Benchmarking Sparse Matrix-Vector Multiply

Contents

List of Figures

List of Tables

Acknowledgments

This work was supported in part by the National Science Foundation under CNS-0325873, ACI-0090127, and ACI-9619020, and by the California State MICRO program, and by gifts from Intel Corporation, Hewlett-Packard, and Microsoft. Experiments in this paper were performed on the computer facilities of the Berkeley Benchmarking and Optimization (BeBOP) group at UC Berkeley, on the Millennium cluster at UC Berkeley, and on the Mobius Cluster at The Ohio State University. The information presented here does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

Introduction

Sparse matrix-dense vector multiplication, abbreviated as SpMV, is an important operation in scientific codes. It is important in iterative methods used to solve sparse linear systems arising from discretiziations of partial differential equations, in information retrieval, and many other applications. Because of its wide use, a benchmark for SpMV would provide much useful information to consumers and vendors alike, especially within benchmark suites like the HPCS suite [5] that attempt to measure the overall performance of computing platforms.

However, SpMV is very difficult to benchmark because a number of factors can have an effect on the performance. The data structure used for the sparse matrix, its density of nonzero entries, its dimensions, and even its specific pattern of nonzero entries all have significant effects on SpMV performance, and must all be taken into account if we are to design an accurate benchmark for SpMV.

This thesis describes a prototype benchmark that runs quickly and can be used in benchmarks like the HPCS suite. We also show that the results it gives cannot be obtained by running other preexisting benchmarks.

Chapter 1

SpMV Basics

Sparse matrix-vector multiply is a common operation in scientific codes. It has uses from iterative methods for solving linear systems to information retrieval. However, it is very hard to consistently achieve good performance when running SpMV, much more so than its analogous cousin GEMV (dense matrix-vector multiply). This is because GEMV performance is not affected by the density of nonzero entries or their specific pattern as is SpMV performance. This issue is a significant one that comes up when designing an accurate benchmark for SpMV, as we want to capture this effect. One example of how it comes up, that we will see later on, is that unlike GEMV performance, our ability to improve SpMV performance with performance optimizations depends strongly on the matrix.

In this chapter, we will examine the basics of how SpMV is performed on a computer and how its performance might be improved, which will give us an idea of what we have to take into account when designing a benchmark for SpMV.

1.1 Basic Sparse Matrix Data Structures

In the case of a dense matrix, the most basic data structure used to store it is a simple contiguous array containing all the entries. Two orderings are most common: *row-major*, where the rows are stored contiguously, or *column-major*, where the columns are stored contiguously.

We need such a structure for a dense matrix because usually all of its entries are nonzero. However, in a sparse matrix, most (often over 99%) of the entries are zero. It would be a waste of space and time to store these entries and do arithmetic with them; we would be better off if we could just store and operate on the nonzero entries. In fact, we can do this, so long as we keep track of where exactly in the matrix these entries lie. There are a number of ways this can be done, each of which is known as a sparse matrix data structure. Here is a quick overview of a few "baseline" sparse matrix data structures for the general case of storing a sparse matrix on a uniprocessor machine, where the term "baseline" is used to mean that no performance optimizations (which will be discussed later) have been applied.

1.1.1 Coordinate Format

The simplest format for storing a sparse matrix is the coordinate format, in which the entries are stored in one array, the row index of each entry is stored in a second array, and the column index of each entry is stored in a third array. This requires three arrays of length equal to the number of nonzero entries in the matrix, as shown by this example.

Note that we use zero-based addressing in this thesis.

$$\begin{bmatrix} 1 & 2 & 0 \\ 0 & 3 & 4 \\ 5 & 0 & 0 \end{bmatrix}$$

$$\texttt{values} = [1, 2, 3, 4, 5]$$

$$\texttt{row} = [0, 0, 1, 1, 2]$$

$$\texttt{col} = [0, 1, 1, 2, 0]$$

As we will see with some other formats, it is possible to store the matrix without storing as much indexing information.

1.1.2 Compressed Sparse-Row (CSR) Format

CSR format is one of two "basic" data structures for storing a sparse matrix in that it stores the nonzero entries in order with no wasted space (i.e. the explicit storing of zeros) plus a minimal amount of indexing information, without making any assumptions about the specific pattern of the entries. In a CSR matrix, the matrix's nonzero entries are stored in row-major order with two auxiliary arrays: `row_start` and `col_idx`. `row_start` has one entry for each row of the matrix, with each entry giving the index of the matrix entry that starts that particular row. `col_idx` has one entry corresponding to each matrix entry, and says which column each entry is in. Here is an example:

$$\begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 & 0 \\ 0 & 5 & 0 & 6 & 0 \\ 0 & 0 & 7 & 0 & 8 \end{bmatrix}$$

$$\texttt{values} = [1, 2, 3, 4, 5, 6, 7, 8]$$

$$\texttt{row\_start} = [0, 2, 4, 6, 8]$$

$$\texttt{col\_idx} = [0, 1, 0, 2, 1, 3, 2, 4]$$

1.1.3 Compressed Sparse-Column (CSC) Fromat

CSC format is basically the column-major analog of CSR. Here the nonzero entries are stored in column-major order, and the auxiliary arrays are `col_start` and `row_idx`. `col_start` has the same function as `row_start` except that `col_start` marks the entries that begin each column. Similarly, `row_idx` marks which row each element belongs to instead of which column. The above CSR example in CSC format would look like this:

$$\texttt{values} = [1, 3, 2, 5, 4, 7, 6, 8]$$

$$\texttt{col\_start} = [0, 2, 4, 6, 7, 8]$$

$$\texttt{row\_idx} = [0, 1, 0, 2, 1, 3, 2, 3]$$

1.1.4 Other Formats

The formats listed above are not the only formats available for storing sparse matrices. Many other formats exist. They can be simple modifications to one of the above formats, such as modified sparse-row (MSR), which is just like CSR except that the main diagonal is stored in a separate array that requires no indexing. Or they can be a hybrid of multiple formats. Such is the case with the skyline format, which stores the main diagonal in its own array, the lower triangle in CSR, and the upper triangle in CSC. Then there are other formats that are optimized for vector machines, such as ELLPACK, jagged diagonal, and segmented scan. Information about segmented scan can be found in [2]; the other formats are discussed in [12].

1.1.5 Comparing Formats

In [12], a series of tests were run on a number of scalar uniprocessor platforms comparing the performance of SpMV on matrices stored in the baseline data structures listed above, with the exception of the coordinate format and segmented scan. In most cases, CSR SpMV outperformed the other formats. Based on these results, we will use CSR as the data structure for measuring the performance of unoptimized SpMV.

1.2 Performance Optimizations

Even if we do the best possible job of selecting an unoptimized general-purpose sparse matrix data structure, there can still be more that can be done to improve performance. Out of the tests of various sparse matrix structures in [12], even selecting the best unoptimized data structure for the job resulted in performance that was typically no better than 10% of machine peak. We would like to do better, and fortunately this is often possible. In many cases, we can apply certain performance optimizations that can increase the speed of SpMV dramatically [12]. Though restructuring the matrix into an appropriate format introduces overhead, repeated use of the matrix in applications such as iterative solvers for linear systems makes up for it.

One very basic performance optimization, and the one this thesis will focus on, is to *block* the matrix, i.e. rearrange the matrix entries into blocks that fit in a particular level of a machine's memory hierarchy. This allows for more of the computation to be done in that level, cutting down on the number of time-consuming accesses to lower levels of the memory hierarchy. One form of blocking, *register blocking*, involves storing the matrix in a data structure where in place of the individual nonzero entries are contiguous dense blocks of a size that fits in a processor's register file.

This requires modification of a data structure to index the matrix in terms of block rows and block columns instead of merely rows and columns. If we register block an $m \times n$ matrix with an arbitrary blocksize $r \times c$, we change its data structure to one with m/r *block rows* (each of which represents r rows of the original matrix) and n/c *block columns* (each of which represents c columns of the original matrix). The entry at block row i, block column j is an $r \times c$ dense block whose first entry is the entry at position (ri, cj) in the original matrix.

Here is an example for the CSR format, which when blocked is called BCSR. The blocksize chosen for this example is 2×2. Many selections are possible for the block size; how to make this selection is discussed in more detail in [6], [7], [12], and [15]. In practice, the dense blocks can be stored in either row-major or column-major order; in the example we will use a row-major ordering. The register blocks are color-coded for clarity.

$$\begin{bmatrix} 1 & 0 & 2 & 3 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 7 \\ 0 & 0 & 0 & 0 & 8 & 9 \end{bmatrix}$$

$$\texttt{values} = [1, 0, 0, 4, 2, 3, 5, 0, 6, 7, 8, 9]$$

$$\texttt{row\_start} = [8, 12]$$

$$\texttt{col\_idx} = [0, 1, 2]$$

The main benefit of doing this is that the computation is performed with less traffic between registers and memory, which can lead to substantial speedups. In some cases, this requires introducing explicit zero entries (as seen in the above example) into the data structure. But even when a lot of zero entries are introduced, the savings can still be substantial, as found by [12] and [15]. [12] also presents a system for quickly register blocking a matrix at runtime, which is frequently used by the sparse matrix kernel library OSKI [13] when performing SpMV. Because we can efficiently make frequent use of this optimization in practice on multiple platforms, we will use it when measuring optimized SpMV performance.

Another form of blocking is *cache blocking*, in which the blocks are smaller sparse matrices of sizes that fit inside a particular machine's cache. This can lead to speedups of up to 2.2 over CSR performance for certain matrices, which has been the subject of study in [10]. However, no efficient runtime system yet exists for determining when to cache block a matrix or what block size(s) to use when doing so. This limits its general-purpose use, so we will not use this optimization when measuring optimized SpMV.

Two other possible optimizations for SpMV are mentioned in [12]. One is splitting a matrix into two or more matrices, running SpMV on each of them individually, and then adding the results to get the final answer. This is useful in the case of matrices that have a natural block structure that cannot be captured by just one blocksize. The other is reordering a matrix to provide a natural block structure where none previously existed. OSKI does contain both of these optimizations [14], but they have to be specified by the user because as in the case of cache blocking, there is currently no runtime implementation of them as there is for register blocking [12]. Therefore, we will not make use of these when measuring the performance of optimized SpMV.

Chapter 2

SpMV Performance and Modeling

Understanding what affects SpMV performance is key for designing a benchmark for SpMV. As we will see shortly, the memory access patterns are complicated in SpMV, leading to the previously mentioned result of typical performance being less than 10% of machine peak. Here, we look at what determines the performance of SpMV and how we might go about using this information when designing a benchmark. We would like to be able to approximate real-life SpMV quickly and effectively because measuring performance on matrices taken from real-world applications is prohibitively expensive, as it relies upon collecting many matrices that take up an immense amount of disk space (the matrix suite used in this thesis takes up 2 GB), and the set of matrices used in real-world applications is constantly changing and growing.

2.1 SpMV Performance

Any form of matrix-vector multiplication $b = Ax$ performed on a computer, dense or sparse, involves accesses to three arrays stored in memory: the actual matrix (A), the *source vector* from which elements are read and the matrix entries are multiplied by (x), and the *destination vector* (b) to which the result is written. In the case of GEMV, these are the only arrays we have to access, and the access pattern is highly regular, with each array streamed through using unit stride accesses, assuming row-major or column-major storage.

Now let us look at SpMV with a matrix stored in CSR or BCSR format. There are two more arrays we have to access, `row_start` and `col_idx`. These will be accessed unit-stride, as will the matrix and destination vector. The source vector, however, is no longer accessed in a regular fashion. It is instead accessed indirectly based on the values in `col_idx`, a change that substantially reduces performance. An indirectly accessed vector cannot be avoided, no matter what the data structure used, as some sort of index array or arrays need to be accessed in any sparse matrix data structure to determine where a matrix entry is located. In the case of CSR or BCSR, it is the source vector, as we said before. In the case of CSC, it is the destination vector.

This distinction between accesses to the matrix and accesses to the source or destination vector leads us to propose three categories for SpMV problems, organized based on the size of the matrix and the source vector:

- Small: both the matrix and source vector fit in a machine's cache
- Medium: the matrix falls out of cache but the source vector still fits in cache

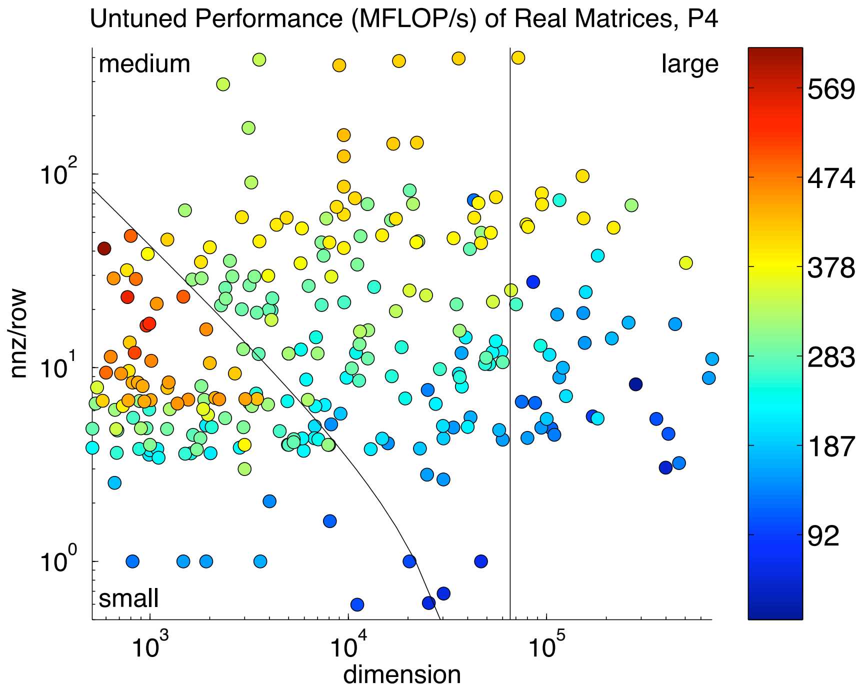
- Large: the source vector no longer fits in cache

Figures 2.1–2.4 show SpMV performance of 275 matrices from real-life applications that fall into each of these three categories. The results were obtained on the platforms decribed in Appendix A. Performance is shown both for untuned (unblocked) and tuned (blocked) cases. The tuning for the blocked case was done by OSKI. As can be seen in the lower plots in each figure, blocking can lead to speedups exceeding 2x on this test set (4x speedups were obtained on some matrices in the test set used in [12] and [15]).

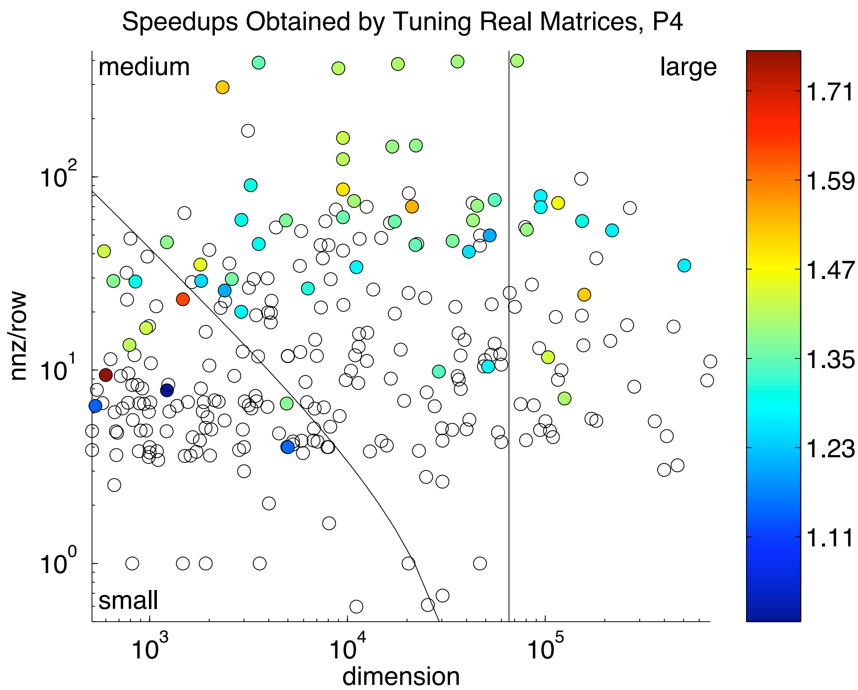
These matrices, which we will use as our sparse matrix test suite in this thesis, were taken from the online collection [3], and are described in more detail in Appendix B.

There are a few general trends to take note of in these plots. There is definitely a decline in performance going from small to medium to large problems. The performance peak is for the densest small problems. In fact, there is a very noticeable trend for all problem sizes for the performance to increase with nnz/row. While this is the trend that jumps out the most, we will focus on capturing the trend of performance decreasing as the dimension increases because larger, sparser matrices are more common in applications, as seen in Appendix B. Going this way also allows us to observe performance for all three problem sizes.

For an SpMV benchmark to be accurate, any operation it uses to approximate real-life SpMV performance should capture both the effects of dependence on the memory hierarchy as outlined above and the effects of performance tuning. We will now look at some possible approaches based on prior work and see where they fall short based on these criteria, and conclude by presenting a new approach that forms the basis for the benchmark

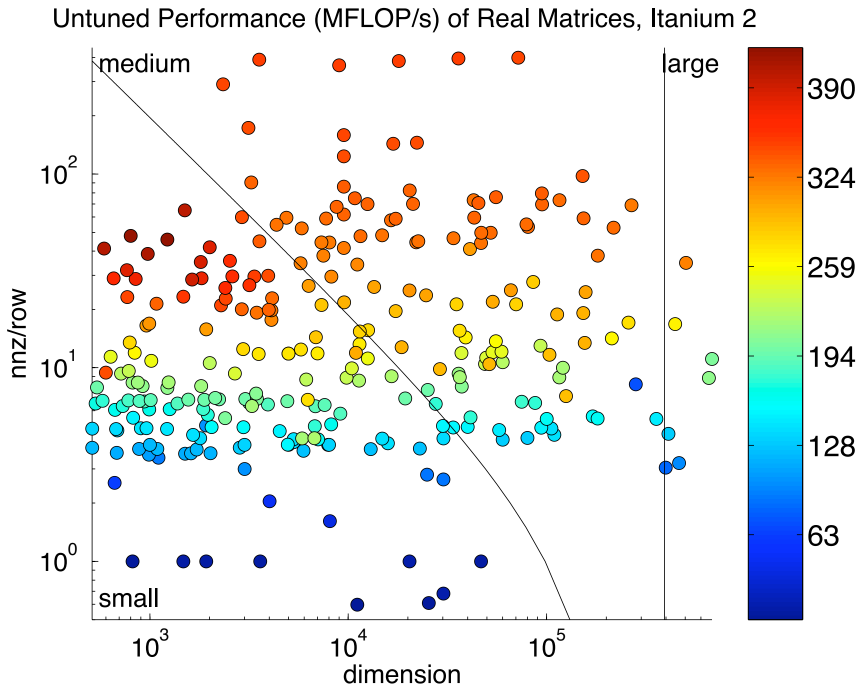


(a) Untuned

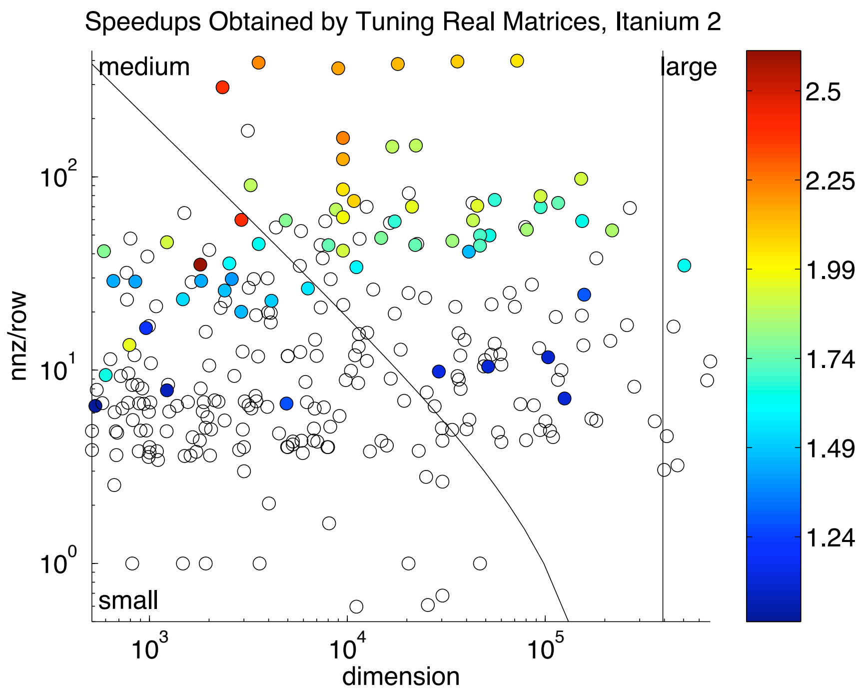


(b) Tuned

Figure 2.1: Small, Medium, Large behavior on the Pentium 4.

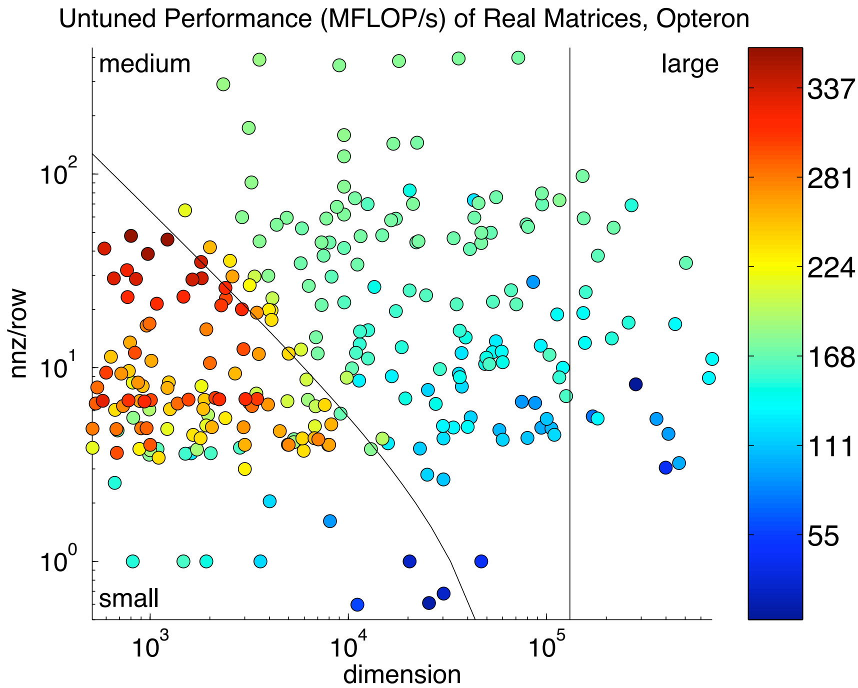


(a) Untuned

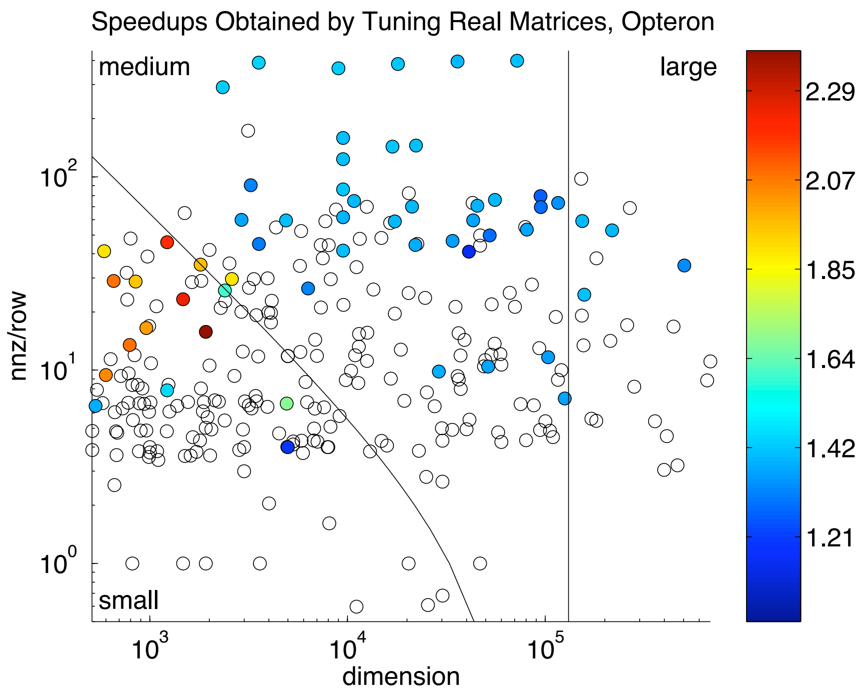


(b) Tuned

Figure 2.2: Small, Medium, Large behavior on the Itanium 2.

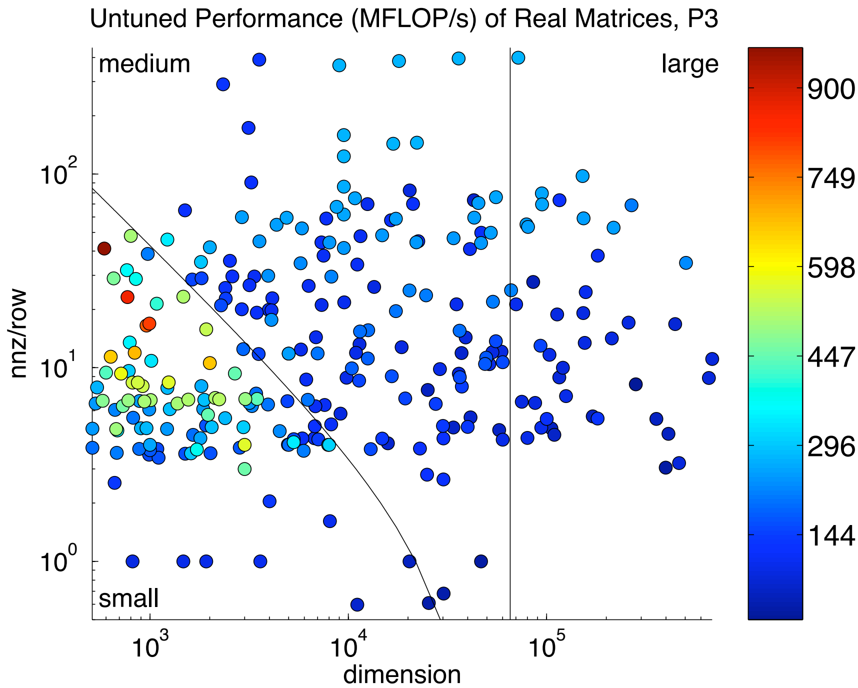


(a) Untuned

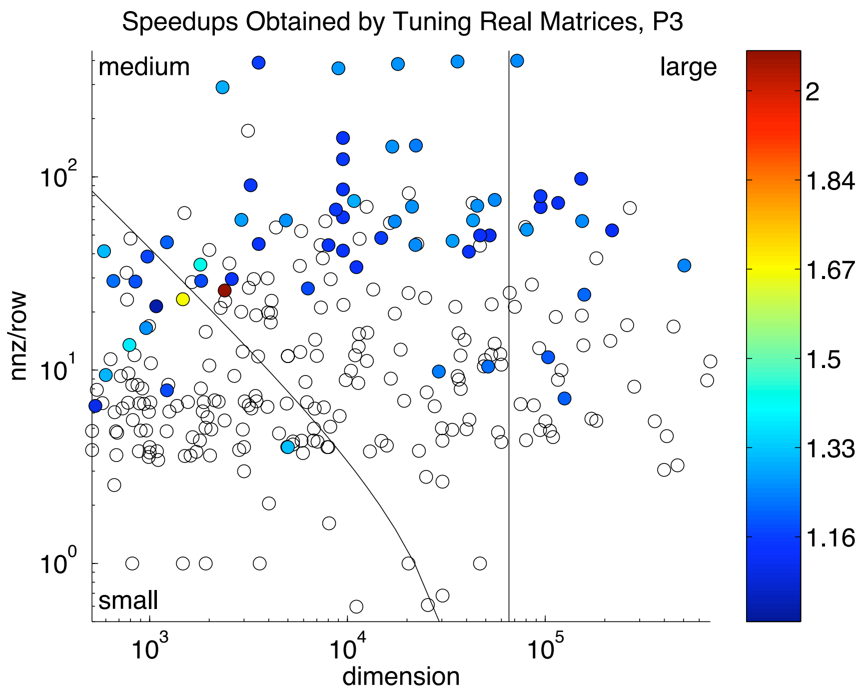


(b) Tuned

Figure 2.3: Small, Medium, Large behavior on the Opteron.



(a) Untuned



(b) Tuned

Figure 2.4: Small, Medium, Large behavior on the Pentium 3.

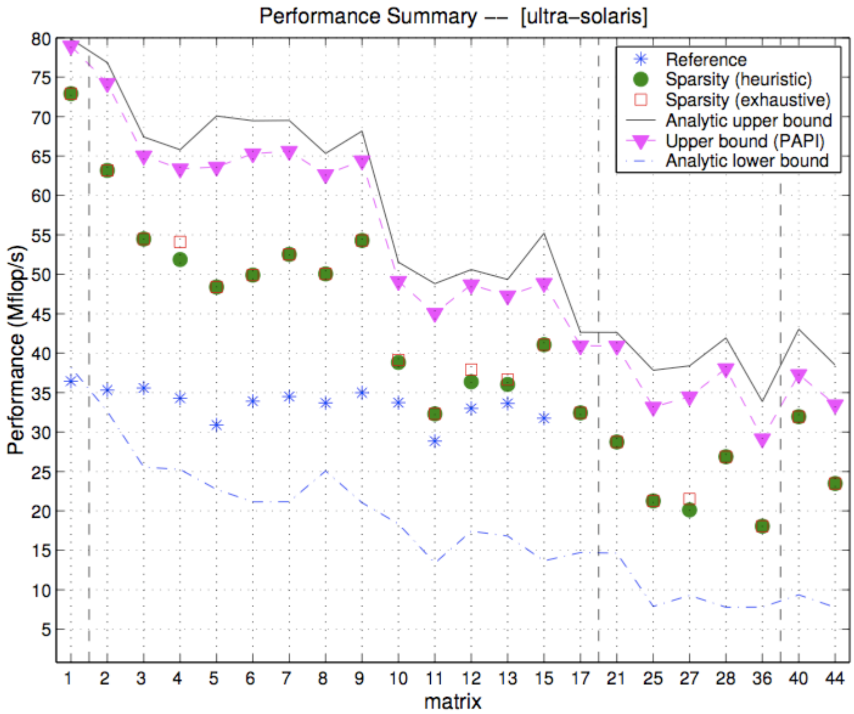


Figure 2.5: Performance bounds vs. SpMV performance on a Sun Ultra 2i. Reproduced from [15] with permission.

that we present in the next chapter.

2.2 Approximation By Performance Bounds

[12] and [15] developed upper and lower bounds on SpMV performance based on the best and worst case scenarios regarding cache misses. As shown there, the bounds very reliably predict best and worst-case SpMV performance on multiple platforms, but do not give any kind of indicator of "expected" performance, as seen in figure 2.2 for example. Notice how actual SpMV performance (untuned in the case of blue stars, and tuned in the case of green circles) is bounded quite loosely by the upper and lower bounds.

[12] also tried using *machine balance* to benchmark SpMV. The machine balance

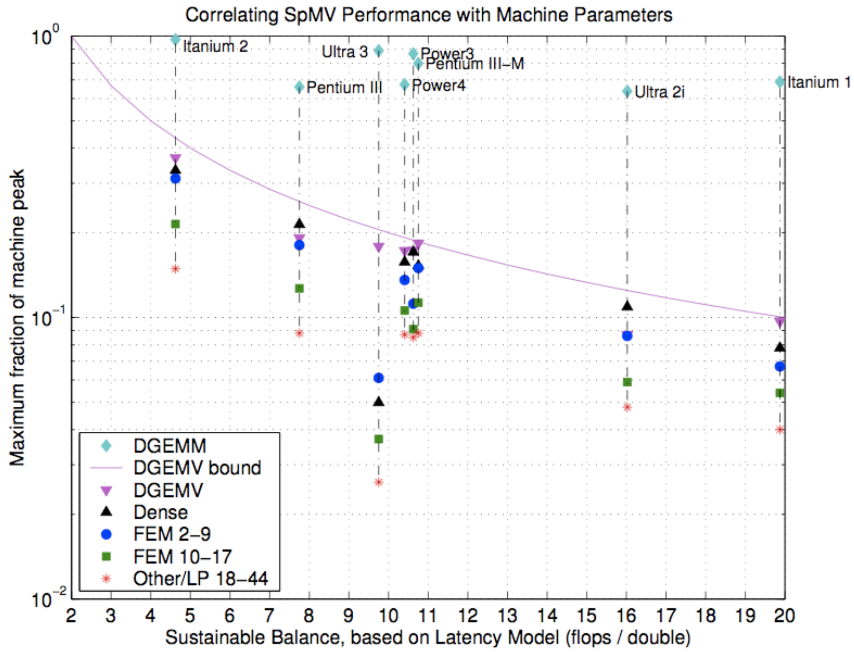


Figure 2.6: Machine balance vs. SpMV performance on 8 platforms. Reproduced from [12] with permission.

of a particular architecture is the ratio of its peak floating point operation rate to its maximum sustainable main memory bandwidth. As this approaches 2, performance for SpMV approaches optimal, which is two floating point operations (multiply and add) per memory reference. This correctly predicted SpMV performance on a number of machines, but missed significantly on one, as shown in figure 2.2. A Sun Ultra 3 was predicted to be the third-best platform for SpMV out of the eight in the graph, but ended up being the worst.

2.3 Approximation By Other Operations

The main operation in an SpMV problem $Ax = b$ with an $m \times n$ matrix A is performing the update

$$b_i := b_i + a_{ij}x_j$$

for $j = 1, \ldots, m$ for each element b_i in the destination vector. Based on this, it might seem like a good idea to try and measure SpMV performance by measuring the performance of this operation. The STREAM TRIAD benchmark [8] measures the sustainable main memory bandwidth for this operation in MB/s, which can be translated into a MFLOP rate. For the platforms we tested, the results were

Platform	STREAM Triad MFLOP/s
Pentium 4	170
Itanium 2	238
Opteron	149
Pentium 3	52

In figures 2.7–2.10, we plot these numbers against the unblocked performance of SpMV for real matrices. STREAM Triad shows itself to be somewhat predictive of average performance on the Itanium 2, but underpredictive of performance on the the Pentium 4, the Opteron, and the Pentium 3 except for large problems. This is expected, as STREAM Triad measures sustainable main memory bandwidth. While a portion of a problem is still in cache, memory bandwidth between the cache and the processor is also a factor and STREAM Triad does not measure that.

This plus there being no way to block STREAM Triad for comparison with register blocked SpMV disqualifies STREAM Triad as a benchmark for SpMV.

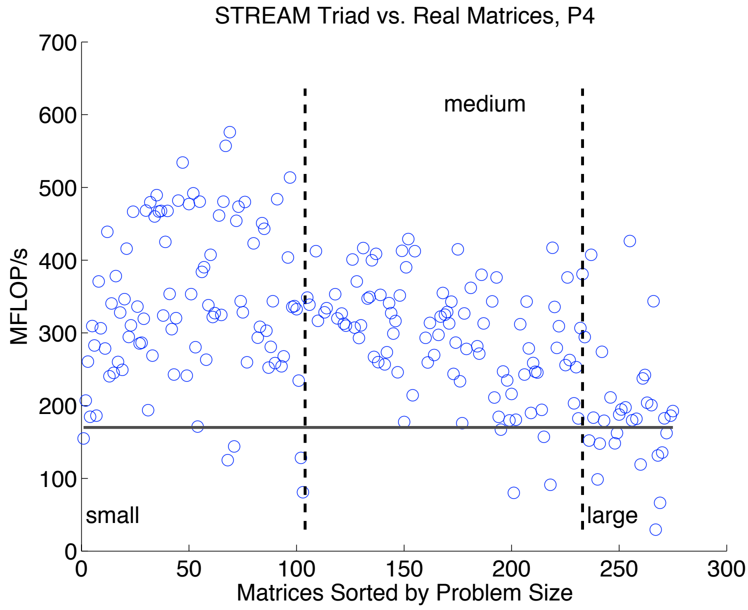


Figure 2.7: Performance of STREAM Triad vs. real matrices on the Pentium 4.

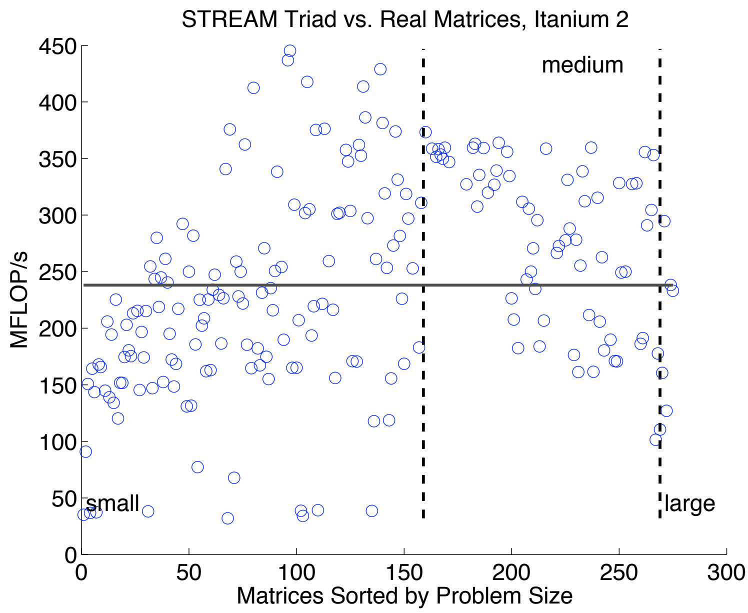


Figure 2.8: Performance of STREAM Triad vs. real matrices on the Itanium 2.

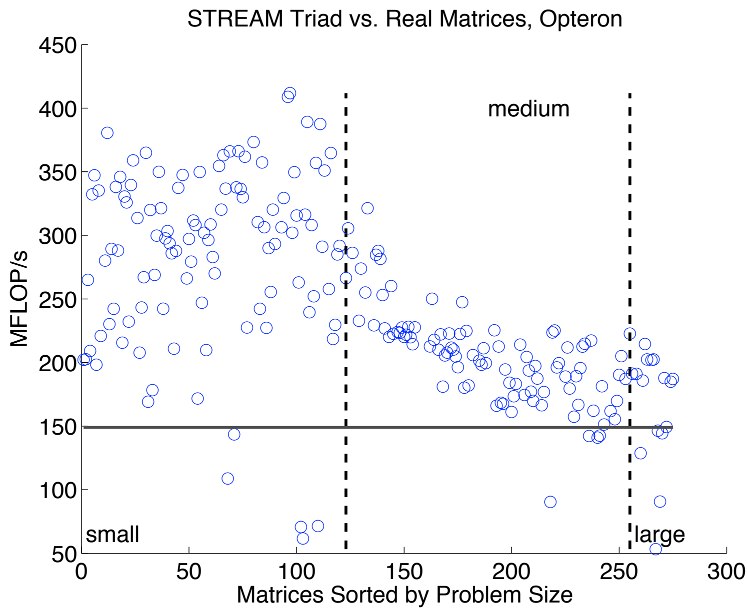


Figure 2.9: Performance of STREAM Triad vs. real matrices on the Opteron.

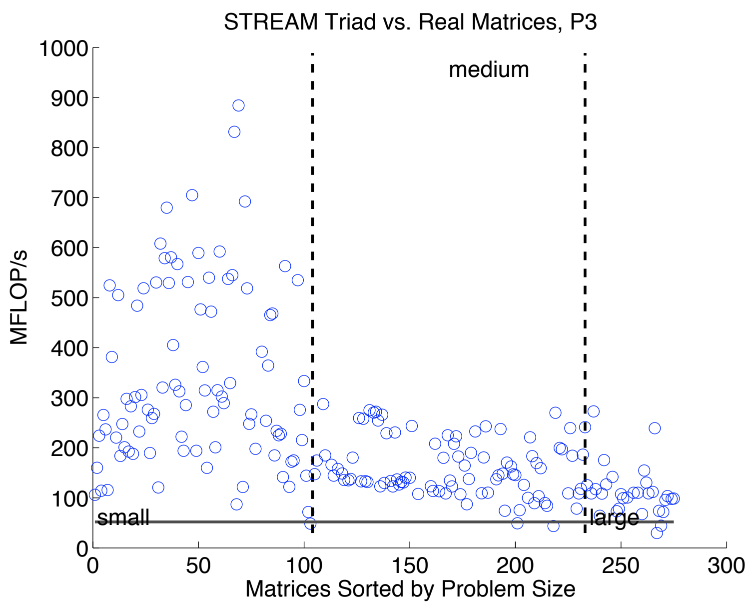


Figure 2.10: Performance of STREAM Triad vs. real matrices on the Pentium 3.

2.4 Preexisting Benchmarks That Perform SpMV

There are three preexisting benchmarks that directly measure the performance of SpMV. However, all of them have shortcomings. The SciMark benchmark suite [9] contains an SpMV benchmark that measures SpMV performance for two CSR matrices, one small and one large, each with a particular structure. The small matrix is 1000×1000 with 5 nonzero entries per row, and the large matrix is 100000×100000 with 10 nonzero entries per row. Data we saw in the first section tells us that performance varies enough with problem size that it would be preferable to measure multiple problem sizes when benchmarking SpMV. More important is that the SpMV performed by SciMark uses no blocking.

SparseBench [4] is a benchmark that measures the performance of iterative methods for sparse linear systems derived from partial differential equations. SpMV is a heavily used operation in these methods, and SparseBench measures how fast this component is performed. However, it suffers from the same crucial flaw as SciMark in that it does not measure optimized SpMV performance.

NAS-CG is one of a suite of popular benchmarks [1] for measuring the performance of supercomputers. It measures the performance of the conjugate gradient operation, which is rich in SpMV. However, it also contains dense vector updates and outer products, and requires the matrix to be symmetric and positive definite, which is not representative of many real-life matrices. And while it does allow for performance optimizations, they must be user-supplied. The standard reference implementation does not use a matrix susceptible to blocking.

2.5 Using Synthetic Matrices to Mimic Real-Life Ones

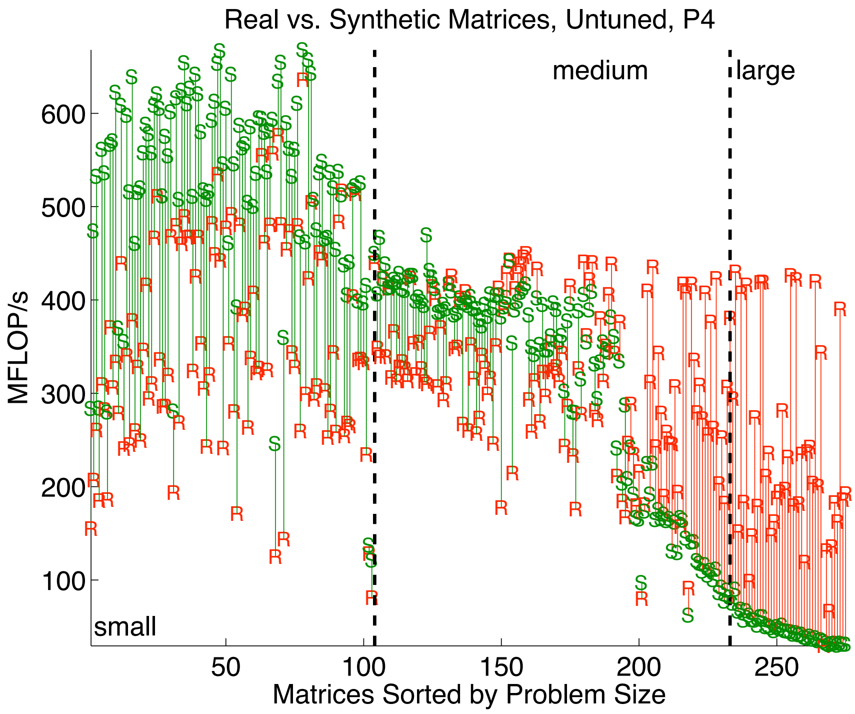
We saw in the previous section that the SpMV in preexisting benchmarks is not sufficient to model SpMV for the multitude of matrices that exist in real-world applications. There are a number of issues that need to be resolved, all centered around this question: what properties of the matrices do we need to preserve in the synthetic matrices we are using to model them? The fewer we need, the better, because that means we will have to generate fewer matrices to model SpMV, but it turns out that we will need to capture a number of properties to effectively model SpMV with synthetically generated matrices.

Beyond the problem size discussed in section 2.1, the first property we have to capture is the register blocksize selected when doing performance tuning. This is vital as it is our way of measuring its effect. The results of matching this along with problem size (dimension and density) are shown in figures 2.11–2.13. In the upper plot in each of these figures, each real matrix is denoted by a red R and is connected to its corresponding synthetic matrix, which is denoted by a green S. The color of the line, red or green, denotes which matrix ran faster. In the lower plot in each of these figures, the results are color-coded by the largest register block dimension of the real matrices being modeled. The synthetic matrices are created to have the same dimensions and nnz/row as the real matrices we are trying to match them with. The entries or dense blocks in the case of blocked matrices are placed randomly in each row, with no particular pattern. We match every matrix in the suite with an unblocked counterpart (figures 2.11(a), 2.12(a), 2.13(a), and 2.14(a)), and the ones that OSKI can tune are also matched with a blocked counterpart (figures 2.11(b), 2.12(b), 2.13(b), and 2.14(b)). The horizontal axis in each plot is sorted by increasing problem size, small to medium to large. For small to medium, this means sorting by the total space taken by the matrix and vectors, and for medium to large and beyond, the space taken by the vectors only. The order of the matrices for each platform are given in appendices D–G.

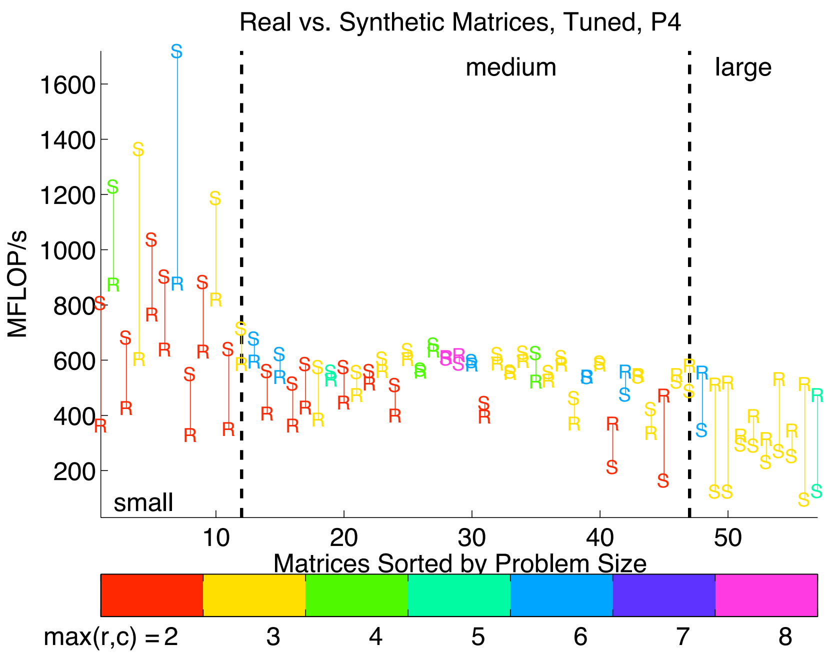
Another perspective is given in figures 2.15–2.18. Here, color is used to show how well the synthetic matrices predict the performance of the real matrices, and the matrices themselves are organized not according to increasing problem size, but by both dimension and nnz/row.

The first thing that stands out from these graphs is that the synthetic matrices substantially underpredict the performance of the real matrices for the large matrices, especially in the case where we use no blocking. To fix this, we will also try to match the nonzero pattern of the real-life matrices. We will do this in terms of the distribution of entries in bands that lie parallel to the diagonal, as an examination of the matrices in [3] reveals that many of them have their nonzero entries in bands that are located relatively close to the diagonal. By this we mean that, if we divide up the matrices into bands that are $10(i-1)$ to $10i$ percent away from the diagonal, where $1 \leq i \leq 10$, then the entries are mostly in the bands where i is small. Figure 2.5 shows this.

Appendix C shows which percentage of the matrix entries lie in these bands for each of the matrices in our test suite. To generate synthetic matrices that match the dimensions, density, and nonzero distribution of real matrices, we use algorithm 2.5.1. Note that all divisions in the algorithm are to be rounded to the nearest integer.

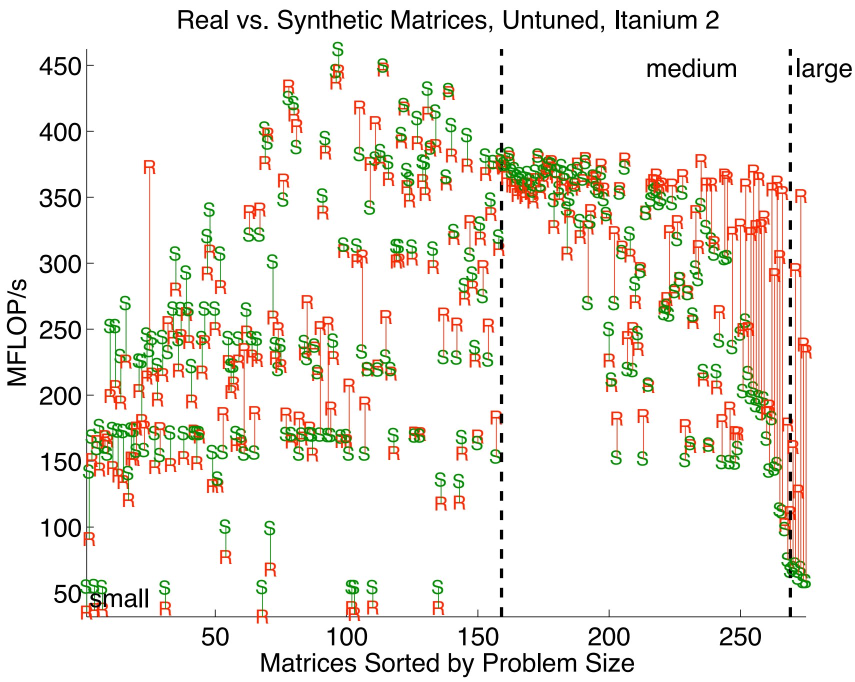


(a) Untuned

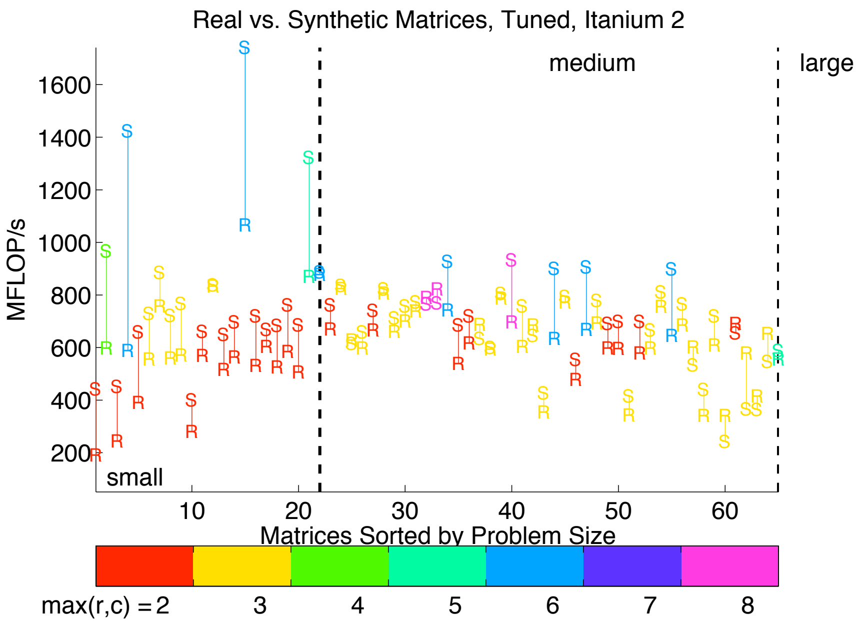


(b) Tuned

Figure 2.11: Real vs. Synthetic matrices on the Pentium 4.

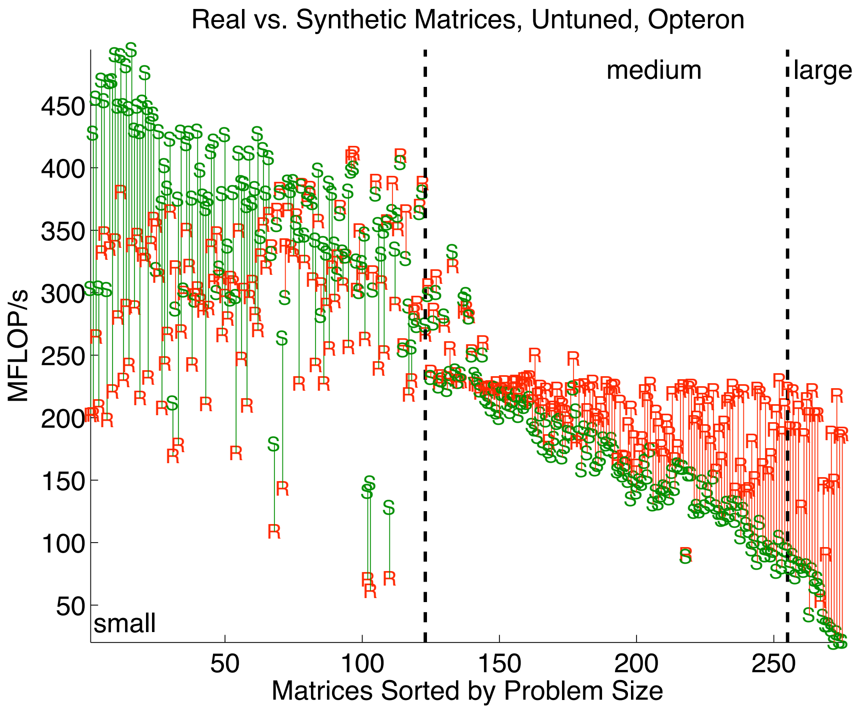


(a) Untuned

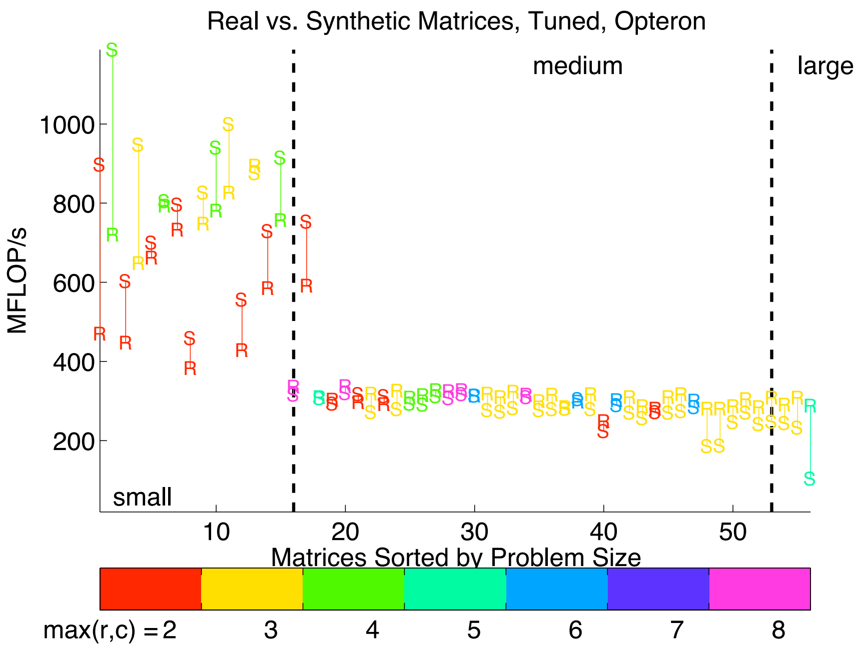


(b) Tuned

Figure 2.12: Real vs. Synthetic matrices on the Itanium 2.

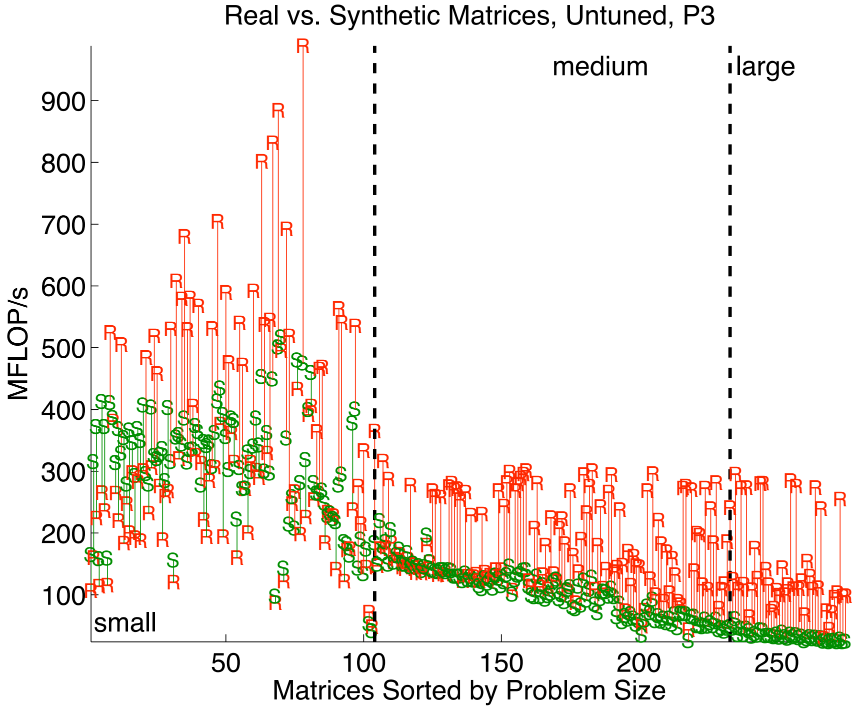


(a) Untuned

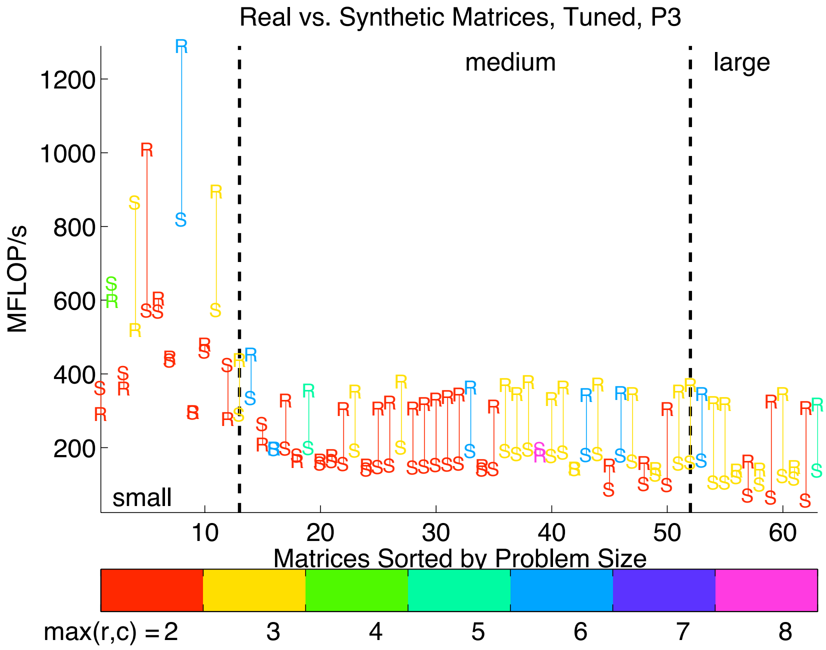


(b) Tuned

Figure 2.13: Real vs. Synthetic matrices on the Opteron.

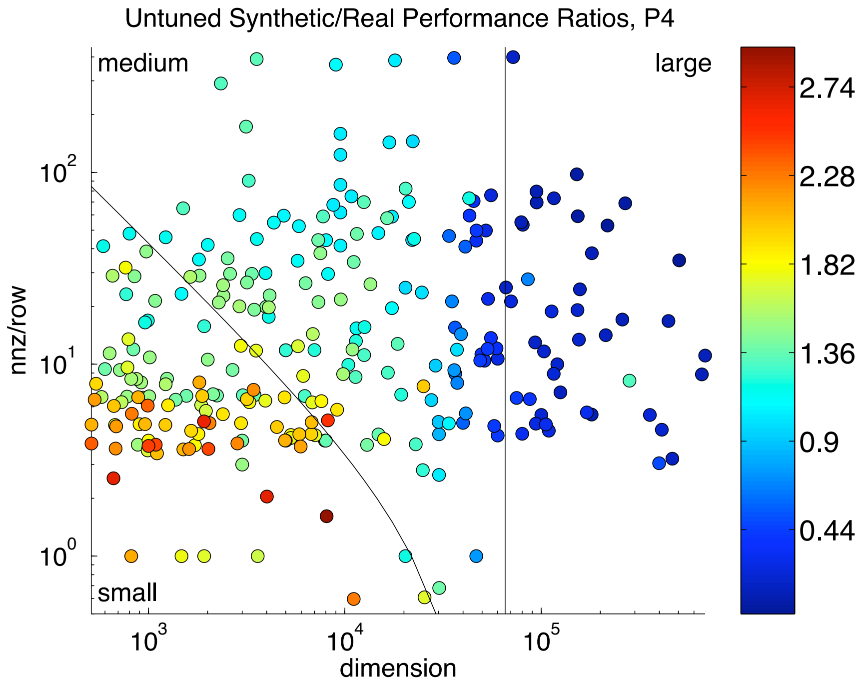


(a) Untuned

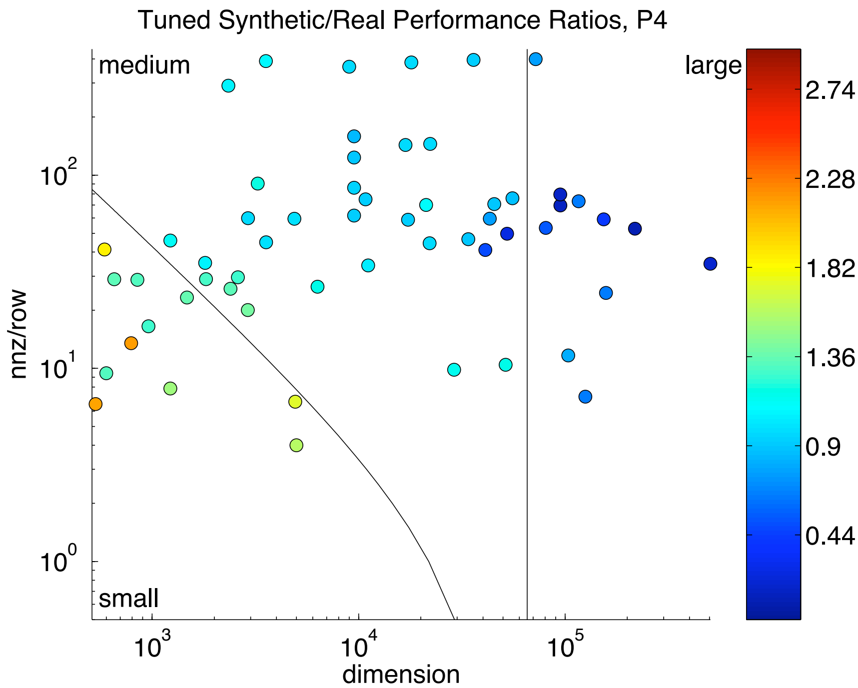


(b) Tuned

Figure 2.14: Real vs. Synthetic matrices on the Pentium 3.

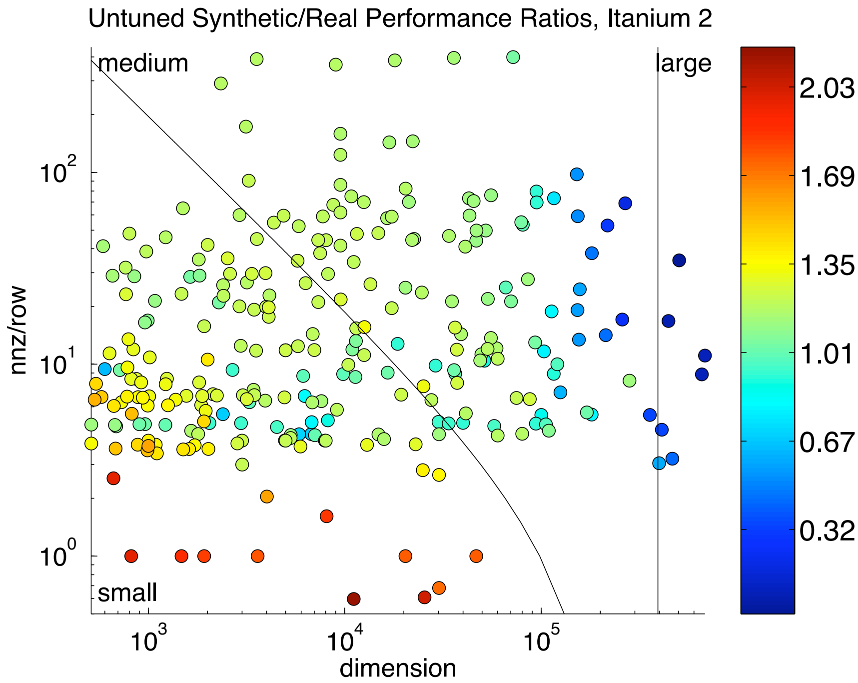


(a) Untuned

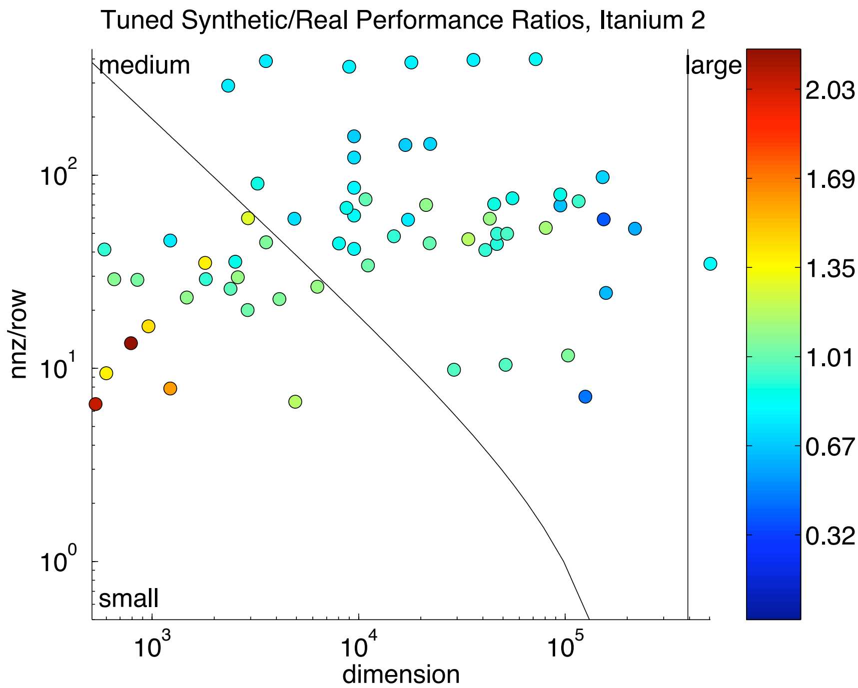


(b) Tuned

Figure 2.15: Real vs. Synthetic matrices on the Pentium 4.

