Benchmarking Sparse Matrix-Vector Multiply in Five Minutes

Hormozd Gahvari, Mark Hoemmen, James Demmel, and Katherine Yelick
Computer Science Division
University of California, Berkeley
Berkeley, California 94720
Email: {hormozd,mhoemmen,demmel,yelick}@cs.berkeley.edu

***Abstract*— We present a benchmark for evaluating the performance of Sparse matrix-dense vector multiply (abbreviated as SpMV) on scalar uniprocessor machines. Though SpMV is an important kernel in scientific computation, there are currently no adequate benchmarks for measuring its performance across many platforms. Our work serves as a reliable predictor of expected SpMV performance across many platforms, and takes no more than five minutes to obtain its results.**

I. Introduction

Sparse matrix-dense vector multiply (SpMV) is a common operation in scientific codes. It is especially prevalent in iterative methods to solve linear systems. Given this, it would be very convenient for consumers to have a convenient way of knowing which machine to buy for their calculations, and for vendors to know how well their machines perform.

There are currently no convenient ways for vendors to know how well their machines perform SpMV. The current standard method for ranking computers' ability to perform scientific compuatations, the Top 500 List [9], uses only the LINPACK benchmark [8]. LINPACK measures the speed of solution of a system of linear equations, which is not representative of all the operations that are performed in scientific computing. There is a benchmark suite under development called the High Performance Computing Challenge Suite (HPCC) that seeks to remedy this [5]. The HPCC suite contains benchmarks that seek to measure computers' performance in performing several different operations, including LINPACK.

The benchmark we will present here is proposed for inclusion into this suite, as none of the other benchmarks in it are suited for approximating the performance of SpMV. We will see why in the next section. One requirement for inclusion in the HPCC suite is a short run-time, which explains our goal of running in five minutes.

II. Approximating SpMV Performance

A critical difference between SpMV and other operations benchmarked in the HPCC suite is that the performance of SpMV depends strongly on the data, i.e. the size and nonzero pattern of the sparse matrix. Since practical sparse matrices do vary widely in these properties, this means that existing benchmarks will not be predictive of SpMV performance, and that we will need to time SpMV itself on a representative set of test matrices.

Furthermore, as machines grow in capacity over time, no fixed set of test matrices would be adequate to test performance. For example, a matrix so large that it cannot be stored in cache on today's platforms (an important size to test) may well fit in cache in a few years. This would make SpMV appear to run at a much higher fraction of peak performance, and be unrepresentative of practical problem sizes, which will also grow over time. This, combined with the sheer size of a collection of fixed test matrices, means we will have to generate appropriate test matrices on the fly that appropriately approximate practical sparse matrices.

Finally, SpMV performance can vary significantly depending on small changes in the data structure used to store the matrix, and in the corresponding algorithm that accesses it to implement SpMV. Just as the LINPACK benchmark depends on tuned BLAS for a true assessment of a machine's performance, we also need to engage in a reasonable level of tuning effort. To make this portable, fast and fair (in the sense that a similar level of machine-dependent and matrix-dependent tuning is done whenever the benchmark is used), we will rely on the OSKI automatic tuning system [11].

Now we explain the above points in more detail.

The main reason different matrix sizes and nonzero patterns impact the performance of SpMV is that they lead to different memory access patterns. Since only the nonzeros (and their locations in the matrix) are stored, memory accesses cannot all be unit-stride. Typically the matrix is still streamed through in unit stride, but the vector it multiplies (the source vector) is accessed indirectly. Our experiments lead us to propose using four matrix properties to characterize practical matrices, and to generate corresponding test matrices on the fly: size (total memory for the matrix and source vector), density (average number of nonzeros per row of the matrix), block size (can the matrix be stored as a collection of dense r-by-c blocks for some $r > 1$ and/or $c > 1$?), and "bandedness" (the distribution of the distances of the nonzeros from the main diagonal).

We propose 3 categories of SpMV problem sizes:

- Small: everything fits in cache.
- Medium: the source vector fits in cache but the matrix does not.
- Large: neither the source vector nor the matrix fit in cache.

These different sizes lead to different memory traffic patterns

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

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

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

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

Fig. 1. The Compressed Sparse-Row Matrix Storage Format

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

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

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

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

Fig. 2. The Blocked Compressed Sparse-Row (BCSR) matrix storage format.

TABLE I
PLATFORMS TESTED

	Pentium 4	Itanium 2	Opteron
Speed	2.4 GHz	1 GHz	1.4 GHz
Cache	512 KB	3 MB	1 MB
Compiler	gcc 3.4.4	icc 9.0	gcc 3.2.3

and different performance. Performance typically peaks for a given density for small or medium sizes, and gradually falls off as the problem dimension increases. Figures 3(a), 4(a), and 5(a) show this behavior in data gathered from running SpMV on a set of 275 matrices taken from the online collection [3]. Each circle is a test matrix, with its x-coordinate equal to its dimension, y-coordinate equal to its density, and color coded by speed in MFLOP/s. Dark lines separate the small, medium and large matrices. The selected matrices are listed in [6]. No performance tuning has been done on these matrices. The compressed sparse-row (CSR) format, is used for storing the matrices because it was found in [10] to be the best general-purpose unoptimized sparse matrix storage format across multiple platforms. An example of this format is seen in Figure 1. The nonzero entries are stored in the `values` array, the index of each entry that starts a new row is stored in the `row_start` array, and the column each entry belongs to is stored in the `col_idx` array. Data was obtained on the platforms in Table I.

