Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms

Samuel Williams*†, Leonid Oliker*, Richard Vuduc§, John Shalf*, Katherine Yelick*†, James Demmel†
*CRD/NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
†Computer Science Division, University of California at Berkeley, Berkeley, CA 94720, USA
§CASC, Lawrence Livermore National Laboratory, Livermore, CA 94551, USA

ABSTRACT

We are witnessing a dramatic change in computer architecture due to the multicore paradigm shift, as every electronic device from cell phones to supercomputers confronts parallelism of unprecedented scale. To fully unleash the potential of these systems, the HPC community must develop multicore specific optimization methodologies for important scientific computations. In this work, we examine sparse matrix-vector multiply (SpMV) – one of the most heavily used kernels in scientific computing – across a broad spectrum of multicore designs. Our experimental platform includes the homogeneous AMD dual-core and Intel quad-core designs, the heterogeneous STI Cell, as well as the first scientific study of the highly multithreaded Sun Niagara2. We present several optimization strategies especially effective for the multicore environment, and demonstrate significant performance improvements compared to existing state-of-the-art serial and parallel SpMV implementations. Additionally, we present key insights into the architectural tradeoffs of leading multicore design strategies, in the context of demanding memory-bound numerical algorithms.

1. INTRODUCTION

Industry has moved to chip multiprocessor (CMP) system design in order to better manage trade-offs among performance, energy efficiency, and reliability [5, 9]. However, the diversity of CMP solutions raises difficult questions about how different designs compare, for which applications each design is best-suited, and how to implement software to best utilize CMP resources.

In this paper, we consider these issues in the design and implementation of sparse matrix-vector multiply (SpMV) on several leading CMP systems. SpMV is a frequent bottleneck in scientific computing applications, and is notorious for sustaining low fractions of peak processor performance. We implement SpMV for one of the most diverse sets of CMP platforms studied in the existing HPC literature, including the homogeneous multicore designs of the dual-socket × dual-core AMD Opteron X2 and the dual-socket × quad-core Intel Clovertown, the heterogeneous local-store based architecture of the STI Cell single-socket PlayStation 3 (PS3) and dual-socket QS20 Cell Blade (containing six and eight cores per socket respectively), as well as the first scientific study of the hardware-multithreaded single-socket × eight-core Sun Niagara2. We show that our SpMV optimization strategies — explicitly programmed and tuned for these multicore environments — attain significant performance improvements compared to existing state-of-the-art serial and parallel SpMV implementations [20].

Additionally, we present key insights into the architectural tradeoffs of leading multicore design strategies, and their implications for SpMV. For instance, we quantify the extent to which memory bandwidth may become a significant bottleneck as core counts increase, motivating several algorithmic memory bandwidth reduction techniques for SpMV. We also find that using multiple cores provides considerably higher speedups than single-core code and data structure transformations alone. This observation implies that — as core counts increase — CMP system design should emphasize bandwidth and latency tolerance over single-core performance. Finally, we show that, in spite of relatively slow double-precision arithmetic, the STI Cell provides significant advantages in terms of absolute performance and power-efficiency, compared with the other multicore architectures in our test suite.

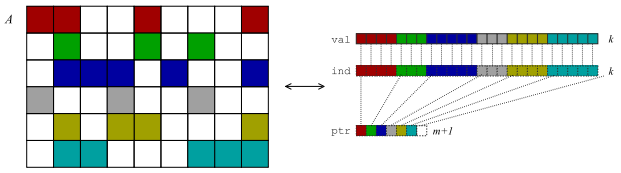
SC07 November 10-16, 2007, Reno, Nevada, USA
(c) 2007 ACM 978-1-59593-764-3/07/0011 ...$5.00.

2. SPMV OVERVIEW

SpMV dominates the performance of diverse applications in scientific and engineering computing, economic modeling and information retrieval; yet, conventional implementations have historically been relatively poor, running at 10% or less of machine peak on single-core cache-based microprocessor systems [20]. Compared to dense linear algebra kernels, sparse kernels suffer from higher instruction and storage overheads per flop, as well as indirect and irregular memory access patterns. Achieving higher performance on these platforms requires choosing a compact data structure and code transformations that best exploit properties of both the sparse matrix — which may be known only at run-time — *and* the underlying machine architecture. This need for optimization and tuning at run-time is a major distinction from the dense case.

We consider the SpMV operation $y \leftarrow y + Ax$, where A is a sparse matrix, and x, y are dense vectors. We refer to x as the *source vector* and y as the *destination vector*. Algorithmically, the SpMV kernel is as follows, (i, j): $\forall a_{i,j} \neq 0 : y_i \leftarrow y_i + a_{i,j} \cdot x_j$, where $a_{i,j}$ denotes an element of A. SpMV has a low computational intensity, given that $a_{i,j}$ is touched exactly once, and on cache-based machines, one can only expect to reuse elements of x and y. If x and y are maximally reused (*i.e.*, incur compulsory misses only), then reading A should dominate the time to execute SpMV. Thus, we seek data structures for A that are small and enable temporal reuse of x and y.

The most common data structure used to store a sparse matrix for SpMV-heavy computations is compressed sparse row (CSR) for-




```
// Basic SpMV implementation,
// y <- y + A*x, where A is in CSR.
 for (i = 0; i < m; ++i) {
   double y0 = y[i];
   for (k = ptr[i]; k < ptr[i+1]; ++k)
     y0 += val[k] * x[ind[k]];
   y[i] = y0;
 }
```

Figure 1: Compressed sparse row (CSR) storage, and a basic CSR-based SpMV implementation.

mat, illustrated in Figure 1. The elements of each row of A are shaded using the same color. Each row of A is packed one after the other in a dense array, `val`, along with a corresponding integer array, `ind`, that stores the column indices of each stored element. Thus, every non-zero value carries a storage overhead of one additional integer. A third integer array, `ptr`, keeps track of where each row starts in `val`, `ind`. We show a naïve implementation of SpMV for CSR storage in the bottom of Figure 1. This implementation enumerates the stored elements of A by streaming both `ind` and `val` with unit-stride, and loads and stores each element of y only once. However, x is accessed indirectly, and unless we can inspect `ind` at run-time, it is difficult or impossible to reuse the elements of x explicitly.

