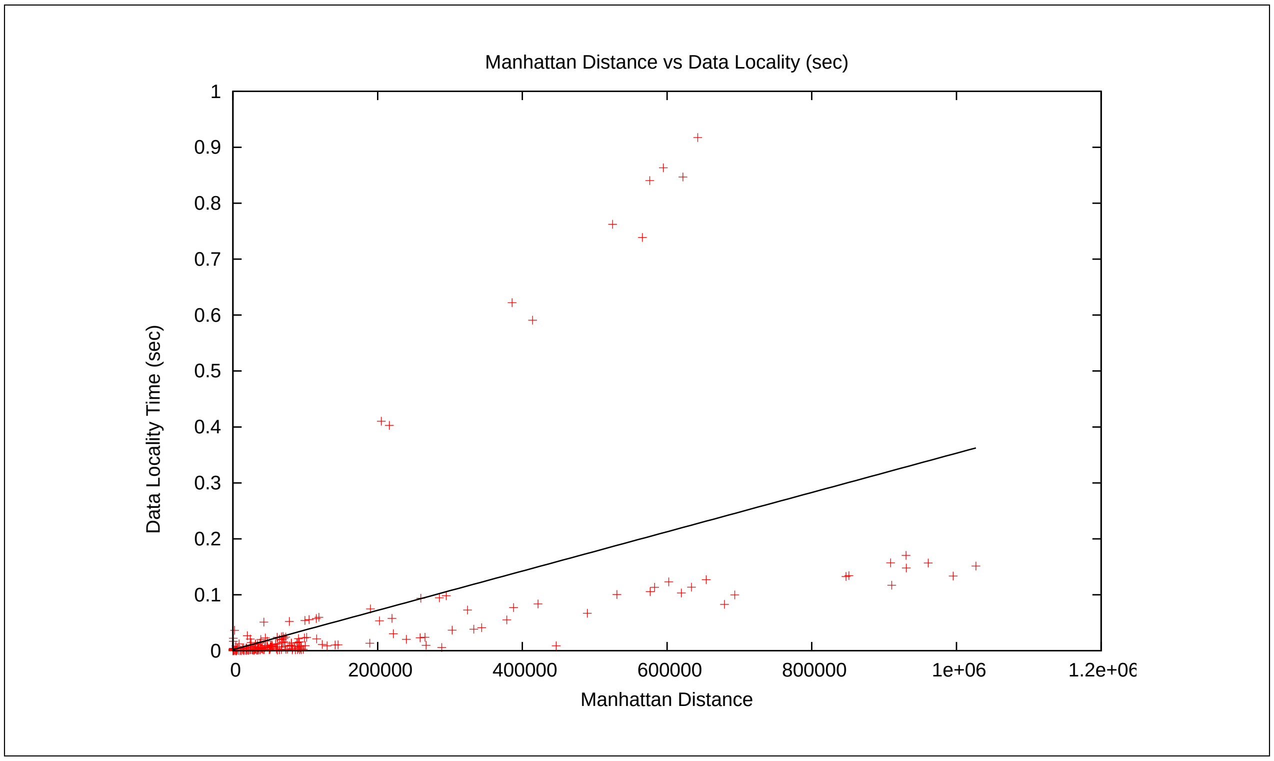


Figure 4.3: Linear Regression over all of the Data Points for Manhattan Distances and Data Locality Time

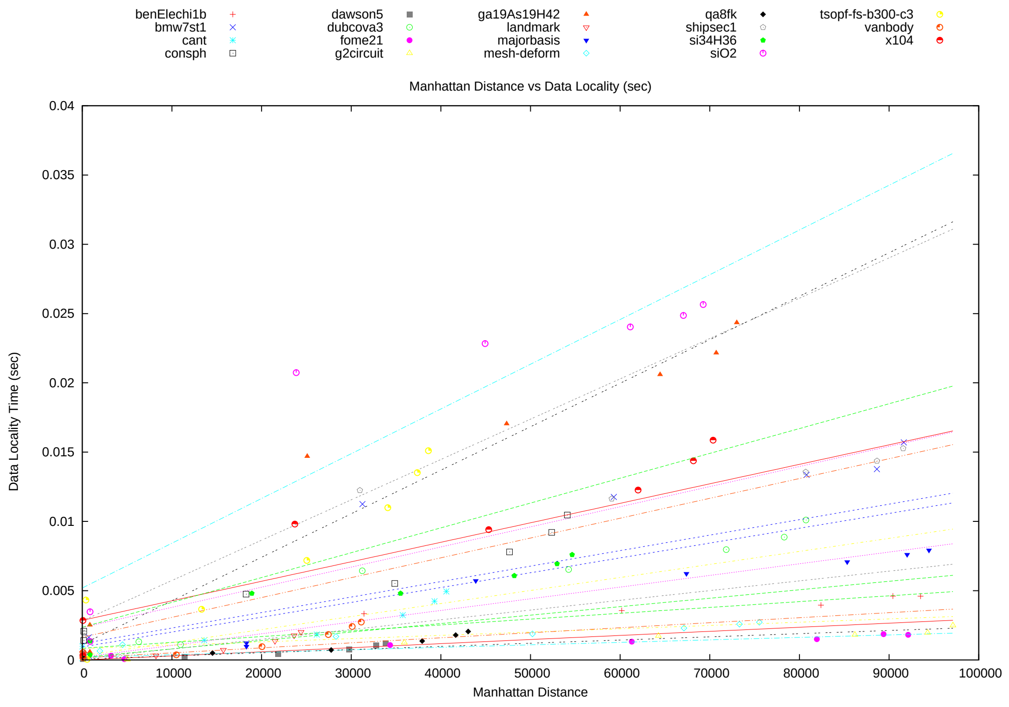
ficient for a linear fit over the various orderings of the entire data set was only 0.29 and the results are shown in Figure 4.3. The y-axis is the data locality execution time (DLT), which is computed as the difference between the total execution time, ET, and the best possible execution time attributed to the memory bandwidth ($MFlb$). The total execution time is an execution time averaged over 100 iterations of SpMV.

$$DLT = ET - MFlb$$

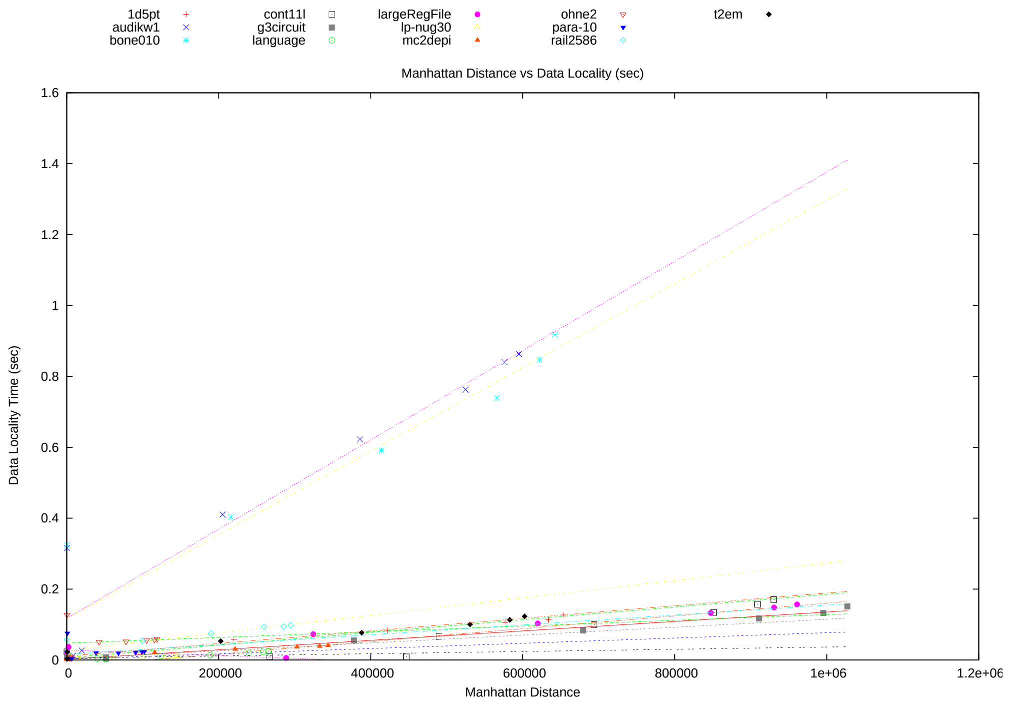
However, the average correlation coefficient for the Manhattan distance and data locality execution time per matrix reordering is 0.95 for the training set used to develop our hypothesis that Manhattan distance can be used to predict data locality execution time. A 95% correlation coefficient indicates a very strong linear relationship between Manhattan distance and data locality execution time per matrix for the listed orderings [23, 24]. Figures 4.4(a) and 4.4(b) show the preliminary results for the per matrix linear fit between the Manhattan distance of each non-zero ordering and data locality execution time. This example included the original COO ordering, a series of percentage randomized orderings, and a by row ordering. The data is divided into two graphs for readability. Figure 4.4(a) contains Manhattan data points that are less than 100K and Figure 4.4(b) contains Manhattan distances greater than or equal to 100K.

The y-axis in Figure 4.4 is the data locality execution time (DLT), which was computed in the equation above. The x-axis in Figure 4.4 is the Manhattan distance between all of the non-zeros in the matrix for the same reorderings shown in Figure 4.3 (COO original, randomized, and by row). Each Manhattan distance is normalized by the number of non-zeros in a matrix, so that data locality of one matrix may be compared to the locality of other matrices in the data set. Given the preliminary by-matrix linear regression results, we hypothesized that Manhattan distance could be used as a data locality metric in the performance model and serve as an indicator of data locality execution time.

