Title

Memory-intensive benchmarks: IRAM vs. cache-based machines

Authors
Gaeke, Brian G.
Husbands, Parry
Kim, Hyun Jin
et al.

Publication Date
2001-09-29

Memory-Intensive Benchmarks: IRAM vs. Cache-Based Machines?

Brian R. Gaeke[1], Parry Husbands[2], Hyun Jin Kim[1], Xiaoye S. Li[2], Hyun Jin Moon[1]
Leonid Oliker[2], Katherine A. Yelick[1], Rupak Biswas[3]

[1]Computer Science Division, University of California Berkeley, CA 94720
[2]NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720
[3]NAS Division, NASA Ames Research Center, Moffett Field, CA 94035

Abstract: The increasing gap between processor and memory performance has lead to new architectural models for memory-intensive applications. In this paper, we explore the performance of a set of memory-intensive benchmarks and use them to compare the performance of conventional cache-based microprocessors to a mixed logic and DRAM processor called VIRAM. The benchmarks are based on problem statements, rather than specific implementations, and in each case we explore the fundamental hardware requirements of the problem, as well as alternative algorithms and data structures that can help expose fine-grained parallelism or simplify memory access patterns. The benchmarks are characterized by their memory access patterns, their basic control structures, and the ratio of computation to memory operation.

1. Introduction

Many high performance applications run well below the peak arithmetic performance of the underlying machine, and the inefficiency is often attributed in part to inadequate memory systems on individual processors. This problem worsens every year as processor performance improves by about 60% each year while DRAM latency improves by only 7%. In conventional processors, this problem is addressed by increasing the size of on-chip SRAM caches and by exploiting dynamically discovered instruction level parallelism to mask off-chip latencies. An alternative is to place denser DRAM memory directly on the processor die, usually called Processor-in-Memory (PIM) or Intelligent RAM (IRAM) designs. The VIRAM processor under development at U.C. Berkeley is one example of such a system—it combines mixed logic and DRAM with a vector instruction set, allowing the compiler to explicitly express fine-grained data parallelism [Koz99,KJG+01]. Other examples of processor-in-memory designs include the DIVA project [HKK+99], the HTMT project [Sterling] and the Mitsubishi M32R processor [M32R].

? This work was supported in part by the Laboratory Directed Research and Development Program of the Lawrence Berkeley National Laboratory, which is supported by the U.S. Department of Energy under contract number DE-AC03-76SF00098, by the Advanced Research Projects Agency of the Department of Defense under contract DABT63-96-C-0056, and by the California State MICRO Program. The VIRAM chip and compiler effort are being supported by donations from IBM, Cray, and MIPS.

The goal of our project was to understand the performance of memory-intensive applications by studying some of the algorithmic kernels in details. In this paper we looked at various characteristics of the applications, such as memory access patterns, ratio of computation to memory operations, and degree of control irregularity to help understand the performance behavior across machines. In particular, we look at the question of whether the performance is actually limited by memory bandwidth or by some other feature of the architecture. We treat each benchmark as a black-box function and explore several alternative algorithms to improve performance, focusing mainly in the vector processing features of the VIRAM architecture.

2. VIRAM Architecture

The VIRAM architecture extends the MIPS instruction set with vector instructions that include integer and floating point operations, as well as memory operations for sequential, strided, and indexed (scatter/gather) access patterns. The processor has 32 vector registers, each containing up to 32 64-bit values. Logically, a vector operation specifies that the operation may be performed on all elements of a vector register in parallel. The current micro-architecture is divided into 4 64-bit lanes, so a single vector instruction is executed by the hardware 4 elements at a time. In contrast, SIMD extensions like Intel MMX or SSE have vector operations whose length matches that of the hardware resources. The advantages of specifying longer vector operation in the instruction set are: 1) it expresses parallelism between groups of elements that can be exploited to overlap memory operations; 2) fewer instructions may be needed to express the same algorithm, thereby lowering instruction bandwidth requirements; and 3) the instruction set need not be changed when the number of lanes changes across processor generations.

The hardware resources devoted to functional units and registers may be subdivided to operate on 8, 16, or 32-bit data. For example, a vector register holds 2K bits, which corresponds to 32 64-bit elements, 64 32-bit elements, or 128 16-bit elements. Similarly, when the data with is cut in half, the peak execution rate of arithmetic operations doubles. The data width (known as the *virtual processor width*) may be set by the application software and changed as different data types are used in the application. VIRAM is designed to run at 200 MHz, and there are two integer functional units and one floating point unit. The instruction set specification allows for 32 and 64-bit floating point as well as 8, 16, 32, and 64-bit integer operations, but the current VIRAM implementation will support only 32 bit floating point and 16, 32, and 64-bit integer operations. The ISA also contains fused multiply-add instructions, but they are not generated by the current compiler and will therefore be omitted from consideration. The resulting peak performance for VIRAM is 1.6 GFLOPS for 32-bit floating point, 3.2 GOPS for 32-bit integer operations, and 6.4 GOPS for 16-bit integer operations.

The VIRAM implementation includes a simple in-order MIPS processor with a cache and floating point unit, a DMA engine for off-chip access, a memory crossbar, and a vector unit, which is managed as a co-processor. The estimated transistor count is over 100 million, and the power estimate is 2 Watts. There are 12 MB of on-chip DRAM organized into 8 banks, and all of the memory is directly accessible from both scalar and vector instructions.

3. Benchmark Applications

Our benchmarks are chosen to stress the limits of a processor's memory system, but they represent the kernels of real applications of interest in large scale scientific computing. Most of them are taken from the DARPA Data Intensive Systems (DIS) stressmark suite [DIS00]. In general, data-intensive applications are characterized by large data sets, significant data movement between processors and memory, and relatively low arithmetic operation counts relative to data access. Many of the problems are further complicated by either irregular memory access patterns or irregular control structures. These characteristics often lead to performance scaling problems when run in parallel and to memory bottlenecks on single processors.

3.1 Transitive Closure

The first benchmark is an algorithm to compute the transitive closure of a graph, and is from the DIS stressmark suite. Rather than simply computing whether a path exists, the problem is to compute the length shortest path between all pairs of graph nodes, based on the Floyd-Warshall algorithm. An important feature of this problem is that the input and output are both expressed as an adjacency matrix, i.e., given a graph with n vertices, the input and output are both nxn matrices. Other algorithms that work on a sparse representation of the graph would have very different memory characteristics. The algorithm runs in O(n^3) time, and does 2*n operations per element. It is similar to a matrix-vector multiplication, except that each step involves an addition (adding the length of two know n paths) and a comparison (to choose the minimum of the sum and the previously computed path length). The original version of the code (taken from a DIS reference implementation) involved running through the rows of a column-major matrices, so the memory operations involved strided load. Through a simple loop transformation, we reorganized this to use a unit-stride load. This benchmark is memory-intensive, in the sense that each step performs only 2 arithmetic operations, while is does 3 loads and 1 store.