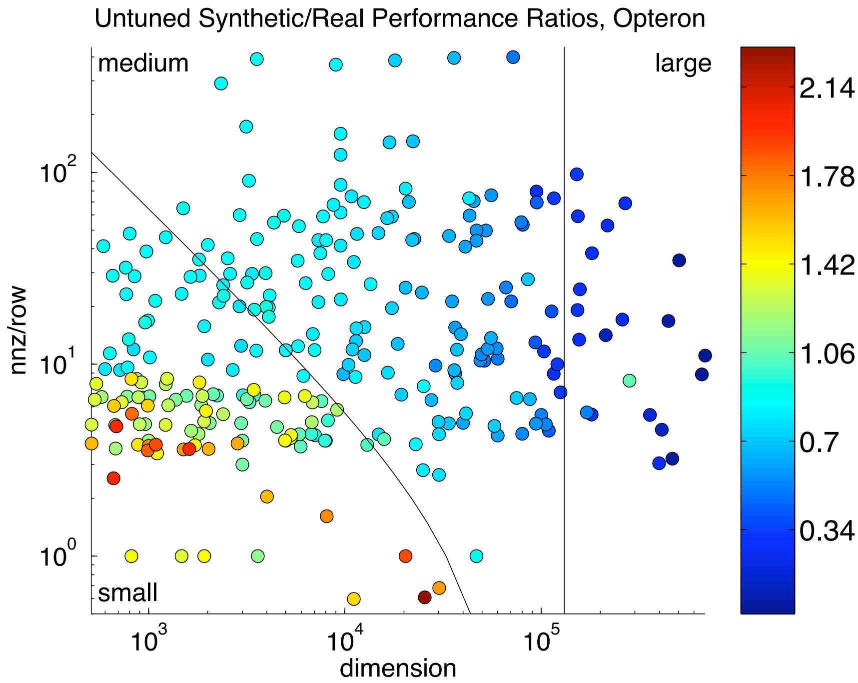


(a) Untuned

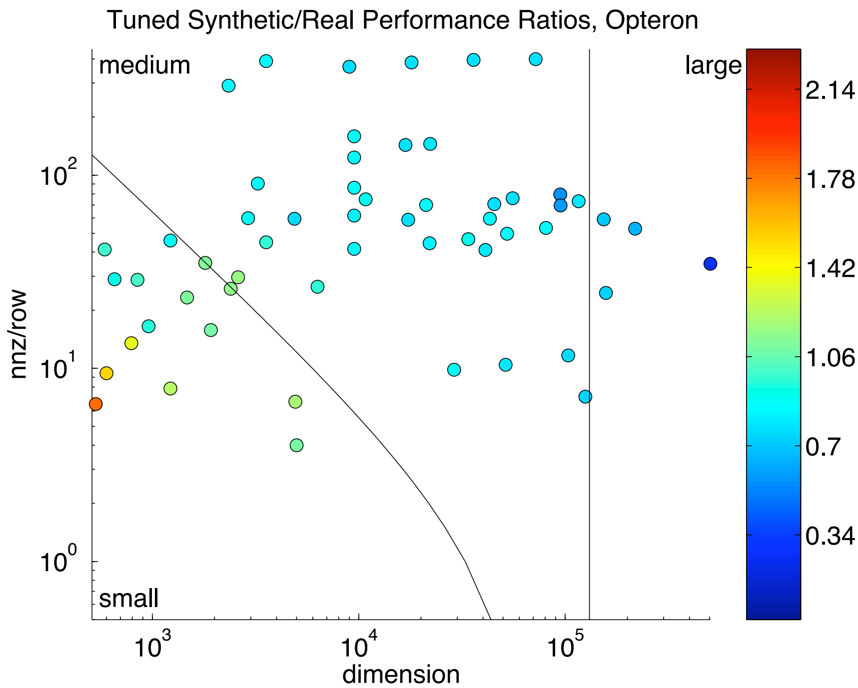


(b) Tuned

Figure 2.16: Real vs. Synthetic matrices on the Itanium 2.

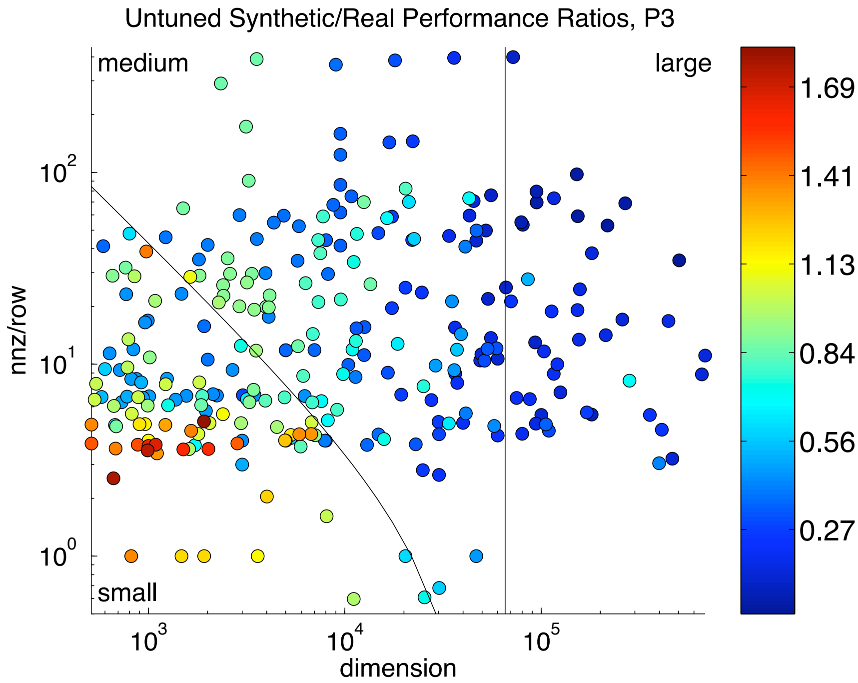


(a) Untuned

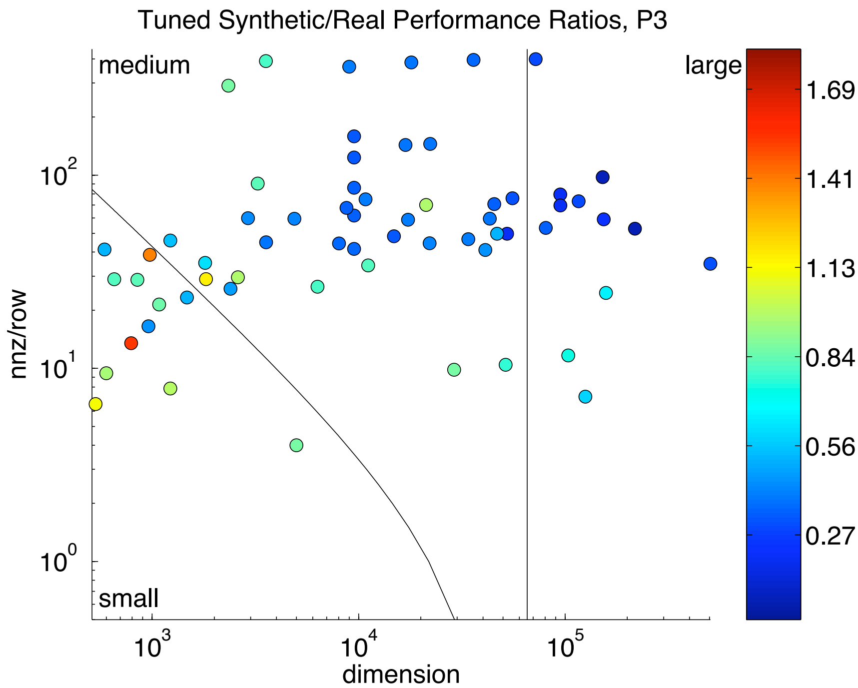


(b) Tuned

Figure 2.17: Real vs. Synthetic matrices on the Opteron.



(a) Untuned



(b) Tuned

Figure 2.18: Real vs. Synthetic matrices on the Pentium 3.

Algorithm 2.5.1: GENERATEMATRIX($m, n, r, c, nnz\_row, stats$)

$num\_block\_rows \leftarrow m/r$

$num\_block\_cols \leftarrow n/c$

$nnzb \leftarrow num\_block\_rows \times num\_block\_cols$

$nnzb\_row \leftarrow nnzb/num\_block\_rows$

for $i \leftarrow 1$ **to** 10

do $\begin{cases} nnzb\_per\_decile(i) \leftarrow \text{nnzb to add to decile } i \\ decile\_size(i) \leftarrow \text{size in nnzb of decile } i \end{cases}$

for $i \leftarrow 1$ **to** num_block_rows

for $j \leftarrow 1$ **to** 10

do

- $decile\_size\_this\_row \leftarrow$ size in nnzb of decile j in row i
- $nnzb\_added \leftarrow \frac{nnzb\_per\_decile(j) \times decile\_size\_this\_row}{decile\_size(j)}$
- add $nnzb\_added$ nonzero blocks at random locations in row i, decile j
- **if** $nnzb\_added > nnzb\_row$
  - **then repeat**
    - take 1 entry away from each decile proportionally
    - decrement $nnzb\_added$
- **until** $nnzb\_added = nnzb\_row$
  - **else if** $nnzb\_added < nnzb\_row$
  - **then repeat**
    - add 1 entry to each decile proportionally
    - increment $nnzb\_added$
- **until** $nnzb\_added = nnzb\_row$

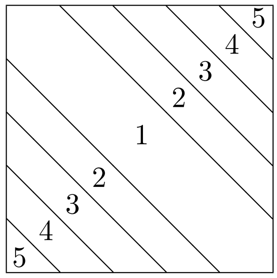
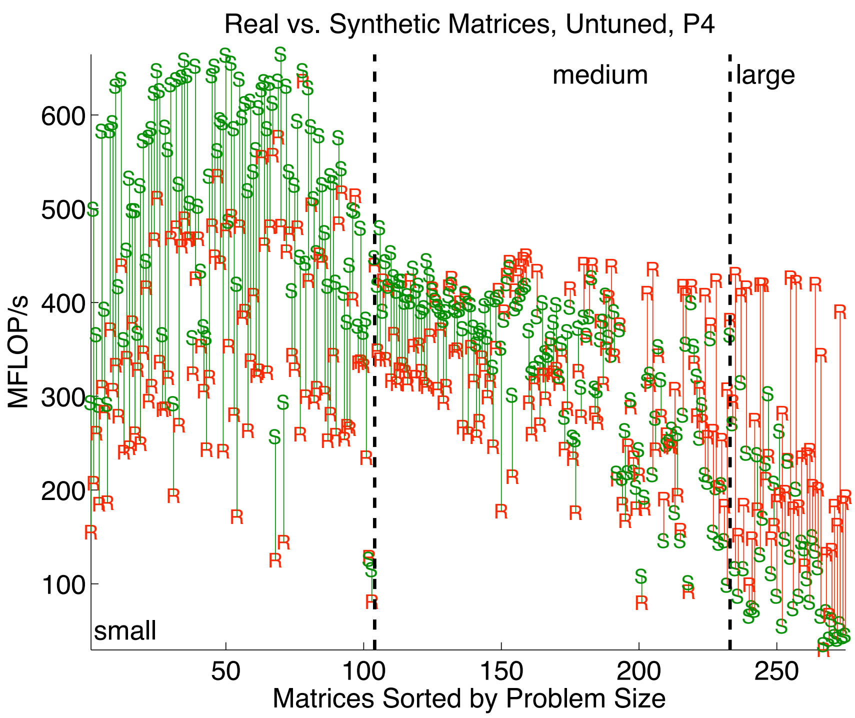


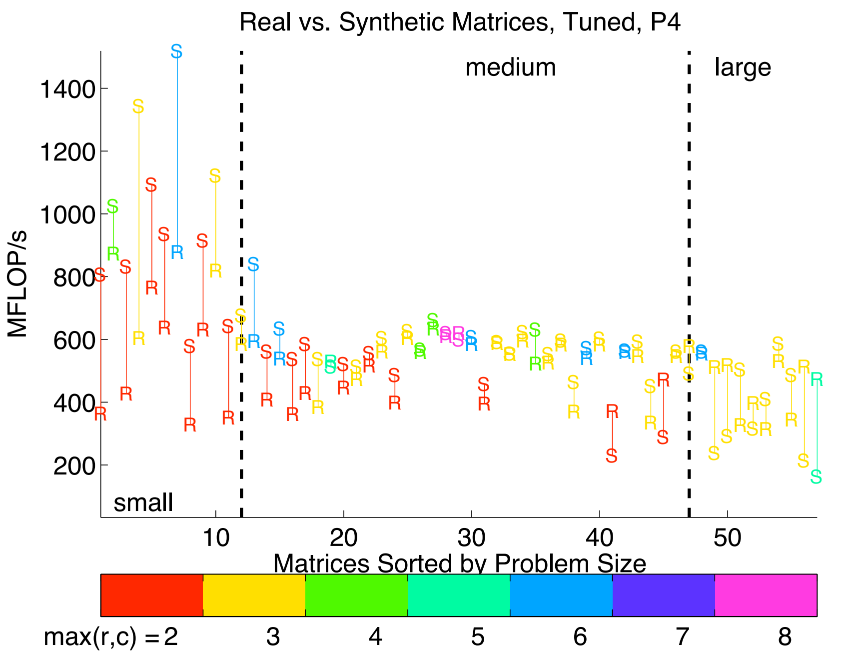
Figure 2.19: Matrix divided up into bands. For simplicity of illustration, this matrix is only divided up into 5 bands instead of 10.

Using algorithm 2.5.1 to generate our matrices, we get the results in Figures 2.20–2.23, with an alternate perspective organized by dimension and nnz/row provided by figures 2.24–2.27. Here, we see that the synthetic matrices still underpredict the performance of the real matrices for large problems, but not to the extent that they do in figures 2.11–2.14. We also see that the synthetic matrices do a pretty good job of predicting the performance of real matrices for medium problems while noticeably overpredicting the performance of real matrices for small problems on two platforms (the Pentium 4 and the Opteron) and noticeably underpredicting performance on the Pentium 3. The misprediction of small problems is something we will see in the next chapter, and discuss further in Chapter 4.

One last thing to note is that while a number of matrices in our test suite are symmetric, the matrix generator we use does not create symmetric matrices. To generate data from which we could accurately gauge how well the synthetic matrices performed, we ran SpMV on the symmetric matrices from our test suite with symmetry disabled. We will return to this issue in Chapter 4.

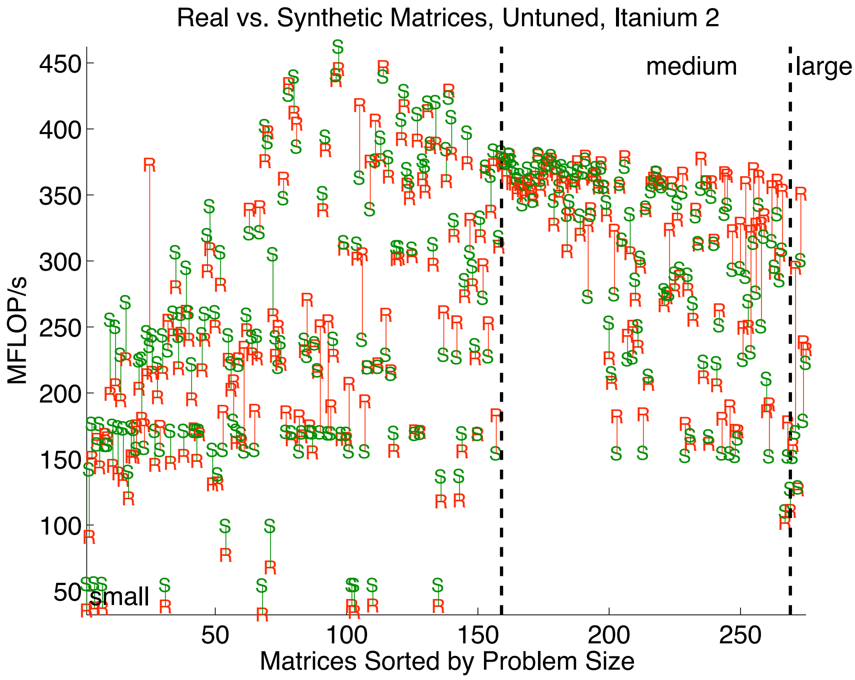


(a) Untuned

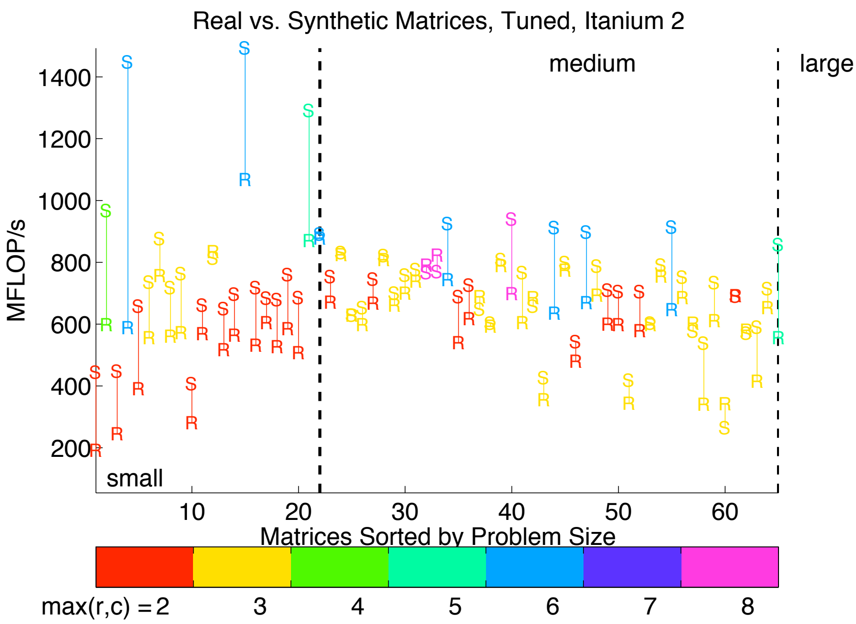


(b) Tuned

Figure 2.20: Real vs. Synthetic matrices on the Pentium 4 where the nonzero distributions of each matrix are also matched.

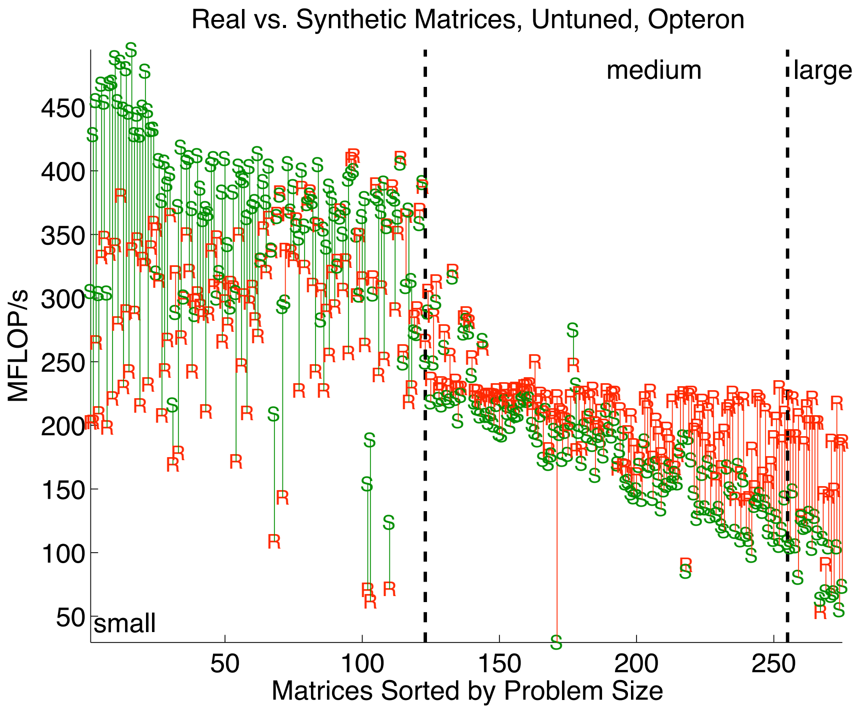


(a) Untuned

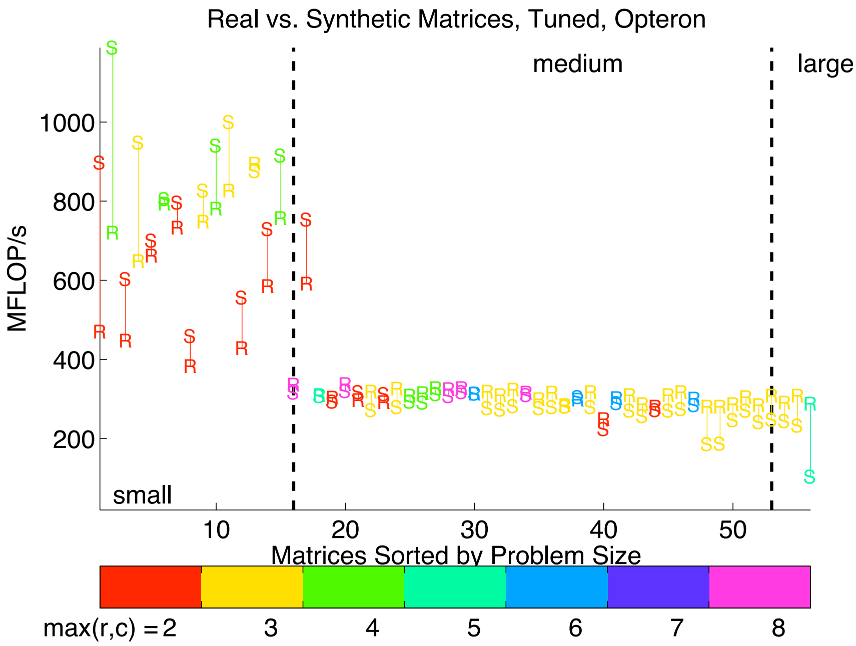


(b) Tuned

Figure 2.21: Real vs. Synthetic matrices on the Itanium 2 where the nonzero distributions of each matrix are also matched.

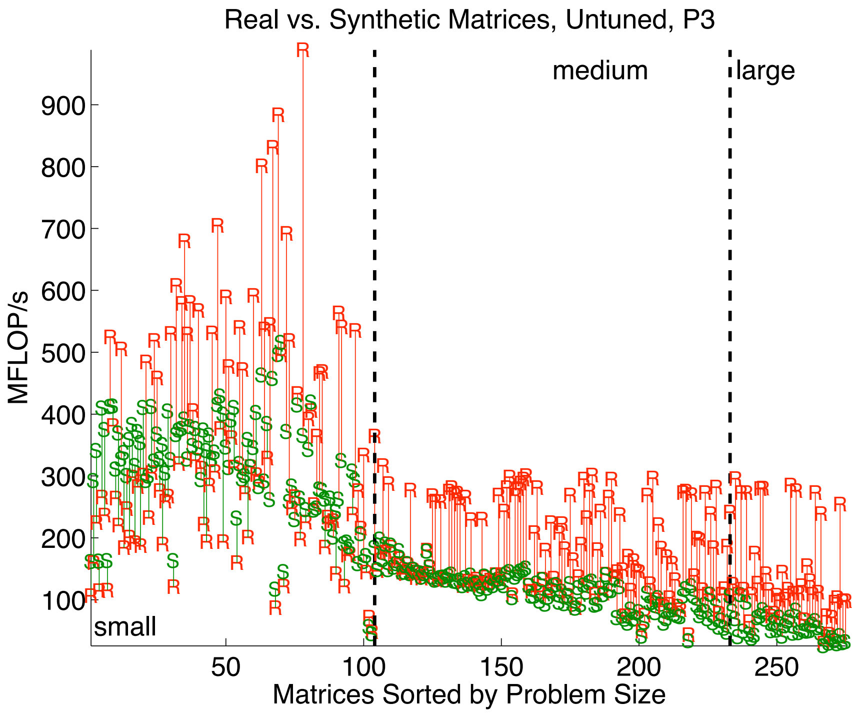


(a) Untuned

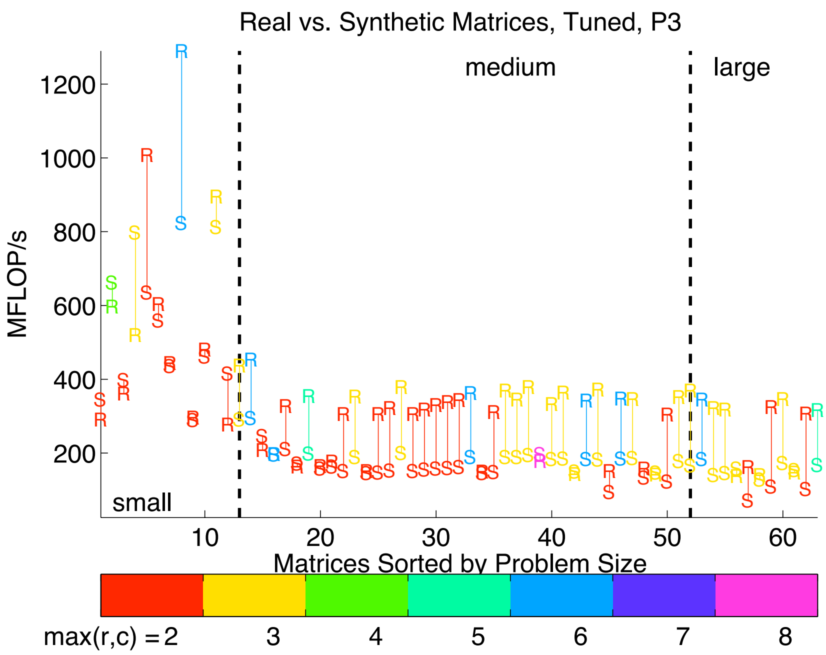


(b) Tuned

Figure 2.22: Real vs. Synthetic matrices on the Opteron where the nonzero distributions of each matrix are also matched.

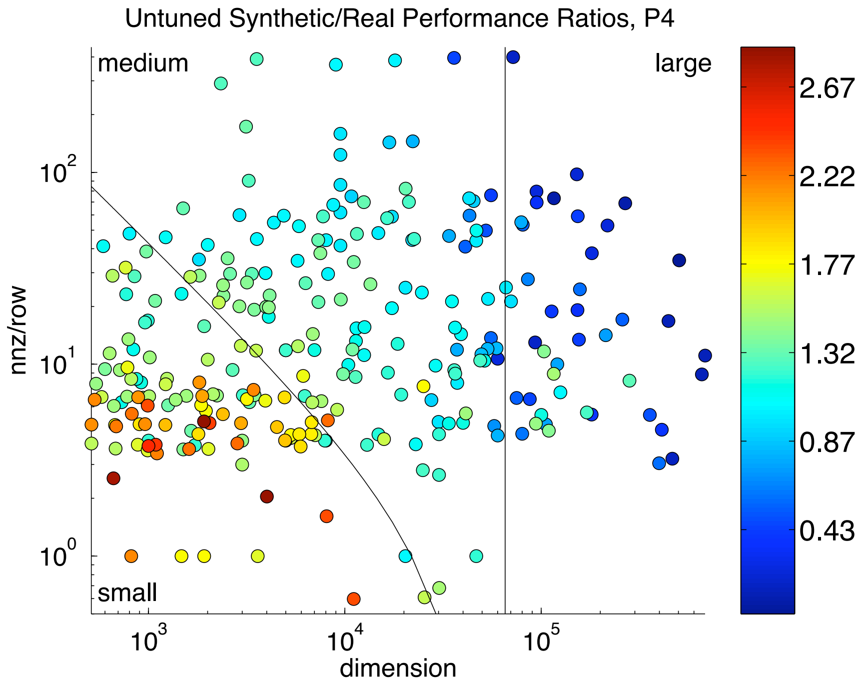


(a) Untuned

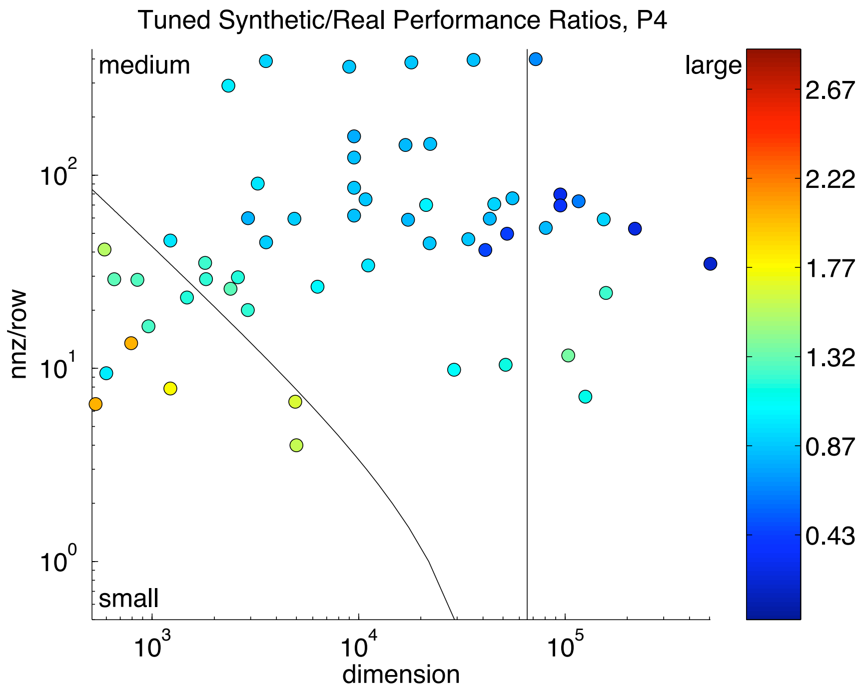


(b) Tuned

Figure 2.23: Real vs. Synthetic matrices on the Pentium 3 where the nonzero distributions of each matrix are also matched.

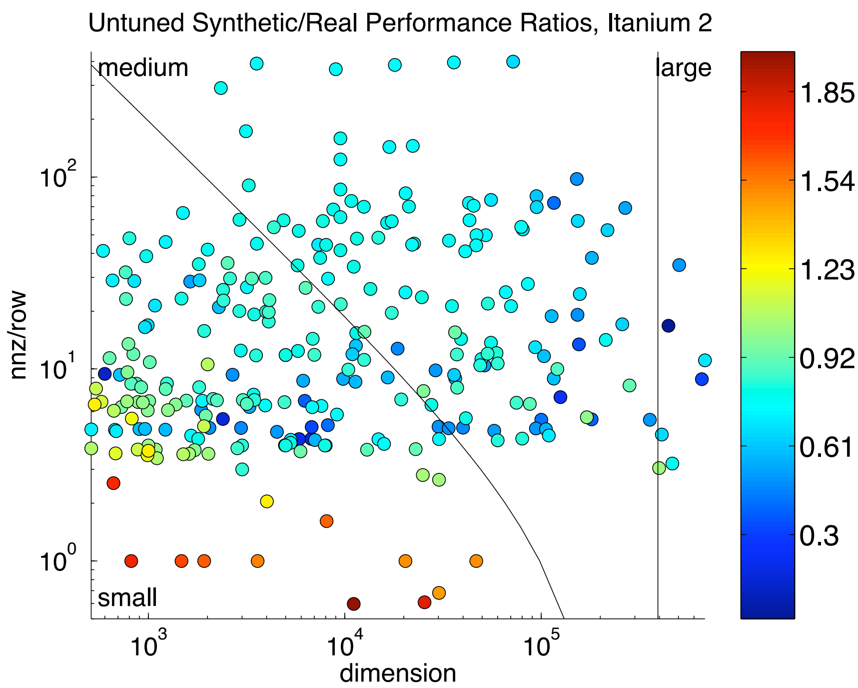


(a) Untuned

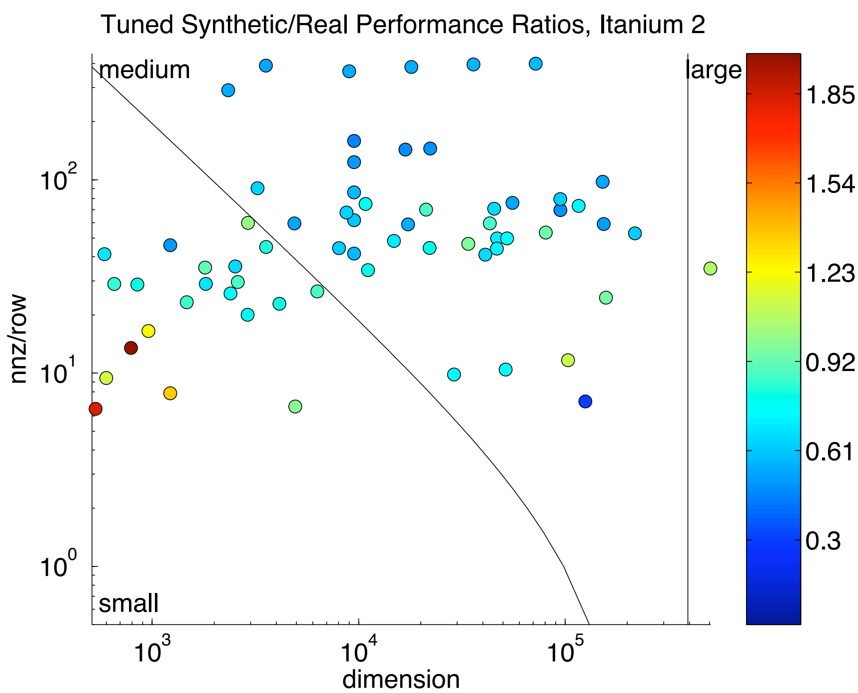


(b) Tuned

Figure 2.24: Real vs. Synthetic matrices on the Pentium 4.

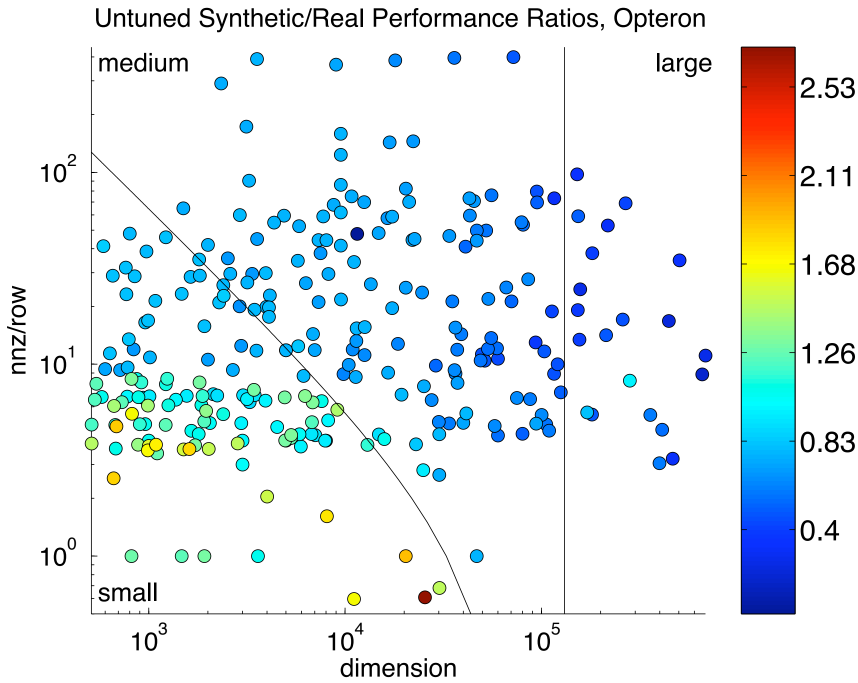


(a) Untuned

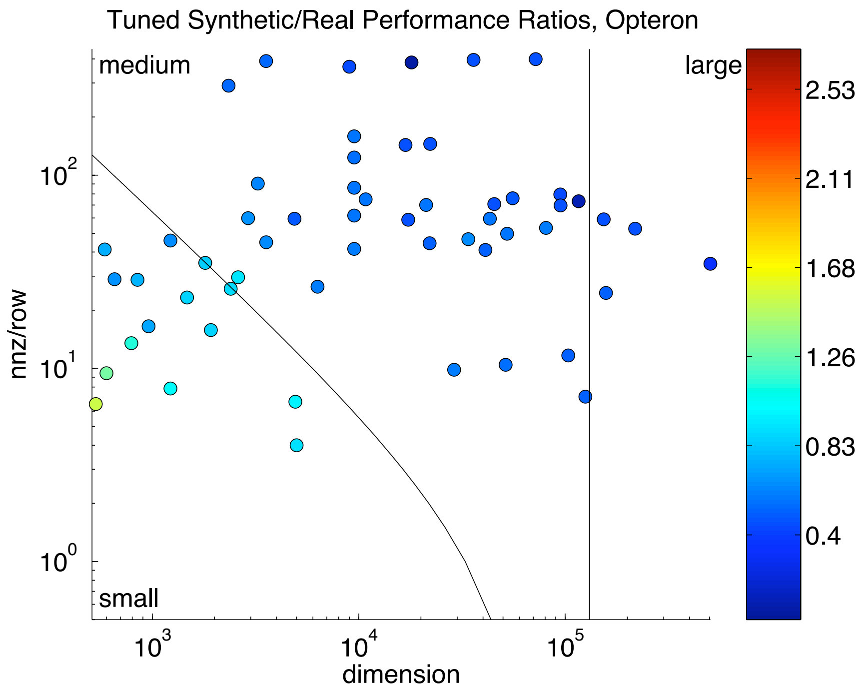


(b) Tuned

Figure 2.25: Real vs. Synthetic matrices on the Itanium 2.

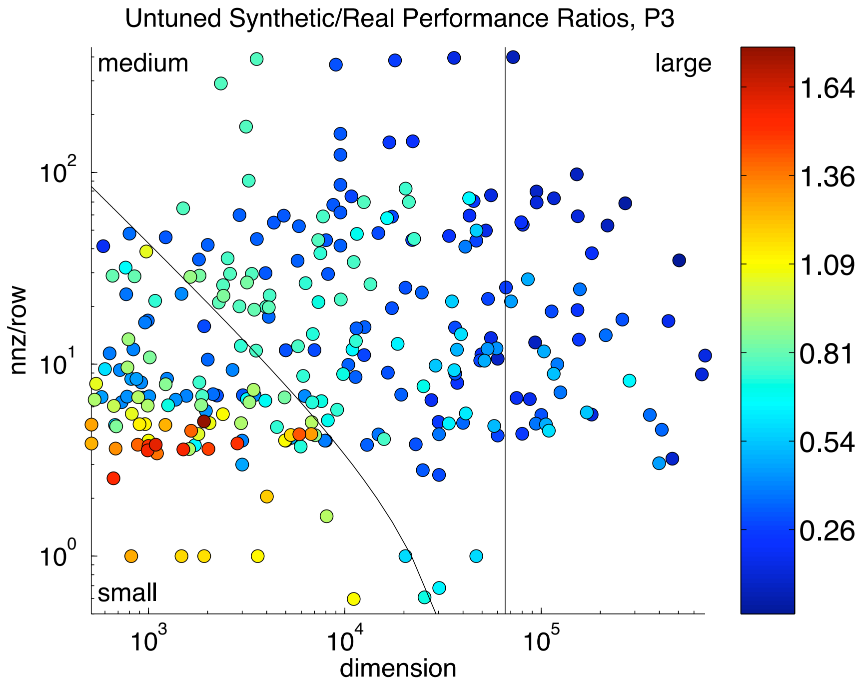


(a) Untuned

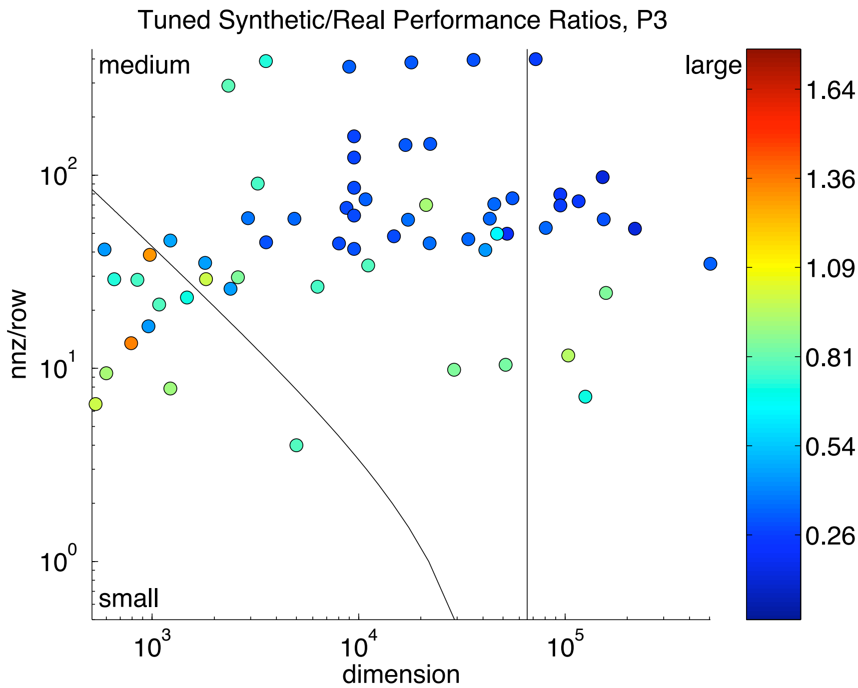


(b) Tuned

Figure 2.26: Real vs. Synthetic matrices on the Opteron.



(a) Untuned



(b) Tuned

Figure 2.27: Real vs. Synthetic matrices on the Pentium 3.

Chapter 3

A Benchmark For Evaluating SpMV Performance

We have seen that we have no better way to predict the performance of SpMV than to perform actual SpMV, and that we can use synthetically generated matrices to effectively approximate SpMV performance on real matrices. Even with synthetically generated matrices, though, there is a very large space of matrices to consider, determined by their density, dimensions, block structure, and nonzero distribution. The matrices in [3] have dimensions ranging from under 100 to over 1 million, with densities ranging from just one nonzero entry per row to almost 400 nonzero entries per row and many possible blocksizes. The matrices in our test suite that form a subset of [3] were found in the previous chapter to have blocksizes with dimensions range from 1 through 8. This forms a set of 64 possible blocksizes, and with the blocksize varying by platform, all of them could turn up depending on which platform we run the matrices on.

3.1 Limiting the Set of Matrices to Benchmark

Testing every possible synthetic matrix with all possible combinations of density, dimension, blocksize, and nonzero distribution would be prohibitively expensive. Instead of doing this, we will cut down on the search space as follows:

1. Test only square matrices whose dimension is a power of 2 no smaller than 512 and no larger than a user-specified upper limit (which is expressed by the user in terms of a constraint on the amount of memory used, as is done in the HPCC benchmarks [5]). In this thesis, we set the upper limit at $2^{20} \approx 10^6$, which is the smallest power of 2 that is larger than the largest matrix dimension in our test suite.

2. Keep the number of nonzero entries per row within a certain small range. We choose $[24, 34] = 29 \pm 5$ because 29 is the average number of nonzero entries per row in our test suite.

3. Look at only matrices with blocksizes found to be common in [12], which uses a matrix test suite with a large proportion of matrices that benefit from tuning. These blocksizes come from the set $\{r, c\} \in \{1, 2, 3, 4, 6, 8\} \times \{1, 2, 3, 4, 6, 8\}$. Our matrix test suite, meant to represent both matrices we can and cannot tune, has a substantial number of matrices for which tuning has no benefit.

4. Generate matrices with nonzero entries distributed as shown in table 3.1. This distribution is the average distribution over all the matrices in our test suite (the individual distributions for each matrix can be found in Appendix C).

Distance From Diagonal	Entries In This Range
0-10%	65.9%
10-20%	11.4%
20-30%	5.84%
30-40%	6.84%
40-50%	2.85%
50-60%	1.86%
60-70%	1.44%
70-80%	2.71%
80-90%	0.774%
90-100%	0.387%

Table 3.1: Distribution of Nonzero Entries in Matrix Test Suite

To ensure accurate measurements, we first measure the timer resolution, and based on this run SpMV enough times so that the time per SpMV is no smaller than 100 times the timer resolution. This ensures that timer measurement error is at most 1%. To guard against reporting an unusual value, at least 10 trials of SpMV are performed no matter what the problem dimension is.

This testing scheme was run on three different platforms (see Appendix A), producing 36 plots for each machine, one for each block size tested, 12 (# dimensions) $\times$ 11 (# nnz/row) $\times$ 36 (# blocksizes) = 4752 matrices in all. These plots can be found in Appendices H–K.

3.2 Condensing the Reported Data

As the plots in Appendices H–K show, testing all the matrices in this reduced search space still produces a lot of data. Benchmark suites like HPCC want data in the form of just a few numbers (and preferably just one number) [5]. We report four MFLOP rates: unblocked maximum, unblocked median, blocked maximum, and blocked median. The unblocked numbers are taken only from data gathered for matrices with 1×1 blocks, and represent the case of the real-life matrices for which tuning was attempted but found to be of no benefit. The blocked numbers are taken from the rest of the data, and represent the case of the real-life matrices for which there was a benefit to tuning. Based on the patterns in the graphs in Appendices H–K, we feel that the four numbers we report are the ones that best capture SpMV performance.

When forced to report one number for condensed output, as all bencharks in the HPCC suite are [5], we report the median blocked MFLOP rate. The results for each platform tested are

Platform	Unblocked		Blocked	
	Max	Median	Max	Median
Pentium 4	699	307	1961	530
Itanium 2	443	343	2177	753
Opteron	396	170	1178	273
Pentium 3	474	97	1017	172

The proportion of matrices generated by the benchmark that fall into each SpMV problem classification are

Platform	Small	Medium	Large
Pentium 4	17%	42%	42%
Itanium 2	33%	50%	17%
Opteron	23%	44%	33%
Pentium 3	17%	42%	42%

This tells us that we are capturing the small/medium/large behavior that we want to capture.

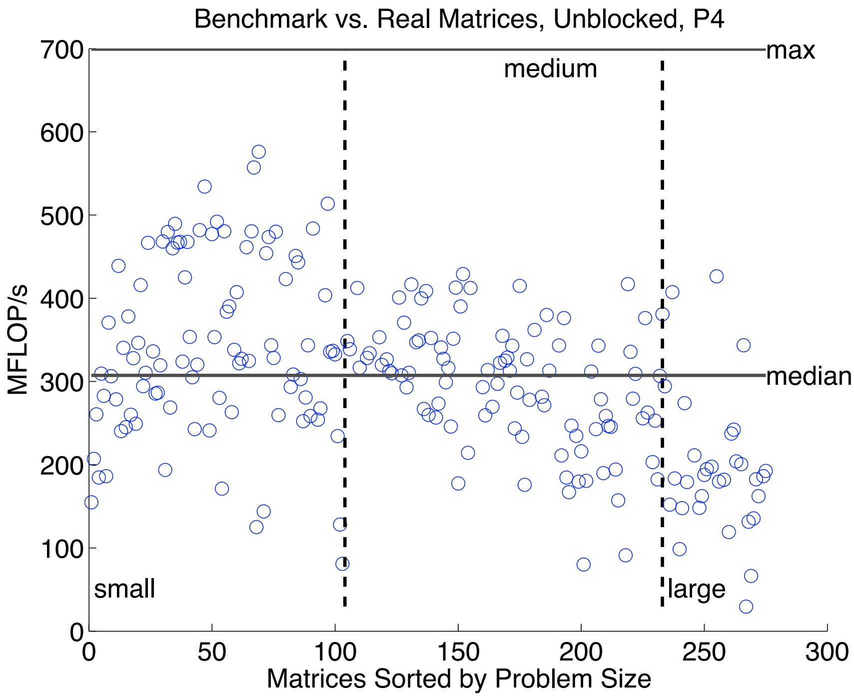
Figures 3.1–3.4 compare our benchmark's output to the performance of real matrices. Plotted in them are maximum, minimum, and median MFLOP rates for each problem dimension, in both the blocked and blocked cases, along with dots to represent real matrices. They show that the numbers output by our benchmark have good predictive power, though the blocked maximum numbers stand out as being noticeably too high.

3.3 Decreasing The Benchmark's Runtime

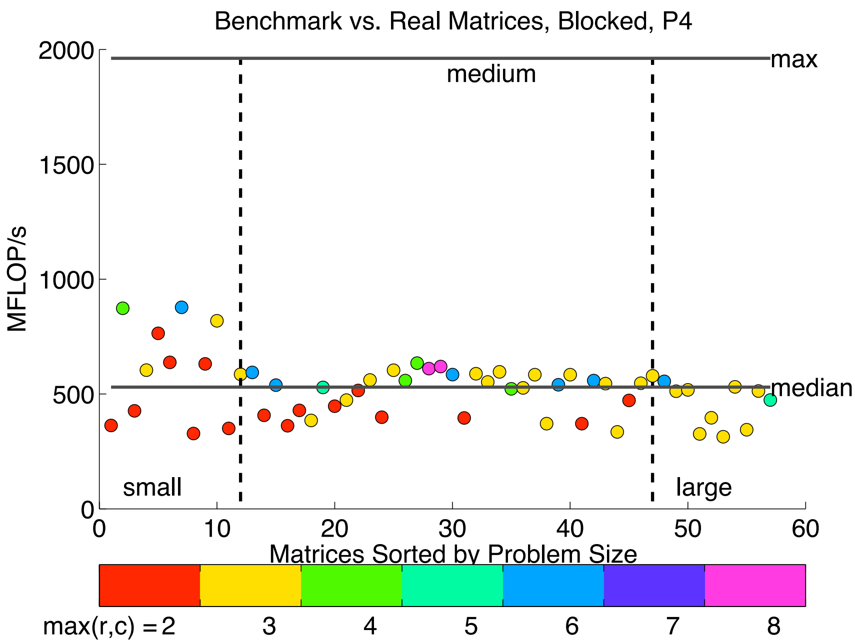
One problem remains with the SpMV benchmarking scheme outlined in the previous sections: its overall runtime. On each machine tested, running the benchmark was quite a time-consuming endeavor, as these runtimes show:

Machine	Runtime (minutes)
Pentium 4	150
Opteron	149
Itanium 2	128
Penium 3	221

We would like to make these runtimes much smaller without drastically affecting the benchmark's output. One obvious way of limiting runtime is to put a further constraint on the largest problem dimension, requiring it not to get so large as to make the benchmark's runtime exceed a certain time limit. Another way can be gleaned from the performance graphs for each blocksize. While the performance of SpMV can be sensitive to the number of nonzero entries per row in a matrix, it is only sensitive for matrices with a very small dimension. Thus, we can also constrain the number of different values of the number of nonzero entries per row when the matrix dimension is not too small. Defining "too small" might seem difficult, but we can easily do this at runtime. To do so, we define what we call a *threshold dimension*. This is a problem dimension below which we consider the task of creating a matrix and performing SpMV with it to be "free." Each register block size for which we generate matrices will have its own threshold dimension. All values of nnz/row

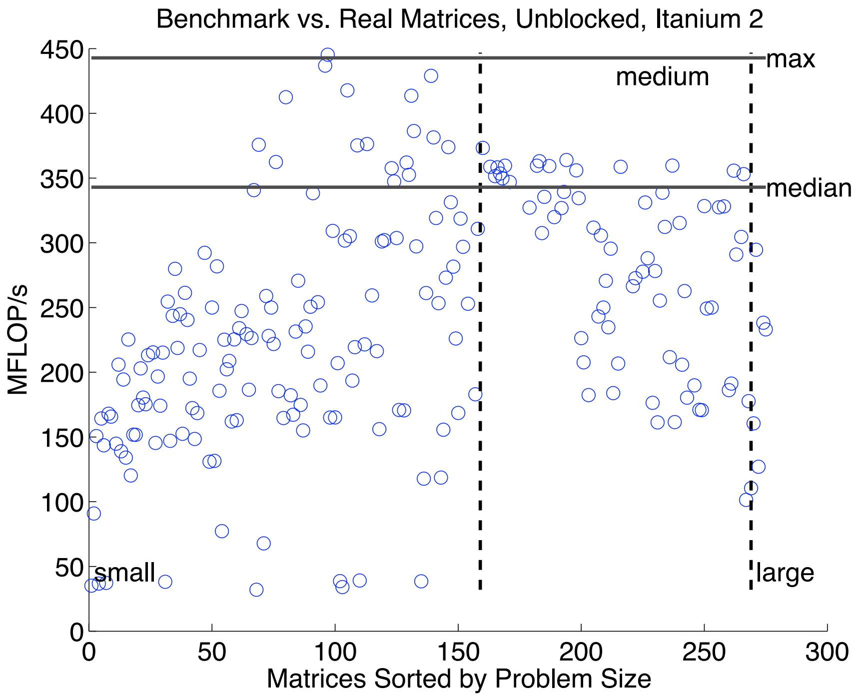


(a) Unblocked benchmark numbers vs. unblocked real matrices

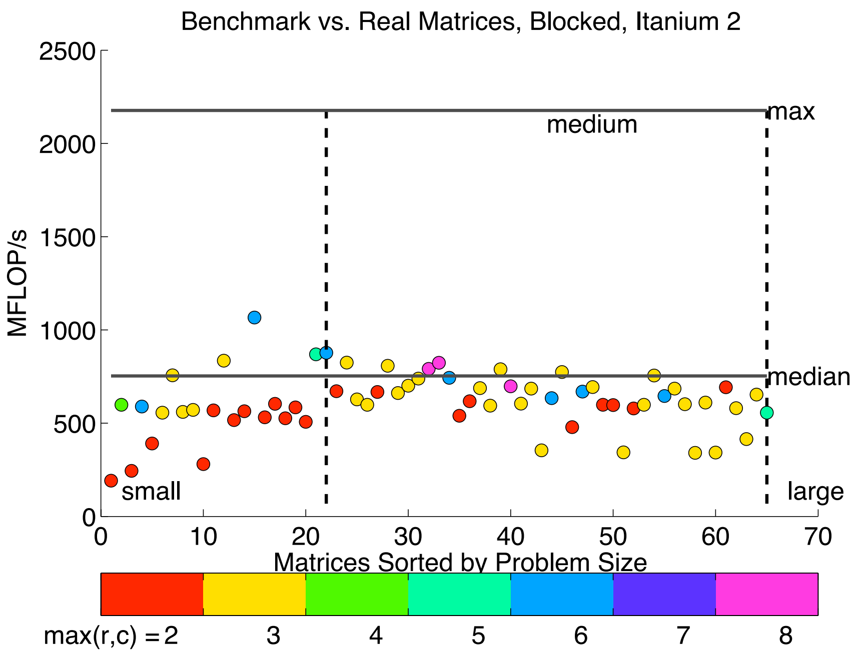


(b) Blocked benchmark numbers vs. blocked real matrices

Figure 3.1: Performance of benchmark vs. real matrices on the Pentium 4.

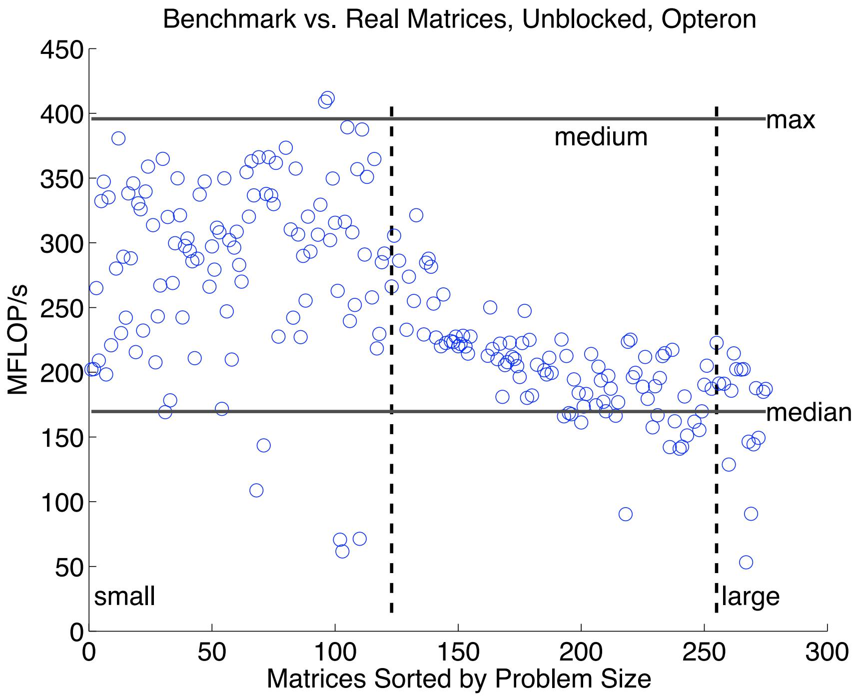


(a) Unblocked benchmark numbers vs. unblocked real matrices

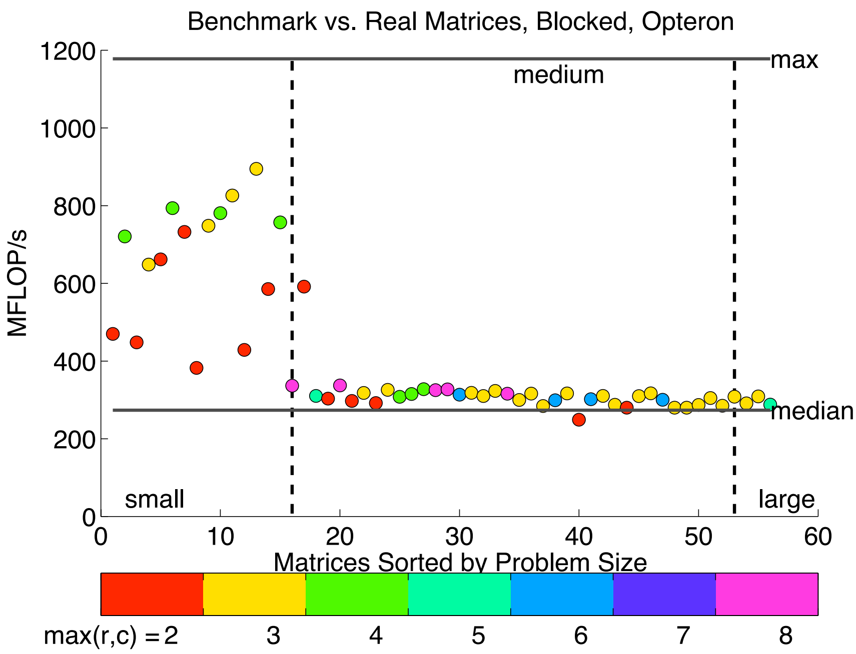


(b) Blocked benchmark numbers vs. blocked real matrices

Figure 3.2: Performance of benchmark vs. real matrices on the Itanium 2.

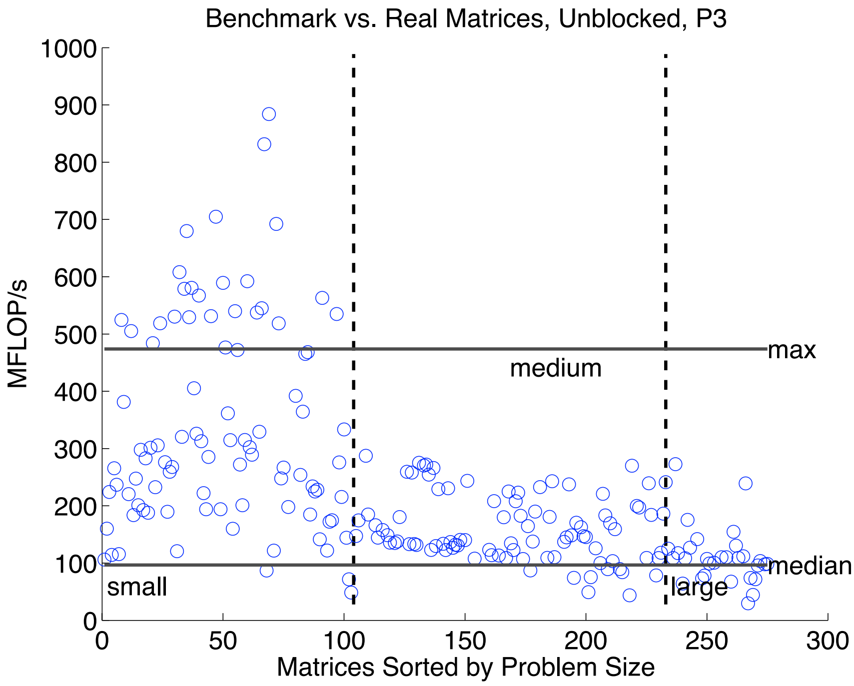


(a) Unblocked benchmark numbers vs. unblocked real matrices

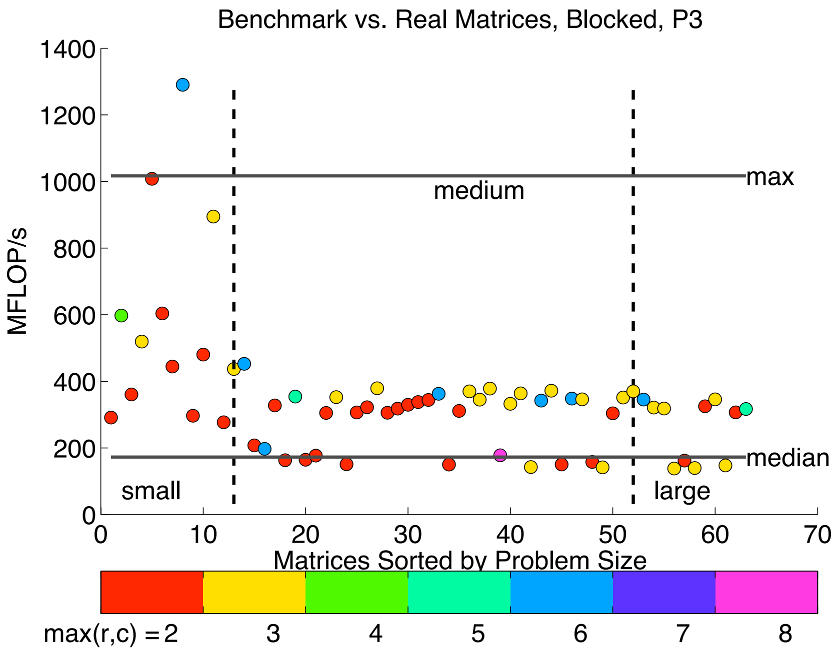


(b) Blocked benchmark numbers vs. blocked real matrices

Figure 3.3: Performance of benchmark vs. real matrices on the Opteron.



(a) Unblocked benchmark numbers vs. unblocked real matrices



(b) Blocked benchmark numbers vs. blocked real matrices

Figure 3.4: Performance of benchmark vs. real matrices on the Pentium 3.

are to be tested for problem dimensions below the threshold dimension. Above it, we can omit values as we see fit.

The actual decision of which trials not to run is made by a runtime estimator that first estimates the runtime of a full benchmark run and then cuts out certain trials from the full run until a user-specified time constraint is satisfied. Based on limitations required by [5], we set this constraint to five minutes. Runtime estimation is performed, for each register blocksize tested, by running an SpMV trial (both matrix generation and performing the actual multiplication) for a matrix with the threshold dimension defined earlier and then doubling the runtime of this trial to obtain runtime estimates for running an SpMV trial for each successive problem dimension we intend to test. The estimator keeps nnz/row at the midpoint of the selected range, and adds up the computed estimates to estimate the runtime of the entire benchmark. Afterwards, it uses the following iteration to determine which SpMV trials to omit:

1. Reduce the number of values of nnz/row to test and adjust the runtime estimate accordingly.

2. If the time limit is still exceeded, cut off the largest dimension to be tested and go back to testing the full nnz/row range, adjusting the estimate accordingly.

3. Repeat the previous two steps until the time limit is satisfied. To account for estimation errors, we allow the time limit to be exceeded by 10%.

This keeps the largest problem dimension tested as large as possible to ensure that the benchmark tests small, medium, and large problems, while still dramatically cutting down on the runtime. Before we see the results, though, we note that cutting out values of nnz/row to test cuts out data points which would greatly change our median statistics because we test the full nnz/row range for problem dimensions below the threshold dimension. To correct this problem, we replace these missing data points so our computed medians make sense by either duplication if we test only one nnz/row value or interpolation if we test more than one. In the latter case, we require the endpoints of the nnz/row range to be included among the nnz/row values tested.

3.4 Reduced-Time Benchmark Results

Running the benchmark from scratch with the time-saving measures described in the previous section and a time limit of 5 minutes yielded the following condensed results, which are not too much different from the results obtained in section 3.2 by running for over two hours. The difference in max numbers between the full and abbreviated runs, which would ideally be the same, fall within the bounds of measurement noise.

	Untuned		Tuned	
	Max	Median	Max	Median
Pentium 4	692	362	1937	555
Itanium 2	442	343	2181	803
Opteron	394	188	1178	286
Pentium 3	474	113	1010	180

The actual runtimes for each platform in minutes were

	Runtime (original)	Runtime (condensed)
Pentium 4	150 minutes	3 minutes
Itanium 2	128 minutes	3 minutes
Opteron	149 minutes	3 minutes
Penium 3	221 minutes	5 minutes

That the runtime was 3 minutes and not 5 means the runtime estimator determined that testing the next largest problem dimension would have pushed the benchmark's runtime over the time limit. The proportion of generated matrices falling in each SpMV category in this reduced test space are as follows:

	Small	Medium	Large
Pentium 4	20%	50%	30%
Itanium 2	40%	60%	0%
Opteron	27%	53%	20%
Penium 3	20%	50%	30%

Chapter 4

Conclusions and Directions for Future Work

In this thesis, we have presented a way to benchmark SpMV that works on multiple architectures, through which accurate results can be obtained in as little as five minutes. Many areas for future work remain, however. A number of them have already been highlighted by graphs in the previous two chapters. We will discuss these and other areas for improvement here. These range from improving the benchmark itself to extending it to new platforms.