Figure 4.5 shows the number of cache and TLB misses per Manhattan distance for the various non-zero reorderings. The y-axis is the number of misses and the x-axis is the Manhattan distance. Three separate trend lines are shown for the L1, L2, and TLB misses. The correlation coefficients for the linear fits shown for the L1, L2, and TLB trend lines are 0.21, 0.25, and 0.24 respectively. The poor quality of these correlation coefficients is indicative of the difficulty of developing a performance model that works for all matrices. This figure shows us that a small Manhattan distance corresponds to a fewer number of cache and TLB misses, which is indicated by the clustering of data points for Manhattan distances less than 200K. This graph also shows that as the Manhattan distance increases



(a) Manhattan Distance < 100K



(b) Manhattan Distance >= 100K

Figure 4.4: Linear Regression per Matrix Manhattan Distance and Data Locality Time: Using Manhattan Distance to Predict Data Locality.

to 600K, the data locality worsens as indicated by a sharply increasing number of cache and TLB misses. Also, there are a few data points, which have large Manhattan distances (greater than 400K) and a relatively small number of misses. This group corresponds with 7 very large matrices. Five of these 7 matrices are the most sparse matrices in the training set and 2 of the matrices are the largest. The largest matrices in the training set correspond with the matrices that have the most non-zeros, but these matrices are also some of the most sparse in the training set. All 7 matrices are among the top 3% in sparsity based on

$$ave\ \%sparsity = (ave\ sparsity\ of\ these\ 7\ matrices)/$$

$$(average\ sparsity\ of\ the\ training\ set).$$

However, relative to the averages for other matrices in the training set, these matrices have

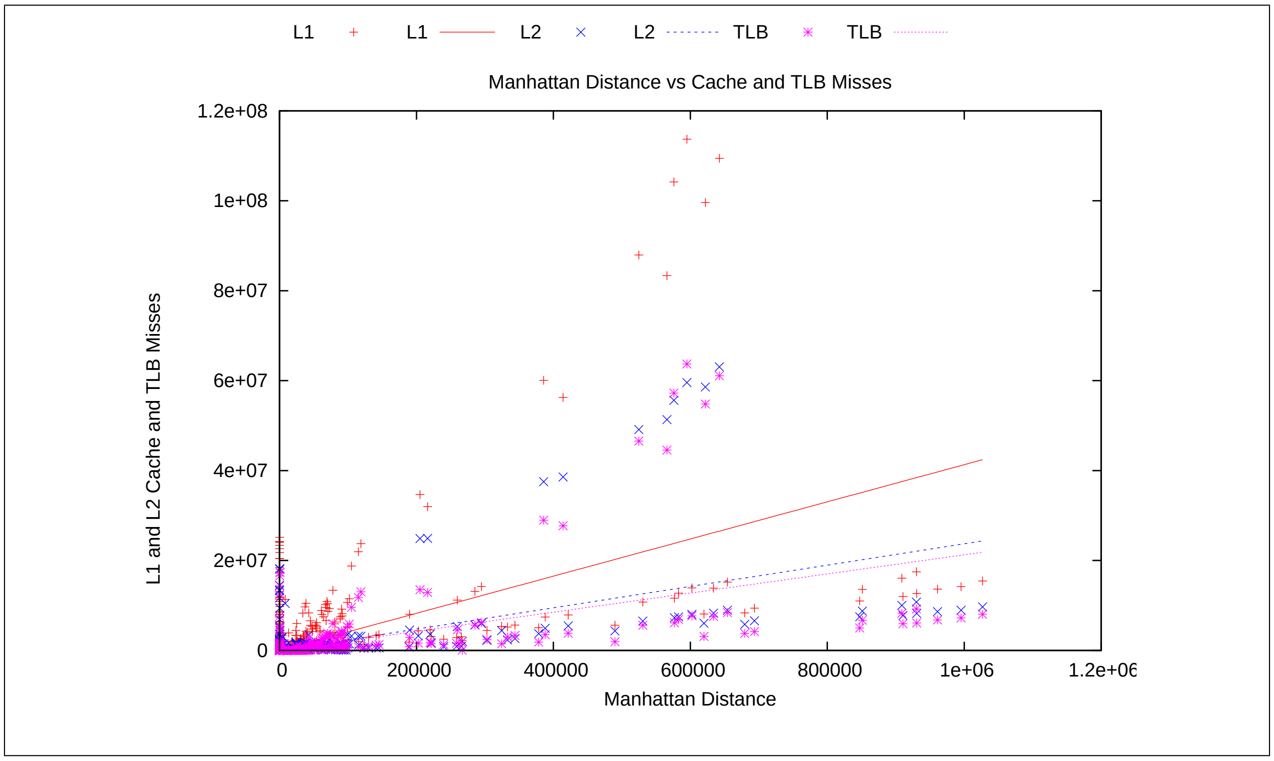


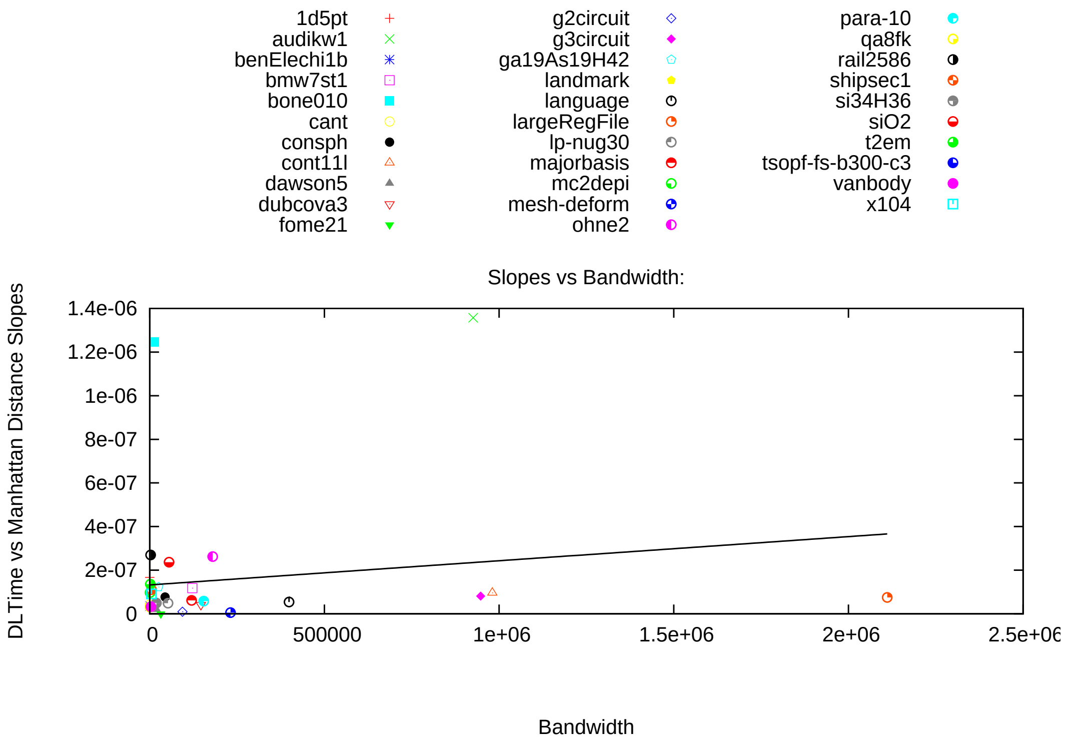
Figure 4.5: Cache and TLB Misses per Manhattan Distance for all COO Non-zero Orderings

a larger nnz/col and very few nnz/row. Few nnz/row would produce a larger Manhattan distance measured along the rows. More nnz/col indicates a increased opportunity for temporal reuse on the source vector and fewer cache misses. These factors combined ameliorate the high degree of sparsity and explains why these matrices have relatively few cache misses, yet large Manhattan distances.

4.4 Predicting Data Locality Time

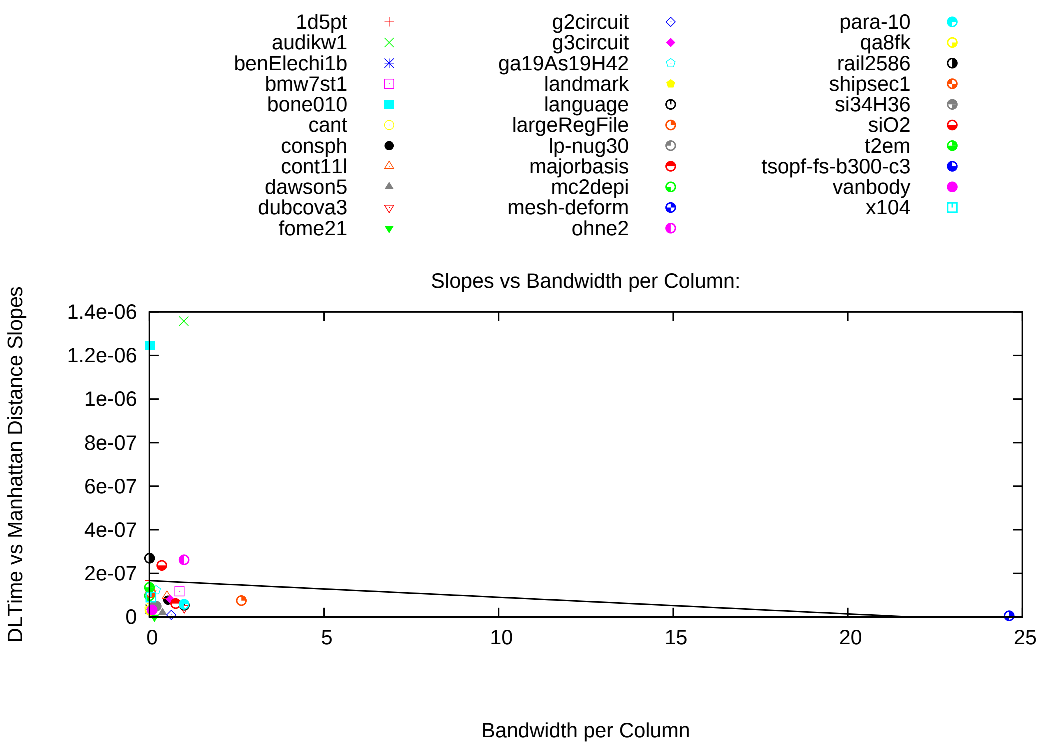
The Manhattan distance metric does well per matrix, but not amongst all of the data as shown by the poor linear fit across the data in Figure 4.3. Therefore, in order to predict data locality execution time across all of the matrices in the data set, we hypothesized that it is necessary to use other data locality metrics. Using a metric that has a linear relationship with execution time across all of the data set would better enable the model to predict the data locality execution time. Each line in Figure 4.4 shows a linear fit for each matrix between Manhattan distance and DLT. We decided to compute a linear fit between each of the 5 data locality metrics (matrix bandwidth, matrix bandwidth per ncols, sparsity, nnz per row, and nnz per cols) and each slope from each of the lines shown in Figure 4.4. Then, we repeated the process using the same metrics as our x_i and each of the y-intercepts from each of the lines shown in Figure 4.4 as our y_i. The formulas for performing linear regression over (x_i, y_i) were explained earlier.

We tried using many different data locality metrics to model the slopes and y-intercepts from Figure 4.4 for the data locality portion of the performance model. The approach computed a linear regression fit and correlation coefficient between each metric and the slopes of the Manhattan distance plot and between each metric and the y-intercepts in the plot. The data locality metrics that we tried were matrix bandwidth, matrix bandwidth per ncols, sparsity, nnz per row, and nnz per cols. However, no strong correlation existed between either the data locality execution time slopes or y-intercepts and each of the metrics shown in Figures 4.6, 4.7, 4.8, 4.9, 4.10, and 4.11. Each linear regression plot in Figures 4.6, 4.7, 4.8, 4.9, 4.10, and 4.11 is labeled with a corresponding correlation coefficient (R^2) value, which indicates the strength and type of relationship that may exist between the metric and slopes or y-intercepts. The largest R^2 value computed for any of the metrics was 0.29, which occured between the y-intercepts from Figure 4.4 and matrix bandwidth.