When the matrix has dense block substructure, as is common finite element method-based applications, practitioners employ a blocked variant of CSR (BCSR), which stores a single column index for each $r \times c$ block, rather than one per non-zero as in CSR.

2.1 OSKI, OSKI-PETSc, and Related Work

We compare our multicore SpMV optimizations against OSKI, a state-of-the-art collection of low-level primitives that provide automatically tuned computational kernels on sparse matrices [20]. The motivation for OSKI is that a non-obvious choice of data structure can yield the most efficient implementations due to the complex behavior of performance on modern machines. OSKI hides the complexity of making this choice, using techniques extensively documented in the SPARSITY sparse-kernel automatic-tuning framework [10]. These techniques include register- and cache-level blocking, exploiting symmetry, multiple vectors, variable block and diagonal structures, and locality-enhancing reordering.

OSKI is a serial library, but is being integrated into higher-level parallel linear solver libraries, including PETSc [3] and Trilinos [24]. This paper compares our multicore implementations against a generic "off-the-shelf" approach that uses our own experimental implementation of PETSc's distributed memory SpMV in which we replace the serial SpMV with calls to OSKI (referred to as OSKI-PETSc). We use MPICH 1.2.7p1 configured to use the shared-memory (ch_shmem) device, where message passing is replaced with memory copying. PETSc's default SpMV uses a block-row partitioning with equal numbers of rows per process.

The literature on optimization and tuning of SpMV is extensive. A number consider techniques that compress the data structure by recognizing patterns in order to eliminate the integer index overhead. These patterns include blocks [10], variable or mixtures of differently-sized blocks [6] diagonals, which may be especially well-suited to machines with SIMD and vector units [19], dense subtriangles arising in sparse triangular solve [22], and symmetry [11], and combinations.

Others have considered improving spatial and temporal locality by rectangular cache blocking [10], diagonal cache blocking [16], and reordering the rows and columns of the matrix. Besides classical bandwidth-reducing reordering techniques [15], recent work has proposed sophisticated 2-D partitioning schemes with theoretical guarantees on communication volume [18], and traveling salesman-based reordering to *create* dense block substructure [14].

Higher-level kernels and solvers provide opportunities to reuse the matrix itself, in contrast to non-symmetric SpMV. Such kernels include block kernels and solvers that multiply the matrix by multiple dense vectors [10], $A^T Ax$ [21], and matrix powers [19]. Better low-level tuning of the kind proposed in this paper, even applied to just a CSR SpMV, are also possible. Recent work on low-level tuning of SpMV by unroll-and-jam [12], software pipelining [6], and prefetching [17] influence our work. See [19] for an extensive overview of SPMV optimization techniques.

3. EXPERIMENTAL TESTBED

Our work examines several leading CMP system designs in the context of the demanding SpMV algorithm. In this section, we briefly describe the evaluated systems, each with its own set of architectural features: the dual-socket × dual-core AMD Opteron X2; the dual-socket × quad-core Intel Clovertown; the single-socket × eight-core hardware-multithreaded Sun Niagara2; and the heterogeneous STI Cell processor configured both as the single-socket six-core (six SPEs) PS3 as well as the dual-socket × eight-core Cell blade. Overviews of the architectural configurations and characteristics appear in Table 1 and Figure 2. Note that the sustained system power data were gathered by actual measurements via a digital power meter. Additionally, we present an overview of the evaluated sparse matrix suite.

3.1 AMD X2 Dual-core Opteron

The Opteron 2214 is AMD's current dual-core processor offering. Each core operates at 2.2 GHz, can fetch and decode three x86 instructions per cycle, and execute 6 micro-ops per cycle. The cores support 128b SSE instructions in a half-pumped fashion, with a single 64b multiplier datapath and a 64b adder datapath, thus requiring two cycles to execute a SSE packed double-precision floating point multiply. The peak double-precision floating point performance is therefore 4.4 GFlop/s per core or 8.8 GFlop/s per socket.

The Opteron contains a 64KB 2-way L1 cache, and a 1MB 4-way victim cache; victim caches are not shared among cores, but are cache coherent. All hardware prefetched data is placed in the victim cache of the requesting core, whereas all software prefetched data is placed directly into the L1. Each socket includes its own dual-channel DDR2-667 memory controller as well as a single cache-coherent HyperTransport (HT) link to access the other sockets cache and memory. Each socket can thus deliver 10.6 GB/s, for an aggregate NUMA (non-uniform memory access) memory bandwidth of 21.3 GB/s for the dual-core, dual-socket SunFire X2200 M2 examined in our study.