None of the other benchmarks in the HPCC suite use indirect accesses that would let us predict the performance shown here. Indeed other research [6], [10], [12] has shown them to be unreliable as benchmarks for SpMV. [10] and [12] also looked at two other approaches for benchmarking SpMV. One was to develop performance bounds. The bounds served as very reliable upper and lower bounds across multiple platforms, but could not offer any hints as to what the expected performance would be on them. Another approach was to use "machine balance", but it was also inadequate on some platforms.

This motivates us to benchmark SpMV by performing the actual operation. There are three existing benchmarks that do this [1], [4], [7], but do not meet our goals. [7] and [4] do not measure the performance of any performance optimizations. [1] allows for performance optimizations, but they must be user-supplied. It also does not measure solely SpMV performance; rather, it measures the performance of the conjugate gradient operation, which is very rich in SpMV but also contains dense vector updates and outer products. Also [1] uses a single kind of random matrix that is not representative of many practical matrices.

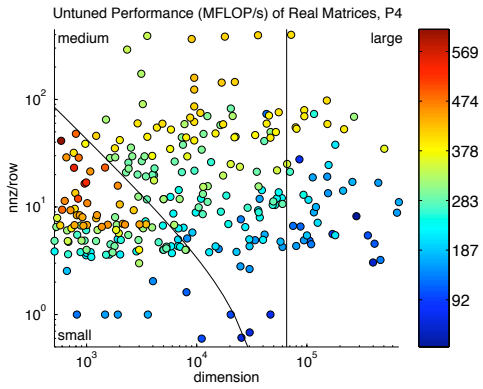
Now we consider block-sizes and performance tuning. Some sparse matrices, particularly those arising from finite element applications, have a natural block structure that can be exploited to improve performance. Others do not have such a structure and are only suited to being run without such optimizations.

We use the register blocking optimization [10] to measure optimized SpMV performance, as it was found in [10] to be the the most widely applicable of all the possible optimizations, and it is implemented in an automatic tuning system [11]. This requires us to use the blocked compressed sparse-row matrix storage format, which is illustrated in Figure 2 using 2×2 register blocks that are color-coded for clarity. The difference between BCSR and CSR is that in BCSR, the blocks are stored contiguously, the `row_start` array says which element starts the next block row, and the `col_idx` array says which block column each block belongs to. In general, it is possible to have blocks of an arbitrary blocksize $r \times c$. Different blocksizes work best with different matrices on different machines. The problem of which one is best is addressed in detail in [10].

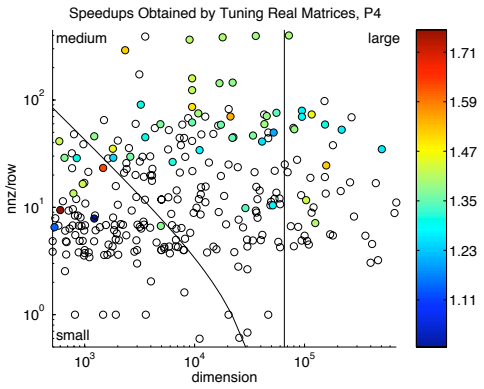
Figures 3(b), 4(b), and 5(b) show that register blocking SpMV on the matrices from the test suite yields different speedups on each of the platforms tested, highlighting the importance of measuring tuned in additioned to untuned SpMV. Register blocked performance data was obtained using the OSKI automatic tuning system [11].

Finally, we consider bandedness. By examining many practical sparse matrices, we find that many of them exhibit a somewhat banded structure in the following sense: a large fraction of the nonzeros in any row are located relatively close to the diagonal, as measured as the percentage of entries that lie in bands $10(i-1)$ to $10i$ percent away from the diagonal, where $1 \leq i \leq 10$). Figure II shows how we divide the matrices into bands.

The statistics for each matrix in our test suite are given in [6]. Ignoring this bandedness in generating random test

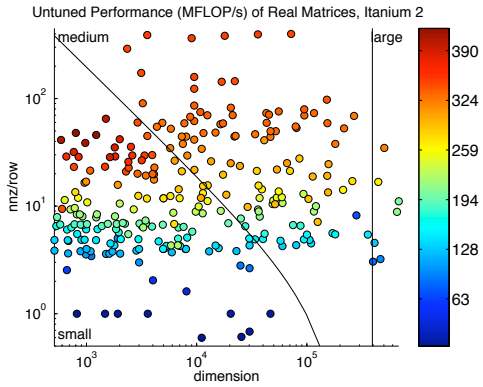


(a) Untuned SpMV performance

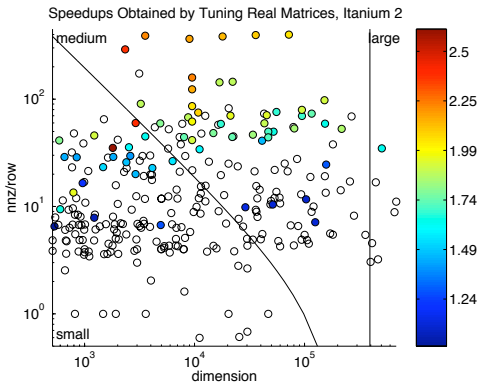


(b) Speedups obtained from tuning

Fig. 3. Small, Medium, Large behavior on the Pentium 4.

