A BENCHMARK FOR REGISTER-BLOCKED SPARSE MATRIX-VECTOR MULTIPLY

HORMOZD GAHVARI AND MARK HOEMMEN

Abstract. We develop a sparse matrix-vector multiply (SMVM) benchmark for block compressed sparse row (BSR) matrices. These occur frequently in linear systems generated by the finite element method (FEM), for example, and are naturally suited for register blocking optimizations. Unlike current SMVM benchmarks, ours accommodates algorithms systematically tuned for particular platforms, and it accomodates adjustment of data set sizes to fit the data at the appropriate levels of the memory hierarchy. Register block sizes are varied to capture both unoptimized (no register blocking) and optimized performance on each architecture tested. Our randomly generated test cases successfully predict SMVM performance for FEM and similar matrices encountered in practice. To demonstrate the applicability of the benchmark, we use it to evaluate the effectiveness of the SSE2 SIMD floating-point instruction set.

1. Introduction

Sparse matrix-vector multiply (SMVM) comprises the bulk of computation for many scientific, industrial and statistical applications. Sparse matrix data structures save storage space when a significant fraction of the entries of a matrix are zeros. In many applications, the matrix would not fit in main memory if stored in dense format. Furthermore, a data structure that explicitly accesses only the nonzero elements should speed up matrix-vector multiplication, since the zero entries do not affect the product. However, the arrangement of memory accesses required by a sparse matrix operation adversely affects performance. An algorithm has complete freedom to access any entry of a densely stored matrix. This permits blocking and tiling rearrangements of matrix-vector and matrix-matrix multiplications, which help improve locality of data accesses. Caching, prefetching and other hardware optimizations can thus circumvent or amortize the costs of long memory latencies. The Basic Linear Algebra Subprograms (BLAS) describe matrix-vector and matrix-matrix operations as Level 2 and Level 3, respectively, to reflect the increased degree of potential optimization latent in each type of operation (see e.g. [11]). In practice, dense (especially Level 3) operations tend to run at a high fraction of the peak floating-point operations rate.

In contrast, most sparse matrix formats require (algorithmically) sequential accesses to the elements of the matrix, in a fixed pattern. In addition, using an individual element may require multiple lookups in a data structure, e.g. in order to access the index of that element. As a result, performance is often limited by load latency. Other than speeding up the memory system, one possibility is to adjust the sparse data structure and SMVM algorithm itself, to take advantage of dense locality on a small scale. This is the approach taken by the Sparsity automatic tuning system, which will be summarized below [15]. Our benchmark uses SMVM algorithms generated by Sparsity.

1.1. Unoptimized and optimized sparse matrix formats.

Although many sparse matrix storage formats are in use, the compressed sparse row (CSR) form appears frequently. For a description of the format as we use it, see Im [15]; also refer to Saad [20, pp. 84–5] and for a variation on the format, the Sparse BLAS proposal [2]. In brief, for an $m \times n$ matrix, the NNZ nonzero values are stored contiguously in a single array, values []. Their corresponding column indices are stored in an array, also of length NNZ, called col_idx []. Finally, the i^{th} element of the length-m array row_start [] gives the index of the col_idx [] and values [] arrays at which the first nonzero element of the i^{th} row can be found. Matrix-vector multiplies are arranged as dot products, and thus accesses to all three arrays representing the matrix are contiguous and sequential.

SMVM with CSR matrices suffers from the aforementioned performance limitations of sparse matrix-vector multiplication algorithms. Adjusting the storage format to include small dense blocks could allow exploitation of some dense locality techniques on each block. Sparsity's strategy involves a heuristic estimation of the optimal block dimensions $r \times c$ for a particular processor. The storage format it uses, block compressed sparse row (BSR), is based on CSR, but each "element" of the matrix referred to by the row_start [] and col_idx [] arrays is an $r \times c$ block. Thus, row_start [] has only m/r elements and col_idx [] only n/c elements (which saves integer storage and memory lookups). The column index of each block is

the index of its upper left hand corner. In the SPARSITY system, the elements in a block are stored contiguously in the values [] array in row-major order. If the first block is as follows:

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix},$$

then the first four entries of the values [] array are a_{00}, a_{01}, a_{10} and a_{11}. The Sparse BLAS proposal uses column-major order, and also proposes a non-contiguous form that is better suited for vector computers, rather than the cache-based microprocessors we use. We chose row-major form for compatibility with SPARSITY.

Storing blocks rather than individual entries does waste some storage and unnecessary arithmetic operations, since a block may include zero entries that in CSR format would not be stored. The *fill ratio*, the ratio of stored entries to actual nonzeros, measures this waste. However, many calculations with a high degree of locality generate matrices with a natural small block structure. This is characteristic of finite difference and finite element methods for elliptic and parabolic partial differential equations, for example, which are a frequent case in the sciences and industry. Typical natural block dimensions are 2×2, 3×3 and 6×6, or multiples of these. SPARSITY offers a low-cost heuristic for determining the optimal block size, given a sample of a particular matrix and a set of machine parameters. The resulting speedups are significant, and sometimes twofold or more.

The results of SPARSITY suggest that in many cases, the potential performance of SMVM is much greater than that anticipated by only testing SMVM with CSR storage. Just as tuned dense linear algebra routines have usurped the "flop" as a unit of work[1], tuned SMVM routines should be included as part of a total package for benchmarking a processor's SMVM performance.

2. Pre-existing SMVM benchmarks

Evaluating SMVM performance may take one of at least three approaches. The first method times SMVM (or algorithms based on SMVM), using a set of matrices that is characteristic of those commonly encountered in applications. SciMark 2.0 and SparseBench follow this scheme [19, 10]. The second approach uses simpler loops that resemble SMVM operations and may correlate with SMVM performance; examples include STREAM Triad and indirect indexed variants thereof [17]. The third technique uses memory hierarchy parameters and a mathematical or heuristic performance model. Examples of each of these methods are discussed below. We chose the first of them, because the model-based approach does not evolve well with future developments in memory hierarchies (the formulae themselves, not just the parameters, must be changed), and because we found that "simpler" benchmarks do not capture the range of behavior of the actual SMVM operation.

[1]With the incorporation of LAPACK with tuned BLAS into Matlab version 6, Matlab no longer counts flops, for example.

Platform	Mflops/s	Load avg.
AMD Athlon	54.01	1.00
Itanium 2	59.48	2.33
Pentium 3	44.76	11.22
Pentium M	156.57	0.05
Pentium 4	290.08	0.03

TABLE 1. SciMark 2.0 SMVM performance. Load average obtained with UNIX "uptime" command. Pentium M and Pentium 4 are single-user workstations, Athlon is a lightly used server, and the other machines are more heavily used.