3.2 Intel Quad-core Clovertown

Clovertown is Intel's foray into the quad-core arena. Reminiscent of their original dual-core designs, two dual-core Xeon chips are paired onto a single multi-chip module (MCM). Each core is based on Intel's Core2 microarchitecture (Woodcrest), running at 2.33 GHz, can fetch and decode four instructions per cycle, and can execute 6 micro-ops per cycle. There is both a 128b SSE adder

Core Architecture	AMD Opteron X2	Intel Clovertown	Sun Niagara2	STI Cell SPE
Type	super scalar out of order	super scalar out of order	MT dual issue*	SIMD dual issue
Clock (GHz)	2.2	2.3	1.4	3.2
L1 DCache	64KB	32KB	8KB	—
Local Store	—	—	—	256KB
DP flops/cycle	2	4	1	4/7
DP GFlop/s	4.4	9.33	1.4	1.83

System	Opteron X2	Clovertown	Niagara2	PS3	Blade
# Sockets	2	2	1	1	2
Cores/Socket	2	4	8	6(+1)	8(+1)
L2 cache	4MB *(1MB/core)*	16MB *(4MB/2cores)*	4MB *(shared)*	—	—
DP GFlop/s	17.6	74.7	11.2	11	29
DRAM Type	DDR2 667 MHz 2×128b	FBDIMM 667 MHz 4×64b	FBDIMM 667 MHz 4×128b	XDR 1.6 GHz 1×128b	XDR 1.6 GHz 2×128b
DRAM (read GB/s)	21.3	21.3	42.6	25.6	51.2
Ratio Flop:Byte	0.83	3.52	0.26	0.43	0.57
Max Socket Pwr (Watts)	190	160	84	<100	200
Sustained Sys Pwr (Watts)	230	330	350	195	285

Table 1: Architectural summary of AMD Opteron X2, Intel Clovertown, Sun Niagara2, and STI Cell multicore chips. Sustained power measured via digital power meter. *Each of the two thread groups may issue up to one instruction

(two 64b floating point adders) and a 128b SSE multiplier (two 64b multipliers), allowing each core to support 128b SSE instructions in a fully-pumped fashion. The peak double-precision performance per core is therefore 9.3 GFlop/s.

Each Clovertown core includes a 32KB, 8-way L1 cache, and each chip (two cores) has a shared 4MB, 16-way L2 cache. Each socket has a single front side bus (FSB) running at 1.33 GHz (delivering 10.66 GB/s) connected to the Blackford chipset. In our study, we evaluate the Dell PowerEdge 1950 dual-socket platform, which contains two MCMs with dual independent busses. Blackford provides the interface to four fully buffered DDR2-667 DRAM channels that can deliver an aggregate read memory bandwidth of 21.3 GB/s. Unlike the AMD X2, each core may activate all four channels, but will likely never attain the peak bandwidth. The full system has 16MB of L2 cache and 74.67 GFlop/s peak performance.

3.3 Sun Niagara2

The Sun UltraSparc T2 "Niagara2" eight-core processor presents an interesting departure from mainstream multicore chip design. Rather than depending on four-way superscalar execution, each of the 8 strictly in-order cores supports two groups of four hardware thread contexts (referred to as Chip MultiThreading or CMT) to provide a total of 64 simultaneous hardware threads per socket. Each core may issue up to one instruction per thread group assuming there is no resource conflict. The CMT approach is designed to tolerate instruction, cache, and DRAM latency through fine-grained multithreading, while consuming relatively little power: 84 Watts per socket at 1.4 GHz.

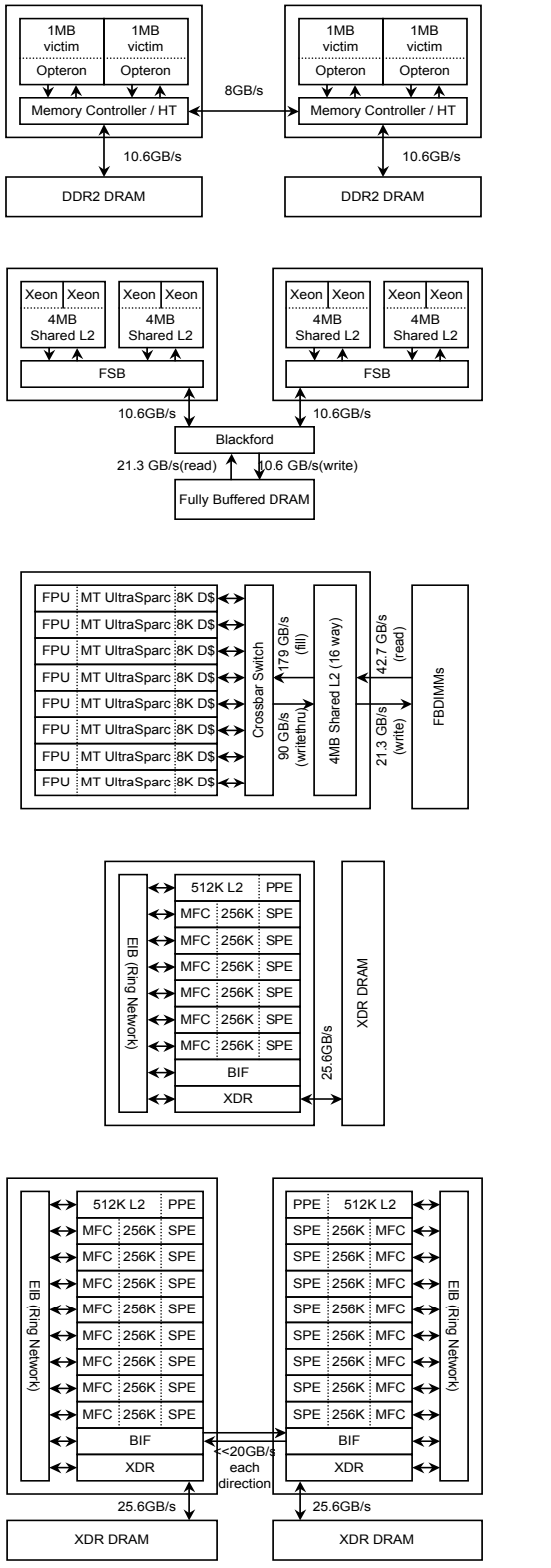


Figure 2: Architectural overview of (a) dual-socket × dual-core AMD Opteron X2 (b) dual-socket × quad-core Intel Clovertown, (c) single-socket × eight-core Sun Niagara2 (d) single-socket × six-core STI PS3 and (e) dual-socket × eight-core STI Cell.