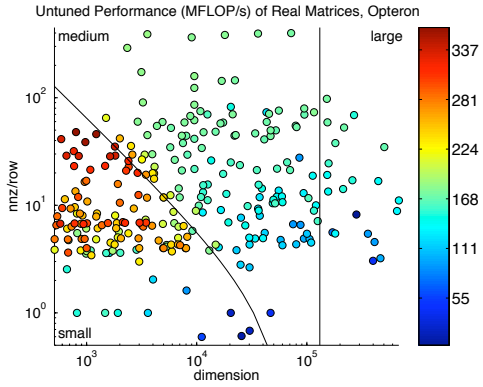


(a) Untuned SpMV performance

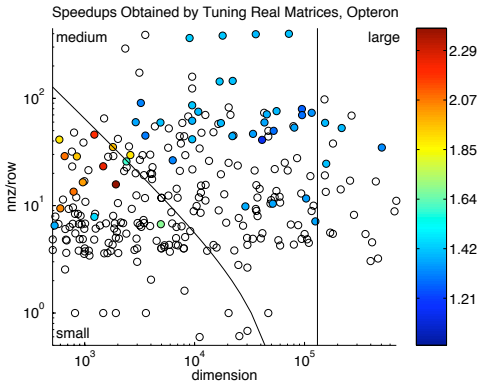


(b) Speedups obtained from tuning

Fig. 4. Small, Medium, Large behavior on the Itanium 2.



(a) Untuned SpMV performance



(b) Speedups obtained from tuning

Fig. 5. Small, Medium, Large behavior on the Opteron.

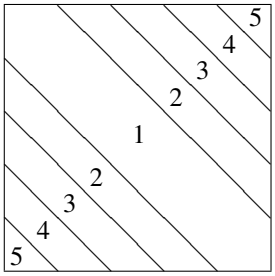


Fig. 6. Matrix divided up into bands. For simplicity of illustration, this matrix is only divided up into 5 bands instead of 10.

matrices tends to underpredict performance, so we generate our matrices to match the statistics in [6].

Randomly generated matrices matching all these criteria do a reasonably good job approximating the performance of the real-life matrices they were created to model, as Figures 8–10 show. In these plots, each real matrix is represented by an R, and is connected by a line to its synthetic counterpart, which is represented by an S. The tuned plots are color-coded by largest blocksize dimension.

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

III. The Benchmark

We will now use what we have developed in the past two sections to construct a benchmark for SpMV. Our first task is to define the set of matrices from which we will take data. The test suite we used in the previous section has dimensions ranging from 512 to nearly a million, with densities ranging from 1 to almost 400 nonzero entries per row. We will consider square matrices of similar dimensions, but only ones that are powers of two ranging from 2^9 to 2^{20}. The number of nonzero entries per row will range within $[24, 34] = 29 \pm 5$, since 29 is the average number of nonzero entries per row in the suite. The distribution of nonzero entries will made to match statistics taken from all of the matrices in the test suite, which are shown in Table II. The register blocksizes for optimized SpMV will come from the set $\{1, 2, 3, 4, 6, 8\} \times \{1, 2, 3, 4, 6, 8\}$. These were the blocksizes most commonly found in work in [10] that used a set of test matrices for which tuning showed benefits.

We take SpMV data from these matrices and report as output 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 tuned 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. Through these numbers, we seek to capture best-case and expected-case performance. Figure 7, which shows the performance of benchmark matrices within our search space for two particular blocksizes on the Itanium 2, illustrates why we select these numbers.

TABLE II
Distribution of Nonzero Entries in Matrix Test Suite

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 III
Benchmark Results, Full Run

	Unblocked		Blocked	
	Max	Median	Max	Median
Pentium 4	699	307	1961	530
Itanium 2	443	343	2177	753
Opteron	396	170	1178	273

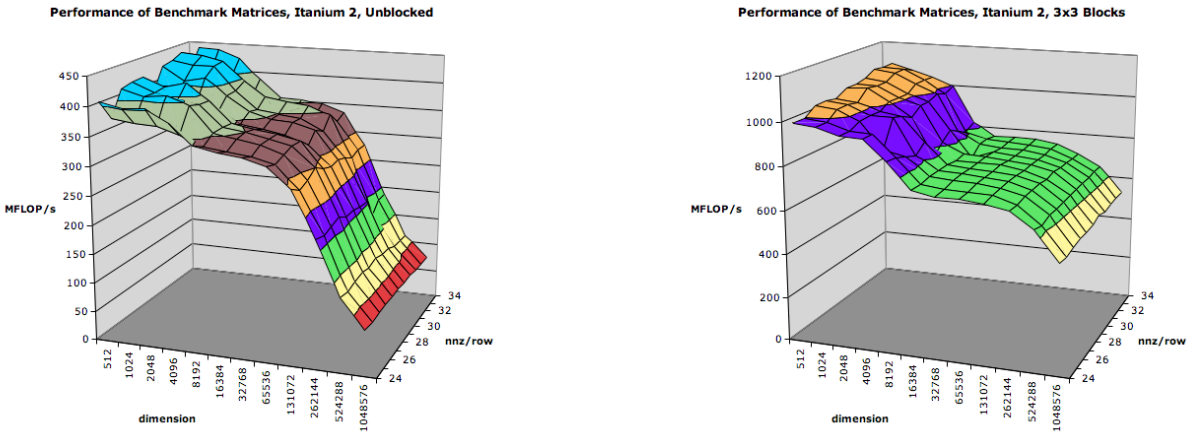
With the max numbers, we are looking to capture peak performance, and with the median numbers, we are looking to capture performance when it levels off. When forced to report one number, as required by the HPCC suite's rules, we will report the blocked median. Table III shows the output for the three platforms tested.