4.1 Improvements to the Benchmark

4.1.1 Synthetic Matrix Generation

As the graphs in the previous two chapters show, there is still plenty of room for improvement in the benchmark itself. The graphs of real versus synthetic matrices in

Chapter 2 tell us that further research into how to generate synthetic matrices that better approximate real ones could give us more a more accurate benchmark. The ends of the spectrum that need to be given the most attention are the small and large problems. For small problems, the synthetic matrices often mispredict the performance of real ones, and we have not yet found a way to correct this. For large problems, even though using nonzero distribution statistics have helped, the synthetic matrices still noticeably underpredict the performance of the real ones.

4.1.2 Benchmark Output

Perhaps the biggest area for improvement comes when looking at the maximum blocked MFLOP rate output by the benchmark. It is clearly very high in most cases. One reason comes from the synthetic matrices with blocksizes on the larger end of the ones we use in out benchmark. While they are represented in larger problems, they are not very often found in small problems, as a scan through [3] reveals. But this is not the only issue at work. There is also the general problem of synthetic matrices mispredicting the performance of real ones, which happens regardless of blocksize. Research addressing these issues is needed to make the benchmark's output more accurate.

4.1.3 Symmetric Matrices

Our benchmark in its current state generates exclusively nonsymmetric matrices. This ignores a large number of real-life matrices that are symmetric. In this thesis, we have converted symmetric real-life matrices to nonsymmetric format before performing SpMV on them. But since symmetric matrices form an important subclass of SpMV problems, being able to predict symmetric SpMV performance is something that would be desirable in an SpMV benchmark. Figures 4.1–4.4 show that our benchmark retains some predictive power when we try and predict symmetric performance. However, because symmetric matrices are frequently found in real-life examples, we would like to be able to somehow incorporate symmetry into our benchmark. How this can best be done is a question that needs further research.

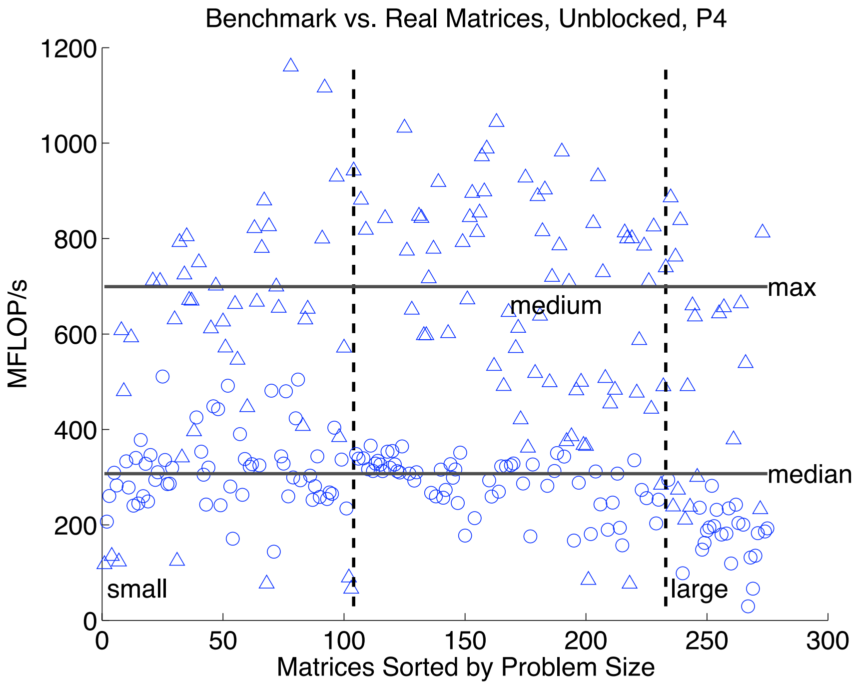
4.1.4 Other Benchmarking Techniques

One current line of research raises the question as to whether other benchmarking approaches are possible [11]. This work seeks to measure application performance by modeling its memory access patterns. If such an approach could be extended to SpMV, it would mean that future approaches to benchmarking it could be much simpler than the one we have given in this thesis.

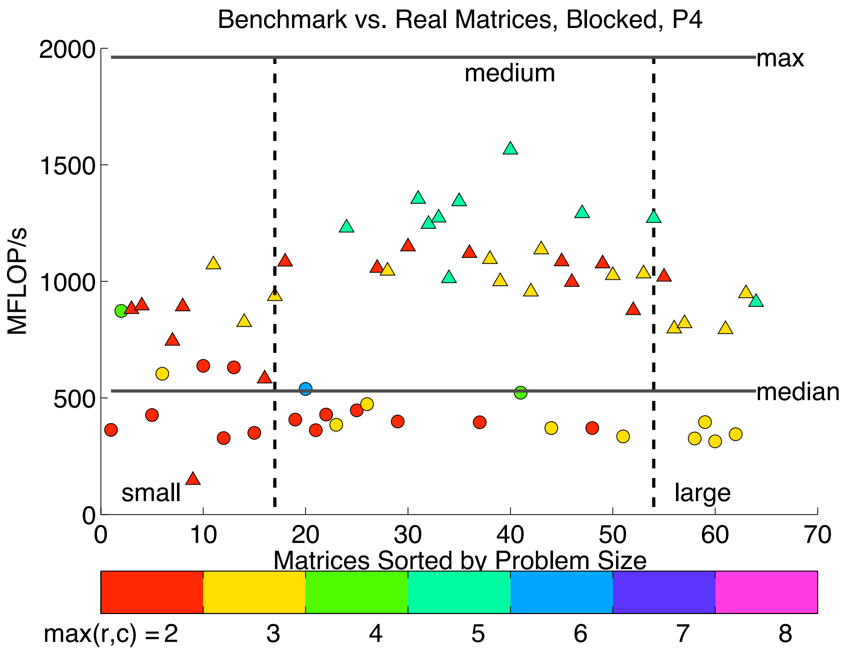
4.2 Extending the Benchmark to New Platforms

Currently, the benchmark only works on scalar uniprocessor machines. This is not an ideal situation, as there are two other kinds of architectures often used in scientific computing that would also benefit from an SpMV benchmark.

The first are vector machines. As discussed in [2] and [12], the CSR and BCSR sparse matrix formats that we used in this thesis may not be fastest for vector architectures. [2] presents a format called *segmented scan* that is very well-suited to vector architectures. The use of this or another suitable format could serve to extend our benchmark

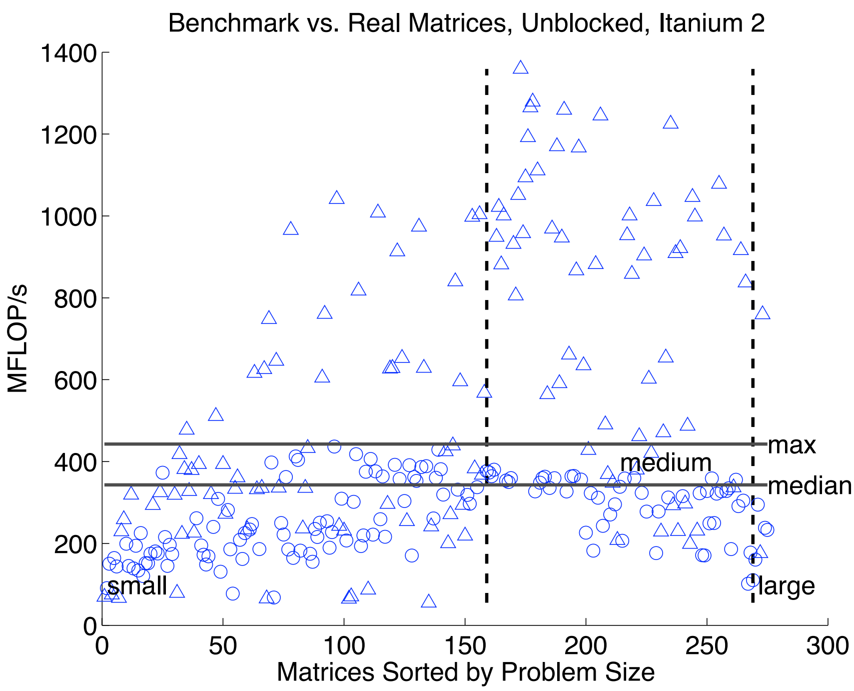


(a) Unblocked benchmark numbers vs. unblocked real matrices

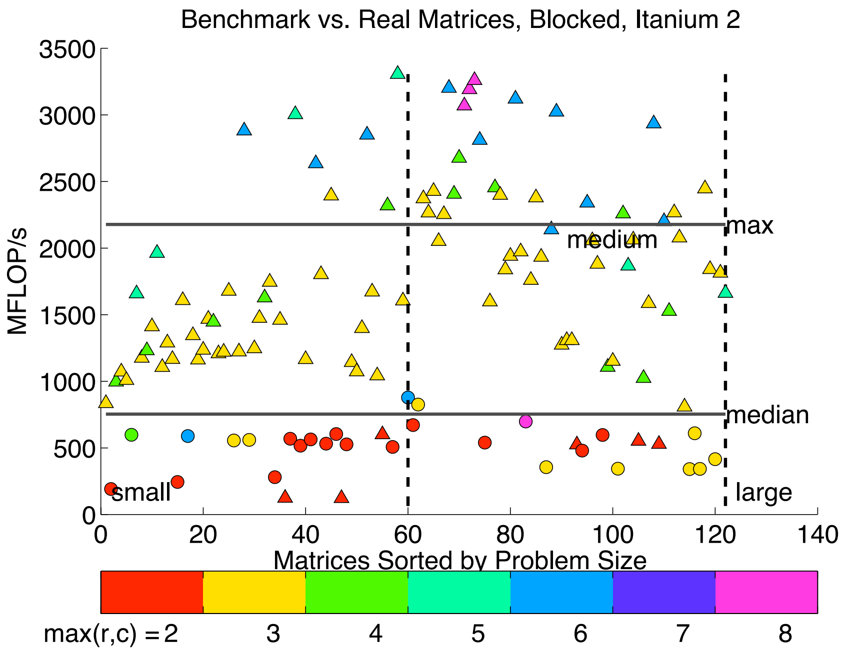


(b) Blocked benchmark numbers vs. blocked real matrices

Figure 4.1: Performance of benchmark vs. real matrices on the Pentium 4 with symmetry taken into account. Triangles represent symmetric matrices and circles represent nonsymmetric ones.

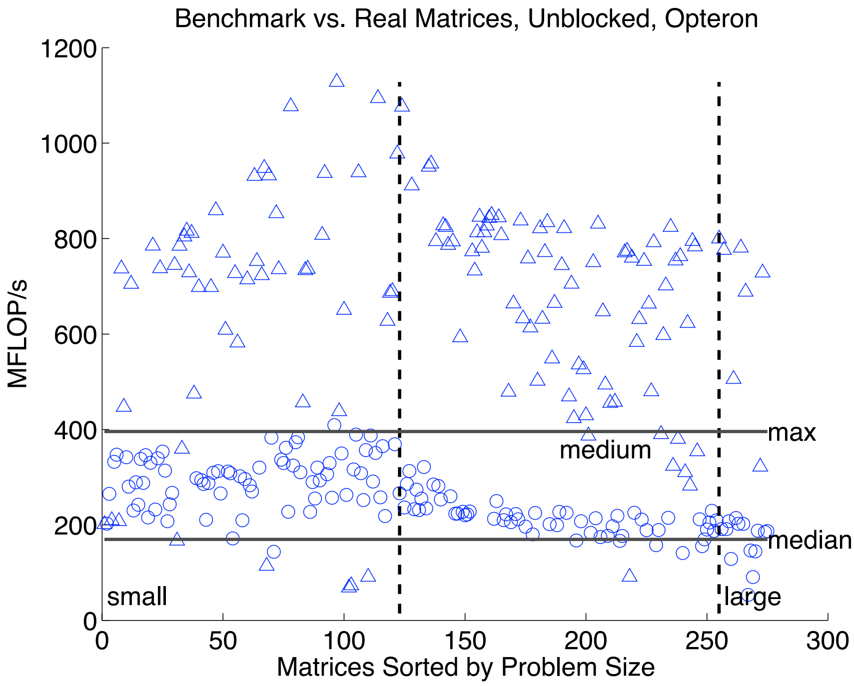


(a) Unblocked benchmark numbers vs. unblocked real matrices

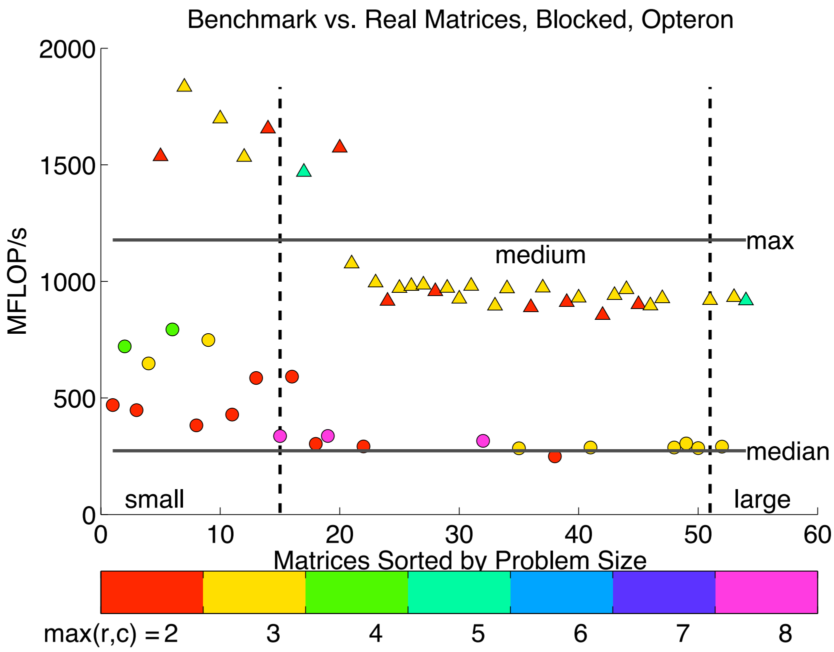


(b) Blocked benchmark numbers vs. blocked real matrices

Figure 4.2: Performance of benchmark vs. real matrices on the Itanium 2 with symmetry taken into account. Triangles represent symmetric matrices and circles represent nonsymmetric ones.

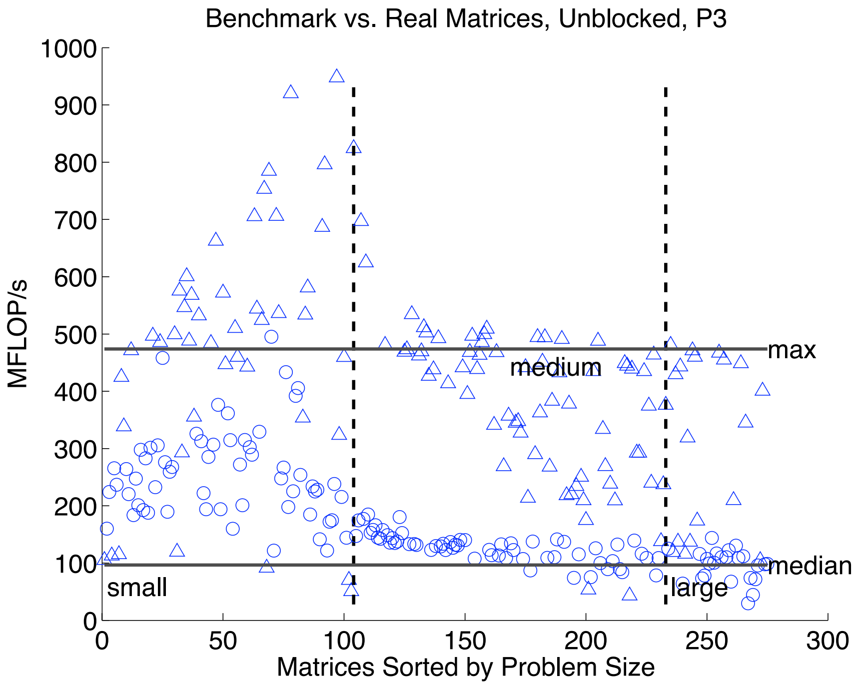


(a) Unblocked benchmark numbers vs. unblocked real matrices

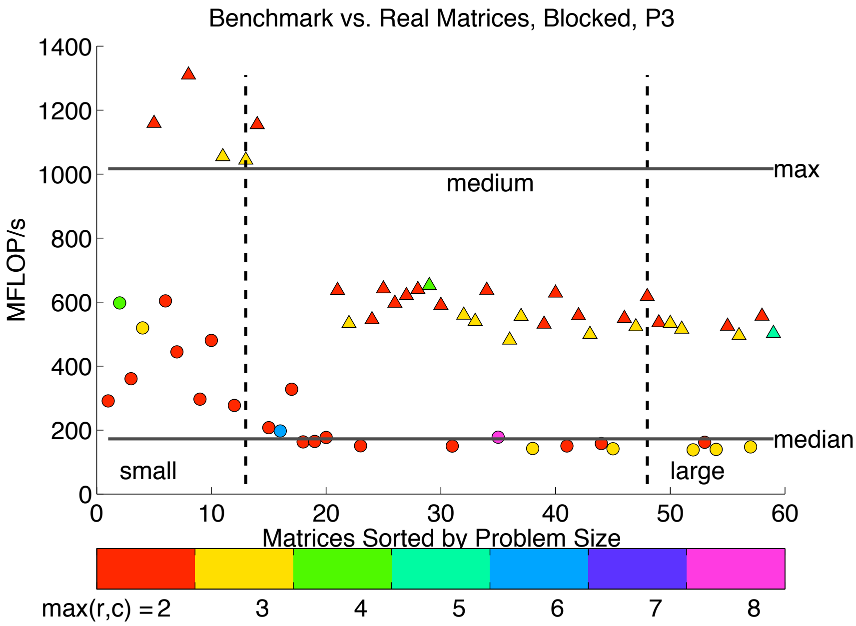


(b) Blocked benchmark numbers vs. blocked real matrices

Figure 4.3: Performance of benchmark vs. real matrices on the Opteron with symmetry taken into account. Triangles represent symmetric matrices and circles represent nonsymmetric ones.



(a) Unblocked benchmark numbers vs. unblocked real matrices



(b) Blocked benchmark numbers vs. blocked real matrices

Figure 4.4: Performance of benchmark vs. real matrices on the Pentium 3 with symmetry taken into account. Triangles represent symmetric matrices and circles represent nonsymmetric ones.

to vector architectures.

The second are parallel machines. Today, most intense scientific computations are done on parallel machines, so any benchmark that measures the performance of an operation used in scientific computing should run on those machines. The benchmarks currently in [5] all have parallel versions, and a benchmark for SpMV should be no exception.

A number of issues come up in the parallel case, though, that do not come up when looking at SpMV on uniprocessor machines. These make designing a parallel SpMV benchmark a challenging problem that needs much further attention. Issues such as the way a matrix is distributed amongst processors and how to handle the different cases of parallel machines, shared-memory machines, SMP's, and distributed-memory machines become very important, and research is needed on the best way for a portable SpMV benchmark to accomodate these differences.

Bibliography

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report 94-007, RNR, March 1994. http://www.nas.nasa.gov/Resources/Software/npb.html.

[2] G. Belloch, M. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, Carnegie Mellon University, 1993.

[3] T. Davis. University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices.

[4] J. Dongarra, V. Eijkhout, and H. van der Vorst. Sparsebench: a sparse iterative benchmark, 2001.

[5] J. Dongarra and P. Luszczek. Introduction to the HPC Challenge Benchmark Suite. Technical Report UT-CS-05-544, University of Tennessee, Knoxville, 2005.

[6] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in

SPARSITY. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 127–136, San Francisco, CA, May 2001. Springer.

[7] E.-J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[8] J. D. McCalpin. Sustainable memory bandwidth in current high performance computers. Technical report, Advanced Systems Division, Silicon Graphics, Inc., 1995.

[9] National Institute of Science and Technology. SciMark 2.0 Java Benchmark for Scientific Computing. http://math.nist.gov/scimark2.

[10] R. Nishtala, R. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication, and Computing: Special Issue on Computational Linear Algebra and Sparse Matrix Computations*, 2005. (*to appear*).

[11] E. Strohmaier and H. Shan. Architecture independent performance characterization and benchmarking for scientific applications. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Volendam, The Netherlands, October 2004.

[12] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, December 2003.

[13] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse

matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing. (*to appear*).

[14] R. Vuduc, J. W. Demmel, and K. A. Yelick. An interface for a self-optimizing sparse matrix kernel library, 2005.

[15] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

Appendix A

Experimental Setup

Here are details on the platforms experiments were carried out on. Included are information about the processors, cache sizes, compiler, and compiler flags used.

	GHz	Bits	Cache	Compiler	Compiler Command
Pentium 4	2.4	32	512 KB	gcc 3.4.4	`cc -O3 -malign-double -march=pentium4 -msse2 -std=c99`
Itanium 2	1	64	3 MB	Intel C v9.0	`icc -O3 -tpp2 -mcpu=itanium2 -std=c99`
Opteron	1.4	64	1 MB	gcc 3.2.3	`cc -O3 -m64 -march=k8 -std=c99`
Pentium 3	1.4	32	512 KB	Intel C v9.0	`icc -O3 -tpp6 -mcpu=pentiumpro -march=pentiumiii -std=c99`

Appendix B

Suite of Test Matrices

The real matrices for which performance data was collected were taken from the online collection [3]. The following table gives information on the properties of each matrix that was used.

Name	Dimension	Nonzeros	nnz/row	Symmetric
dw256A	512	2480	5	no
gre_512	512	1976	4	no
pores_3	532	3474	7	no
fs_541_1	541	4282	8	no
lshp_577	577	3889	7	yes
bcsstm34	588	24270	41	yes
steam2	600	5660	9	no
can_634	634	7228	11	yes
ex21	656	18964	29	no
shl_0	663	1687	3	no
nnc666	666	4032	6	no
nos6	675	3255	5	yes
fs_680_3	680	2471	4	no
685_bus	685	3249	5	yes
can_715	715	6665	9	yes
nos7	729	4617	6	yes
mcfe	765	24382	32	no

Name	Dimension	Nonzeros	nnz/row	Symmetric
Si2	769	17801	23	yes
lshp_778	778	5272	7	yes
bfwa782	782	7514	10	no
rotor2	791	10685	14	no
G1	800	38352	48	yes
bcsstk19	817	6853	8	yes
bcsstm19	817	817	1	yes
bp_800	822	4534	6	no
can_838	838	10010	12	yes
ex25	848	24369	29	no
dwt_869	869	7285	8	yes
qh882	882	3354	4	no
orsirr_2	886	5970	7	no
pde900	900	4380	5	no
dwt_918	918	7384	8	yes
jagmesh1	936	6264	7	yes
nos3	960	15844	17	yes
cdde1	961	4681	5	no
ex27	974	37652	39	no
west0989	989	3518	4	no
jpwh_991	991	6027	6	no
dwt_992	992	16744	17	yes
saylr3	1000	3750	4	no
tub1000	1000	3996	4	no
lshp1009	1009	6865	7	yes
cage8	1015	11003	11	no
sherman2	1080	23094	21	no
b_dyn	1089	4144	4	no
sherman4	1104	3786	3	no
fpga_dcop_01	1220	5892	5	no
bcsstk27	1224	56126	46	yes
pores_2	1224	9613	8	no
dwt_1242	1242	10426	8	yes
mahindas	1258	7682	6	no
jagmesh6	1377	8993	7	yes
bcsstk11	1473	34241	23	yes
bcsstm11	1473	1473	1	yes
comsol	1500	97465	65	no
west1505	1505	5414	4	no
lshp1561	1561	10681	7	yes
bcspwr07	1612	5824	4	yes

Name	Dimension	Nonzeros	nnz/row	Symmetric
ex7	1633	46626	29	no
lung1	1650	7419	4	no
bcspwr09	1723	6511	4	yes
epb0	1794	7764	4	no
bcsstk14	1806	63454	35	yes
adder_trans_01	1814	14579	8	no
ex3	1821	52685	29	no
watt_1	1856	11360	6	no
rajat12	1879	12818	7	no
plsk1919	1919	9662	5	no
bcsstk26	1922	30336	16	yes
bcsstm26	1922	1922	1	yes
rajat02	1960	11187	6	yes
bcsstk13	2003	83883	42	yes
bcsstm13	2003	21181	11	yes
west2021	2021	7310	4	no
dw1024	2048	10114	5	no
blckhole	2132	14872	7	yes
lshp2233	2233	15337	7	yes
ex24	2283	47901	21	no
heart2	2339	680341	291	no
add20	2395	13151	5	no
rdist3a	2398	61896	26	no
ex10	2410	54840	23	no
orani678	2529	90158	36	no
ex28	2603	77031	30	no
dwt_2680	2680	25026	9	yes
extr1	2837	10967	4	no
meg1	2904	58142	20	no
nasa2910	2910	174296	60	yes
lhr02	2954	36875	12	no
pde2961	2961	14585	5	no
G50	3000	12000	4	yes
laser	3002	9000	3	yes
lshp3025	3025	20833	7	yes
psmigr_3	3140	543160	173	no
swang1	3169	20841	7	no
garon1	3175	84723	27	no
raefsky2	3242	293551	91	no
bayer05	3268	20712	6	no
ex9	3363	99471	30	no

Name	Dimension	Nonzeros	nnz/row	Symmetric
shermanACa	3432	25220	7	no
thermal	3456	66528	19	no
lshp3466	3466	23896	7	yes
cage9	3534	41594	12	no
heart1	3557	1385317	389	no
bcsstk24	3562	159910	45	yes
bcsstm21	3600	3600	1	yes
lns_3937	3937	25407	6	no
bcsstk15	3948	117816	30	yes
ex12	3973	79077	20	no
poli	4008	8188	2	no
sts4098	4098	72356	18	yes
lhr04	4101	81057	20	no
rdist1	4134	94408	23	no
struct4	4350	237798	55	yes
circuit_2	4510	21199	5	no
bcsstk16	4884	290378	59	yes
gemat11	4929	33108	7	no
add32	4960	19848	4	no
G58	5000	59140	12	yes
G59	5000	59140	12	yes
olm5000	5000	19996	4	no
bcspwr10	5300	21842	4	yes
hydr1	5308	22680	4	no
SiNa	5743	198787	35	yes
ex18	5773	71701	12	no
Na5	5832	305630	52	yes
meg4	5860	25258	4	no
Hamrle2	5952	22162	4	no
shermanACd	6136	53329	9	no
Alemdar	6245	42581	7	yes
raefsky5	6316	167178	26	no
bayer03	6747	29195	4	no
jan99jac020	6774	33744	5	no
rajat01	6833	43250	6	no
ex15	6867	98671	14	no
G64	7000	82918	12	yes
cell1	7055	30082	4	no
goodwin	7320	324772	44	no
lhr07	7337	154660	21	no
sinc12	7500	283992	38	no

Name	Dimension	Nonzeros	nnz/row	Symmetric
rajat13	7598	48762	6	no
ex40	7740	456188	59	no
commanche_dual	7920	31680	4	yes
G65	8000	32000	4	yes
bcsstk38	8032	355460	44	yes
Pd	8081	13036	2	no
dw4096	8192	41746	5	no
benzene	8219	242669	30	yes
bcsstk33	8738	591904	68	yes
nd3k	9000	3279690	364	yes
mark3jac020	9129	52883	6	no
nemeth02	9506	394808	42	yes
nemeth16	9506	587012	62	yes
nemeth19	9506	818302	86	yes
nemeth21	9506	1173746	123	yes
nemeth26	9506	1511760	159	yes
coater2	9540	207308	22	no
fv2	9801	87025	9	no
shuttle_eddy	10429	103599	10	yes
pkustk02	10800	810000	75	yes
igbt3	10938	130500	12	no
k3plates	11107	378927	34	no
m3plates	11107	6639	1	yes
coupled	11341	97193	9	yes
cage10	11397	150645	13	no
t2dah_a	11445	176117	15	yes
sinc15	11532	551184	48	no
sme3Da	12504	874887	70	no
stokes64	12546	140034	11	yes
skirt	12598	196520	16	yes
tuma2	12992	49365	4	yes
poisson3Da	13514	352762	26	no
Pres_Poisson	14822	715804	48	yes
rajat07	14842	63913	4	yes
powersim	15838	64424	4	no
sinc18	16428	948696	58	no
pds10	16558	149658	9	yes
pkustk07	16860	2418804	143	yes
gyro_k	17361	1021159	59	yes
gyro_m	17361	340431	20	yes
nd6k	18000	6897316	383	yes

Name	Dimension	Nonzeros	nnz/row	Symmetric
nmos3	18588	237130	13	no
bodyy6	19366	134208	7	yes
t3dl_a	20360	509866	25	yes
t3dl_e	20360	20360	1	yes
ns3Da	20414	1679599	82	no
raefsky3	21200	1488768	70	no
pkustk01	22044	979380	44	yes
pkustk08	22209	3226671	145	yes
rim	22560	1014951	45	no
tuma1	22967	87760	4	yes
crystm03	24696	583770	24	yes
dtoc	24993	69972	3	yes
mult_dcop_01	25187	193276	8	no
bcsstm37	25503	15525	1	yes
brainpc2	27607	179395	6	yes
3D_28984_Tetra	28984	285092	10	no
bloweya	30004	150009	5	yes
aug2dc	30200	80000	3	yes
rajat10	30202	130303	4	yes
bcsstm35	30237	20619	1	yes
Zhao1	33861	166453	5	no
pkustk09	33960	1583640	47	yes
lhr34	35152	746972	21	no
nd12k	36000	14220946	395	yes
onetone1	36057	335552	9	no
wathen120	36441	565761	16	yes
pwt	36519	326017	9	yes
rajat15	37261	443573	12	no
finance256	37376	298496	8	yes
cage11	39082	559722	14	no
torsion1	40000	197608	5	yes
av41092	41092	1683902	41	no
jan99jac120	41374	229385	6	no
sme3Dc	42930	3148656	73	no
pkustk06	43164	2571768	60	yes
3dtube	45330	3213618	71	yes
bcsstk39	46772	2060662	44	yes
bcsstm39	46772	46772	1	yes
rma10	46835	2329092	50	no
gridgena	48962	512084	10	yes
stokes128	49666	558594	11	yes

Name	Dimension	Nonzeros	nnz/row	Symmetric
ibm_matrix_2	51448	537038	10	no
ct20stif	52329	2600295	50	yes
g7jac180	53370	641290	12	no
struct3	53570	1173694	22	yes
copter2	55476	759952	14	yes
pkustk04	55590	4218660	76	yes
bayer01	57735	275094	5	no
g7jac200	59310	717620	12	no
a5esindl	60008	255004	4	yes
blockqp1	60012	640033	11	yes
qa8fk	66127	1660579	25	yes
lhr71	70304	1494006	21	no
nd24k	72000	28715634	399	yes
ncvxqp3	75000	499964	7	yes
t3dh_e	79171	4352105	55	yes
a2nnsnsl	80016	347222	4	yes
pkustk10	80676	4308984	53	yes
poisson3Db	85623	2374949	28	no
ncvxqp7	87500	574962	7	yes
boyd1	93279	1211231	13	yes
tandem_dual	94069	460493	5	yes
pkustk12	94653	7512317	79	yes
pkustk13	94893	6616827	70	yes
ford2	100196	544688	5	yes
matrix_9	103430	1205518	12	no
hcircuit	105676	513072	5	no
lung2	109460	492564	4	no
barrier2-1	113076	2129496	19	no
torso2	115967	1033473	9	no
torso1	116158	8516500	73	no
twotone	120750	1206265	10	no
matrix-new_3	125329	893984	7	no
pkustk14	151926	14836504	98	yes
para-6	153226	2930882	19	no
gearbox	153746	9080404	59	yes
para-10	155924	2094873	13	no
xenon2	157464	3866688	25	no
scircuit	170998	958936	6	no
cont-300	180895	988195	5	yes
ohne2	181343	6869939	38	no
stomach	213360	3021648	14	no

Name	Dimension	Nonzeros	nnz/row	Symmetric
pwtk	217918	11524432	53	yes
torso3	259156	4429042	17	no
Ga41As41H72	268096	18488476	69	yes
Stanford	281903	2312497	8	no
rajat24	358172	1946979	5	no
language	399130	1216334	3	no
rajat21	411676	1876011	5	no
cage13	445315	7479343	17	no
boyd2	466316	1500397	3	yes
af_shell1	504855	17562051	35	yes
pre2	659033	5834044	9	no
Stanford_Berkeley	683446	7583376	11	no

Appendix C

Nonzero Distributions of the Matrix Test Suite

Here are the statistics for distribution of nonzero entries in the matrices in our test suite. The numbers in the table are the percentages of the entries of each matrix that are in decile i, i.e. between $10(i-1)$ and $10i$ percent of the diagonal. Each number is rounded to the nearest integer. A blank entry means that there are either no entries in that decile or that the amount of entries is less than 1% after rounding.

	Decile									
	1	2	3	4	5	6	7	8	9	10
`dw256A`	98%					2%				
`gre_512`	82%	18%								
`pores_3`	69%	31%								
`fs_541_1`	38%	10%	11%	8%	11%	8%	5%	5%	3%	1%
`lshp_577`	85%	9%	2%	1%	1%	1%				
`bcsstm34`	39%	61%	1%							
`steam2`	17%				9%	72%	2%			
`can_634`	63%	9%	7%	6%	3%	2%	2%	5%	3%	

	Decile									
	1	2	3	4	5	6	7	8	9	10
ex21	46%	52%	2%							
shl_0	2%	8%	31%	26%	3%	5%	5%	4%	11%	3%
nnc666	74%	25%	1%							
nos6	100%									
fs_680_3	43%	15%	16%	3%	8%	5%	5%		5%	2%
685_bus	79%	14%	4%	1%			2%	1%		
can_715	79%	6%	6%	2%	4%		2%	2%		
nos7	72%	28%								
mcfe	69%	24%	8%							
Si2	59%	12%	10%	6%	4%	5%	2%	1%		
lshp_778	87%	8%	2%	1%	1%	1%				
bfwa782	48%	5%		8%	10%	10%	10%	8%		
rotor2	87%	1%	1%	2%	1%	2%	1%	2%	1%	3%
G1	9%	18%	16%	14%	12%	10%	8%	6%	4%	2%
bcsstk19	84%		2%	9%	3%	2%				
bcsstm19	100%									
bp_800	9%	18%	14%	13%	14%	11%	10%	7%	3%	1%
can_838	41%	5%	19%	20%	2%	11%			1%	
ex25	56%	44%								
dwt_869	97%	2%					1%			
qh882	49%	6%	5%	7%	9%	15%	9%			
orsirr_2	69%	20%	2%	3%	2%	2%	4%			
pde900	100%									
dwt_918	81%	5%	2%	1%	1%	1%	3%	4%	2%	1%
jagmesh1	83%	14%	2%						1%	
nos3	100%									
cdde1	100%									
ex27	53%	47%								
west0989	23%	22%	22%	15%	10%	6%	1%		1%	1%
jpwh_991	49%	50%	1%							
dwt_992	50%					50%				
saylr3	83%	17%								
tub1000	100%									
lshp1009	89%	7%	1%	1%	1%					
cage8	53%	25%	12%	8%	2%					
sherman2	70%		30%							
b_dyn	6%	13%	10%	12%	12%	13%	12%	9%	8%	4%
sherman4	81%			19%						
fpga_dcop_01	35%	11%	8%	10%	10%	11%	9%	5%		
bcsstk27	100%									
pores_2	73%	10%	14%	2%						

	Decile									
	1	2	3	4	5	6	7	8	9	10
dwt_1242	89%	2%	2%	2%	2%	1%	1%	1%		
mahindas	11%	13%	8%	6%	32%	12%	5%	3%	8%	
jagmesh6	99%		1%							
bcsstk11	94%	1%	1%		5%					
bcsstm11	100%									
comsol	11%	17%	15%	16%	11%	9%	8%	7%	3%	2%
west1505	18%	23%	25%	16%	9%	6%	2%			1%
lshp1561	91%	6%	1%	1%						
bcspwr07	88%	9%	1%							
ex7	57%	43%	1%							
lung1	78%	22%								
bcspwr09	43%	14%	13%	10%	6%	6%	4%	2%	1%	1%
epb0	42%		12%	12%	3%	5%	7%	7%	6%	3%
bcsstk14	65%	35%								
adder_trans_01	27%	13%	11%	10%	9%	8%	8%	7%	4%	3%
ex3	76%	24%								
watt_1	100%									
rajat12	53%	12%	6%	4%	4%	4%	4%	4%	4%	4%
plsk1919				74%			1%	25%		
bcsstk26	62%	38%								
bcsstm26	100%									
rajat02	72%	9%	2%		3%	6%	2%	1%	2%	2%
bcsstk13	62%	28%	5%	1%	2%	1%				
bcsstm13	64%	31%	5%							
west2021	16%	24%	24%	19%	8%	5%	4%			1%
dw1024	99%					1%				
blckhole	91%	3%	1%	3%	2%					
lshp2233	92%	5%	1%	1%						
ex24	94%	6%								
heart2	41%	31%	7%	7%	8%	2%	1%	1%	1%	1%
add20	55%	9%	4%	8%	6%		4%	8%	5%	
rdist3a	98%	2%								
ex10	100%									
orani678	21%	46%	6%	8%	4%	13%	1%			
ex28	100%									
dwt_2680	82%	8%	3%	2%		1%	1%		2%	1%
extr1	5%	12%	11%	12%	12%	10%	11%	10%	12%	5%
meg1	1%	13%	22%	21%	21%	17%		1%	2%	1%
nasa2910	60%	21%	13%	6%						
lhr02	49%	43%	1%	5%	1%					
pde2961	100%									

	Decile									
	1	2	3	4	5	6	7	8	9	10
G50	98%									
laser	11%			22%				67%		
lshp3025	93%	5%	1%	1%						
psmigr_3	27%	14%	13%	12%	8%	9%	7%	4%	4%	2%
swang1	58%		3%	7%	7%	7%	7%	7%	2%	1%
garon1	19%	7%	16%	12%	20%	2%	8%	8%	1%	5%
raefsky2	35%	65%								
bayer05	1%	7%	6%	6%	12%	18%	12%	17%	15%	6%
ex9	100%									
shermanACa	35%	16%	10%	10%	9%	6%	5%	4%	3%	1%
thermal	43%			57%						
lshp3466	93%	4%	1%	1%						
cage9	63%	21%	9%	5%	2%					
heart1	39%	21%	10%	7%	5%	6%	4%	4%	2%	1%
bcsstk24	80%	4%	2%	1%	1%	1%	4%	2%	3%	2%
bcsstm21	100%									
lns_3937	56%		5%		23%		5%		11%	
bcsstk15	61%	39%								
ex12	100%									
poli	62%	4%	4%	5%	5%	4%	4%	4%	4%	2%
sts4098	51%	16%	5%	2%	6%	6%	12%	2%	1%	
lhr04	36%	22%	18%	18%	4%				2%	
rdist1	98%	1%								
struct4	44%	56%								
circuit_2	46%	25%	13%	5%	1%	2%	3%	2%	1%	1%
bcsstk16	100%									
gemat11	13%	22%	19%	19%	16%	7%	2%		1%	
add32	49%		6%	12%	6%		6%	13%	7%	
G58	13%	18%	14%	12%	10%	9%	7%	6%	5%	3%
G59	13%	18%	14%	12%	10%	9%	7%	6%	5%	3%
olm5000	100%									
bcspwr10	35%	12%	11%	12%	10%	7%	6%	4%	3%	1%
hydr1	6%	12%	11%	12%	12%	12%	11%	10%	9%	4%
SiNa	62%	19%	9%	7%	3%					
ex18	100%									
Na5	53%	28%	11%	6%	2%					
meg4	59%	27%	2%	2%	2%	2%	2%	2%	1%	1%
Hamrle2	10%	19%	18%	17%	22%	8%	6%			
shermanACd	32%	16%	9%	9%	7%	8%	7%	6%	4%	2%
Alemdar	14%	16%	16%	11%	11%	16%	6%	5%	4%	2%
raefsky5	89%	10%								

	Decile									
	1	2	3	4	5	6	7	8	9	10
`bayer03`	7%	14%	12%	12%	13%	12%	13%	9%	5%	2%
`jan99jac020`	60%	17%	13%	6%	2%	1%				
`rajat01`	68%	9%	3%	3%	4%	2%	4%	4%	2%	1%
`ex15`	100%									
`G64`	13%	18%	15%	12%	10%	9%	7%	6%	5%	3%
`cell1`	100%									
`goodwin`	99%									
`lhr07`	28%	21%	29%	18%	2%	2%				
`sinc12`	1%	18%	33%	33%	13%	1%				
`rajat13`	57%	9%	4%	4%	4%	4%	4%	4%	4%	4%
`ex40`	100%									
`commanche_dual`	48%	2%	6%	5%	11%	10%	9%	2%	5%	2%
`G65`	100%									
`bcsstk38`	90%	8%	1%					1%		
`Pd`	98%	1%								
`dw4096`	97%					3%				
`benzene`	72%	13%	10%	5%						
`bcsstk33`	83%	17%								
`nd3k`	27%	21%	17%	10%	9%	7%	3%	3%	2%	1%
`mark3jac020`	87%	12%								
`nemeth02`	100%									
`nemeth16`	100%									
`nemeth19`	100%									
`nemeth21`	100%									
`nemeth26`	100%									
`coater2`	99%	1%								
`fv2`	100%									
`shuttle_eddy`	85%	1%	2%						3%	10%
`pkustk02`	68%	24%	4%	1%	1%	1%	1%	1%		
`igbt3`	28%	3%	4%	38%	3%	2%	2%	18%	1%	1%
`k3plates`	100%									
`m3plates`	100%									
`coupled`	32%	13%	12%	11%	9%	8%	6%	5%	3%	2%
`cage10`	68%	18%	8%	4%	1%					
`t2dah_a`	100%									
`sinc15`	2%	18%	33%	35%	12%	1%				
`sme3Da`	11%	16%	15%	17%	11%	9%	8%	7%	3%	2%
`stokes64`	53%			23%				23%		
`skirt`	98%	2%								
`tuma2`	40%	2%	7%	7%	9%	17%	13%	5%		
`poisson3Da`	17%	14%	13%	11%	12%	10%	6%	7%	6%	3%

	Decile									
	1	2	3	4	5	6	7	8	9	10
`Pres_Poisson`	89%	7%	2%			1%		1%		
`rajat07`	98%	1%								
`powersim`	65%	14%	12%	4%	1%	1%	2%	1%		
`sinc18`	2%	18%	32%	36%	11%	1%				
`pds10`	76%	2%	4%	2%	2%	2%	2%	3%	3%	2%
`pkustk07`	65%	14%	8%	2%	3%	1%	2%	1%	1%	1%
`gyro_k`	90%	8%	2%							
`gyro_m`	90%	8%	2%							
`nd6k`	29%	22%	19%	11%	8%	5%	3%	2%	2%	
`nmos3`	30%	2%	2%	40%	3%	1%	2%	20%		
`bodyy6`	96%	1%			1%		1%			
`t3dl_a`	98%	2%								
`t3dl_e`	100%									
`ns3Da`	9%	16%	15%	15%	11%	10%	9%	6%	4%	3%
`raefsky3`	100%									
`pkustk01`	87%	7%	3%	1%	1%					
`pkustk08`	67%	16%	5%	3%	3%	1%	1%	1%	1%	1%
`rim`	100%									
`tuma1`	40%	2%	7%	7%	9%	17%	13%	6%		
`crystm03`	100%									
`dtoc`					86%		4%	7%	4%	
`mult_dcop_01`	31%	13%	10%	9%	8%	7%	7%	6%	4%	3%
`bcsstm37`	100%									
`brainpc2`	14%	12%	12%	14%	6%	34%			8%	
`3D_28984_Tetra`	74%	1%	24%							
`bloweya`	34%	2%	2%	5%	18%	18%	18%	3%		
`aug2dc`				2%	30%	31%	31%	6%		
`rajat10`	98%	1%								
`bcsstm35`	100%									
`Zhao1`	20%		13%	26%	1%	1%	25%	1%	13%	1%
`pkustk09`	89%	6%	2%	1%	1%					
`lhr34`	33%	59%	7%	1%						
`nd12k`	31%	25%	16%	11%	8%	5%	2%	1%	1%	
`onetone1`	76%	16%	7%							
`wathen120`	100%									
`pwt`	81%	1%	3%	4%	3%	3%	3%	2%	1%	
`rajat15`	40%	11%	11%	8%	6%	5%	7%	6%	5%	2%
`finance256`	61%			1%	12%	13%	13%	1%		
`cage11`	73%	15%	8%	3%	1%					
`torsion1`	100%									
`av41092`	15%	33%	29%	10%	7%	3%	1%	1%		

	Decile									
	1	2	3	4	5	6	7	8	9	10
`jan99jac120`	91%	7%	1%							
`sme3Dc`	11%	17%	15%	17%	11%	9%	8%	7%	3%	2%
`pkustk06`	86%	8%	3%	1%	1%					
`3dtube`	93%	7%								
`bcsstk39`	100%									
`bcsstm39`	100%									
`rma10`	70%				17%	12%				
`gridgena`	100%									
`stokes128`	53%			23%				23%		
`ibm_matrix_2`	72%	27%								
`ct20stif`	84%	8%	2%	3%	2%					
`g7jac180`	54%	22%	21%	3%						
`struct3`	100%									
`copter2`	49%	22%	2%	3%	8%	7%	7%	1%	1%	1%
`pkustk04`	80%	11%	5%		1%	1%	1%			
`bayer01`	5%	11%	13%	11%	11%	10%	11%	10%	10%	5%
`g7jac200`	55%	24%	20%							
`a5esindl`	14%			24%		24%	3%	13%	13%	9%
`blockqp1`	23%	21%	21%	18%	6%	6%	6%	1%		
`qa8fk`	100%									
`lhr71`	62%	38%								
`nd24k`	34%	30%	18%	9%	5%	2%	1%			
`ncvxqp3`	19%	18%	17%	12%	12%	4%	4%	13%	2%	1%
`t3dh_e`	98%	2%								
`a2nnsnsl`	13%			23%		23%	6%	13%	10%	12%
`pkustk10`	93%	3%	2%			1%				
`poisson3Db`	18%	19%	14%	10%	7%	7%	8%	6%	6%	4%
`ncvxqp7`	18%	19%	15%	12%	7%	5%	18%	3%	2%	1%
`boyd1`	12%	9%	9%	9%	9%	8%	8%	10%	11%	9%
`tandem_dual`	89%	2%	2%	6%						
`pkustk12`	81%	10%	5%	1%	1%	1%				
`pkustk13`	90%	5%	2%	1%	1%					1%
`ford2`	84%	3%	3%	2%	1%	1%	1%	1%	2%	1%
`matrix_9`	100%									
`hcircuit`	57%		7%	11%		3%	7%	8%	3%	3%
`lung2`	100%									
`barrier2-1`	28%	4%	4%	38%	3%	2%	2%	18%	1%	1%
`torso2`	99%									
`torso1`	27%	19%	11%	6%	6%	7%	8%	7%	5%	3%
`twotone`	79%	18%	3%							
`matrix-new_3`	99%									

	Decile									
	1	2	3	4	5	6	7	8	9	10
`pkustk14`	90%	6%	1%	1%	1%					
`para-6`	30%	3%	3%	39%	2%	2%	2%	19%	1%	
`gearbox`	98%	1%	1%							
`para-10`	30%	3%	3%	39%	2%	2%	2%	19%	1%	
`xenon2`	94%	6%								
`scircuit`	72%	5%	4%	6%	4%	3%	2%	2%	1%	1%
`cont-300`	9%				36%	31%	5%	18%		
`ohne2`	32%	1%	1%	42%	1%	1%	1%	21%		
`stomach`	99%	1%								
`pwtk`	97%							1%	2%	
`torso3`	97%			1%		1%	1%			
`Ga41As41H72`	76%	24%								
`Stanford`	14%	19%	15%	13%	11%	9%	8%	6%	4%	2%
`rajat24`	81%	6%	4%	3%	2%	2%	1%			
`language`	47%	36%	4%	2%	2%	3%	3%	2%	1%	
`rajat21`	82%	5%	3%	3%	2%	2%	1%			
`cage13`	78%	13%	6%	2%	1%					
`boyd2`	19%				28%	8%	23%	6%	16%	
`af_shell1`	100%									
`pre2`	93%	2%	4%							
`Stanford_Berkeley`	92%	2%	2%	1%	1%		1%			

Appendix D

SpMV Performance on the Penium 4

Here we present information about SpMV performance for each of the matrices in our test suite, as well as which ones fall into the categories small, medium, and large, on the Pentium 4. The matrices are sorted in order of increasing problem size. Symmetric and nonsymmetric performance are also compared for symmetric matrices. All performance numbers are in MFLOP/s. Blank values in the tuned columns indicate that OSKI did not tune SpMV for that particular matrix. Raw MFLOP rates (counting operations performed on explicitly stored zero entries that were introduced during blocking) are given for the tuned numbers so that a proper comparison with synthetic matrices, which have no filled in zero entries, can be made. For an MFLOP rate that counts only operations performed on nonzero entries, divide by the fill ratio.

D.1 Small Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
bcsstm19	155	293				
shl_0	207	499				
gre_512	260	365				
bcsstm11	185	290				
dw256A	310	583				
fs_680_3	283	393				
bcsstm26	186	292				
nos6	371	583				
685_bus	306	592				
pores_3	333	630	2×1	1.06	363	806
qh882	279	417				
lshp_577	439	638				
west0989	241	360				
nnc666	340	456				
saylr3	245	532				
fs_541_1	378	498				
sherman4	260	498				
tub1000	328	367				
b_dyn	249	525				
pde900	347	572				
nos7	416	444				
bp_800	294	576				
cdde1	310	585				
lshp_778	467	623				
steam2	511	647	4×4	1	873	1025
orsirr_2	336	626				
west1505	286	366				
jpwh_991	287	587				
fpga_dcop_01	320	563				
jagmesh1	468	632				
bcsstm21	194	292				
can_715	480	638				
bcspwr07	269	526				
bcsstk19	460	640				
can_634	489	659				
lshp1009	467	642				
dwt_869	468	506				
bcspwr09	324	362				
bfwa782	425	652				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
dwt_918	468	503				
mahindas	354	433				
lung1	305	374				
west2021	243	362				
epb0	320	535				
jagmesh6	482	642				
pores_2	449	652	2×1	1	427	832
can_838	534	562				
rotor2	443	595	3×3	1.32	604	1343
plsk1919	241	592				
dwt_1242	477	664				
laser	353	486				
cage8	492	655				
dw1024	280	585				
poli	171	419				
lshp1561	481	638				
rajat02	384	597				
watt_1	390	612				
extr1	263	519				
rajat12	338	614				
G50	407	540				
add20	322	563				
adder_trans_01	327	607				
nos3	555	627	2×2	1.02	764	1093
blckhole	461	636				
pde2961	325	585				
lshp2233	481	634				
dwt_992	557	612				
m3plates	125	257				
Si2	576	636				
ex21	481	665	2×1	1.1	638	935
Pd	144	294				
bcsstm13	454	630				
lshp3025	474	539				
bayer05	344	413				
swang1	328	525				
sherman2	480	593				
add32	260	448				
bcsstm34	636	647	6×2	1.26	877	1519
olm5000	299	442	1×2	1.25	328	578
mcfe	423	629				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
ex25	504	587	2×1	1.06	631	914
circuit_2	294	511				
bcspwr10	308	453				
lshp3466	451	578				
dwt_2680	443	526				
hydr1	303	475				
Hamrle2	252	417				
shermanACa	281	535				
lns_3937	344	527				
meg4	259	421				
bcsstk26	484	576				
bcsstk11	517	543	3×3	1.06	819	1121
bayer03	254	410				
cell1	268	379				
gemat11	265	439	2×1	1.31	350	642
ex27	404	499				
G1	514	498				
commanche_dual	336	374				
lhr02	337	479				
G65	333	368				
jan99jac020	235	382				
t3dl_e	128	126				
bcsstm37	81	115				

D.2 Medium Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
bcsstk27	440	448	3×2	1.27	585	676
comsol	349	418				
ex7	339	480				
bcsstk14	423	391	6×2	1.17	594	840
ex3	340	443	2×1	1.15	407	561
bcsstk13	412	417				
ex24	317	453				
heart2	366	427	2×6	1.29	538	634
rdist3a	313	412	1×2	1.21	362	537
ex10	328	422				
orani678	334	423				
ex28	326	400	2×2	1.11	429	585
meg1	313	407	1×3	1.37	385	537
nasa2910	423	401	1×5	1.23	529	514
psmigr_3	353	437				
garon1	320	412				
raefsky2	355	419	2×2	1.02	447	522
ex9	327	390				
thermal	313	394				
cage9	310	445				
heart1	364	429	1×3	1.15	473	513
bcsstk24	415	413	2×2	1.03	515	556
bcsstk15	401	402				
ex12	307	396				
sts4098	371	391				
lhr04	293	381				
rdist1	310	390				
struct4	417	407				
bcsstk16	426	419	3×3	1.02	561	605
G58	347	395				
G59	349	391				
SiNa	400	392				
ex18	267	372				
Na5	409	410				
shermanACd	260	399				
Alemdar	352	398				
raefsky5	316	373	2×2	1.12	399	486
rajat01	257	356				
ex15	273	367				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
G64	341	370				
goodwin	327	397				
lhr07	299	363				
sinc12	316	405				
rajat13	246	331				
ex40	351	403				
bcsstk38	413	405				
dw4096	178	351				
benzene	390	381				
bcsstk33	429	420				
nd3k	443	437	3×3	1.11	604	627
mark3jac020	214	301				
nemeth02	412	393				
nemeth16	429	398	4×1	1.22	559	568
nemeth19	439	408	4×4	1.27	634	662
nemeth21	446	416	8×1	1.19	610	620
nemeth26	450	419	8×1	1.21	619	599
coater2	293	370				
fv2	259	324				
shuttle_eddy	314	329				
pkustk02	433	384	6×2	1	584	607
igbt3	270	336				
k3plates	323	400	2×1	1.14	395	457
coupled	297	346				
cage10	323	343				
t2dah_a	355	362				
sinc15	325	379				
sme3Da	329	398				
stokes64	313	322				
skirt	343	356				
tuma2	244	278				
poisson3Da	287	369				
Pres_Poisson	415	395				
rajat07	233	256				
powersim	176	254				
sinc18	327	373				
pds10	278	309				
pkustk07	441	384	3×3	1	588	591
gyro_m	362	365				
gyro_k	422	385	3×3	1	552	555
nd6k	440	427	3×3	1.12	597	623

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
nmos3	282	308				
bodyy6	272	306				
t3dl_a	380	360				
ns3Da	313	380				
raefsky3	351	408	4×4	1	522	631
pkustk01	406	363	3×3	1	526	553
pkustk08	439	344	3×3	1	584	597
rim	343	390				
tuma1	211	218				
crystm03	376	371				
dtoc	185	212				
mult_dcop_01	167	263				
brainpc2	247	218				
3D_28984_Tetra	288	296	3×3	1.03	371	464
bloweya	235	219				
aug2dc	180	203				
rajat10	216	243				
bcsstm35	80	108				
Zhao1	181	192				
pkustk09	410	316	6×2	1	540	572
lhr34	312	323				
nd12k	436	216	3×3	1.12	584	603
onetone1	243	276				
wathen120	343	353				
pwt	279	318				
rajat15	190	146				
finance256	259	248				
cage11	247	252				
torsion1	246	265				
av41092	308	176	2×1	1.05	371	230
jan99jac120	194	260				
sme3Dc	157	146				
pkustk06	417	280	6×2	1	559	564
3dtube	408	355	3×3	1.02	546	593
bcsstm39	91	101				
bcsstk39	417	400				
rma10	336	353				
gridgena	279	322				
stokes128	309	255				
ibm_matrix_2	273	291	3×3	1.03	335	452
ct20stif	407	216	2×1	1.1	472	289

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
`g7jac180`	256	208				
`struct3`	376	361				
`copter2`	263	155				
`pkustk04`	423	203	3×3	1	546	562
`bayer01`	203	143				
`g7jac200`	253	206				
`a5esindl`	182	146				
`blockqp1`	307	99				

D.3 Large Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
qa8fk	381	365				
lhr71	294	271				
nd24k	430	116	3×3	1.12	579	490
ncvxqp3	152	87				
t3dh_e	407	315				
a2nnsnsl	184	116				
pkustk10	416	239	6×2	1	555	560
poisson3Db	99	65				
ncvxqp7	148	74				
boyd1	274	72				
tandem_dual	179	238				
pkustk12	419	131	3×1	1.1	512	237
pkustk13	419	170	3×1	1.1	518	292
ford2	211	228				
matrix_9	236	302	3×3	1.01	326	503
hcircuit	148	111				
lung2	162	207				
barrier2-1	188	87				
torso2	195	263				
torso1	282	55	2×3	1.18	396	316
twotone	198	147				
matrix-new_3	232	231	3×3	1.03	313	410
pkustk14	426	128				
para-6	180	73				
gearbox	422	201	3×3	1	531	586
para-10	182	87				
xenon2	235	144	3×3	1.06	345	486
scircuit	119	137				
cont-300	237	105				
ohne2	242	80				
stomach	204	150				
pwtk	420	135	3×1	1.11	513	214
torso3	201	117				
Ga41As41H72	344	65				
Stanford	30	35				
rajat24	132	70				
language	66	41				
rajat21	136	61				
cage13	183	43				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
boyd2	162	41				
af_shell1	390	57	5×1	1	473	163
pre2	186	44				
Stanford_Berkeley	193	45				

D.4 Symmetric Matrices

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
lshp_577	593			
bcsstm34	1160	1×3	1.17	1360
can_634	805			
nos6	608			
685_bus	480			
can_715	792			
nos7	712			
Si2	826			
lshp_778	711			
G1	929			
bcsstk19	725	1×2	1.38	998
bcsstm19	118			
can_838	701			
dwt_869	670			
dwt_918	750			
jagmesh1	631			
nos3	822	2×2	1.08	888
dwt_992	880	1×2	1.34	1181
lshp1009	671	1×2	1.58	1061
bcsstk27	942	1×3	1.16	1092
dwt_1242	626			
jagmesh6	611			
bcsstk11	1116	3×3	1.15	1278
bcsstm11	135			
lshp1561	663			
bcspwr07	341			
bcspwr09	396			
bcsstk14	881	2×2	1.15	1011
bcsstk26	800			
bcsstm26	123			
rajat02	546			
bcsstk13	818			
bcsstm13	699			
blckhole	667			
lshp2233	780			
dwt_2680	653			
nasa2910	843	5×1	1.26	1066
G50	447			
laser	572			

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
lshp3025	655			
lshp3466	630			
bcsstk24	1032	2×2	1.06	1090
bcsstm21	124			
bcsstk15	775			
sts4098	651			
struct4	848			
bcsstk16	843	1×3	1.04	878
G58	597			
G59	597			
bcspwr10	407			
SiNa	717			
Na5	779			
Alemdar	918			
G64	602			
commanche_dual	384			
G65	571			
bcsstk38	793			
benzene	672			
bcsstk33	845	2×2	1.32	1114
nd3k	895	5×1	1.25	1123
nemeth02	814			
nemeth16	854	5×1	1.31	1120
nemeth19	972	5×1	1.26	1229
nemeth21	899	5×1	1.22	1099
nemeth26	988	5×1	1.24	1222
shuttle_eddy	533			
pkustk02	1044	2×2	1.01	1058
m3plates	77	1×2	1.67	129
coupled	491			
t2dah_a	646			
stokes64	570			
skirt	613			
tuma2	421			
Pres_Poisson	928			
rajat07	362			
pds10	518			
pkustk07	889	1×3	1.01	901
gyro_k	815	1×3	1.03	843
gyro_m	638			
nd6k	903	5×1	1.26	1139

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
bodyy6	499			
t3dl_a	719			
t3dl_e	90			
pkustk01	786	1×3	1.04	820
pkustk08	982	1×3	1.01	996
tuma1	375			
crystm03	709			
dtoc	386			
bcsstm37	66	1×2	1.81	120
brainpc2	482			
bloweya	500			
aug2dc	367	1×2	2.5	920
rajat10	365			
bcsstm35	84			
pkustk09	832	2×2	1.02	850
nd12k	930	5×1	1.26	1174
wathen120	729			
pwt	508			
finance256	454			
torsion1	483			
pkustk06	812	2×2	1.02	826
3dtube	800	1×3	1.04	831
bcsstk39	800			
bcsstm39	77			
gridgena	477			
stokes128	587			
ct20stif	785	2×2	1.23	963
struct3	711			
copter2	443			
pkustk04	825	1×3	1.03	846
a5esindl	284			
blockqp1	490			
qa8fk	740			
nd24k	886	5×1	1.26	1119
ncvxqp3	238			
t3dh_e	762			
a2nnsnsl	274			
pkustk10	838	2×2	1.02	853
ncvxqp7	211			
boyd1	490			
tandem_dual	238			

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
pkustk12	660	1×3	1.16	763
pkustk13	637	3×1	1.12	712
ford2	300			
pkustk14	643			
gearbox	656	1×3	1.03	679
cont-300	379			
pwtk	664	3×1	1.15	760
Ga41As41H72	539			
boyd2	233			
af_shell1	812	5×1	1.11	903

Appendix E

SpMV Performance on the Itanium 2

Here we present information about SpMV performance for each of the matrices in our test suite, as well as which ones fall into the categories small, medium, and large, on the Itanium 2. The matrices are sorted in order of increasing problem size. Symmetric and nonsymmetric performance are also compared for symmetric matrices. All performance numbers are in MFLOP/s. Blank values in the tuned columns indicate that OSKI did not tune SpMV for that particular matrix. Raw MFLOP rates (counting operations performed on explicitly stored zero entries that were introduced during blocking) are given for the tuned numbers so that a proper comparison with synthetic matrices, which have no filled in zero entries, can be made. For an MFLOP rate that counts only operations performed on nonzero entries, divide by the fill ratio.