Although the original Niagara only offered a single non-pipelined floating point unit that was shared by all of the processor cores, Niagara2 instantiates one FPU per core (shared among 8 threads). Our study examines the Sun UltraSparc T5120 with a single T2 processor operating at 1.4 GHz. It has a peak performance of 1.4 GFlop/s (no FMA) performance per core (11.2 GFlop/s per socket). Each core has access to its own private 8KB write-through L1 cache, but is connected to a shared 16-way set-associative L2 cache via a 179 GB/s(read) on-chip crossbar switch. The socket is fed by four dual channel 667 MHz FBDIMM memory controllers that deliver an aggregate bandwidth of 64 GB/s (42.6 GB/s for reads, and 21.3 GB/s for writes) to the L2. Niagara has no hardware prefetching and software prefetching only places data in the L2. Although multithreading may hide instruction and cache latency, it may not be able to fully hide DRAM latency.

3.4 STI Cell

The Sony Toshiba IBM (STI) Cell processor is the heart of the Sony PlayStation 3 (PS3) video game console, whose aggressive design is intended to meet the demanding computational requirements of video games. Cell adopts a heterogeneous approach to multicore, with one conventional processor core (Power Processing Element / PPE) to handle OS and control functions, combined with up to eight simpler SIMD cores (Synergistic Processing Elements / SPEs) for the computationally intensive work [7, 8]. The SPEs differ considerably from conventional core architectures due to their use of a disjoint software controlled local memory instead of the conventional hardware-managed cache hierarchy employed by the PPE. Rather than using prefetch to hide latency, the SPEs have efficient software-controlled DMA engines which asynchronously fetch data from DRAM into the 256KB local store. This approach allows more efficient use of available memory bandwidth than is possible with standard prefetch schemes on conventional cache hierarchies, but also makes the programming model more complex. In particular, the hardware provides enough concurrency to satisfy Little's Law [2] and conflict misses, while potentially eliminating write fills, however capacity misses must be handled in software.

Each SPE is a dual issue SIMD architecture: one slot can issue only computational instructions, whereas the other can only issue loads, stores, permutes and branches. This make instruction issue VLIW-like. Each core includes a half-pumped partially pipelined FPU. In effect, each SPE can execute one double-precision SIMD instruction every 7 cycles, for a peak of 1.8 GFlop/s per SPE — clearly far less than the AMD X2's 4.4 GFlop/s or the Xeon's 9.3 GFlop/s. In this study we examine two variants of the Cell processors: the PS3 containing a single socket with six SPEs (11 GFlop/s peak), and the QS20 Cell blade comprised of two sockets with eight SPEs each (29.2 GFlop/s peak). Each socket has its own dual channel XDR memory controller delivering 25.6 GB/s. The Cell blade connects the chips with a separate coherent interface delivering up to 20 GB/s; thus, like the AMD X2 system, the Cell blade is expected show strong variations in sustained bandwidth if NUMA is not properly exploited.

3.5 Evaluated Sparse Matrices

To evaluate the performance of our SpMV multicore platforms suite, we conduct experiments on 14 sparse matrices from a wide variety of actual applications, including finite element method-based modeling, circuit simulation, linear programming, a connectivity graph collected from a partial web crawl, as well as a dense matrix stored in sparse format. These matrices cover a range of properties relevant to SpMV performance, such as overall matrix dimension, non-zeros per row, the existence of dense block substructure, and degree of non-zero concentration near the diagonal. An overview of their salient characteristics appears in Figure 3.

spyplot	Name	Dimensions	Nonzeros (nnz/row)	Description
	Dense	2K x 2K	4.0M (2K)	Dense matrix in sparse format
	Protein	36K x 36K	4.3M (119)	Protein data bank 1HYS
	FEM / Spheres	83K x 83K	6.0M (72)	FEM concentric spheres
	FEM / Cantilever	62K x 62K	4.0M (65)	FEM cantilever
	Wind Tunnel	218K x 218K	11.6M (53)	Pressurized wind tunnel
	FEM / Harbor	47K x 47K	2.37M (50)	3D CFD of Charleston harbor
	QCD	49K x 49K	1.90M (39)	Quark propagators (QCD/LGT)
	FEM/Ship	141K x 141K	3.98M (28)	FEM Ship section/detail
	Economics	207K x 207K	1.27M (6)	Macroeconomic model
	Epidemiology	526K x 526K	2.1M (4)	2D Markov model of epidemic
	FEM / Accelerator	121K x 121K	2.62M (22)	Accelerator cavity design
	Circuit	171K x 171K	959K (6)	Motorola circuit simulation
	webbase	1M x 1M	3.1M (3)	Web connectivity matrix
	LP	4K x 1.1M	11.3M (2825)	Railways set cover Constraint matrix

Figure 3: Overview of sparse matrices used in evaluation study.

4. SPMV OPTIMIZATIONS

In this section, we discuss our considered set of SpMV tuning optimizations, in the order in which they are applied during autotuning. These tuning routines are designed to be effective on both conventional cache-based multicores as well as the Cell architecture; as a result, several optimizations were restricted to facilitate Cell programming. The complete set of optimizations can be classified into three areas: low-level code optimizations, data structure optimizations (including the requisite code transformations), and parallelization optimizations. The first two largely address single-

core performance, while the third examines techniques to maximize multicore performance in a both single- and multi-socket environments. We examine a wide variety of optimizations including software pipelining, branch elimination, SIMD intrinsics, pointer arithmetic, numerous prefetching approaches, register blocking, cache blocking, TLB blocking, block-coordinate (BCOO) storage, smaller indices, threading, row parallelization, NUMA-aware mapping, process affinity, and memory affinity. A summary of these optimizations appears in Table 2.