2.1. SMVM benchmarks that do SMVM.

SciMark 2.0, produced by NIST, is a set of scientific computing benchmarks that includes an example SMVM calculation [19]. Its primary purpose appears to be comparing Java and C performance for typical numerical algorithms. SciMark uses a matrix in CSR format. The "large" version of the benchmark tests a $10^4 \times 10^4$ matrix with 10^6 nonzeros (a size chosen so that the source and destination vectors, but not the matrix itself, fit in the largest cache of most workstation processors available at the time of release). The matrix has a particular sparsity pattern that varies the stride of memory accesses to each row. Since the SMVM benchmark uses CSR alone, it fails to account for machine-dependent optimizations such as register blocking. Results are in Table 1.

SparseBench, a "benchmark suite of iterative methods on sparse data," also does not exploit algorithmic optimizations of SMVM [10]. Since the 1×1 block case usually executes more slowly than all other block sizes in BSR SMVM, both SciMark and SparseBench do offer a lower bound on performance. Furthermore, many types of matrices, such as those produced by latent semantic indexing (LSI), lack a regular block structure and are more efficiently stored and manipulated in CSR format. An additional problem with both benchmarks is their fixed data set size, which requires that they

Loop name	Kernel
Copy	a[i] = b[i];
Scale	b[i] = scalar * c[i];
Sum	c[i] = a[i] + b[i];
Triad	a[i] = b[i]+scalar*c[i];

TABLE 2. Loop kernels for STREAM benchmarks. a[], b[] and c[] are double-precision arrays, and scalar is a double-precision scalar. Fixed array length $N = 2 \times 10^6$ is chosen so the data cannot all fit in cache.

Loop name	Kernel
Sum	s += a[i];
Dot	s += a[i] + b[i];
Load Sparse Matrix	s += a[i]; k += ind[i];
SMVM (on-chip cache)	s += a[i] * x[ind[i]];
SMVM (external cache)	s += a[i] * x[ind[i]];

TABLE 3. Vuduc microbenchmark suggestions for SMVM [25]. a[], b[] and x[] are double-precision arrays, and ind[] an array of integer indices. x[] has length one-third the size of either the largest on-chip cache or the largest external cache (as benchmark name indicates). The other vectors are sized at least ten times larger than largest cache.

be modified over time to keep pace with cache size increases.

2.2. **Simplified array operations: STREAM and variants.** The STREAM benchmarks measure what the author John McCalpin calls "sustainable memory bandwidth," incontrast to the "peak memory bandwidth" cited by chipset manufacturers [17]. STREAM consists of four computational loops: Copy, Scale, Sum and Triad, which are shown in Table 2. The STREAM loops mimic typical dense vector and matrix-vector operations. For example, Triad resembles dense matrix-vector multiplication. McCalpin also shows that Triad correlates well with the SPEC2000 `swim` benchmark, which is a dense computational fluid dynamics code with a high ratio of memory accesses to floating-point arithmetic. Furthermore, all the STREAM loops are written so there is no data reuse within a single benchmark. Thus, they seem like a natural model of CSR SMVM.

Unfortunately, the results of STREAM do not correlate well with SMVM tests using matrices from applications. Authors such as Vuduc have observed that STREAM does not accurately predict memory bandwidth for sparse matrix-vector multiplication in practice [25]. This observation inspired Vuduc to create his own benchmarks that reflect the indexed, indirect memory accesses encountered in SMVM. They are listed in Table 3. For the four platforms tested (Sun UltraSPARC IIi, Pentium III, IBM POWER3 and Itanium 1), the on-chip cache version of SMVM corresponds to the mean performance of SMVM over various block dimensions, but the best performance of SMVM for any pair of block dimensions always outperforms this benchmark. Further testing is needed to determine whether Vuduc's loops offer a simpler alternative to our SMVM benchmark.

2.3. **Performance models.** Constructing a model of SMVM, based on machine parameters such as cache and TLB sizes, associativities and latencies, and page size, can establish lower and upper bounds on performance. For example, Vuduc's recent work demonstrates the predictive power of bounds based on cache misses and latencies, without considering the TLB or page faults [25]. His experiments suggest that STREAM is not a good predictor of SMVM performance: streaming performance for large arrays is more related to memory bandwidth, but SMVM appears more dependent upon cache and memory latencies than memory bandwidth.

What we most need is a more sophisticated model for determining a minimal data set size that produces consistent results across many processors. Our model should be supported by hardware counts of cache and TLB misses and page faults. We plan to use platform-independent performance counter interfaces such as PAPI whenever possible [3]. We need access to data on TLB misses and page faults because these have the most potential to disturb the expected load and store latencies that allow us to compare results.

2.4. **Memory hierarchy characterization and performance models.** There are tools available for automatic characterization of a processor's memory hierarchy. Their output alone does not suffice to gauge a machine's SMVM performance, although it could provide the necessary parameters for a mathematical or heuristic model that predicts SMVM performance. Saavedra introduced a set of "microbenchmarks" which uses STREAM-like array accesses, but varying the array sizes and access strides, to probe the memory hierarchy [22, 21]. The Memory Access Patterns (MAPS) code carries out similar operations on a single node of a multiprocessor, and the results are used to predict

parallel performance for a wide range of scientific applications [23]. Even computer architecture textbooks have examples of such codes (see [13, pp. 513-515], for example).

Analyzing the output of MAPS-like codes can be used to determine a processor's memory hierarchy traits. However, the additional complexity of modern memory hierarchies has made this task more difficult. We are considering using a similar code so that our benchmark can guess cache sizes. User input of these parameters is prone to error, since they often differ across different revisions of a processor. Furthermore, the MAPS benchmark itself (which appears the best maintained of the aforementioned codes) requires manual modifications for the specific processor. It may even require the inclusion of preprocessor directives and/or inline assembly, in order to activate features such as prefetching. This would defeat our ideal goal of creating a platform-independent benchmark that can run with little or no user input of machine parameters.

3. Experimental procedure

Our benchmark uses the BSR algorithms generated by Sparsity to test SMVM on the following types of matrices:

(1) FEM-like sparse matrices stored in BSR, using their natural block dimensions (to test potential SMVM performance);
(2) More irregular sparse matrices, with no natural block dimensions, stored in CSR (as a lower bound);
(3) Dense matrices, stored in either CSR or BSR (with varying block sizes) sparse format (to investigate the effects of fill).