3.2 GUPS

The second benchmark is a simple, synthetic problem known as the GUPS benchmark, which measures Giga-updates-per-second. This benchmark repeatedly reads and updates distinct, pseudo-random memory locations. It reflects the kind of memory access pattern that might occur when inserting elements into a large hash table, or from sorting or building a histogram of a large data set. Our implementation is freely available for download [Gae01]. The updates are simple arithmetic/logical operations, in our case adding 1 to the result of the read. The memory locations are specified as indexed into a large array, and they are precomputed using a pseudo-random number generator based on an LFSR (linear feedback shift register) whose feedback term is chosen to provide a maximal-length period [Koo00]. In our study, we parameterized the runs over three variables: number of iterations of the update loop, the size of the array in which updates are performed, and the width of the data type (8-, 16-, 32-, or 64-bit integer).

The memory addresses are selected a priori, so there is no random number generation overhead in the inner loop. Furthermore, since the locations are chosen to be independent of one another, accesses can proceed in parallel. Each step of the GUPS algorithm performs only one addition, along with two memory reads, and one memory write. The write and one of the reads are indexed operations (scatter/gather), and the remaining read is unit stride. On a cache-based machine, the unit stride read will perform reasonably well, but the index read is essentially random, while the indexed write should hit in a cache. The very low arithmetic operation count and poor spatial locality will make this benchmark a challenge for any memory system.

3.3 Sparse Matrix-Vector Multiplication (SPMV)

Sparse matrix-vector multiplication is a very common kernel in scientific simulations, and like the GUPS problem, is has random memory access patterns and a low number of arithmetic operations, in this case floating point operations. The DIS stressmarks and the NAS Parallel benchmarks include a Conjugate Gradient (CG) problem, for which the running time is dominated by a SPMV. We have run a full implementation of CG on IRAM, but in this paper we discuss only the SPMV kernel, since it highlights the memory behavior of the system.

A matrix vector multiplication is to compute a vector y = A*x, given a matrix A and vector x. A sparse matrix contains mostly zero elements (a nonzero fraction below .001 is typical), so the matrix is stored in a "compressed" form by storing the nonzero values along with their indices. SPMV multiplies such a matrix times a dense vector (all elements are stored explicitly in x and y), and the vectors will be accesses using the indexes of the nonzero elements (not simply unit stride). Assuming A is nxn and has m nonzeros, the total number of floating-point operations for SPMV is 2*m. Each step of the algorithm requires one addition, one multiplication and at least three memory operations. The exact nature and count of the memory operations depends on the storage format and algorithm organization. Sparse storage schemes usually allocate contiguous memory only for the nonzero elements of the matrix, and sometimes for a limited number of zeros. Some auxiliary index arrays are used to map each nonzero into the full matrix. We briefly describe the different storage schemes used in our study, some of which are specifically designed for vector architectures.

CRS format: Compressed Row Storage (CRS) is one of the most general and widely used compact representations for sparse matrices. This format stores the nonzeros of the matrix rows in contiguous memory locations. Three vectors are used to represent a matrix: one with m floating-point entries (val), one with the m column indices of the nonzeros (colind), and a third with n+1 pointers to the beginning of each row in val and colind (rowptr).

The CRS format leads to a simple algorithm, in which a one takes a dot product of x with each row of A, and writes the result into y. The reads into the x vector are indexed, so this algorithm has 3 loads (2 unit stride on A and 1 indexed on x) and two arithmetic operations on each step. There is only one store per row, which is amortized over the entire dot product and so is not counted. The CRS has some disadvantages for the VIRAM-1 architecture, in addition to the indexed loads and stores on x, because the row length varies from row to row, and is usually not very long. A dot product computation requires a reduction operation, which perform poorly for short vectors, because once the reduced vector fits in a register, the vector length is halved on each step. Therefore, we studied some other formats and optimization techniques that can overcome these difficulties.

CRS with narrow band: In this format, we attempt to mimic the effect of bandwidth reduction orderings, such as reverse Cuthill-McKee (RCM) [CM69,Geo71]. The advantage is that accessing the vector x requires a much smaller stride. In our experiments, we modified the matrix generation algorithm to confine the non-zero entries to a prescribed bandwidth.

Ellpack format: The Ellpack (or Itpack) format forces all rows to have the same length as the longest row by padding them with zeros. Thus, after padding, we get a matrix of dimension nxk, where k is the length of the longest row (k is usually much less than n). SPMV can then be effectively vectorized by expressing it as a series of operations along the columns of the new matrix, which have long vectors of exactly the same length. The Ellpack format is efficient for matrices with the longest row close to the average row size. Otherwise, the efficiency can be arbitrarily bad for matrices with general sparsity patterns, because of wasted computation and storage for the padded zeros.

Segmented-sum: This representation was first proposed in [BHZ93] for the Cray PVP vector architecture. The data structure is an augmented form of the CRS format. The computational structure is similar to Ellpack, because we can vectorize along many rows at the same time. In short, a sparse matrix is represented as a segmented vector by treating each row as a segment. This segmented vector is conceptually laid out as a two-dimensional array (in column-major order) of size s*l ~= m. The parameter l is chosen to approximate the hardware vector length so that the SPMV operation can be expressed in terms of vectorizable row operations. The Segmented-sum is then performed bottom-up, with each ``row-sum'' of A*x stored at the beginning of each segment.

The advantages of this format are that it allows long vector lengths and that the parameter l can be adjusted depending on the size of the hardware vector registers. Furthermore, all the vectors are of the same length. However, the existing Cray code did not show much performance improvement, mainly because some data structures in the inner loop are accessed with a large stride. On Cray vector machines, performance is acceptable as long as the stride is not a power of two. Unfortunately, such big strides can slow data accesses on the VIRAM by as much as a factor of four. We therefore modified the Cray code so that the inner loop contains mostly unit stride data access.

3.4 Histogram

Computing a histogram of a set of integers is a class problem that can be used as a subroutine in sorting, but also occurs in an image processing problem in the DIS stressmark suite called the “Neighborhood” problem, which approximates the gray-level co-occurrence matrix (GLCM [Par97]) entropy and energy descriptors of an input image. These approximations are computed through sum and difference histograms of neighboring pixel values. Two important considerations govern the algorithmic choice for computing histograms—the number of buckets, b, in the resulting histogram, and the likelihood of duplicates in the data. For the neighborhood problem, the number of bits per pixel determines the number of buckets, and collisions in the histogram calculation are very common, because there are typically many occurrences of some colors (white) in an image.