Most of these optimizations complement those available in OSKI. In particular, OSKI includes register blocking and single-level cache or TLB blocking, but not with reduced index sizes. OSKI also contains optimizations for symmetry, variable block size and splitting, and reordering optimizations, which we do not exploit in this paper (*e.g.*, we do not exploit symmetry in our experiments). Except for unrolling and use of pointer arithmetic, OSKI does not explicitly control low-level instruction scheduling and selection details, such as software pipelining, branch elimination, and SIMD intrinsics, relying instead on the compiler back-end. Finally, we apply the optimizations in a slightly different order to facilitate parallelization and Cell development.

Explicit low-level code optimizations, including SIMD intrinsics, can be very beneficial on each architecture. Applying these optimizations helps ensure that the cores are bound by memory bandwidth, rather than by instruction latency and throughput. Unfortunately, implementing them can be extremely tedious and time-consuming. We thus implemented a Perl-based code generator that produces the innermost SpMV kernels using the subset of optimizations appropriate for each underlying system. Our generators helped us explore the large optimization space effectively and productively — less than 500 lines of generators produced more than 30,000 lines of optimized kernels.

4.1 Thread Blocking

The first phase in the optimization process is exploiting *thread-level parallelism*. The matrix is partitioned into `NThreads` thread blocks, which may in turn be individually optimized. There are three approaches to partitioning the matrix: by row blocks, by column blocks, and into segments. An implementation could use any or all of these, *e.g.*, 3 row thread blocks each of 2 column thread blocks each of 2 thread segments. In both row and column parallelization, the matrix is explicitly blocked to exploit NUMA systems. To facilitate implementation on Cell, the granularity for partitioning is a cache line.

Our implementation attempts to statically load balance SpMV by approximately equally dividing the number of nonzeros among threads, as the transfer of this data accounts for the majority of time on matrices whose vector working sets fit in cache. It is possible to provide feedback and repartition the matrix to achieve better load balance, and a thread-based segmented scan could achieve some benefit without reblocking the matrix. In this paper, we only exploit *row partitioning*, as *column partitioning* showed little potential benefit; future work will examine segmented scan.

To ensure a fair comparison, we explored a variety of barrier implementations and utilized the fastest implementation on each architecture. For example, the emulated Pthreads barrier (with mutexes and broadcasts) scaled poorly beyond 32 threads and was therefore replaced with a simpler version where only thread 0 is capable of unlocking the other threads.

Finally, for the two NUMA architectures in our study, Cell and AMD X2, we apply *NUMA-aware* optimizations in which we explicitly assign each thread block to a specific core and node. To ensure that both the process and its associated thread block are mapped to a core (*process affinity*) and the DRAM (*memory affinity*) proximal to it, we used the NUMA routines that are most appropriate for a particular architecture and OS. The `libnuma` library performed well on the Cell, but showed poor behavior on the AMD X2; we thus implemented AMD X2 affinity via the Linux scheduler. Native Solaris scheduling routines were used on the Niagara2. We highlight that benchmarking with the OS schedulers must be handled carefully, as the mapping between processor ID and physical ID is unique to each machine. For instance, the socket ID is specified by the least-significant bit on the evaluated Clovertown system and the most-significant bit on the AMD X2.

Note that our set of SpMV optimizations, discussed in the subsequent subsections, are performed on thread blocks individually.

4.2 Cache and Local Store Blocking

For sufficiently large matrices, it is not possible to keep the source and destination vectors in cache, potentially causing numerous capacity misses. Prior work shows that explicitly *cache blocking* the nonzeros into tiles ($\approx$ 1K $\times$ 1K) can improve SpMV performance [10,13]. We extend this idea by accounting for cache utilization, rather than only spanning a fixed number of columns. Specifically, we first quantify the number of cache lines available for blocking, and span enough columns such that the number of source vector cache lines touched is equal to those available for cache blocking. Using this approach allows each cache block to touch the same number of cache lines, even though they span vastly different numbers of columns. We refer to this optimization as *sparse cache blocking*, in contrast to the classical dense cache blocking approach. Unlike OSKI, where the block size must be specified or searched for, we use a heuristic to cache block.

Given the total number of cache lines available, we specify one of three possible blocking strategies: block neither the source nor the destination vector, block only the source vector, or block both the source and destination vectors. In the second case, all cache lines can be allocated to blocking the source vector. However, when implementing the third strategy, the cache needs to store the potential row pointers as well as the source and destination vectors; we thus allocated 40%, 40%, and 20% of the cache lines to the source, destination, and row pointers respectively.

Although these three options are explored for cache-based systems, the Cell architecture always requires cache blocking (*i.e.*, blocking for the local store) for both the source and destination vector — this Cell variant requires *cache blocking for DMA*. To facilitate the process, a DMA list of all source vector cache lines touched in a cache block is created.

This list is used to gather the stanzas of the source vector and pack them contiguously in the local store via the DMA `get list` command. This command has 2 constraints: each stanza must be less than 16KB, and there can be no more than 2K stanzas. Thus, the sparsest possible cache block cannot touch more than 2K cache lines, and in the densest case, multiple contiguous stanzas are required. (We unify the code base of the cache-based and Cell implementations by treating cache-based machines as having large numbers of stanzas and cache lines.) Finally, since the DMA get list packs stanzas together into the disjoint address space of the local store, the addresses of the source vector block when accessed from an SPE are different than when accessed from DRAM. As a results, the column indices must be re-encoded to be local store relative.