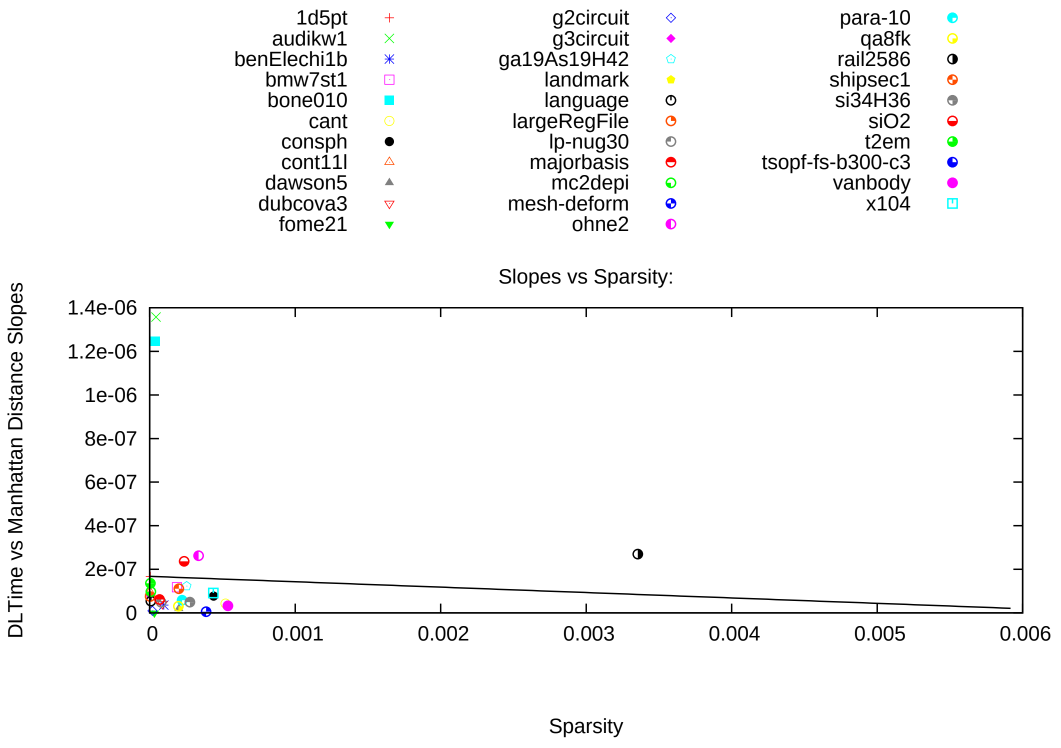


(a) Slopes vs Bandwidth $R^2 = 0.025$

Figure 4.6: Linear Fit for Data Locality Execution Time Slopes versus Data Locality Metric per Matrix: Bandwidth.

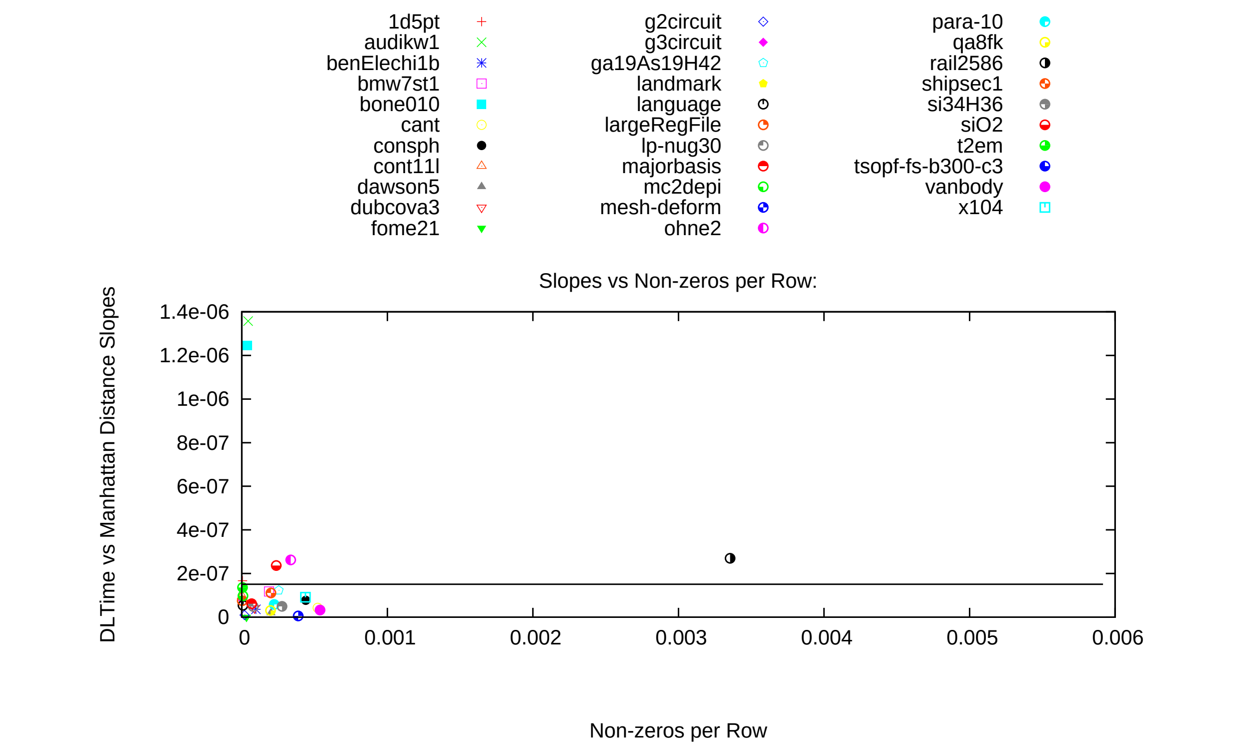


(a) Slopes vs Bandwidth/NCols: $R^2 = 0.012$

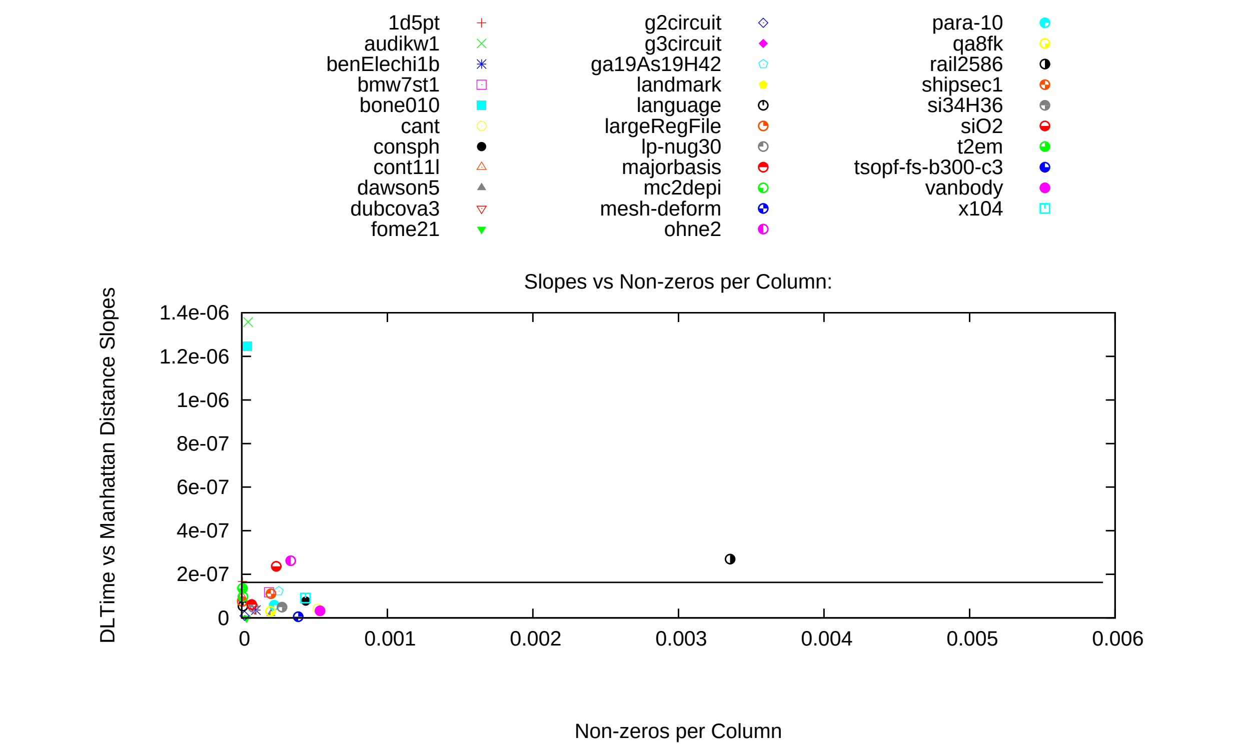


(b) Slopes vs Sparsity $R^2 = 0.056$

Figure 4.7: Linear Fit for Data Locality Execution Time Slopes versus Data Locality Metrics per Matrix: Bandwidth per Number of Columns and Sparsity.

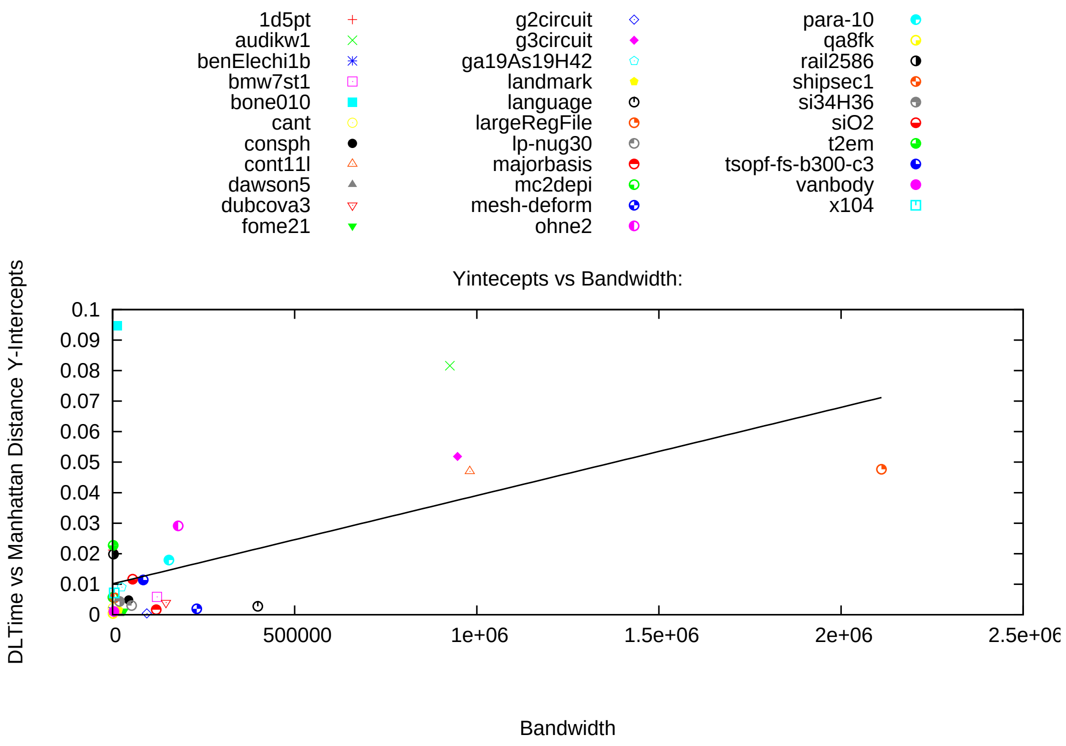


(a) Slopes vs NNZ per Row $R^2 = 0.006$



(b) Slopes vs NNZ per Cols $R^2 = 0.002$

Figure 4.8: Linear Fit for Data Locality Execution Time Slopes versus Data Locality Metrics per Matrix: NNZ per Row and NNZ per Column.