The characteristics of the histogram problem are nearly identical to those of the GUPS benchmark: there is 1 arithmetic operation (an increment) per step, along two loads and one store. As in SPMV, some of the memory operations will be indexed, but exactly which ones will depend on the algorithm. We attempted three optimizations of the histogram calculation—the straightforward implementation of a histogram is not vectorizable because of the possibility of having duplicates in the input data array. The first optimization (denoted as vcc hereafter) is the default provided by Cray's vcc compiler. It attempts to vectorize the histogram computation by detecting and compensating for duplicates. This required almost no changes to the source code as the compiler performed the optimization automatically. The second method uses array privatization to ensure that updates to the histogram are independent: each strip of 16 data elements updates 16 different histogram copies (the number 16 was obtained empirically) that are summed at the end.

The third technique is the most sophisticated. It uses the sort-diff-find-diff algorithm [Math00] that makes use of the fact that it is relatively simple to update a histogram on a vector processor when the input data elements are already sorted. It therefore sorts 64 elements (the size of a VIRAM register of integers) at a time and performs the histogram update. This may seem counter-intuitive as histograms are usually functions in sorting codes [Zag91], but the approach is viable if it can be quickly implemented (without using histograms, of course) [Jan95]. Our strategy uses the permutations provided by the vector instruction set to implement a fast parallel sort in registers. Bitonic sort [Bat68] was used because the communication requirements are very regular and it proved to be a good match for VIRAM's ``butterfly'' permutation instructions, which were designed primarily for reductions and FFTs [TY99].

3.5 Unstructured Mesh Adaptation

The final benchmark is a two-dimensional unstructured mesh adaptation algorithm [OB00]. Unstructured meshes allow automatic grid generation around highly complex geometries and facilitate dynamic adaptation to efficiently capture physical features of interest that evolve with time. The resulting simulations benefit from improved efficiency when compared to a uniform grid, but the adaptive unstructured remeshing is challenging for single processors. Our benchmark involves a two-dimensional 2D_TAG unstructured mesh adaptation algorithm based on triangular elements; complete details are given for the three-dimensional procedure in [BS94]. Briefly, local mesh adaptation involves adding points to the existing grid in regions where some specified error indicator is high, and removing points from regions where the indicator is low. The advantage of such strategies is that relatively few mesh points need to be added/deleted at each refinement/coarsening step. However, complicated logic and data structures are required to keep track of the mesh objects that are inserted and removed. It involves a great deal of pointer chasing, leading to irregular and dynamic data access patterns.

In 2D_TAG, all edges of the unstructured mesh are first marked to indicate whether or not they need to be bisected, either based on geometric information or solution-driven error tolerance. In this work, every edge was bisected to eliminate propagation and simplify vectorization. The process of subdividing each triangular element isotropically into four smaller triangles is easily vectorized since there are no dependencies and the calculations are performed in a regular fashion. For example, when bisecting edges, the ids of all the new edges are precomputed and explicitly inserted into the code instead of calculating them on the fly, allowing effective vectorization. Also, mesh coarsening has not been tested at this time.

To give an indication of how vectorization and unstructured mesh adaptation are algorithmically at odds, consider the event when an edge is bisected. Every vertex in the mesh has a linked list of pointers to all the edges that are incident upon it. When an edge is bisected, the vertex pointers to the parent edge have to be updated to the newly-created child edges. To vectorize this operation, the underlying code in the reference implementation had to be rewritten. Instead of using a complex while loop to determine the position of the parent edge, a temporary array is used to copy the ids of the first edge (lets call them l_1 edges) that each vertex points. A vectorized operation then scans all the l_1 edges and appropriately replaces those that are found. The next iteration then similarly stores all the l_2 edges in the temporary array (from those vertices whose edges have not yet been replaced), and so on. The increased level of complexity for performing such a simple operation is now obvious.

Note that a similar (but much easier) strategy was implemented when adding a new pointer from a vertex to an edge. By assuming that the combined array representation of all the linked lists is densely packed, new pointers can be inserted in the correct positions by precomputing. This routine is then easily vectorized. However, our assumption would not hold true if the computational mesh were coarsened; a packing phase would then be required before inserting new pointers.

Before refining the triangular elements, they are colored to form independent sets, where two triangles have different colors if they share a vertex. This graph coloring guarantees that the data structures will not be updated inconsistently within vectorized sections of the code. It should be noted that a vectorized graph coloring algorithm was not developed; thus, the time to color the mesh is not reported. Since only one level of mesh refinement was performed, the initial coloring scheme requiring eight colors was sufficient. Note that the serial version of 2D_TAG does not require graph coloring.

The data structures are then reordered by element color (and by subdivision type within each color). This allows all the elements of a given color and subdivision pattern to be easily vectorized, since blocks of elements that have the same refinement type without any dependencies can safely be processed concurrently. Recall that in the current vectorized version of the code, the mesh is refined isotropically; thus, all triangular elements are subdivided into four smaller triangles. (In general, this 1:4 operation is responsible for over 90% of the total refinement overhead). Note that this data reordering section of the code has not yet been vectorized, since vectorizing data reshuffle is a difficult task. As a result, reordering is responsible for over 50% of the cycles during the simulation. Again, the serial implementation of 2D_TAG does not require this reordering overhead.

Many other modifications are also required to vectorize the individual element subdivision routines. First, several global variables have to be precomputed (instead of dynamic assignment that is used in the original serial code) to help the compiler understand that there are no dependencies. We handle dependency analysis explicitly through graph coloring.

Second, the compiler does not allow functions to be included in a vectorized code segment, and cannot perform inlining correctly. Thus, all function calls (and any functions that are subsequently called) had to be hand unrolled. The body of the main loop over elements for each subdivision type was quite large; as a result, the compiler was unable to effectively vectorize it. This is probably due to register spilling, and the high complexity of performing dependency and flow analysis for such a large loop body. As a result, the loops needed to be reduced to small chunks. The 1:4 loop body was divided into 12 separate subloops, where each subloop processes all the colors before proceeding to the next subloop. It was only in this manner that the 1:4 subdivisions could be effectively vectorized.

3.6 Summary of Benchmark Characteristics

The following table summarizes the key features of each of the benchmarks. All of these benchmarks are memory-intensive in the following sense: the number of arithmetic or logical operations per step of the algorithm (Ops/step) is never more than the number of memory operations per step (Mem/step). Most of them involve some amount of irregular memory access, indicated in the table as indexed, although in the case of SPMV and histogram, the number and kind of indexed memory operation differs across different versions. Although not shown in the table, the algorithms also different in the degree of data-parallelism, which may limit the vector length on VIRAM. For the sparse matrix algorithms, it is typically a function of the number of nonzeros per row, while in something like GUPS, it is the total data set size. For histogram, while there may be parallelism across the data set, we cannot know that without knowing the data is free of duplicate elements. Finally, we note that the mesh adaptation code is the only benchmark in this set with significant control irregularity, since there are several different cases for subdividing triangles, based on their shape. Like histogram, mesh has some limits to parallelism, since triangles may shared vertices.