4.3 TLB Blocking

Prior work showed that TLB misses can vary by an order of magnitude depending on the blocking strategy, highlighting the importance of *TLB blocking* [13]. Thus, on the three evaluation platforms

Code Optimization	x86	N2	Cell
SIMDization	✓	N/A	✓
Software Pipelining			✓
Branchless	✓[10]		✓
Pointer Arithmetic		✓[10]	
PF/DMA[1] Values & Indices	✓	✓	✓
PF/DMA[1] Pointers/Vectors			✓
inter-SpMV data structure caching	✓	✓	

Data Structure Optimization	x86	N2	Cell
BCSR	✓	✓	
BCOO	✓	✓	✓
16-bit indices	✓	✓	✓
32-bit indices	✓	✓	N/A
Register Blocking	✓	✓	✓[9]
Cache Blocking	✓[2]	✓[2]	✓[3]
TLB Blocking	✓	✓	

Parallelization Optimization	x86	N2	Cell
Row Threading	✓[4]	✓[4]	✓[5]
Process Affinity	✓[6]	✓[8]	✓[7]
Memory Affinity	✓[6]	N/A	✓[7]

Table 2: Overview of SpMV optimizations attempted in our study for the x86 (AMD X2 and Clovertown), Niagara2, and Cell architectures. Notes: [1]PF/DMA (Prefetching or Direct Memory Access), [2]sparse cache blocking, [3]sparse cache blocking for DMA, [4]Pthreads, [5]libspe 2.0, [6]via Linux scheduler, [7]via libnuma, [8]via Solaris bind, [9]2x1 and larger, [10]implemented but resulted in no significant speedup.

running Linux with 4KB pages for heap allocation (Niagara2 running Solaris uses 256MB pages), we heuristically limit the number of source vector cache lines touched in a cache block by the number of unique pages touched by the source vector. Rather than searching for the TLB block size, which would require costly re-encoding of the matrix, we explore two tuning settings: no TLB blocking and using the heuristic. Note that for the AMD X2 we found it beneficial to block for the L1 TLB (each cache block is limited to touching 32 4KB pages of the source vector).

4.4 Register Blocking and Format Selection

The next phase of our implementation is to optimize the SpMV data format. For memory-bound multicore applications, we believe that minimizing the memory footprint is more effective than improving *single* thread performance. Indeed, technology trends indicate that it is easier and more cost effective to double the number of cores rather than double the DRAM bandwidth [1]; thus we expect future multicore codes to become increasingly memory bound. In a naïve coordinate approach, 16 bytes of storage are required for each matrix nonzero: 8 bytes for the double-precision nonzero, plus 4 bytes each for row and column coordinates. Our data structure transformations can, for some matrices, cut these storage requirements nearly in half. A possible future optimization, exploiting symmetry, could cut the storage in half again.

Register blocking groups adjacent nonzeros into rectangular tiles, with only one coordinate index per tile [10]. Since not every value has an adjacent nonzero, it is possible to store zeros explicitly in the hope that the 8 byte deficit is offset by index storage savings on many other tiles. For this paper we limit ourselves to power-of-two block sizes up to 8×8 (and on Cell, 8×2), to facilitate *SIMDization*, minimize register pressure, and allow a footprint minimizing heuristic to be applied. Instead of searching for the best register block, we implement a two-step heuristic. First, we register block each cache block using 8×8 tiles. Then, for each cache block, we inspect each 8×8 tile hierarchically in powers of two and determine how many tiles are required for each $r \times c$ power-of-two blocking, (*i.e.*, we count the number of tiles for 8×4 tiles, then for 8×2 tiles, then for 8×1 tiles, and so on).

For cache block encoding, we consider *block coordinate* (BCOO) and *block compressed sparse row* (BCSR; Section 2) formats. BCSR requires a pointer for every register blocked row in the cache block and a single column index for each register block, whereas BCOO requires 2 indices (row and column) for each register block, but no row pointers. The BCSR can be further accelerated either by specifying the first and last non-empty row, or by using a generalized BCSR format that stores only non-empty rows while associating explicit indices with either each row or with groups of consecutive rows (both available in OSKI). We also select the appropriate $r \times c \times$ format combination that minimizes each cache block's footprint. Thus, each cache block (not just each thread) may have a unique encoding.

As Cell is a SIMD architecture, it is expensive to implement 1x1 blocking. In addition, since Cell streams nonzeros into buffers [25], it is far easier to implement BCOO than BCSR. Thus to maximize our productivity, for each architecture we specify the minimum and maximum r and c, as well as a mask that enables each format. On Cell, this combination of configuration variables requires block sizes of at least 2×1, which will tend to increase memory traffic requirements and thereby potentially degrade Cell performance, while facilitating productivity.

4.5 Index Size Selection

As a cache block may span fewer than 64K columns, it is possible to use *memory efficient indices* for the columns (and rows) using 16b integers. Doing so provides a 20% reduction in memory traffic for some matrices, which could translate up to a 20% increase in performance. On Cell, no more than 32K unique doubles can reside in the local store at any one time, and unlike caches, this address space is disjoint and contiguous. Thus, on Cell we can always use 16b indices, even if the entire matrix in DRAM spans 1 million columns. This is the first advantage our implementation conferred on the Cell. Note that our technique is less general but simpler than a recent index compression approach [23].

4.6 Architecture Specific Kernels

The SpMV multithreaded framework is designed to be modular. Each architecture specifies which of the previously described optimizations are implemented in a configuration file. The optimization routines then block the matrix accordingly and the execution routines take this information to call the corresponding kernels. To maximize productivity, all kernels as well as the unique configuration file for each architecture are generated from a single Perl script. Each kernel implements a sparse cache block matrix multiplication for the given encoding (register blocking, index size, format). There is no restriction on how this functionality is implemented, thus the generator is custom tailored for each architecture.

4.7 SIMDization

The Cell SPE ISA is rather restrictive compared to conventional RISC instruction sets. All operations are on 128 bits (quadword) of data, all loads are of 16 bytes and must be 16 byte aligned. There are no double, word, half or byte loads. When a scalar is required it must be moved from its position within the quadword to a so-called preferred slot. Thus, the number of instructions required to implement a given kernel on Cell far exceeds that of other architectures, despite the computational advantage of SIMD. To overcome certain

limitations in the compiler's (xlc's) code generation, our Cell kernel generator explicitly produces *SIMDization* intrinsics. The xlc static timing analyzer provides information on whether cycles are spent in instruction issue, double-precision issue stalls, or stalls for data hazards, thus simplifying the process of kernel optimization.

