

Improving the Performance of Sparse Matrix-vector Multiplication by Blocking

William D. Gropp [1]

Dinesh K. Kaushik [1,2]

David E. Keyes [2,3,4]

Barry F. Smith [1]

[1] Argonne National Laboratory

[2] Old Dominion University

[3] Lawrence Livermore National Laboratory

[4] ICASE, NASA Langley Research Center

Outline

- Performance analysis of sparse matrix-vector product for 1 to N independent vectors
- Complexity models (flops, loads, stores, “stream loads and stores”)
- Refinements and future directions
- Summary of architectural “headroom” for multivectors

Background

- Historic memory performance improvement rate (7% per year) is far behind the CPU performance growth (about 55% per year)
- Sequential performance on many machines is a low percentage of “peak”
- Complexity analysis of algorithms based on floating point operations alone is deceptive
- A proper complexity model for the sparse matrix-vector product illustrates the issues
 - ◆ Same data access considerations as stencil-op kernel in explicit methods for PDEs
 - ◆ Same as Krylov kernel and similar to preconditioner application kernel in implicit methods for PDEs

Three Potential Rate Limiters on Arithmetic Performance

- Memory Bandwidth
 - ◆ Processor does not get data at the rate it requires
- Instruction Issue Rate
 - ◆ If the loops are load/store bound, cannot perform a floating point operation in every cycle even if the operands are available in primary cache
 - ◆ Several constraints (like primary cache latency, latency of floating point units etc.) must be observed in deriving an optimal schedule
- Fraction of Floating Point Operations
 - ◆ Not every instruction is a floating point instruction
- Each of these forces counting something besides *just* floating point operations

Implications of Bandwidth Limitations in Shared Memory Systems

- The processors on a node compete for the available memory bandwidth
- The computational phases that are memory bandwidth limited will not scale
 - ◆ may even run slower because of the extra synchronizations