	Data type	**Memory access**	**Data size**	**Total Ops**	**Ops/step**	**Mem/step**
Transitive	32-bit int	unit	n^2	n^3	2 (min, +)	2 ld
GUPS	8,16,32,64-bit int	indexed + unit	2n	2n	1 (+)	2 ld, 1st
SPMV	32-bit float	indexed + unit	2m + 2n	2m	2 (*, +)	3 ld
Histogram	16,32-bit int	indexed + unit	n + b	n	1 (+)	2 ld, 1 st
Mesh	32-bit int, float	indexed + unit	1000n	NA	NA	NA

4. Simulation Results

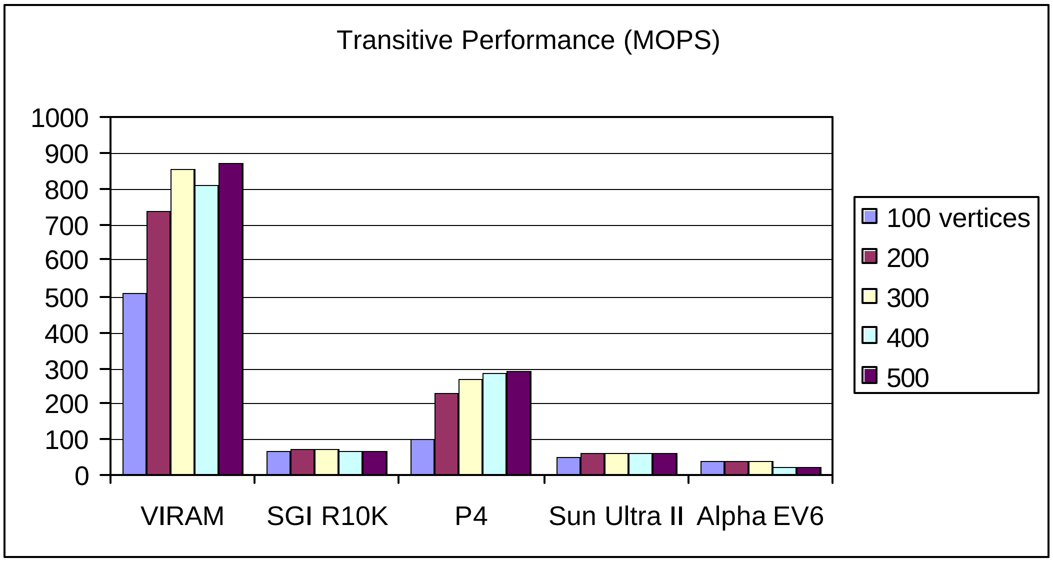
In the course of evaluating our benchmarks and the VIRAM simulation, we also used several commercial platforms, generally of the workstation or small server variety, for comparison. Hardware and software details of these machines are listed below:

Architecture	UltraSPARC IIi	MIPS R10000	Pentium III	Pentium 4	Alpha EV6
Make	Sun Ultra 10	SGI Origin2000	Intel Mobile	Dell	Compaq DS10
Clock speed	333 MHz	180 MHz	600 MHz	1.5 GHz	466 MHz
L1 cache	16 KB + 16 KB	32 KB + 32 KB	None	12 KB + 8KB	None
L2 cache	2 MB	1 MB	256 KB	256 KB	256 KB
Memory	256 MB	1 GB	128 MB	1 GB	512 MB
Operating system	SunOS 5.8	IRIX64 6.5	Debian GNU/Linux 2.2r3 Kernel 2.2.19	Red Hat Linux 7.1 Kernel 2.4.7	Red Hat Linux 6.0 Kernel 2.2.19
Compiler	Sun Workshop 6 v 1C5.2	MIPSpro v 7.2.1.3m	Debian GNU/Linux v 2.95.2	Intel v 5.0.1 beta	Compaq v C.6.4-005
Compiler options	cc -fast	cc -O3 -n32 -mips4 -TARG: processor=r10000	gcc -O2 -funroll_loops -march=pentiumpro	icc -X -O3 -tpp7 -xW -unroll	ccc -arch ev6 -fast -tune ev6

The VIRAM compiler is based on Cray's vectorizing compiler, which is based on over 20 years of experience in automatic vectorization and is used on supercomputers like the Cray C90 and T90. The VIRAM version has its own backend that generates a mixture of MIPS scalar instructions and VIRAM vector instructions. While the vectorizer is quite sophisticated, the code generator has not gone through the kind of rigorous performance tuning that we would expect from a commercial compiler. In particular, there are cases in which the compiler generates extra boundary checks and redundant loads. The extra boundary checking mainly involved generating strip-mining code for loops where it was provably not necessary. The redundant loads (i.e., reloading the same value to a register which had not been overwritten or otherwise killed) were of unknown origin; probably they represent some bug in the register allocation or spill handling. The VIRAM chip is scheduled for fabrication near the end of this year, so numbers reported here are based on a cycle-accurate simulator of the chip.

4.1 Transitive Closure

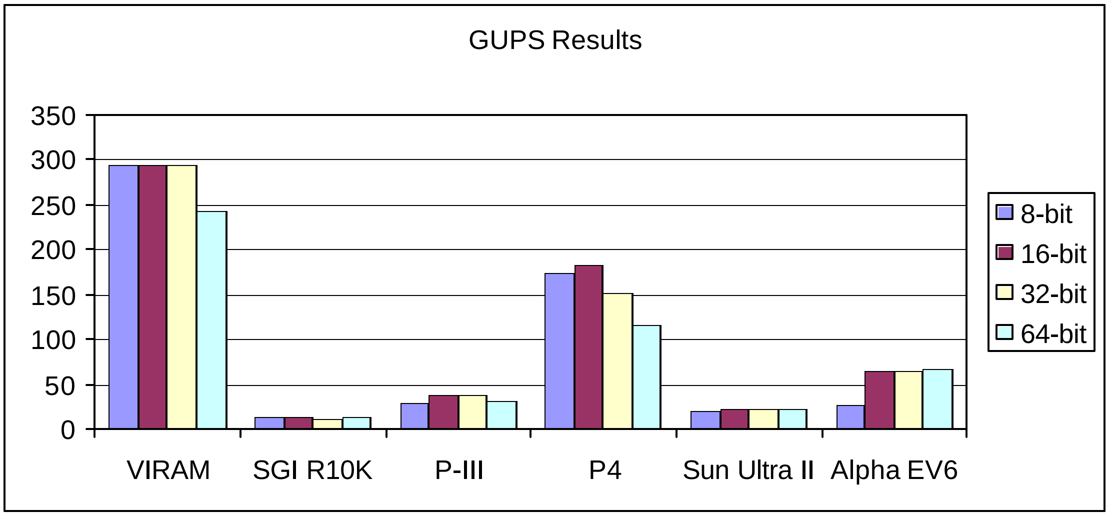
The following graph shows the performance of the transitive closure benchmark in MOPS. On all machines, the version shown uses the unit stride memory access. Although transitive closer is an O(n^3) algorithm on O(n^2), similar in may ways to matrix multiply, the cache-based machines perform well below their peak performance. This is consistent with results we have sense on dense matrix-vector multiplication, and demonstrates the advantage of the IRAM architecture for a problem with abundant parallelism and low arithmetic/memory operation ratio.