Figures 11–13 show that the numbers, especially the median numbers, give a good indicator of expected SpMV performance. Table IV reassures us that we tested small, medium, and large SpMV problems, and thus got a good set of data on which to base our benchmark numbers.

However, each of these runs took over 2 hours. If we want to cut this down, we will need to prune the test space run by the benchmark in such a way that we can capture the same data while running far fewer SpMV trials. Looking back at

TABLE IV
Proportion of Problem Sizes Tested by the Benchmark, Full Run

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



(a) Unblocked

(b) 3x3 Blocks

Fig. 7. Performance of benchmark matrices on the Itanium 2.

Figure 7, here are two key observations:

1) For large problem dimensions, the performance hardly changes with small fluctuations in nnz/row. This is not the case for small problem dimensions.
2) The performance levels off for large problem dimensions, so the statistics will hardly change if we drop the very largest problem dimensions from the test space.

To help us handle the first observation, we introduce 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 are to be tested for problem dimensions below the threshold dimension. Above it, we will be free to cut out values as we see fit.

The actual decision of which parts of the search space to cut out is made by a runtime estimator that first estimates the runtime of the benchmark on the entire search space and then cuts out certain parts of it until a user-specified time constraint is satisfied, in our case five minutes. The estimation is carried out by, for each register blocksize tested, running an SpMV trial (both matrix generation and performing the actual multiplication) for a matrix whose dimension is the threshold dimension as defined above and then doubling this value to obtain runtime estimates for running an SpMV trial for each successive problem dimension in the test space. Here, nnz/row is kept at the midpoint of the selected range. The computed estimates are then added up, yielding an estimate for the runtime of the entire benchmark.

The estimator then decides which elements of the search space to cut using the following iteration:

1) Reduce the number of values of nnz/row to test by 1 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.

TABLE V
BENCHMARK RESULTS, 5 MINUTE RUN

	Unblocked		Blocked	
	Max	Median	Max	Median
Pentium 4	692	362	1937	555
Itanium 2	442	343	2181	803
Opteron	394	188	1178	286

3) Repeat the previous two steps until the time limit is satisfied.

In this way, the largest problem dimension tested is kept as large as possible to ensure that the benchmark tests small, medium, and large problems as defined in the previous section, while still dramatically cutting down on the runtime, as we will see shortly. First, though, we note that cutting out values of nnz/row to test cuts out data points that we need to compute our statistics, since the full nnz/row range is going to be tested for problem dimensions below the threshold dimension. To correct this problem, we refill the data points by either duplication if we only test one nnz/row value or interpolation if we test more than one. In the latter case, the endpoints of the nnz/row range are required to be included among the nnz/row values tested.

In the case of each of the three tested platforms, the estimator reduced the maximum matrix dimension to 2^{18} and the number of nnz/row values to test to one for large enough problems. Table V shows that this resulted in dramatic savings in runtime while maintaining the accuracy of the data, Table VI shows the runtime savings, and Table VII shows the proportion of problem sizes tested by the shortened benchmark run. The difference in max numbers between the full and abbreviated runs, which would ideally be the same, fall within the bounds of measurement noise.

TABLE VI
BENCHMARK RUNTIME COMPARISON

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

TABLE VII
PROPORTION OF PROBLEM SIZES TESTED BY THE BENCHMARK, 5 MINUTE RUN

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

IV. CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK

We have presented a benchmark that quickly and effectively evaluates the fitness of different architectures for performing SpMV. The benchmark runs in at most five minutes and gives a good indicator of expected SpMV performnce on multiple platforms. There are however substantial areas for future work. These range from improving the benchmark itself to extending it to new platforms.

A. Improving the Benchmark

The most obvious question about the benchmark right now is the why the number it outputs for the tuned maximum is so high. For all the platforms tested, it is too high when compared with the performance of SpMV on real-life matrices. Future work to address this problem is needed. Another area where more work can be done is in the generation of synthetic matrices. We have identified four parameters (size, density, block-size, bandedness) that characterize the performance of test matrices, bu there are still gaps in the ability of synthetic matrices to model real-life ones, and closing these gaps will help lead to a more accurate benchmark.

Another case the benchmark does not currently handle explicitly is that of symmetric matrices. Many matrices from real-life applications are symmetric. Figures 13–15 show that our benchmark retains some predictive power when symmetry is taken into account, but there are many symmetric matrices for which it could do better. Thus, finding a way to integrate symmetric matrices into the ones used by our benchmark to run its trials would very much improve it.