In addition, we explicitly implement SSE instructions on the x86 kernels. Use of SSE made no performance difference on the AMD X2, but resulted in significant improvements on the Clovertown. However, when the optimal gcc tuning options are applied, SSE optimized code was only slightly faster than straight C code. Programs produced by the Intel compiler (icc 9.1) saw little benefit from SSE instructions or compiler tuning options.

4.8 Loop Optimizations

A conventional CSR-based SpMV (Figure 1) consists of a nested loop, where the outer loop iterates across all rows and the inner loop iterates across the nonzeros of each row via a start and end pointer. However, the CSR data storage is such that the end of one row is immediately followed by the beginning of the next, meaning that the column and value arrays are accessed in a streaming (unit-stride) fashion. Thus, it is possible to simplify the loop structure by iterating from the first nonzero to the last; although this approach still requires a nested loop, it includes only a single loop variable and often results in higher performance.

Additionally, since our format uses a single loop variable coupled with nonzeros that are processed in-order, we can explicitly *software pipeline* the code to hide any further instruction latency. In our experience, this technique is useful on in-order architectures like Cell, but is of little value on the out-of-order superscalars.

Finally, the code can be further optimized using a *branchless implementation*, which is in effect a segmented scan of vector-length equal to one [4]. On Cell, we implement this technique using the `select bits` instruction. A branchless BCOO implementation simply requires resetting the running sum (selecting between the last sum or next value of Y). Once again, we attempted this branchless implementations via SSE, `cmov`, and `jumps`, but without Cell's xlc static timing analyzer, could not determine why no performance improvement where seen.

4.9 Software Prefetching

We also consider *explicit prefetching*, using our code generator to tune the prefetch distance from 0 (no prefetching) to 1024 bytes. The x86 architectures rely on hardware prefetchers to overcome memory latency, but prefetched data is placed in the L2 cache, so L2 latency must still be hidden. Although Clovertown has a hardware prefetcher to transfer data from the L2 to the L1, software prefetch via intrinsics provides an effective way of not only placing data directly into the L1 cache, but also tagging it with the appropriate temporal locality. Doing so reduces L2 cache pollution, since nonzero values or indices that are no longer useful will be evicted. The Niagara2 platform, on the other hand, supports prefetch but only into the L2 cache. As a result the L2 latency can only be hidden via multithreading. Despite this, Niagara2 still showed benefits from software prefetching.

4.10 Auto-tuning Framework

For each threading model, we implement an auto-tuning framework that can incorporate the architecture-optimized kernels. For parallel computations, we attempt three cases: no cache and no TLB blocking, cache blocking with no TLB blocking, as well as cache and TLB blocking. Within each of these, heuristics optimize the appropriate block size, register blocking, format, and index size. Additionally, we exhaustively search for the best prefetch distance on each architecture (this process is relatively fast as it does not require data structure changes). The Cell version does not require a search for the optimal prefetch distance searching due to the fixed size of the DMA buffer. We report the peak performance for each tuning option.

5. SPMV PERFORMANCE RESULTS

In this section, we present SpMV performance on our sparse matrices and multicore systems. We compare our implementations to serial OSKI and parallel (MPI) PETSc with OSKI. PETSc was run with up to 8 tasks, but we only present the fastest results for the case where fewer tasks achieved higher performance. OSKI was compiled with both `gcc` and `icc`, with the best results shown. For our SpMV code we used `gcc 4.1.2` on AMD X2 and Clovertown (`icc` was no faster), `gcc 4.0.4` on Niagara2, and `xlc` on the Cell. We start with a discussion of the ramifications of sparse matrix structure, followed by a detailed analysis of the dense matrix case (the easiest to understand), and finally a broad discussion of the full matrix suite.

For clarity, we present the performance of each optimization condensed into a stacked bar format as seen in Figure 4. Each bar segment corresponds to an individual trial rather than to components of a total. In addition, we provide median performance (half perform better/worse) our matrix set for each optimization. Readers interested in a specific area (*e.g.*, finite element meshes or linear programming) should focus on those matrices rather than median.

5.1 Performance Impact of Matrix Structure

Before presenting experimental results, we first explore the structure and characteristics of several matrices in our test suite, and consider their expected effect on runtime performance. In particular, we examine the impact that few nonzero entries per row have on the CSR format and the flop:byte ratio (the upper limit is 0.25, two flops for eight bytes), as well as the ramifications of cache blocking on certain types of matrices.

Note that all of our CSR implementations use a nested loop structure. As such, matrices with few nonzeros per row (inner loop length) cannot amortize the loop startup overhead. This cost, including a potential branch mispredict, can be more expensive than processing a nonzero tile. Thus, even if the source/destination vectors fit in cache, we expect matrices like Webbase, Epidemiology, Circuit, and Economics to perform poorly across all architectures. Since the Cell version successfully implements a branchless BCOO format it will not suffer from the loop overhead problem, but may suffer from poor register blocking and an inherent low flop:byte ratio,

Matrices like Epidemiology, although structurally nearly diagonal, have very large vectors. As such, those vectors cannot reside in cache, and thus suffer capacity misses. Assuming a cache line fill is required on a write miss, the destination vector generates 16 bytes of traffic per element. Thus, the Epidemiology matrix has a flop:byte ratio of about $2 * 2.1M/(12 * 2.1M + 8 * 526K + 16 * 526K)$ or 0.11. Given the AMD X2's and Clovertown's peak sustained memory bandwidths are 13.35 GB/s and 11.24 GB/s respectively, we do not expect the performance of Epidemiology to exceed 1.47 GFlop/s and 1.24 GFlop/s (respectively), regardless of CSR performance. The results of Figure 4 (discussed in detail in Section 5) confirm this prediction.