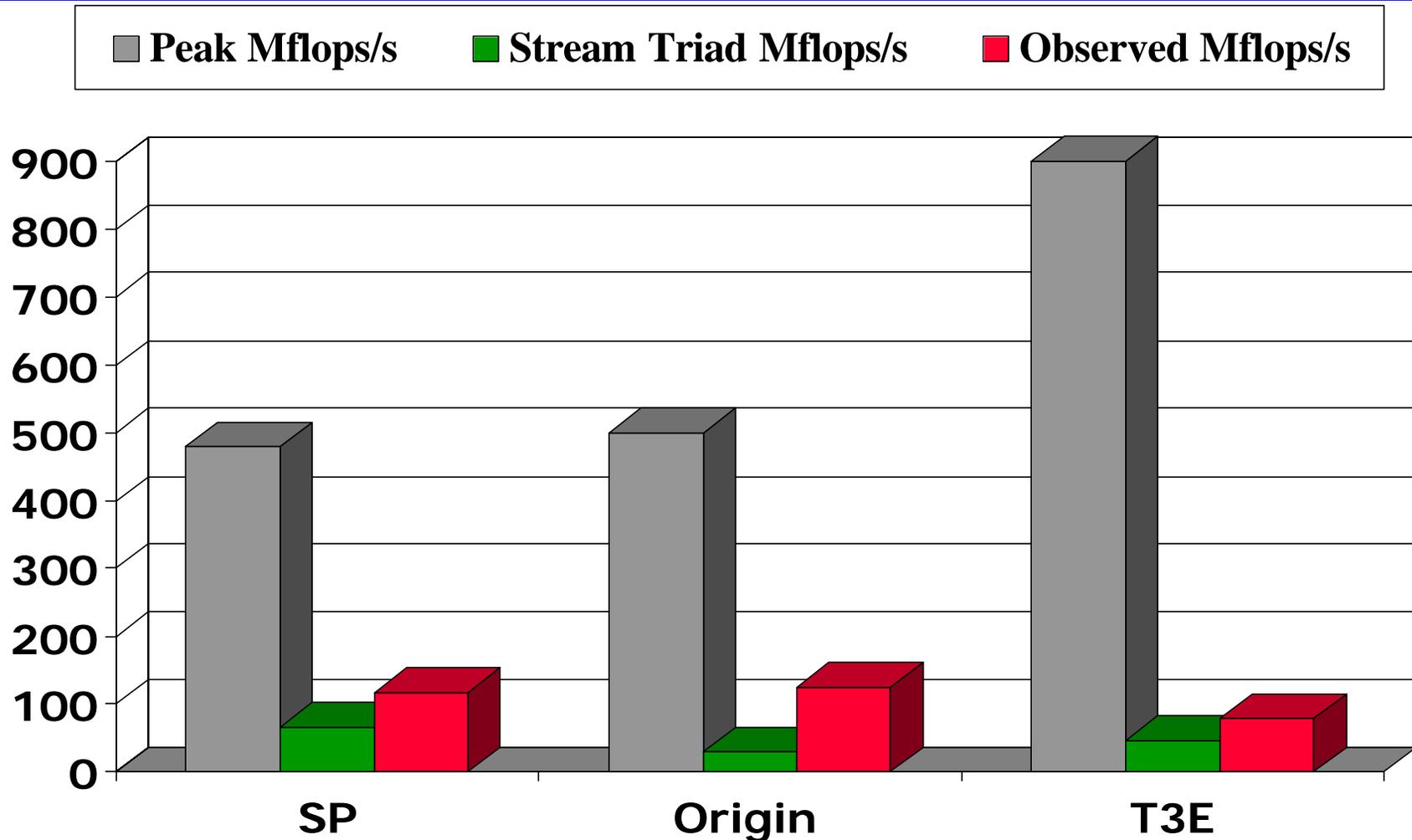
Stream Benchmark on ASCI Red

MB/s for the Triad Operation

Vector Size	1 Thread	2 Threads
1E04	666	1296
5E04	137	238
1E05	140	144
1E06	145	141
1E07	157	152

Larger vectors in last three rows do not fit into cache and are bandwidth-limited

Sequential Performance of PETSc-FUN3D



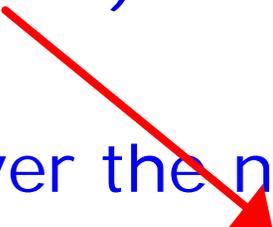
Note the poor prediction by "Peak Mflops/s" and better prediction based on "Stream"

Analysis of a Simple Kernel: Sparse Matrix Product

- Sparse matrix vector product is important part of many iterative solvers
- Simple analysis predicts much better performance bounds (based on the three architectural limits) than the “marketing” peak of a processor

Matrix-vector Multiplication for a Single Vector

```
do i=1, n
  fetch ia(i+1)
  sum = 0
  ! loop over the non-zeros of the row
  do j = ia(i), ia(i + 1)-1 {
    fetch ja(j), a(j), x (ja(j))
    sum = sum + a(j) * x(ja(j))
  }
  enddo
  Store sum into y(i)
enddo
```



Matrix Vector Multiplication for N Independent Vectors

```
do i = 1, n
  fetch ia(i+1)
  ! loop over the non-zeros of the row
  do j = ia(i), ia(i + 1) - 1
    fetch ja(j), a(j), x1(ja(j)), .....xN(ja(j))
    do N fmadd (floating multiply add)
  enddo
  Store y1(i) .....yN(i)
enddo
```

This version performs $A \{x_1, \dots, x_N\}$

Performance Issues for Sparse Matrix Vector Product

- Low available memory bandwidth combined with little data reuse
- High ratio of loads/stores to instructions/floating-point ops
- Stalling of multiple load/store units on the same cache line

Multivectors: Increasing Matrix Cache-line Reuse

- Multiplying more than one vector with the same matrix reuses the matrix-entry cache lines
- With enhanced matrix temporal locality, vector spatial locality can suffer, however
 - ◆ Possibility of more vector-entry cache line conflicts
 - ◆ Possibility of output-vector cache line conflicts analogous to *false sharing* multithreaded applications
 - ◆ Allocation of the vectors in memory – interlaced or non-interlaced?
 - TLB misses (effect is to punish applications that refer to many different memory pages, even if total cache usage isn't large)
 - ◆ The lack of a universal, detailed model of performance makes detailed performance prediction difficult and case-by-case

Estimating the Memory Bandwidth Limitation

Assumptions

- Perfect Cache (only compulsory misses; no overhead)
- No memory latency
- Unlimited number of loads and stores per cycle