B. Extending the Benchmark to New Platforms

Our benchmark currently only works on scalar uniprocessor machines. These are not the only machines on which SpMV is performed. Vector and parallel machines are also common platforms on which SpMV is run, making a benchmark designed for those kinds of architectures very useful. In the case of vector machines, there are sparse matrix data structures specifically created for them that should be used, such as segmented scan [2]. So the benchmark can be extended to vector machines by changing the data structure used for storing sparse matrices to one that is optimized for vector machines.

In the case of parallel machines, many more issues come into play. The matrix, instead of belonging to just one processor, is instead distributed over many processors. This will require a whole new benchmark, and one that will be sought after by many, as the benchmarks in the HPCC suite expects all of its benchmarks to have parallel versions [5].

ACKNOWLEDGMENT

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 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.

REFERENCES

[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," RNR, Tech. Rep. 94-007, March 1994.

[2] G. Belloch, M. Heroux, and M. Zagha, "Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors," Carnegie Mellon University, Tech. Rep. CMU-CS-93-173, 1993.

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

[4] J. Dongarra, V. Eijkhout, and H. van der Vorst. (2001) Sparsebench: a sparse iterative benchmark. [Online]. Available: http://www.netlib.org/benchmark/sparsebench

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

[6] H. Gahvari, "Benchmarking Sparse Matrix-Vector Multiply," Master's thesis, University of California, Berkeley, December 2006.

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

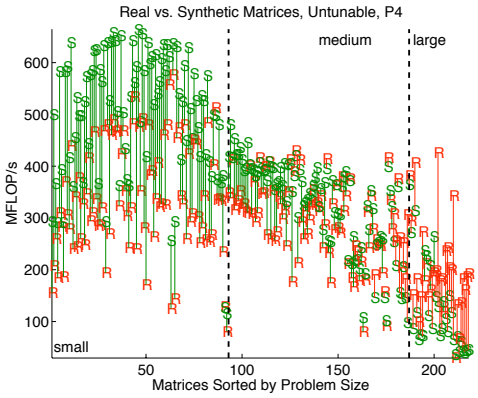
[8] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. (2004) HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. [Online]. Available: http://www.netlib.org/benchmark/hpl

[9] Top500 Supercomputer Sites. [Online]. Available: http://www.top500.org

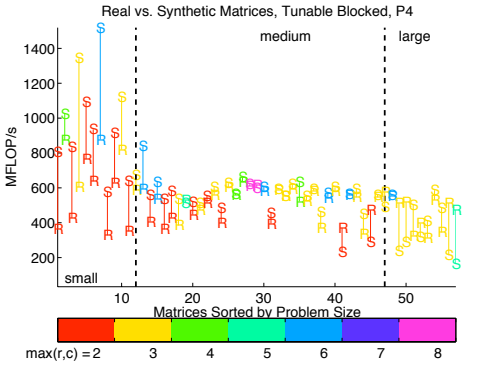
[10] R. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, University of California, Berkeley, December 2003.

[11] R. Vuduc, J. Demmel, and K. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proceedings of SciDAC 2005*, ser. Journal of Physics: Conference Series. San Francisco, CA, USA: Institute of Physics Publishing, June 2005, (*to appear*).

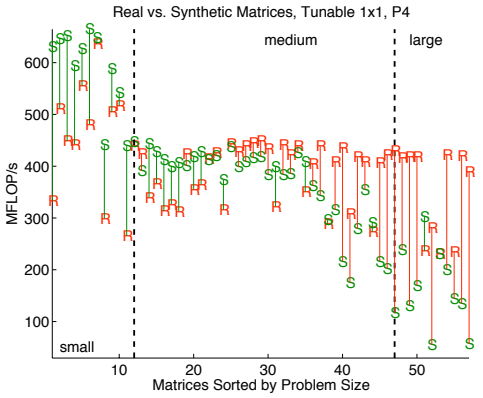
[12] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply," in *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.



(a) Synthetic vs. untunable real matrices

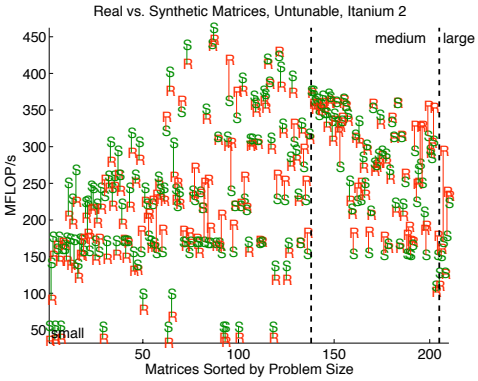


(b) Synthetic vs. tuned real matrices

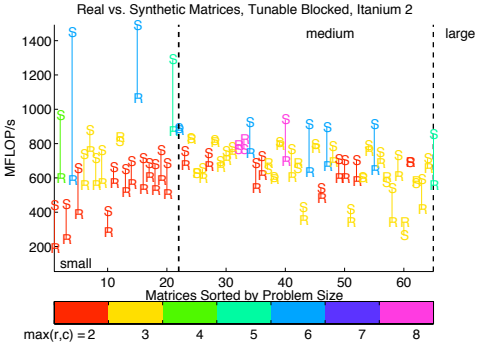


(c) Synthetic vs. tunable real matrices run 1x1

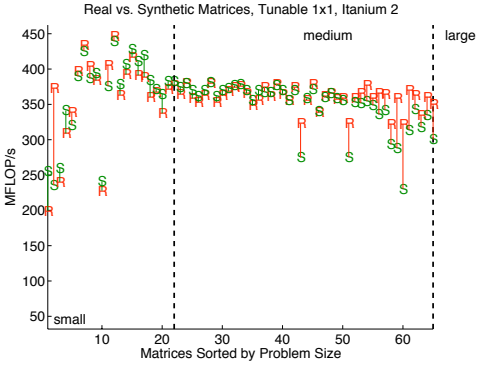
Fig. 8. Performance of synthetic vs. real matrices on the Pentium 4.