E.1 Small Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
bcsstm19	35	55				
shl_0	91	142				
gre_512	151	177				
bcsstm11	37	55				
dw256A	164	161				
fs_680_3	144	177				
bcsstm26	37	55				
nos6	168	161				
685_bus	166	161				
pores_3	199	255	2×1	1.06	192	444
qh882	145	175				
lshp_577	206	250				
west0989	139	174				
nnc666	194	229				
saylr3	134	173				
fs_541_1	225	269				
sherman4	120	140				
tub1000	152	174				
b_dyn	152	173				
pde900	175	158				
nos7	203	225				
bp_800	180	226				
cdde1	175	158				
lshp_778	213	246				
steam2	373	236	4×2	1	598	968
orsirr_2	215	244				
west1505	145	171				
jpwh_991	197	222				
fpga_dcop_01	174	157				
jagmesh1	215	244				
bcsstm21	38	55				
can_715	255	232				
bcspwr07	147	171				
bcsstk19	244	261				
can_634	280	307				
lshp1009	219	243				
dwt_869	245	260				
bcspwr09	152	171				
bfwa782	261	295				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
dwt_918	240	261				
mahindas	195	222				
lung1	172	171				
west2021	149	171				
epb0	169	171				
jagmesh6	217	244				
pores_2	240	260	2×1	1	245	447
can_838	292	319				
rotor2	309	341	6×3	1.61	589	1448
plsk1919	131	157				
dwt_1242	250	261				
laser	132	138				
cage8	282	306				
dw1024	186	157				
poli	77	99				
lshp1561	225	244				
rajat02	202	223				
watt_1	209	179				
extr1	162	171				
rajat12	225	221				
G50	163	169				
add20	234	156				
adder_trans_01	247	259				
nos3	339	321	2×2	1.02	391	658
blckhole	229	243				
pde2961	187	157				
lshp2233	226	243				
dwt_992	341	322				
m3plates	32	54				
Si2	376	402				
ex21	398	390	2×3	1.42	556	736
Pd	68	99				
bcsstm13	259	305				
lshp3025	228	242				
bayer05	250	220				
swang1	222	239				
sherman2	362	348				
add32	185	170				
bcsstm34	434	426	3×2	1.24	757	876
olm5000	165	170				
mcfe	413	439				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
ex25	404	387	2×3	1.39	560	718
circuit_2	182	156				
bcspwr10	167	170				
lshp3466	231	240				
dwt_2680	271	228				
hydr1	175	170				
Hamrle2	155	170				
shermanACa	235	238				
lns_3937	216	217				
meg4	251	170				
bcsstk26	338	351				
bcsstk11	384	394	3×1	1.04	571	763
bayer03	254	169				
cell1	190	169				
gemat11	228	241	2×1	1.31	281	408
ex27	437	441				
G1	445	462				
commanche_dual	165	169				
lhr02	309	314				
G65	165	169				
jan99jac020	207	156				
t3dl_e	39	55				
bcsstm37	34	55				
cage9	302	313				
ex7	418	363				
Alemdar	305	240				
dw4096	194	156				
rajat01	219	220				
ex24	375	339				
bcsstm35	39	55				
ex3	406	376	2×1	1.15	569	661
rajat13	222	220				
ex10	376	394				
bcsstk27	447	439	3×2	1.27	835	813
shermanACd	259	229				
meg1	364	379	2×2	1.4	517	650
mark3jac020	216	214				
tuma2	156	170				
G58	301	310				
G59	302	310				
rdist3a	392	407	2×2	1.37	564	696

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
bcsstk14	417	429	6×4	1.42	1067	1493
thermal	358	369				
sts4098	347	360				
ex18	304	309				
rajat07	171	169				
ex28	391	411	2×2	1.11	532	717
powersim	171	170				
ex12	362	376				
lhr04	353	372				
bcsstk13	414	420				
garon1	386	389				
G64	297	313				
orani678	389	420	2×1	1.23	604	683
bcsstm39	38	54				
dtoc	118	137				
fv2	261	229				
rdist1	360	385	2×2	1.41	527	679
comsol	429	424				
ex9	381	409				
ex15	319	331				
coupled	253	227				
aug2dc	119	137				
tuma1	156	169				
shuttle_eddy	273	286				
bcsstk15	374	397				
igbt3	331	307				
stokes64	282	296				
bodyy6	226	236				
rajat10	169	169				
lhr07	319	333				
cage10	297	272				
bcsstk24	368	371	2×2	1.03	585	760
pds10	253	228				
raefsky5	337	365	2×2	1.12	508	685
nasa2910	372	383	5×5	1.22	870	1290
bloweya	183	154				
t2dah_a	311	318				

E.2 Medium Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
heart2	377	382	3×6	1.34	878	891
psmigr_3	373	376				
raefsky2	364	375	2×2	1.02	671	753
heart1	380	380	2×3	1.26	825	831
struct4	359	376				
bcsstk16	358	370	3×1	1.01	628	629
SiNa	352	362				
Na5	358	365				
goodwin	354	343				
sinc12	350	359				
ex40	360	370				
bcsstk38	353	362	3×1	1.24	598	654
benzene	347	346				
bcsstk33	363	370	2×2	1.31	667	745
nd3k	380	381	3×2	1.15	808	822
nemeth02	354	362	3×2	1.4	661	700
nemeth16	363	370	3×2	1.3	701	758
nemeth19	369	373	3×2	1.21	739	777
nemeth21	374	377	8×1	1.19	791	766
nemeth26	377	379	8×1	1.21	824	770
coater2	327	346				
pkustk02	366	371	6×6	1	744	925
k3plates	349	357	2×2	1.28	540	689
sinc15	360	372				
sme3Da	363	369				
skirt	308	335				
poisson3Da	336	345				
Pres_Poisson	357	370	2×2	1.35	618	726
sinc18	359	361				
pkustk07	375	368	3×1	1	688	648
gyro_m	320	334				
gyro_k	362	367	3×1	1	594	603
nd6k	379	377	3×2	1.16	789	808
nmos3	327	273				
t3dl_a	339	335				
ns3Da	364	366				
raefsky3	365	370	8×8	1	698	940
pkustk01	356	357	3×2	1.12	605	766
pkustk08	374	371	3×1	1	685	659

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
rim	356	356				
crystm03	335	344				
mult_dcop_01	226	253				
brainpc2	208	215				
3D_28984_Tetra	323	275	3×1	1.02	355	427
Zhao1	182	154				
pkustk09	356	358	6×6	1	635	912
lhr34	312	316				
nd12k	379	372	3×2	1.16	774	799
onetone1	243	226				
wathen120	306	336				
pwt	250	227				
rajat15	271	286				
finance256	235	251				
cage11	295	302				
torsion1	184	155				
av41092	339	341	2×1	1.05	479	542
jan99jac120	207	213				
sme3Dc	359	353				
pkustk06	362	360	6×6	1	670	897
3dtube	366	367	3×2	1.14	694	787
bcsstk39	358	359	2×2	1.35	599	710
rma10	360	356	2×2	1.29	597	705
gridgena	267	278				
stokes128	273	275				
ibm_matrix_2	324	275	3×1	1.02	344	419
ct20stif	359	354	2×2	1.21	579	705
g7jac180	278	290				
struct3	331	344				
copter2	288	294				
pkustk04	366	352	3×1	1	599	603
bayer01	176	153				
g7jac200	278	290				
a5esindl	161	168				
blockqp1	255	268				
qa8fk	339	334				
lhr71	312	314				
nd24k	378	355	3×2	1.16	756	791
ncvxqp3	212	224				
t3dh_e	360	359				
a2nnsnsl	162	167				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
pkustk10	360	349	6×6	1	646	913
poisson3Db	315	313				
ncvxqp7	206	223				
boyd1	263	251				
tandem_dual	180	154				
pkustk12	366	336	3×2	1.26	686	752
pkustk13	364	343	3×1	1.1	602	577
ford2	190	154				
matrix_9	323	293	3×1	1	341	537
hcircuit	171	152				
lung2	171	166				
barrier2-1	328	295				
torso2	249	225				
torso1	359	288	2×3	1.18	610	734
twotone	250	269				
matrix-new_3	322	231	3×1	1.02	342	264
pkustk14	370	314	2×2	1.37	692	691
para-6	327	276				
gearbox	364	343	3×1	1	580	570
para-10	328	250				
xenon2	335	319	3×1	1.04	415	591
scircuit	186	211				
cont-300	191	152				
ohne2	356	315				
stomach	291	295				
pwtk	361	336	3×2	1.24	654	714
torso3	305	285				
Ga41As41H72	353	309				
Stanford	101	110				
rajat24	178	152				

E.3 Large Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
language	111	127				
rajat21	161	152				
cage13	295	170				
boyd2	127	128				
af_shell1	351	301	5×5	1	557	857
pre2	238	179				
Stanford_Berkeley	233	223				

E.4 Symmetric Matrices

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
lshp_577	319	4×2	3.21	1022
bcsstm34	966	6×6	1.43	1382
can_634	477	3×2	2.8	1339
nos6	230	3×2	2.94	674
685_bus	259			
can_715	418	3×2	2.44	1020
nos7	294	3×3	2.82	827
Si2	748			
lshp_778	325	3×2	2.72	884
G1	1041			
bcsstk19	382	4×3	2.75	1048
bcsstm19	69			
can_838	511	3×2	2.93	1496
dwt_869	379	3×2	2.31	875
dwt_918	394	3×2	2.62	1030
jagmesh1	319	5×3	4.49	1429
nos3	616	4×2	1.5	922
dwt_992	626	3×2	2.11	1321
lshp1009	328	5×5	5.54	1817
bcsstk27	1008	5×3	1.49	1506
dwt_1242	394	3×2	2.5	984
jagmesh6	320	3×2	2.58	825
bcsstk11	760	3×2	1.3	988
bcsstm11	75			
lshp1561	333	3×2	2.71	903
bcspwr07	223			
bcspwr09	226			
bcsstk14	913	6×3	1.3	1190
bcsstk26	605	4×2	1.92	1161
bcsstm26	67			
rajat02	361	3×2	3.41	1231
bcsstk13	973	3×2	1.86	1810
bcsstm13	646			
blckhole	332	3×2	2.73	907
lshp2233	335	3×2	2.76	927
dwt_2680	433	3×2	2.73	1184
nasa2910	1004	5×5	1.29	1291
G50	234	3×2	4.5	1053
laser	272			

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
lshp3025	336	3×2	2.66	894
lshp3466	335	3×2	2.74	920
bcsstk24	998	4×2	1.25	1247
bcsstm21	79			
bcsstk15	840	6×3	1.89	1589
sts4098	653	3×2	2.74	1789
struct4	949	3×2	1.86	1766
bcsstk16	1022	3×3	1.05	1075
G58	627			
G59	628			
bcspwr10	236			
SiNa	881			
Na5	1001	3×2	2.38	2377
Alemdar	818			
G64	629			
commanche_dual	243			
G65	231	3×2	4.5	1040
bcsstk38	931	3×2	1.45	1354
benzene	806			
bcsstk33	1051	3×2	1.53	1609
nd3k	1359	6×3	1.24	1691
nemeth02	958	4×2	1.52	1453
nemeth16	1094	4×2	1.36	1492
nemeth19	1192	8×1	1.35	1612
nemeth21	1265	8×1	1.29	1633
nemeth26	1279	8×1	1.29	1650
shuttle_eddy	439	3×2	2.25	989
pkustk02	1111	6×6	1.07	1184
m3plates	65			
coupled	425			
t2dah_a	568	3×2	2.42	1374
stokes64	596	3×2	2.5	1491
skirt	565	3×2	2.52	1422
tuma2	296	3×2	3.77	1118
Pres_Poisson	969	4×2	1.75	1695
rajat07	254			
pds10	383			
pkustk07	1170	3×3	1.01	1186
gyro_k	947	3×3	1.03	979
gyro_m	591	3×3	3	1773
nd6k	1259	6×3	1.25	1578

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
`bodyy6`	294	3×2	2.52	741
`t3dl_a`	661	3×2	2.99	1976
`t3dl_e`	65	2×2	2	130
`pkustk01`	867	3×3	1.04	905
`pkustk08`	1167	3×3	1.01	1183
`tuma1`	271	3×2	3.79	1027
`crystm03`	635	3×3	3	1905
`dtoc`	241			
`bcsstm37`	70			
`brainpc2`	428			
`bloweya`	368			
`aug2dc`	200	3×2	5.54	1111
`rajat10`	219	2×2	2.6	569
`bcsstm35`	87			
`pkustk09`	882	6×6	1.1	975
`nd12k`	1245	6×3	1.25	1562
`wathen120`	490	3×2	2	983
`pwt`	369	3×2	2.93	1081
`finance256`	348	3×2	3.4	1182
`torsion1`	208	2×2	2	417
`pkustk06`	953	6×6	1.08	1032
`3dtube`	1001	3×2	1.17	1172
`bcsstk39`	858	3×2	1.58	1360
`bcsstm39`	55	2×2	2	110
`gridgena`	380	4×2	2.08	791
`stokes128`	462	3×2	2.49	1148
`ct20stif`	903	4×2	1.5	1360
`struct3`	602	5×5	2.16	1299
`copter2`	421			
`pkustk04`	1036	3×3	1.03	1063
`a5esindl`	229	2×2	2.35	536
`blockqp1`	471	2×4	1.88	887
`qa8fk`	654	3×2	2.11	1380
`nd24k`	1225	6×3	1.25	1536
`ncvxqp3`	293			
`t3dh_e`	909			
`a2nnsnsl`	230	2×2	2.39	550
`pkustk10`	921	6×6	1.09	1006
`ncvxqp7`	297			
`boyd1`	487	4×4	2.02	985
`tandem_dual`	199			

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
pkustk12	1046	3×2	1.3	1362
pkustk13	998	3×3	1.26	1261
ford2	231	3×2	3.33	770
pkustk14	1078	3×2	1.69	1821
gearbox	952	3×3	1.04	986
cont-300	337			
pwtk	916	3×2	1.27	1165
Ga41As41H72	837			
boyd2	176			
af_shell1	759	5×5	1.11	844

Appendix F

SpMV Performance on the Opteron

Here we present information about SpMV performance for each of the matrices in our test suite, as well as which ones fall into the categories small, medium, and large, on the Opteron. The matrices are sorted in order of increasing problem size. Symmetric and nonsymmetric performance are also compared for symmetric matrices. All performance numbers are in MFLOP/s. Blank values in the tuned columns indicate that OSKI did not tune SpMV for that particular matrix. Raw MFLOP rates (counting operations performed on explicitly stored zero entries that were introduced during blocking) are given for the tuned numbers so that a proper comparison with synthetic matrices, which have no filled in zero entries, can be made. For an MFLOP rate that counts only operations performed on nonzero entries, divide by the fill ratio.

F.1 Small Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
bcsstm19	202	305				
shl_0	203	429				
gre_512	265	455				
bcsstm11	209	304				
dw256A	332	468				
fs_680_3	347	454				
bcsstm26	198	304				
nos6	335	468				
685_bus	221	470				
pores_3	342	489	2×1	1.06	470	879
qh882	280	455				
lshp_577	381	485				
west0989	230	449				
nnc666	289	480				
saylr3	242	446				
fs_541_1	338	495				
sherman4	288	428				
tub1000	346	444				
b_dyn	216	428				
pde900	331	450				
nos7	326	478				
bp_800	232	447				
cdde1	340	432				
lshp_778	359	433				
steam2	354	320	2×4	1.09	721	1182
orsirr_2	314	408				
west1505	208	376				
jpwh_991	243	407				
fpga_dcop_01	267	390				
jagmesh1	365	398				
bcsstm21	169	216				
can_715	320	289				
bcspwr07	178	372				
bcsstk19	269	419				
can_634	300	301				
lshp1009	350	407				
dwt_869	321	410				
bcspwr09	242	370				
bfwa782	298	287				
dwt_918	303	412				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
mahindas	294	387				
lung1	286	361				
west2021	211	370				
epb0	288	366				
jagmesh6	337	405				
pores_2	309	410	2×1	1	448	614
can_838	347	300				
rotor2	313	320	3×3	1.32	648	955
plsk1919	266	383				
dwt_1242	297	410				
laser	279	342				
cage8	312	293				
dw1024	308	383				
poli	172	304				
lshp1561	350	404				
rajat02	247	394				
watt_1	302	392				
extr1	210	363				
rajat12	296	403				
G50	309	372				
add20	283	376				
adder_trans_01	270	414				
nos3	330	330	2×2	1.02	662	702
blckhole	355	392				
pde2961	320	375				
lshp2233	363	404				
dwt_992	337	338				
m3plates	109	209				
Si2	366	364				
ex21	383	381	4×1	1.34	794	800
Pd	144	293				
bcsstm13	338	297				
lshp3025	366	406				
bayer05	337	368				
swang1	330	388				
sherman2	362	357				
add32	228	347				
bcsstm34	386	401	2×2	1.2	733	799
olm5000	324	356	1×2	1.25	382	482
mcfe	373	374				
ex25	384	381	2×3	1.39	748	825

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
circuit_2	310	376				
bcspwr10	242	358				
lshp3466	357	405				
dwt_2680	306	283				
hydr1	227	354				
Hamrle2	290	340				
shermanACa	255	388				
lns_3937	320	379				
meg4	293	366				
bcsstk26	329	325	2×4	1.86	781	952
bcsstk11	369	363	3×3	1.06	826	998
bayer03	306	350				
cell1	329	371				
gemat11	257	393	2×1	1.31	429	566
ex27	409	398				
G1	412	400				
commanche_dual	302	350				
lhr02	350	302				
G65	316	364				
jan99jac020	263	378				
t3dl_e	71	154				
bcsstm37	62	189				
cage9	316	303				
ex7	389	380				
Alemdar	240	378				
dw4096	308	368				
rajat01	252	390				
ex24	357	356				
bcsstm35	71	124				
ex3	388	380				
rajat13	291	378				
ex10	351	364				
bcsstk27	410	406	3×2	1.27	895	878
shermanACd	258	249				
meg1	365	309				
mark3jac020	219	369				
tuma2	230	314				
G58	285	273				
G59	292	272				
rdist3a	369	359	2×2	1.37	585	728
bcsstk14	388	391	2×4	1.43	757	914

F.2 Medium Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
comsol	266	250				
bcsstk13	306	289				
heart2	236	218	1×8	1.25	337	295
orani678	286	249				
ex28	313	297	2×2	1.11	591	755
nasa2910	228	220	5×1	1.21	310	309
psmigr_3	233	216				
garon1	274	261				
raefsky2	231	222	2×2	1.02	303	294
ex9	255	219				
thermal	321	316				
heart1	235	221	1×8	1.34	337	298
bcsstk24	229	203	2×2	1.03	297	294
bcsstk15	229	223				
ex12	285	273				
sts4098	288	282				
lhr04	282	273				
rdist1	253	239				
struct4	227	218				
bcsstk16	227	214	3×3	1.02	318	263
SiNa	220	207				
ex18	260	266				
Na5	223	208				
raefsky5	224	202	2×2	1.12	292	274
ex15	224	209				
G64	223	200				
goodwin	228	217				
lhr07	220	194				
sinc12	222	193				
ex40	228	219				
bcsstk38	220	210				
benzene	214	199				
bcsstk33	228	211				
nd3k	228	208	3×3	1.11	326	258
nemeth02	219	209	4×1	1.32	308	288
nemeth16	227	220	4×1	1.22	315	292
nemeth19	230	221	2×4	1.25	327	302
nemeth21	231	219	1×8	1.19	325	289
nemeth26	232	221	1×8	1.21	327	298

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
coater2	213	201				
fv2	250	195				
shuttle_eddy	218	189				
pkustk02	225	208	2×6	1.03	313	289
igbt3	210	173				
k3plates	222	212				
coupled	181	170				
cage10	206	177				
t2dah_a	208	196				
sinc15	223	29				
sme3Da	212	193				
stokes64	210	173				
skirt	205	195				
poisson3Da	196	177				
Pres_Poisson	223	202				
rajat07	247	275				
powersim	180	232				
sinc18	225	195				
pds10	182	170				
pkustk07	226	199	3×3	1	318	257
gyro_m	206	192				
gyro_k	222	207	3×3	1	310	250
nd6k	229	185	3×3	1.12	323	128
nmos3	202	161				
bodyy6	198	192				
t3dl_a	211	188				
ns3Da	200	176				
raefsky3	227	212	1×8	1.01	316	289
pkustk01	219	197	3×3	1	300	254
pkustk08	225	191	3×3	1	316	256
rim	225	206				
tuma1	166	170				
crystm03	213	180				
dtoc	168	193				
mult_dcop_01	168	155				
brainpc2	195	156				
3D_28984_Tetra	208	147	3×3	1.03	284	270
bloweya	184	144				
aug2dc	161	149				
rajat10	174	168				
Zhao1	183	142				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
pkustk09	221	188	2×6	1.04	299	295
lhr34	214	164				
nd12k	227	169	3×3	1.12	317	260
onetone1	175	143				
wathen120	204	170				
pwt	194	157				
rajat15	177	134				
finance256	170	149				
cage11	197	153				
torsion1	187	168				
av41092	219	155	2×1	1.05	249	211
jan99jac120	166	162				
sme3Dc	177	155				
pkustk06	222	180	2×6	1.03	302	280
3dtube	225	191	3×3	1.02	310	255
bcsstm39	90	85				
bcsstk39	224	191				
rma10	225	173				
gridgena	196	147				
stokes128	200	127				
ibm_matrix_2	211	135	3×3	1.03	287	258
ct20stif	219	162	2×2	1.21	280	259
g7jac180	189	135				
struct3	212	171				
copter2	180	129				
pkustk04	222	163	3×3	1	310	256
bayer01	157	129				
g7jac200	189	135				
a5esindl	167	121				
blockqp1	196	117				
qa8fk	213	167				
lhr71	215	150				
nd24k	225	147	3×3	1.12	317	258
ncvxqp3	142	117				
t3dh_e	217	169				
a2nnsnsl	162	110				
pkustk10	221	157	2×6	1.04	301	281
poisson3Db	141	114				
ncvxqp7	143	110				
boyd1	181	98				
tandem_dual	151	137				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
pkustk12	223	140	3×1	1.1	280	219
pkustk13	220	146	3×1	1.1	280	229
ford2	162	133				
matrix_9	212	140	3×3	1.01	287	248
hcircuit	155	116				
lung2	170	130				
barrier2-1	190	110				
torso2	205	129				
torso1	230	106	2×3	1.18	305	130
twotone	187	118				
matrix-new_3	210	143	3×3	1.03	285	243

F.3 Large Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
pkustk14	223	108				
para-6	191	104				
gearbox	221	149	3×3	1	308	253
para-10	191	105				
xenon2	208	80	3×3	1.06	291	247
scircuit	129	128				
cont-300	186	120				
ohne2	215	121				
stomach	202	131				
pwtk	222	103	3×3	1.22	309	250
torso3	202	129				
Ga41As41H72	202	116				
Stanford	53	63				
rajat24	146	111				
language	91	69				
rajat21	144	104				
cage13	188	66				
boyd2	149	68				
af_shell1	218	104	5×1	1	288	197
pre2	185	55				
Stanford_Berkeley	187	73				

F.4 Symmetric Matrices

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
lshp_577	705			
bcsstm34	1077	3×1	1.18	1269
can_634	816			
nos6	737			
685_bus	447			
can_715	785			
nos7	785			
Si2	932			
lshp_778	738			
G1	1128			
bcsstk19	804			
bcsstm19	202			
can_838	859			
dwt_869	812			
dwt_918	698			
jagmesh1	745			
nos3	931	2×2	1.08	1006
dwt_992	948			
lshp1009	729			
bcsstk27	1094	1×3	1.16	1268
dwt_1242	770			
jagmesh6	698			
bcsstk11	937	3×1	1.12	1052
bcsstm11	210			
lshp1561	727			
bcspwr07	359			
bcspwr09	475			
bcsstk14	978	2×2	1.15	1121
bcsstk26	807			
bcsstm26	208			
rajat02	582			
bcsstk13	1076			
bcsstm13	853			
blckhole	753			
lshp2233	723			
dwt_2680	736			
nasa2910	911	5×1	1.26	1153
G50	714			
laser	609			

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
lshp3025	736			
lshp3466	734			
bcsstk24	950	2×2	1.06	1003
bcsstm21	167			
bcsstk15	957			
sts4098	795			
struct4	828			
bcsstk16	824	3×3	1.05	867
G58	686			
G59	690			
bcspwr10	457			
SiNa	786			
Na5	794			
Alemdar	939			
G64	593			
commanche_dual	438			
G65	650			
bcsstk38	773			
benzene	733			
bcsstk33	813			
nd3k	845	3×3	1.12	945
nemeth02	780			
nemeth16	813	2×1	1.12	912
nemeth19	827	3×1	1.17	965
nemeth21	843	3×1	1.15	967
nemeth26	849	3×1	1.16	988
shuttle_eddy	844			
pkustk02	807	2×2	1.01	817
m3plates	114			
coupled	479			
t2dah_a	664			
stokes64	838			
skirt	633			
tuma2	627			
Pres_Poisson	758			
rajat07	614			
pds10	502			
pkustk07	821	3×3	1.01	833
gyro_k	771	3×3	1.03	798
gyro_m	632			
nd6k	834	3×3	1.12	935

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
bodyy6	549			
t3dl_a	665			
t3dl_e	69			
pkustk01	744	3×3	1.04	777
pkustk08	822	3×3	1.01	833
tuma1	469			
crystm03	705			
dtoc	423			
bcsstm37	73			
brainpc2	536			
bloweya	526			
aug2dc	430			
rajat10	387			
bcsstm35	91			
pkustk09	750	2×2	1.02	766
nd12k	831	3×3	1.12	933
wathen120	647			
pwt	494			
finance256	455			
torsion1	458			
pkustk06	771	2×2	1.02	784
3dtube	774	3×3	1.05	811
bcsstk39	760			
bcsstm39	91			
gridgena	583			
stokes128	631			
ct20stif	753	2×1	1.12	845
struct3	664			
copter2	480			
pkustk04	792	3×3	1.03	813
a5esindl	390			
blockqp1	598			
qa8fk	702			
nd24k	824	3×3	1.12	925
ncvxqp3	324			
t3dh_e	754			
a2nnsnsl	380			
pkustk10	763	2×2	1.02	777
ncvxqp7	310			
boyd1	623			
tandem_dual	282			

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
pkustk12	795	1×3	1.16	919
pkustk13	783	3×1	1.12	876
ford2	354			
pkustk14	800			
gearbox	776	3×3	1.04	804
cont-300	506			
pwtk	781	3×1	1.15	894
Ga41As41H72	689			
boyd2	322			
af_shell1	729	5×1	1.11	810

Appendix G

SpMV Performance on the Pentium 3

Here we present information about SpMV performance for each of the matrices in our test suite, as well as which ones fall into the categories small, medium, and large, on the Pentium 3. The matrices are sorted in order of increasing problem size. Symmetric and nonsymmetric performance are also compared for symmetric matrices. All performance numbers are in MFLOP/s. Blank values in the tuned columns indicate that OSKI did not tune SpMV for that particular matrix. Raw MFLOP rates (counting operations performed on explicitly stored zero entries that were introduced during blocking) are given for the tuned numbers so that a proper comparison with synthetic matrices, which have no filled in zero entries, can be made. For an MFLOP rate that counts only operations performed on nonzero entries, divide by the fill ratio.

G.1 Small Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
bcsstm19	106	162				
shl_0	160	293				
gre_512	224	341				
bcsstm11	114	163				
dw256A	266	410				
fs_680_3	237	375				
bcsstm26	116	163				
nos6	525	412				
685_bus	381	413				
pores_3	264	312	2×1	1.06	291	345
qh882	221	369				
lshp_577	505	327				
west0989	184	338				
nnc666	248	301				
saylr3	201	359				
fs_541_1	298	383				
sherman4	192	317				
tub1000	283	372				
b_dyn	188	354				
pde900	301	411				
nos7	484	293				
bp_800	233	300				
cdde1	306	412				
lshp_778	519	324				
steam2	458	367	2×4	1.09	597	662
orsirr_2	276	327				
west1505	190	351				
jpwh_991	260	305				
fpga_dcop_01	268	405				
jagmesh1	531	327				
bcsstm21	121	163				
can_715	608	367				
bcspwr07	320	374				
bcsstk19	579	377				
can_634	680	392				
lshp1009	529	327				
dwt_869	581	380				
bcspwr09	405	349				
bfwa782	326	376				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
`dwt_918`	567	345				
`mahindas`	313	294				
`lung1`	222	374				
`west2021`	194	341				
`epb0`	285	366				
`jagmesh6`	531	327				
`pores_2`	307	339	2×1	1	360	397
`can_838`	705	416				
`rotor2`	376	429	3×3	1.32	519	798
`plsk1919`	194	400				
`dwt_1242`	589	347				
`laser`	476	314				
`cage8`	362	390				
`dw1024`	315	410				
`poli`	160	232				
`lshp1561`	540	320				
`rajat02`	472	295				
`watt_1`	272	301				
`extr1`	201	359				
`rajat12`	315	315				
`G50`	592	346				
`add20`	302	403				
`adder_trans_01`	289	283				
`nos3`	802	463	2×2	1.02	1008	634
`blckhole`	538	292				
`pde2961`	329	385				
`lshp2233`	545	262				
`dwt_992`	831	457				
`m3plates`	87	117				
`Si2`	884	496				
`ex21`	495	516	2×1	1.1	604	559
`Pd`	122	145				
`bcsstm13`	692	372				
`lshp3025`	519	258				
`bayer05`	248	283				
`swang1`	267	229				
`sherman2`	434	414	1×2	1.17	444	436
`add32`	198	264				
`bcsstm34`	989	366	2×6	1.31	1290	823
`olm5000`	226	246	1×2	1.25	297	289
`mcfe`	392	343				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
ex25	406	419	1×2	1.08	480	461
circuit_2	254	245				
bcspwr10	364	248				
lshp3466	465	279				
dwt_2680	469	278				
hydr1	185	263				
Hamrle2	234	223				
shermanACa	225	264				
lns_3937	228	201				
meg4	142	242				
bcsstk26	563	263				
bcsstk11	541	325	3×3	1.06	895	812
bayer03	122	191				
cell1	172	193				
gemat11	175	186				
ex27	238	308	2×1	1.18	277	416
G1	535	293				
commanche_dual	276	158				
lhr02	215	207				
G65	333	162				
jan99jac020	144	172				
t3dl_e	72	56				
bcsstm37	48	45				

G.2 Medium Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
bcsstk27	365	184	1×3	1.14	437	290
comsol	147	149				
ex7	175	198				
bcsstk14	317	165	2×6	1.22	453	294
ex3	177	190	2×1	1.15	207	246
bcsstk13	287	148				
ex24	185	188				
heart2	153	153	2×6	1.29	197	196
rdist3a	158	158	1×2	1.21	327	210
ex10	166	179				
orani678	145	144				
ex28	142	145	1×2	1.08	163	171
meg1	158	158				
nasa2910	278	142	5×5	1.22	354	198
psmigr_3	149	150				
garon1	136	144				
raefsky2	144	144	1×2	1.02	164	158
ex9	135	136				
thermal	138	142				
cage9	180	178				
heart1	153	153	1×2	1.1	177	164
bcsstk24	268	137	1×2	1.04	305	151
bcsstk15	260	131				
ex12	133	133				
sts4098	258	135				
lhr04	133	136				
rdist1	132	131				
struct4	275	138				
bcsstk16	281	141	3×3	1.02	353	190
G58	270	130				
G59	272	129				
SiNa	255	131				
ex18	123	122				
Na5	266	136				
shermanACd	130	125				
Alemdar	229	139				
raefsky5	129	125	2×1	1.07	151	146
rajat01	134	127				
ex15	123	117				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
G64	231	110				
goodwin	137	138				
lhr07	127	119				
sinc12	132	127				
rajat13	131	122				
ex40	141	140				
bcsstk38	269	133	1×2	1.18	306	147
dw4096	140	128				
benzene	244	123				
bcsstk33	281	142	1×2	1.17	322	153
nd3k	299	145	3×3	1.11	379	200
mark3jac020	108	103				
nemeth02	269	137	1×2	1.17	305	151
nemeth16	279	142	1×2	1.13	318	155
nemeth19	289	146	1×2	1.09	330	157
nemeth21	295	148	1×2	1.08	338	159
nemeth26	301	151	1×2	1.09	344	162
coater2	123	123				
fv2	114	106				
shuttle_eddy	208	106				
pkustk02	281	127	2×6	1.03	362	189
igbt3	113	97				
k3plates	133	132	1×2	1.16	150	146
coupled	180	93				
cage10	109	104				
t2dah_a	225	115				
sinc15	135	122				
sme3Da	123	124				
stokes64	208	96				
skirt	223	114				
tuma2	183	94				
poisson3Da	107	107				
Pres_Poisson	269	126	1×2	1.19	311	148
rajat07	165	93				
powersim	87	86				
sinc18	138	119				
pds10	190	92				
pkustk07	295	124	3×3	1	370	189
gyro_m	233	112				
gyro_k	276	128	3×3	1	345	189
nd6k	301	130	3×3	1.12	378	195

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
nmos3	109	92				
bodyy6	181	93				
t3dl_a	243	119				
ns3Da	110	108				
raefsky3	141	141	2×8	1.03	178	197
pkustk01	266	115	3×3	1	332	184
pkustk08	295	115	3×3	1	364	185
rim	137	135				
tuma1	145	71				
crystm03	238	121				
dtoc	148	73				
mult_dcop_01	74	69				
brainpc2	171	62				
3D_28984_Tetra	115	85	3×3	1.03	142	150
bloweya	163	68				
aug2dc	147	69				
rajat10	145	81				
bcsstm35	49	41				
Zhao1	75	64				
pkustk09	269	111	2×6	1.04	342	184
lhr34	126	94				
nd12k	296	96	3×3	1.12	372	182
onetone1	100	79				
wathen120	221	103				
pwt	183	92				
rajat15	89	62				
finance256	170	69				
cage11	103	83				
torsion1	159	83				
av41092	132	77	2×1	1.05	151	94
jan99jac120	89	74				
sme3Dc	84	73				
pkustk06	273	106	2×6	1.03	348	185
3dtube	276	118	3×3	1.02	346	186
bcsstm39	44	34				
bcsstk39	270	123				
rma10	139	106	1×2	1.16	158	133
gridgena	200	91				
stokes128	197	74				
ibm_matrix_2	116	80	3×3	1.03	142	147
ct20stif	273	92	1×2	1.12	304	123

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
`g7jac180`	109	70				
`struct3`	239	109				
`copter2`	184	63				
`pkustk04`	282	91	3×3	1	351	179
`bayer01`	78	56				
`g7jac200`	109	67				
`a5esindl`	118	54				
`blockqp1`	186	48				

G.3 Large Matrices

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
qa8fk	241	106				
lhr71	125	81				
nd24k	296	70	3×3	1.12	370	166
ncvxqp3	109	40				
t3dh_e	273	108				
a2nnsnsl	117	49				
pkustk10	273	96	2×6	1.04	345	184
poisson3Db	64	44				
ncvxqp7	109	38				
boyd1	175	43				
tandem_dual	127	71				
pkustk12	280	73	1×3	1.12	321	140
pkustk13	281	84	1×3	1.11	318	145
ford2	142	68				
matrix_9	116	78	3×3	1.01	138	158
hcircuit	73	50				
lung2	78	63				
barrier2-1	107	50				
torso2	99	71				
torso1	144	41	1×2	1.06	162	71
twotone	101	58				
matrix-new_3	117	65	3×3	1.03	140	128
pkustk14	285	72	1×2	1.18	325	109
para-6	110	46				
gearbox	276	83	3×3	1	346	173
para-10	110	47				
xenon2	122	68	3×3	1.06	147	153
scircuit	67	49				
cont-300	155	49				
ohne2	131	50				
stomach	109	60				
pwtk	273	66	1×2	1.13	307	103
torso3	112	55				
Ga41As41H72	239	46				
Stanford	30	25				
rajat24	74	40				
language	44	29				
rajat21	72	38				
cage13	96	34				

	Untuned Performance		Tuned Performance			
	Real	Synthetic	Blocksize	Fill Ratio	Real	Synthetic
boyd2	104	29				
af_shell1	255	42	5×5	1	317	168
pre2	98	30				
Stanford_Berkeley	99	32				

G.4 Symmetric Matrices

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
lshp_577	472			
bcsstm34	920	2×1	1.16	1065
can_634	601			
nos6	425			
685_bus	339			
can_715	576			
nos7	497			
Si2	785			
lshp_778	486			
G1	948			
bcsstk19	547			
bcsstm19	105			
can_838	663			
dwt_869	568			
dwt_918	532			
jagmesh1	500			
nos3	706	2×2	1.08	763
dwt_992	753			
lshp1009	488			
bcsstk27	824	1×3	1.16	955
dwt_1242	572			
jagmesh6	484			
bcsstk11	796	1×3	1.11	885
bcsstm11	113			
lshp1561	510			
bcspwr07	293			
bcspwr09	356			
bcsstk14	697	2×2	1.15	800
bcsstk26	687			
bcsstm26	115			
rajat02	460			
bcsstk13	625			
bcsstm13	706			
blckhole	544			
lshp2233	524			
dwt_2680	581			
nasa2910	481			
G50	443			
laser	447			

	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
lshp3025	536			
lshp3466	534			
bcsstk24	469	2×2	1.06	495
bcsstm21	120			
bcsstk15	473			
sts4098	534			
struct4	462			
bcsstk16	470	1×3	1.04	489
G58	512			
G59	503			
bcspwr10	354			
SiNa	427			
Na5	438			
Alemdar	493			
G64	414			
commanche_dual	324			
G65	460			
bcsstk38	442			
benzene	395			
bcsstk33	468	2×1	1.17	547
nd3k	497	2×2	1.18	586
nemeth02	439			
nemeth16	463	2×2	1.23	571
nemeth19	486	2×2	1.16	564
nemeth21	500	2×2	1.16	577
nemeth26	509	4×1	1.21	615
shuttle_eddy	341			
pkustk02	468	2×2	1.01	474
m3plates	92			
coupled	269			
t2dah_a	357			
stokes64	345			
skirt	348			
tuma2	328			
Pres_Poisson	442			
rajat07	214			
pds10	290			
pkustk07	495	1×3	1.01	501
gyro_k	451	3×1	1.03	467
gyro_m	363			
nd6k	494	2×2	1.19	586

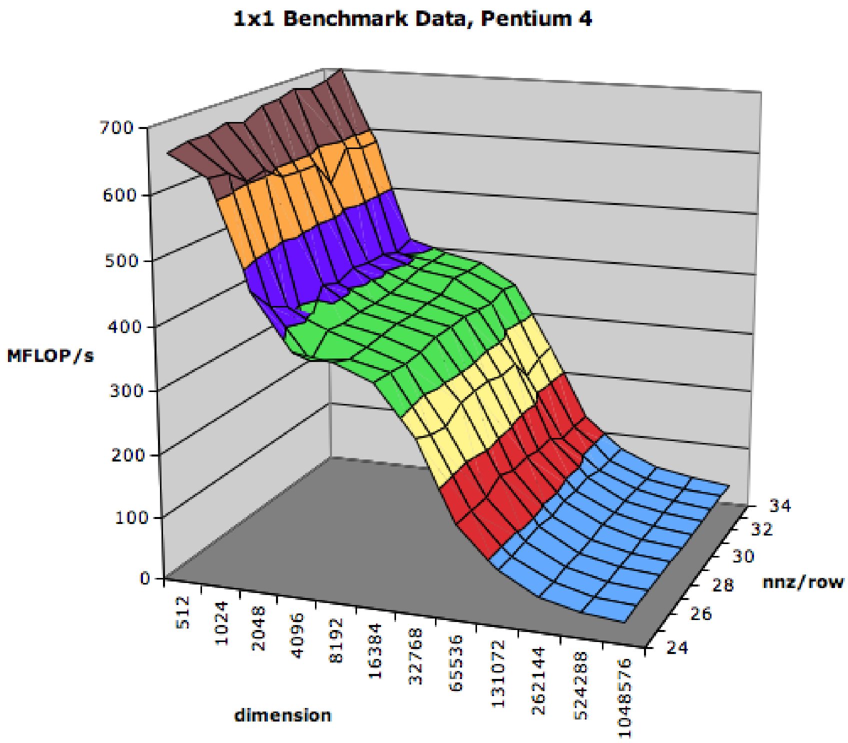
	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
`bodyy6`	268			
`t3dl_a`	383			
`t3dl_e`	70			
`pkustk01`	433	1×3	1.04	452
`pkustk08`	491	1×3	1.01	498
`tuma1`	218			
`crystm03`	378			
`dtoc`	220			
`bcsstm37`	51			
`brainpc2`	234			
`bloweya`	251			
`aug2dc`	210			
`rajat10`	175			
`bcsstm35`	54			
`pkustk09`	435	2×2	1.02	444
`nd12k`	488	2×2	1.19	579
`wathen120`	334			
`pwt`	269			
`finance256`	238			
`torsion1`	209			
`pkustk06`	449	2×2	1.02	457
`3dtube`	444	1×3	1.04	461
`bcsstk39`	439			
`bcsstm39`	43			
`gridgena`	293			
`stokes128`	293			
`ct20stif`	435	2×2	1.23	534
`struct3`	375			
`copter2`	240			
`pkustk04`	464	1×3	1.03	476
`a5esindl`	138			
`blockqp1`	238			
`qa8fk`	376			
`nd24k`	481	2×2	1.19	570
`ncvxqp3`	119			
`t3dh_e`	430			
`a2nnsnsl`	137			
`pkustk10`	443	2×2	1.02	451
`ncvxqp7`	116			
`boyd1`	319			
`tandem_dual`	137			

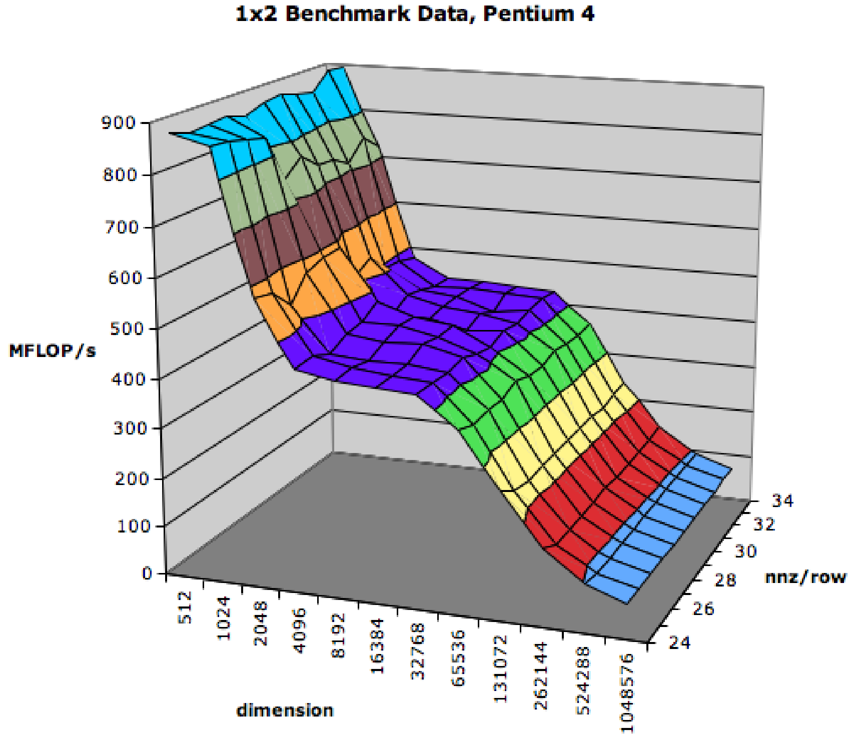
	Untuned Performance	Tuned Performance		
	MFLOP/s	Blocksize	Fill Ratio	MFLOP/s
`pkustk12`	471	1×3	1.16	545
`pkustk13`	460	1×3	1.13	519
`ford2`	174			
`pkustk14`	467	1×2	1.18	550
`gearbox`	455	1×3	1.03	471
`cont-300`	210			
`pwtk`	449	2×2	1.24	555
`Ga41As41H72`	345			
`boyd2`	105			
`af_shell1`	401	5×5	1.11	446

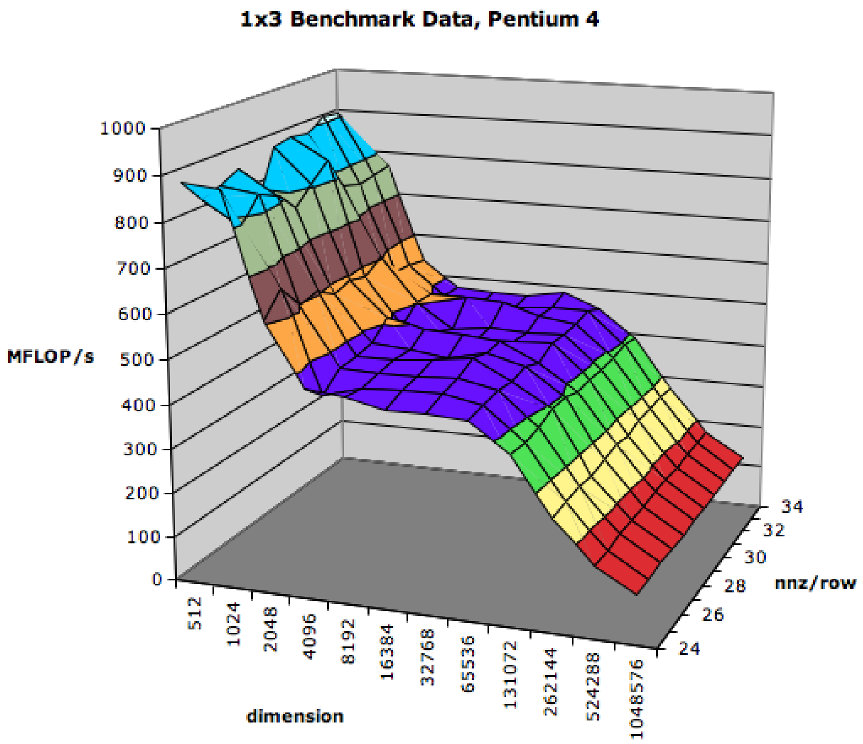
Appendix H

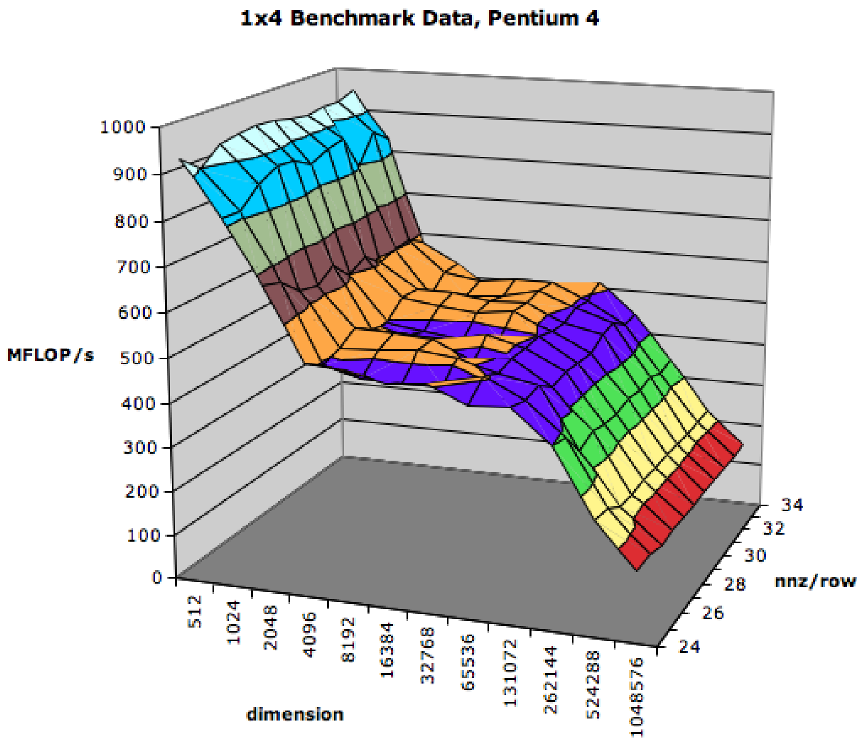
Pentium 4 Benchmark Data

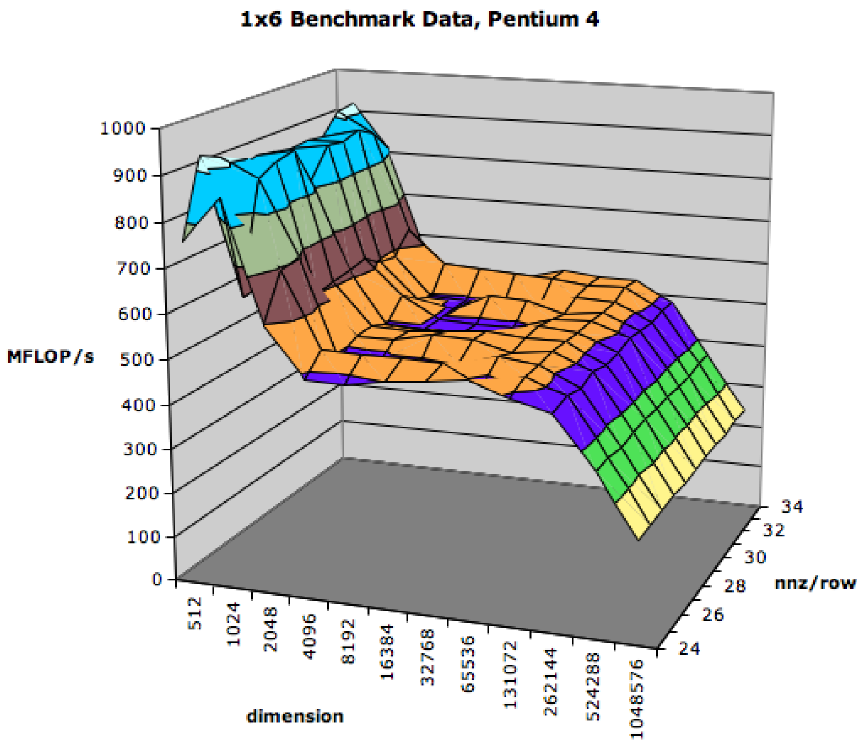
Here we graphically present the full output of our benchmark on the Pentium 4.



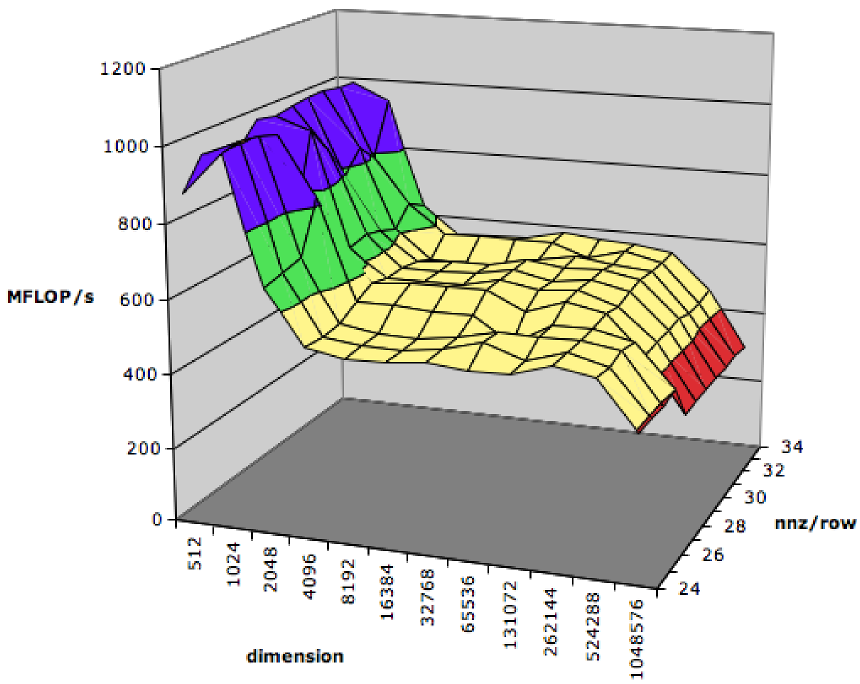

1x2 Benchmark Data, Pentium 4
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
nnz/row
24
26
28
30
32
34


1x3 Benchmark Data, Pentium 4
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
nnz/row
24
26
28
30
32
34

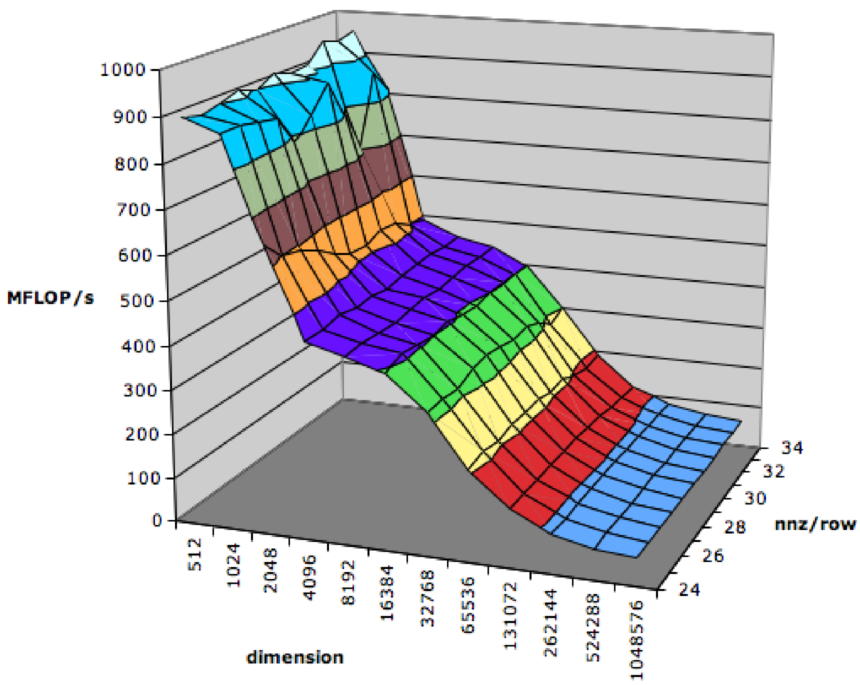

1x4 Benchmark Data, Pentium 4
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

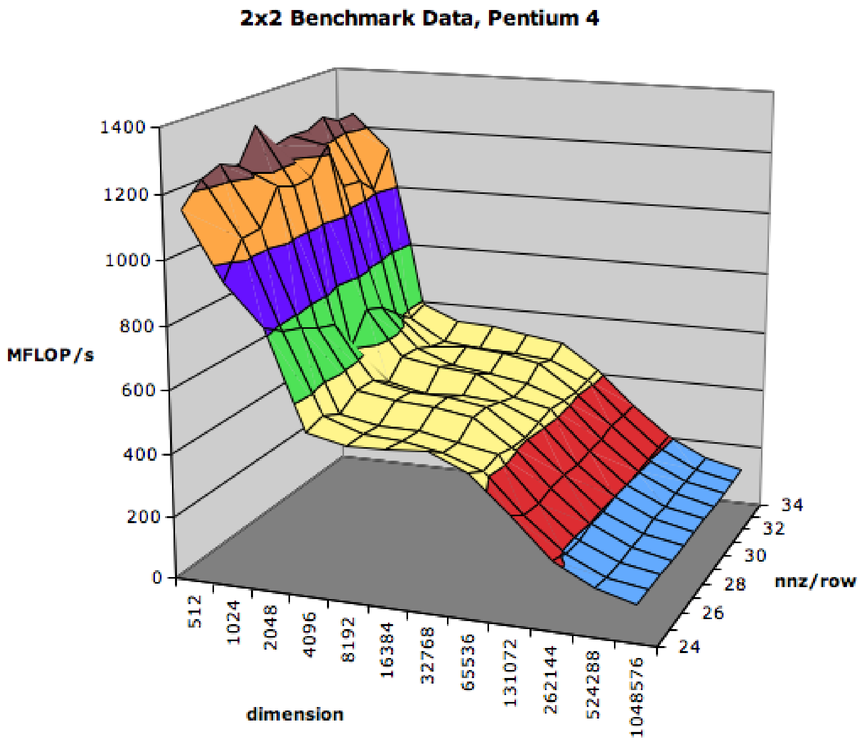

1x6 Benchmark Data, Pentium 4
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

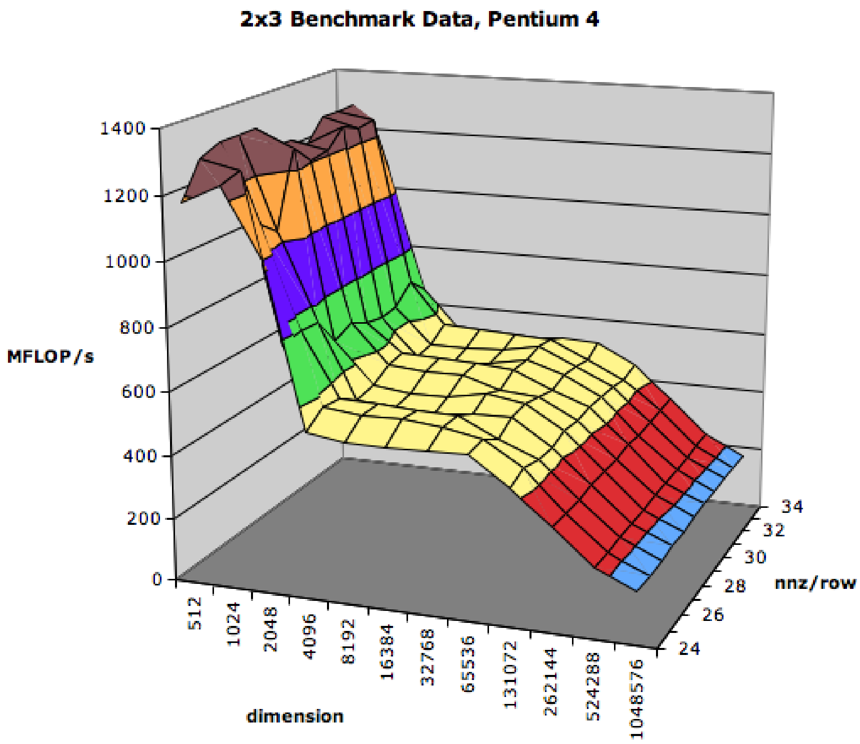
1x8 Benchmark Data, Pentium 4

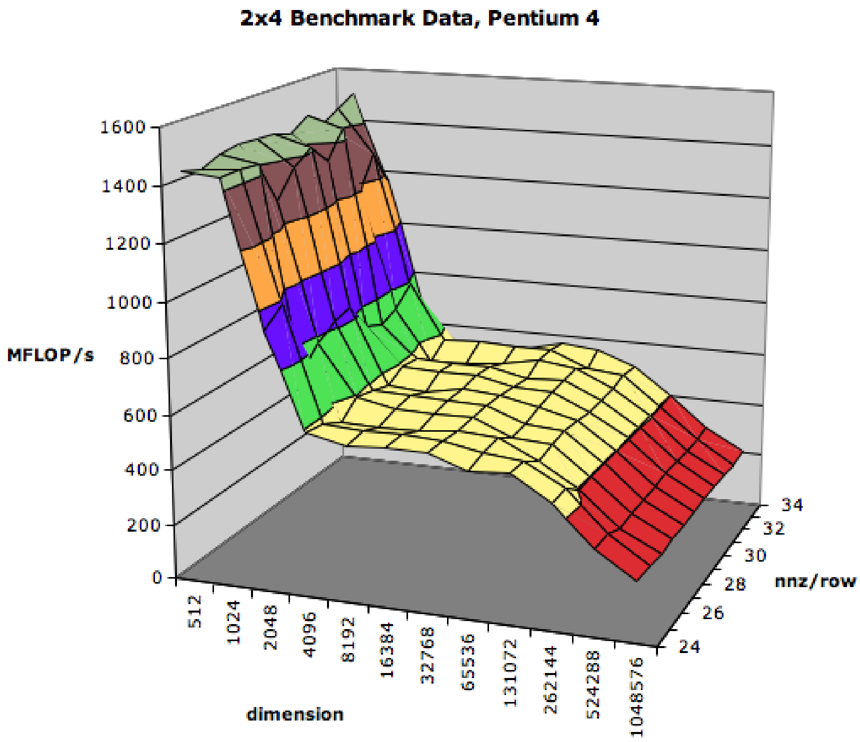


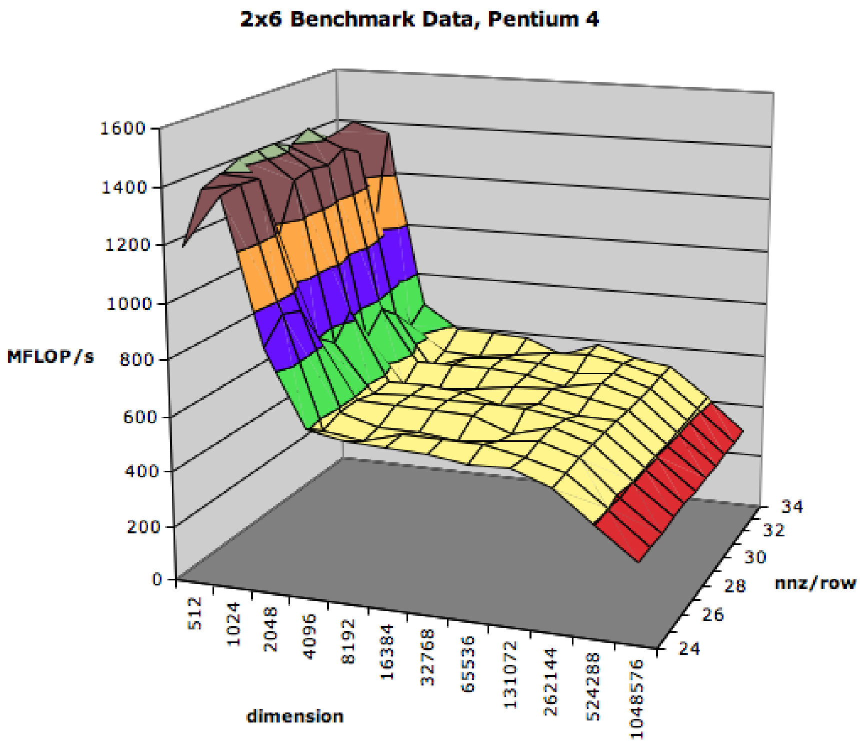
2x1 Benchmark Data, Pentium 4

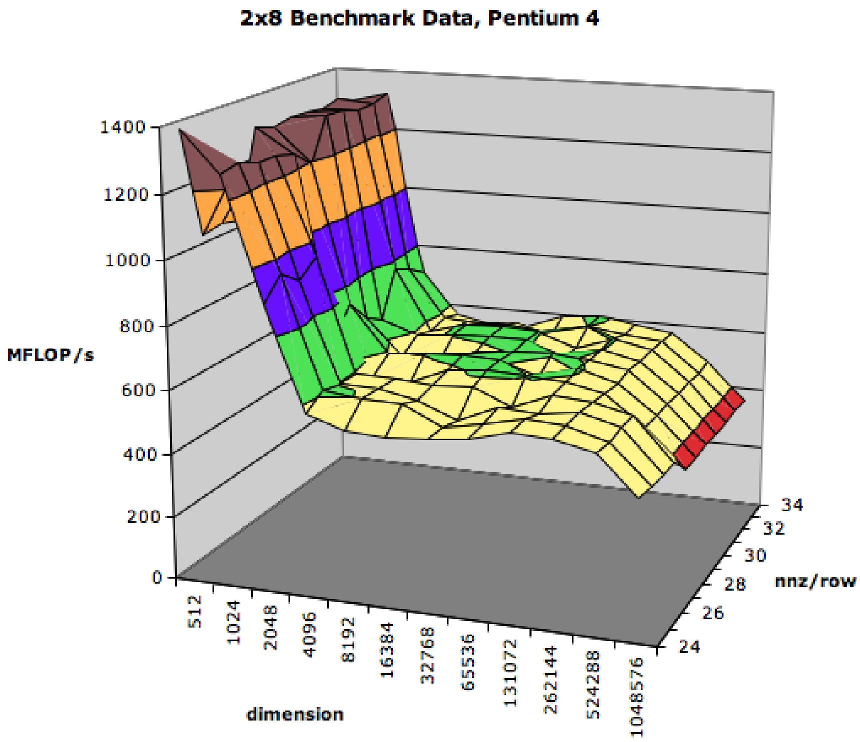


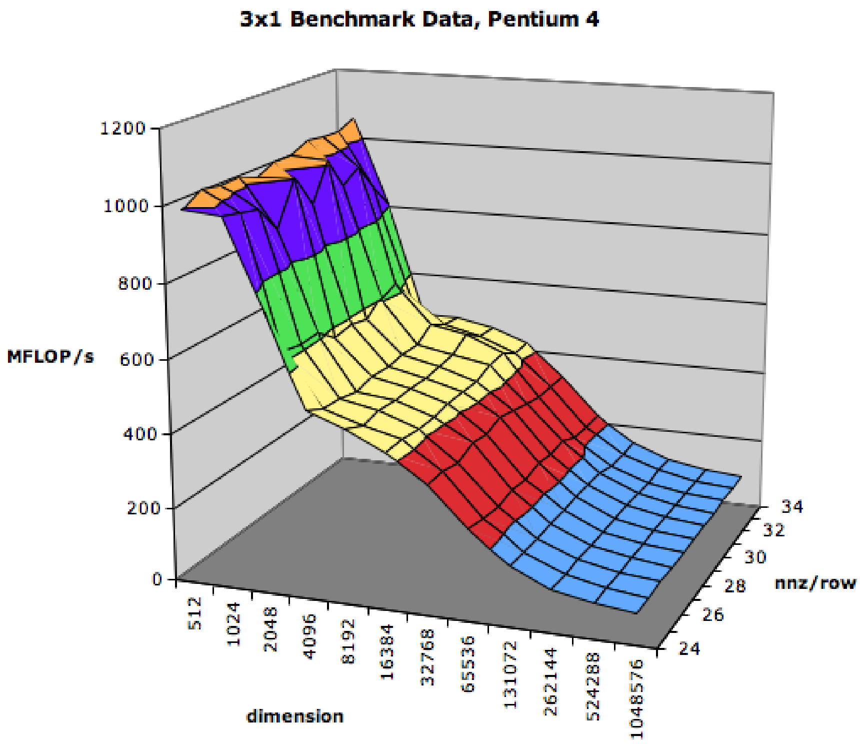

2x2 Benchmark Data, Pentium 4
MFLOP/s
0
200
400
600
800
1000
1200
1400
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

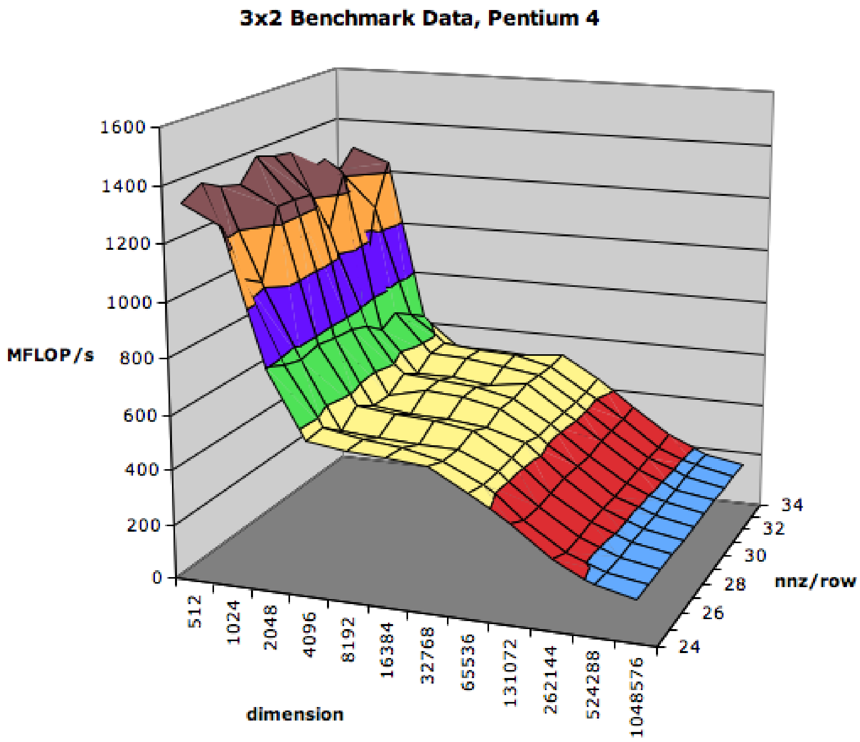

2x3 Benchmark Data, Pentium 4
MFLOP/s
0
200
400
600
800
1000
1200
1400
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

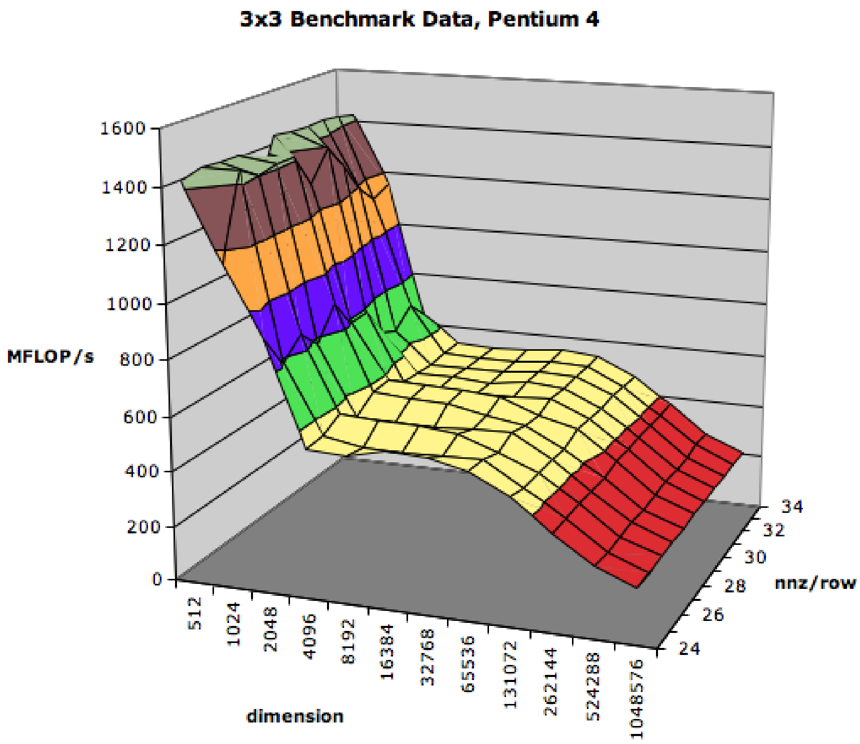

2x4 Benchmark Data, Pentium 4
MFLOP/s
0
200
400
600
800
1000
1200
1400
1600
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