Data Volume (m*n matrix in AIJ Format)

$m * \text{sizeof}(\text{int}) + N * (m+n) * \text{sizeof}(\text{double})$

(ia, N input (size n) and output (size m) vectors)

+ nz* (sizeof(int) + sizeof(double))

(ja, and a arrays)

= $4 * (m+nz) + 8 * (N * (m+n) + nz)$

Estimating the Memory Bandwidth Limitation II

- Number of Floating-Point Multiply Add (fmadd) Ops = $N * nz$
- For square matrices ($m=n$) in **AIJ** format,

$$\text{Bytes transferred/fmadd} = \left(16 + \frac{4}{N}\right) * \frac{n}{nz} + \frac{12}{N}$$

(Since $nz \gg n$, Bytes transferred / fmadd $\sim 12/N$)

- Similarly, for **Block AIJ (BAIJ)** format (blocksize b)

$$\text{Bytes transferred/fmadd} = \left(16 + \frac{4}{N * b}\right) * \frac{n}{nz} + \left(\frac{4}{N * b} + \frac{8}{N}\right)$$

$N = \#$ vectors, $n = \#$ rows, $nz = \#$ nonzeros in A, double=8 bytes, int = 4 bytes

Computing an Estimate of Maximum Possible Performance

- Bytes per floating multiply-add combined with memory bandwidth (bytes/second) give a bound on rate of execution of multiply-adds
- Quoted (vendor-supplied) memory bandwidth numbers are often useless
 - ◆ Details of memory system hardware strongly affect performance and can be difficult to uncover
- Fortunately, a simple measurement is often adequate

Performance Summary on 250 MHz R10000

- Matrix size, $n = 90,708$; number of nonzero entries, $nz = 5,047,120$ (from computational aerodynamics, $b=4$)
- Stream performance is 358 MB/sec (for triad vector operation) <http://www.cs.virginia.edu/stream>
- Number of Vectors, $N = 1$, and 4

Format	Number of Vectors	Bytes / fmadd	Bandwidth		MFlops	
			Required	Achieved	Ideal	Achieved
AIJ	1	12.36	3090	276	58	45
AIJ	4	3.31	827	221	216	120
BAIJ	1	9.31	2327		84	55
BAIJ	4	2.54	635	229	305	175

- Ratio of 2.7 for **AIJ** and 3.2 for **BAIJ** in going from 1 to 4

Prefetching - Fully Use the Available Memory Bandwidth

- Many programs are not able to use the available memory bandwidth for various reasons
- Ideally a memory operation should be scheduled in each cycle since each cycle is a lost opportunity
- Compilers do not do enough prefetching
- Implementing and estimating the right amount of prefetching is hard

Estimating the Operation Issue Limitation, I

AT: address transln; **Br**: branch; **Iop**: integer op; **Fop**: floating point op;
Of: offset calculation; **Ld**: load; **St**: store

```
do i=1, m
  jrow = ia(i+1)           // 1Of, AT, Ld
  ncol = ia(i+1) - ia(i)  // 1 Iop
  Initialize, sum1 .....sumN // N Ld
  do j=1,ncol             // 1 Ld
    fetch ja(jrow), a(jrow), x1(ja(jrow)), .....xN(ja(jrow))
    // 1 Of, N+2 AT,
    // N+2 Ld
    do N fmadd (floating multiply add) // 2N Fop
  enddo // 1 Iop, 1 Br
  Store sum1.....sumN in y1(i) .....yN(i) // 1 Of, N AT, and St
enddo // 1 Iop, 1 Br
```

Estimating the Operation Issue Limitation, II

- Assumptions:
 - ◆ Data items are in cache
 - ◆ Each operation takes only one cycle to complete but multiple operations can graduate in one cycle
- If only one load or store can be issued in one cycle (as is the case on R10000 and many other processors), the best we can hope for is

$$\frac{\text{Number of floating point instructions}}{\text{Number of Loads and Stores}} * \text{Peak MFlops/s}$$

- Other restrictions (like primary cache latency, latency of floating point units etc.) need to be taken into account while creating the best schedule