4.2 GUPS Benchmark

The graph belows shows a comparison of all our machines against VIRAM-1 for the GUPS benchmark. Notice that the VIRAM simulations obtain about 0.12 more GUPS than its closest competitor (Pentium-4), and that most machines perform their best when using 16- or 32-bit integers.

To evaluate the effectiveness of the benchmarks themselves as measures of memory system performance, we were able to look at various statistics from the VIRAM simulation. One salient characteristic of each benchmark is the fraction of memory bandwidth used compared to the theoretical maximum. GUPS achieves 1.77, 2.36, 3.54, and 4.87 GB/s memory bandwidth on the VIRAM-1 simulator at 8-, 16-, 32-, and 64-bit data widths, respectively. Given other instruction overhead, this is close to the peak memory bandwidth of 6.4 GB/s, showing that GUPS.



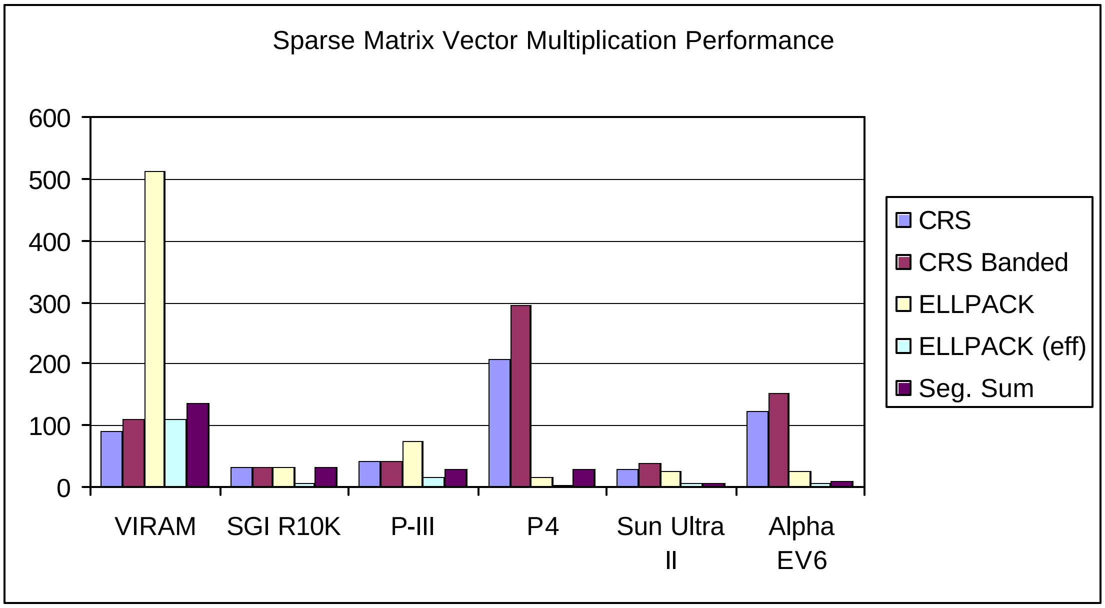
For the GUPS benchmark, we tidied up the compiler-generated assembly instructions for the inner loops by hand. The resulting code was usually about 20--25% faster, but was 60% faster in the 64-bit case. We believe that the correct approach is to use the C compiler for benchmark codes, and to resort to assembly language only when code generation problems significantly impede progress.

As with the Transitive Closure benchmark, GUPS shows off the benefits of VIRAM for this problem that is truly memory-intensive, especially for 64-bit data. This problem is much more challenging for both classes of machines due to the nearly random memory access pattern within the data vector. None of these machines show significant scaling when the data width is decreased. On IRAM, at least, one might expect the subdivision of lanes could double the performance as the width is halved. This has been demonstrated for arithmetically intensive problems, and also for unit stride applications. A limitation of the VIRAM microarchitecture is as single address generation unit per lane, so at any data width, only 4 distinct addresses can be generated per cycle. This was a conscious design choice in VIRAM, because of the hardware costs of parallel address generation would have incurred significant costs in both chip area and design complexity.

4.3 Sparse Matrix Stressmark

In our simulations, the matrix A and the right-hand side vector y are not read from a file. Instead, they are generated at runtime based on the dimension n, the number of nonzero elements m, and a seed value for the random number generator. All the entries of A and b are therefore random. Care is taken to make A symmetric positive definite, thereby ensuring termination of the algorithm. Furthermore, A is populated such that its nonzero values yield a strictly diagonally dominant matrix. The detailed matrix generation algorithm can be found in the DIS Stressmark specification [DIS00].

For our benchmark matrix, we set n=10000 and m=177782, i.e., there were about 18 nonzeros per row. The graph below shows the peformance of SPMV using our different matrix formats and algorithms. When using a bandwidth reduction ordering (simulated by the CRS-banded format), the x vector is still accessed using indexed loads, so address generation bandwidth is a problem, but because the elements are nearby, bank conflicts are not a problem. (Other data that varies the bank structure of the memory system confirm this.) The performance for the Ellpack format is divided into observed and effective (eff). Recall that the Ellpack format adds a significant number of zeros to the matrix: the observed MFLOP rates counts operations on those zeros, while the effective rate does not. The Ellpack format is best suited to vectorization, but for this randomly constructed DIS matrix, the additional zeros make it impractical. Many matrices that occur in practice have less variance in the number of nonzeros per row, so the Ellpack algorithm may prove useful. For this matrix, Segmented-sum consistently delivers the best performance. This is not surprising, since Segmented-sum vectorizes well and does not incur the overhead of Ellpack.