(a) Synthetic vs. untunable real matrices

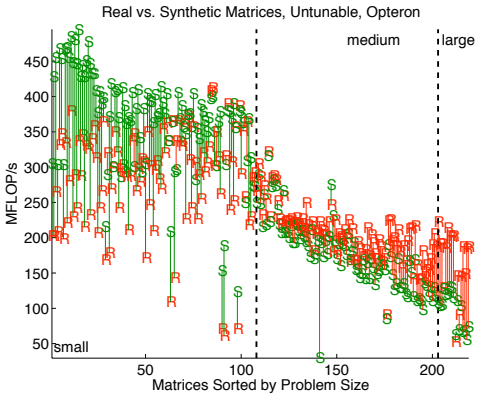


(b) Synthetic vs. tuned real matrices

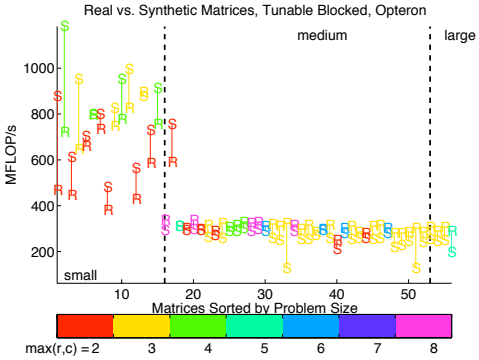


(c) Synthetic vs. tunable real matrices run 1x1

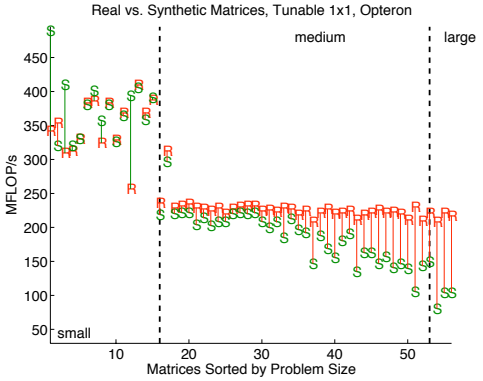
Fig. 9. Performance of synthetic vs. real matrices on the Itanium 2.



(a) Synthetic vs. untunable real matrices

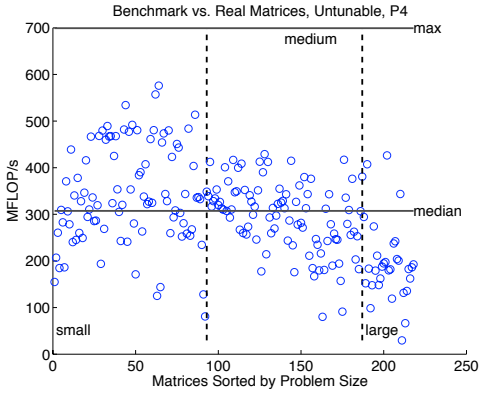


(b) Synthetic vs. tuned real matrices

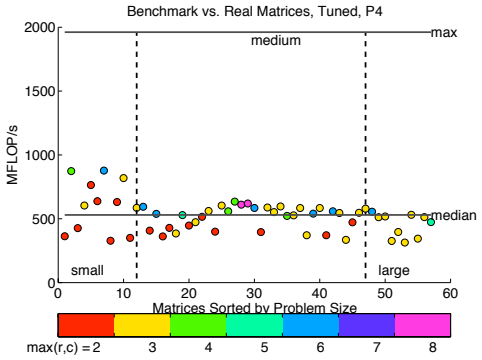


(c) Synthetic vs. tunable real matrices run 1x1

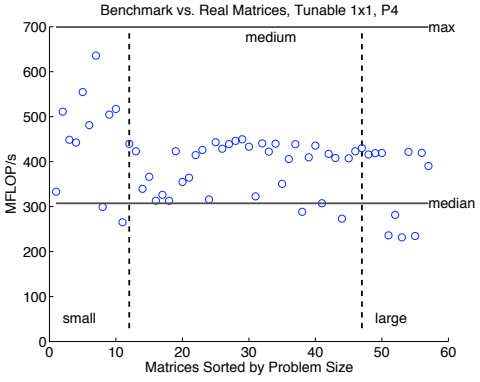
Fig. 10. Performance of synthetic vs. real matrices on the Opteron.