Estimating the Fraction of Floating Point Operations

- Estimated number of floating point operations out of the total instructions:

Total number of instructions completed (I_t) = $m * (3 * N + 8) + nz * (4 * N + 9)$

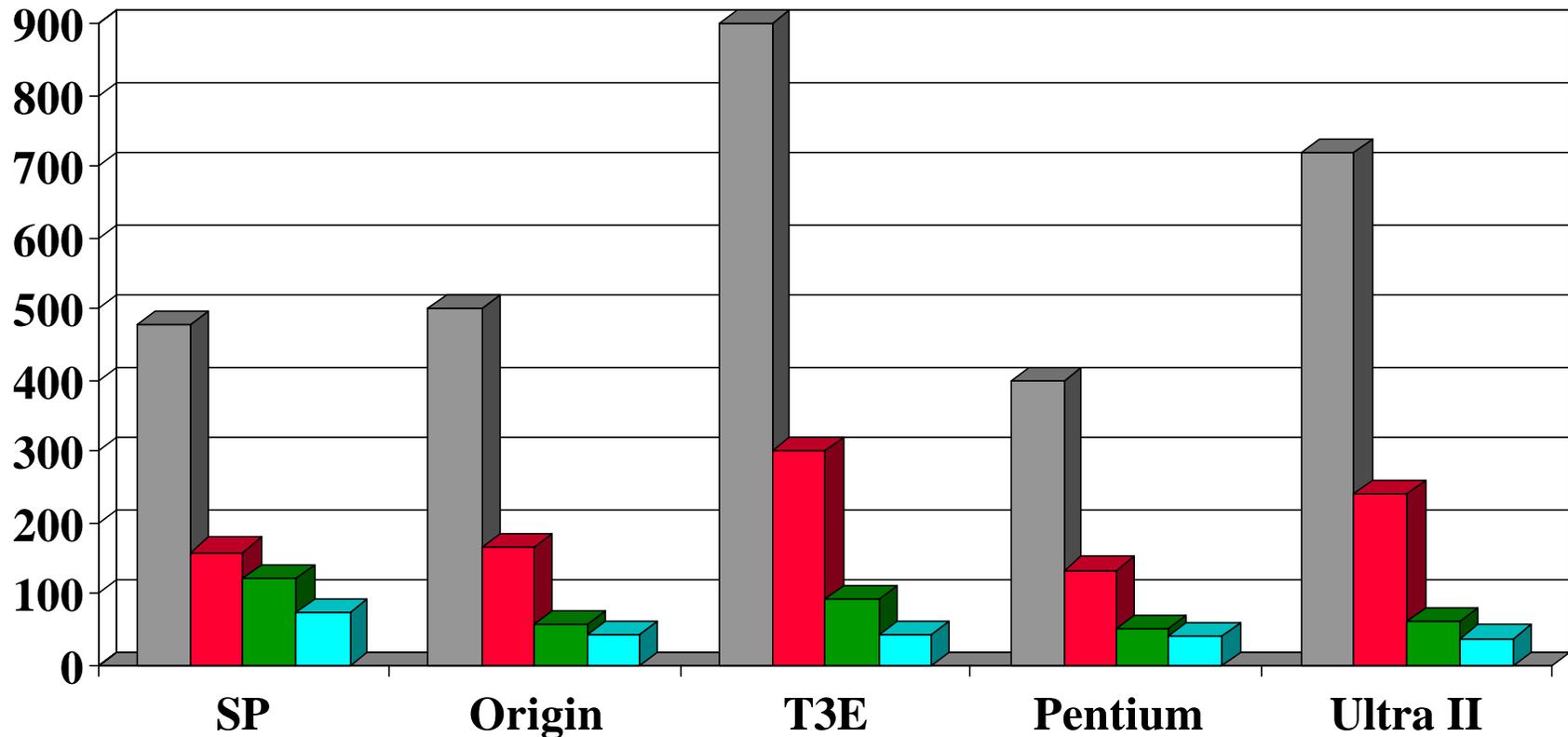
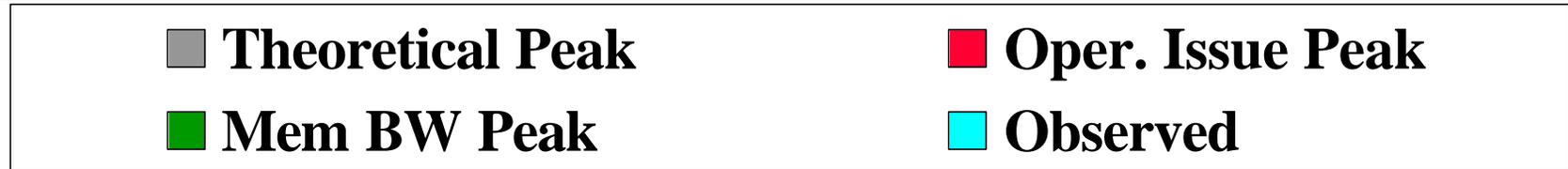
Fraction spent on floating point work (I_f) = $\frac{2 * N * nz}{m * (3 * N + 8) + nz * (4 * N + 9)}$