Square $n \times n$ BSR matrices with NNZ nonzero entries were generated. For the sparse matrices, the parameters n and NNZ were determined using the procedure in Section 3.1, which means that the fill was not the same for each processor. Data set sizes and fills for the various platforms tested are shown in Table 4. Unfortunately, we lacked the time to develop a heuristic for sizing the dense matrix in sparse format, so we chose $n = 4096$ so that the matrix would not fit in the caches of any of the processors.

All block dimensions (r, c) such that r and c divide 6 were tested, because in the work of Im and Vuduc, most of the matrices drawn from applications had block dimensions no greater than 6 [15, 25]. The block entries in the matrix were (uniformly) randomly distributed, with random nonzero entries between -1 and 1. The number of nonzero entries did not vary for different block sizes, but the number of blocks varied. Empirical evidence by colleagues had shown that an additional SMVM test run to "warm up" the cache has little effect on performance, so our code performed no particular "cache cleansing" operations. For each machine and each pair of block dimensions (r, c) tested, 10 runs were taken. As with the STREAM benchmark, the minimum time of the test runs was used to calculate performance metrics dependent upon the inverse of time. For sparse and dense SMVM, we noted both the best performance among all the block sizes, and the 1×1 performance.

3.1. Selection of data set size parameters.

Two parameters govern the SMVM benchmark: the length n of the source vector, and the number NNZ of nonzeros in the matrix. Altering n and NNZ influences the level at which the data resides in the memory hierarchy; data set size affects capacity misses, and fill affects spatial and temporal locality. This coupling, and the additional complexity of the sparse matrix data structure, makes the choice of data set size more difficult than with memory benchmarks that work with dense arrays. SMVM's goal is to test memory bandwidth of accesses to the matrix itself, and not to the vectors. Thus, the vectors are sized to fit within the largest cache, whereas the number of nonzeros is chosen so that the matrix cannot reside in cache. The largest cache is used because typical applications often prefer the source vector to be as large as possible.

Data locality in the SMVM operation depends in a more complex way upon the fill as well as the source vector length. Locality is possible only for the source vector. Long stretches of zero blocks in a block row of the matrix cause skips in accesses to the source vector, so decreasing fill adversely affects locality. Our results will show that for the machines tested, fill has minimal effect on SMVM performance.

Restricting data lengths in order to avoid page faults is more difficult, since this depends heavily on the operating system and the number and types of running processes. Since Vuduc's lower and upper bounds for SMVM performance need only model cache properties for sufficient predictive power, we decided to focus on sizing the data to achieve the desired cache properties. In any case, we always used several test runs, at times when the system was lightly loaded.

3.2. Machine parameters.

k: The number of levels of caches in the memory hierarchy of the processor under investigation.
L_k, L_{k-1}: The number of bytes in the deepest (Lk) cache, and the next-deepest (L(k-1)) cache,

respectively. For example, if there are three levels of cache, $k = 3$ and L_3 is the size in bytes of the L3 cache. We assume Lk is the largest cache, as well as the deepest in the hierarchy.

D: **sizeof(double)** = 8, the number of bytes required for an IEEE 754 double precision value.

I: **sizeof(int)**, the number of bytes required for an integer value on a given platform. On the IA-32 architecture, for example, $I = 4$.

U: A hypothetical upper bound on the number of bytes that can be allocated for a process, without forcing paging out to disk. In reality this depends upon the operating system, what other processes are running, and real-time events, so a conservative (low) guess for U is used and verified by testing.

3.3. **Memory usage model.** We assume the $m \times n$ matrix has $r \times c$ blocks, aligned in block rows and columns (so that the index (i, j) of the upper left corner of each block satisfies $r|i$ and $c|j$ with Fortran-style indexing). For convenience we assume $r|m$ and $c|n$.

The row_start [] array consumes $I * n/r$ bytes. The values [] array requires $D * NNZ$ bytes, and col_idx [] takes $I(NNZ/c + 1)$ bytes. Neglecting storage for size constants, the memory usage of a BSR matrix is:

$$\frac{I * n}{r} + (D * NNZ) + I(\frac{NNZ}{c} + 1) \qquad (1)$$

3.4. **Source vector length.** Ideally, the source vector should be as large as one-fourth the deepest cache, assuming that the next-higher cache is smaller than this. This forces the source vector into the Lk cache, while avoiding conflicts with the sparse matrix data structures. A source vector this big might cause page faults, however, especially if many other processes are running at the same time. Setting the source vector as large as L_{k-1} should cause enough capacity conflicts in this level to force use of the Lk cache. However, if the system uses a victim cache [16], or some other technique for avoiding conflict misses at the $L(k-1)$ level, this may not suffice. We thus chose n as follows:

$$n = 2\frac{L_{k-1}}{D}. \qquad (2)$$

3.5. **Number of nonzeros.** The amount of memory consumed by the BSR data structure for 1×1 blocks is:

$$NNZ * D + NNZ * I + n * I \leq U \qquad (3)$$

The number NNZ of nonzeros in the matrix should exceed the size of the largest cache level:

$$NNZ \geq \frac{L_k}{D}. \qquad (4)$$

Platform	n	NNZ	Fill
AMD Athlon	16384	2790549	1.034×10^{-2}
Itanium 2	68036	38443539	8.305×10^{-3}
Pentium 3	16384	2790735	1.040×10^{-2}
Pentium M	8192	2789911	4.157×10^{-1}
Pentium 4	68036	3559737	7.690×10^{-4}
PowerPC 750	8192	9797318	1.460×10^{-1}
UltraSPARC II	4096	2795807	1.666×10^{-1}
UltraSPARC III	16384	2795345	1.041×10^{-2}

TABLE 4. Data set sizes and fill (NNZ/n^2) for the platforms tested. NNZ varies slightly (within a few hundred) each time, due to an artifact of the random block scattering method. Itanium 2 apparent cache size had to be modified slightly to ensure NNZ would not be too large.

Platform	L1 size	L2 size	L3 size
AMD Athlon	64 KB	256 KB	
Itanium 2	32 KB	256 KB	6 MB
Pentium 3	64 KB	256 KB	
Pentium M	32 KB	1 MB	
Pentium 4	8 KB	512 KB	1 MB
PowerPC 750	32 KB	1 MB	
UltraSPARC II	16 KB	2 MB	
UltraSPARC III	64 KB	8 MB	

TABLE 5. Cache sizes for the different platforms tested. If cache is split, then data cache size only is shown. Not all processors have an L3. For the Pentium 4, since the L2 size is close to the L3 size, we had to use half the actual L2 size to determine NNZ. Otherwise paging to and from disk could have become a major factor.