(a) Benchmark vs. untunable real matrices

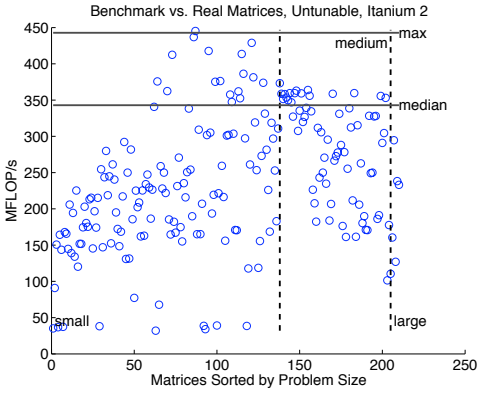


(b) Benchmark vs. tuned real matrices

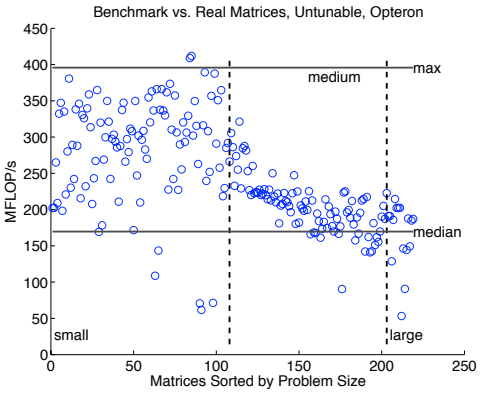


(c) Benchmark vs. tunable real matrices run 1x1

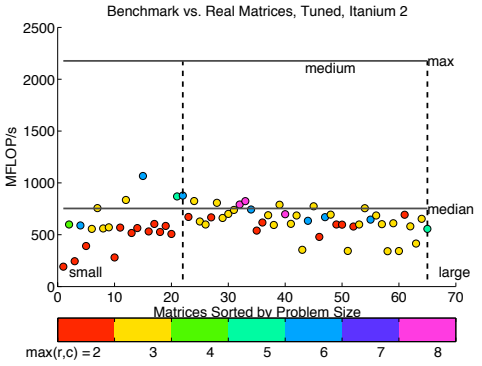
Fig. 11. Performance of benchmark vs. real matrices on the Pentium 4.



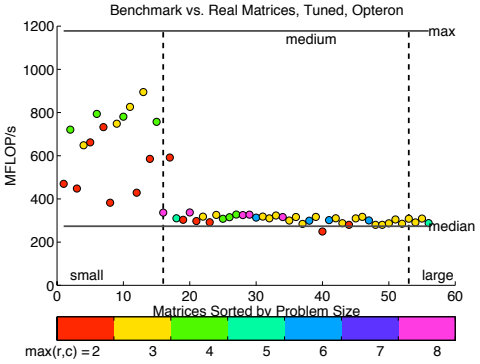
(a) Benchmark vs. untunable real matrices



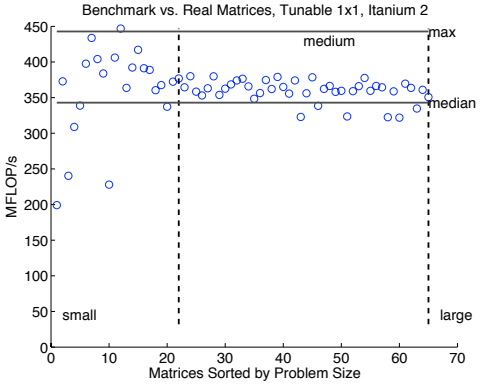
(a) Benchmark vs. untunable real matrices



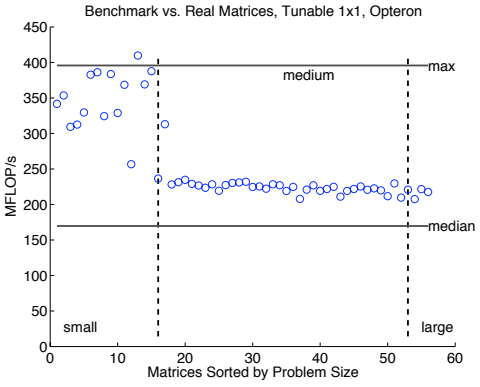
(b) Benchmark vs. tuned real matrices



(b) Benchmark vs. tuned real matrices



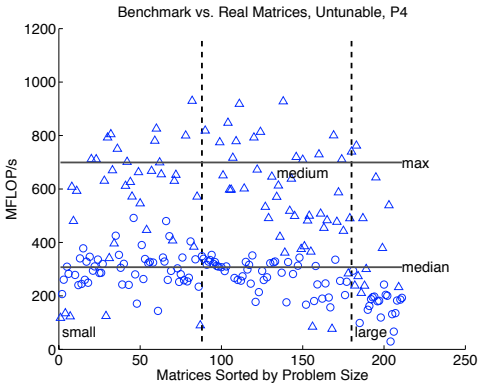
(c) Benchmark vs. tunable real matrices run 1x1



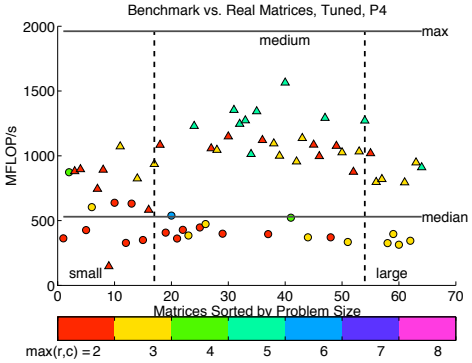
(c) Benchmark vs. tunable real matrices run 1x1

Fig. 12. Performance of benchmark vs. real matrices on the Itanium 2.