- For **$N=1$** , **$I_f = 0.18$**
- For **$N = 4$** , **$I_f = 0.34$** , this is **one-third** of “peak” performance (for the aero example)

Realistic Measures of Peak Performance

Sparse Matrix Vector Product

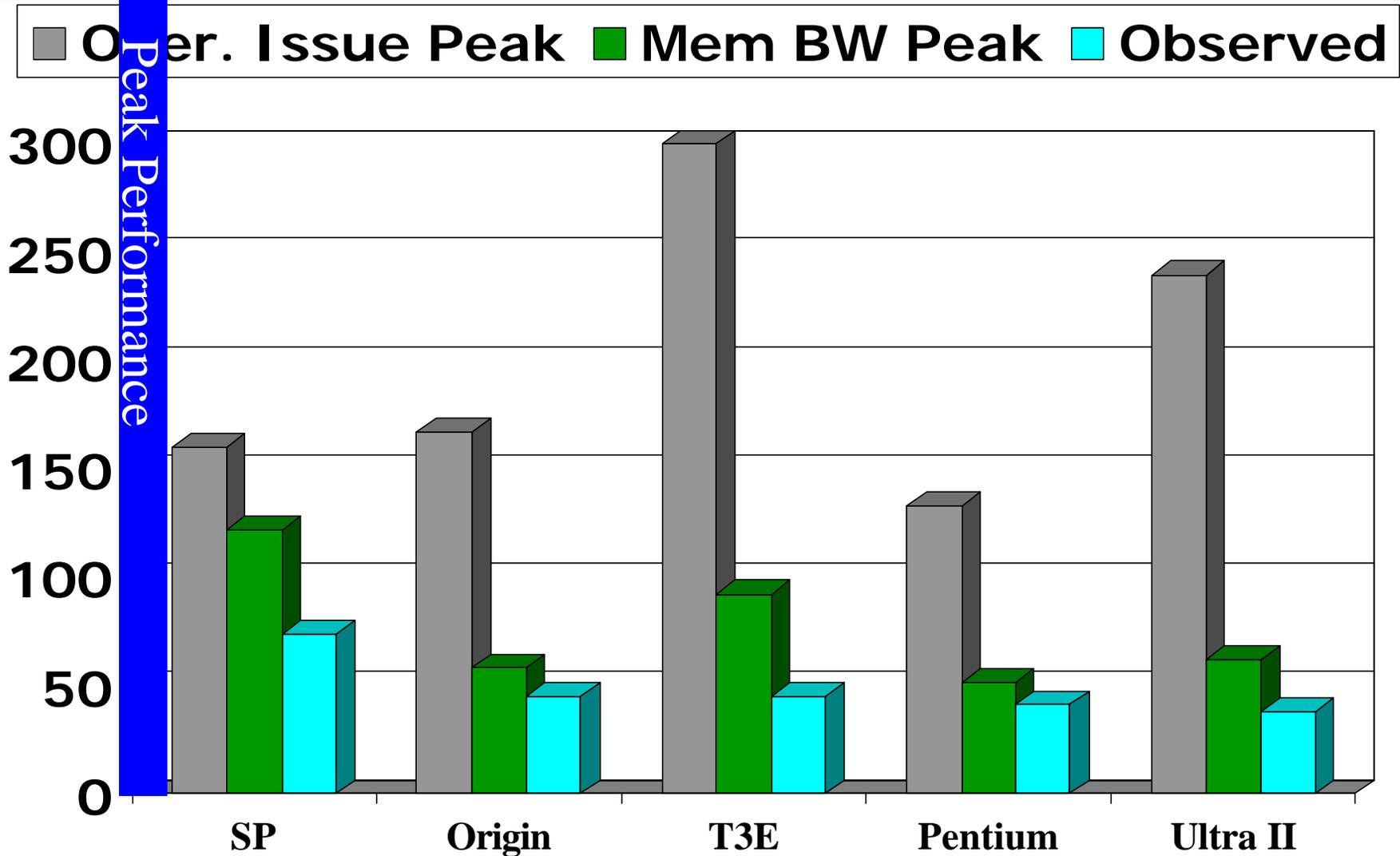
one vector, matrix size, $m = 90,708$, nonzero entries $nz = 5,047,120$