In addition, NNZ should not be large enough to cause page faults. We set $U = 2^{25}$ bytes = 32 MB, changing this to 64 MB if necessary in order to satisfy the above bounds. Including all of the above results, but leaving n intact as a parameter, we obtain the bounds on NNZ in Equation (5), and choose the maximal NNZ. The resulting n and NNZ values for each of the platforms tested are shown in Table 4, and the cache sizes which resulted in those values are in Table 5.

$$\frac{L_k}{D} \leq NNZ \leq \min\left(\frac{U - nI}{D + I}, n^2\right) \qquad (5)$$

4. Results

The "best performance" metric is shown in Figures 2 and 3. In those images, STREAM Triad data (recalculated in terms of Mflops per second, rather than

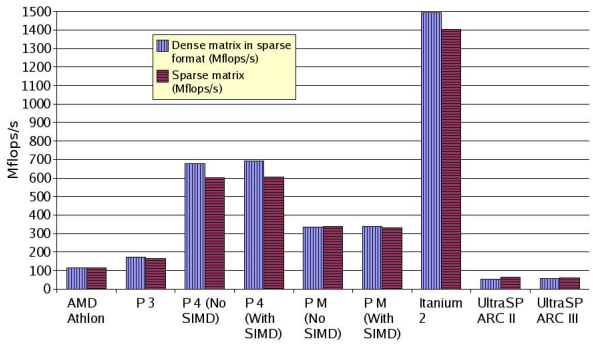

FIGURE 1. SMVM benchmark (Mflops/s) for various platforms, tested with a sparse matrix (see Section 3.1), and with a dense matrix in BSR format.

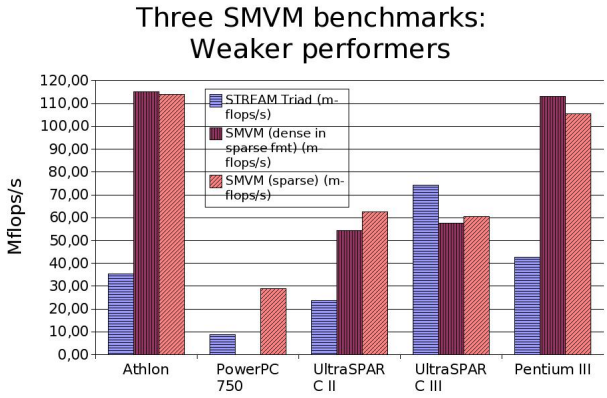

FIGURE 2. Results of running STREAM, SMVM with a dense matrix, and SMVM with a sparse matrix. Machines with weaker performance are shown (see Figure 3). Dense SMVM results were not available for the PowerPC 750.

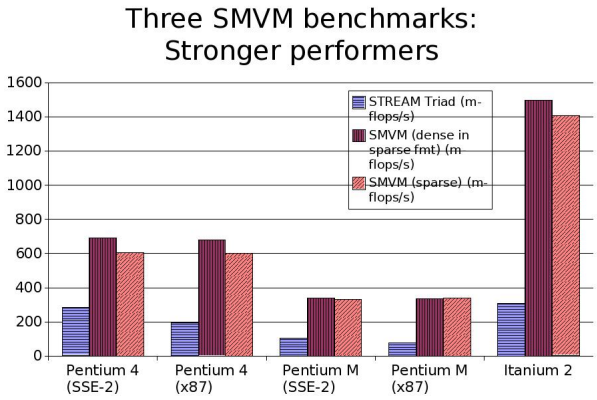

FIGURE 3. Results of running STREAM, SMVM with a dense matrix, and SMVM with a sparse matrix. Machines with stronger performance are shown (see Figure 2).

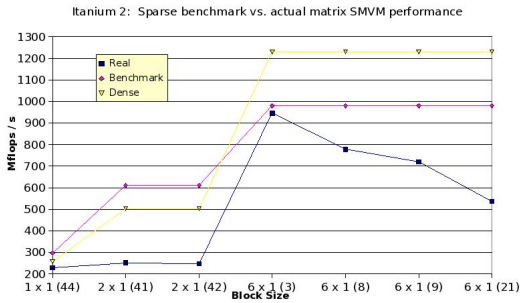

FIGURE 4. Itanium 2 data. Matrices 3, 8 and 9 come from FEM problems, and 21 from a fluid mechanics problem. The remaining three matrices are from linear programming. Block dimensions shown for each matrix are those which the SPARSITY system selected as optimal for that matrix.

MB per second) is shown for comparison. More important now, however, is to show the verisimilitude of our benchmarking strategy.

4.1. **Fill effects.** Processor dependent data set size selection affects locality of accesses to the source vector. We found, however, that SMVM benchmark results using optimal block sizes, for matrices with widely varying fill, are comparable. We compared the sparse benchmark results against running SMVM with a dense matrix in the same sparse format (taking the best performance among all the block dimensions tested). Figure 1 demonstrates that only platforms with unusally high-performance memory subsystems – the Itanium 2 and Pentium 4 – show significant difference between the two metrics, and the difference is less than fifteen percent.

4.2. **Comparison with matrices from applications.** Next is to show that our benchmark successfully predicts SMVM performance for matrices found in practice. We examined results for several matrices from Vuduc's work, including his SuperComputing 2002 paper as well as from a draft of his yet-unpublished thesis [25]. Two platforms were represented: the Itanium 2 and the UltraSPARC III. Itanium 2 results are clear: FEM matrices and other types of matrices with large block structures correlate well with our sparse benchmark. See 4. Furthermore, the dropoff in performance scales linearly with the fill ratio (ratio of stored values to actual nonzeros in the matrix – a measure of the space wasted by the block format). This is seen in Figure 5. For the UltraSPARC III, performance with the actual matrices was overall closer to our benchmark's results, and much closer for those matrices with larger natural block sizes, as one can see in Figure 6.