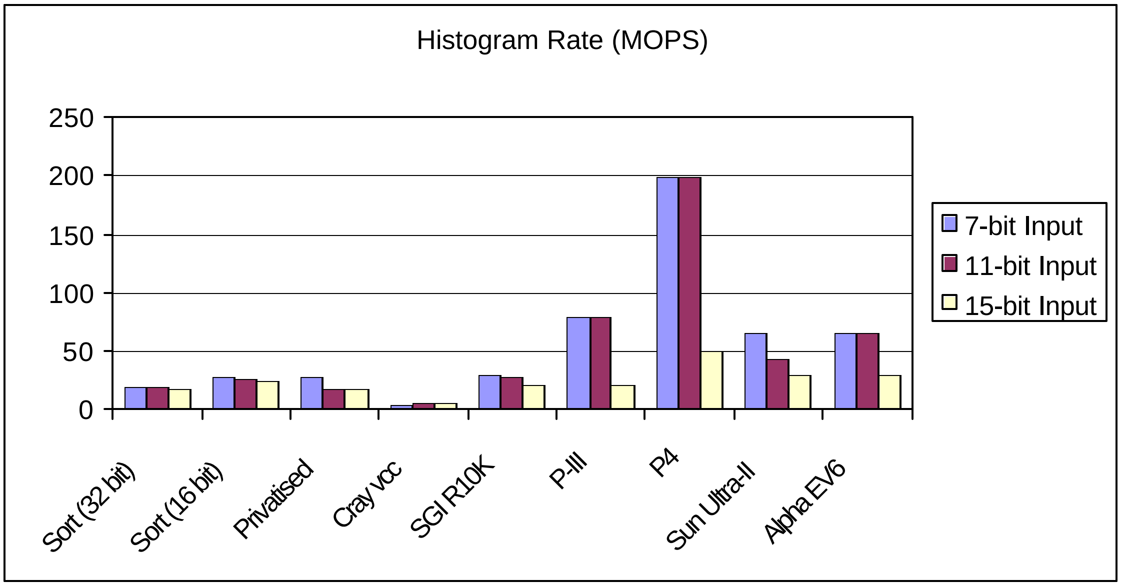


Performance on the superscalar cache-based architectures differs significantly from that on the VIRAM. The Segmented-sum format is the worst among all four formats. On the Pentium 4 and Alpha EV6, it is more than an order of magnitude poorer than the CRS-banded format. This is because in the SPMV inner-loop when using Segmented-sum, there are ``if-then-else'' tests for detecting each segment's boundary, which severely limits the utilization of the floating-point pipeline. The vector architectures, such as the Cray PVP and VIRAM, usually have vector mask instructions that can execute conditional statements very efficiently. The CRS-banded format (simulating the use of a bandwidth reduction ordering) is always better than CRS. This is because the source vector is accessed with much better spatial locality, leading to better cache line utilization. We notice a surprisingly low performance for Ellpack on the Pentium 4 and EV6, even when observed performance is used. Ellpack performs more stores than the other version, which partially explains this data, although the drop relative to other formats is higher than we would expect.

Comparisons with the VIRAM runtimes also demonstrate that the performance of the Segmented-sum and Ellpack formats are always much worse than that for the VIRAM. The CRS and CRS-banded formats exhibit better performance on the two newer machines with much higher clock rates, Pentium 4 and Alpha EV6; but they are much worse on the UltraSPARC IIi and MIPS R10000.

4.4 Histogram Benchmark

Our performance data for the histogram problem varies the number of buckets in the histogram (which is expressed as log base 2, namely the bit depth in an image).



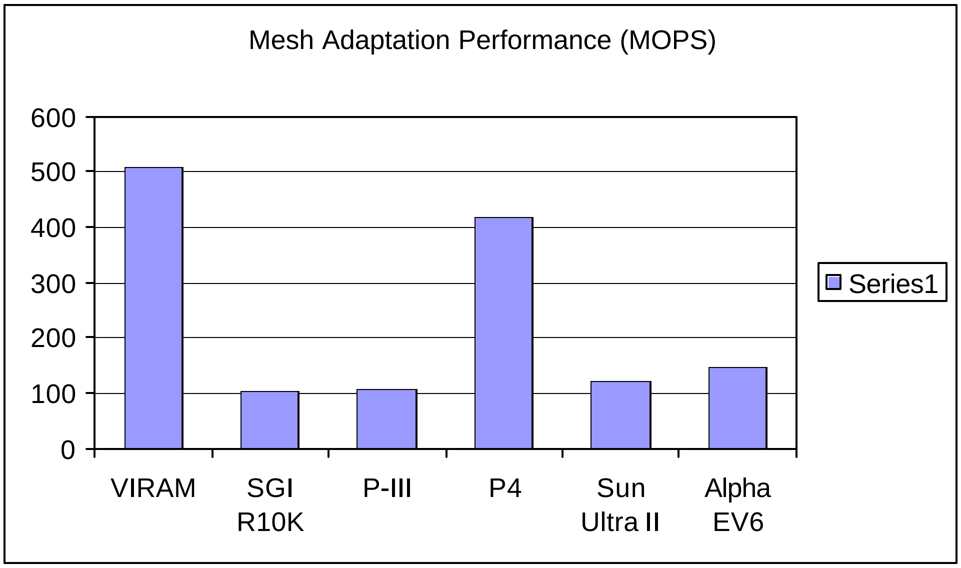
This impacts the size of the histogram and hence, computation time. We present two different timings for the sort-based optimization method, one that work for data in this image processin problem, where the pixels are always less than 16bits, and another for data that is upt ot 32-bits wide. When the narrower width, we can exploit the fact that vector operations on "shorts" are usually twice the speed of vector operations on "ints" on VIRAM. We also present results on our five other comparable architectures.

Results show that on the VIRAM-1, the sort-based and privatization optimization methods consistently give the best performance over the range of bit depths. It also demonstrates the improvements that can be obtained when the algorithm is tailored to shorter bit depths. The Cray default compiler optimization performs poorly primarily because of the presence of many duplicates. The code to compensate for duplicates essentially executes in scalar mode and thus degrades performance. Note that, unlike all the other results, the vcc optimization improved performance as the bit depth increased due to fewer duplicates in the sum and difference data arrays. The privatization code suffers because it needs to sum up the histogram copies at the end and so requires more operations as the size of the histogram increases.

On superscalar cache-based machines, duplicates are not much of an issue. In fact, they may register a cache hit when updated a second or third time. We therefore see excellent timings for the histogram computation on these machines without any special optimizations. For larger histograms that do not fit entirely in cache, VIRAM's performance compares well with the faster microprocessors. The VIRAM logarithm routine vectorized well and provided excellent performance resulting in low overall execution times. However, this was only an issue for the largest bit depth (largest histogram).

4.5 Unstructured Mesh Adaptation

The computational mesh used for our experiments consisted of 4802 triangular elements, 2500 vertices, and 7301 edges. Only one level of isotropic refinement was performed that increased the mesh to 19208 elements, 9801 vertices, and 29008 edges. A large fraction of 2D_TAG has been successfully vectorized, and the results below show that, in general, the simulated runtimes are significantly lower than on cache-based systems.