(a) Y-int vs Bandwidth $R^2 = 0.29$

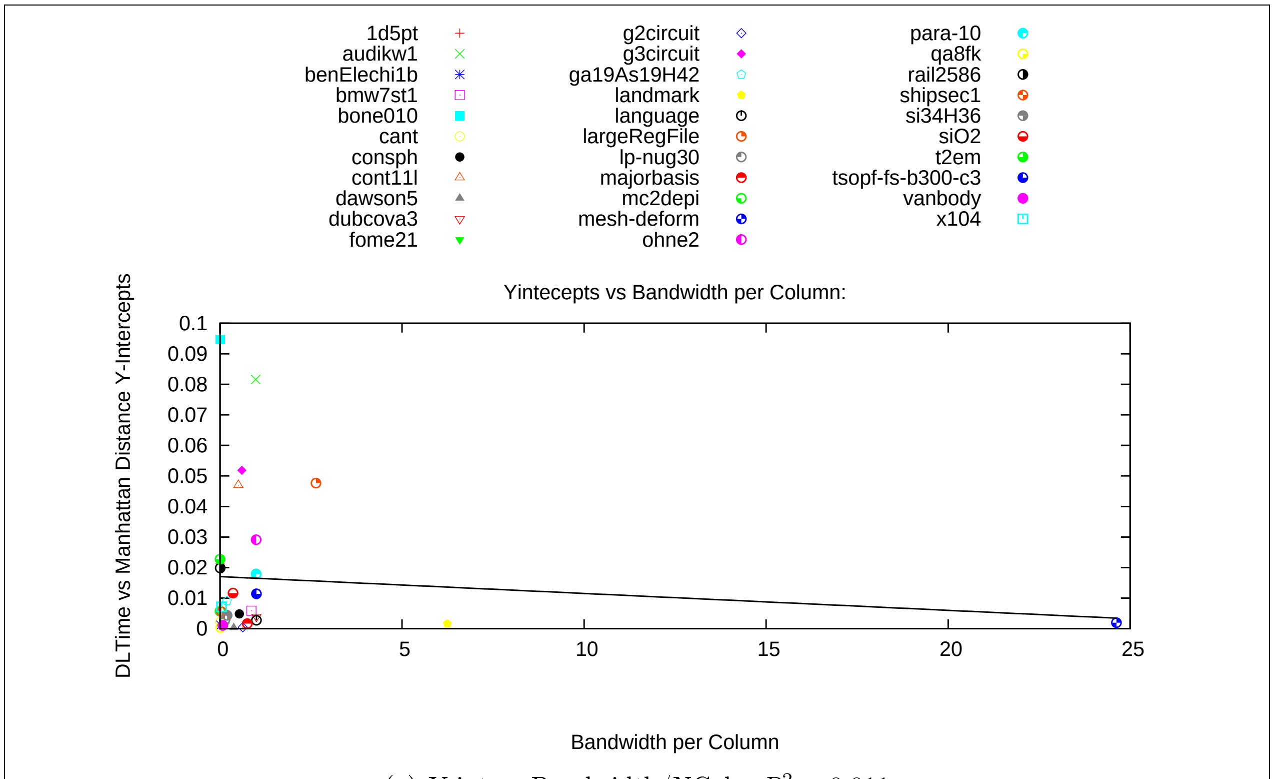
Figure 4.9: Linear Fit for Data Locality Execution Time Y-intercepts versus Data Locality Metric per Matrix: Bandwidth.

The primary difference between these other data locality metrics and Manhattan distance is that Manhattan distance changes with each reordering of the non-zeros and the other metrics do not. Each of the chosen metrics shown in Figures 4.6, 4.7, 4.8, 4.9, 4.10, and 4.11 are primarily indicative of the amount of reuse that occurs either on the source or solution vector during the SpMV computation. Matrix bandwidth ($matrixBW$) is the maximum distance between non-zeros in a row of a matrix computed by Gibbs et al., [25] using the following formula:

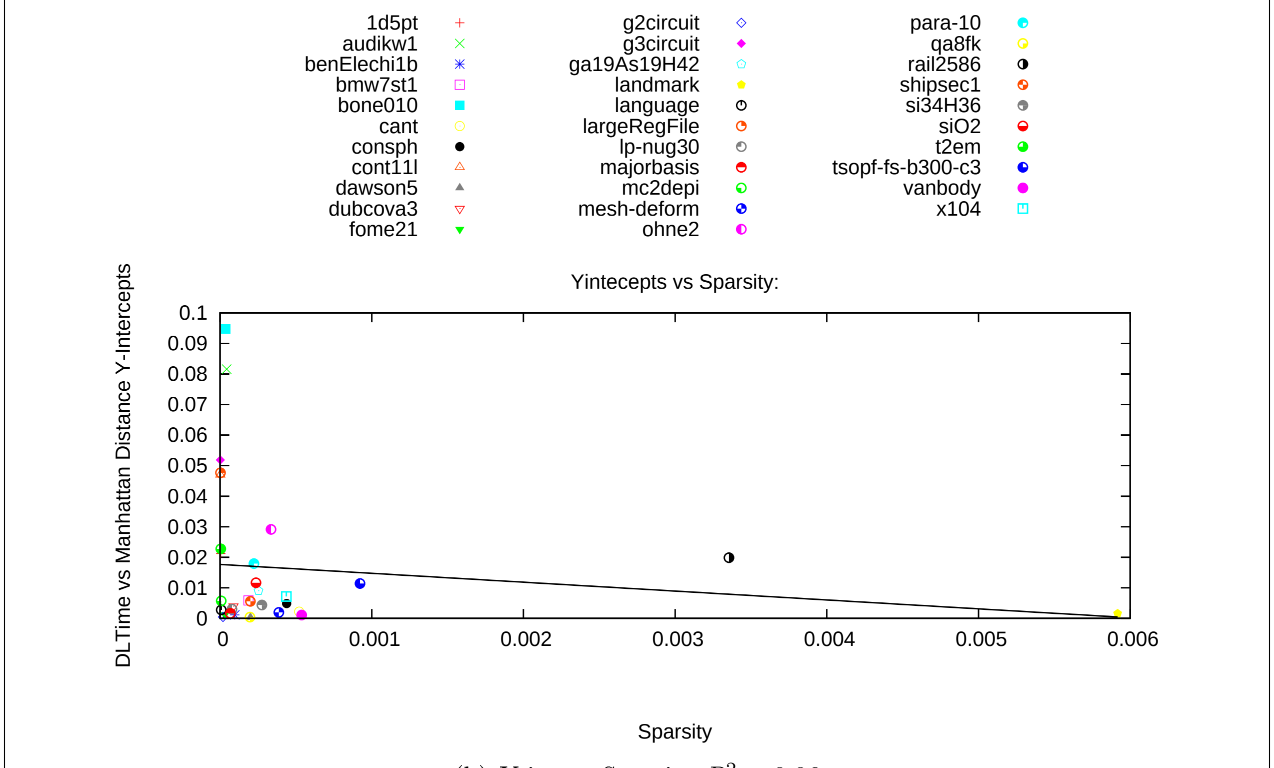
$$matrixBW = max|i - j|$$

$$a_{ij} \neq 0$$

The bandwidth of a matrix is the same for any permutation of the non-zeros in a matrix, where each non-zero is a tuple consisting of the row index, column index, and value. Matrix bandwidth, matrix bandwidth per number of columns, and the average nnz per row indicate