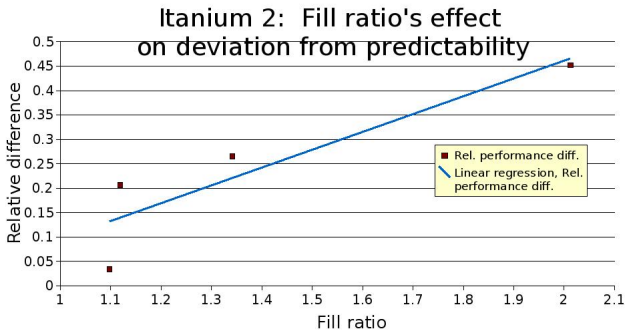

FIGURE 5. Itanium 2 results. Data points are for matrices 3, 8, 9 and 21, respectively (reading left to right on the graph). "Relative difference" is absolute value of the Mflops/s difference between SMVM on the actual matrix and with our synthetic (sparse) benchmark, scaled by the results of our benchmark.

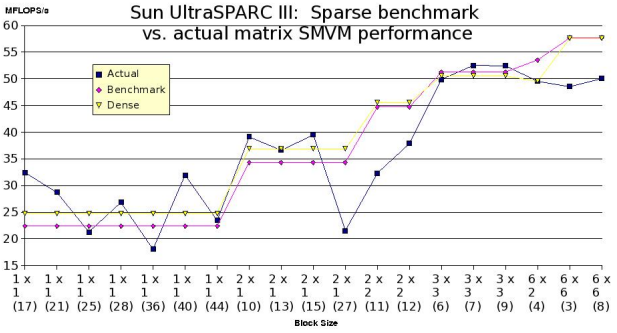

FIGURE 6. Sun UltraSPARC III: sparse SMVM benchmark results, compared with those for executing SMVM on actual matrices. Matrices 3–9 come from FEM problems.

4.3. **Peak memory bandwidth and lack of correlation with STREAM.** We also found that the STREAM benchmark results did not correlate well with our SMVM benchmark, using the optimized algorithm. For example, the large SMVM performance difference between Pentium 4 and Itanium 2 seems more related to peak memory bandwidth than to STREAM, as Figure 7 suggests. It appears that register blocking conceals most of the load and store latency as well as the slower peak floating-point rate of the Itanium (when the Pentium's SIMD floating-point operations are considered), so that memory bandwidth contributes more significantly than latency to performance. Since the Itanium 2 has a peak bandwidth of 6.4 GB/s and the Pentium 4 only half that, the Itanium achieves twice the performance for optimized SMVM.

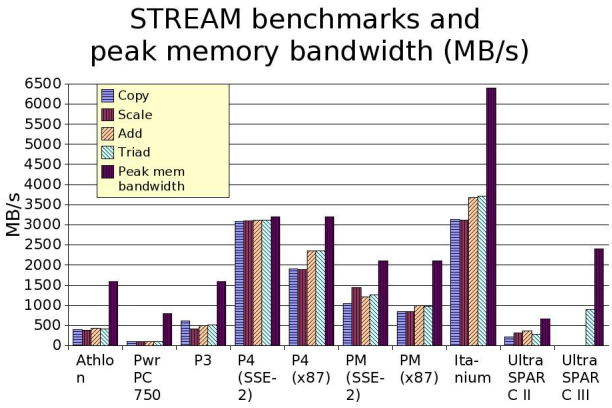

FIGURE 7. STREAM benchmark results in MB/s, along with peak memory bandwidths (maximum data flow rates between memory and cache) in MB/s.

4.4. **Unoptimized SMVM performance.** In contrast, the unoptimized (1×1) SMVM algorithm cannot fully exploit high peak memory bandwidth. The correlation of STREAM results with 1×1 SMVM performance confirms this. In particular, STREAM predicts the relationship between Pentium 4 and Itanium 2 SMVM performance. Figure 8 compares the sparse 1×1 Mflops/s rate with that of the dense matrix in sparse format. Although the Itanium outperformed the Pentium 4 for the sparse case, this is reversed for the dense case. Due to the Itanium's cache sizes and configurations, it was the only processor tested for which the dense matrix in sparse format had more elements (16777216, as opposed to 38443539 for the sparse matrix – see Table 4). When we tested a sparse SMVM on the Itanium with the same n as in Table 4, but $NNZ = 16777216$, the Mflops/s rate dropped to 76.88, which is slightly poorer than the Pentium 4. It corresponds to the closeness of the Itanium 2 and Pentium 4 results for the STREAM benchmarks (see Figure 7). The 1×1 case is truly a test of "streaming from memory." The IA-32 assembly code unrolls the 1×1 SMVM inner loop the same number of times as it unrolls the STREAM loops, so the compiler must recognize a similarity in scheduling. (Although the SMVM loops were not successfully vectorized in our study, both SMVM and STREAM use the SSE2 ISA and registers.) Furthermore, the two notebook computers (the Pentium M and the PowerPC) did most poorly, and notebooks typically have a lower-performance, lower-cost memory system.

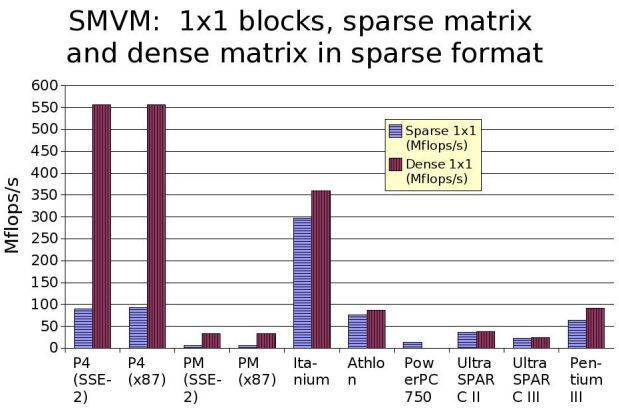

FIGURE 8. SMVM benchmark results for the unoptimized 1×1 case. No dense results available for the PowerPC 750.

5. Application: Evaluating SIMD floating-point units

Determining how fast a particular machine can execute sparse matrix-vector multiplication is useful in itself. However, an SMVM benchmark can also help evaluate the effectiveness of specific processor and memory subsystem features. The SIMD floating-point and load instructions present in many recent processors for desktops and workstations – specifically, the SSE2 instruction set for parallel operations on dual double-precision values – provide a readily available test case.

Our results are as yet inconclusive. The compiler tested (Intel C/C++ version 7.1) refused to issue SIMD instructions automatically for SMVM, even when preprocessor directives were used to reassure it that the arrays satisfied the necessary alignment constraints. Only for the much simpler STREAM loops would it generate the SIMD memory and floating-point instructions that provided STREAM with a significant performance boost. We attempted (unsuccessfully) to replace the "hand-written" block matrix-vector multiplications in the Sparsity code with inline assembly that explicitly invoke the SIMD instructions. The attempt demonstrates the difficulty of aligning data structures for effective use of SIMD hardware, which suggests a reason why Intel decided not to support the double-precision SIMD instructions in the Itanium processor.