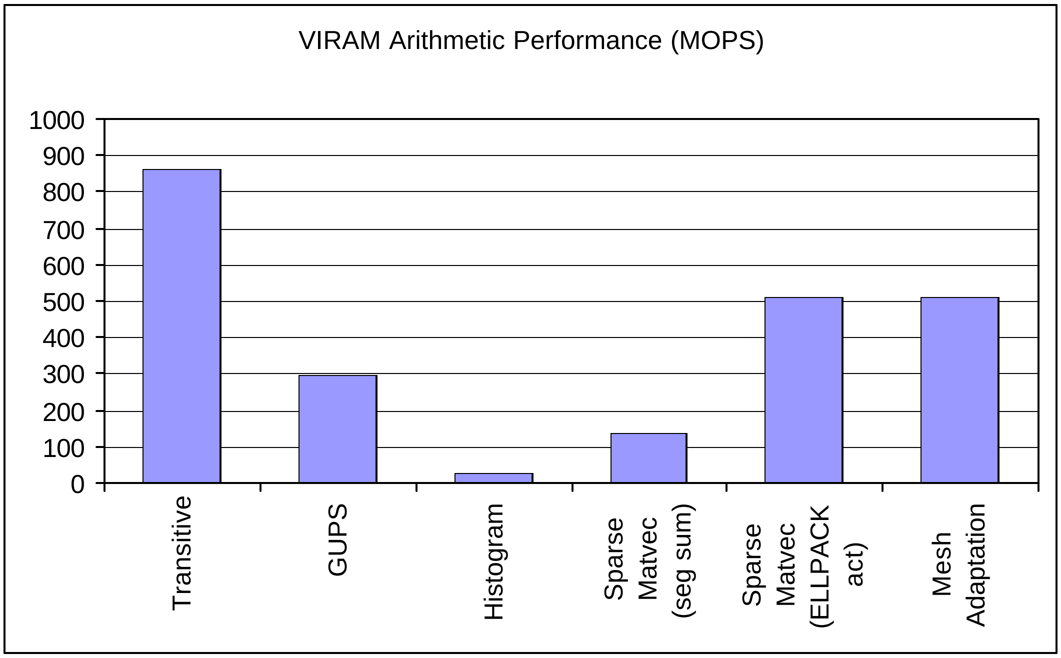
This performance gain is due to several reasons. Since mesh adaptation has irregular and dynamic data structures, these relatively small cache sizes are insufficient to hold the working set, and as a result must go off-chip to access main memory. The on-chip DRAM of the VIRAM greatly improves the bandwidth and latency characteristics of memory. In addition, more than 77% of the operations are vectorized, enabling VIRAM to take advantage of multiple vector functional units which allow up to eight parallel 32-bit arithmetic operations and load/stores per cycle. Finally, there are many conditional branches within the subdivision phase of 2D_TAG. VIRAM handles these by processing each branch separately using bit masks. However, the cache-based systems use speculative branch predictions, with the penalty for branch mispredictions growing with deeper pipelines and wider issue rates. Since branching in 2D_TAG is quite unpredictable, it reduces the effectiveness of speculative prediction.

Notice that there is little performance degradation when reducing the number of vector lanes from four to two. Since 2D_TAG mostly performs indexed addressing and VIRAM can generate a maximum of four independent addresses per cycle, it cannot use all eight functional units (for 32-bit operations) in a cycle. However, there is more than a 40% slowdown between four lanes and one lane since fewer vector operations are performed with the narrow one lane configuration. This is also seen in the memory bandwidth to/from the vector units, which decreases with fewer vector lanes. Finally, increasing the number of subbanks to four in the simulation does not have a dramatic impact. This indicates that memory-bank conflicts are a significant bottleneck for our dynamic adaptation code.

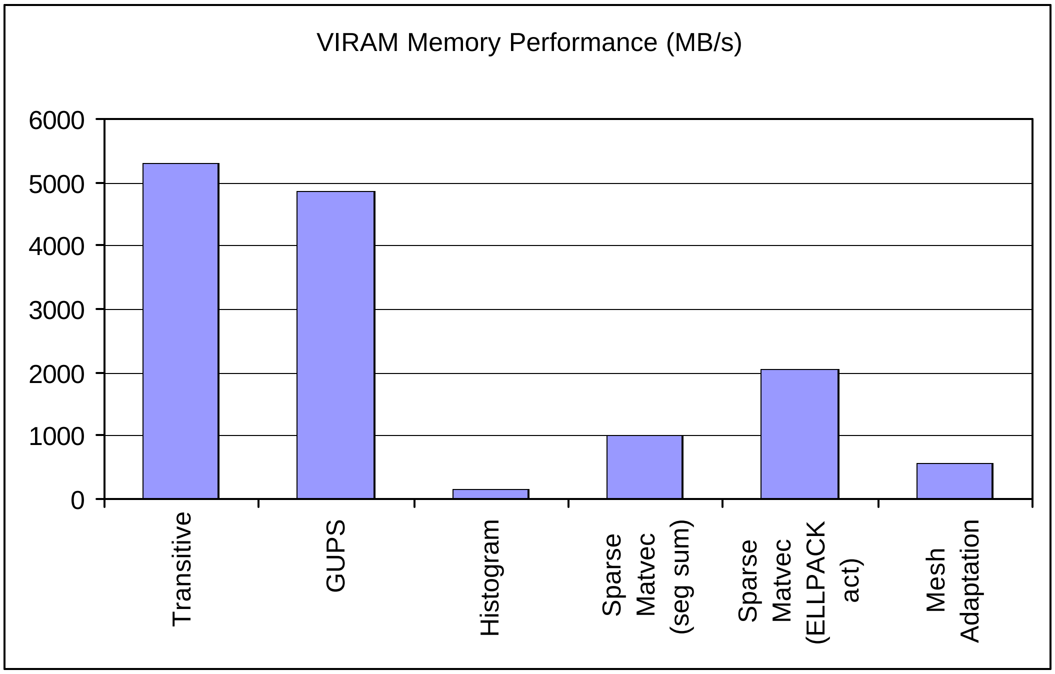
4.6 Summarizing VIRAM Results

One of our underlying questions in this work was what would prove to be the limiting factor in these memory-intensive benchmarks. The following two graphs give some indication of this. The first summarizes the MOPS rate achieved on each of the benchmarks using the best algorithm on VIRAM. (We include the observed performance of Ellpack in addition to segmented scan for SPMV, because we believe Ellpack may be of practical interest for other matrices.)