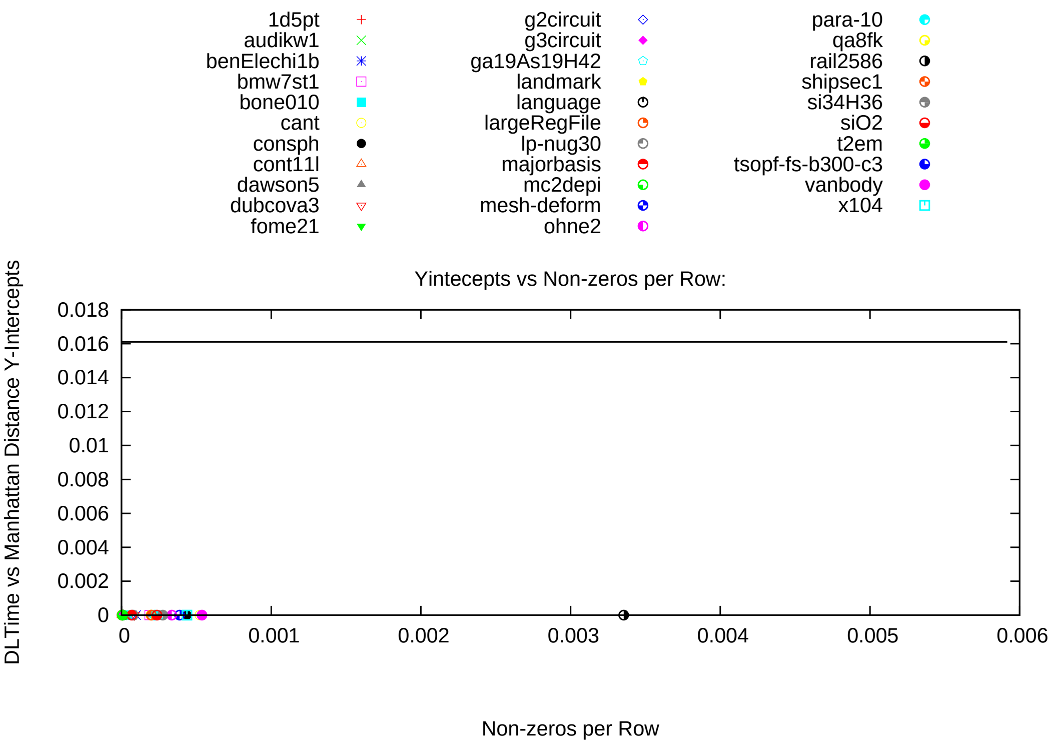


(a) Y-int vs Bandwidth/NCols: $R^2 = 0.011$

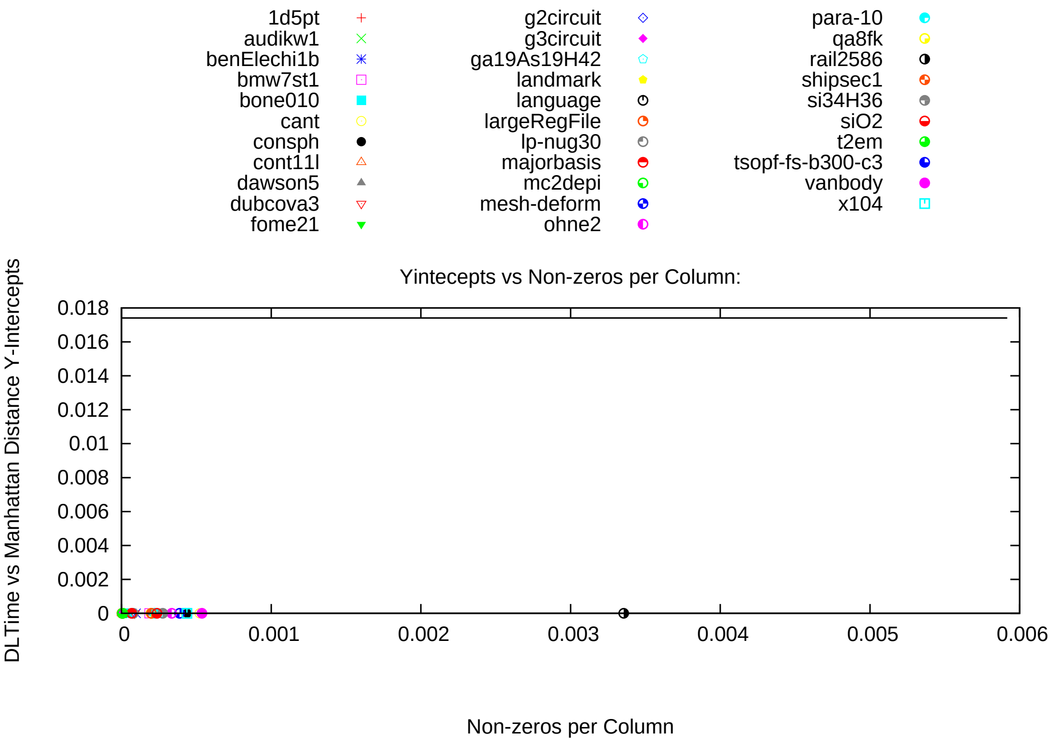


(b) Y-int vs Sparsity $R^2 = 0.06$

Figure 4.10: Linear Fit for Data Locality Execution Time Y-intercepts versus Various Data Locality Metrics per Matrix: Bandwidth per Number of Columns and Sparsity.



(a) Y-int vs NNZ per Row $R^2 = 0.001$



(b) Y-int vs NNZ per Cols $R^22 = 0.009$

Figure 4.11: Linear Fit for Data Locality Execution Time Y-intercepts versus Various Data Locality Metrics per Matrix: NNZ per Row and NNZ per Column.

the potential spatial locality and temporal reuse that occurs on the solution vector if the same data is accessible in a cache line between multiple accesses. These two metrics are also indicative of spatial locality for SpMV, which occurs on the source vector when contiguously stored data is fetched from the cache and used. A small matrix bandwidth, but not less than one, or many non-zeros per row may correspond with greater spatial locality. The average nnz per column indicate how much reuse may occur on the source vector if the temporal reuse occurs close enough in time that the data is still in the cache. Sparsity is the $nnz/(nrows \times ncols)$, which indicates the amount of spatial and temporal locality that occurs on both the source and solution vectors when contiguously stored data is accessed within a short enough time period that that the cache line being used is still in the cache.

Linear regression between the Manhattan distance slopes and matrix bandwidth produced a positive slope and R^2 value of 0.025, which does not indicate a strong linear relationship between the metric and the slopes. However, we found that of the 5 metrics used to model Manhattan distance slopes, that the matrix bandwidth produced the best accuracy in predicting relative performance. No metric was discovered, which models the y-intercept for the data locality execution time as well as the actual y-intercepts. Without a metric to model the y-intercepts we are unable to predict the ordering that will produce the best performance.

4.5 Impact of Indexing

An initial model with just memory bandwidth requirements and Manhatten distance did not sufficiently model some of the sparse matrix data structures we investigated, specifically the cache block data structure. Figure 4.12 shows the result of performing linear regression on Manhattan distance and data locality execution time after the addition of the cache blocked ordering to include 2K, 8K, and 32K columns per cache block.

The addition of the cache block orderings resulted in the negative slopes that appear in Figure 4.12. Recall from previous chapters that our implementation of the cache blocked data structure has the overhead of an indexing array into each row of a cache block. Smaller

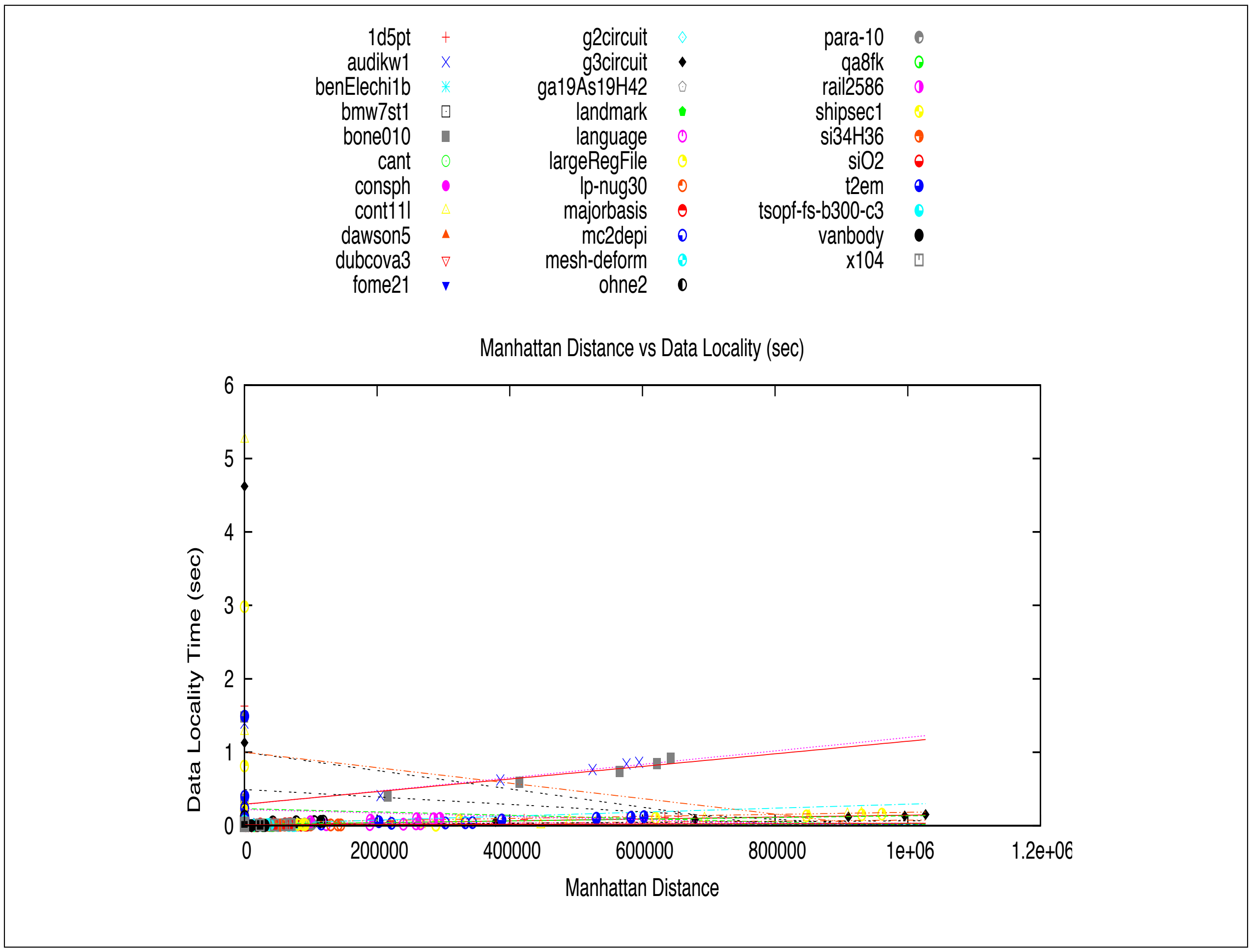


Figure 4.12: Using Manhattan Distance to Predict Data Locality per data structure and orderings, which includes the coo original, coo random, coo by row, and cache blocked ordering with 2K, 8K, and 32K columns per cache block.

cache block sizes generate a greater number of blocks than larger cache block sizes for a matrix, thereby, increasing the overhead of the indexing array that points to each row of a cache block. Also, smaller cache blocks increase the likelihood of empty rows in a cache block, which are not skipped over. If there are a large number of empty $cache\ block\ \times nrow$ combinations then, the indexing overhead of the data structure starts to contribute to the execution time.

The negative slopes in Figure 4.12 correspond to an increased execution time as a result of the indexing overhead associated with smaller cache block sizes. To illustrate the point, let us consider the t2em matrix, which is an 921,632 x 921,632 matrix with 4,590,832 non-zeros. Using a cache block size of 2K columns per cache block, the cache blocking data structure must iterate over 451 $cache\ blocks\ \times 921,632\ rows$, which amounts to 415,656,032 row index accesses far exceeding the nnz in the matrix. By comparison, a cache block size of 32K columns per cache block only requires 29 $cache\ blocks\ \times 921,632\ rows$, which requires 26,727,328 row accesses. This data structure requires that the smaller cache block size of 2K columns per cache block perform 16 times the number of row index accesses as that of the 32K column cache block, which greatly increases the indexing overhead and execution time.

4.5.1 Computing Indexing Overhead

Each data structure is associated with a different indexing overhead. Recall from Chapter 2 that the simplest data structure, COO, is a flat data structure with a 1 to 1 mapping between each row and column index to a value and has an algorithm complexity of $\Theta(nnz)$. Indexing overhead associated with COO is equal to the number of index accesses to nnz.

SpMV executed on the CSR data structure has an algorithmic complexity of $\Theta(nrows + nnz)$ and performance is dominated by the indexing overhead of the row pointer array, which indexes into the first non-zero of each row. Finally, our implementation of the cache blocked data structure has an algorithm complexity of $\Theta(ncb\ \times nrows) + nnz$, where ncb refers to the number of cache blocks generated. The indexing overhead associated with the cache blocking data structure is the array that indexes into each cache block row. As the size of that array increases to the nnz in the matrix, the the overhead greatly impacts performance and must be accounted for in the performance model. In 4.6 we discuss in detail how indexing overhead is computed in the model for the COO, CSR, and cache blocked data structures.