5.1. **The whys and whats of desktop SIMD.** Multimedia and signal processing applications often perform operations on streams of k-tuples of small data, where k is fixed and small: coordinate pairs, for instance. This has stirred many processor manufacturers to include k-lane vector units in their desktop and workstation models, that operate on a single larger register (usually 64 or 128 bits) divided into several shorter data values. This began with instruction sets for short integers (such as Alpha MAX, SPARC VIS and Intel Pentium MMX) [9], and evolved into dual floating-point functional units for graphics applications [13, pp. 109–110].

Examples of processors with SIMD floating-point units grace many desktops and higher-end embedded systems (as well as graphics cards). The Streaming SIMD Extensions (SSE) to the IA-32 ISA, for example, specify parallel floating-point operations on two 32-bit single-precision values, packed into a single 64-bit floating-point register [5]. The MIPS-3D extensions to the MIPS 64 ISA perform a similar task [24], as do the AltiVec instructions in IBM's PowerPC 970 (except with four 32-bit values instead of two) [7]. Intel's Pentium III and AMD's Athlon both implement SSE.

The second version of SSE, SSE2, extends SSE to parallel operations on two 64-bit double-precision values. Intel's Pentium 4 and AMD's Opteron both implement these instructions [1, 14]. Intel carefully points out that only specialized scientific and medical visualizations need the additional precision [4], which suggests the difficulty of using SSE2 effectively in a general application. Nevertheless, the ATLAS project for automatic optimization of the BLAS (and certain LAPACK routines) takes advantage of both SSE and SSE2 [26], as does FFTW version 3.0 [12]. There are also libraries available for small matrix-vector manipulations using SSE and SSE2 (e.g. [8, 18]).

Intel's Itanium 2 processor, released in 2003, only supports the original SSE instruction set, although its two fused multiply and add (FMAC) functional units offer potentially twice the bandwidth of SSE2 [6]. Furthermore, the Itanium 2 supports twice as many loads per cycle (four) as FMAC operations (two), unlike (for example) the Pentium 4's NetBurst microarchitecture, which can only execute a single SIMD load in parallel with a SIMD floating-point operation. We had hypothesized that for more general applications such as sparse matrix-vector multiplication, the SSE2 instructions offer minimal speedup, due to their inflexibility.

5.2. **SIMD and register blocks.** Each register block operation in SMVM is a small, dense matrix-vector multiplication, and applying SIMD instructions to it is a vectorization of the operation. The special instructions required for successful vectorization depend upon the orientation of the blocks. Column-major order calls for scalar-vector multiplication; row-major order requires a reduce (dot product) operation, or the ability to sum the elements of a vector. Column-major is preferable, since in that case vectorization does not

affect associativity. Most deskop SIMD ISAs, including SSE2, only allow SIMD loads for contiguous values. This means that the ordering is fixed (a non-unit stride load could circumvent it). SSE2 lacks a scalar-vector multiplication instruction, but a "shuffle" instruction enables reduce operations. In addition, SPARSITY uses row-major order. Thus, we stayed with row-major order.

In the SSE2 ISA, there are eight 128-bit XMM registers. Assembler mnemonics ending in "PD" (packed double) operate on two 64-bit double-precision values. The following sums the two elements of `%xmm1`, leaving the result in the high double-word of `%xmm1`.

```
SHUFPD    $0x0, %xmm1, %xmm0
ADDPD     %xmm1, %xmm0
```

This code fragment can be used to vectorize each dot product in the row-oriented block matrix-vector multiply. The first listing below shows the C source of the routine for matrix-vector multiply of a 2×2 block, and the listing below it shows the corresponding IA-32 SSE2 assembly for the inner loop. (The `%xmm0` register holds `d0` in its lower double-word and `d1` in its upper double-word.)

```
for (i=start_row; i<=end_row; i++)
{
  register double d0, d1;
  d0 = dest[2*i+0];
  d1 = dest[2*i+1];
  for (j=row_start[i]; j<row_start[i+1]; j++)
  {
    d0 += value[j*4+0] * src[col_idx[j]+0];
    d1 += value[j*4+2] * src[col_idx[j]+0];
    d0 += value[j*4+1] * src[col_idx[j]+1];
    d1 += value[j*4+3] * src[col_idx[j]+1];
  }
  dest[2*i+0] = d0;
  dest[2*i+1] = d1;
}
```

```
movapd    (%eax,%edi), %xmm1
movapd    (%ecx,%edx), %xmm2
movapd    8(%ecx,%edx), %xmm3
mulpd     %xmm3, %xmm1
mulpd     %xmm3, %xmm2
shufpd    $0x0,%xmm1,%xmm4
shufpd    $0x0,%xmm2,%xmm5
addpd     %xmm4, %xmm1
addpd     %xmm5, %xmm2
shufpd    $0x3,%xmm2,%xmm1
addpd     %xmm2, %xmm0
addl      $32, %ecx
addl      $1, %esi
cmpl      %ebp, %esi
jl        ..B14.5
```

5.3. **Compiler vectorization issues.** When we actually compiled the SMVM code for the SSE2 ISA, we found that the compiler issued SSE2 instructions, but treated the XMM registers as scalar registers, holding only 64-bit values. The performance difference between the x87 and SSE2 implementations is therefore miniscule (see Figure 9). Floating-point latency for SSE2 add and multiply is one cycle less than for x87 floating-point instructions, and the x87 and SSE2 floating-point instructions use the same functional unit in the Pentium 4's NetBurst microarchitecture, so it makes sense to use SSE2 arithmetic operations if possible [14]. However, the SIMD load instruction, `MOVAPD` (MOVe Aligned Packed Double), requires the two memory locations to be both adjacent and aligned on 16-byte boundaries. The instruction for loading two unaligned values (`MOVUPD`) has a longer latency, long enough in this case that it is preferable to use scalar arithmetic. Since the compiler could not guarantee that the input arrays were aligned on 16-byte boundaries, it conservatively assumed they were not aligned. The preprocessor directives suggested by the Intel C++ Compiler documentation, including `#pragma vector aligned`, did not alleviate the problem. Manually ensuring data alignment (by forcing array pointers to start at values divisible by 16) and explicitly coding an SMVM routine in IA-32 assembly caused a run-time error. Further work is needed.