It is interesting to note that the GUPS and histogram applications are nearly identical in their memory behavior and in their arithmetic operations; the difference in running time is entirely due to the lack of parallelism in the specification of histogram (duplicates must be preserved) and the large number of duplicates that occur in the particular dataset. Thus, histogram is entirely limited by lack of fine-grained parallelism for this data set, not by memory system performance.



In terms of actual memory bandwidth limitations, we see that both transitive and GUPS are near the peak VIRAM bandwidth of 6.4 GB/s. None of the other applications appear to be limited by memory bandwidth per se. Instead, because all of these involved indexed operations on data less than 64-bits wide, all of them are probably limited in part by address generation bandwidth. In addition, there are some limits to the fine-grained parallelism in all of these applications, which may appear as masked operations which waste processing resources, short vector lengths, or unnecessary computation.



5. Conclusions and Future Work

Our experience with these memory-intensive benchmarks suggest that VIRAM-1 is significantly faster than the cache-based machines for problems that are limited only by DRAM bandwidth and latency. Because IRAM achieves its high performance through parallelism, rather than clock rate, the advantages are even larger if one is interested in building a energy efficient system for scientific computation: compare hundreds of Watts for a Pentium based processing node, compared with 2 Watts for VIRAM.

The dependence on fine-grained data parallelism also limits the VIRAM advantage when the problem does not exhibit these characteristics. Although the histogram and mesh adaptation problem are both parallelizable, the potential for data sharing even within the vector operations limits performance. The need to statically specify parallelism also requires algorithms that are highly regular. In SPMV this lead to data structure padding, and in both mesh adaptation and histogram there was some level of sorting to group uniform data elements together.

The complexity of vectorizing implicitly parallel C codes so that they may be run efficiently on a vector architecture like VIRAM remains high in the sense that while certain codes yield quickly to vectorization techniques, it can be quite challenging for many others. As a result, a conscientious programmer cannot expect the compiler to do a satisfactory job when performance is critical, simply because some instruction sequences are not generated by the compiler (and their non-vectorized alternatives are quite a bit more costly). However, the Cray compiler technology for VIRAM does a better job of vectorization than existing workstation competitors, e.g., the Intel C compiler for the Pentium III and Pentium 4. Our experience on VIRAM has been that typical codes yield 20% better performance with a small amount of human attention to the generated assembly code output from the compiler, but this figure varies wildly depending on the algorithm and the data structures used. More hand-optimizing of generated assembly code may be required, depending on progress made on the compiler.

Moreover, the memory subsystem interfaces for vector instructions (indexed and variable-stride loads/stores, in addition to the usual unit-stride operations) make VIRAM assembly language programming much simpler and more flexibly than the Intel MMX/SSE extensions. A caveat is that indexed and variable stride operations are very costly in terms of address generation and conflict resolution, which becomes especially noticeable for narrow data types. Codes that significantly depend on these capabilities may find tremendous performance improvements when they are reorganized to use unit stride; data structures optimized to support unit stride may need to be substituted for more traditional data structures in such codes.

In our work so far, we have concentrated on comparing a simulated VIRAM system with modern workstation-class machines, although VIRAM is designed for hand-held, low power devices. Comparisons with other processor-in-memory chips and with stream computation machines such as Stanford's Imagine chip will help place our performance results in better perspective. In addition, this study has explored the performance of computations that fit on a single chip VIRAM, without addressing the problems that arise when either the data set size or the computational demands are too large for a single chip. The DIVA effort at ISI is exploring the space of multiprocessor PIM systems. Short of building a multiprocessor VIRAM system, we believe that insight into these performance issues can be gained by combining the kind of parallel benchmarking results they have published [HKK+99] with our own detailed single processor numbers. Finally, of course, it will be indispensable to re-run our benchmarks on the real VIRAM hardware once the system is available.

References

[Bat68] K. Batcher, “Sorting networks and their applications,” *Proc. AFIPS Spring Joint Compute Conf.*, 1968.

[BS94] R. Biswas and R. C. Strawn, “A new procdure for dynamic adoption of three-dimensional unstructured grids,” *Appl. Numer. Math.* 13(6) pp. 437-452, 1994.

[BHZ93] G.E. Blelloch, M. A. Heroux, and M. Zagha, “Segmented operations for sparse matrix computation on vector multiprocessors,” Tech. Rep. CMU-CS-93-173, Carnegie Mellon Univ., Pittsburgh, PA, 1993.

[CM69] E. Cuthill and J. McKee, “Reducing the bandwidth of sparse symmetric matrices,” *Proc. ACM Natl. Conf.*, pp. 157-192, 1969.
[DIS00] DIS Stressmark Suite: Specifications for the Stressmarks of the DIS Benchmark Project, v 1.0, Titan Systems Corporation, 2000, available at http://www.aaec.com/projectweb/dis/DIS_Stressmarks_v1_0.pdf.

[Gae01] B. R. Gaeke, “GUPS Benchmark Manual,” Univ. of California, Berkeley, CA, available at http://iram.cs.berkeley.edu/~brg/dis/stresscode/README.gups.20010625.txt.

[GTM+97] G. Gao, K. Theobald, A. Marquez, T. Sterling, "The HTMT Program Execution Model," CAPSL Technical Memo 09, University of Delaware, Newark, Delaware, July 1997.

[Geo71] A. George, "Computer implementation of the finite element method," Tech. Rep. STAN-CS-208, Stanford Univ., Stanford, CA, 1971.

[HKK+99] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," SC'99, November 1999.

[Koo00] P. Koopman, "Maximal Length LFSR Feedback Terms," Carnegie Mellon Univ., Pittsburgh, PA, available at http://www.cs.cmu.edu/~koopman/lfsr.

[Koz99] C. Kozyrakis: A Media-Enhanced Vector Architecture for Embedded Memory Systems", Technical Report UCB//CSD-99-1059, University of California, Berkeley, July 1999.

[KJG+01] C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, K. Yelick, "Hardware/Compiler Co-development for an Embedded Media Processor," Proceedings of the IEEE. To appear.

[Math00] How do I Vectorize My Code? Tech. Note 1109, The MathWorks, 2000, available at http://www.mathworks.com/support/tech-notes/1100/1109.shtml.

[OB00] L. Oliker and R. Biswas, "Parallelization of a dynamic unstructured algorithm using three leading programming paradigms," *IEEE Trans. Parallel and Distributed Systems*, 11(9), pp. 931-940, 2000.

[PP95] J. Jang, H. Park, and V. K. Prasanna, "A fast algorithm for computing a histogram on reconfigurable mesh," *IEEE Trans. Pattern Analysis and Machine Intelligence*, 17(2), 1995.

[TY99] R. Thomas and K. Yelick, "Efficient FFTs on IRAM," First Workshop on Media Processors and DSPs, November 15, 1999.