4.6 Training the Model

The most general form of the model used to predict the relative performance of SpMV is

$$Execution\ time\ = (Time\ to\ fetch\ data\ from\ memory\ once) +$$

$$(Indexing\ overhead\ for\ cache\ blocked\ data\ structure) + (Time\ to\ refetch\ data).$$

The next two sections discuss each of the components of the model in much greater detail, beginning with the memory component, which is used to predict the execution time attributed to by the memory bandwidth requirements.

4.6.1 Computing the Time to Fetch the Data from Memory Once

Much of the details describing how we compute the memory component of the model were discussed in the Bounding Performance with Memory Access Time under Section 4.2. To recap, we compute the minimum memory bandwidth requirement for the computation based on the assumption that the matrix, source vector and solution vectors are each read in only once, such that that the

$$minBW = (memory\ footprint\ of\ a\ data\ structure) + (8 * nrows) + (8 * ncols).$$

We multiply the number of rows and columns by 8 because the source and solution vectors, X and Y, hold doubles, which are 8 bytes each. The formulas used to compute the memory footprints for each of the data structures in the analysis was discussed in Section 4.2. The best possible execution time attributed to the memory bandwidth is computed for each data structure using the machine bandwidth, which was measured using the STREAM benchmark [13] as $MFlb = (least\ memory\ required/Machine\ Bandwidth)$, where $MFlb$ is the best possible execution time for a given memory footprint and SpMV.

4.6.2 Computing the Data Locality Component

In this section we discuss how we modeled the execution time attributed to data locality, where data locality time (DLT) is computed as

$$DLT = ET - MFlb,$$

and ET refers to total execution time. In Section 4.4 we discussed the limitations of using Manhattan distance to predict data locality execution time per matrix and the need to model the slope and y-intercept for the data locality portion of the performance model. We tried to model the slope and y-intercept using multiple phases of linear regression using 5 different data locality metrics. The results of using linear regression to select an appropriate data locality metric to model the slopes and y-intercepts were shown in Figures 4.6, 4.7, 4.8, 4.9, 4.10, and 4.11.

4.6.3 Data Locality Model 1 Based on Linear Regression of Manhattan Distance

The model for the data locality component is composed a model slope, a data locality metric, and a model of the y-intercept. Initally, we applied this approach using linear regression to predict the model slope and model y-intercept. The initial model for data locality took the form:

$$ET - MFlb = (model\ slope\ * MD) + (model\ y - intercept),$$

where MD refers to the Manhattan distance. Taking the preliminary model a step further, we can decompose the model slope and model y-intercept as follows:

$$(a0\ * data\ locality\ metric1\ + a1) * Manhattan\ Distance$$

$$+(b0\ * data\ locality\ metric2\ + b1).$$

In this equation, a0 and a1 are the slopes and y-intercepts resulting from computing the linear fit between the 5 data locality metrics and the slopes taken from the per matrix linear fit between Manhattan distance and DLT. The results of the the linear regression between the slopes and each metric were shown in Figures 4.6, 4.7, and 4.8. Similarly, we attempted to model the y-intercept component of the data locality model by computing a linear regression between each of the 5 data locality metrics identified in Figures 4.9, 4.10, and 4.11 and the y-intercept taken from the linear regression between Manhattan distance and DLT, which was not successful. The y-intercept component is modeled above using b0, b1, and data locality metric2. This initial approach of modeling the slopes and y-intercepts did not work because no strong linear relationshiop exists between the 5 data locality metrics and the model slope or model y-intercept.

4.6.4 Data Locality Model 2 Based on Multi-parameter Linear Regression

The results of this initial model prompted an approach, which uses a multi-parameter linear fit library from the GNU Scientific Library [26] and accounts for the indexing overhead discussed in Section 4.5.1. This approach uses the formula $L = Pc$ to solve for the coefficients in c, which result from a multi-parameter linear fit on various data locality metrics, which is used to model the locality component of the performance model. L is an $n \times 1$ vector of data locality execution time values and P is an $n \times ncoeffs$ matrix where $ncoeffs$ refers to the number of coefficients.

The input parameters to the model include the L vector, which is equal to the $ET - MFbest$. The 8 input parameters for matrix P include 1, MD , IOH, nnz, $(BWCols*MD)$, $BWCols$, $(matrixBW * MD)$, and BW are used to solve for the coefficient vector c.

Initially, the input parameters included all 5 of the data locality metrics modeled in Figures 4.6, 4.7, 4.8, 4.9, 4.10, and 4.11. The reason being that very small coefficients would result for those parameters if the fit was poor and essentially cancel out the effect of those poor fitting parameters. However, the fit was unable to converge with sparsity, nnz per row and column. The IOH and nnz components are used to model indexing overhead. We include nnz because the overhead of the indexing arrays is due in-part to the non-zeros contained in the rows, which are indexed into by the row pointers of the CSR and cache blocking data structures. Also, $BWCols$ parameter refers to the bandwidth per number of columns in the parameter list above. The multi-parameter linear fit approach results in the following formula used to predict $ET - MFlb$ time.

$$ET - MFlb = a + (b * MD) + (c * IOH) + (d * nnz) +$$

$$(e * BWCols * MD) + (f * BWCols) + (g * BW * MD) + (h * BW).$$

We can decompose this formula into 3 components:

$$Modeled\ slope\ = a + (b * MD),$$

$$Modeled\ indexing\ overhead\ = (c * IOH) + (d * nnz),$$

and

$$Modeled\ y - intercept\ = (e * BWCols * MD) + (f * BWCols) + (g * BW * MD)$$

$$+ (h * BW).$$

Assembling all of the components together produces our second performance model:

$$\boxed{\begin{array}{c} ET = MFlb + a + (b * MD) + (c * IOH) + (d * nnz) + \\ (e * BWCols * MD) + (f * BWCols) + (g * BW * MD) + (h * BW). \end{array}}$$

Figure 4.13: Model Used to Predict the Relative Performance of SpMV.

Table 4.2 shows a representative example of the coefficients obtained using the model shown in Figure 4.13 after executing on the Intel Core2 Duo E8300 architecture. A negative coefficient indicates that the parameter has a negative relationship with L [27]. A coefficient value close to zero indicates that no relationship exists between the parameter and L.

Table 4.2: Multi-parameter Linear Regression Fit Coefficients

Term	Coefficient
a:	$3.0x10-^1$
b:	$-5.5x10^-7$
c:	$7.0x10^-8$
d:	$8.6x10^-9$
e:	$1.4x10^-7$
f:	$2.9x10^-2$
g:	$1.8x10^-12$
h:	$-1.3x10^-6$
i:	$-4.9x10^-2$
j:	$-1.5x10^-5$

The parameters with the strongest relationship with L are the constant coefficient a and $BWCols$ coefficient f. We observed this same pattern of coefficient strength on the other two machines as well, which we hypothesized indicated that the $BWCols$ and coefficient a are good predictors of relative performance over the data set. However, $BWCols$ does not change with data locality and we found that the R^2 value between $BWCols$ and DLT indicated that no relationship exists between them. This discovery lead to our third and final model.

4.6.5 Data Locality Model 3 Based on Average Predicted Data Locality Time

The constant coefficient a in Table 4.2 is by far the strongest coefficient. Based on this finding we determined that modeling data locality time as an average is a better performance indicator than any linear fit to the terms shown in Table 4.2. The final model follows:

$$ET = MFlb + (ave\ DLT\ remaining).$$

Chapter 5

Evaluating the Performance Model

Our initial performance models use various data locality metrics to model data locality execution time, but the average relative error for predicting execution time using those models was 111% for the first model and 254% for the second model. Our final performance model for predicting actual execution time had an average relative error of only 7.2%, but this performance model does not model data locality and therefore is able to predict the fastest performing data structure and not the optimal non-zero ordering. However, we identified Manhattan distance as a metric that is capable of estimating the impact of data locality on execution time for different non-zero orderings of a particular matrix.

The purpose of this performance model is to select the non-zero ordering AND data structure, which will produce the fastest execution time for the SpMV computation and a given set of machine and matrix parameters. We hypothesized that given a set of possible sparse data structures (in terms of their memory footprint equations) and an estimate of the data locality of possible orderings (in terms of Manhattan distance), the performance model could select the data structure and ordering that will produce the fastest performance. However, the first two models failed to determine the impact of data locality on execution time.

Our approach was to select a representative training set, perform the training as described in Section 4.6, and then evaluate the accuracy of the execution time model on another set of sparse matrices.

However, using multiple phases of linear regression and additional data locality matrices, our model is unable to predict which non-zero reordering produces the fastest execution times for the data set. Ideally, we could predict execution time, but the average relative error for predicting execution times was too high. Our SpMV performance model was developed on three Intel architectures: two Intel Core2 Duo architectures E8300 and 6400, and an Intel Xeon (Harpertown).

5.1 Selecting Representative Training Matrices and the Testing Set

Ninety-six matrices with a memory footprint greater than 6MB were randomly selected from the Florida sparse matrix collection [28] and benchmarked on SpMV. A table of the 32 matrices that compose the training set are shown in Table A. The other 64 matrices, which make up our test set are shown in the other two tables of the Appendix. The size of the cache in the machines used for our experiments were 6MB, and we chose matrices that exceeded the size of the caches used in our experiments. Matrices with a memory footprint, which exceeds the size of the cache, ensured our ability to observe cache misses and the impact of memory bandwidth pressure on performance.

In order to ensure that the training set of 32 matrices was representative of the entire data set of 96 matrices, the training set was selected by mapping each matrix in the entire data set to a 3 dimensional space composed of several locality metrics: sparsity, the average nnz per row, and the average nnz per column for each matrix. Sparsity is the $nnz/(nrows \times ncols)$, which indicates the amount of reuse that occurs on both the source and solution vectors and is also indicative of spatial locality as is the average nnz per row. The averge number of non-zeros per column indicate the amount of reuse that occurs in this computation for a particular matrix. Columns in the tables of Appendix A include the memory footprint(MB), nnz/row, $nnz/column$, and sparsity for each matrix in the data set.

Randomly selecting a data set of 32 matrices from this space produced a training set that is representive of some of the data locality characteristics present across the set. Figure 5.1 shows this 3 dimensional space that each of the 96 matrices were mapped into. A random sampling approach that divides that sampling space into evenly sized blocks from which to sample was used to select 32 matrices for the training set. A 4×8 grid was superimposed over the 3D space to randomly sample 1 matrix from within each block [29]. Elements of the training set were selected nearest the center of each of the 32 blocks. If no sample was present inside of a block then, no sample was collected from that block. In such a case, multiple samples were collected randomly from another block, which was selected such that the sampling was evenly distributed over the rows and columns of the 4×8 grid. No more than two samples were collected from a single block and only 5 of the blocks were double sampled due to 5 empty blocks within the 4×8 grid.