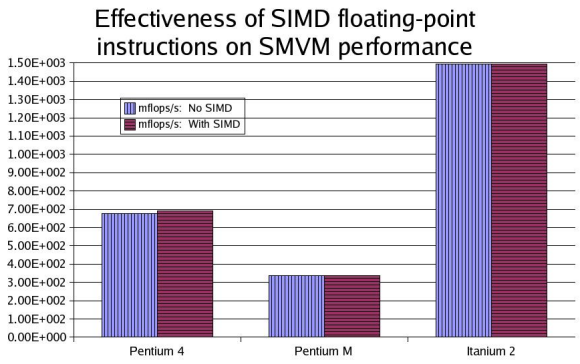

FIGURE 9. Itanium 2 performance data provided for comparison. The processor has no SIMD support for double-precision values, so the "SIMD" and "no SIMD" values for the Itanium 2 are the same.

5.4. **SMVM and STREAM with SIMD.** That SIMD floating-point instructions have a limited benefit is suggested by SPECfp2000 statistics cited in the *Intel Technology Journal* when the Pentium 4 was first released. The new SSE/SSE 2 instructions provided only a five percent speedup over using the x87 instructions alone for floating point calculations [14]. The authors argue

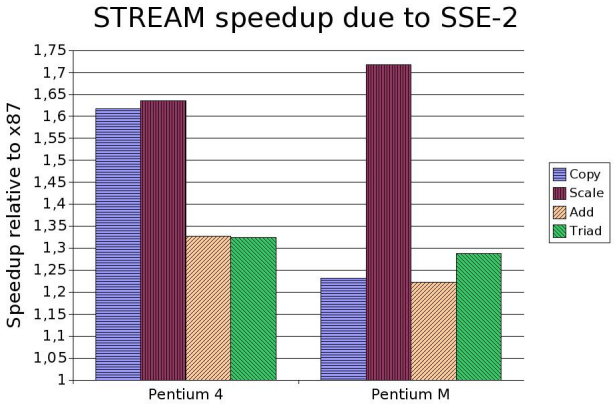

FIGURE 10. For the Pentium 4 and M processors, SSE2-induced speedups in the STREAM benchmarks.

that in 2001 (the date of publication of the article), compilers and libraries for the new SIMD hardware were still immature.

One could say that SMVM, and also the floating-point SPEC 2000 benchmarks, represent more "complex" tasks, involving more irregular and indirect memory accesses. Therefore, it would be useful to explore the benefits of SIMD instructions for a set of simplified computational loops. The STREAM Scale, Add and Triad benchmarks (See Section 2.2 above) are ideal test cases.

The STREAM benchmarks do show measurable speedup when the SSE2 instruction set is used, as Figure 10 shows, and examination of the assembly code shows that they use SSE2 floating-point operations and the aligned SIMD load and store instructions. However, of the three STREAM kernels involving floating-point arithmetic, only Scale has a nearly ideal (twofold) speedup. It seems that a greater rate of consumption of memory operands (two per iteration for Add and Triad, as opposed to one per iteration for Scale) limits potential speedup by challenging memory bandwidth more (considering that STREAM tests practical limitations on bandwidth).

6. Conclusions

Our sparse matrix-vector multiplication benchmark exposes the potential SMVM performance of the processors tested, by using an SMVM algorithm tuned for each machine. The potential performance is realized by certain classes of matrices. Our benchmark's strategy of sizing the source vector length and number of nonzeros in the matrix according to each machine's cache characteristics allows fair comparison of SMVM speed across different processors. We found that optimized SMVM performance correlates with peak memory bandwidth, whereas unoptimized (1×1) SMVM matches the "sustained memory bandwidth" as measured by the STREAM benchmark. From an architectural perspective, this suggests that platforms tuned for dense BLAS2 and BLAS3-type operations can profit best from optimized algorithms, even if their performance using unoptimized algorithms lags behind.

The Itanium 2 has a unique streaming floating-point technology: It has a high peak memory bandwidth, can complete four floating-point loads and two stores in a cycle, and can complete potentially two DAXPY operations in a cycle. It is optimized for streaming in long arrays by restricting floating-point operands to the L2 or L3 caches and sending floating-point loads directly to the L2 cache. Despite its much slower clock rate than the comparable Pentium 4, and the Itanium's lackluster performance on unoptimized SMVM, its advantage over the Pentium 4 for optimized SMVM actually exposes its greater peak memory bandwidth. Given the recent success of automatic optimization of mathematical libraries such as the BLAS, LAPACK and the Fast Fourier Transform, that try as much as possible to replicate dense matrix operations, processor manufacturers can afford to neglect other factors and optimize this common case of dense matrix-vector and matrix-matrix multiplication.

7. Platforms tested and compiler optimization flags

7.1. **AMD Athlon.** This is an 1145 MHz single processor with 515184 KB main memory (peak memory throughput 1.6 GB/s), running GNU/Linux kernel 2.6.0. Benchmarks were compiled using `gcc`, version 3.3.2. (We unfortunately lacked access to The Portland Group's `pgcc` C compiler, which is specifically targeted for AMD architectures.) Base compiler optimization flags:

```
-O3 -fomit-frame-pointer -march=athlon
-fstrict-aliasing -malign-double
```

Additional flags `-funroll-loops` and `-fprefetch-loop-arrays` were varied, but were found to have little effect on SMVM performance.

7.2. **IBM PowerPC 750.** This is a 300 MHz PowerPC 740/750 with 160 MB main memory (probably 800 MB/s peak memory bandwidth) in an Apple G3 notebook, running GNU/Linux kernel 2.4.21. Benchmarks were compiled using `gcc`, version 3.2.3. Base compiler optimization flags:

```
-O3 -mpowerpc -mcpu=750
-fomit-frame-pointer -fstrict-aliasing
```

Additional flags `-funroll-loops` and `-fprefetch-loop-arrays` were varied, and also found to have little effect on overal SMVM performance.

7.3. **Intel Pentium 3.** This Pentium 3 ("Coppermine" 0.18 μ chipset) is a dual-processor 1000 MHz with 2 GB RAM and 1.6 GB/s peak memory throughput. It is part of an eight-node cluster running GNU/Linux 2.4.18-27.7.xsmp (RedHat 7.3). Benchmarks were compiled with the Intel C/C++ Compiler version 7.1, with the following optimization flags:

```
-O3 -xK -tpp6 -unroll
```

7.4. **Intel Pentium 4, M.** The Pentium 4 platform is a Dell workstation with two 2.4 GHz Pentium 4 processors and 512 KB main memory, running GNU/Linux 2.4.20-8smp (RedHat 9.0). Peak memory bandwidth is 3.2 GB/s. The Pentium M machine is an IBM ThinkPad R40 with 256 KB RAM, running GNU/Linux 2.4.22 (RedHat 9.0), with peak memory bandwidth 2.1 GB/s (DDR RAM on a 133 MHz bus). Both were tested using the Intel C/C++ Compiler, version 7.1, with the same base optimization flags, which were as follows:

```
-O3 -tpp7 -unroll -restrict -fno-alias
```

Both processors have SSE2 SIMD instructions. This feature is enabled with the `-xW` flag, whose presence was varied, as mentioned above.