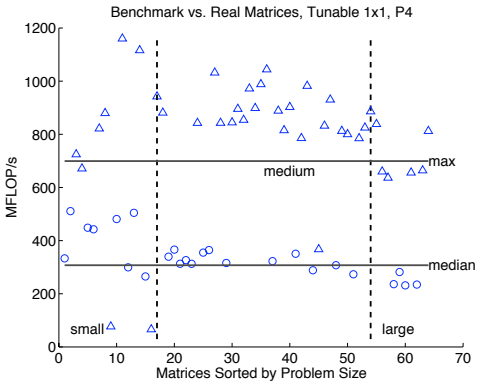
Fig. 13. Performance of benchmark vs. real matrices on the Opteron.



(a) Benchmark vs. untunable real matrices

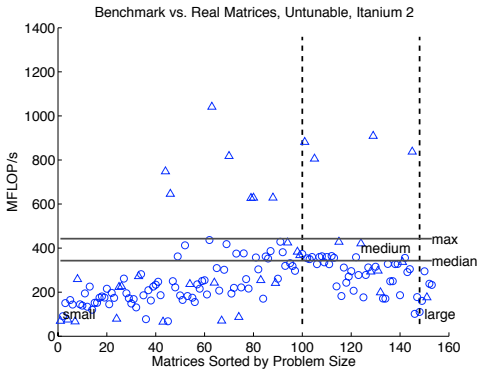


(b) Benchmark vs. tuned real matrices

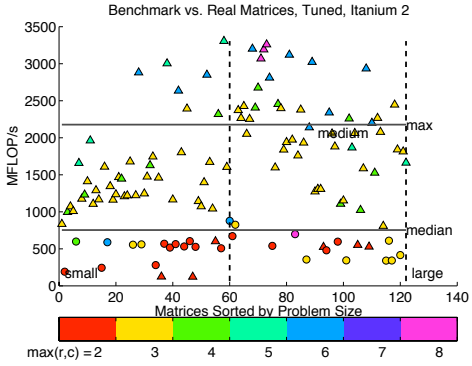


(c) Benchmark vs. tunable real matrices run 1x1

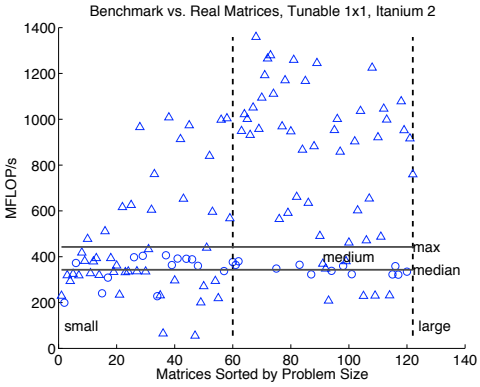
Fig. 14. Performance of benchmark vs. real matrices on the Pentium 4 with symmetry taken into account. Triangles represent symmetric matrices and circles represent nonsymmetric ones.



(a) Benchmark vs. untunable real matrices

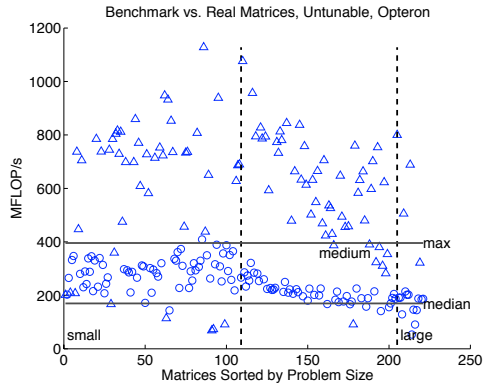


(b) Benchmark vs. tuned real matrices

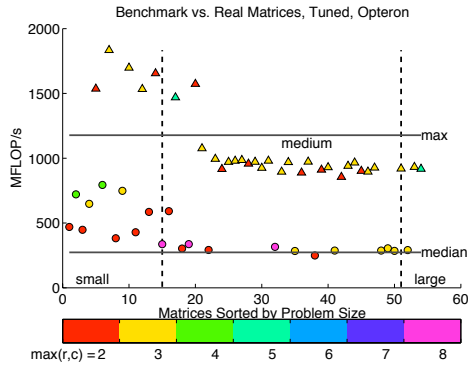


(c) Benchmark vs. tunable real matrices run 1x1

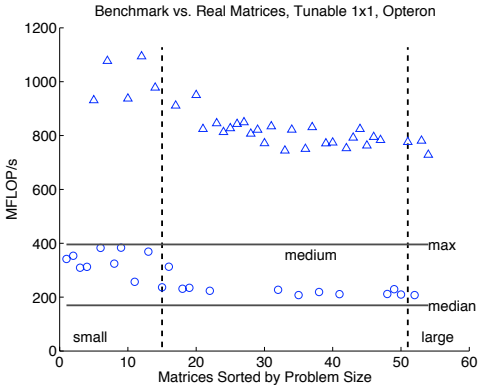
Fig. 15. Performance of benchmark vs. real matrices on the Itanium 2 with symmetry taken into account. Triangles represent symmetric matrices and circles represent nonsymmetric ones.



(a) Benchmark vs. untunable real matrices



(b) Benchmark vs. tuned real matrices



(c) Benchmark vs. tunable real matrices run 1x1

Fig. 16. Performance of benchmark vs. real matrices on the Opteron with symmetry taken into account. Triangles represent symmetric matrices and circles represent nonsymmetric ones.