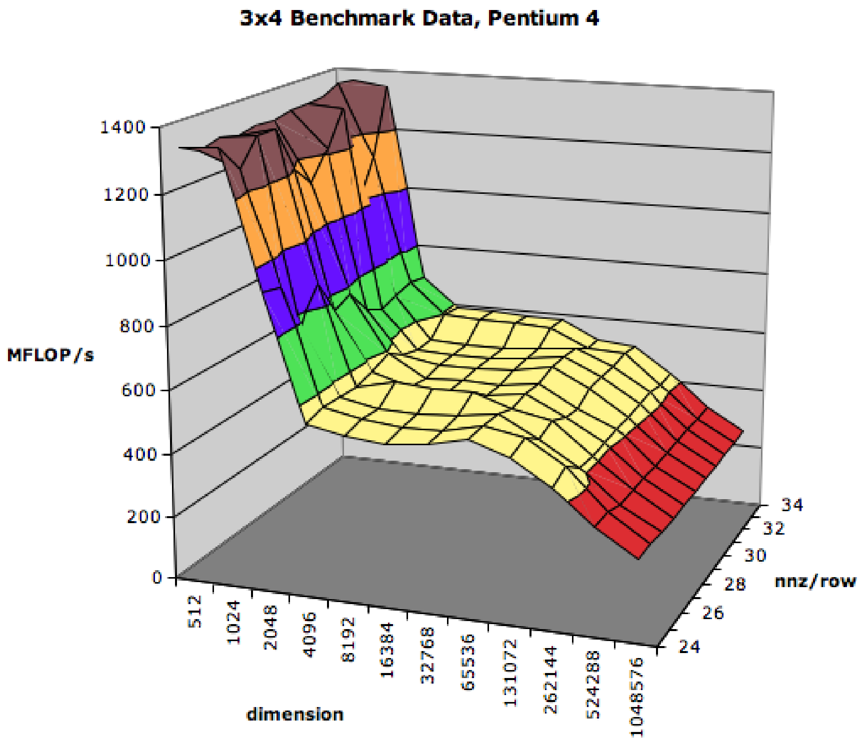

2x6 Benchmark Data, Pentium 4
MFLOP/s
0
200
400
600
800
1000
1200
1400
1600
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

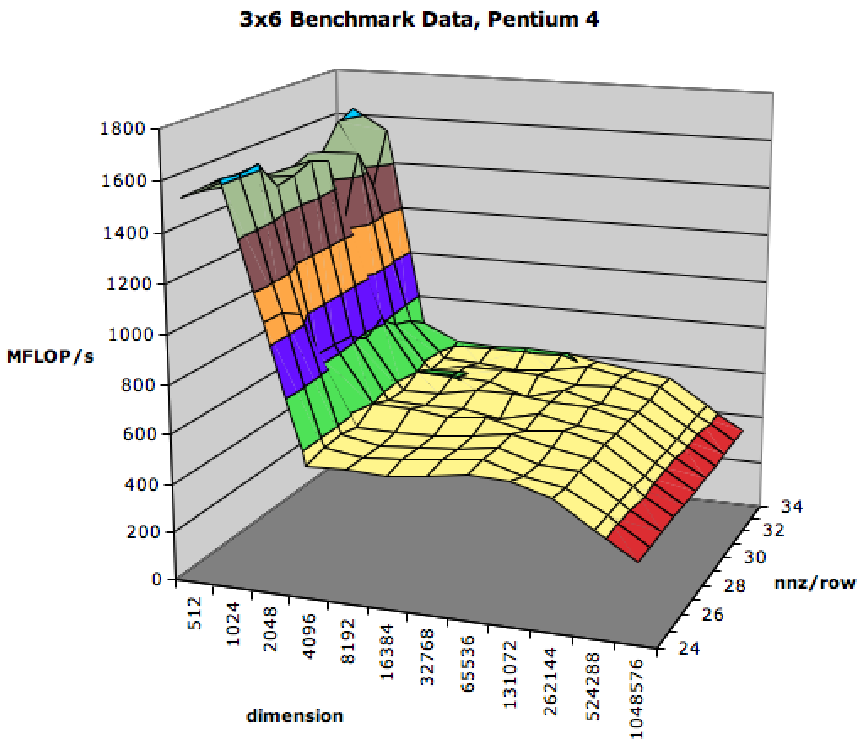

2x8 Benchmark Data, Pentium 4
MFLOP/s
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row


3x1 Benchmark Data, Pentium 4
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

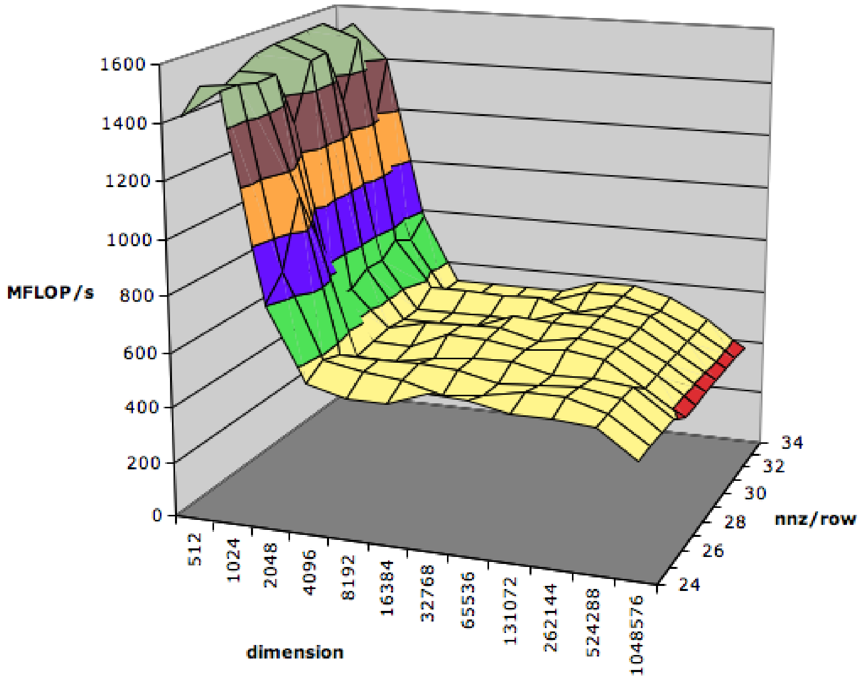

3x2 Benchmark Data, Pentium 4
MFLOP/s
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row


3x3 Benchmark Data, Pentium 4
MFLOP/s
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

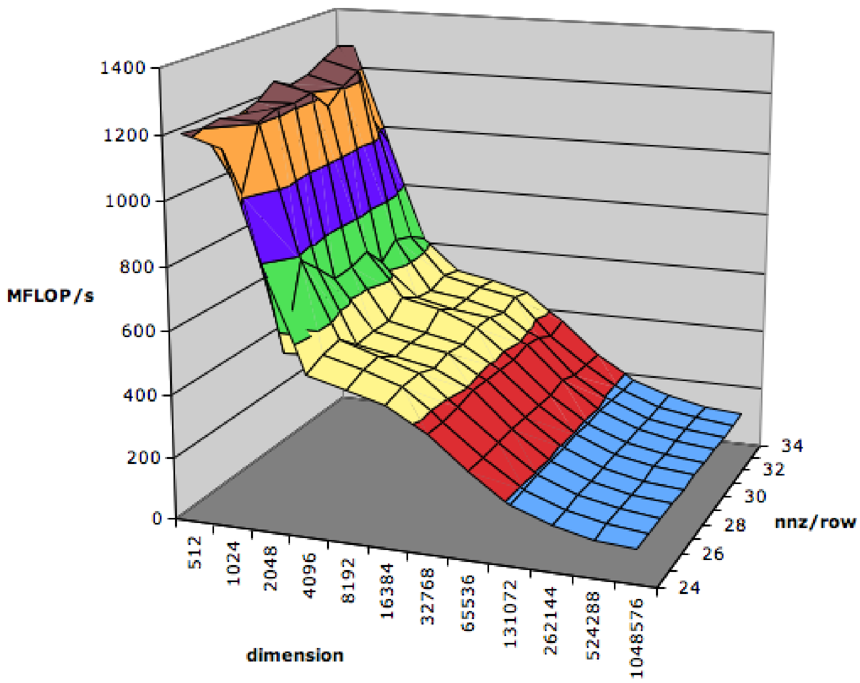

3x4 Benchmark Data, Pentium 4
MFLOP/s
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

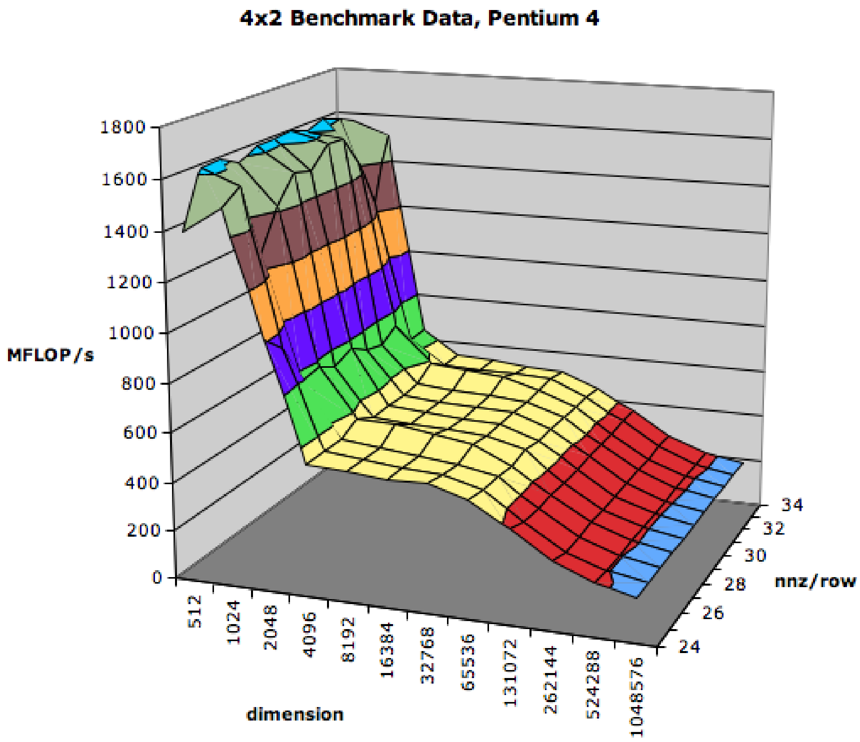

3x6 Benchmark Data, Pentium 4
MFLOP/s
1800
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

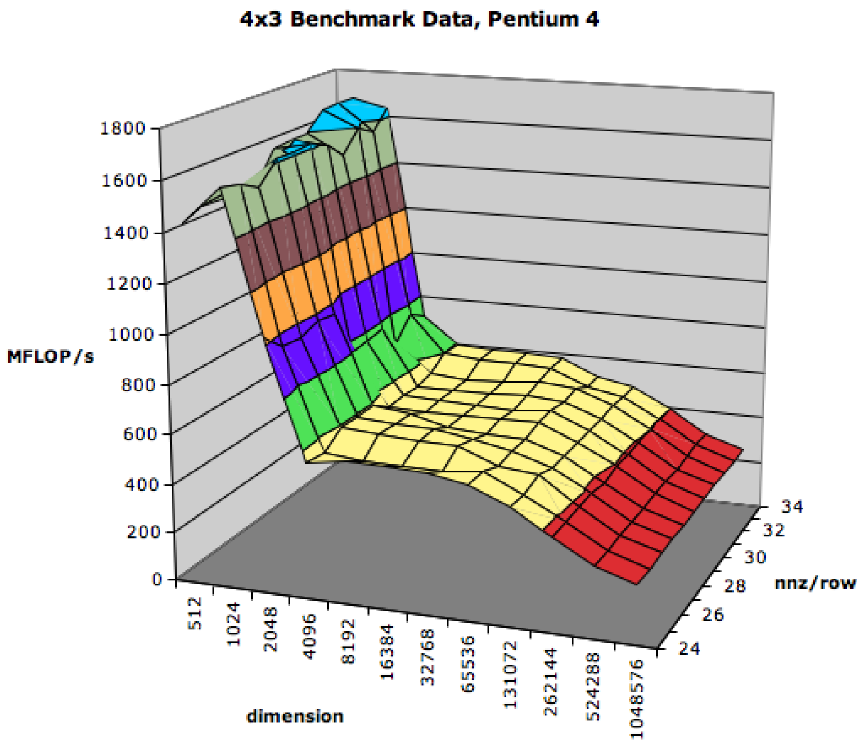
3x8 Benchmark Data, Pentium 4



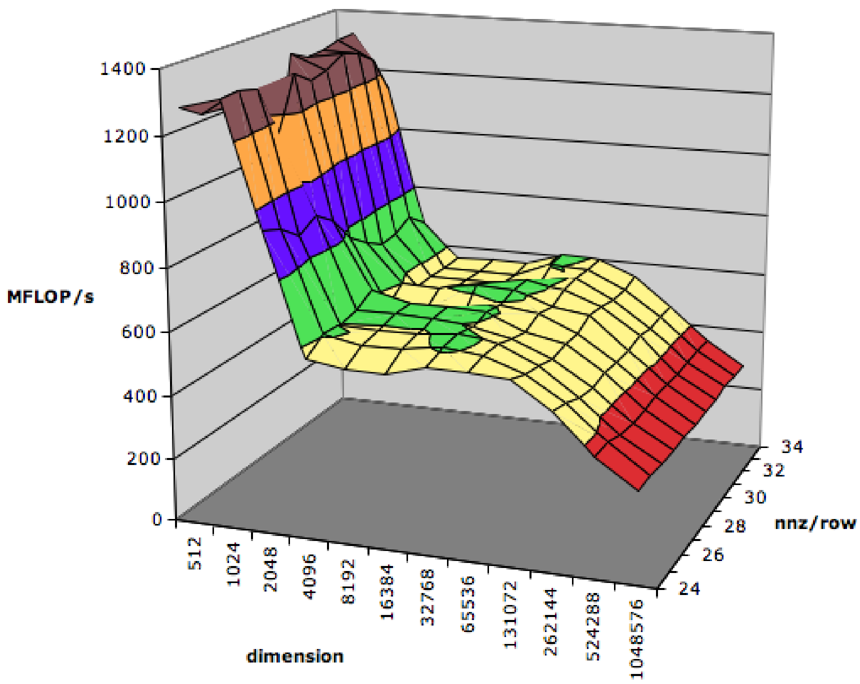
4x1 Benchmark Data, Pentium 4



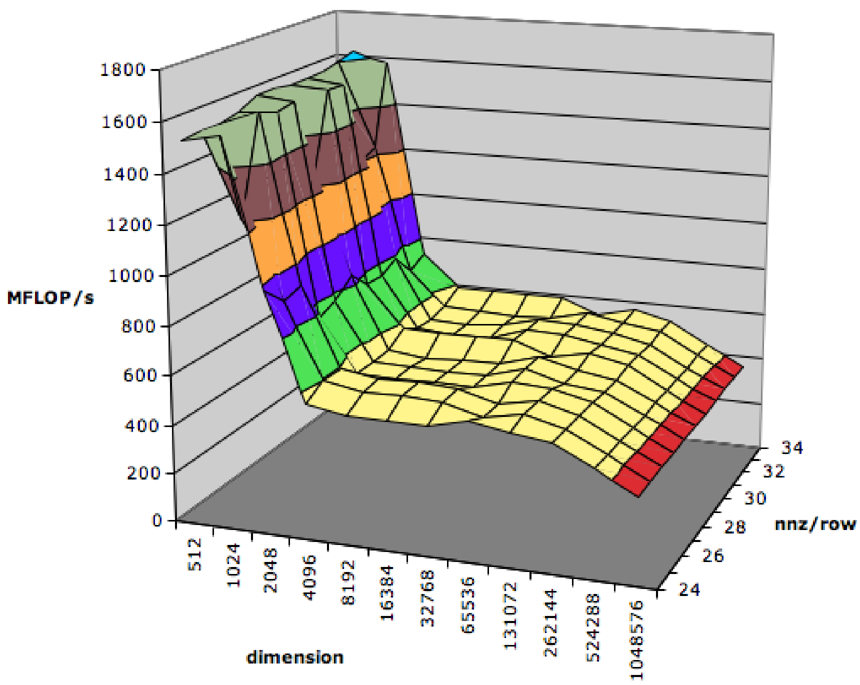

4x2 Benchmark Data, Pentium 4
MFLOP/s
1800
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row


4x3 Benchmark Data, Pentium 4
MFLOP/s
1800
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

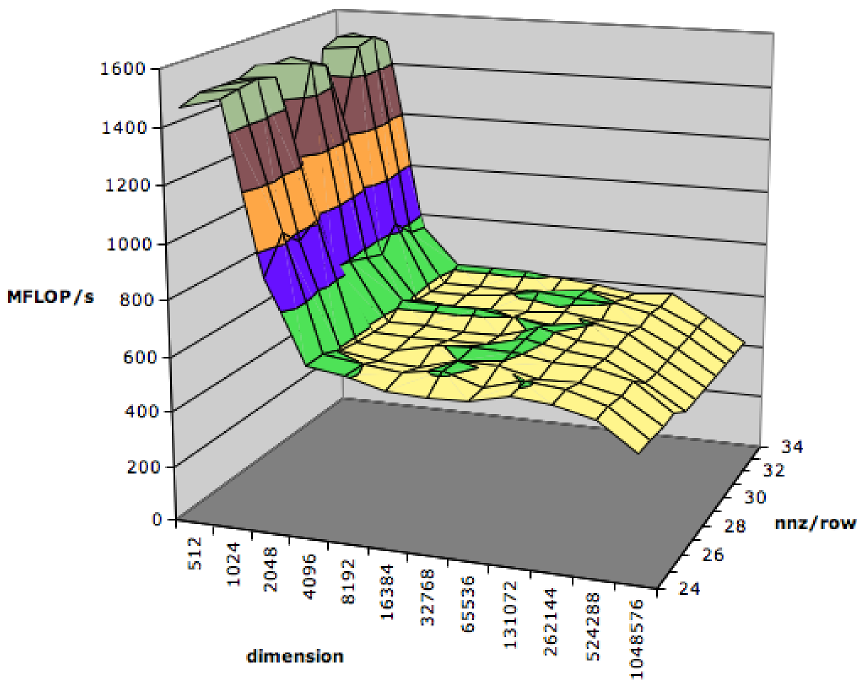
4x4 Benchmark Data, Pentium 4



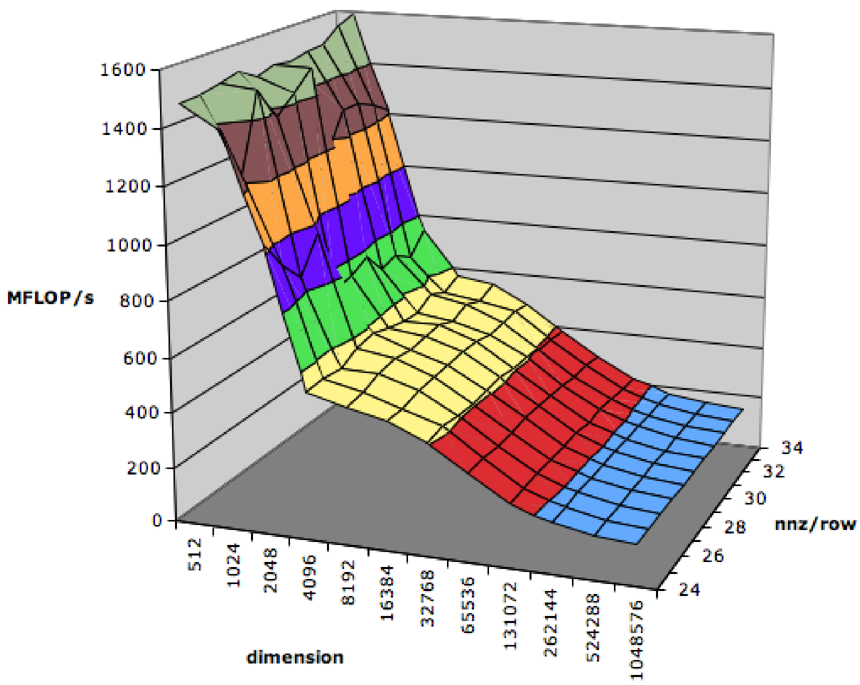
4x6 Benchmark Data, Pentium 4

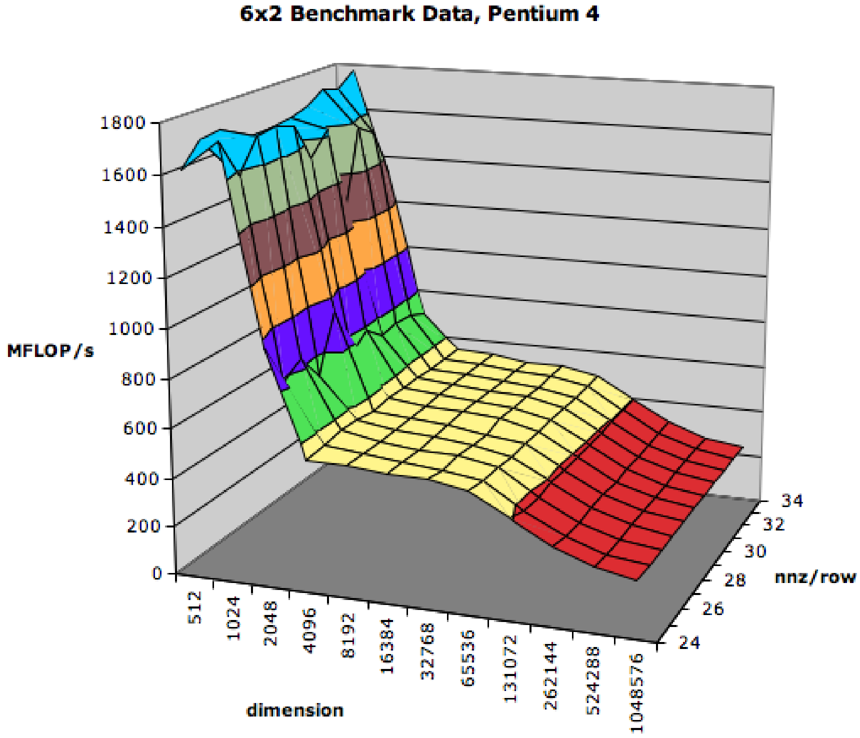


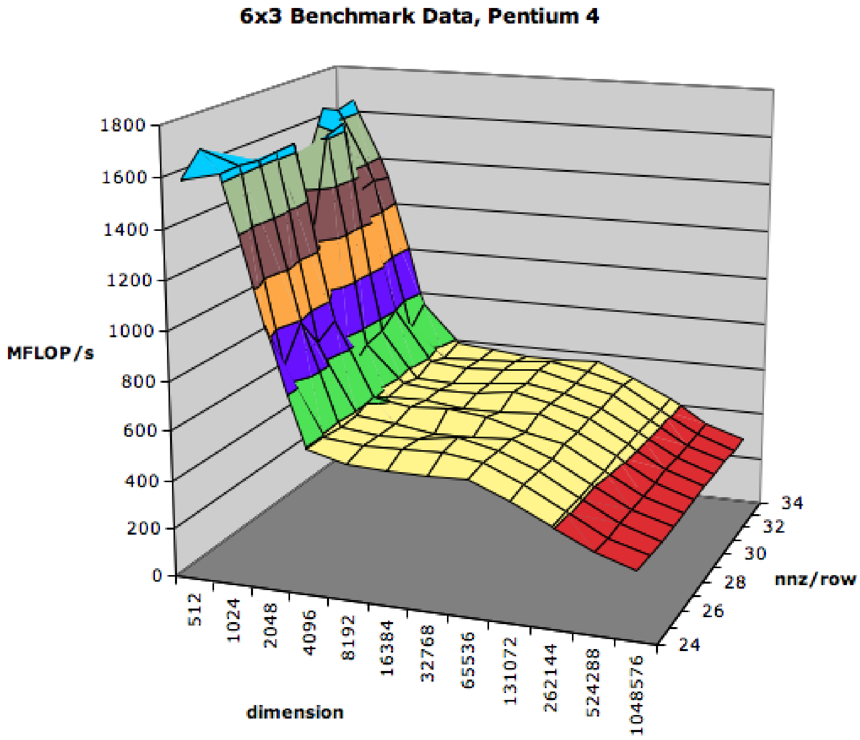
4x8 Benchmark Data, Pentium 4

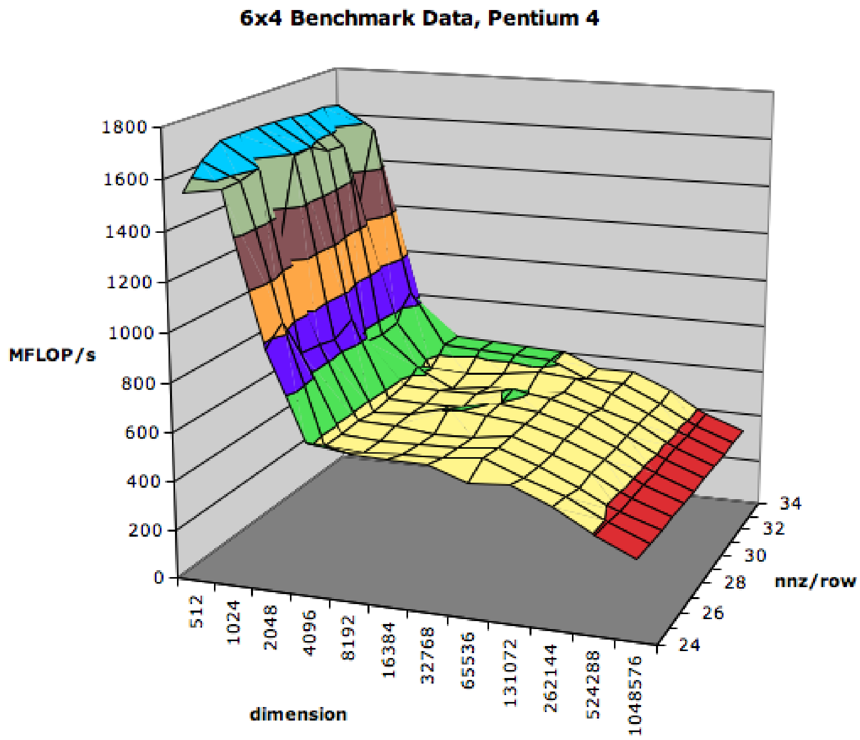


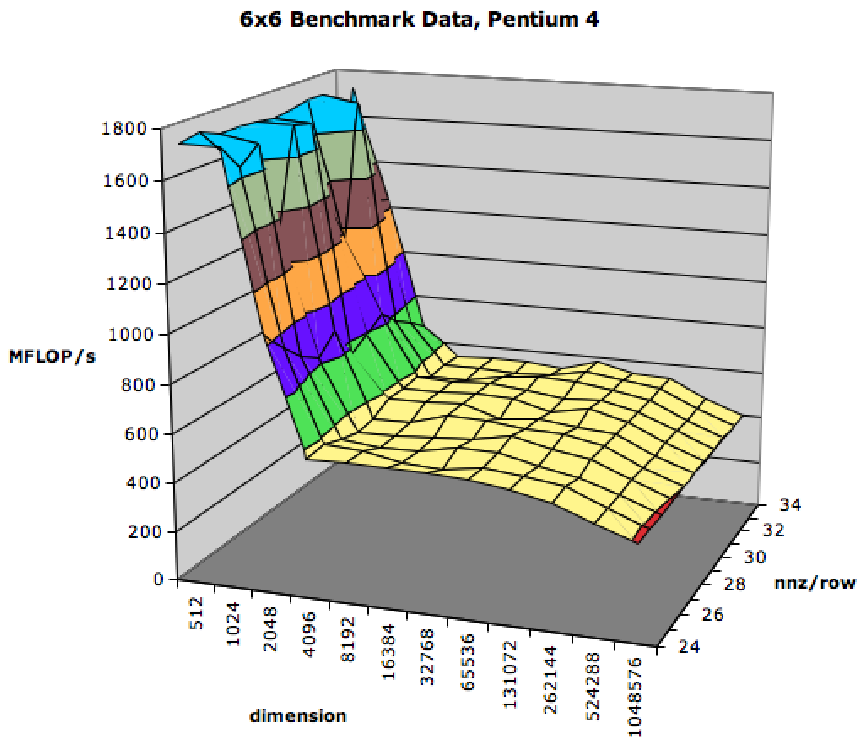
6x1 Benchmark Data, Pentium 4



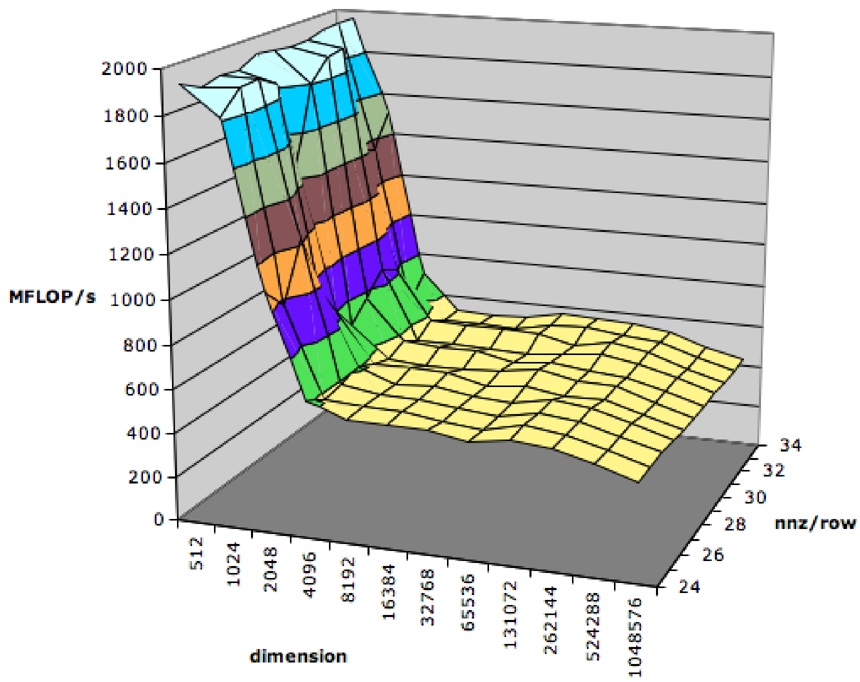

6x2 Benchmark Data, Pentium 4
MFLOP/s
1800
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row


6x3 Benchmark Data, Pentium 4
MFLOP/s
1800
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

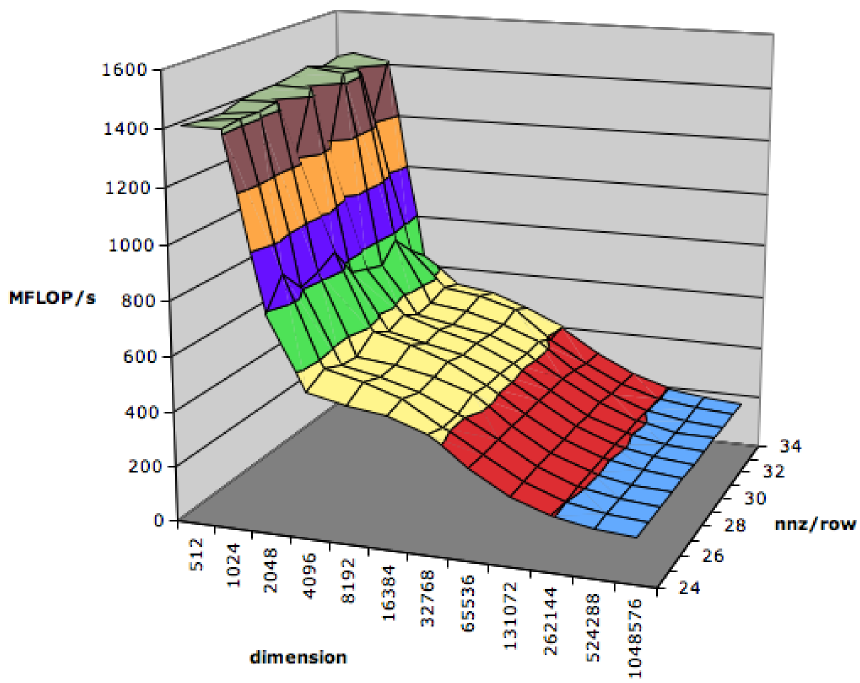

6x4 Benchmark Data, Pentium 4
MFLOP/s
1800
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

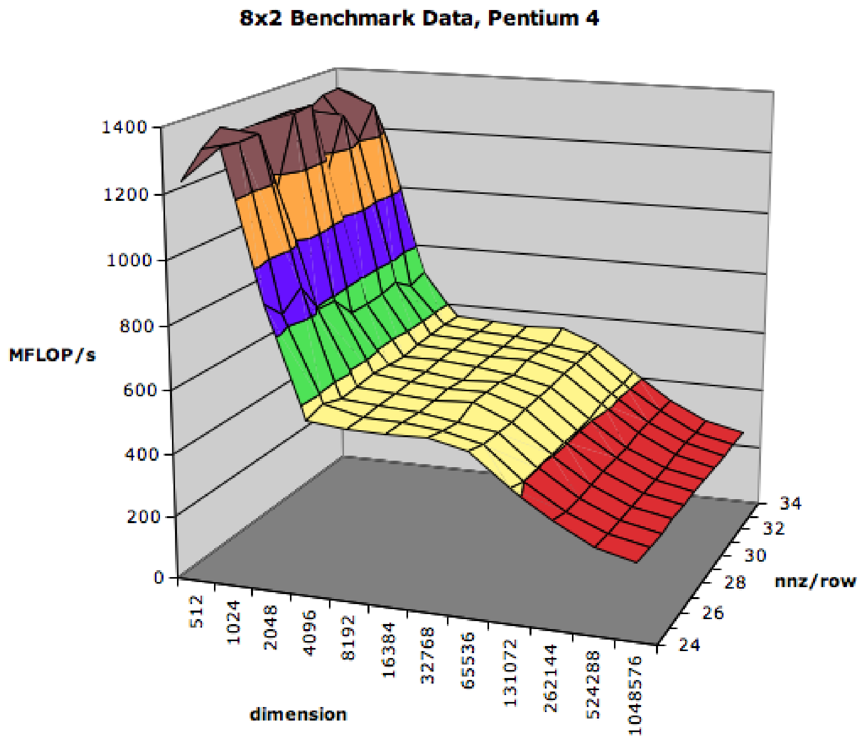

6x6 Benchmark Data, Pentium 4
MFLOP/s
1800
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

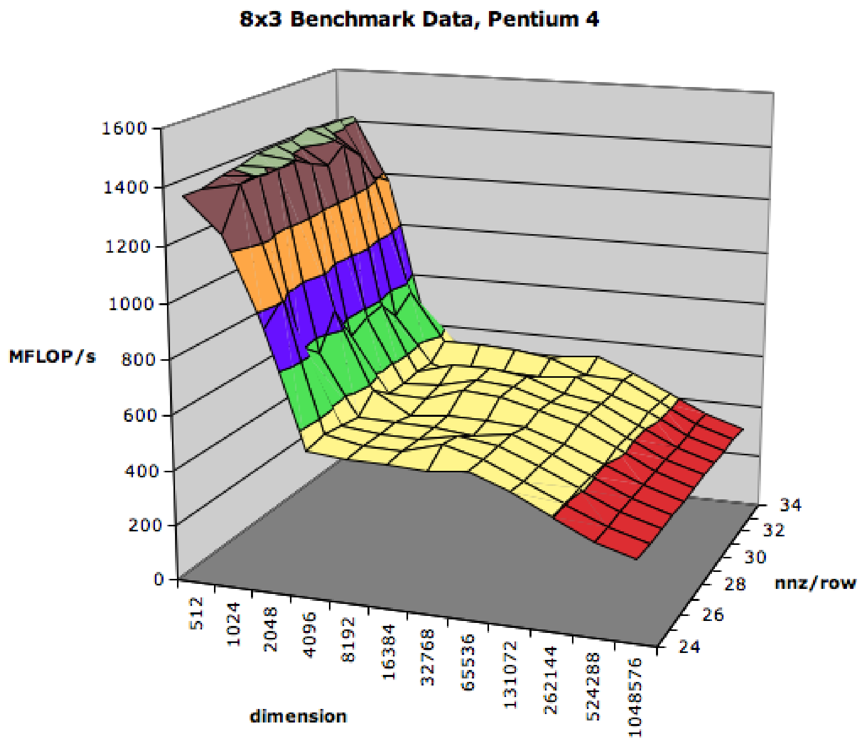
6x8 Benchmark Data, Pentium 4



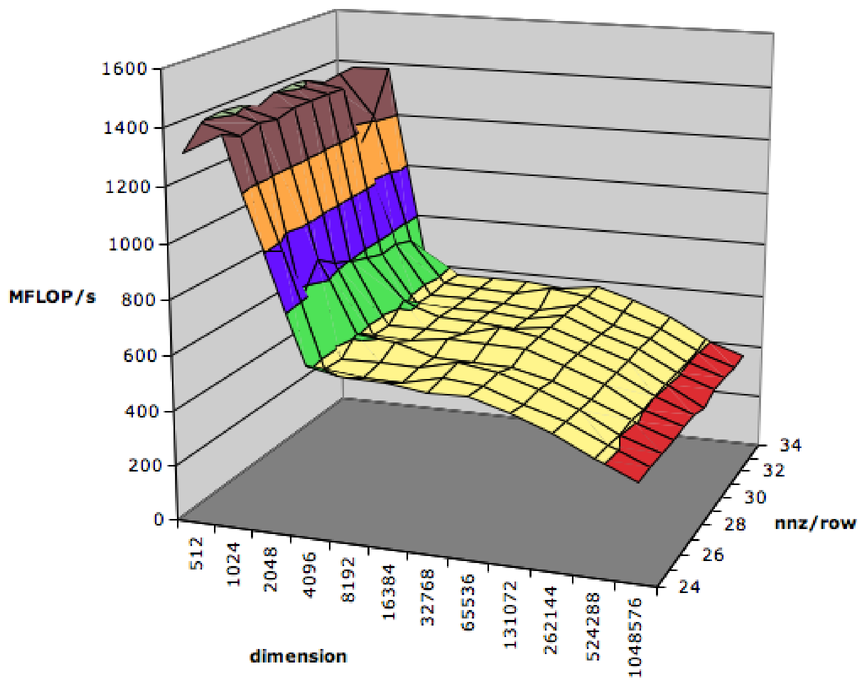
8x1 Benchmark Data, Pentium 4



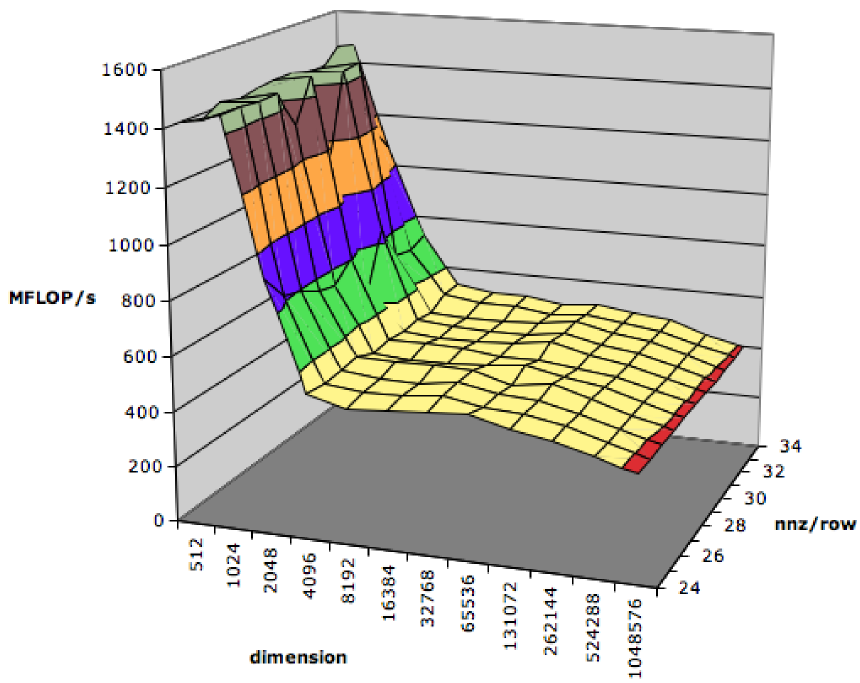

8x2 Benchmark Data, Pentium 4
1400
1200
1000
800
600
400
200
0
MFLOP/s
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

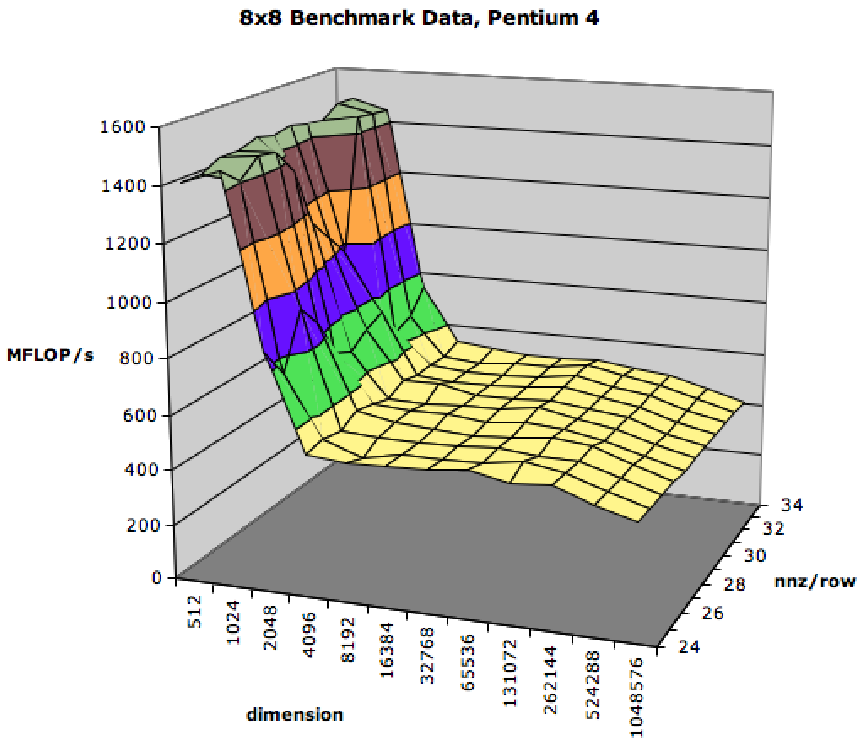

8x3 Benchmark Data, Pentium 4
1600
1400
1200
1000
800
600
400
200
0
MFLOP/s
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

8x4 Benchmark Data, Pentium 4



8x6 Benchmark Data, Pentium 4

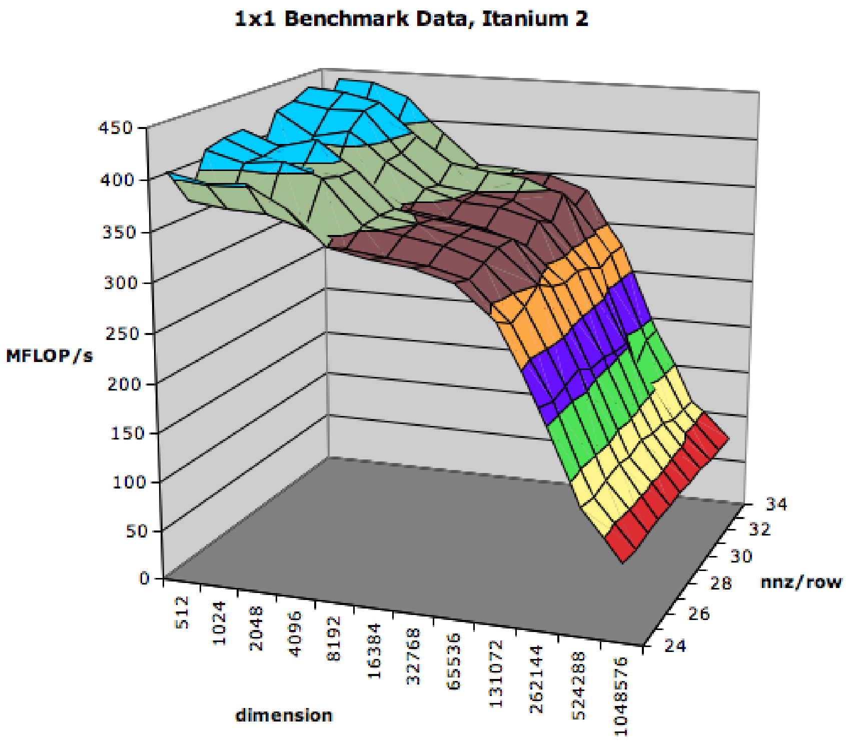



8x8 Benchmark Data, Pentium 4
MFLOP/s
0
200
400
600
800
1000
1200
1400
1600
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

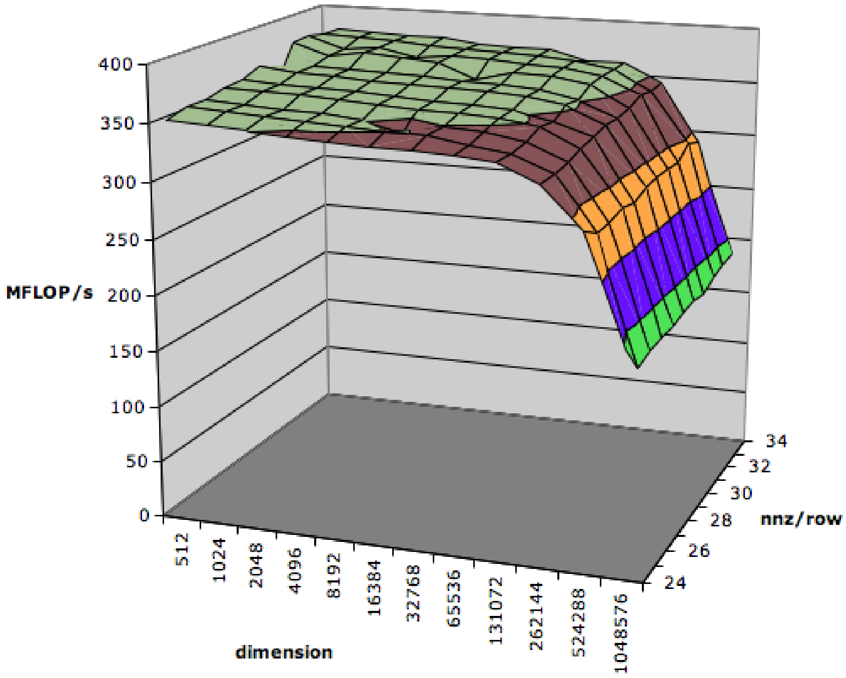
Appendix I

Itanium 2 Benchmark Data

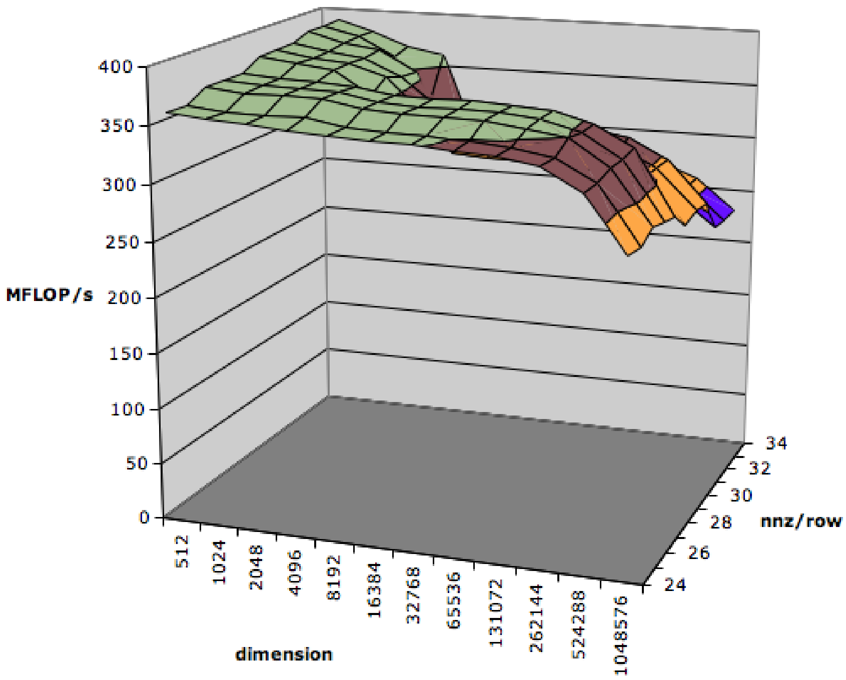
Here we graphically present the full output of our benchmark on the Itanium 2.



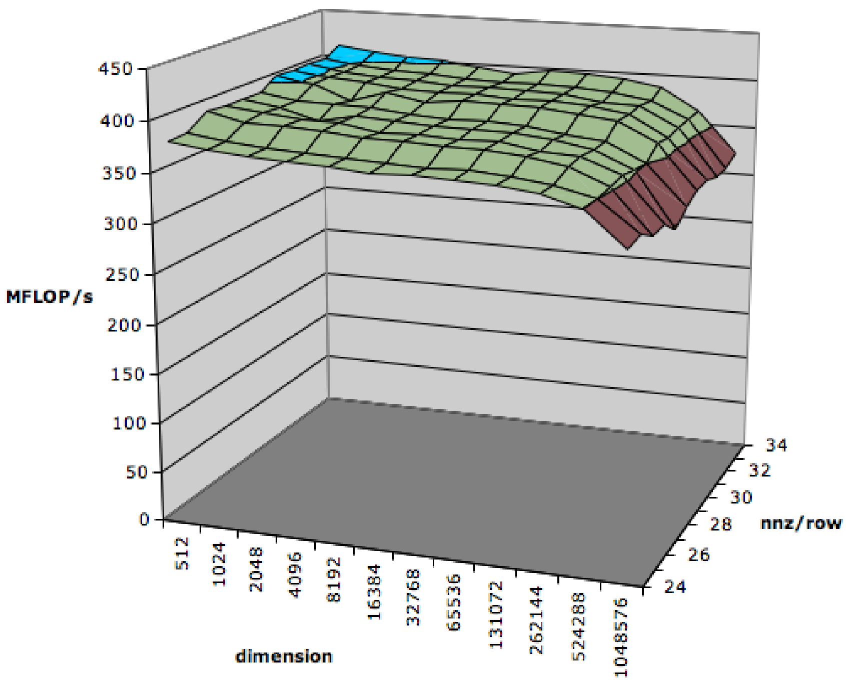
1x2 Benchmark Data, Itanium 2



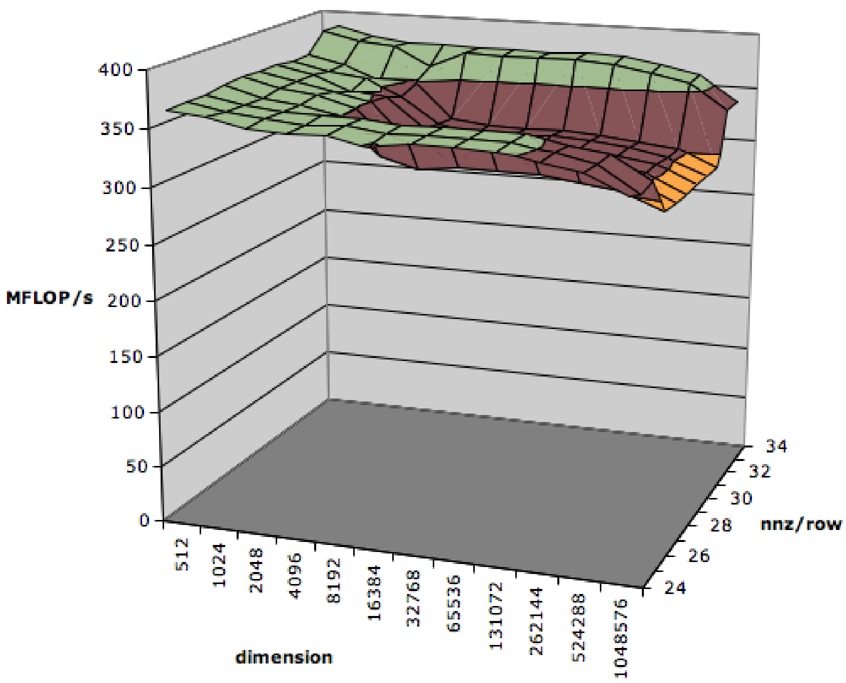
1x3 Benchmark Data, Itanium 2

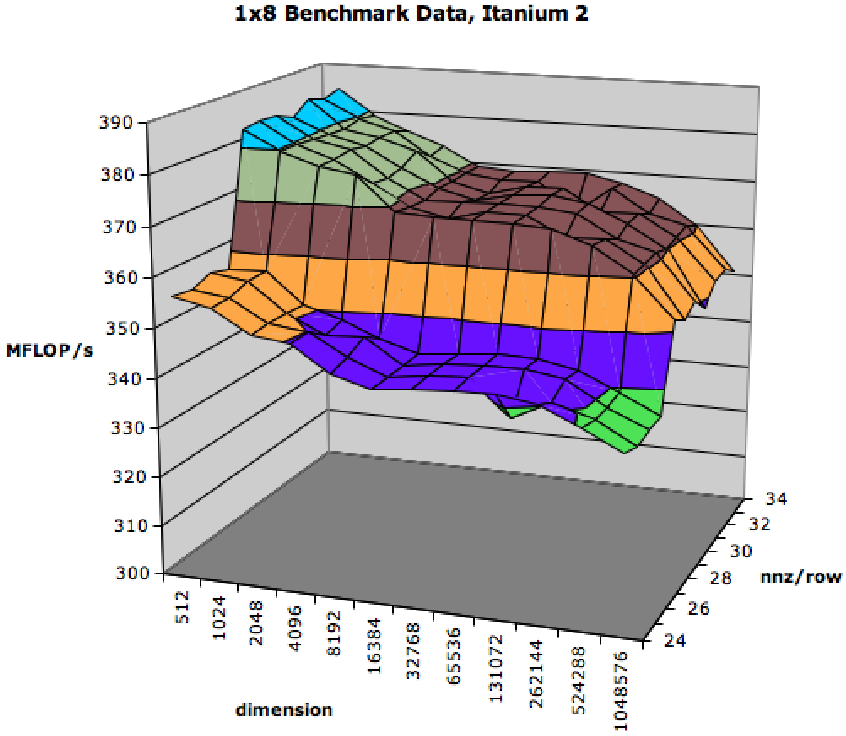


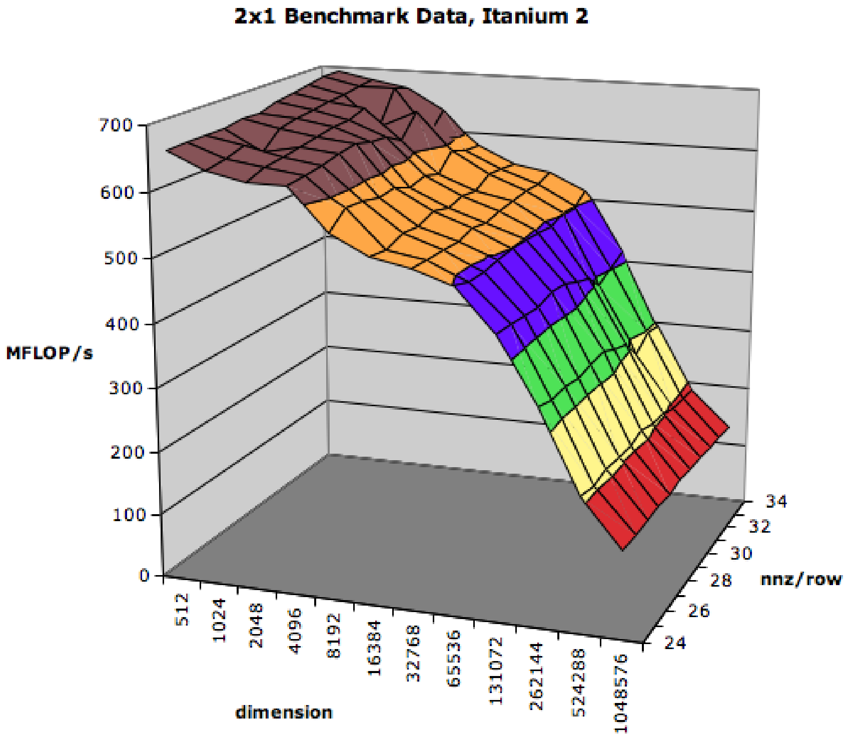
1x4 Benchmark Data, Itanium 2



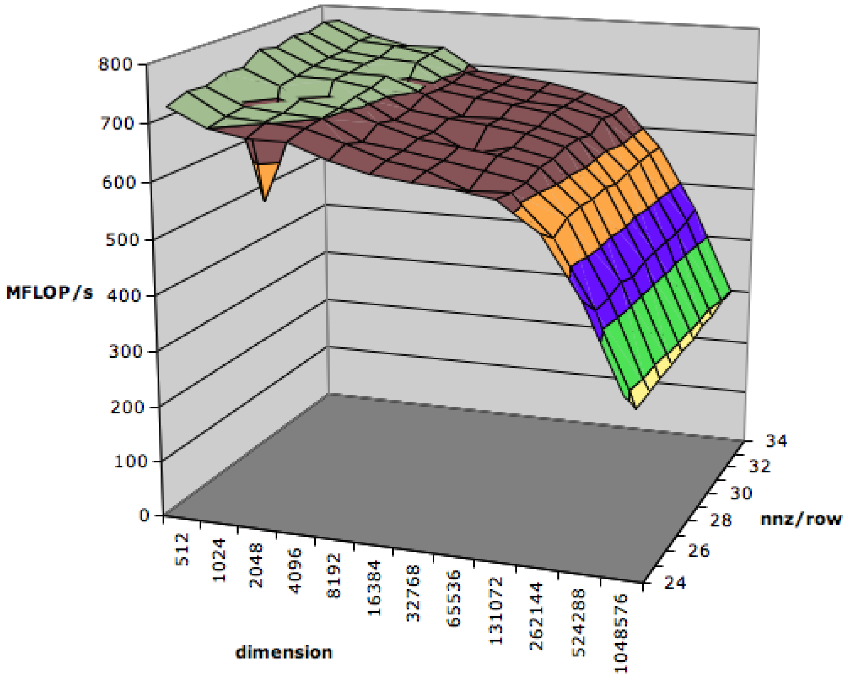
1x6 Benchmark Numbers, Itanium 2



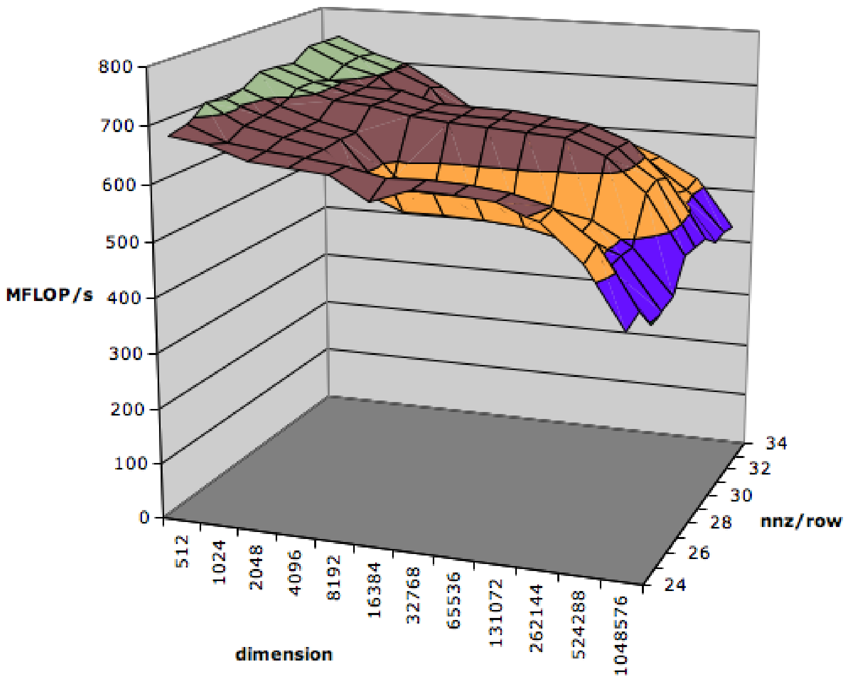

1x8 Benchmark Data, Itanium 2
MFLOP/s
390
380
370
360
350
340
330
320
310
300
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
nnz/row
24
26
28
30
32
34

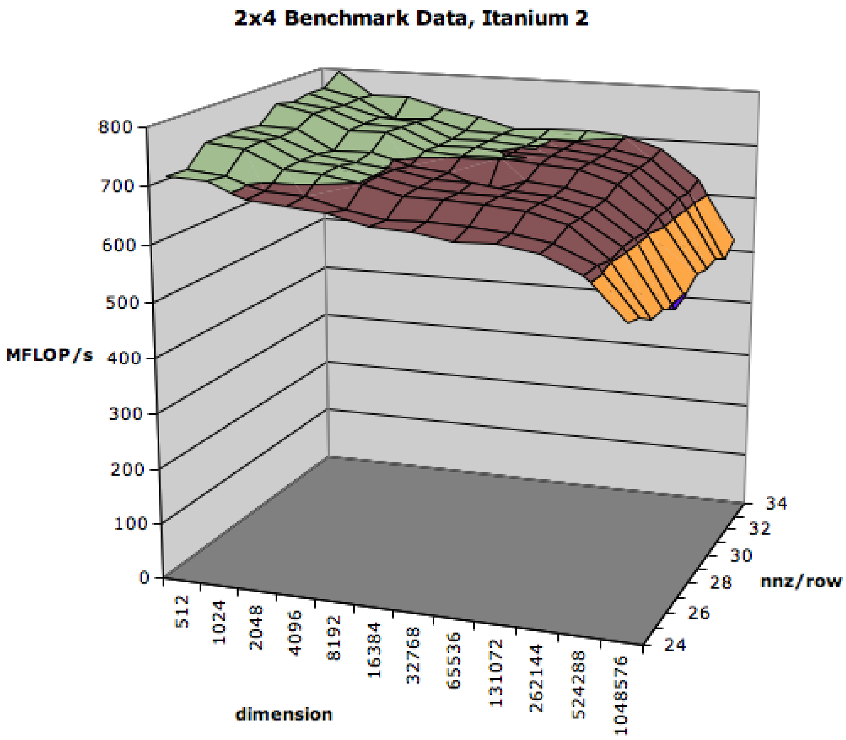

2x1 Benchmark Data, Itanium 2
MFLOP/s
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
nnz/row
24
26
28
30
32
34