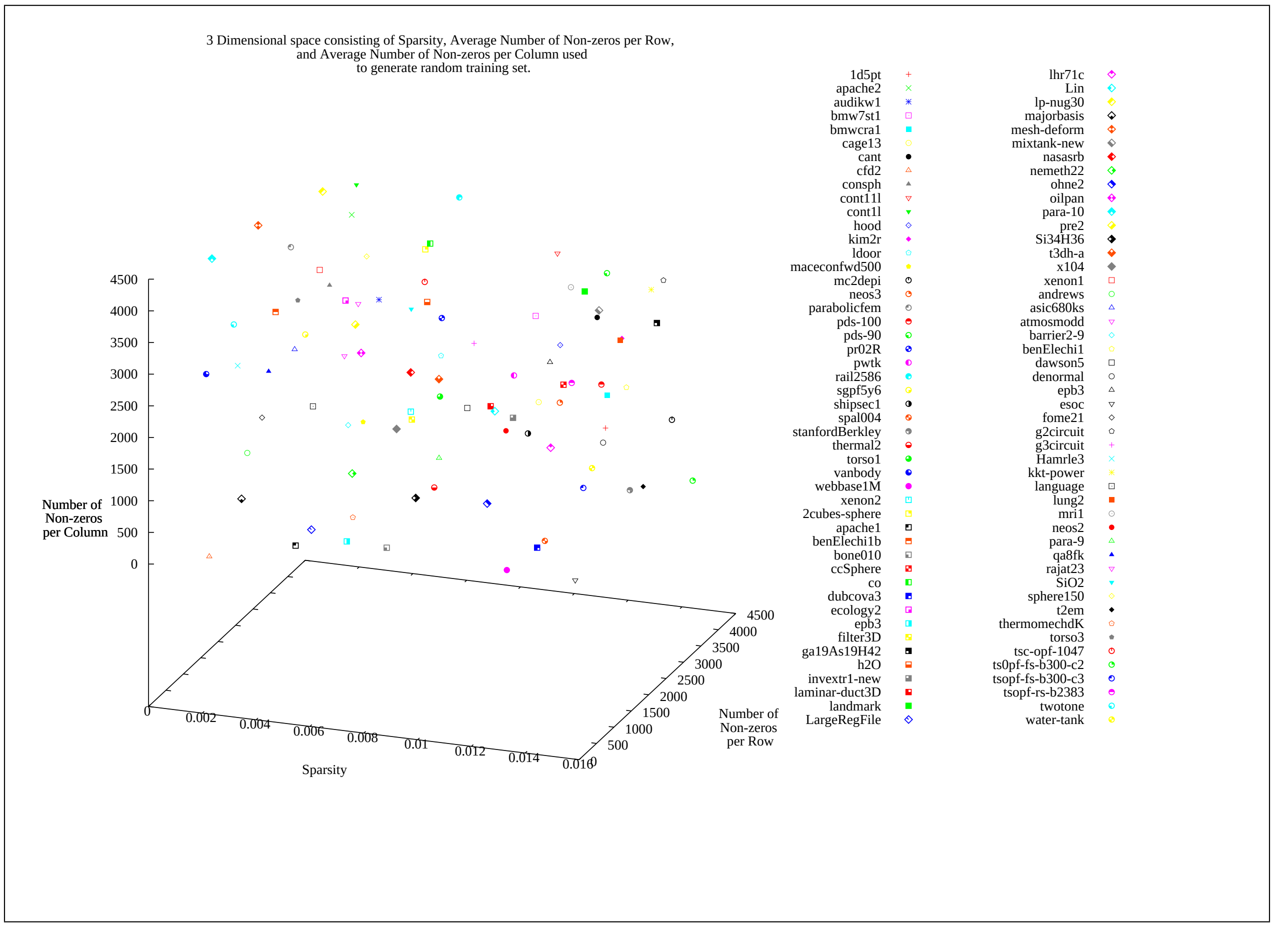


Figure 5.1: Random Selection of Training Data Set

5.2 Models for Intel Core2 Duo and Intel Xeon Architectures

We evaluate each model based on its ability to predict execution time, select the best performing data structure and reordering, and performance degredation when it does not. Using the formulas for each model, the remainder of the data set was benchmarked and modeled using each model's components. The accuracy for the execution time prediction and the relative performance prediction follow.

5.3 Accuracy of Execution Time Prediction

The average relative error for predicting execution time on the test set for the final model is 7.2%. The error is computed for the three models using actual execution time (ET) and model execution time (MET).The formula used to compute relative error of execution time follows:

Execution time prediction error:

$$\%error\ in\ predicted\ execution\ time = 100 * (ET - MET)/ET$$

5.4 Accuracy of Relative Performance Prediction

In this section we evaluate the relative performance of our model based on its ability to accurately predict the data structure that will produce the fastest execution times per matrix. The average accuracy for predicting, which data structure will produce the fastest execution times using our final model is 78.6%. Relative performance is computed by comparing the execution time of the data structure that actually performs the fastest against the execution time of the data structure recommended by the model.

Percent Correct:

$$\%of\ Time\ model\ picks\ winner\ =$$

$$(100 * total\ number\ of\ correct\ pick)/$$

$$(total\ number\ of\ matrices)$$

Percent Degredation:

$$\%degredation\ in\ performance\ when\ model\ fails\ =$$

$$100 * (ET - ET\ of Model\ recommended\ data\ structure)/ET$$

The results for model error computation are shown in below in Tables 5.1, 5.2, and 5.3.

Model No.	% Correct	Ave. % Degredation	Ave. % Exec Time Pred Error	R^2
1	48.4%	49.8%	25.7%	0.38
2	14.0%	265.4%	228.1%	-0.019
3	84.3%	38.1%	5.95%	0.002

Table 5.1: Error prediction results for the 3 models executed on Intel Core2 Duo E8300 on a test set of 64 matrices.

Model No.	% Correct	Ave. % Degredation	Ave. % Exec Time Pred Error	R^2
1	0%	257.9%	257.9%	0.34
2	12.5%	213.8%	187%	0.009
3	78.1%	17.8%	3.9%	0.003

Table 5.2: Error prediction results for the 3 models executed on Intel Xeon E5405 on a test set of 64 matrices.

Model No.	% Correct	Ave. % Degredation	Ave. % Exec Time Pred Error	R^2
1	15.6%	62%	52%	-0.62
2	14%	405%	348%	-0.074
3	73.4%	44.2%	11.7%	0.002

Table 5.3: Error prediction results for the 3 models executed on Intel Core2 Duo 6400 on a test set of 64 matrices.

In order to better understand why our third model is able to predict the best data structure let us look at an example. Figure 5.4 shows the case where data locality determines execution time. The model predicts that the CSR data structure is the fastest performing, but the actual fastest in this case is the COO. We computed the percent difference for the memory footprint between CSR and COO as:

$$\%Memory\ Footprint\ Error = (COO\ Footprint\ - CSR\ Footprint)/$$

$$COO\ Footprint$$

and found a 21% difference.

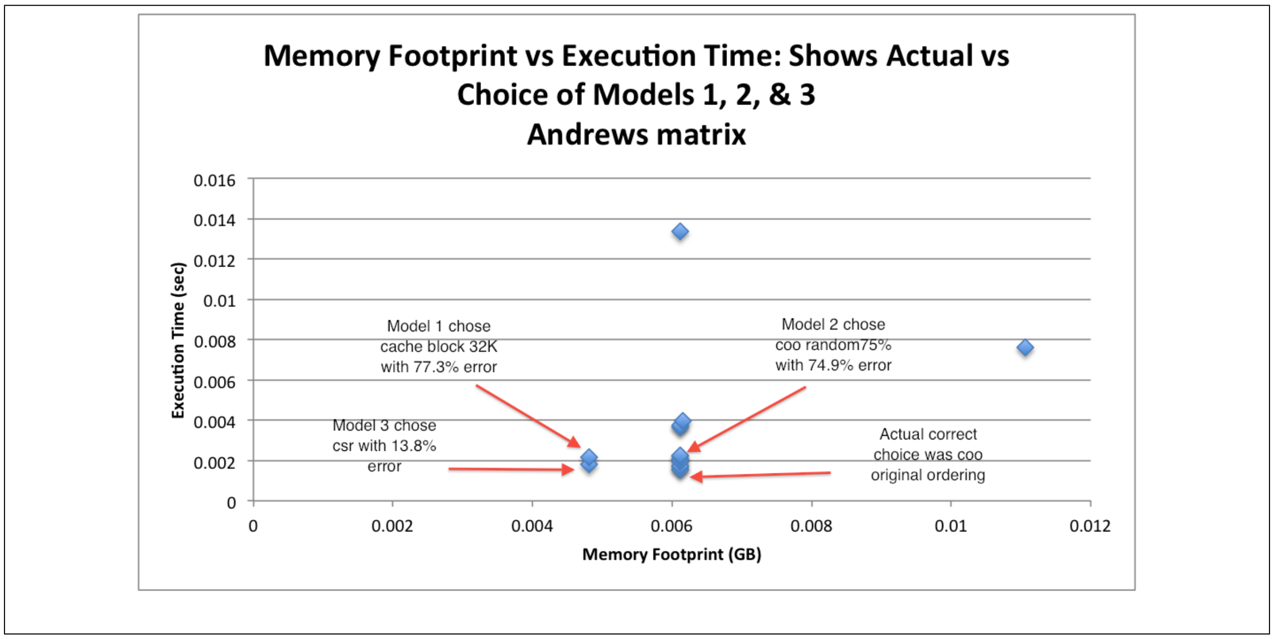


Figure 5.2: Example showing the amount of error that occurs when the model fails to accurately predict the correct data structure

Chapter 6

Conclusions and Future Work

The goal of this thesis was to understand the performance trade-offs in SpMV that are due to memory bandwidth pressure and data locality. We hypothesized that a performance model based on these factors could accurately predict execution time based on our preliminary findings for using Manhattan distance as a data locality performance indicator. However, our experiments show that Manhattan distance is an excellent data locality performance metric per matrix, but not amongst all of the data.

6.1 Understanding Performance Trade-offs in SpMV

Based on our observation that Manhattan distance is not a good performance predictor over all of the data, we hypothesized that these results indicated that other data locality metrics are needed to predict data locality execution time across the data set. Linear regression showed that none of the 5 data locality metrics that we identified could be used to model either the slope or the y-intercepts of the Manhattan distance graph and predict data locality time. Our next approach was to use a multi-parameter linear fit to predict actual execution times and relative performance, which was only able to accurately predict relative performance for an average of 14% of the test set.

During the investigation we discovered that the memory bandwidth component of the model is often able to successfully predict the data structure with the fastest relative performance. These findings indicate that the smallest footprint leads to the best performance for most of the matrices in this test set and that the trade-off between memory versus data locality requirements will most often favor the data structure with the smallest memory bandwidth requirements.

This finding also confirms our hypothesis about the faster performance of OSKI's implementation of SpMV for smaller cache block sizes. The data structure that OSKI uses for the smaller cache block sizes is slightly more compressed than CSR, which indicates that the GCSR's memory bandwidth needs are less than the needs of a CSR storing the same matrix. The model that predicts the performance bounds based on memory access times can be computed using memory footprint information for a matrix, and the size of the line of cache and machine bandwidth. Predicting relative performance based on these parameters can be computed without the need to train or execute the SpMV code. However, a model that does not consider the impact of data locality on execution time, is unable to model instances in which data locality dominates performance.