Experimental Performance

Sparse Matrix Vector Product

one vector, matrix size, $m = 90,708$, nonzero entries $nz = 5,047,120$

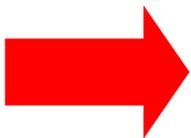


Implications

- Memory traffic is rate-limiting in sparse matrices
 - ◆ Reuse data items
 - ◆ Reuse items in cache
 - ◆ Avoid jumping far in memory (TLB misses)
- Reducing the number of non-floating-point instructions is also important
 - ◆ Reuse items in registers (reduce loads, address computation)

Forming an Accurate Complexity Model

- If the problem does not fit in cache:
 - ◆ Loads/Stores can dominate
 - ◆ Compute memory bound assuming “perfect” cache for smaller data items
 - ◆ Use “Stream” numbers for memory bandwidth
- If individual domains fit in cache:
 - ◆ Still must consider cost of loading domains into cache
 - ◆ Count total operations, not just flops
 - L1 cache usually small
 - L2, L3 cache take more cycles to access (weight operations accordingly)
 - ◆ Time estimate is based on worst of the limits imposed by number of flops, memory references, and total instruction count
 - ◆ Superlinear speedup is possible by staying within a faster memory level



Uniprocessor Memory Performance

- AlphaServer 8200 read latencies (3.33ns clock)

Memory Level	Latency		Bandwidth GB/sec
	ns	cycles	
Cache	6.7	2	4.8
L2 Cache	20	6	4.8
L3 Cache	26	8	0.96
Main	253	76	1.2
DRAM	60	18	.03-.1

- Note that $a[i] = b[i] * c[i]$ requires 7.2 GB/sec to keep processor fully busy

Parallel Processor Memory Performance

- Average read latency

CPUs MHz	AlphaServer 300		Origin2000 195	
	ns	cycles	ns	cycles
1	176	53		
2	190	57	313	61
4	220	66	405	79
8	299	117	528	103
16			641	125
32			710	138
64			796	155
128			903	176

... and worse (cluster and cluster-like scalable systems)

Conclusions

- Using multivectors can improve the performance of sparse matrix-vector product significantly
- “Algorithmic headroom” is available for modest blocking
- Simple models predict the performance of sparse matrix-vector operations on a variety of platforms, including the effects of **memory bandwidth**, and **instruction issue rates**
 - ♦ achievable performance is a small fraction of stated peak for sparse matrix-vector kernels, independent of code quality
 - ♦ compiler improvements and intelligent prefetching can help but the problem is fundamentally an architecture-algorithm mismatch and needs an algorithmic solution

Future Directions

- Design better data structures and implementation strategies for sparse matrix vector and related operations
- Integrate understanding of the performance issues with developments in block-structured algorithms to produce linear and nonlinear solvers that achieve a higher fraction of peak performance on a per-node basis
- Look at important special cases in hierarchical algorithms where performance modeling suggests alternative data structures and algorithmic directions

Relevant URLs

- PETSc-FUN3D Project at Argonne

<http://www.mcs.anl.gov/petsc-fun3d>

- PETSc

<http://www.mcs.anl.gov/petsc>

- ODU NSF and ASCI projects

<http://www.math.odu.edu/~keyes/nsf>

<http://www.math.odu.edu/~keyes/asci>