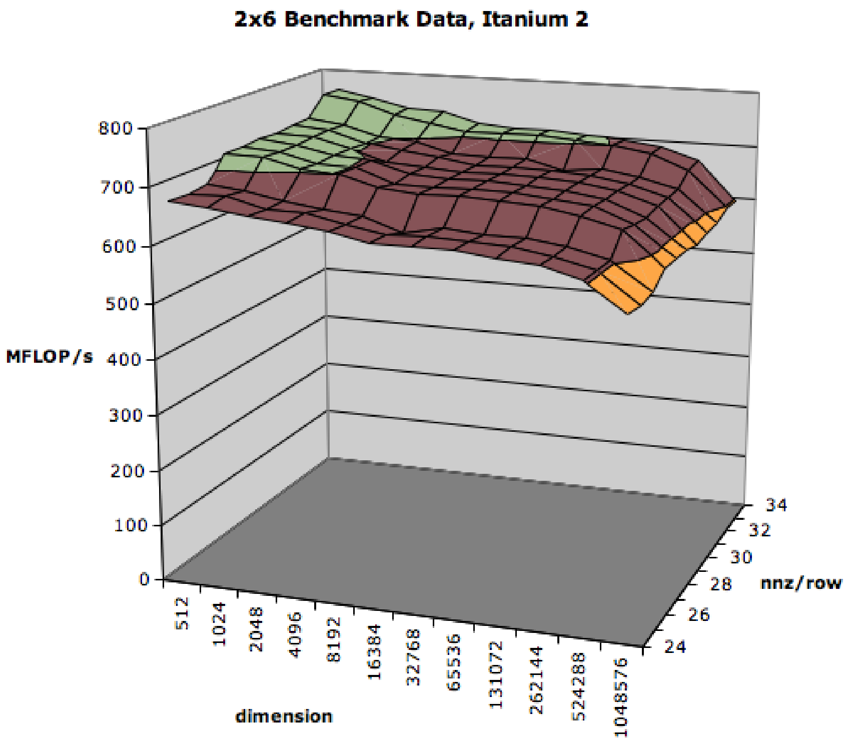
2x2 Benchmark Data, Itanium 2

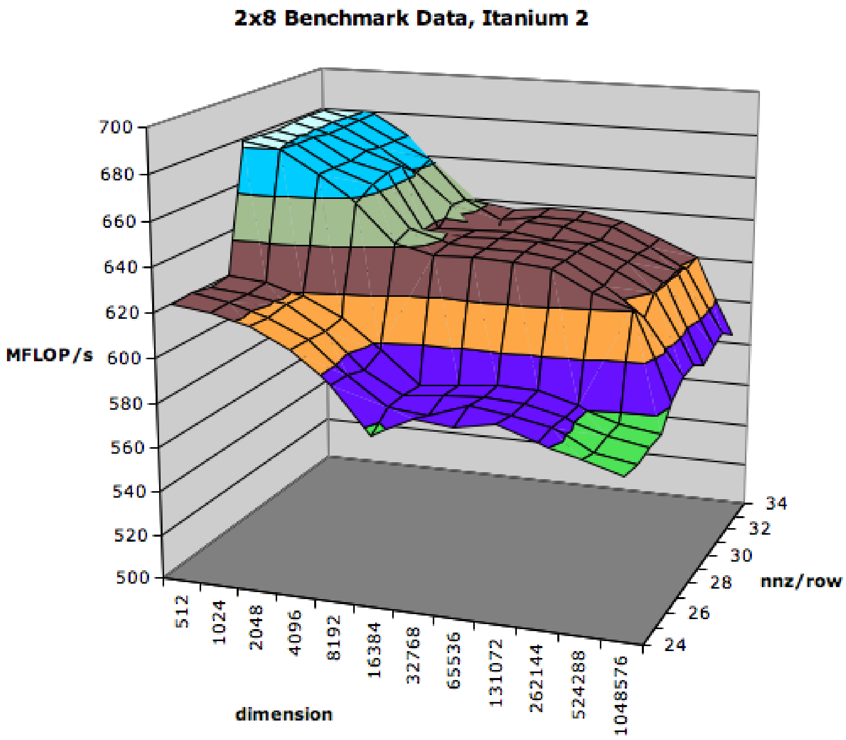


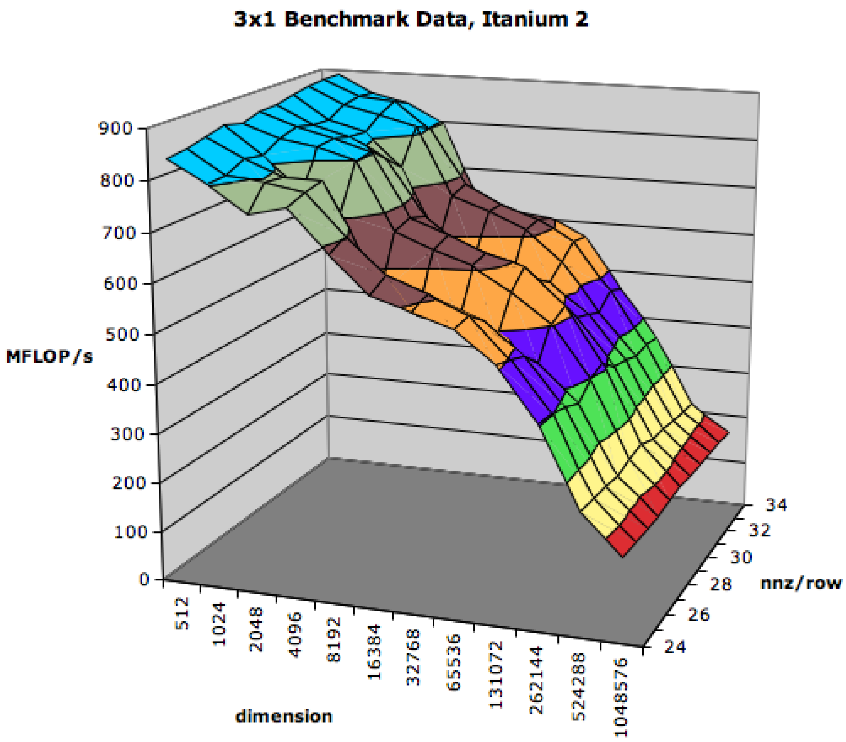
2x3 Benchmark Data, Itanium 2

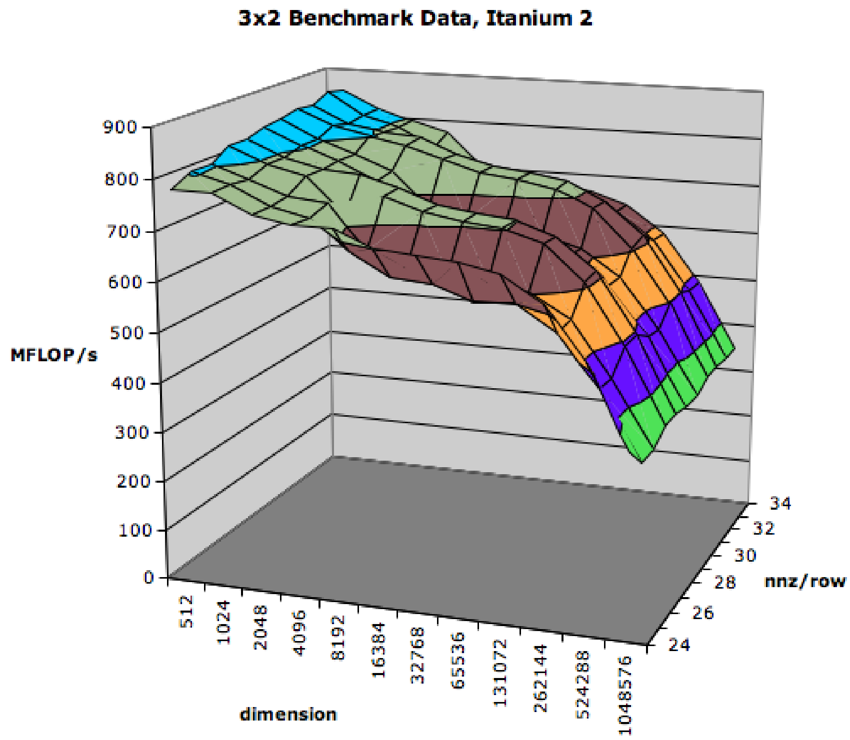


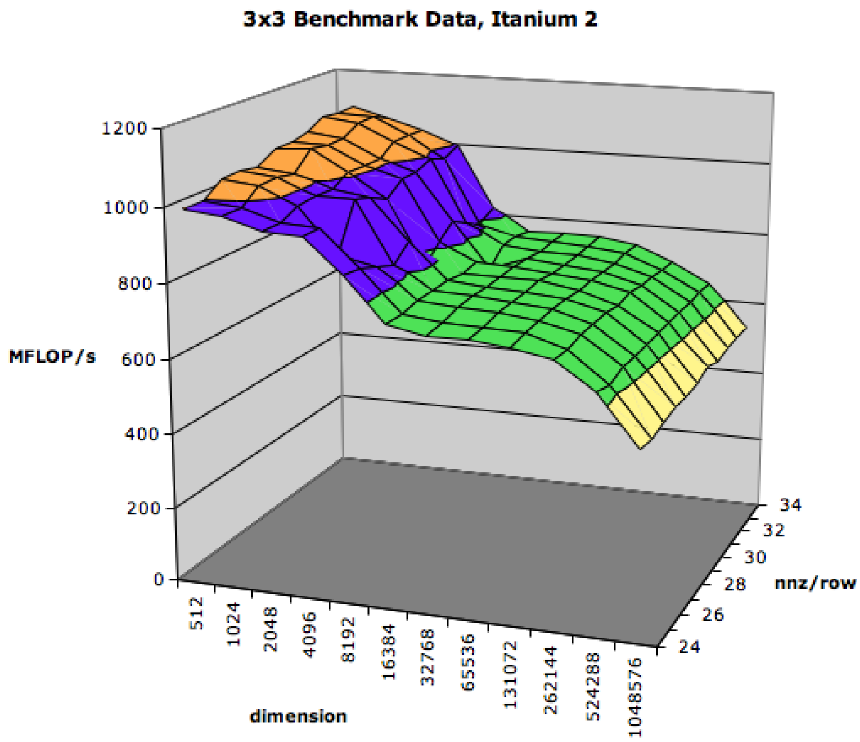

2x4 Benchmark Data, Itanium 2
MFLOP/s
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

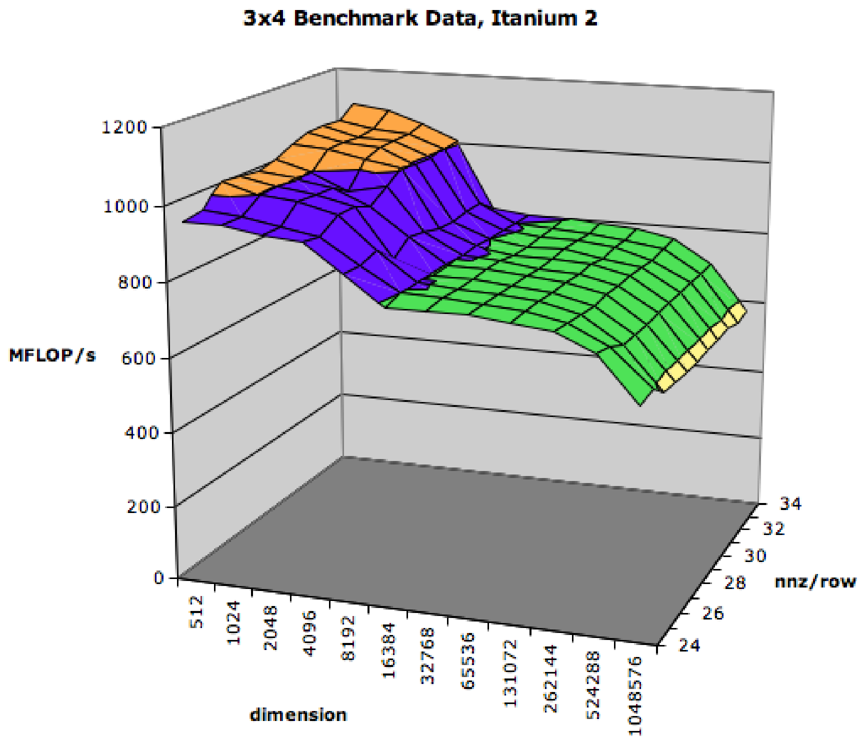

2x6 Benchmark Data, Itanium 2
MFLOP/s
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

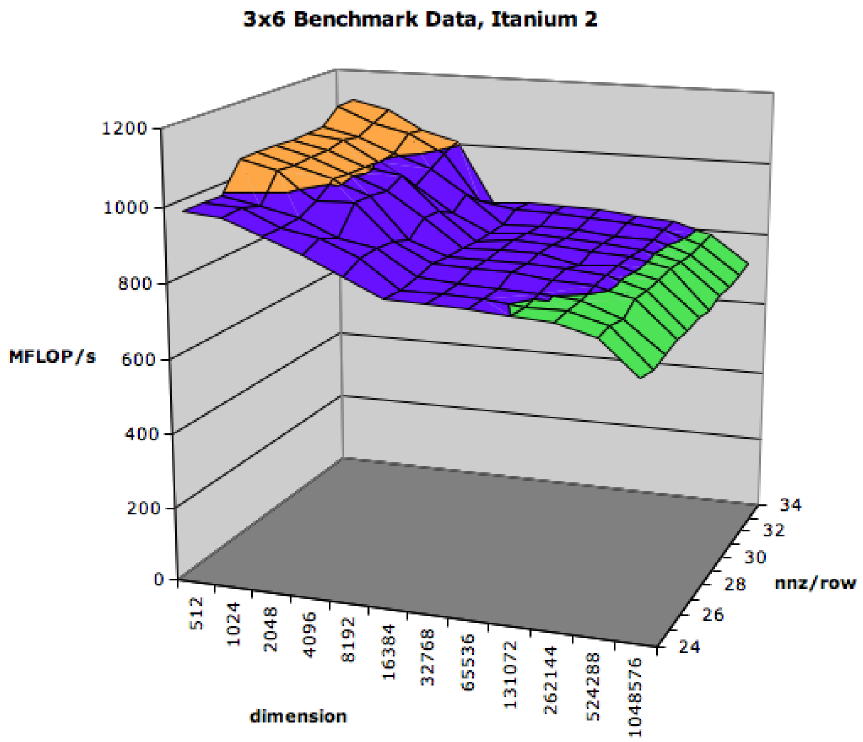

2x8 Benchmark Data, Itanium 2
MFLOP/s
700
680
660
640
620
600
580
560
540
520
500
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
nnz/row
24
26
28
30
32
34

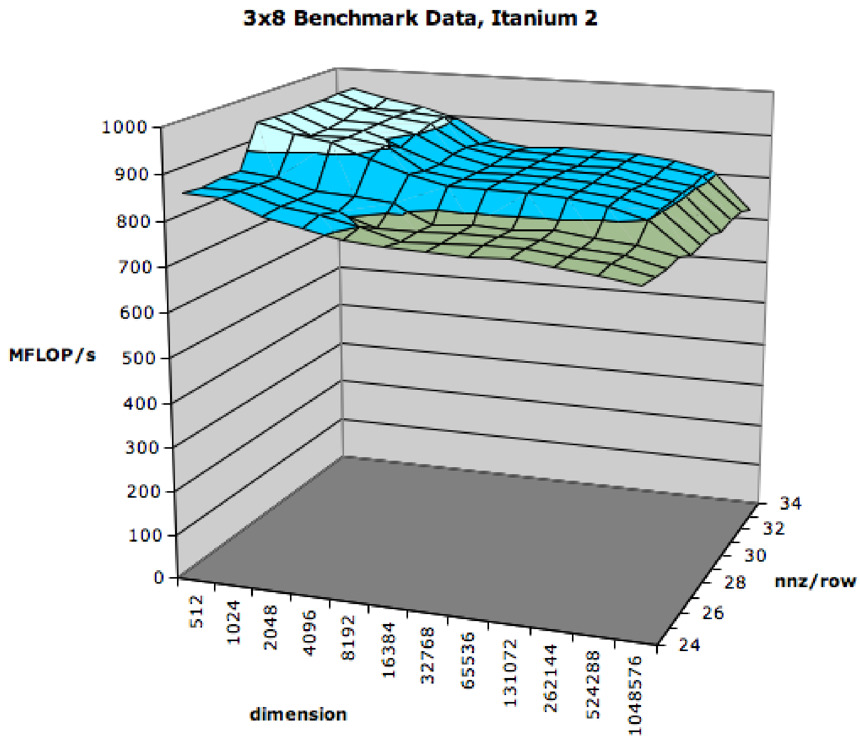

3x1 Benchmark Data, Itanium 2
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
nnz/row
24
26
28
30
32
34

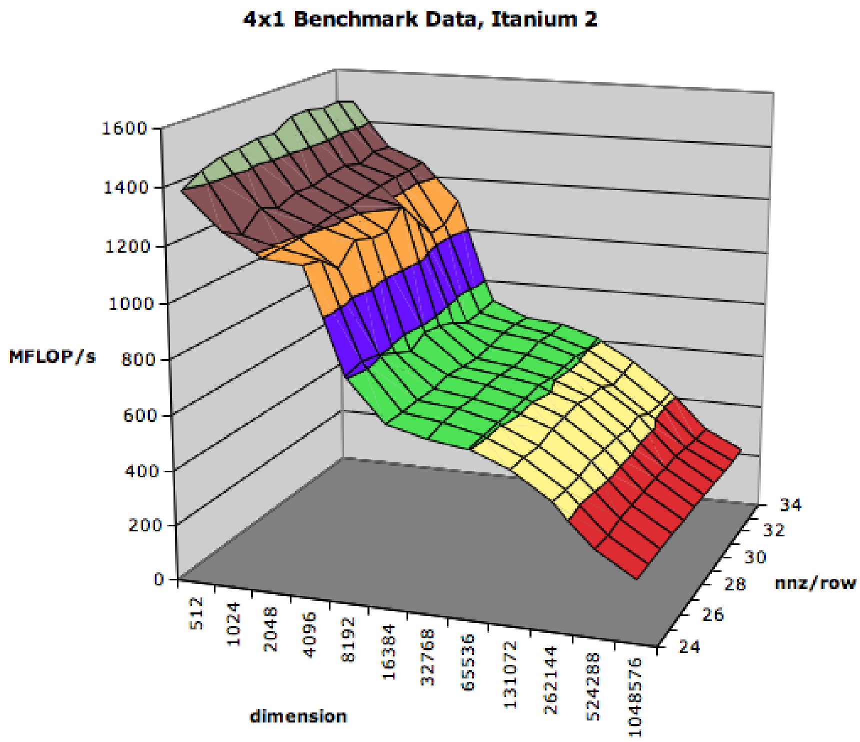

3x2 Benchmark Data, Itanium 2
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

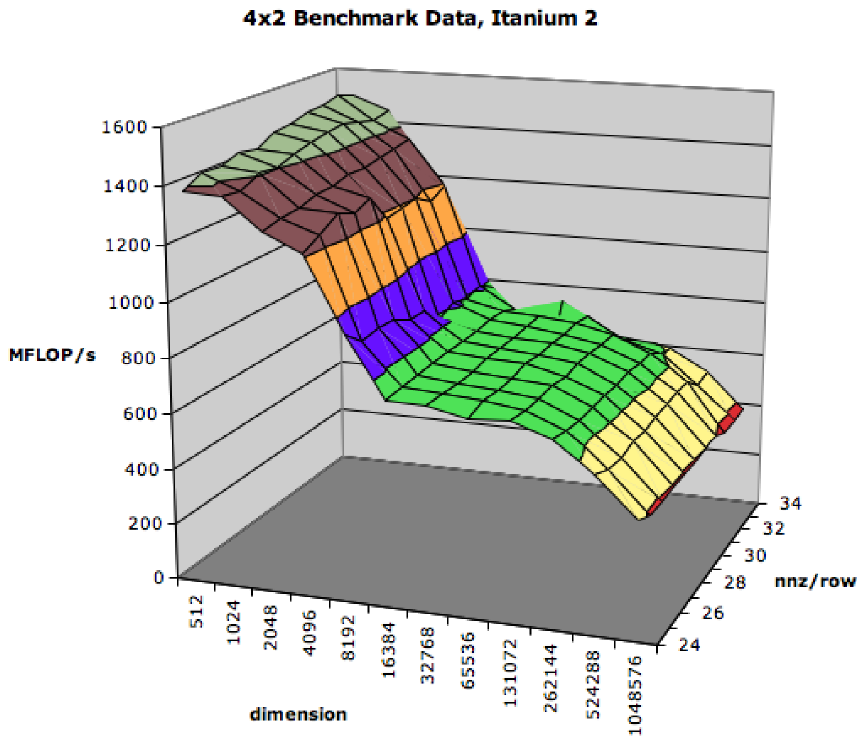

3x3 Benchmark Data, Itanium 2
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

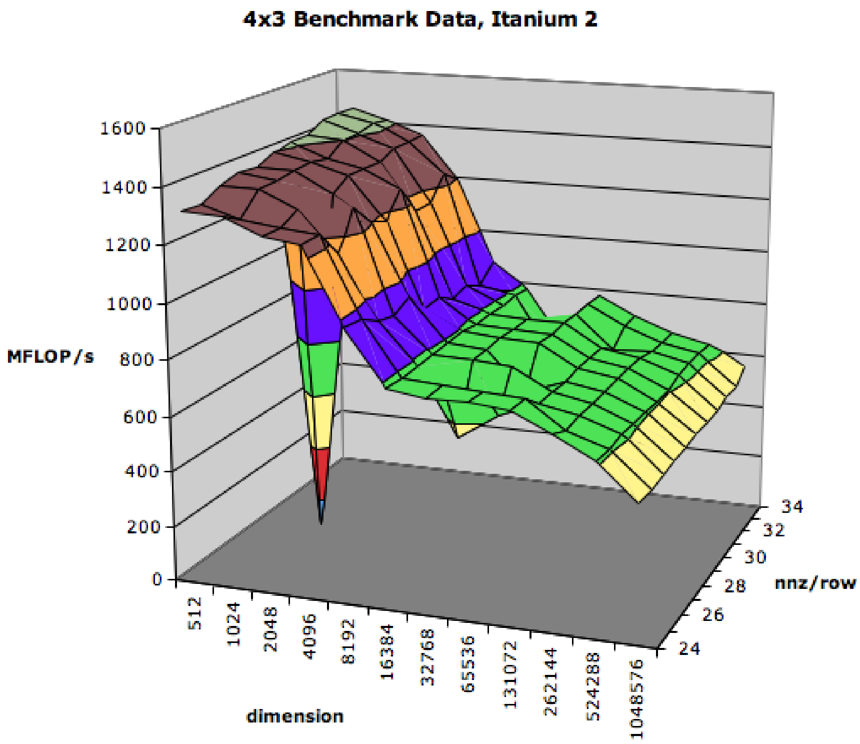

3x4 Benchmark Data, Itanium 2
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

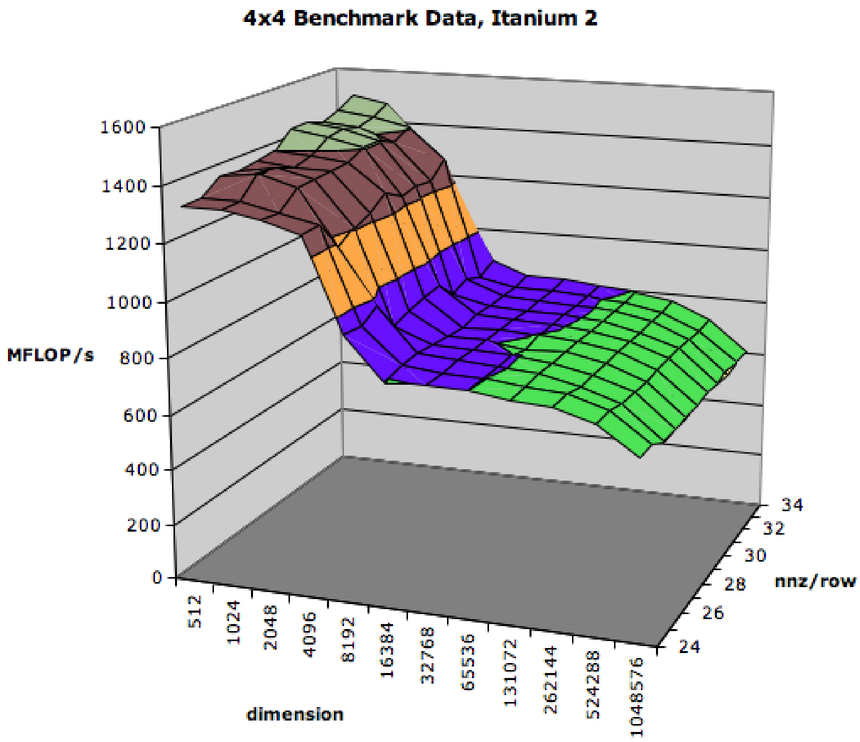

3x6 Benchmark Data, Itanium 2
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

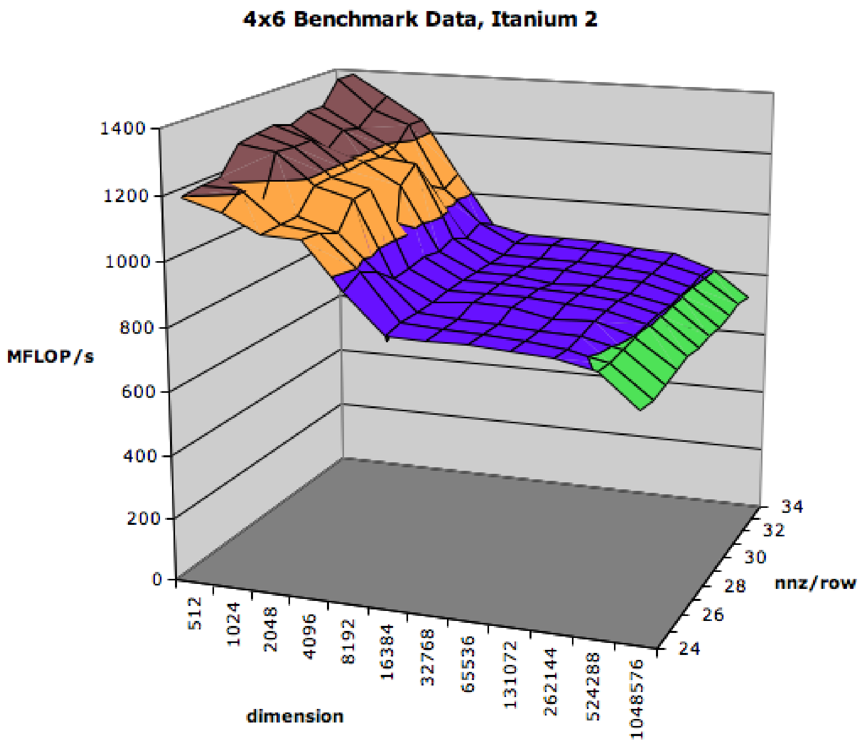

3x8 Benchmark Data, Itanium 2
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

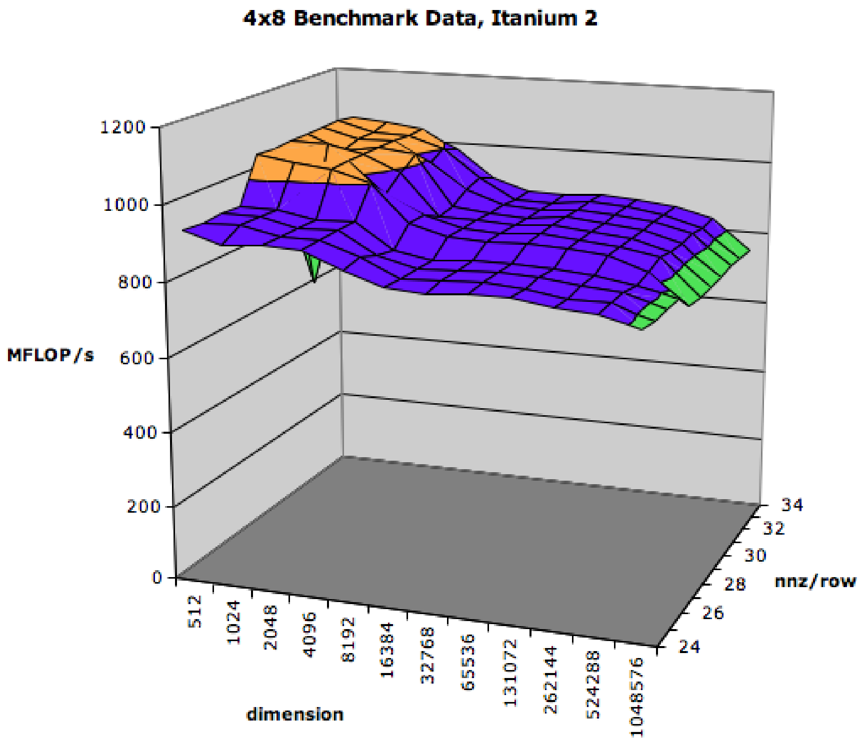

4x1 Benchmark Data, Itanium 2
MFLOP/s
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

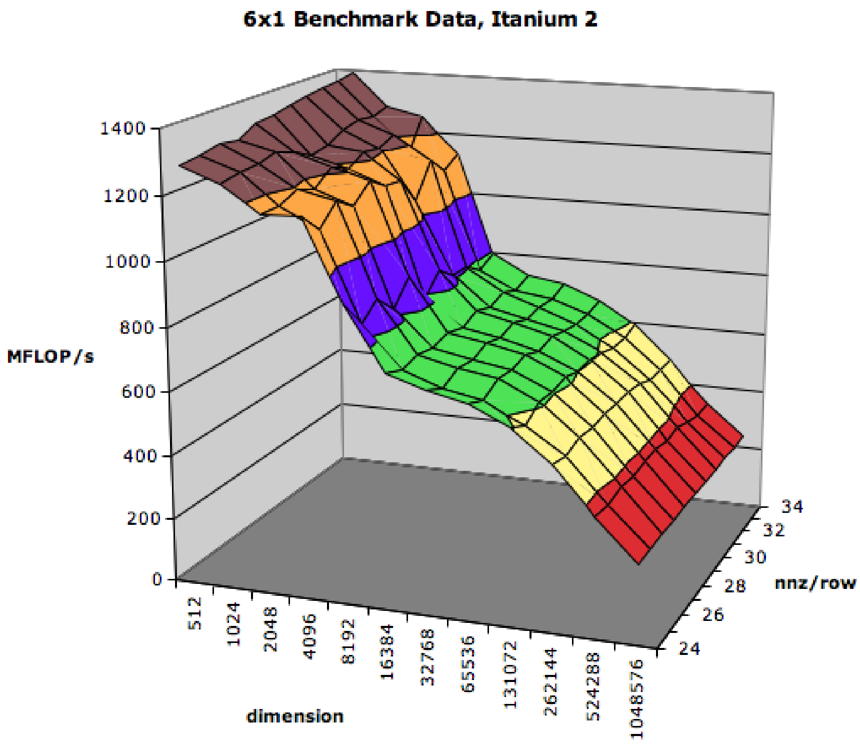

4x2 Benchmark Data, Itanium 2
MFLOP/s
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

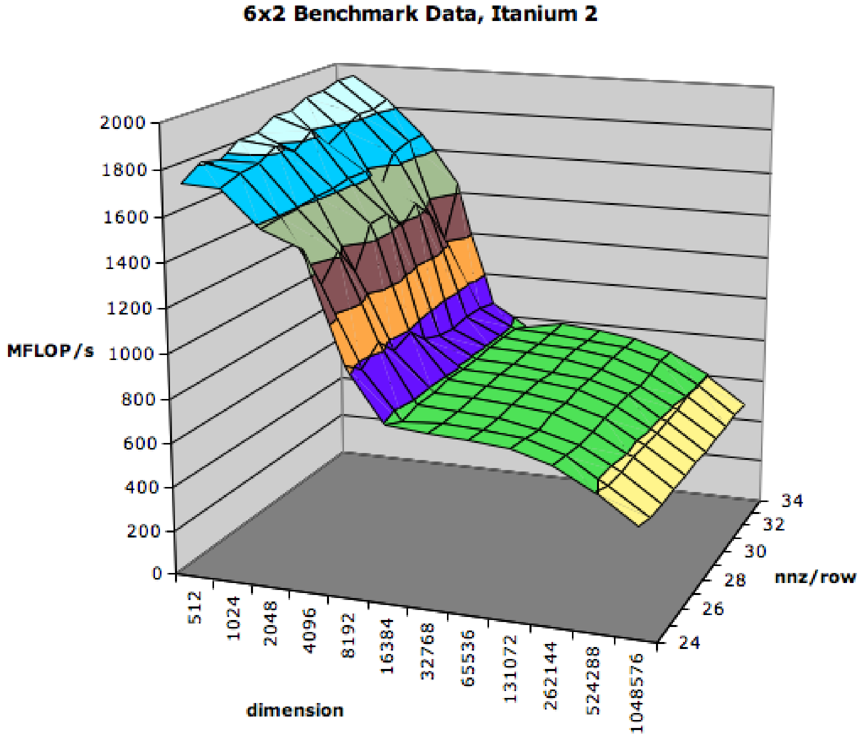

4x3 Benchmark Data, Itanium 2
MFLOP/s
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

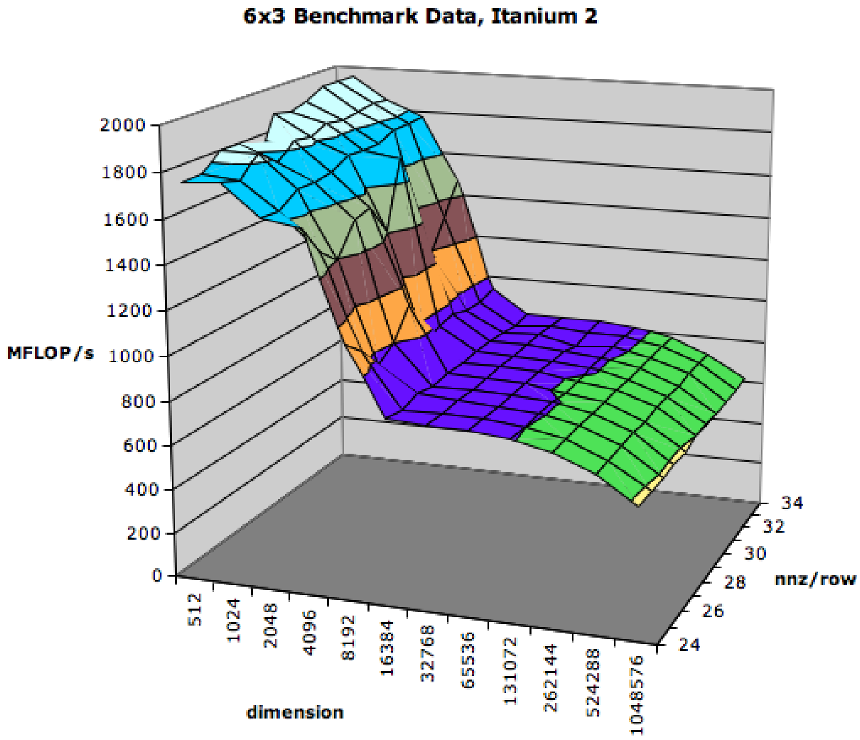

4x4 Benchmark Data, Itanium 2
MFLOP/s
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

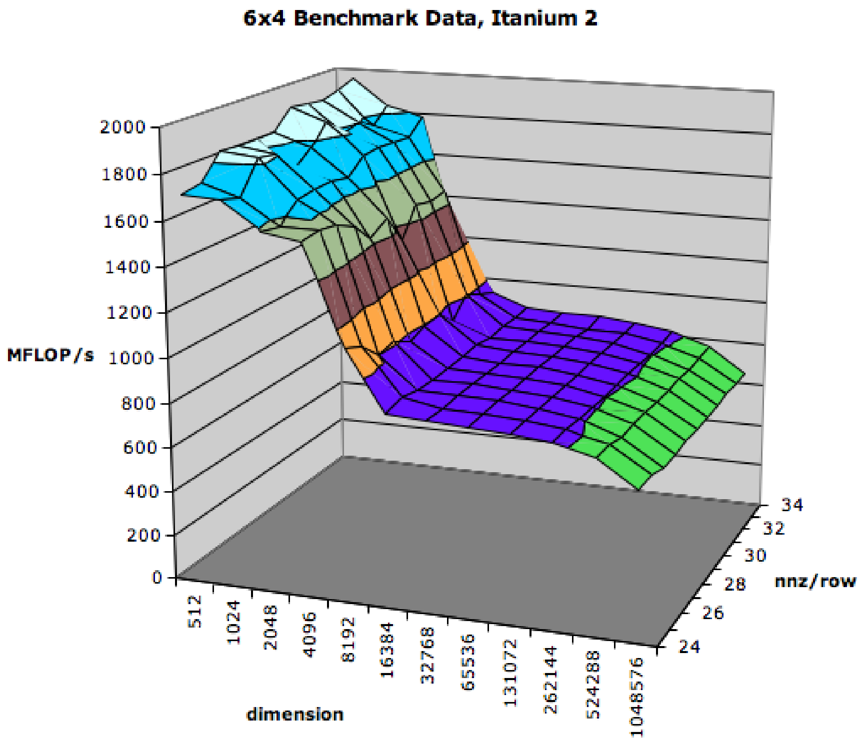

4x6 Benchmark Data, Itanium 2
MFLOP/s
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

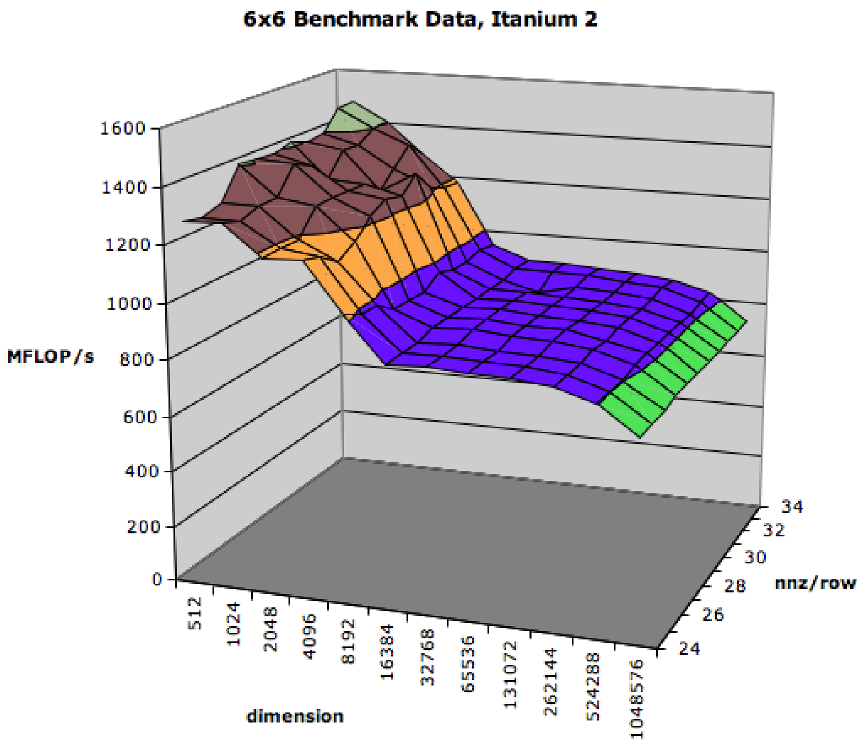

4x8 Benchmark Data, Itanium 2
MFLOP/s
0
200
400
600
800
1000
1200
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

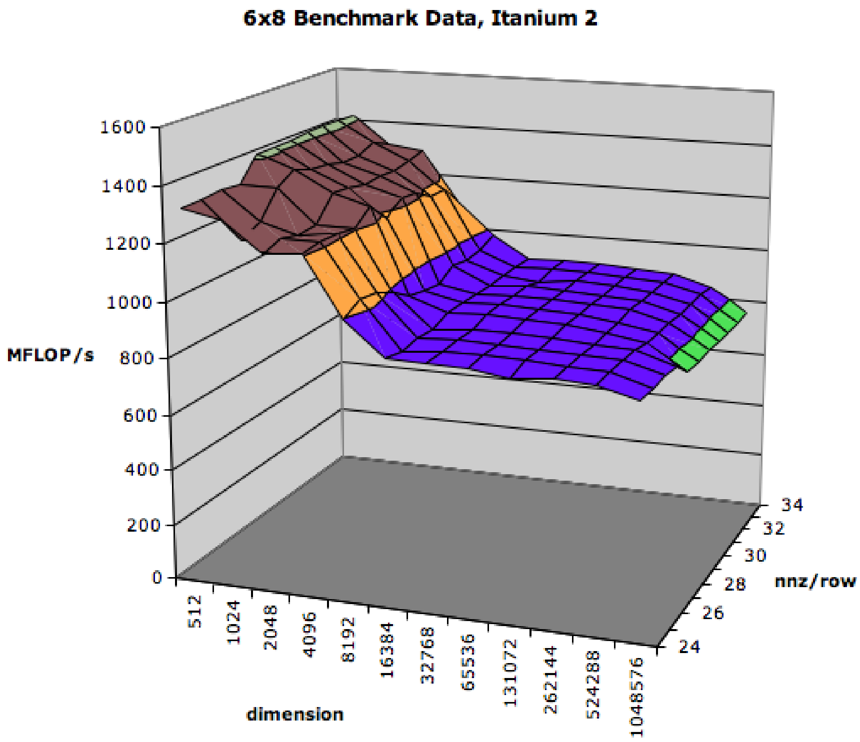

6x1 Benchmark Data, Itanium 2
MFLOP/s
0
200
400
600
800
1000
1200
1400
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

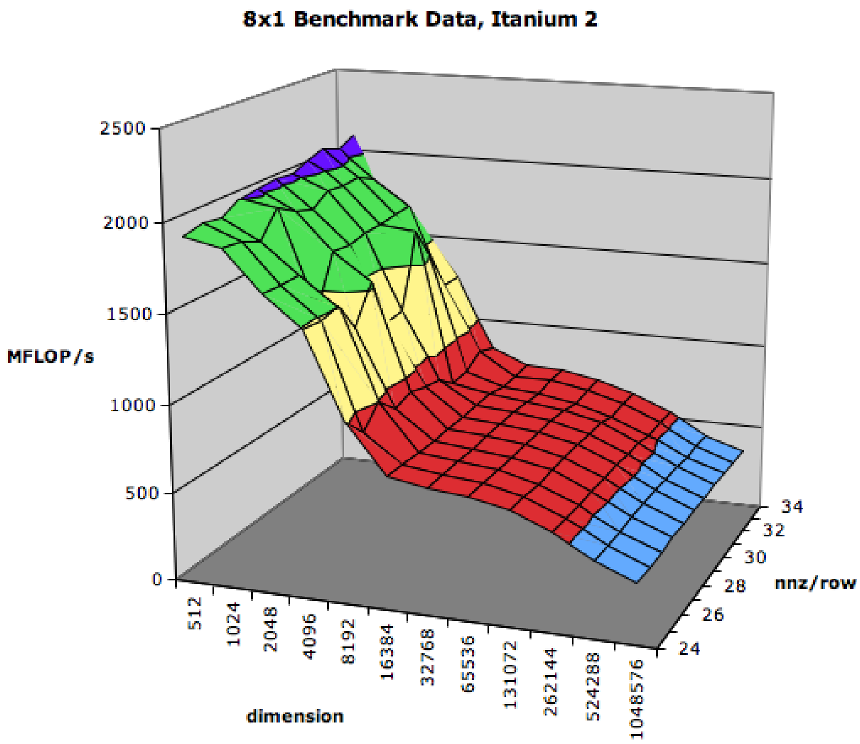

6x2 Benchmark Data, Itanium 2
MFLOP/s
2000
1800
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row


6x3 Benchmark Data, Itanium 2
MFLOP/s
2000
1800
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

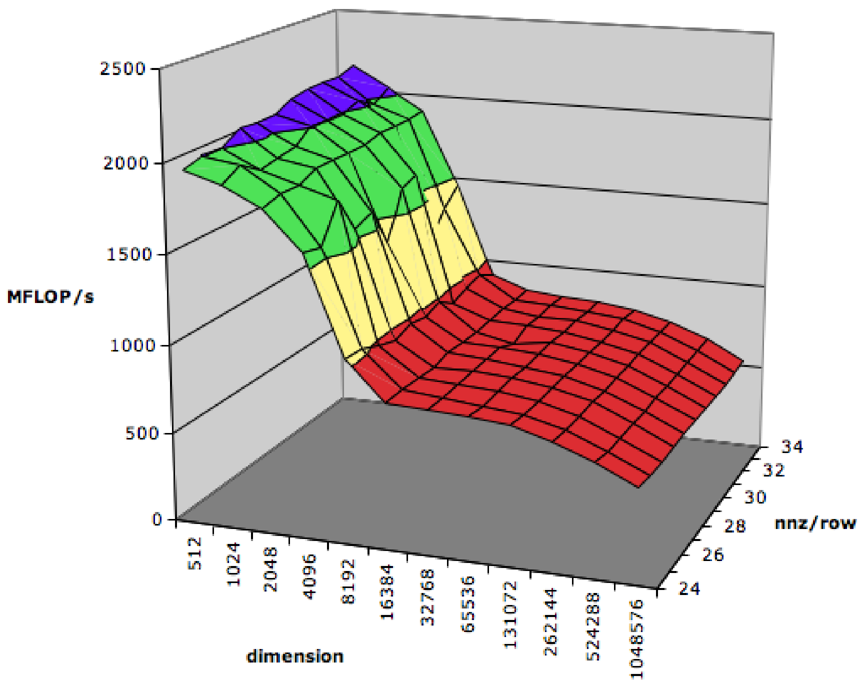

6x4 Benchmark Data, Itanium 2
MFLOP/s
2000
1800
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row


6x6 Benchmark Data, Itanium 2
MFLOP/s
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

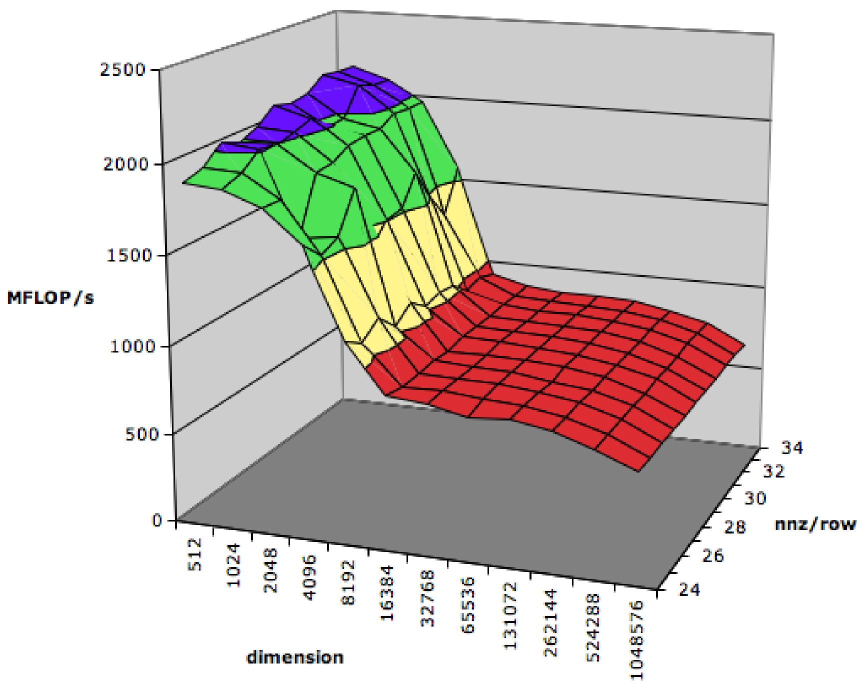

6x8 Benchmark Data, Itanium 2
MFLOP/s
1600
1400
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row


8x1 Benchmark Data, Itanium 2
MFLOP/s
2500
2000
1500
1000
500
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

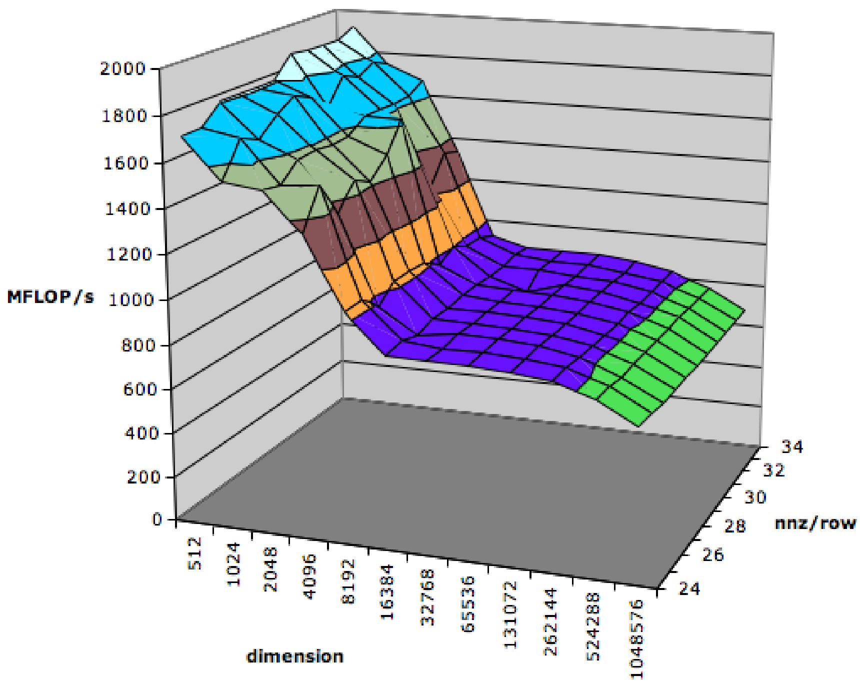
8x2 Benchmark Data, Itanium 2



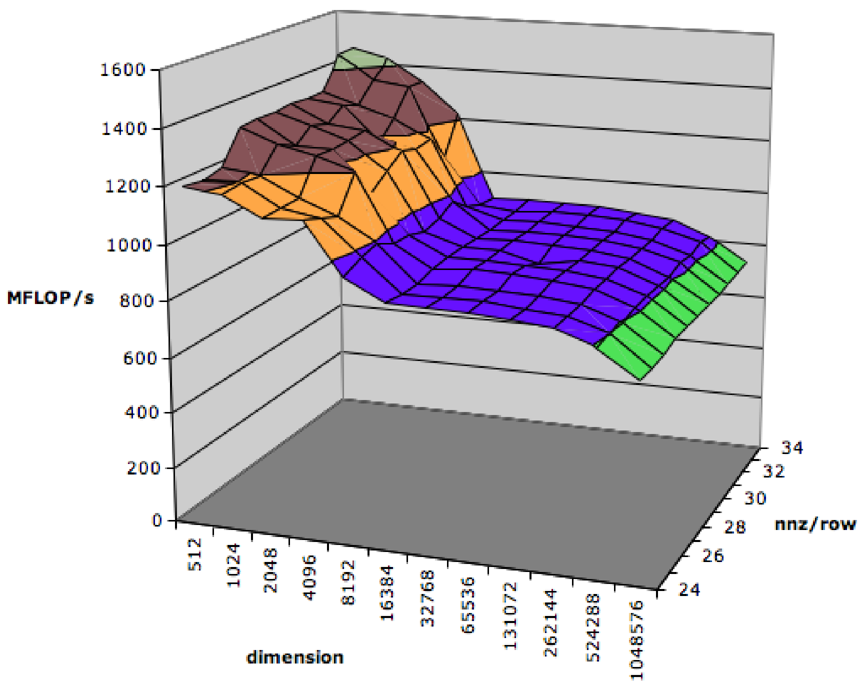
8x3 Benchmark Data, Itanium 2

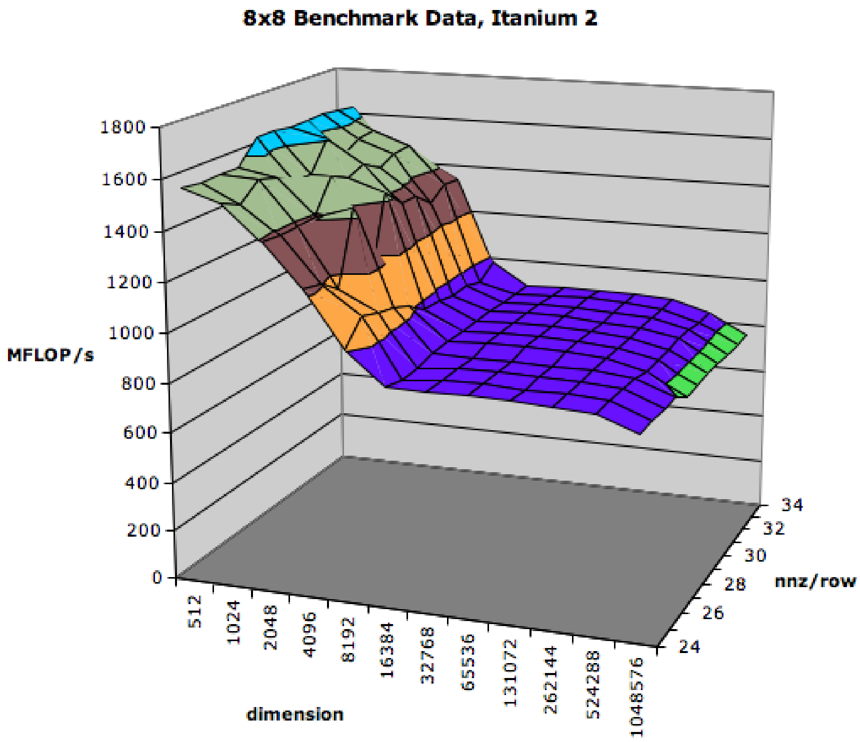


8x4 Benchmark Data, Itanium 2



8x6 Benchmark Data, Itanium 2

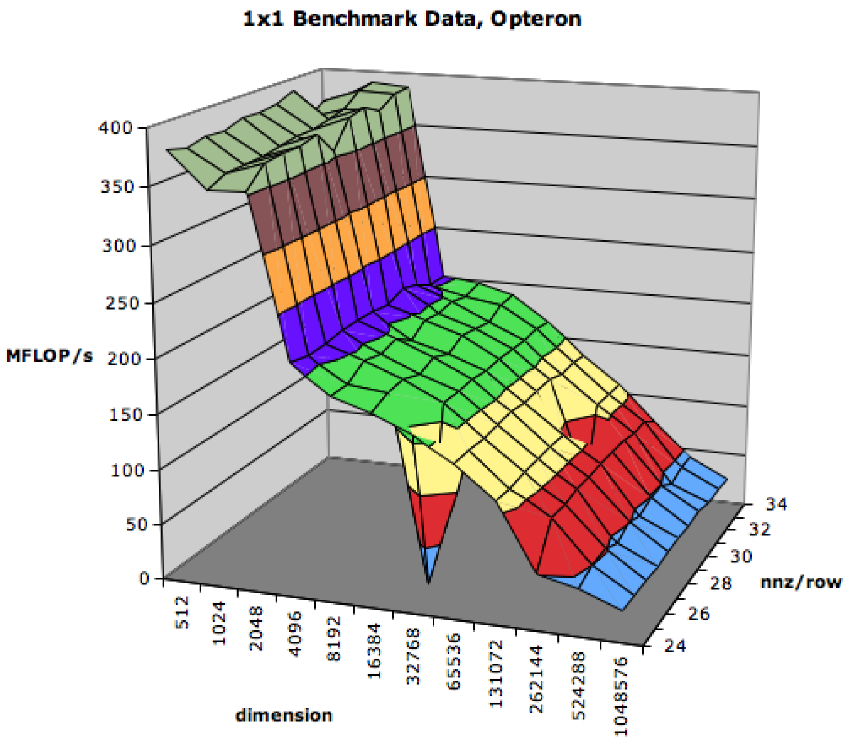



8x8 Benchmark Data, Itanium 2
MFLOP/s
0
200
400
600
800
1000
1200
1400
1600
1800
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

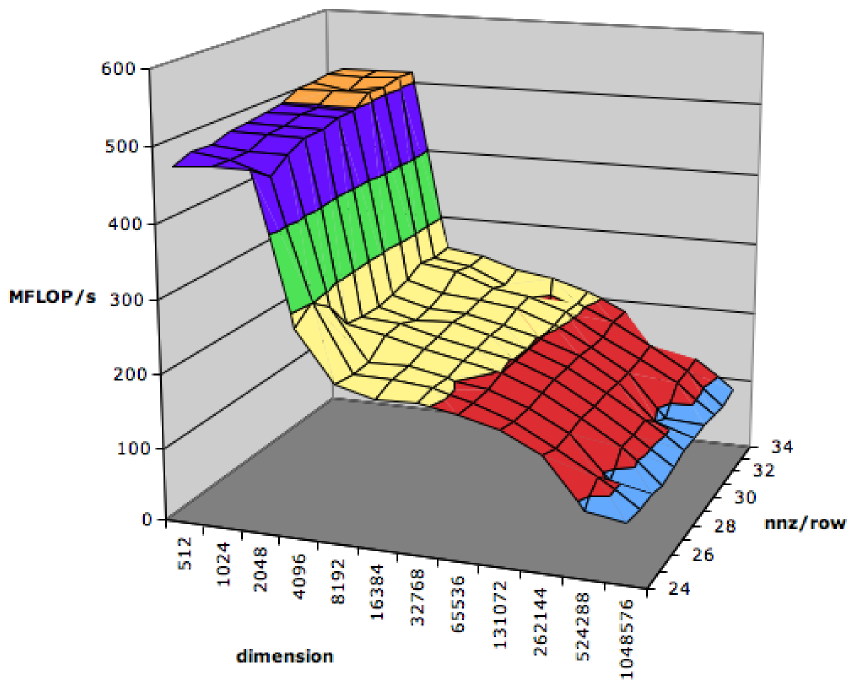
Appendix J

Opteron Benchmark Data

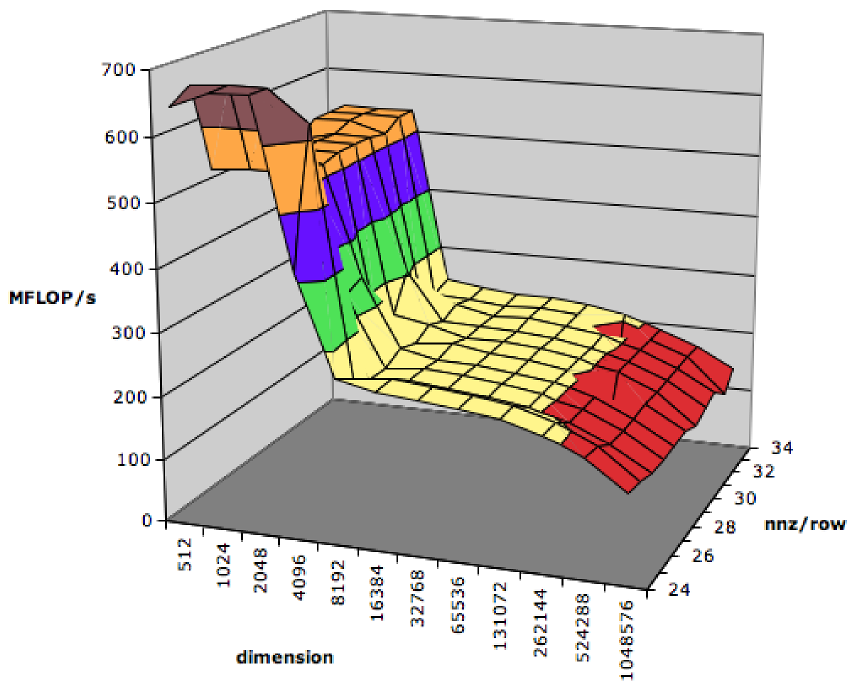
Here we graphically present the full output of our benchmark on the Opteron.