7.5. **Intel Itanium 2.** This is a 900 MHz (revision 7) dual processor system with 4 GB RAM (peak bandwidth 6.4 GB/s), and is one node of a cluster of similar machines. Benchmarks were compiled using the Intel C/C++ Compiler, version 7.1. Base compiler optimization flags were as follows:

```
-O3 -fno-alias -fno-fnalias -IPF_fma
```

The `-IPF_fma` flag activates the dual fused multiply-add units, which both speeds up and increases the accuracy of the multiply-add operation that is the standard operation in matrix-vector multiplication.

7.6. **Sun UltraSPARC II.** This machine is a four-processor 450 MHz E420 running Solaris 8, serving SunRay thin clients, with a peak memory bandwidth of 664 MB/s. The benchmark was compiled for the UltraSPARC II using Sun WorkShop 6 (update 1 C 5.2 2000/09/11). The following base optimization flags were used:[2]

```
-xtarget=ultra2 -xO5 -xrestrict=%all
-xprefetch=auto
```

[2]The `-xO5` flag is comparable to the `-O3` flag in `gcc` and Intel's compiler, and indicates the highest level of optimization.

7.7. **Sun UltraSPARC III.** This machine is a 1.2 GHz SunFire 280R workstation running Solaris. The UltraSPARC III can issue two floating-point operations and one load or store per cycle. The benchmark was compiled for the UltraSPARC III using Sun WorkShop 6. The following base optimization flags were used:

```
-xtarget=ultra3plus -xO5 -xrestrict=\%all
-xarch=v9b -xprefetch=auto
```

8. Acknowledgments

Sincerest thanks to Professors James Demmel, Katherine Yelick and John Kubiatowicz for advising our research. Members of the Berkeley Benchmarking and Optimization Group (BeBOP) provided invaluable assistance, especially Rajesh Nishtala and Rich Vuduc. We also owe thanks to Tyler Berry (`tyler at arete dot cc`, `http://www.arete.cc/`) for web page hosting and access to his AMD Athlon server and IBM PowerPC 750 desktop. Felipe Gasper also offered the use of several of his desktop and server machines.

Experiments on the Sun Ultra3-based platform were performed at the High Performance Computing Research Facility, Mathematics and Computer Science Division, Argonne National Laboratory. This research was supported in part by the Department of Energy under DOE Grant No. DE-FC02-01ER25478, and a gift from Intel.

References

1. Advanced Micro Devices, Inc., *AMD Opteron^{TM} processor data sheet, publication 23932, rev. 3.06*, November 2003.
2. Basic Linear Algebra Subprograms Technical (BLAST) Forum, *Document for the Basic Linear Algebra Subprograms (BLAS) standard*, August 1997, Draft.
3. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, *A portable programming interface for performance evaluation on modern processors*, The International Journal of High Performance Computing Applications **14** (2000), no. 3, 189–204.
4. Intel Corporation, *SSE2: Perform a double-precision 3D transform*, Tech. report, 2001.
5. Intel Corporation, *IA-32 Intel® architecture optimization reference manual*, 2003.
6. Intel Corporation, *Intel Itanium 2 processor reference manual for software development and optimization*, April 2003.
7. International Business Machines Corporation, *PowerPC microprocessor family: AltiVec^{TM} technology programming environments manual, version 2.0*, July 2003.
8. Zvi Devir, *Optimized matrix library for use with the Intel Pentium 4 processor's Streaming SIMD Extensions (SSE2)*, Tech. report.
9. Henry G. Dietz, *SWAR: SIMD within a register*, March 1997.
10. J. Dongarra, V. Eijkhout, and H. van der Horst, *SparseBench: A sparse iterative benchmark, version 0.9.7*, Tech. report, 2000.

11. J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft. **16** (1990), 1–17.
12. M. Frigo and S. G. Johnson, *Fftw: An adaptive software architecture for the fft*, vol. 3, 1998, ICASSP conference proceedings, pp. 1381–4.
13. John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach*, third ed., Morgan Kaufmann Publishers, San Francisco, 2003.
14. G. Hinton, D. Sager, M. Upton, D. Boggs, D. M. Carmean, A. Kyker, and P. Roussel, *The microarchitecture of the Pentium® 4 processor*, Intel Technology Journal (Q1, 2001).
15. Eun-Jin Im, *Optimizing the performance of sparse matrix-vector multiply*, Ph.D. thesis, University of California, Berkeley, May 2000.
16. Norman P. Jouppi, *Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers*, 25 Years ISCA: Retrospectives and Reprints, 1998, pp. 388–397.
17. J. D. McCalpin, *Sustainable memory bandwidth in current high performance computers*, Tech. report, 1995.
18. Iain Nicholson, *libSIMD: The single instruction multiple data library*, Tech. report, September 2003.
19. R. Pozo and B. Miller, *Scimark 2.0*, Tech. report, NIST, 2000.
20. Y. Saad, *Iterative Methods for Sparse Linear Systems*, second ed., 2000.
21. Rafael H. Saavedra and Alan Jay Smith, *Measuring cache and TLB performance and their effect on benchmark runtimes*, IEEE Transactions on Computers **44** (1995), no. 10, 1223–1235.
22. R. H. Saavedra-Barrera, *CPU performance evaluation and execution time prediction using narrow spectrum benchmarking*, Ph.D. thesis, May 1992.
23. Allan Snavely, Nicole Wolter, and Laura Carrington, *Modeling application performance by convolving machine signatures with application profiles*, December 2001, IEEE 4th Annual Workshop on Workload Characterization.
24. MIPS Technologies, *MIPS-3D^{TM} ASE*, 2003.
25. Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee, *Performance optimizations and bounds for sparse matrix-vector multiply*, SuperComputing 2002, November 2002.
26. R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimization of software and the ATLAS project*, Parallel Computing **27** (2001), no. 1–2, 3–25.

University of California, Berkeley
E-mail address: {mhoemmen,hormozd}@cs.berkeley.edu