6.2 Predicting Data Locality Across a Data Set With a Clustering Algorithm

We have shown that memory bandwidth pressure can often be used to predict the data structure with the fastest relative performance. However, predicting actual execution time will require modeling the data locality component as well. One of the primary challenges in developing this model was identifying strong data locality performance indicators that could predict data locality across the data set. Figure 4.5 identified 3 clusters of matrices within the data set, which included matrices with small and large Manhattan distances that behaved as we exepcted, and matrices with a large Manhattan distance, but small data locality execution time. Based on the presence of these 3 clusters, perhaps it is possible to use a clustering algorithm to identify shared data locality characteristics amongst matrices (e.g. nnz/row, nnz/column, sparsity, Manhattan distance, etc.), which could be used to predict execution time.

REFERENCES

[1] P. Colella, "Defining software requirements for scientific computing," 2004, http://view.eecs.berkeley.edu/wiki/DwarfMine.

[2] Y. Saad, *Iterative methods for sparse linear systems*, 2nd ed. SIAM, Philadephia, 2000.

[3] E.-J. Im, "Optimizing the performance of sparse matrix-vector multiplication," Ph.D. dissertation, University of California, Berkeley, May 2000.

[4] J. D. R. Nishtala, R. Vuduc and K. Yelick, "When cache blocking sparse matrix vector mul- tiply works and why," *Applicable Algebra in Engineering, Communication, and Computing*, vol. 18(3), pp. 297–311, March 2007.

[5] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick, "Performance modeling and analysis of cache blocking in sparse matrix vector multiply," University of California, Berkeley, EECS Dept., Tech. Rep. UCB/CSD-04-1335, 2004.

[6] B. Benchmarking and O. Group, "Oski: Optimized sparse kernel interface."

[7] M. M. Strout, A. LaMielle, L. Carter, and J. Ferrante, "Inspector/executor generation for run-time reordering transformations," Journal paper in submission, talk at The Second Annual Concurrent Collections Workshop 2010.

[8] G. Moro, "Overview of sparse matrix formats," *unpublished*, January 2010.

[9] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proc. SciDAC, J. Physics: Conf. Ser.*, vol. 16, 2005, pp. 521–530.

[10] D. P. Samuel Williams, "The roofline model: A pedagogical tool for program analysis and optimization."

[11] *Steps Toward Simplifying Sparse Matrix Data Structures.* Proceedings of the Colorado Celebration of Women in Computing (CCWIC), November 4-5 2010.

[12] A. W. Samuel Williams and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," University of California, Berkeley, Tech. Rep. UCS/EECS-2008-134, 20-8.

[13] P. John D. McCalpin, "http://www.cs.virginia.edu/stream/."

[14] 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: ACM, June 2003.

[15] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *ICS '06: Proceedings of the 20th annual international conference on Supercomputing.* New York, NY, USA: ACM, 2006, pp. 307–316.

[16] K. Y. R. Vuduc, JW Demmel, "Performance optimizations and bounds for sparse matrix-vector multiply," *Supercomputing, ACM/IEEE 2002 Conference*, 2002.

[17] P. P. A. P. Interface, "http://icl.cs.utk.edu/papi/."

[18] e. a. Jack Dongarra, "Accurate cache and tlb characterization using hardware counters."

[19] C. Annis, "http://www.statisticalengineering.com/correlation.htm."

[20] code cogs, "http://www.codecogs.com/code/maths/approximation/regression/linear.php."

[21] NCSU, "http://www.ncsu.edu/labwrite/res/gt/gt-reg-home.html."

[22] D. Stockburger, "http://www.webster.edu/ woolflm/correlation/correlation.html."

[23] U. of Hawaii, "http://math.hawaii.edu/research-links/."

[24] NIST, "http://itl.nist.gov/div898/handbook/prc/section1/prc14.htm."

[25] P. S. N. Gibbs, W. Poole, "An algorithm for reducting the bandwidth and profile of a sparse matrix," *SIAM Journal of Numerical Analysis*, vol. 13, no. 2, p. 16, April 1976.

[26] GNU, "http://www.gnu.org/software/gsl/."

[27] StatSoft, "http://www.statsoft.com/textbook/multipleregression/."

[28] T. Davis, "The university of florida sparse matrix collection."

[29] R. O. Gilbert, *Statistical Methods for Environmental Pollution Monitoring.* John Wiley and Sons, 1987.

Appendix A

Training and Testing Matrices

Table A.1: List of the training set matrices selected from the University of Florida Sparse Matrix Collection. Matrices are sorted by Size(MB).

Matrix	Size(MB)	nnz/row	nnz/col	Sparsity
g2circuit	6.68877	2.9206	2.920601	0.000019
fome21	7.09939	6.86801	2.150654	0.000032
dawson5	8.10496	10.3063	10.306324	0.000200
mesh-deform	13.0284	3.64848	90.900566	0.000388
qa8fk	13.1738	13.056	13.055984	0.000197
benElechi1b	14.8265	3.95195	23.966702	0.000097
landmark	17.5667	16.0	425.751465	0.005917
vanbody	18.1883	25.3226	25.322592	0.000538
language	18.56	3.04746	3.047463	0.000008
lp-nug30	23.9227	30.0	4.132859	0.000079
majorbasis	26.709	10.9401	10.940100	0.000068
dubcova3	28.8645	12.8958	12.895780	0.000088
cant	31.0508	32.5842	32.584217	0.000522
mc2depi	32.0471	3.99415	3.994152	0.000008
si34H36	40.0845	26.9243	26.924269	0.000276
consph	46.4916	36.5626	36.562592	0.000439
bmw7st1	57.0757	26.4633	26.463293	0.000187
shipsec1	60.6863	28.2319	28.231888	0.000200
ga19As19H42	68.8015	33.8708	33.870789	0.000254
t2em	70.0508	4.9812	4.981199	0.000005
g3circuit	70.5434	2.91594	2.915936	0.000002
largeRegFile	75.4422	2.34194	6.169655	0.000003
1d5pt	76.2941	4.99999	4.999994	0.000005
x104	78.3995	47.4056	47.405560	0.000437
cont11l	82.1381	3.6654	2.744476	0.000002
para-10	82.647	34.7372	34.737167	0.000223
siO2	87.2714	36.8208	36.820835	0.000237
tsopf-fs-b300-c3	100.384	77.9344	77.934380	0.000923
rail2586	122.244	3097.97	8.677170	0.003355
ohne2	168.817	61.0089	61.008945	0.000336
bone010	554.298	36.8161	36.816055	0.000037
audikw1	599.636	41.6424	41.642448	0.000044

Table A.2: List of the test set matrices selected from the University of Florida Sparse Matrix Collection. Matrices are sorted by Size(MB).

Matrix	Size(MB)	nnz/row	nnz/col	Sparsity
apache1	4.75341	3.8551	3.855099	0.000048
andrews	6.25766	6.83462	6.834617	0.000114
epb3	7.07482	5.4791	5.479100	0.000065
lung2	7.51616	4.49995	4.499945	0.000041
rajat23	8.49818	5.04679	5.046785	0.000046
mri1	8.99994	9.0	4.00000	0.000061
denormal	9.50374	6.96658	6.966577	0.000078
nemeth22	10.4397	71.9723	71.972336	0.007571
neos2	10.454	5.16782	5.107710	0.000039
2cubes-sphere	13.3417	8.61524	8.615241	0.000085
linmat	15.4296	3.95	3.950000	0.000015
tsc-opf-1047	15.4501	124.388	124.388329	0.015281
pds-90	15.4747	7.10065	2.133011	0.000015
pds-100	16.724	7.01473	2.129909	0.000014
h2O	17.4234	17.0369	17.036882	0.000254
xenon1	18.0224	24.3029	24.302881	0.000500
twotone	18.6798	10.1385	10.138501	0.000084
maceconfwd500	19.4304	6.16653	6.166533	0.000030
nasasrb	20.8445	24.897	24.896975	0.000454
filter3D	21.4661	13.2173	13.217283	0.000124
lhr71c	23.3165	21.7355	21.735491	0.000309
cfd2	24.5002	13.0077	13.007688	0.000105
invextr1-new	27.3725	58.986	58.985958	0.001940
oilpan	28.0074	24.8871	24.887054	0.000337
mixtank-new	30.4415	66.5968	66.596825	0.002223
water-tank	31.0559	33.5081	33.508083	0.000552
neos3	31.3569	4.01208	3.960866	0.000008
parabolicfem	32.0471	3.99415	3.994152	0.000008
t3dh-a	33.8084	27.9855	27.985474	0.000353
asic680ks	35.54	3.41165	3.411652	0.000005
apache2	42.2134	3.86831	3.868311	0.000005
thermomechdK	43.4299	13.9305	13.930519	0.000068

Table A.3: List of the test set matrices selected from the University of Florida Sparse Matrix Collection. Matrices are sorted by Size(MB).

Matrix	Size(MB)	nnz/row	nnz/col	Sparsity
ecology2	45.7462	2.998	2.997998	0.000003
webbase1M	47.3866	3.10552	3.105520	0.000003
laminar-duct3D	58.4878	57.0628	57.062763	0.000849
xenon2	59.0008	24.556	24.556013	0.000156
barrier2-9	59.4719	33.7086	33.708603	0.000292
comat	60.1743	17.8347	17.834686	0.000081
tsopf-fs-b300-c2	67.0024	77.2885	77.288536	0.001360
torso3	67.582	17.0903	17.090254	0.000066
thermal2	74.8319	3.99348	3.993485	0.000003
bmwcra1	82.3419	36.2733	36.273350	0.000244
para-9	82.647	34.7372	34.737167	0.000223
hood	83.839	24.9136	24.913572	0.000113
hamrle3	84.1411	3.80986	3.809862	0.000003
pre2	89.0204	8.85243	8.852430	0.000013
pwtk	90.4264	27.1945	27.194500	0.000125
ccSphere	91.7125	72.1252	72.125183	0.000865
esoc	91.8569	18.4061	159.131348	0.000487
benElechi1	102.206	27.2423	27.242348	0.000111
cont1l	107.3	3.66556	3.659457	0.000002
crankseg2	108.434	111.318	111.318459	0.001744
cage13	114.126	16.7956	16.795624	0.000038
stanfordBerkley	115.713	11.0958	11.095794	0.000016
kkt-power	124.06	3.94009	3.940086	0.000002
pr02R	124.895	50.8173	50.817261	0.000315
torso1	129.952	73.3182	73.318237	0.000631
atmosmodd	134.504	6.93849	6.938490	0.000005
kim2r	172.882	24.7935	24.793468	0.000054
sphere150	175.604	74.2774	74.277390	0.000479
nd24k	219.633	199.914	199.914124	0.002777
tsopf-rs-b2383	246.752	424.217	424.217438	0.011128
ldoor	362.203	24.9289	24.928865	0.000026
spal004	704.47	4524.96	143.514755	0.014066