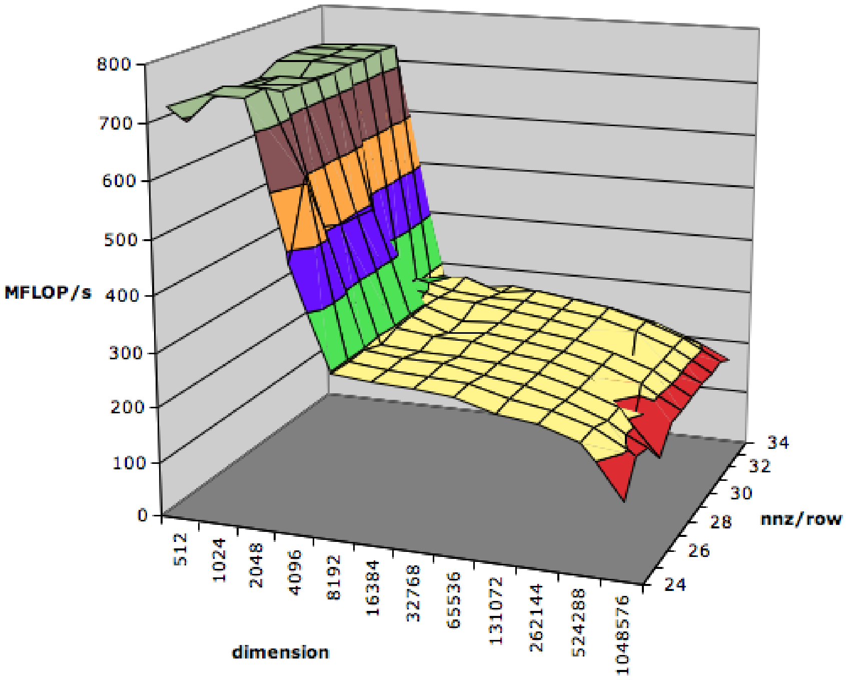
1x2 Benchmark Data, Opteron



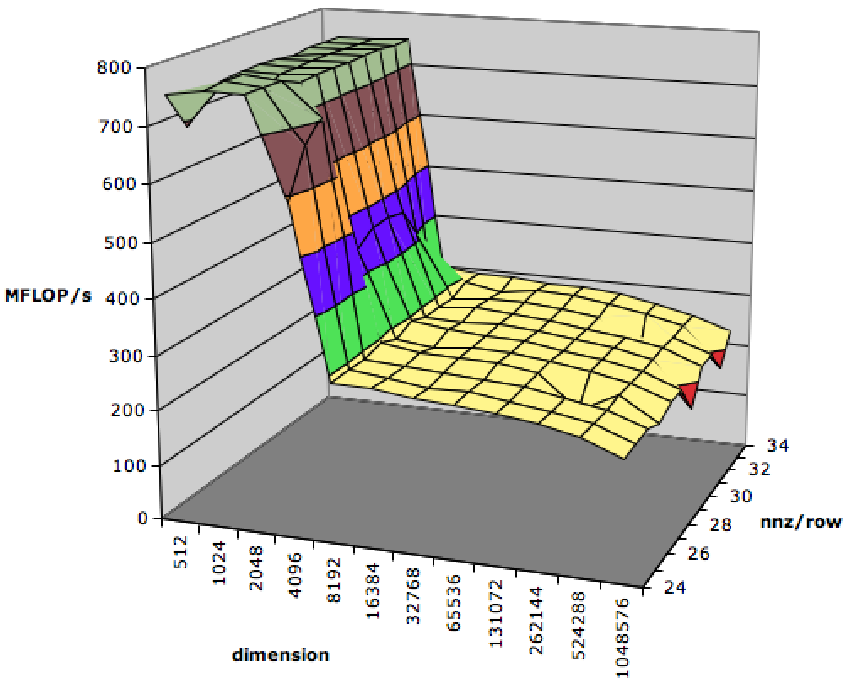
1x3 Benchmark Data, Opteron

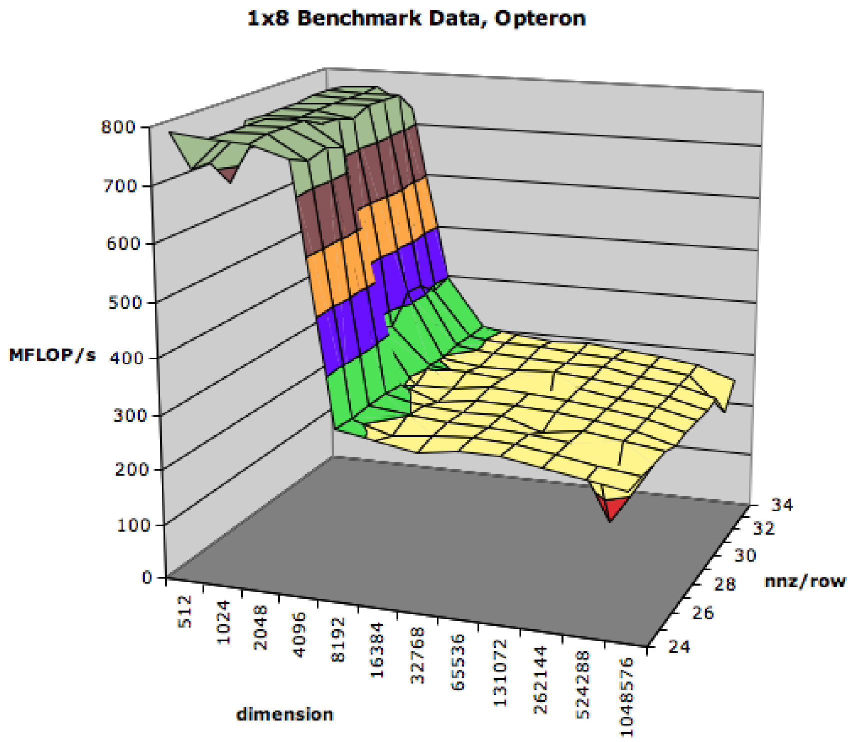


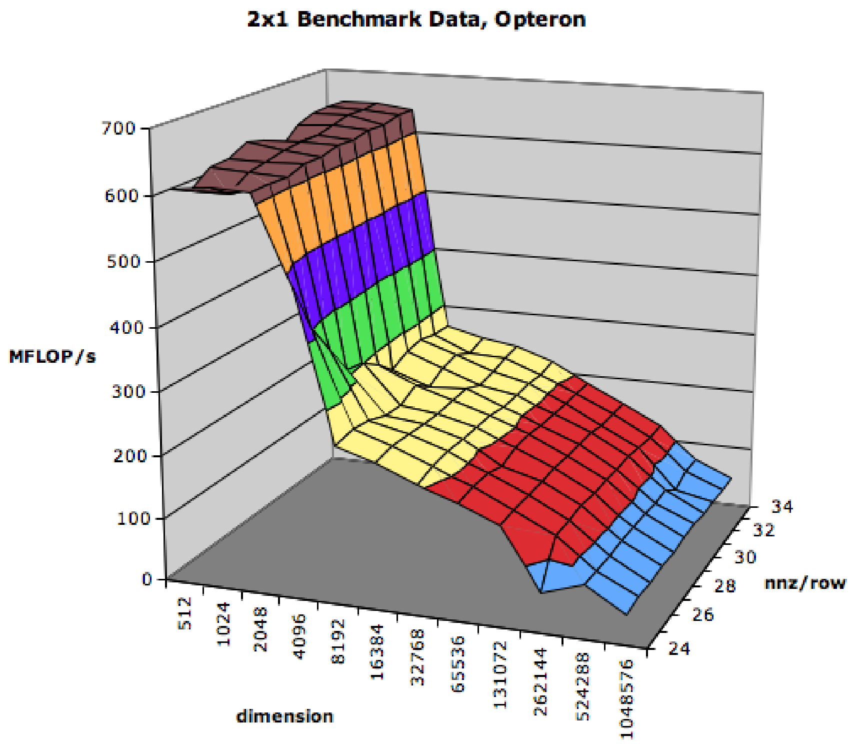
1x4 Benchmark Data, Opteron



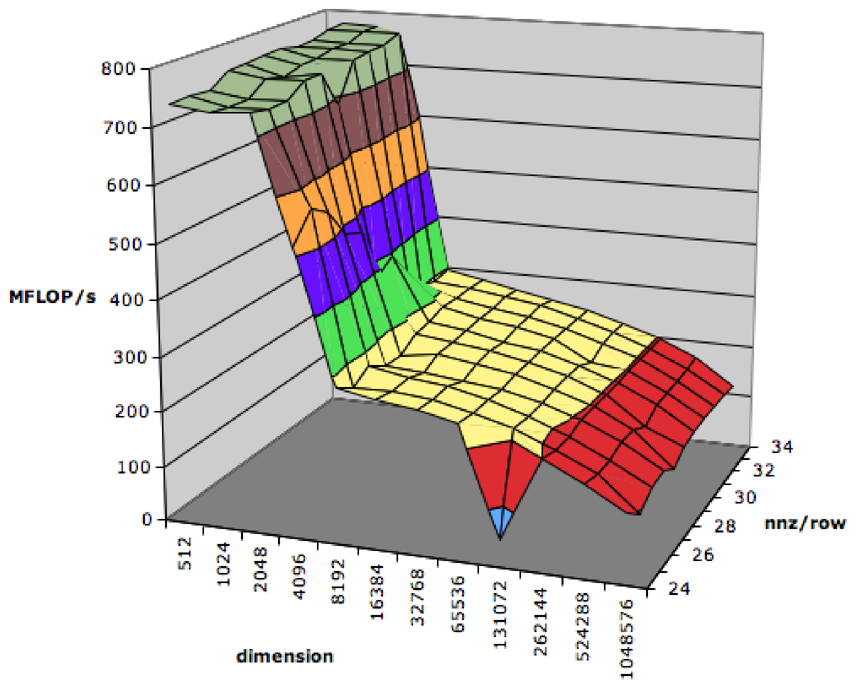
1x6 Benchmark Data, Opteron



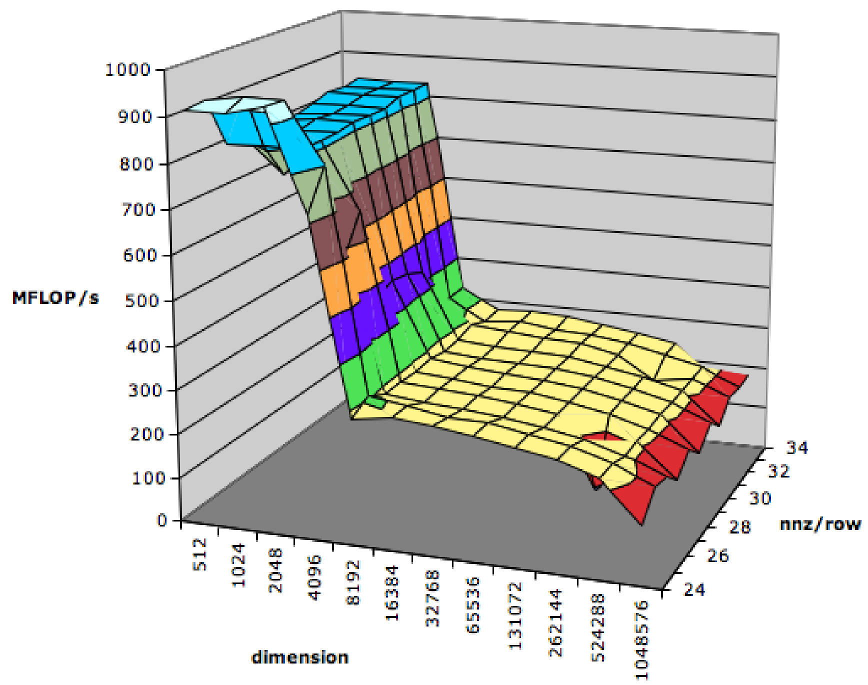

1x8 Benchmark Data, Opteron
MFLOP/s
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

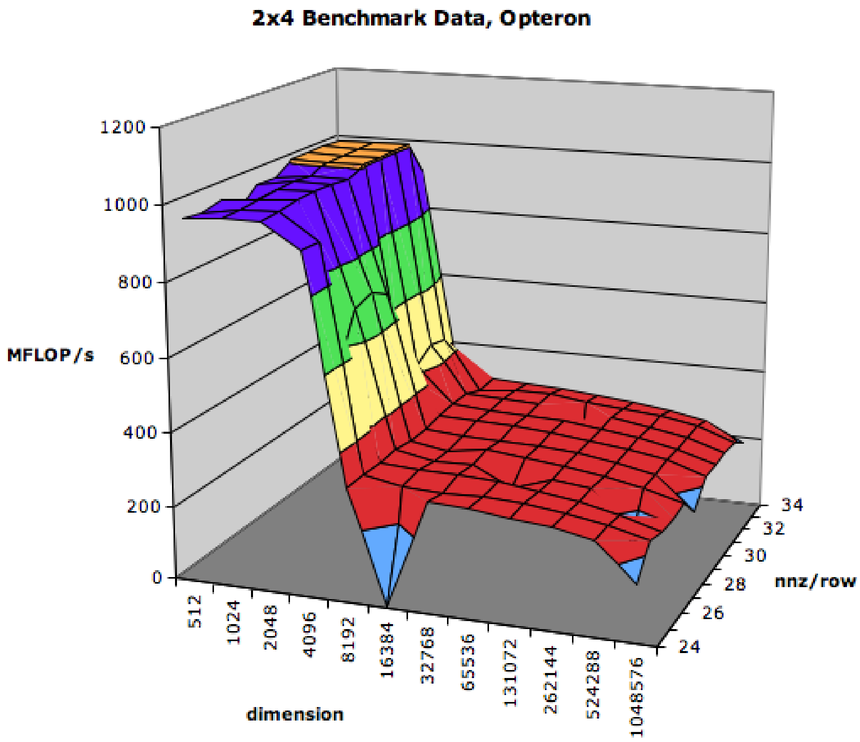

2x1 Benchmark Data, Opteron
MFLOP/s
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

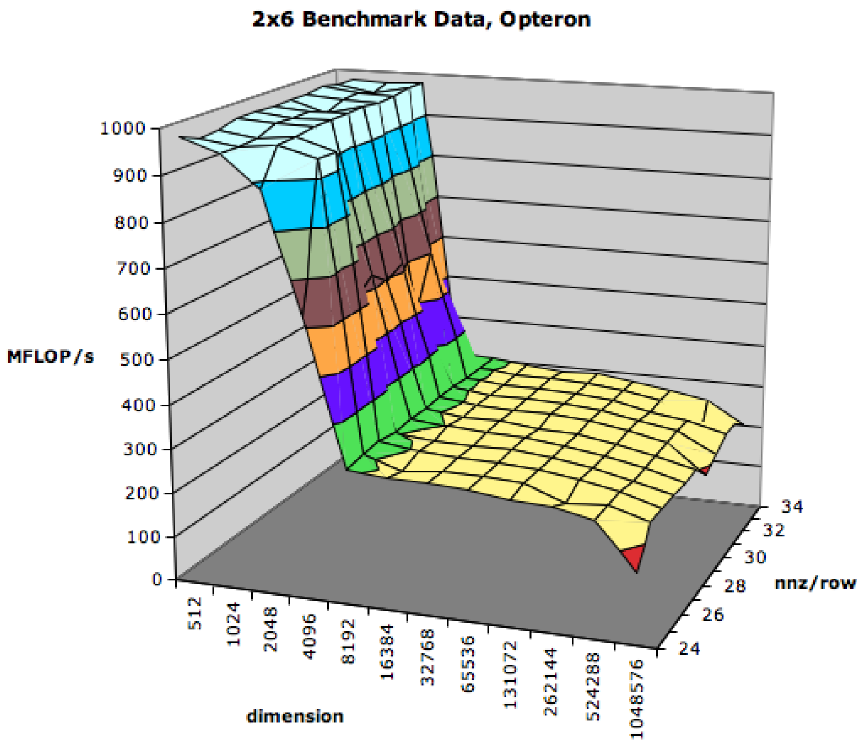
2x2 Benchmark Data, Opteron

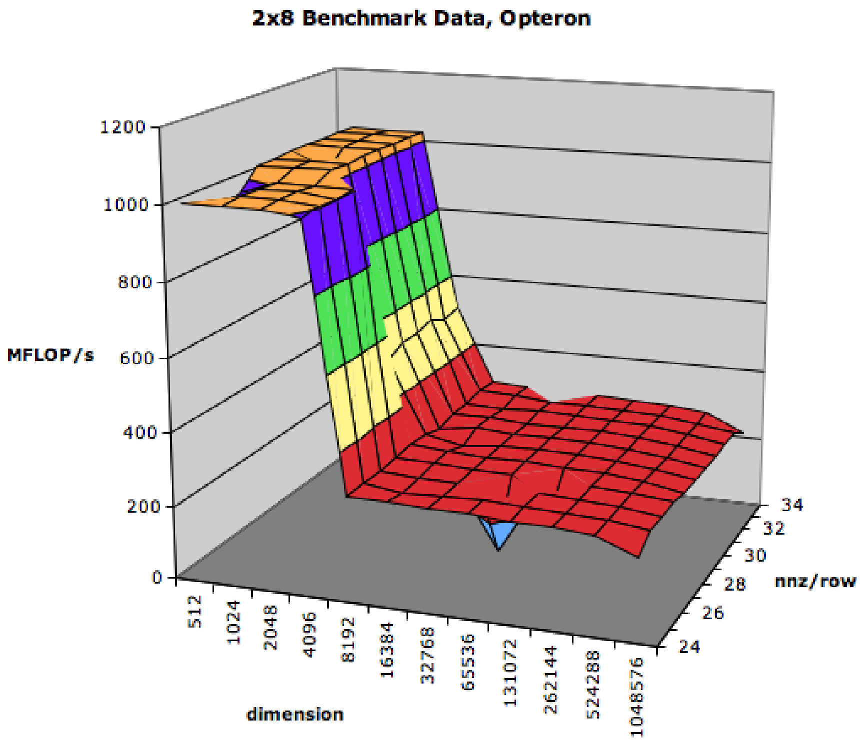


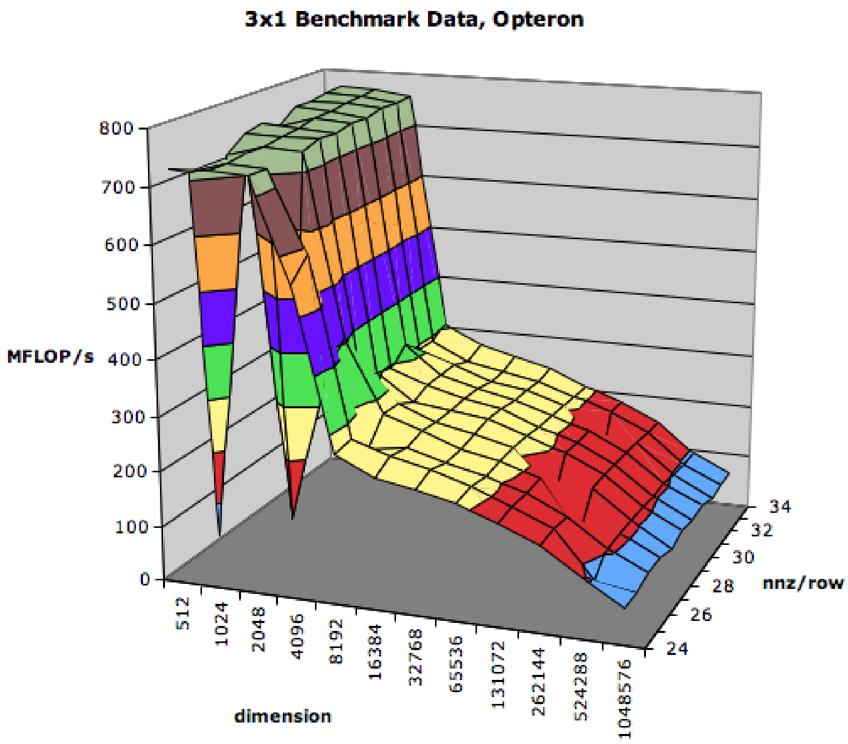
2x3 Benchmark Data, Opteron

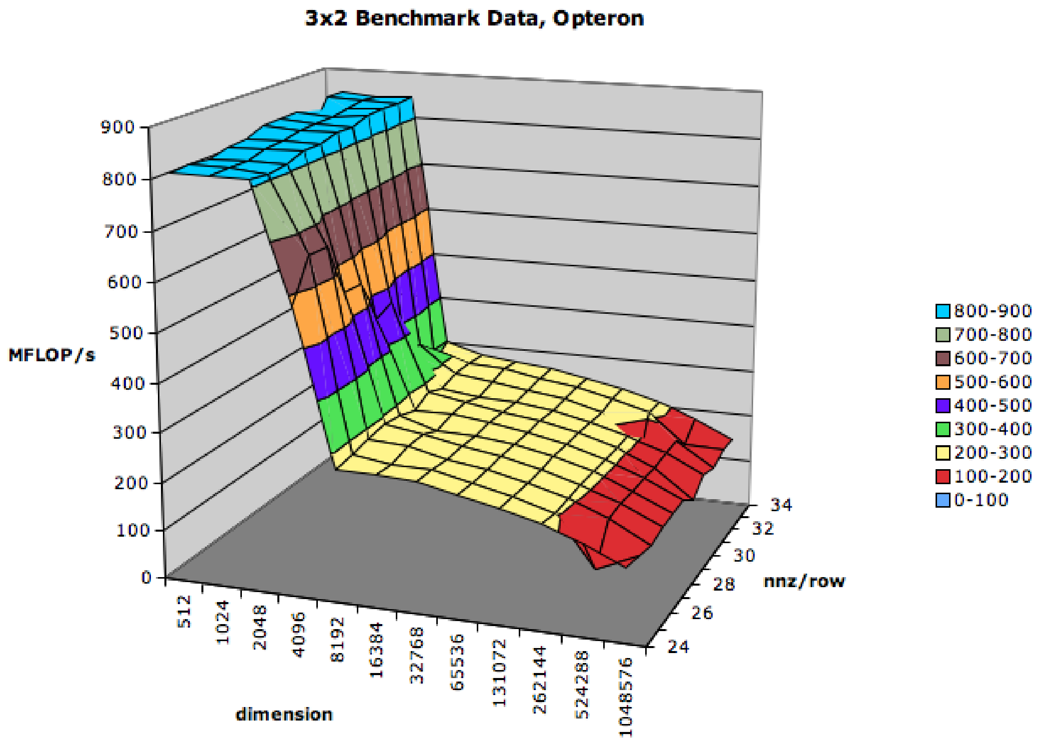


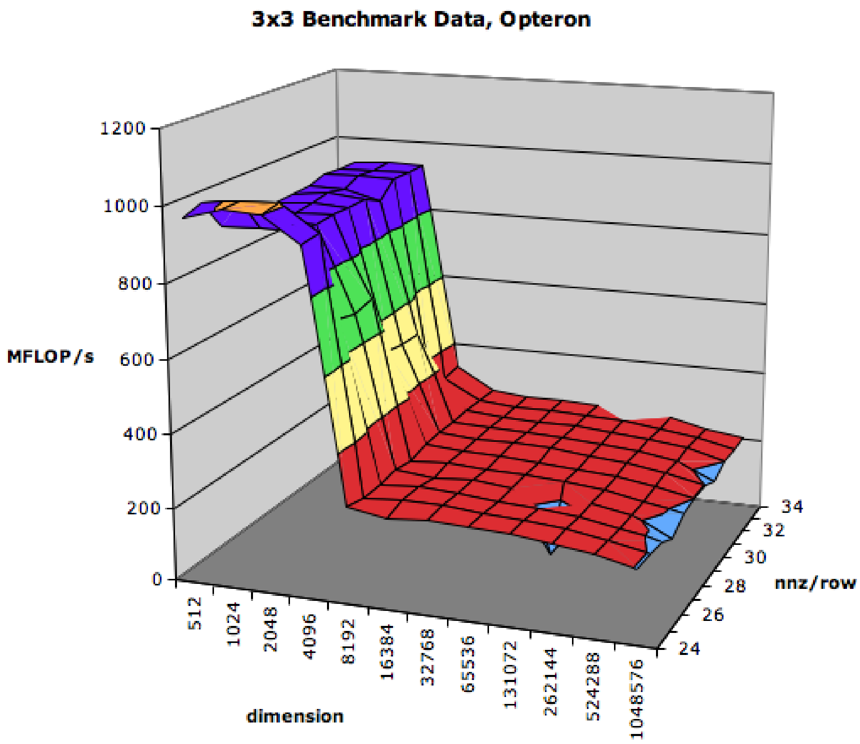

2x4 Benchmark Data, Opteron
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

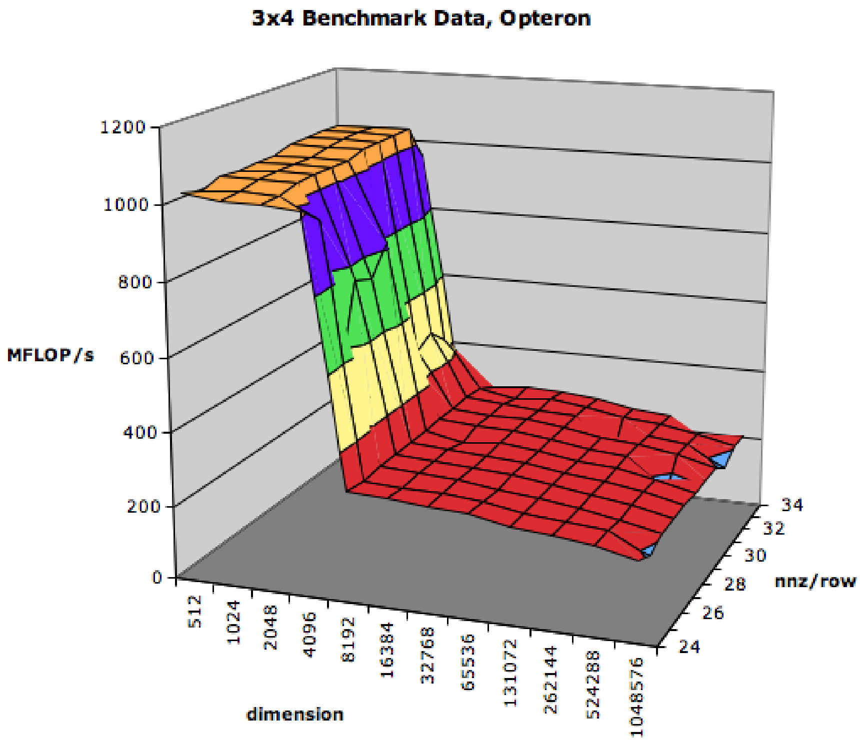

2x6 Benchmark Data, Opteron
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

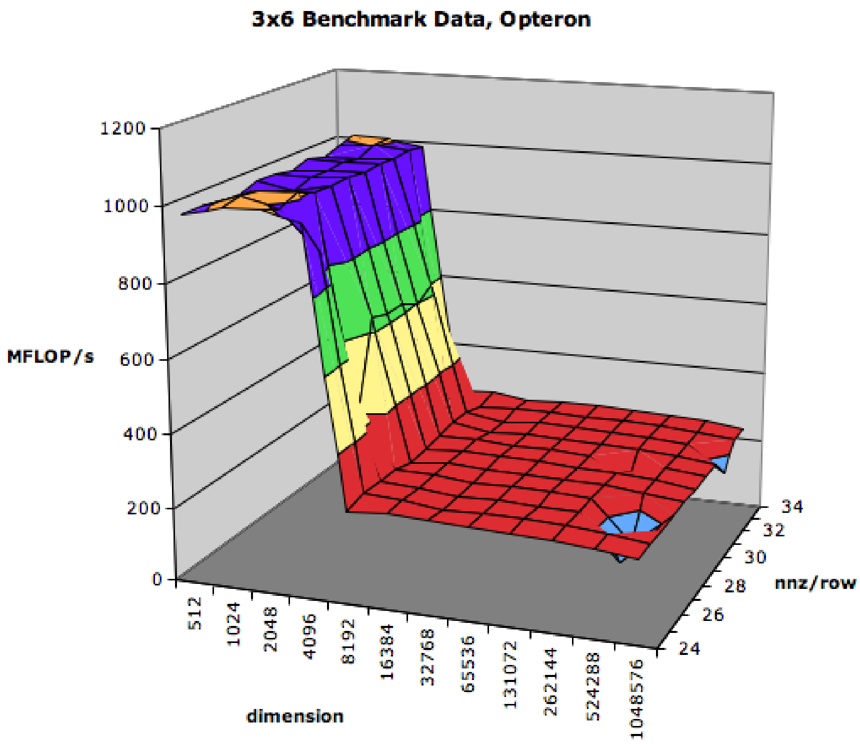

2x8 Benchmark Data, Opteron
1200
1000
800
MFLOP/s
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
nnz/row
26
24


3x1 Benchmark Data, Opteron
800
700
600
500
MFLOP/s
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
nnz/row
26
24

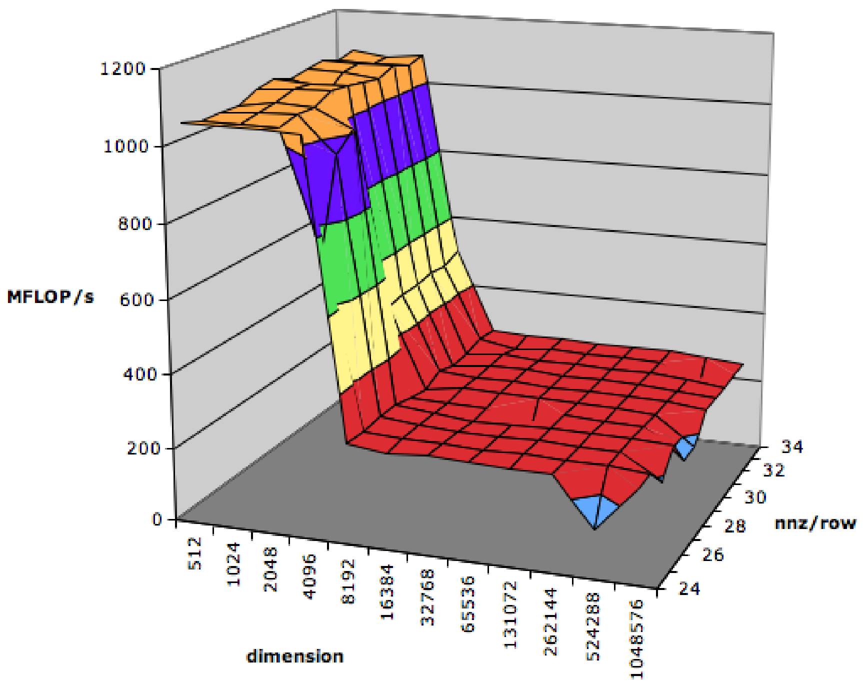

3x2 Benchmark Data, Opteron
MFLOP/s
0
100
200
300
400
500
600
700
800
900
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34
800-900
700-800
600-700
500-600
400-500
300-400
200-300
100-200
0-100


3x3 Benchmark Data, Opteron
MFLOP/s
0
200
400
600
800
1000
1200
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

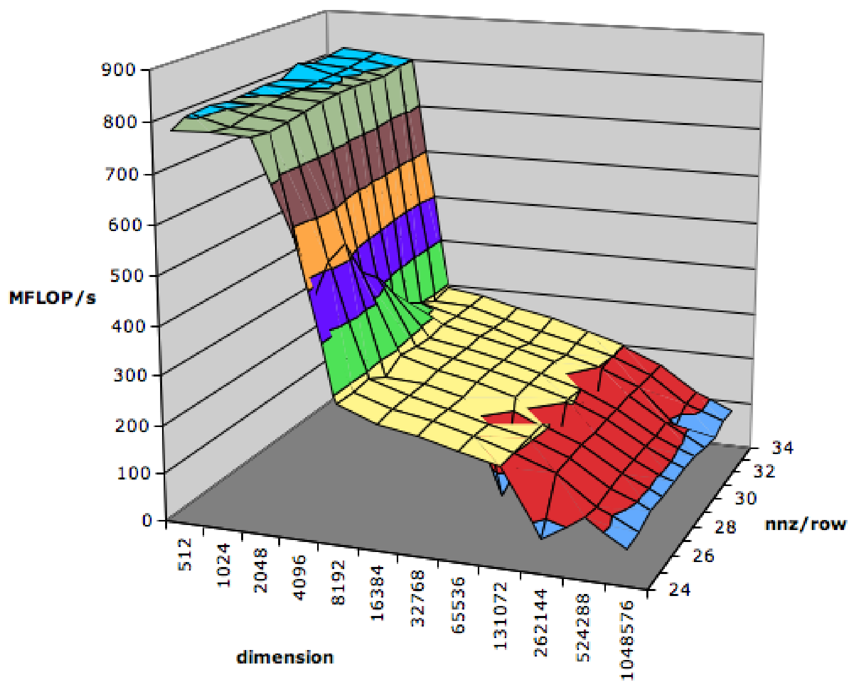

3x4 Benchmark Data, Opteron
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

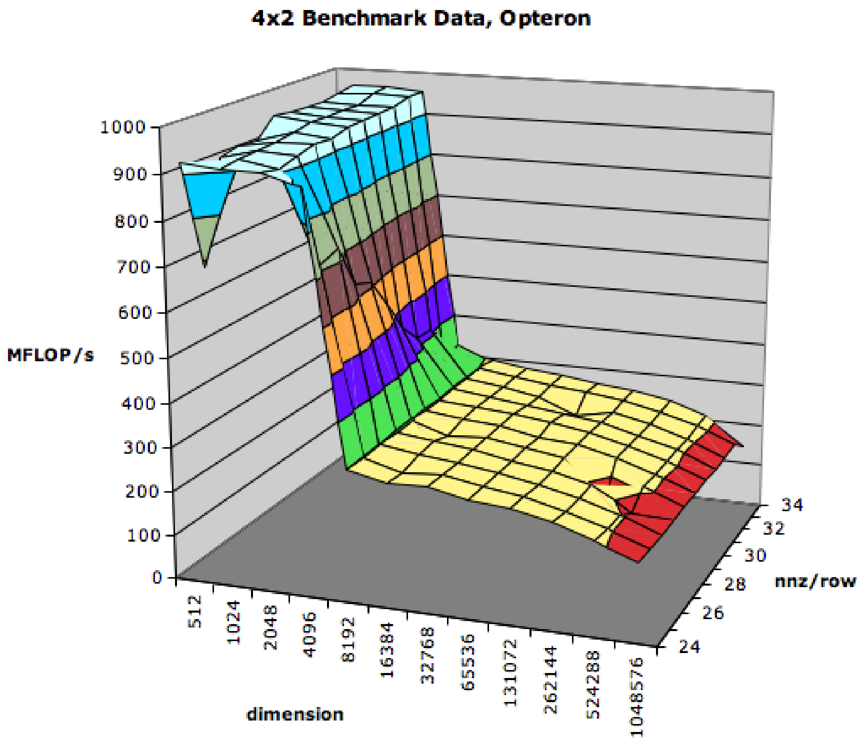

3x6 Benchmark Data, Opteron
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

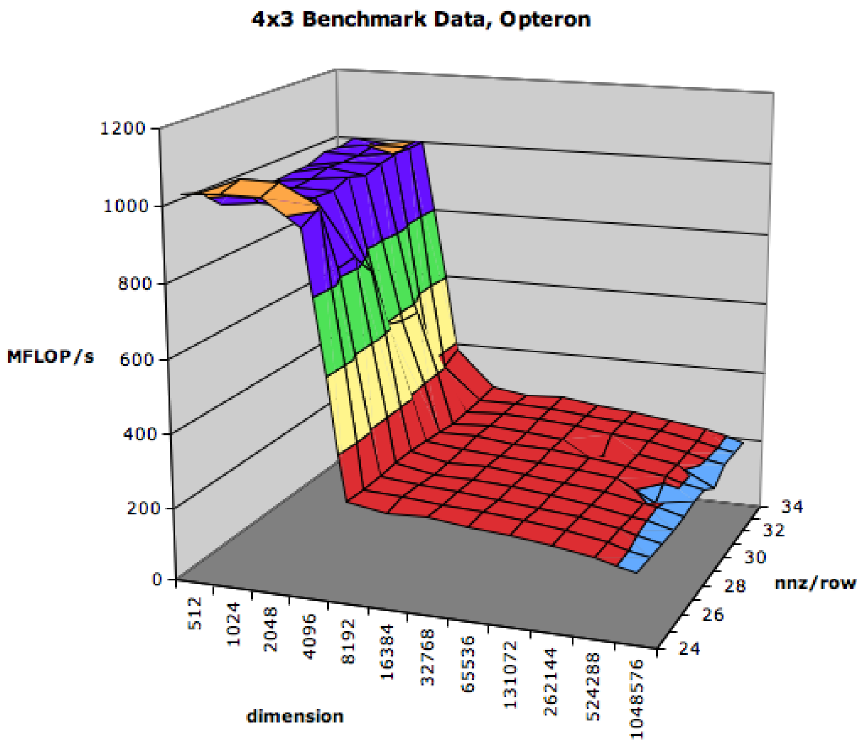
3x8 Benchmark Data, Opteron



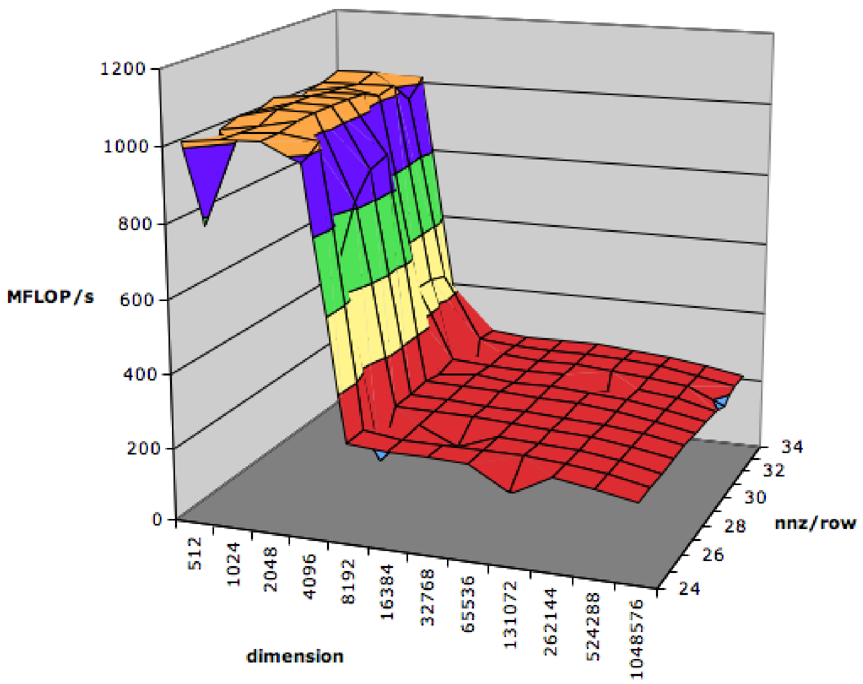
4x1 Benchmark Data, Opteron



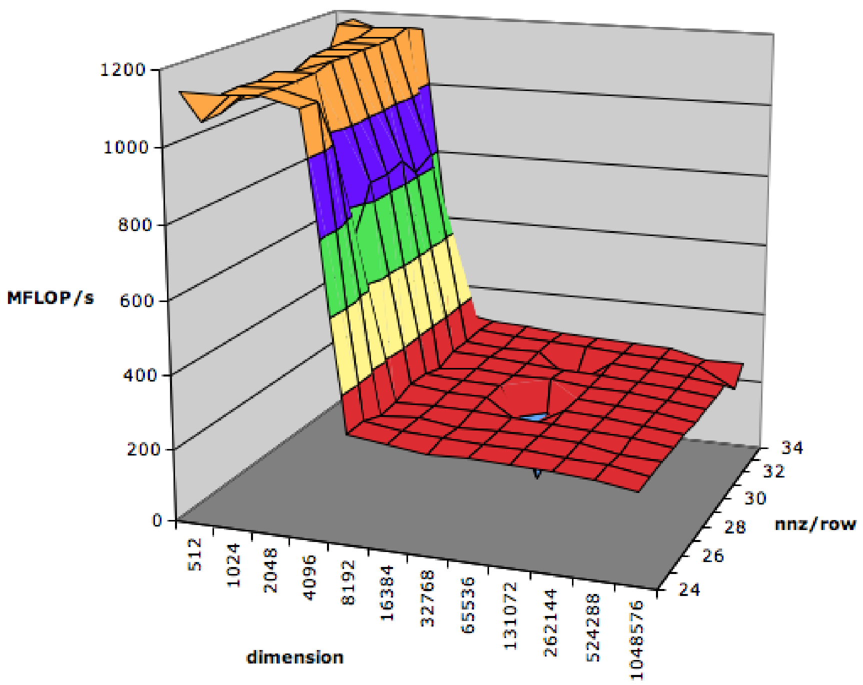

4x2 Benchmark Data, Opteron
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

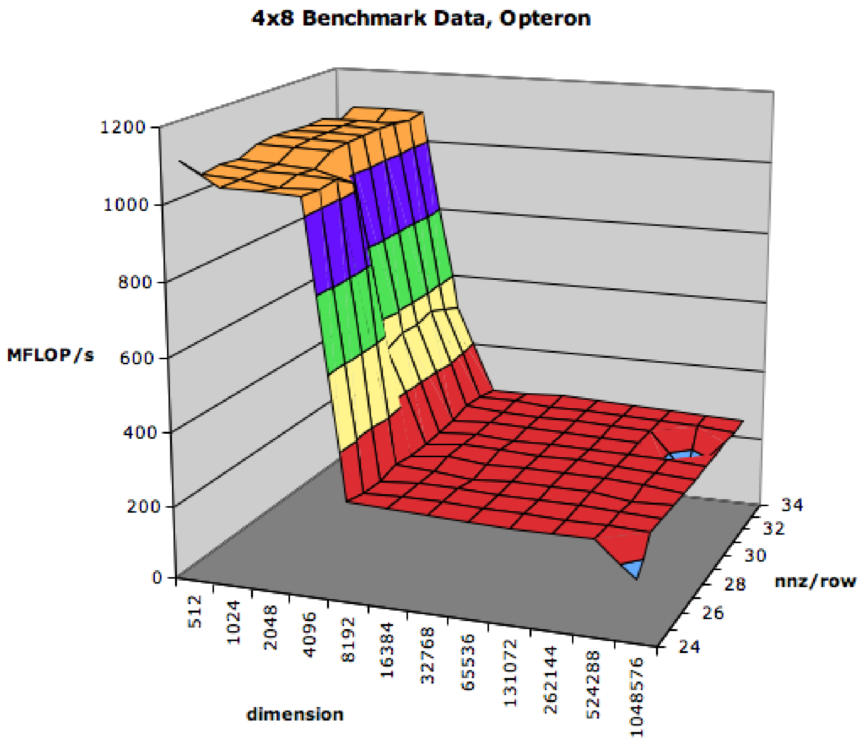

4x3 Benchmark Data, Opteron
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

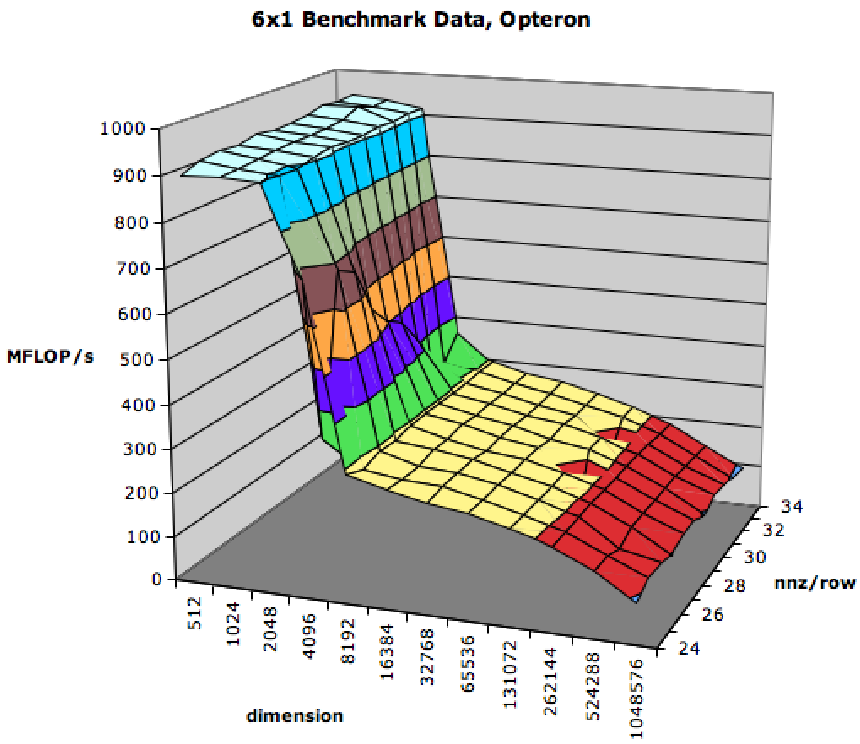
4x4 Benchmark Data, Opteron

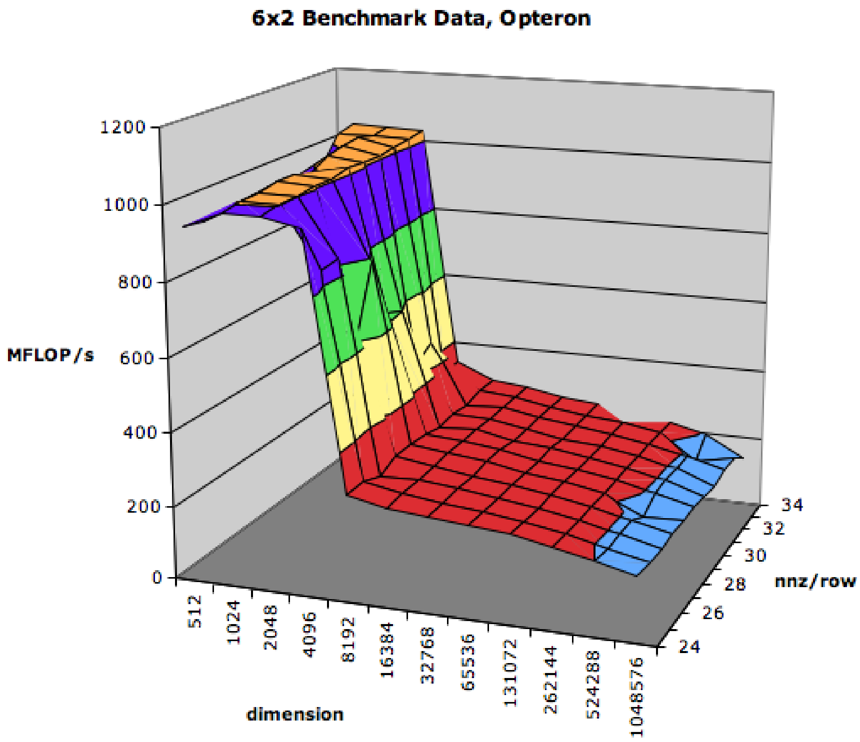


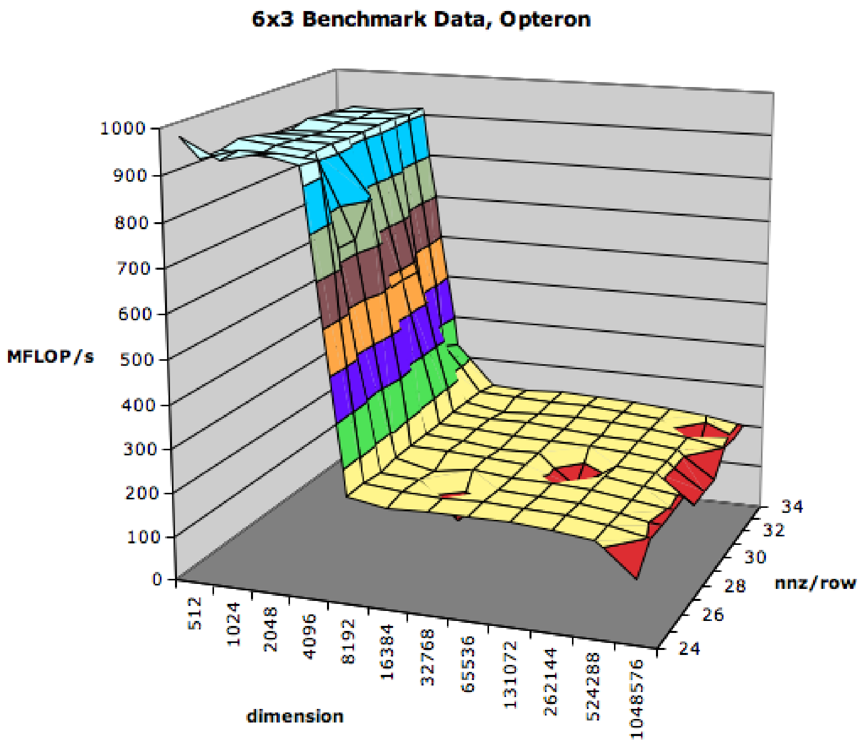
4x6 Benchmark Data, Opteron

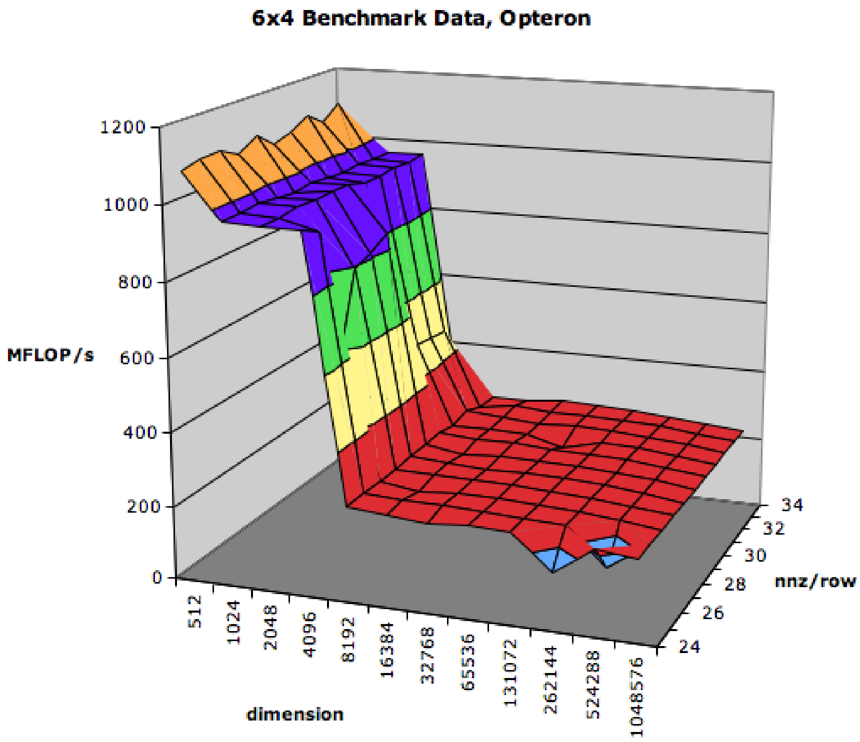


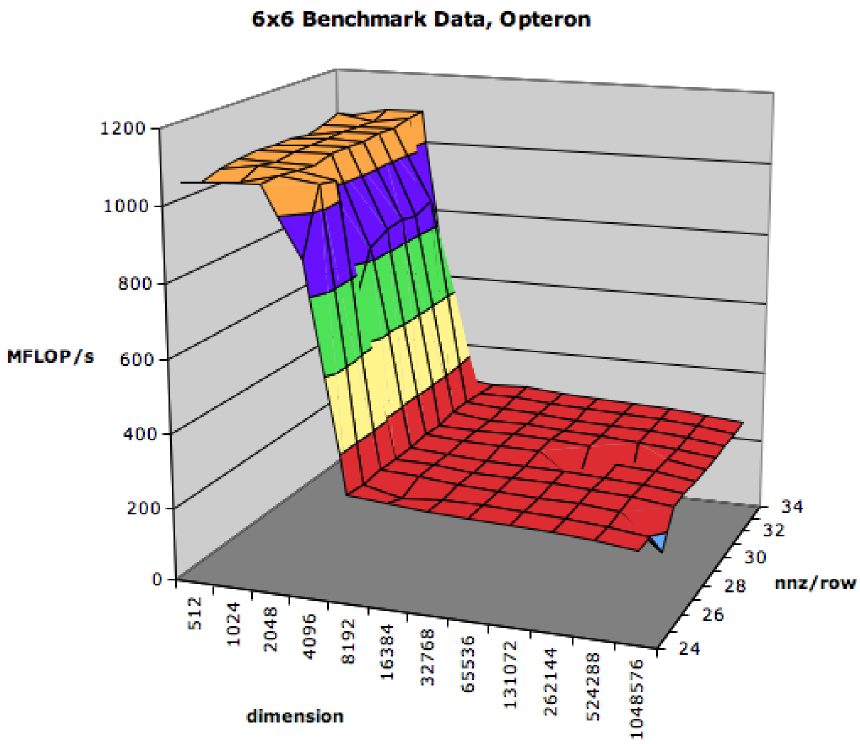

4x8 Benchmark Data, Opteron
MFLOP/s
0
200
400
600
800
1000
1200
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34


6x1 Benchmark Data, Opteron
MFLOP/s
0
100
200
300
400
500
600
700
800
900
1000
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

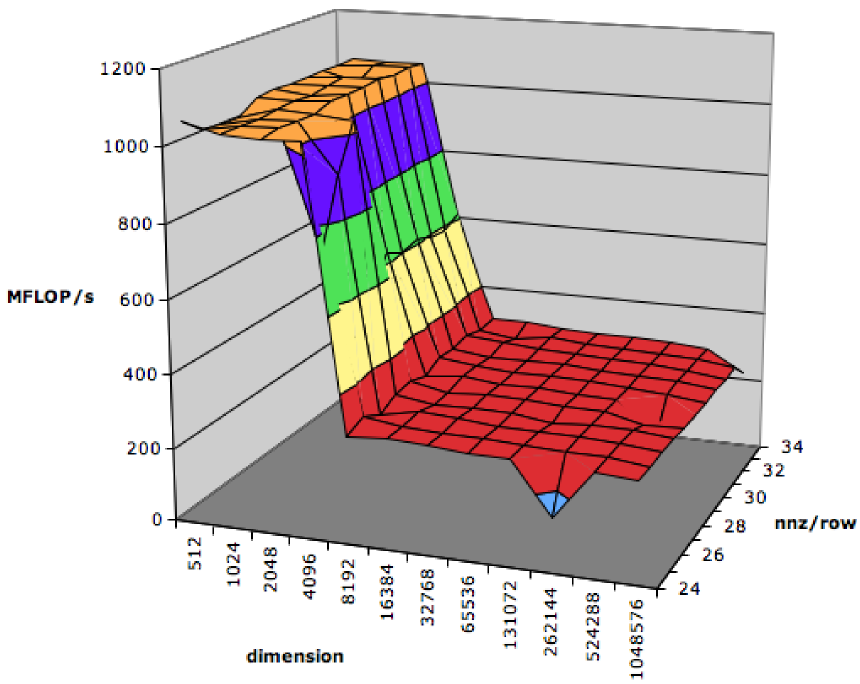

6x2 Benchmark Data, Opteron
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row


6x3 Benchmark Data, Opteron
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

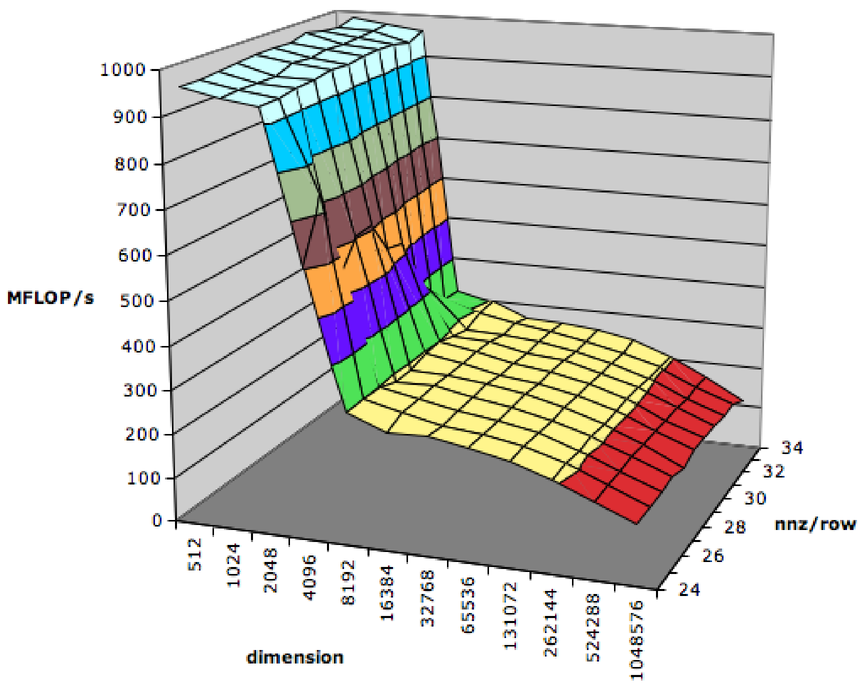

6x4 Benchmark Data, Opteron
MFLOP/s
0
200
400
600
800
1000
1200
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

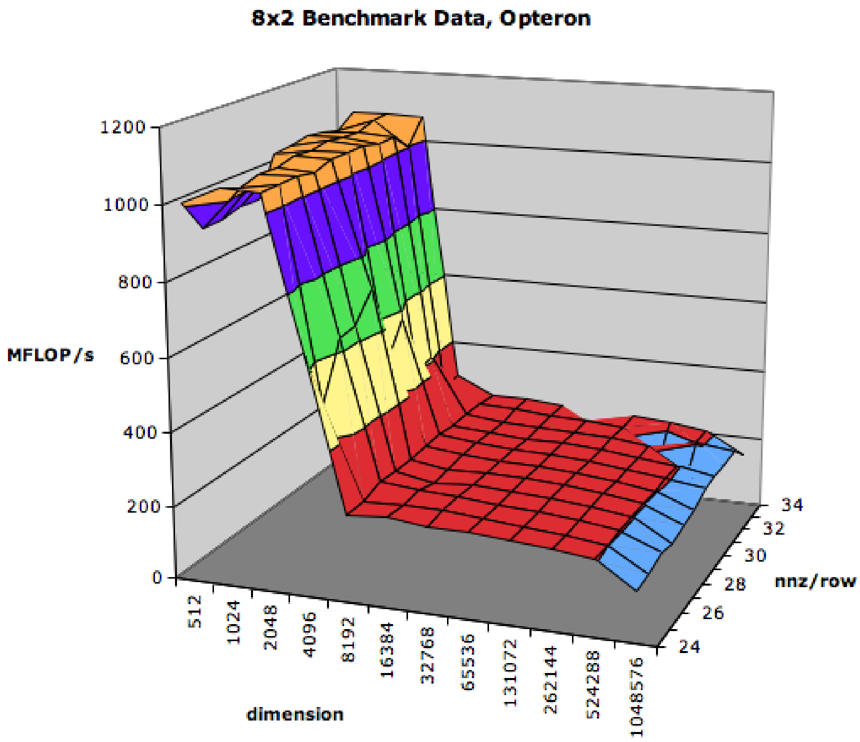

6x6 Benchmark Data, Opteron
MFLOP/s
0
200
400
600
800
1000
1200
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

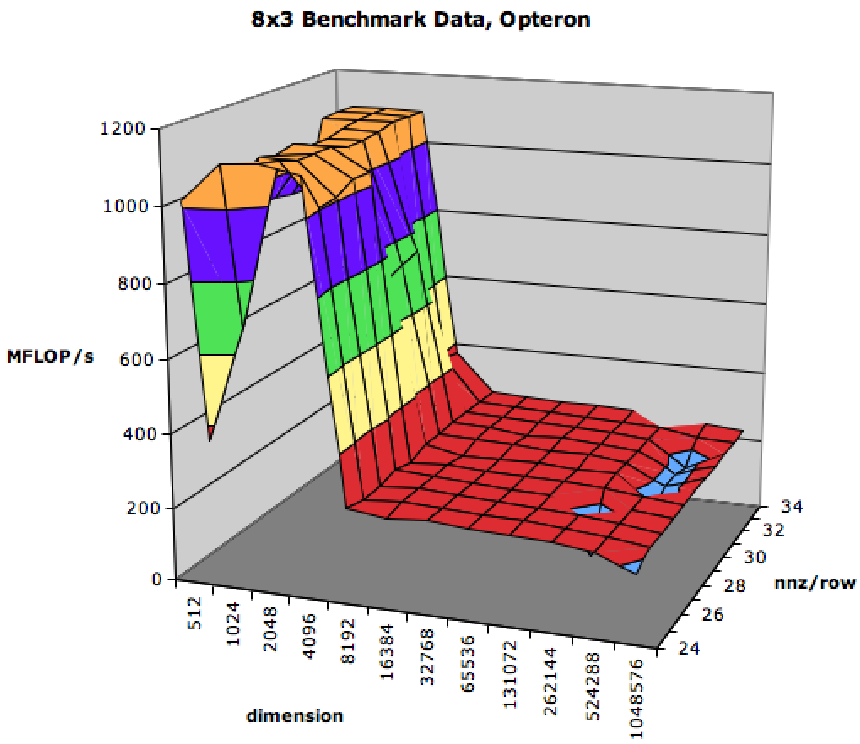
6x8 Benchmark Data, Opteron

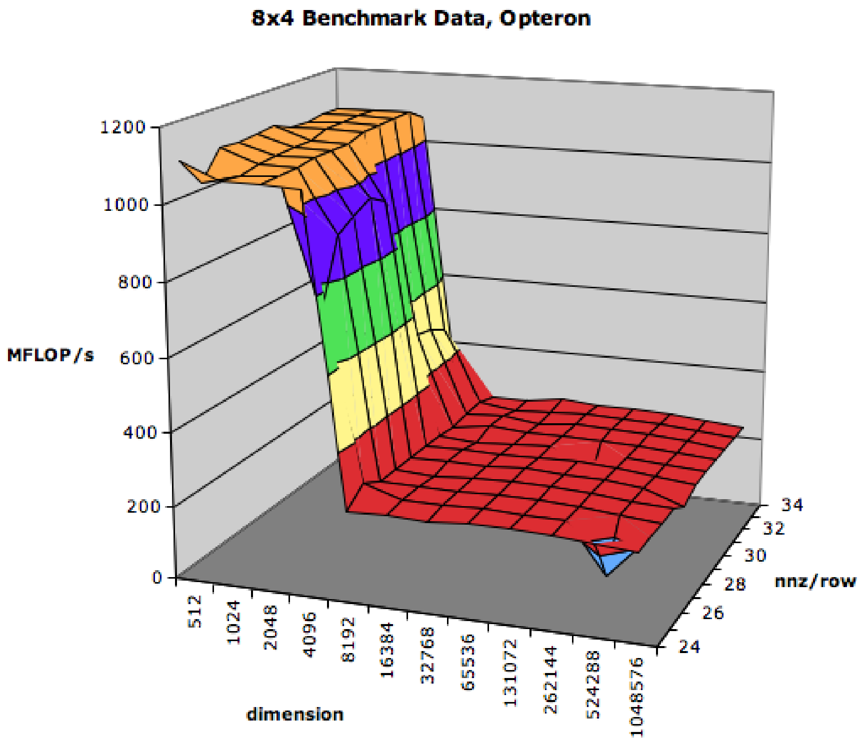


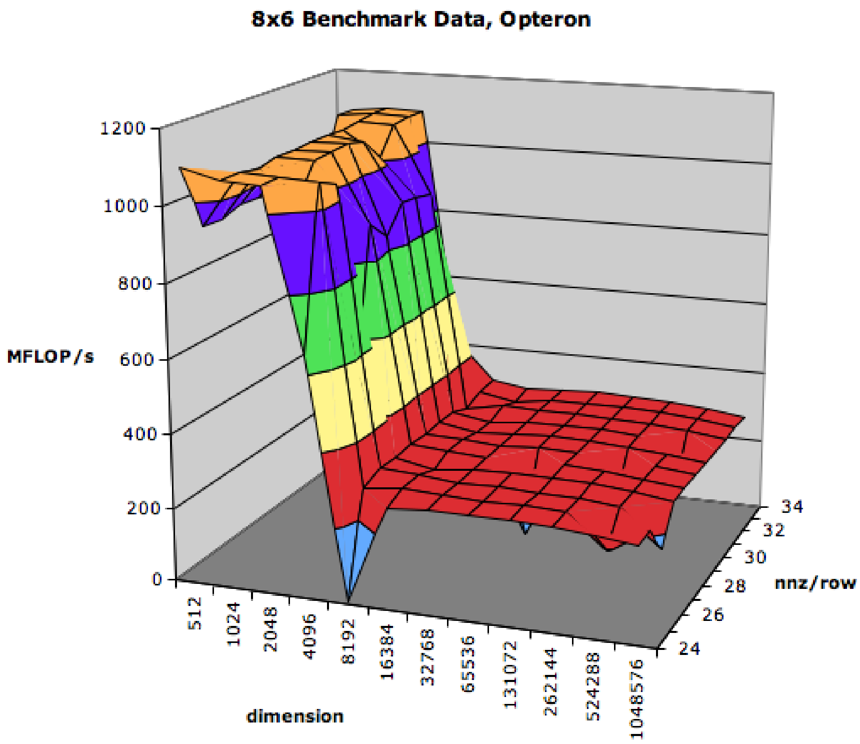
8x1 Benchmark Data, Opteron

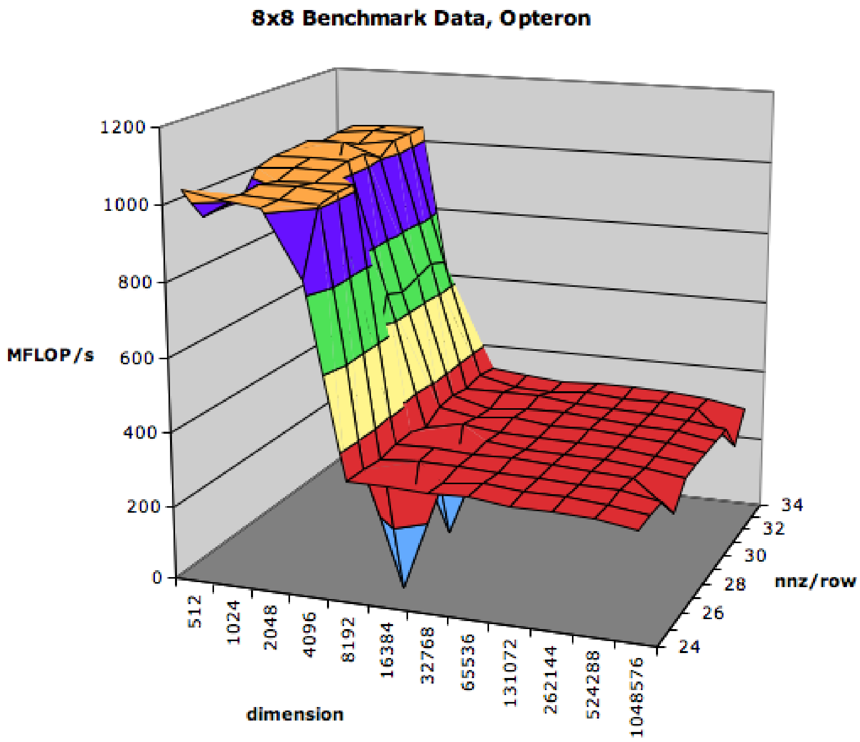



8x2 Benchmark Data, Opteron
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row


8x3 Benchmark Data, Opteron
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row


8x4 Benchmark Data, Opteron
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

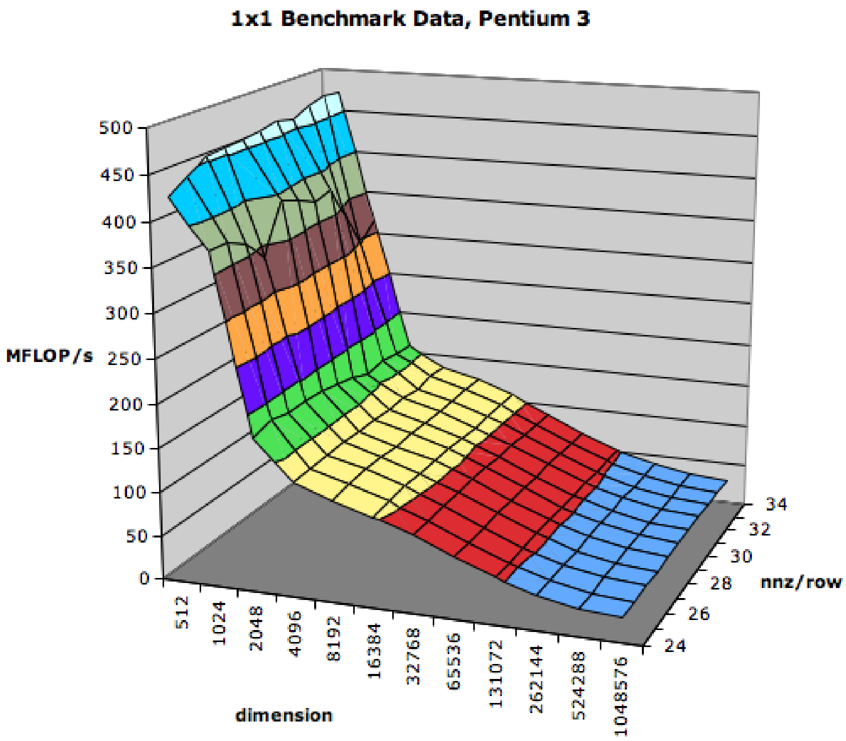

8x6 Benchmark Data, Opteron
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

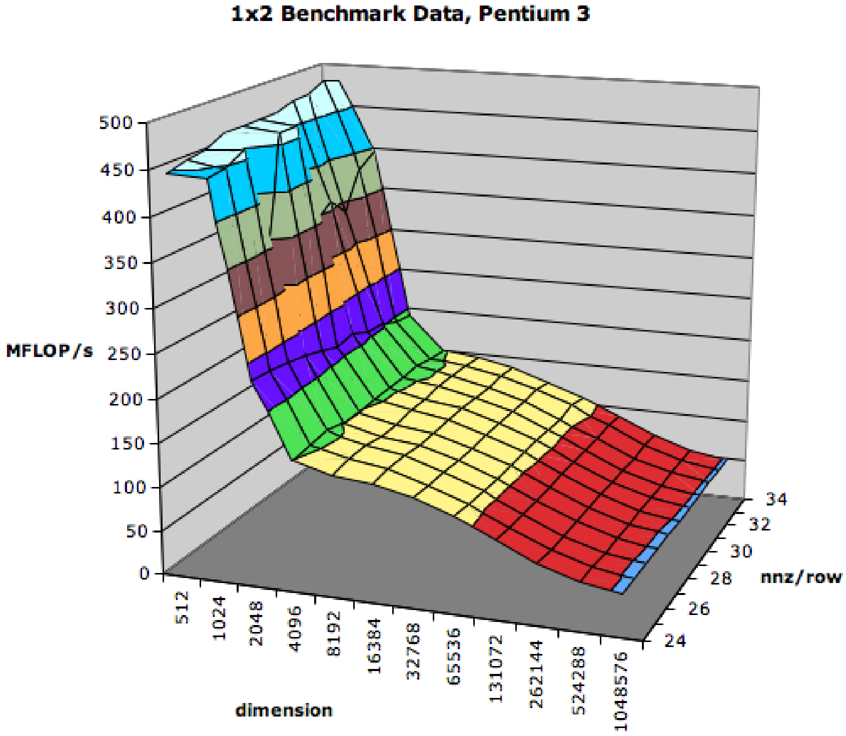

8x8 Benchmark Data, Opteron
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

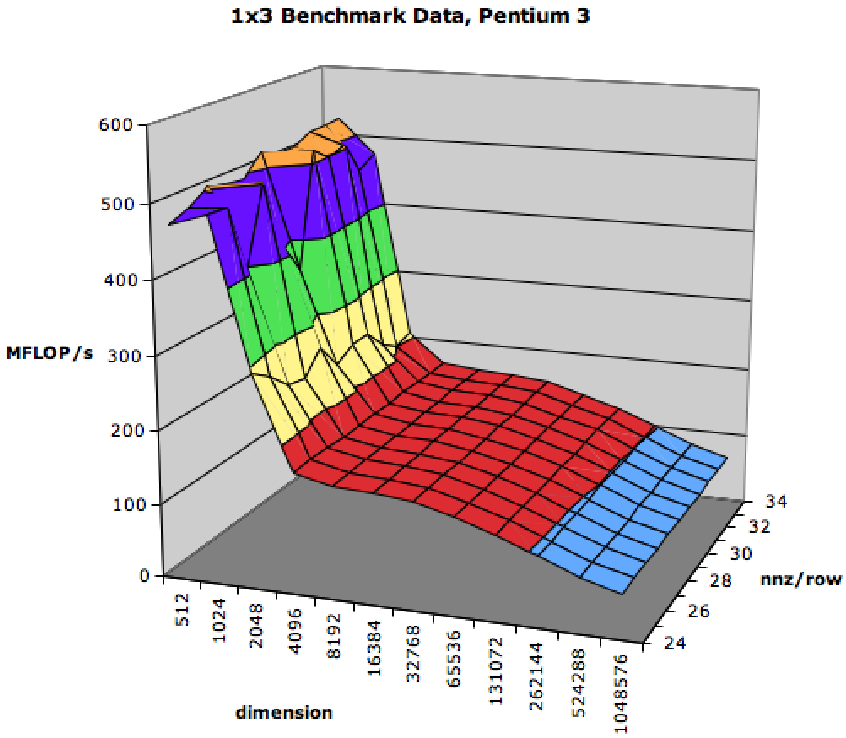
Appendix K

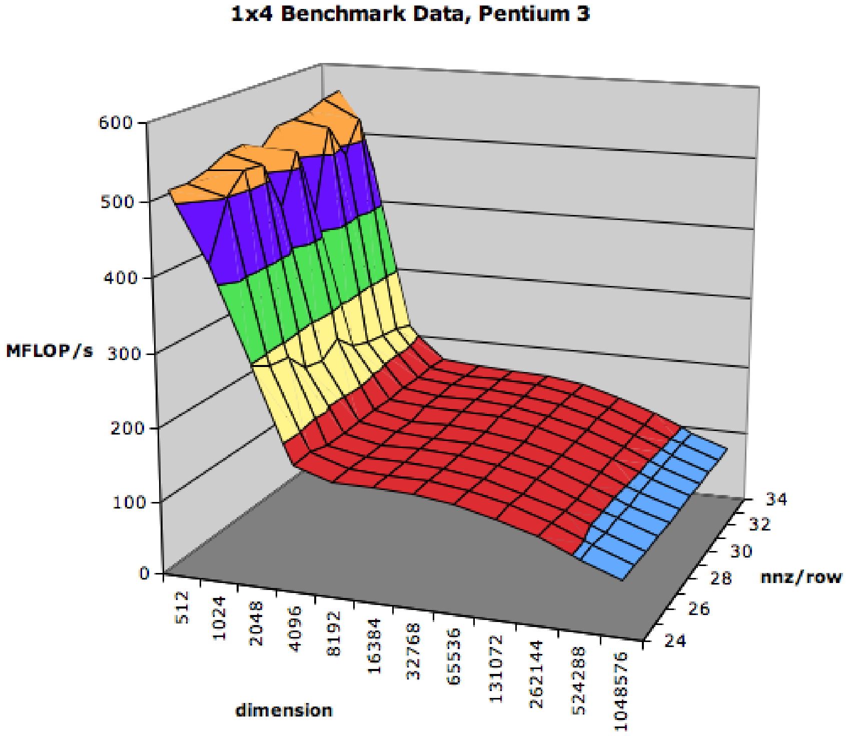
Pentium 3 Benchmark Data

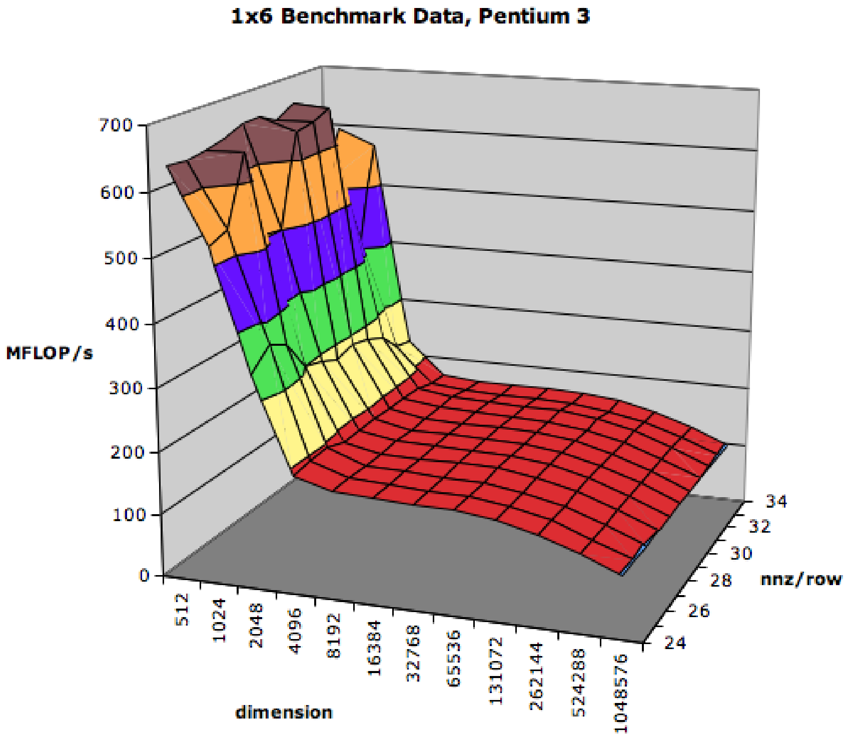
Here we graphically present the full output of our benchmark on the Pentium 3.

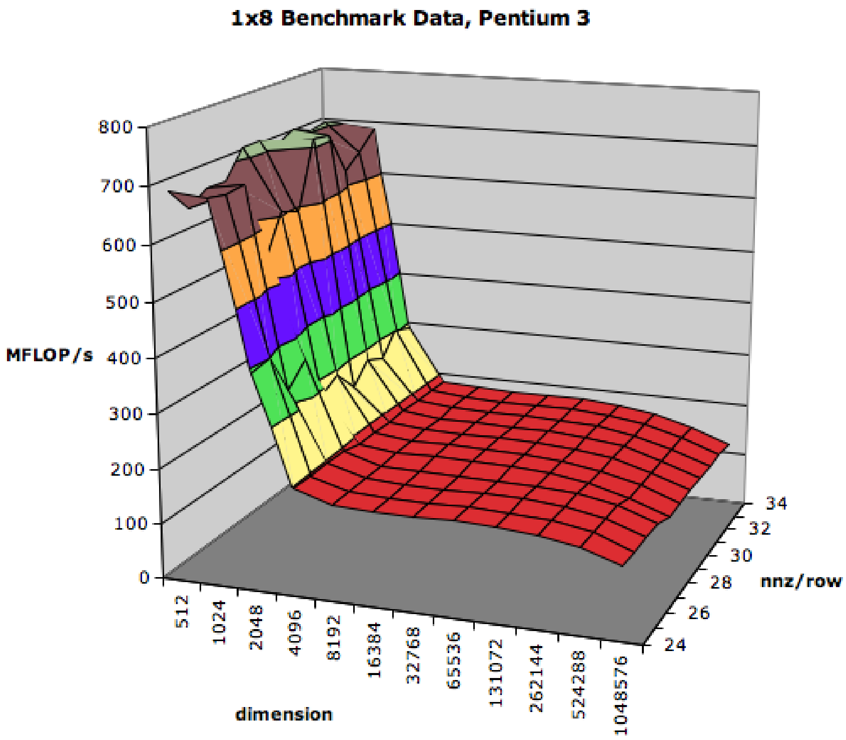


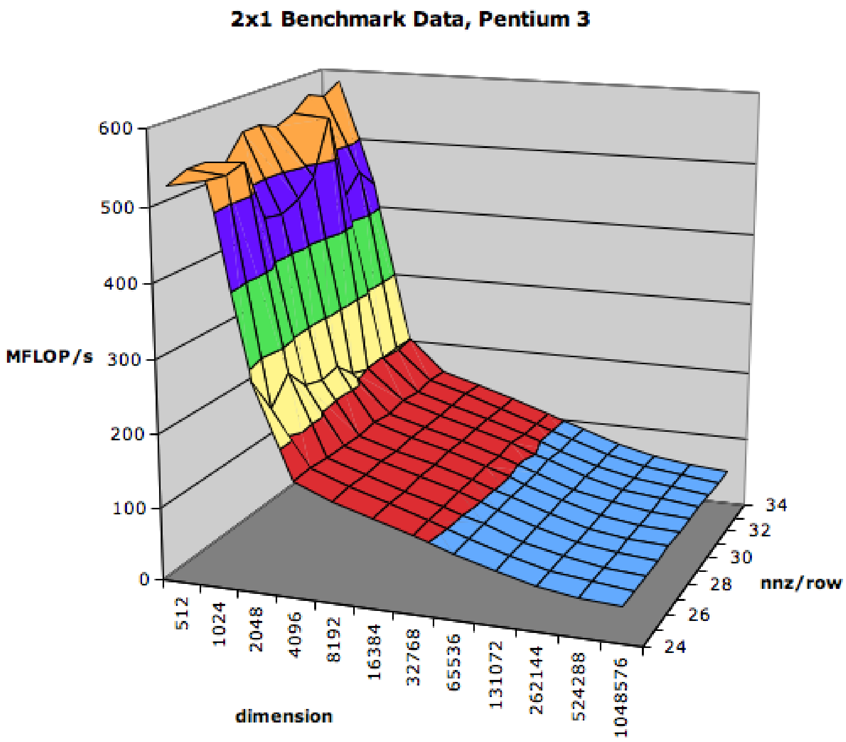

1x2 Benchmark Data, Pentium 3
MFLOP/s
500
450
400
350
300
250
200
150
100
50
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row


1x3 Benchmark Data, Pentium 3
MFLOP/s
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

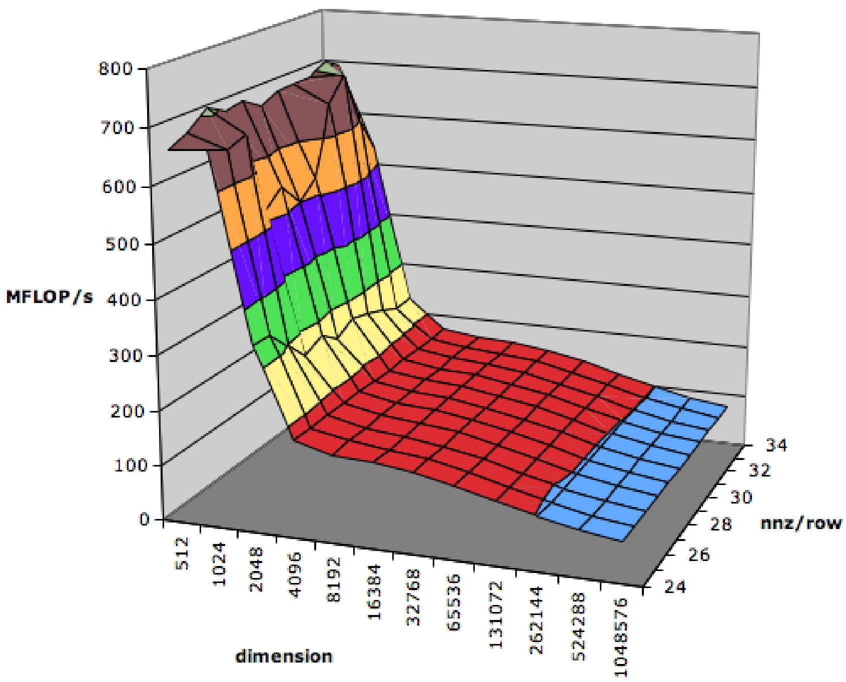

1x4 Benchmark Data, Pentium 3
MFLOP/s
0
100
200
300
400
500
600
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34


1x6 Benchmark Data, Pentium 3
MFLOP/s
0
100
200
300
400
500
600
700
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

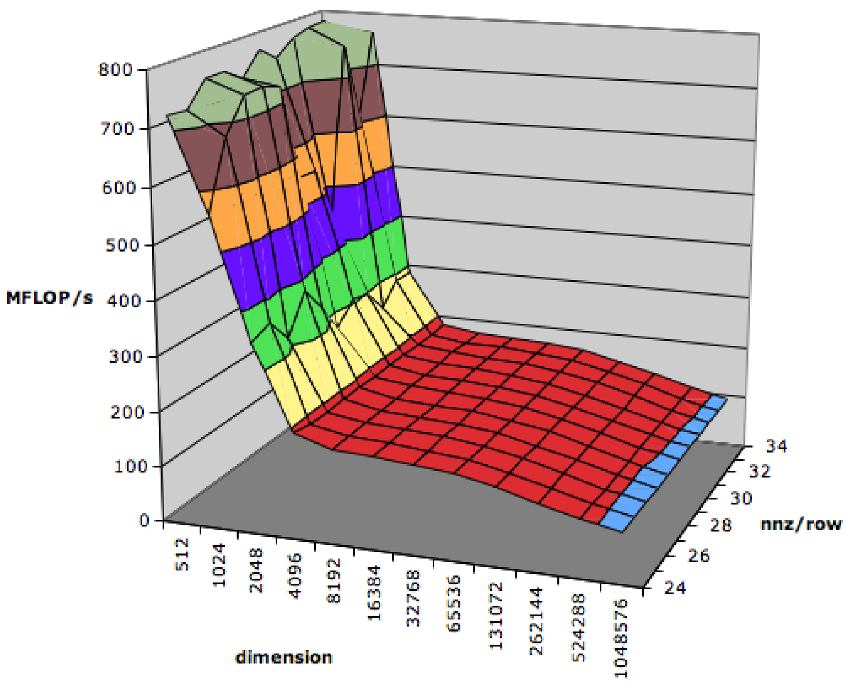

1x8 Benchmark Data, Pentium 3
MFLOP/s
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

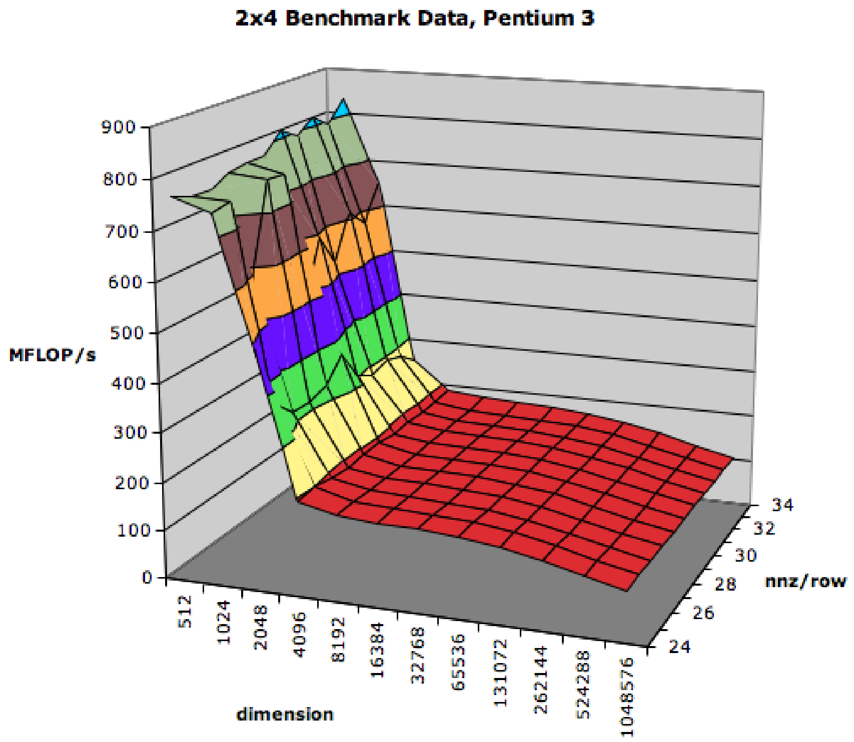

2x1 Benchmark Data, Pentium 3
MFLOP/s
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

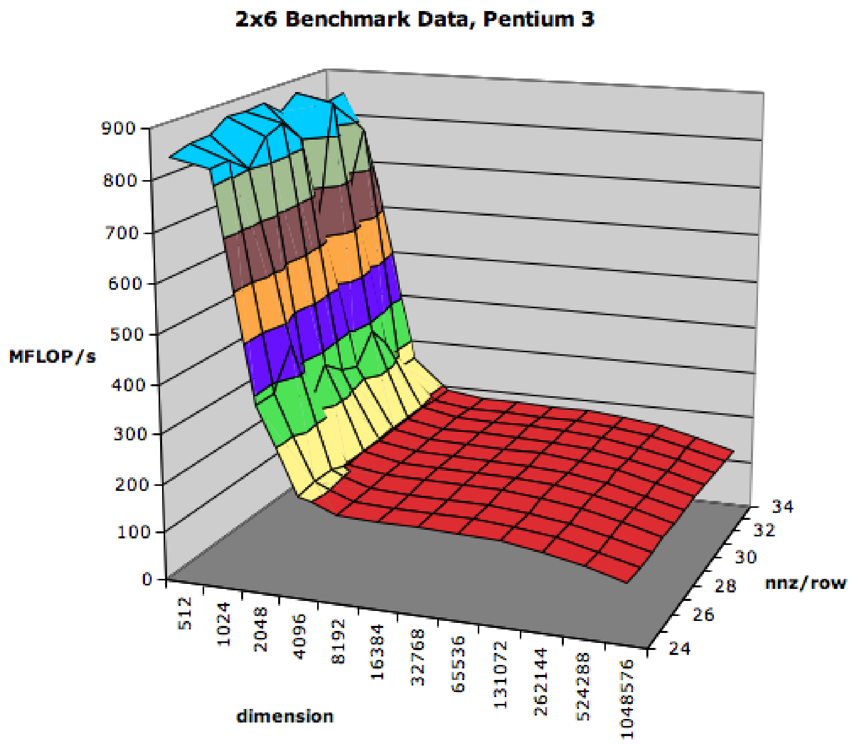
2x2 Benchmark Data, Pentium 3

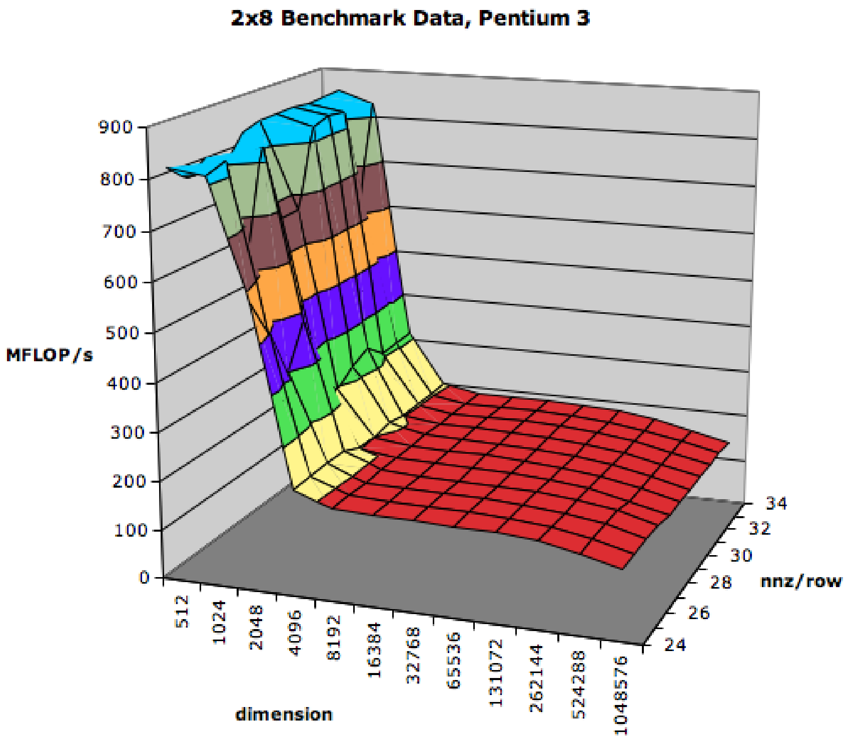


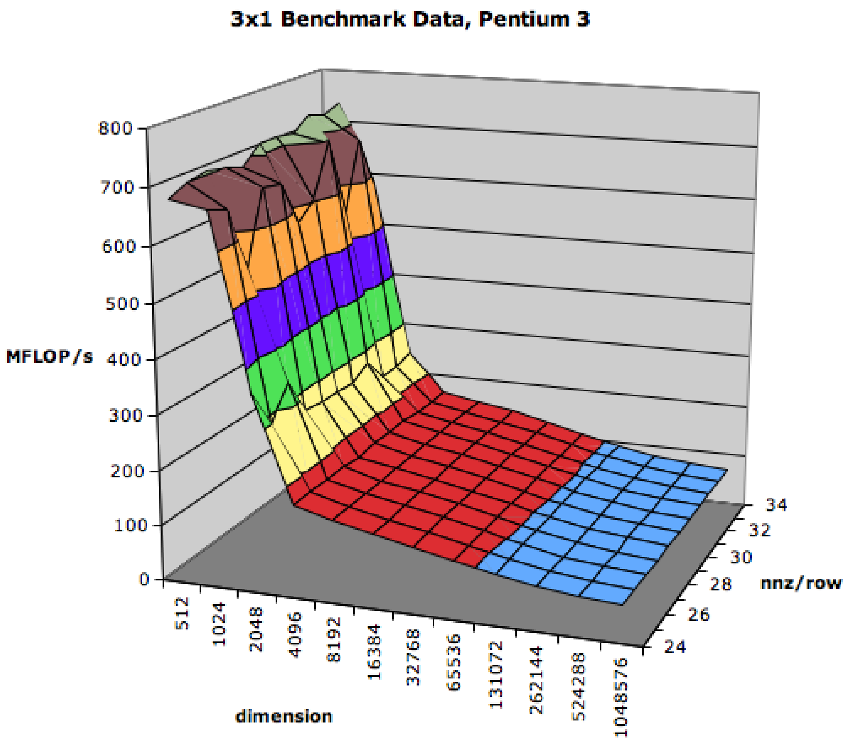
2x3 Benchmark Data, Pentium 3

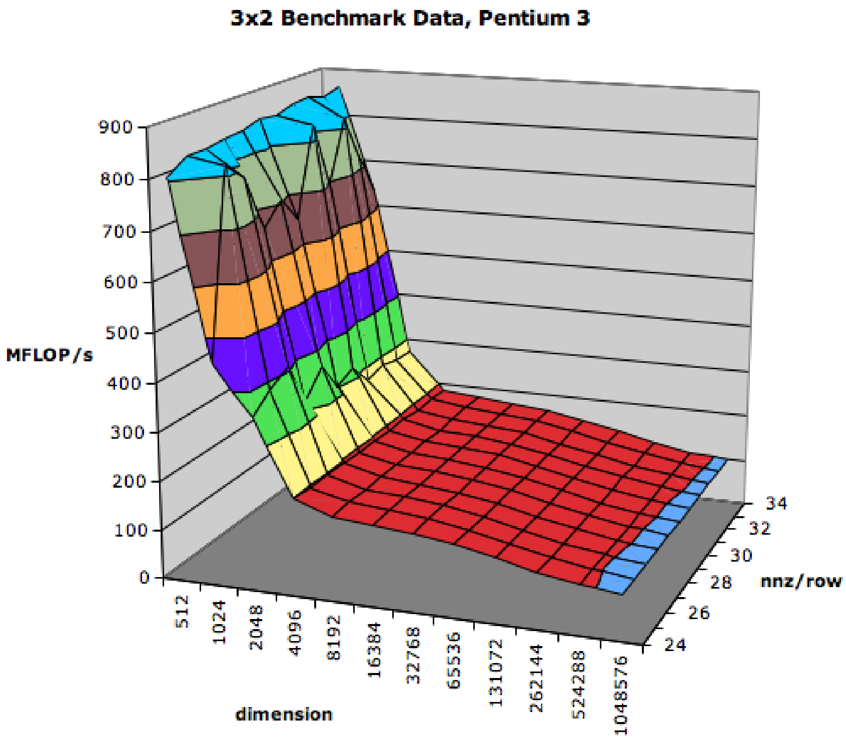


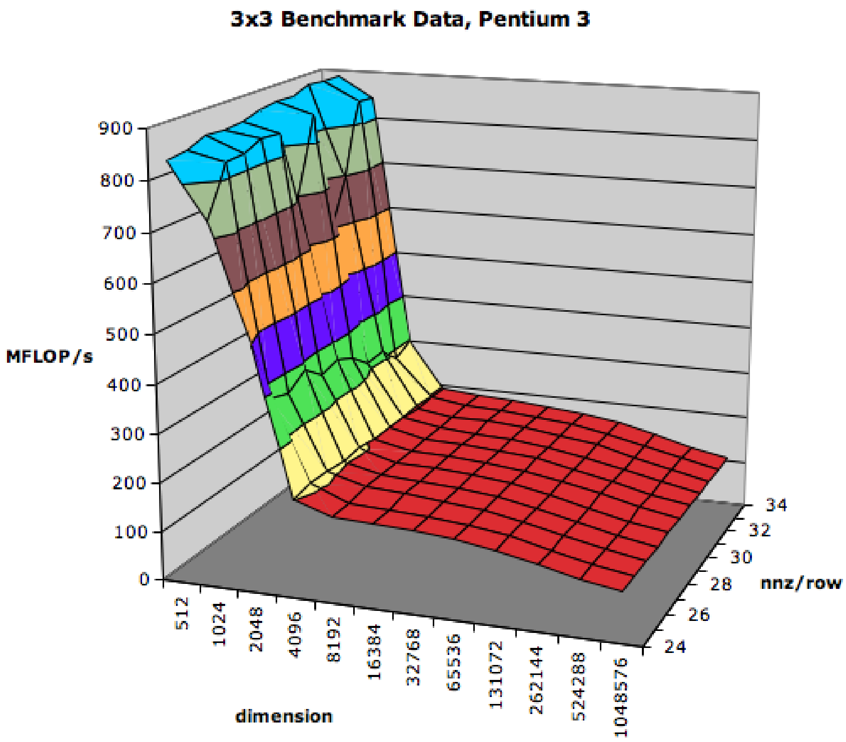

2x4 Benchmark Data, Pentium 3
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

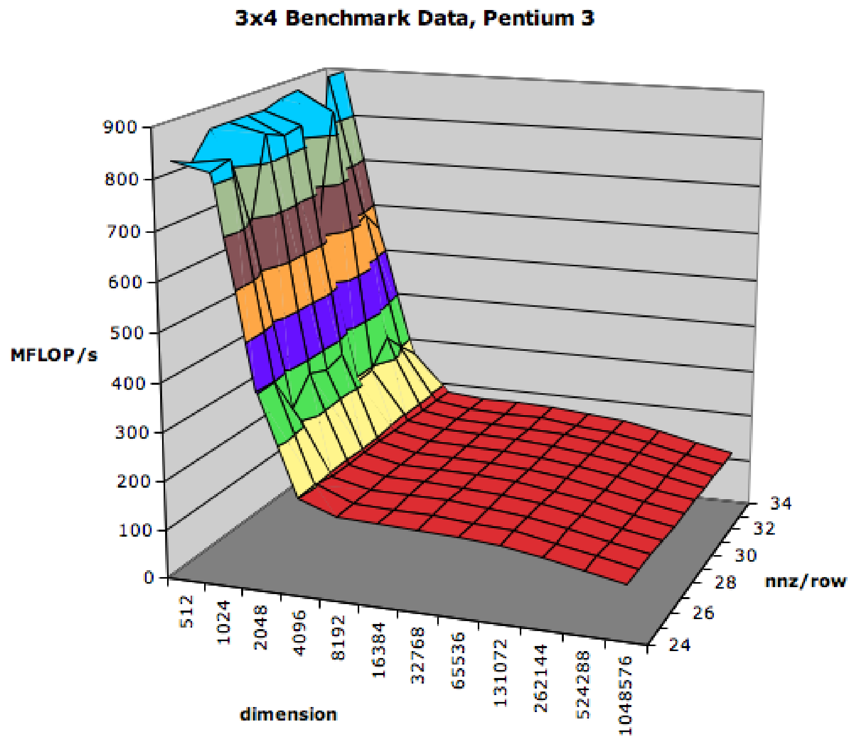

2x6 Benchmark Data, Pentium 3
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

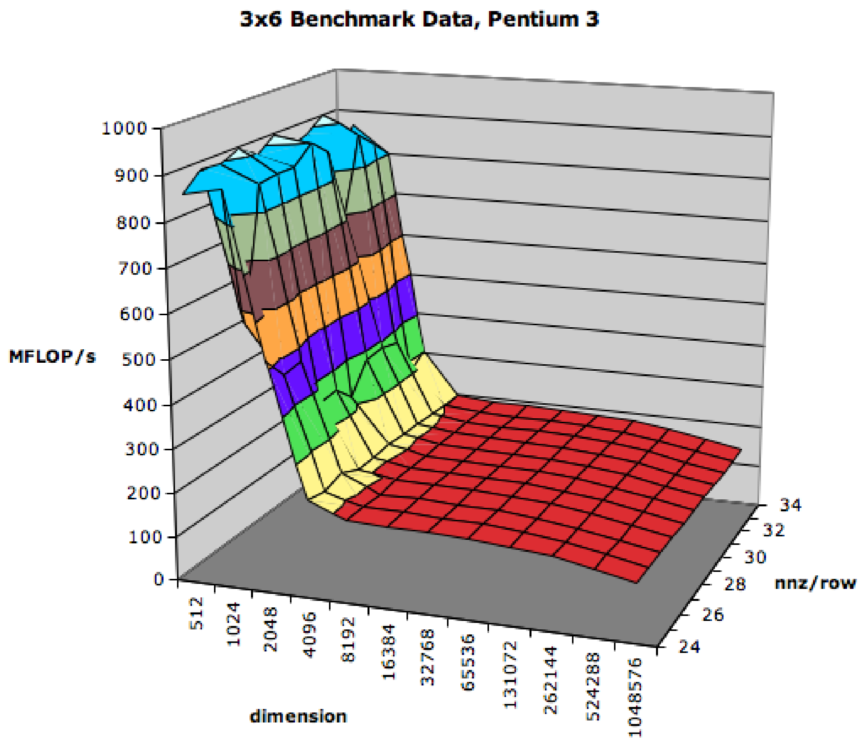

2x8 Benchmark Data, Pentium 3
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

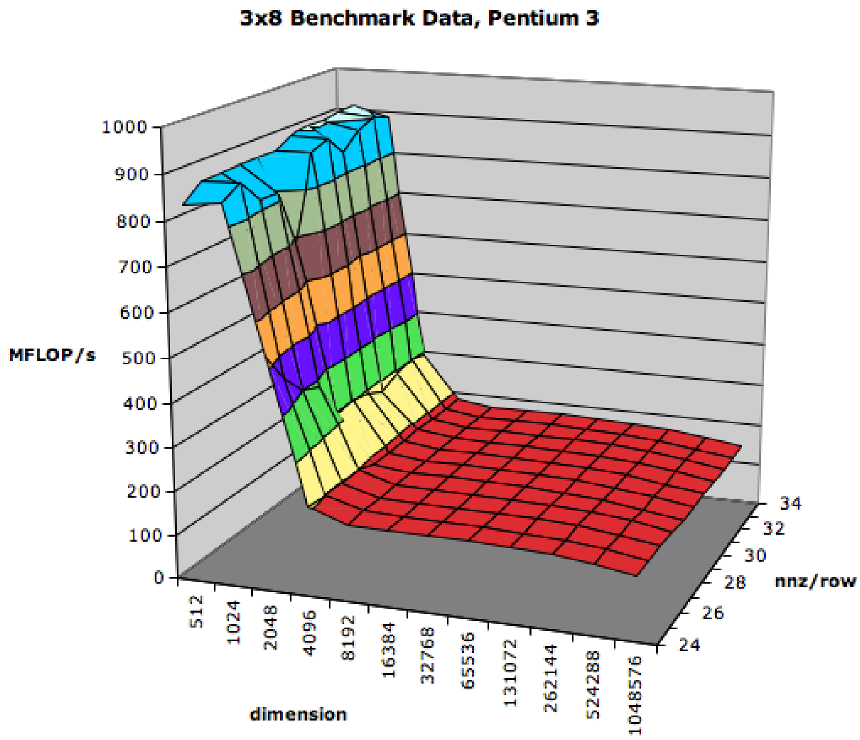

3x1 Benchmark Data, Pentium 3
MFLOP/s
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

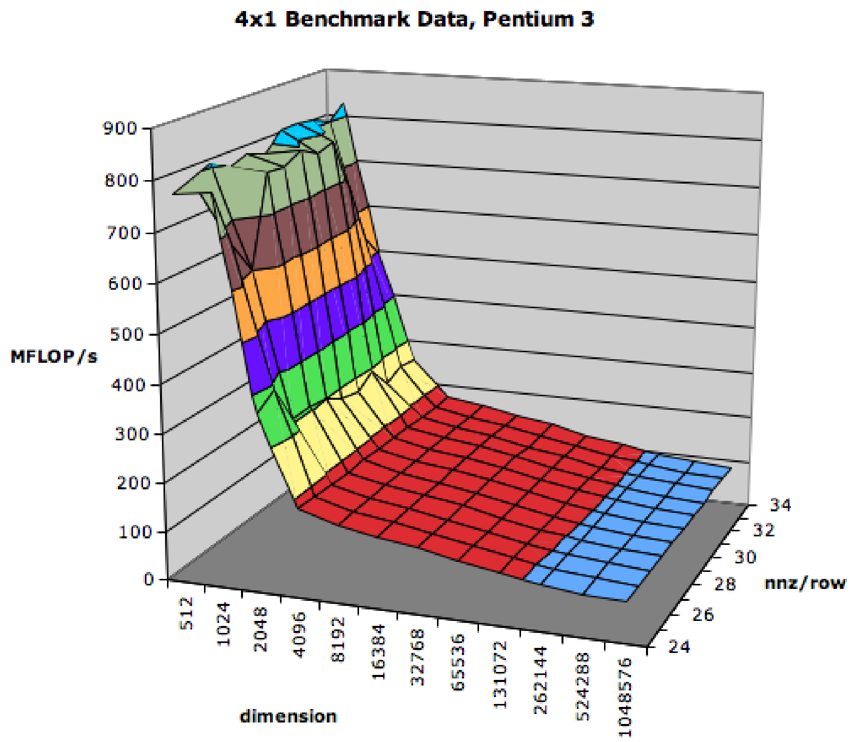

3x2 Benchmark Data, Pentium 3
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row


3x3 Benchmark Data, Pentium 3
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

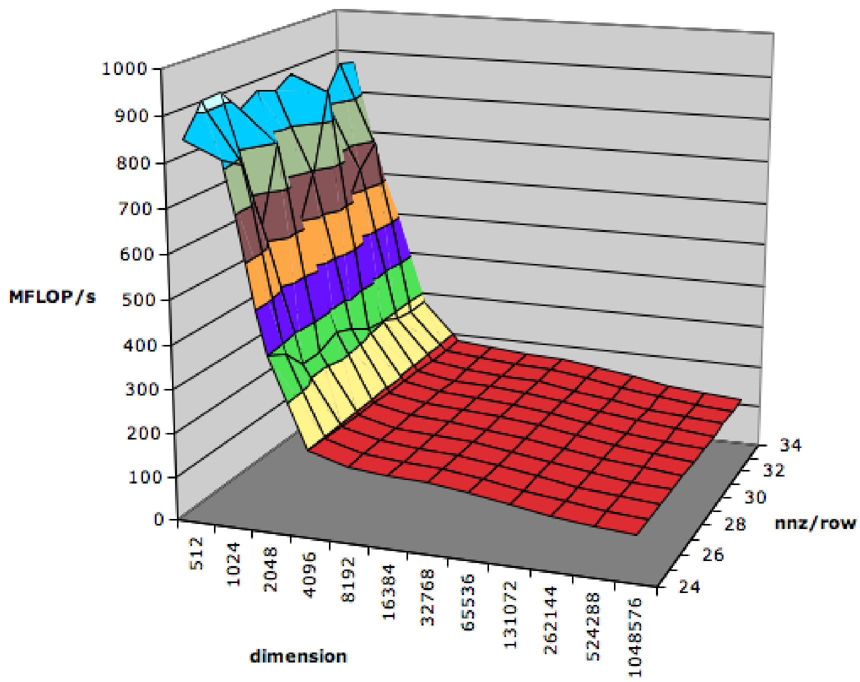

3x4 Benchmark Data, Pentium 3
MFLOP/s
0
100
200
300
400
500
600
700
800
900
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34


3x6 Benchmark Data, Pentium 3
MFLOP/s
0
100
200
300
400
500
600
700
800
900
1000
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

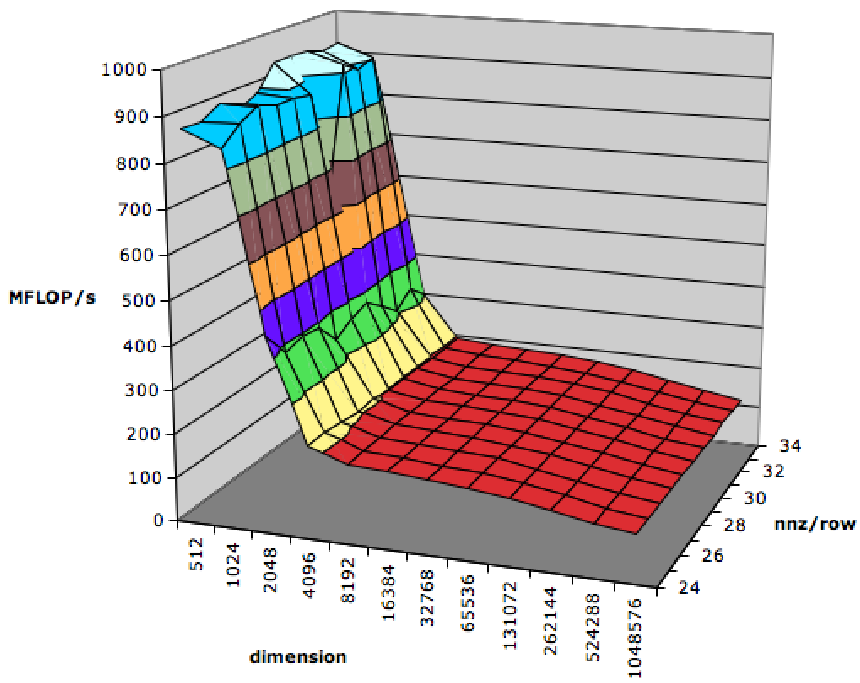

3x8 Benchmark Data, Pentium 3
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
nnz/row
24
26
28
30
32
34

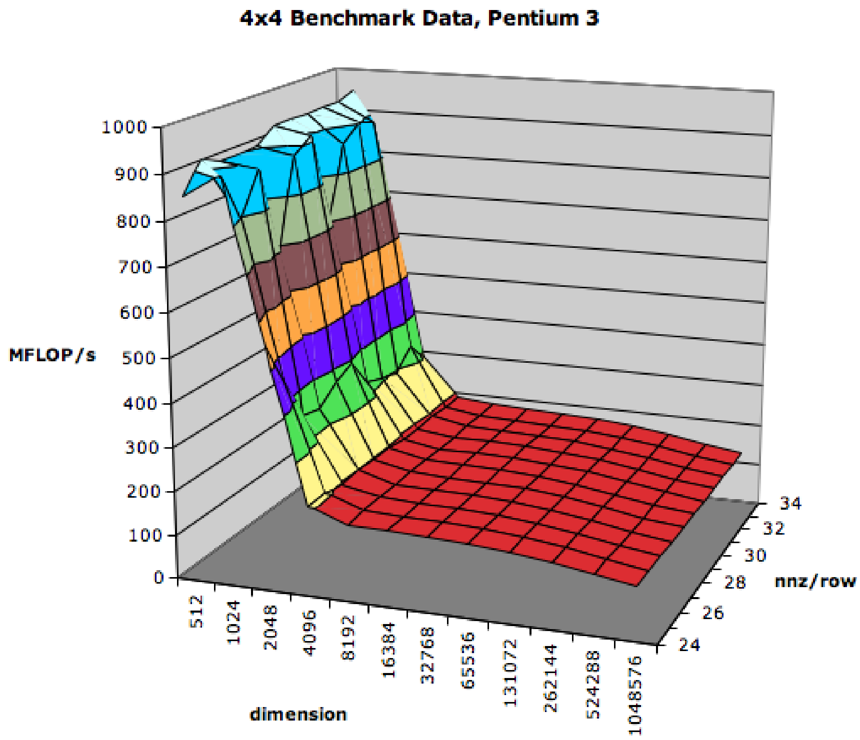

4x1 Benchmark Data, Pentium 3
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
nnz/row
24
26
28
30
32
34

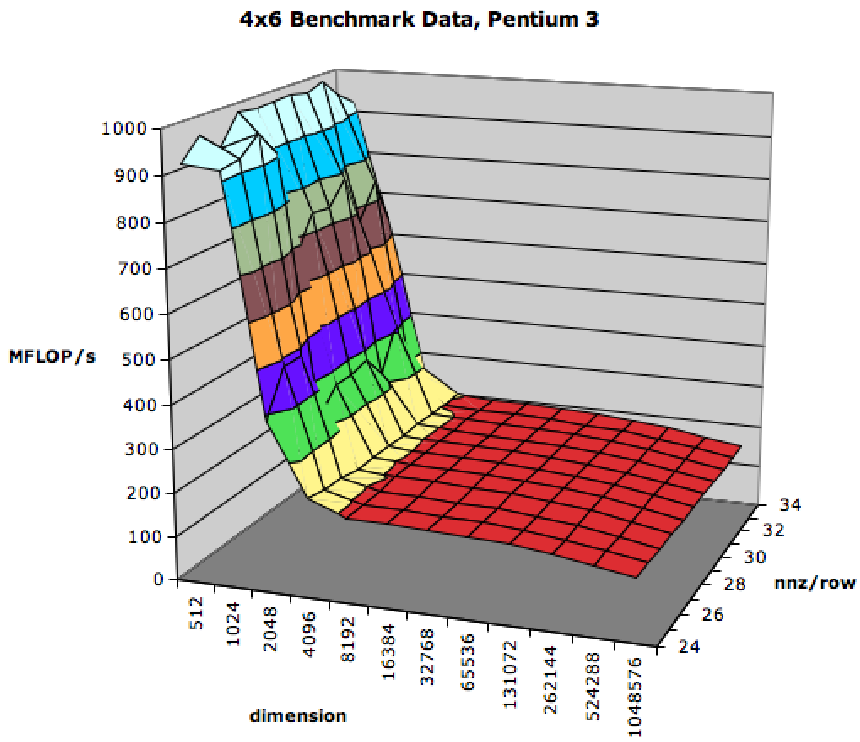
4x2 Benchmark Data, Pentium 3



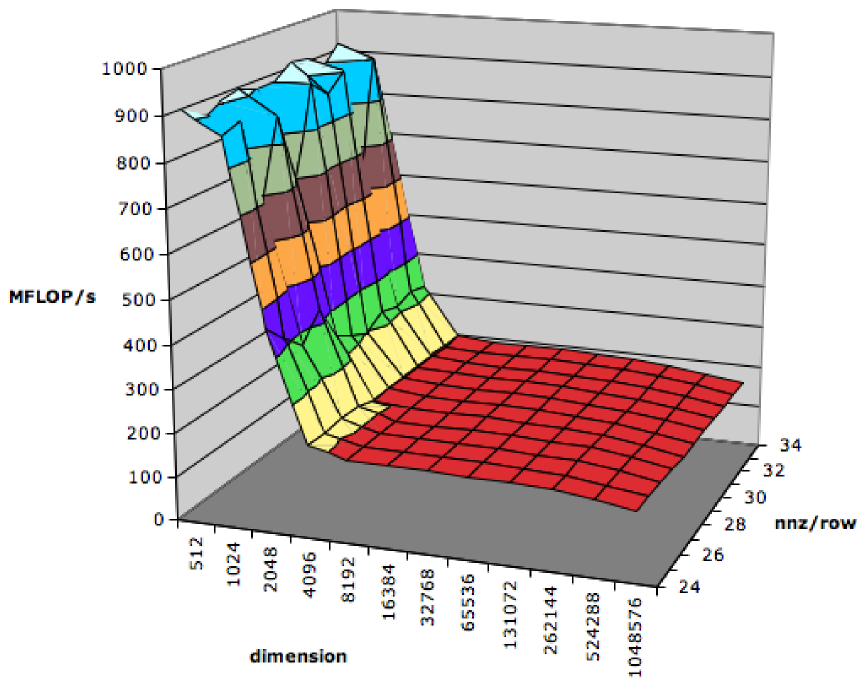
4x3 Benchmark Data, Pentium 3



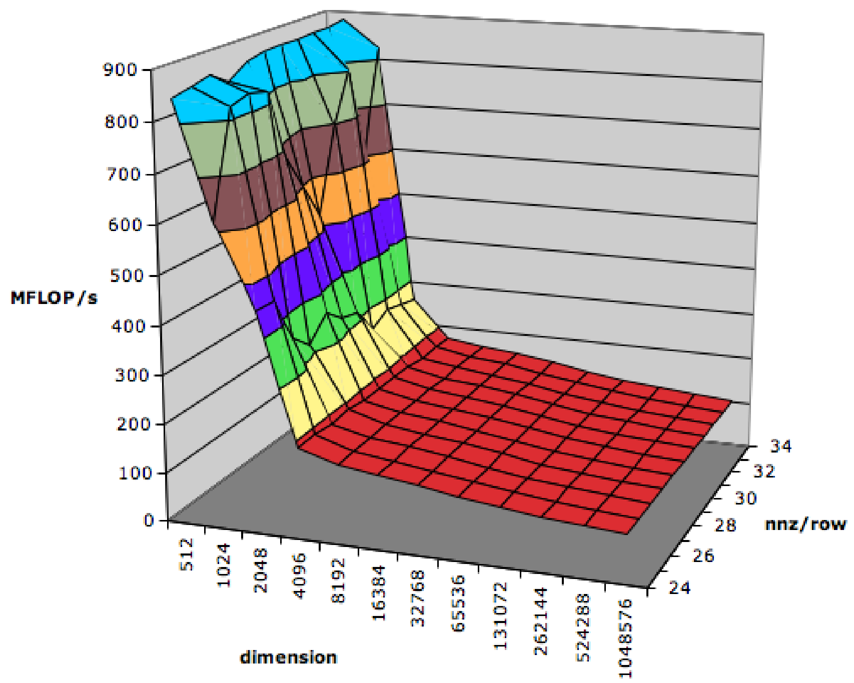

4x4 Benchmark Data, Pentium 3
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

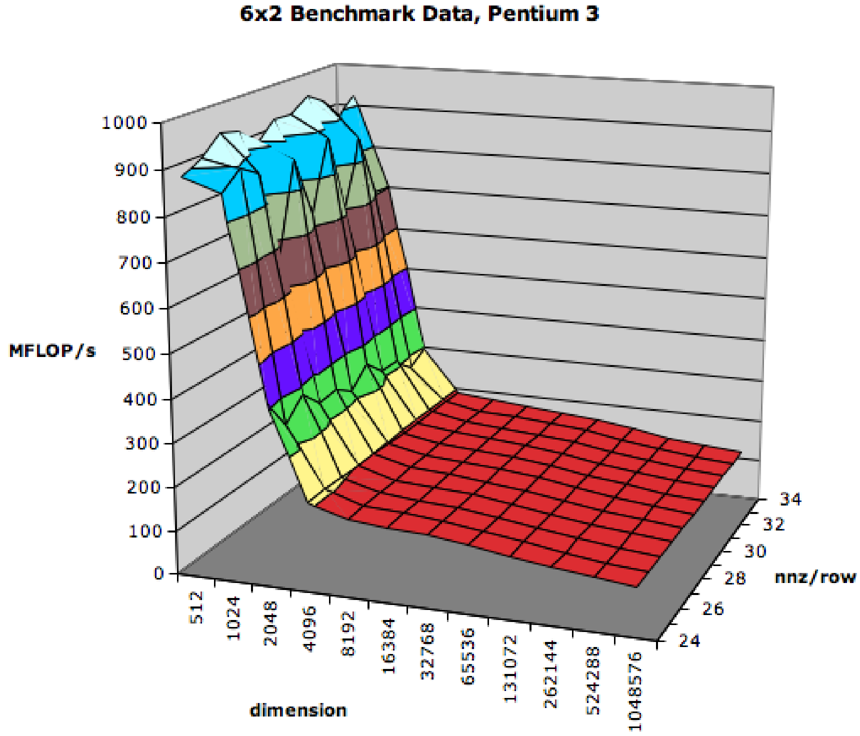

4x6 Benchmark Data, Pentium 3
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

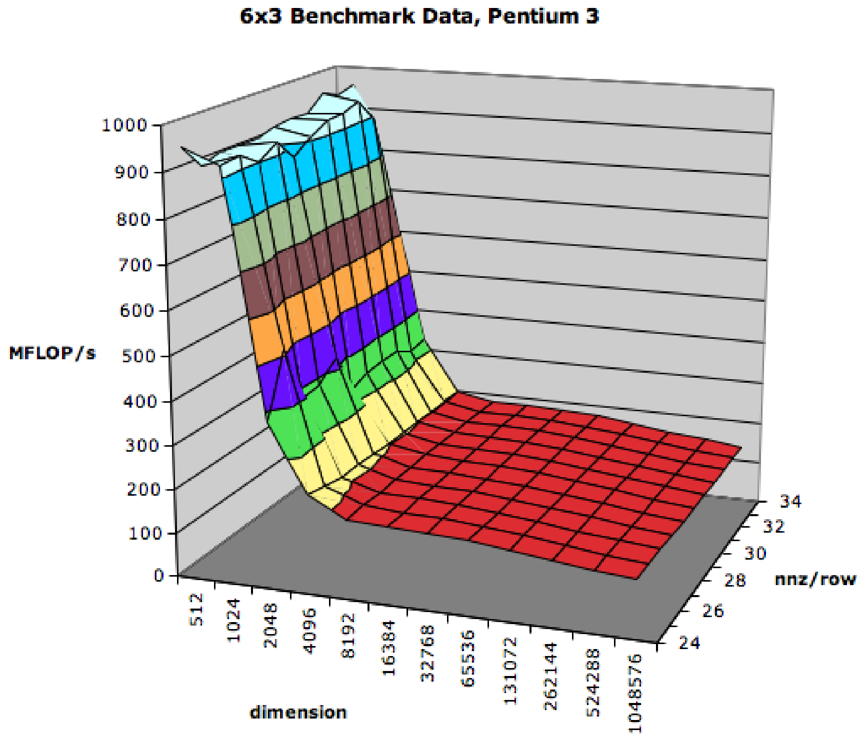
4x8 Benchmark Data, Pentium 3

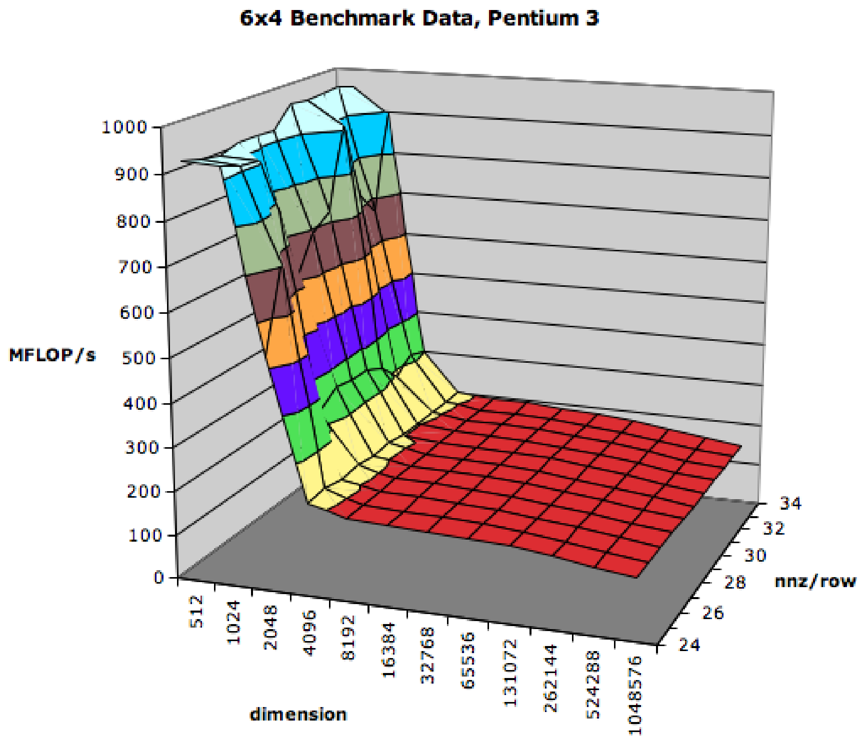


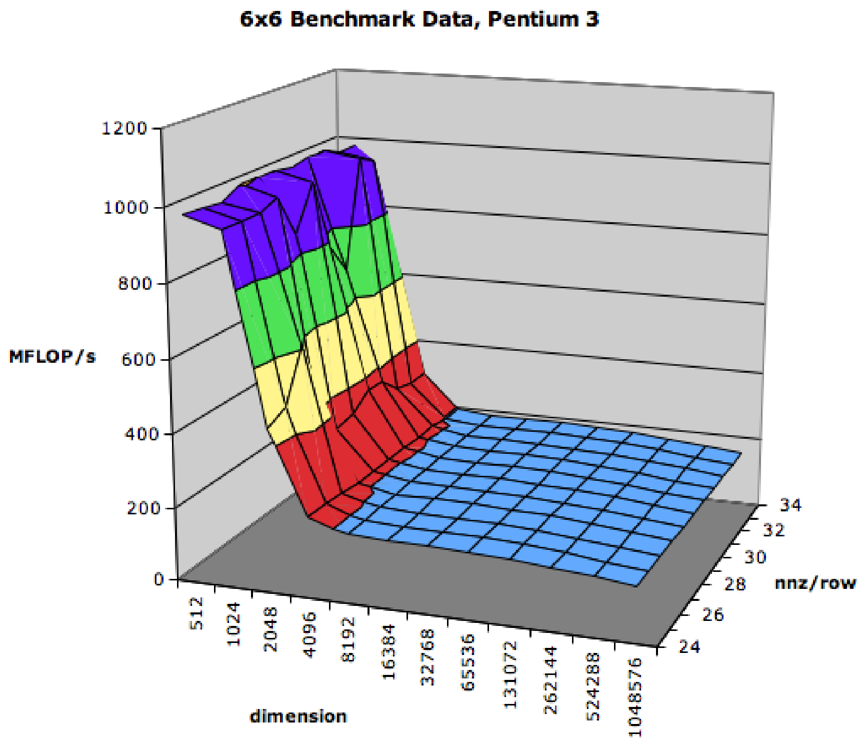
6x1 Benchmark Data, Pentium 3

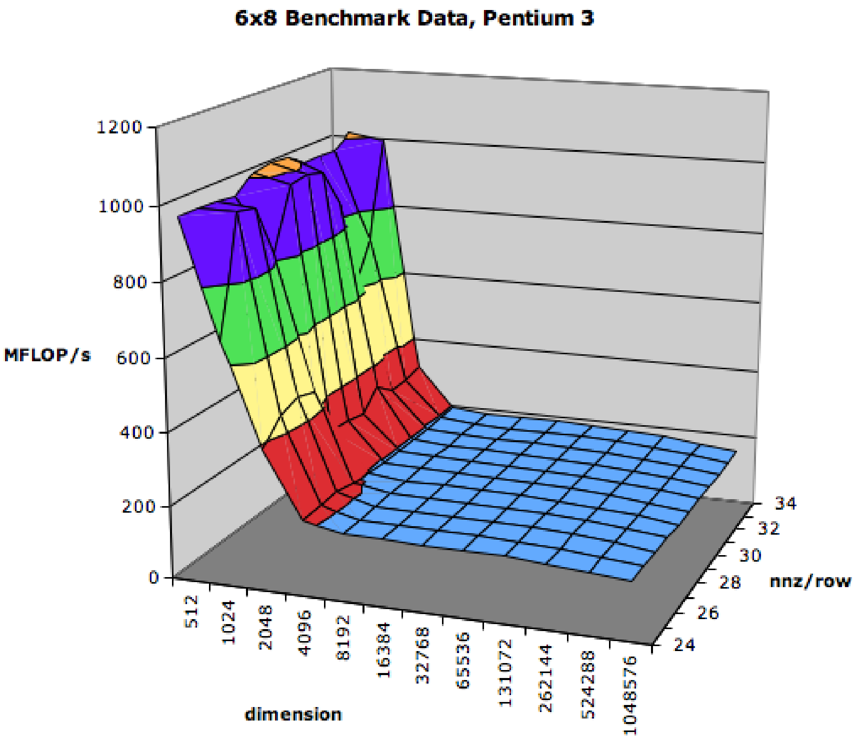


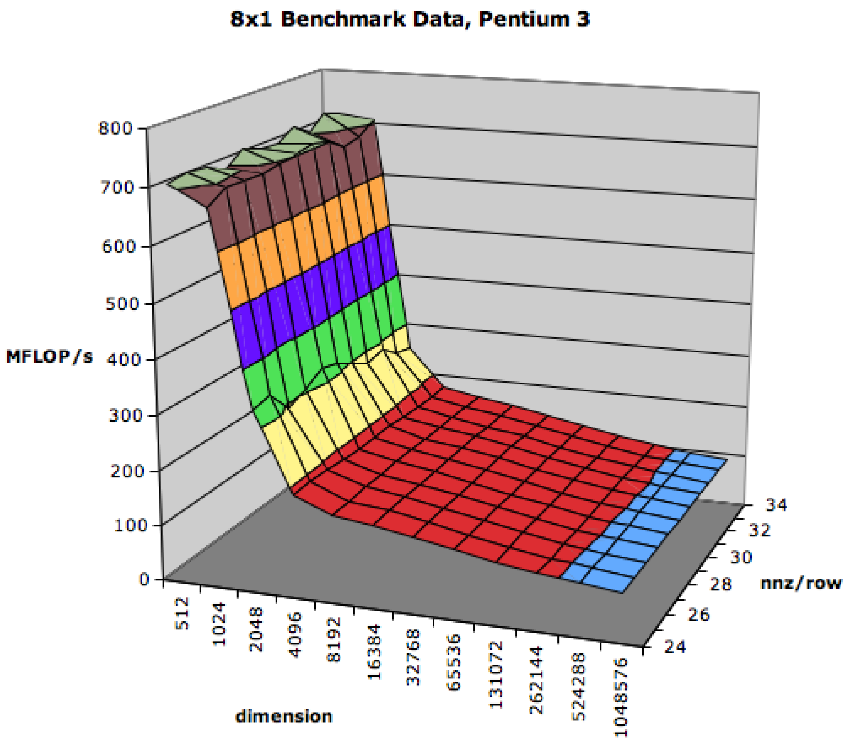

6x2 Benchmark Data, Pentium 3
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

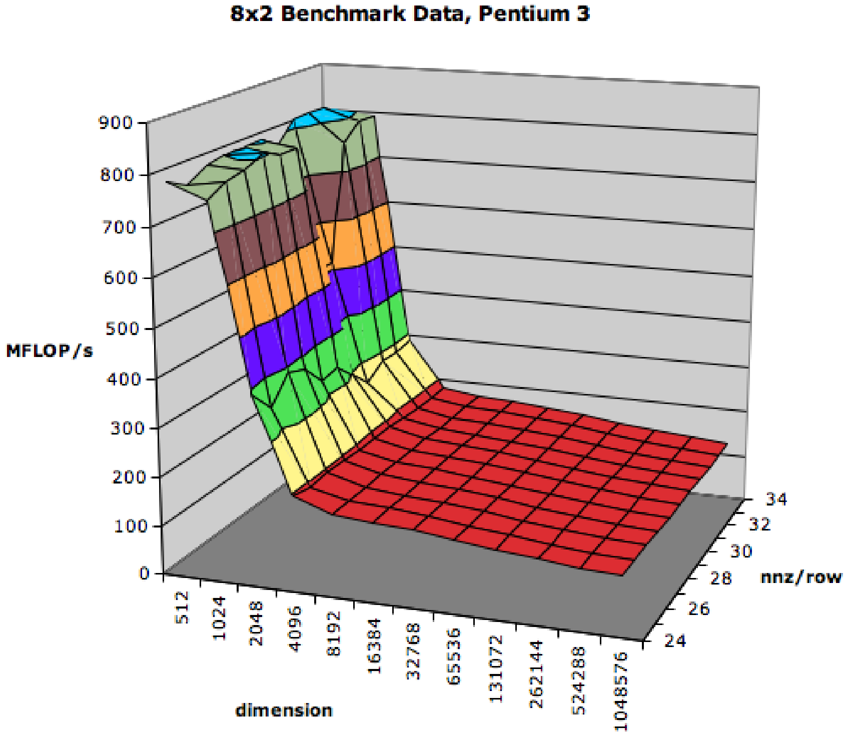

6x3 Benchmark Data, Pentium 3
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row

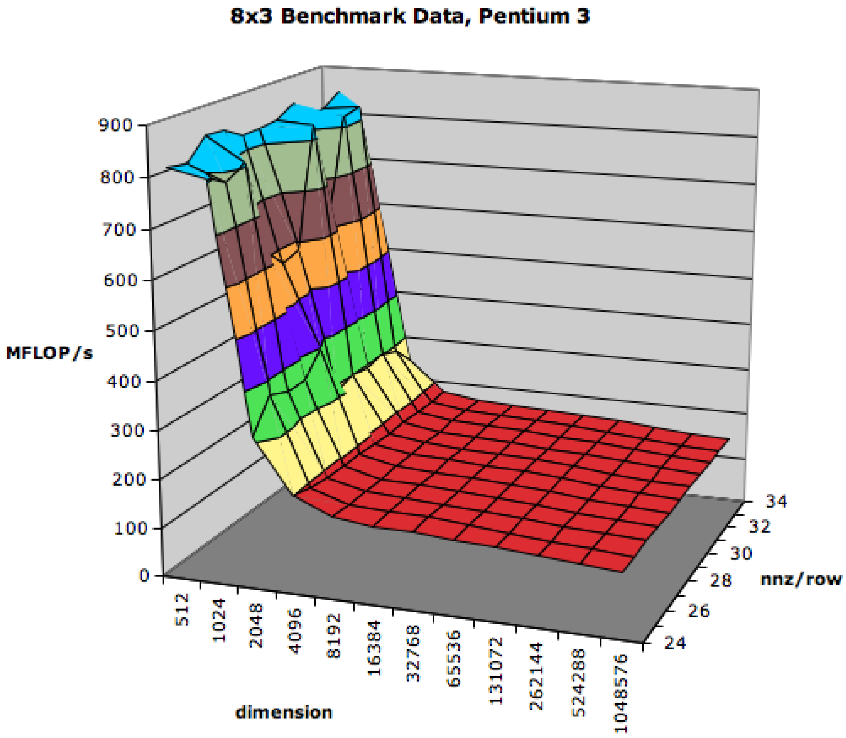

6x4 Benchmark Data, Pentium 3
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

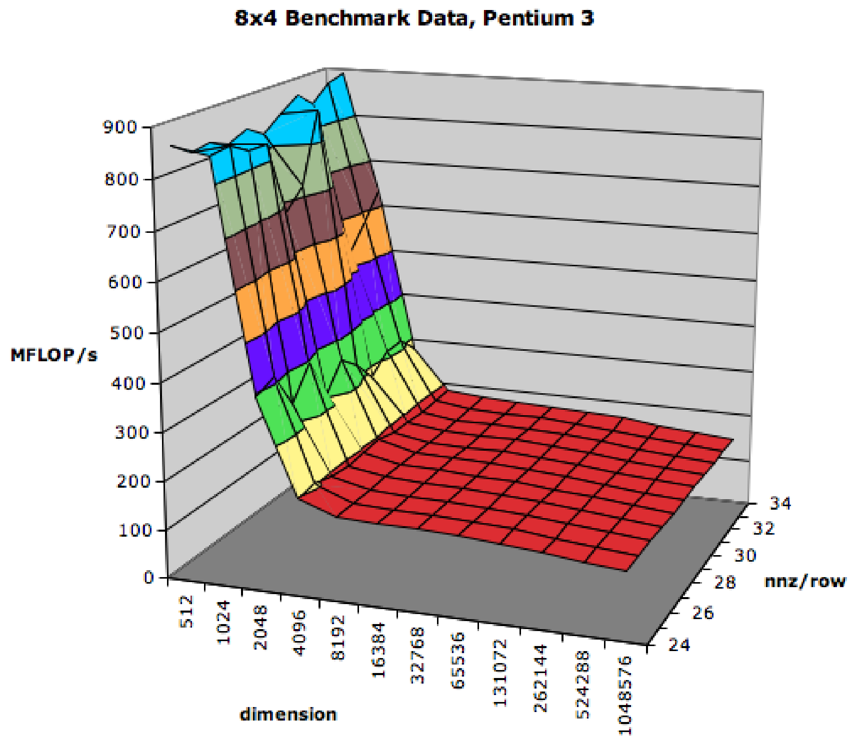

6x6 Benchmark Data, Pentium 3
MFLOP/s
1200
1000
800
600
400
200
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row

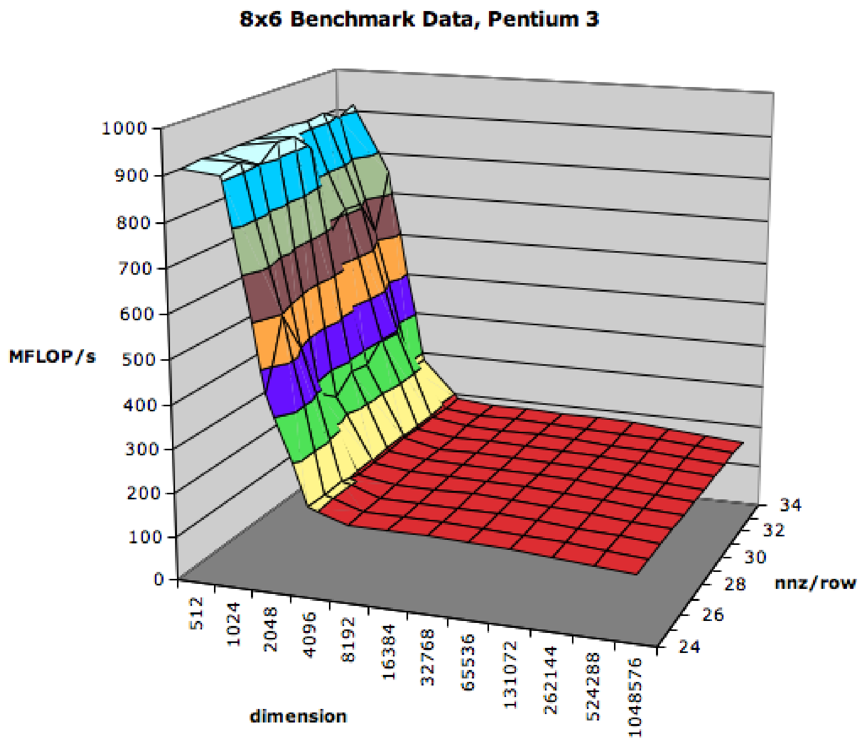

6x8 Benchmark Data, Pentium 3
MFLOP/s
1200
1000
800
600
400
200
0
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34

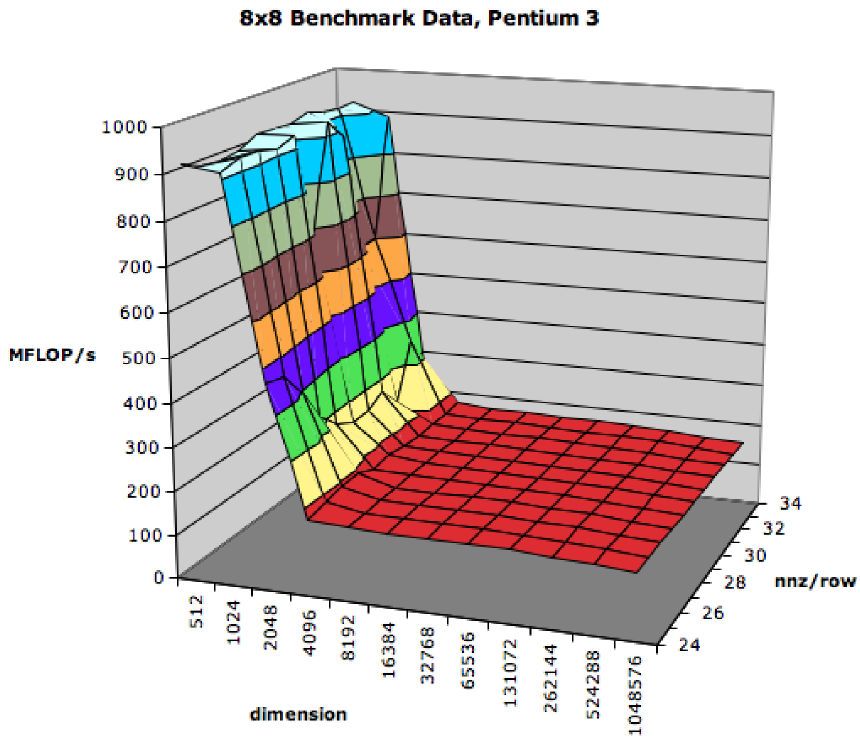

8x1 Benchmark Data, Pentium 3
MFLOP/s
800
700
600
500
400
300
200
100
0
dimension
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
nnz/row
24
26
28
30
32
34


8x2 Benchmark Data, Pentium 3
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row


8x3 Benchmark Data, Pentium 3
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
24
26
28
30
32
34
nnz/row


8x4 Benchmark Data, Pentium 3
MFLOP/s
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row


8x6 Benchmark Data, Pentium 3
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row


8x8 Benchmark Data, Pentium 3
MFLOP/s
1000
900
800
700
600
500
400
300
200
100
0
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
dimension
34
32
30
28
26
24
nnz/row