Aside from Cell, we flush neither the matrix nor the vectors from the cache between SpMVs. Thus, matrices such as QCD and Economics, having fewer than 2M nonzeros and a footprint as little as 10 bytes per nonzero, may nearly fit in the Clovertown's collective 16MB cache. If all lines in a given set are tagged with the

Machine	Sustained Memory Bandwidth in GB/s (% of configuration peak bandwidth) Sustained Performance in GFlop/s (% of configuration peak computation) one socket, one core, one thread	one socket, one core, all threads	one socket, all cores, all threads	all sockets, all cores, all threads
AMD X2	$5.24\,GB/s$ (49.2%) $1.31\,GF/s$ (29.7%)	$5.24\,GB/s$ (49.2%) $1.31\,GF/s$ (29.7%)	$6.73\,GB/s$ (63.0%) $1.68\,GF/s$ (19.1%)	$13.35\,GB/s$ (62.6%) $3.31\,GF/s$ (18.8%)
Clovertown	$3.82\,GB/s$ (35.8%) $0.95\,GF/s$ (10.2%)	$3.82\,GB/s$ (35.8%) $0.95\,GF/s$ (10.2%)	$5.37\,GB/s$ (50.3%) $1.33\,GF/s$ (3.6%)	$11.24\,GB/s$ (52.7%) $2.79\,GF/s$ (3.7%)
Niagara2	$0.66\,GB/s$ (1.5%) $0.16\,GF/s$ (11.7%)	$3.79\,GB/s$ (8.9%) $0.94\,GF/s$ (67.3%)	$23.28\,GB/s$ (54.6%) $5.80\,GF/s$ (51.8%)	$23.28\,GB/s$ (54.6%) $5.80\,GF/s$ (51.8%)
Cell(PS3)	$4.76\,GB/s$ (18.6%) $1.15\,GF/s$ (62.9%)	$4.76\,GB/s$ (18.6%) $1.15\,GF/s$ (62.9%)	$21.16\,GB/s$ (82.6%) $5.12\,GF/s$ (46.6%)	$21.16\,GB/s$ (82.6%) $5.12\,GF/s$ (46.6%)
Cell(Blade)	$4.75\,GB/s$ (18.6%) $1.15\,GF/s$ (62.9%)	$4.75\,GB/s$ (18.6%) $1.15\,GF/s$ (62.9%)	$24.73\,GB/s$ (96.6%) $5.96\,GF/s$ (40.7%)	$47.29\,GB/s$ (92.4%) $11.35\,GF/s$ (38.8%)

Table 3: Sustained bandwidth and computational rate for a dense matrix stored in sparse format, in GB/s (and percentage of configuration's peak bandwidth) and GFlop/s (and percentage of configuration's peak performance).

non-temporal hint, cache replacement will be nearly random. Thus, unlike the least-recently-used (LRU) cache replacement policy, it is possible that a cache line may remain resident when it is required on a subsequent iteration. Thus, we expect to see superlinear benefits for these matrix patterns, even for matrices that barely do not fit in the Clovertown cache.

Finally, we examine the class of matrices represented by Linear Programming (LP). LP is very large, containing (on average) nearly three thousand nonzeros per row; however, this does not necessarily assure high performance. Upon closer examination, we see that LP has a dramatic aspect ratio with over a million columns, for only four thousand rows, and is structured in a highly irregular fashion. As a result each processor must maintain a large working set of the source vector (between 6MB–8MB). Since no single core, or even pair of cores, in our study has this much available cache, it is logical to conclude that performance will suffer greatly due to source vector cache misses. On the other hand, this matrix structure is amenable to effective cache and TLB blocking as there are plenty of nonzeros per row. As a result, LP should benefit from cache blocking on both the AMD X2 and Clovertown. Figure 4 confirms this prediction. Note that this is the only matrix that showed any benefit for column threading; thus we only implemented cache and TLB blocking as it improves LP as well as most other matrices.

5.2 Peak Effective Bandwidth

On any balanced modern machine, SpMV should be limited by memory throughput, we thus start with the best case for the memory system, which is a dense matrix in sparse format. This dense matrix is likely to provide a performance upper bound, because it supports arbitrary register blocks without adding zeros, loops are long-running, and accesses to the source vector are contiguous and have high re-use. As the optimization routines result in a matrix storage format with a flop:byte ratio of nearly 0.25, one can easily compute best-case GFlop/s or GB/s from time. Since all the multicore systems in our study except Niagara2 have a flop:byte ratio greater than 0.25, we expect these platforms to be memory bound on this kernel — if the deliverable streaming bandwidth is close to the advertised peak bandwidth. A summary of the results for the dense matrix experiments are shown in Table 3.

Observe that the systems achieve a wide range of the available memory bandwidth, however, only the full version of the Cell (8 SPEs) comes close to fully saturating the socket bandwidth, utilizing an impressive 96% of the theoretical potential. This is due, in part, to the explicit local store architecture of the Cell, which allows double-buffered DMAs to hide the majority of memory latency. This high memory bandwidth utilization translates to high sustained performance, attaining over 11 GFlop/s (92% of theoretical bandwidth) on the dual-socket Cell blade. Results also show that the PS3 is actually not memory bound, as full socket (6 SPE) performance increases nearly linearly, but does not saturate the bandwidth.

Looking at the other end of the spectrum, the data in Table 3 shows that the Niagara2 system sustains only 1.5% of its memory bandwidth when utilizing a single thread on a single core. There are numerous reasons for this poor performance. Most obviously, Niagara2 has more single socket bandwidth than the other systems. Secondly, the Niagara2 architecture cannot deliver a 64-bit operand in less than 14 cycles (on average) for a single thread, as the L1 line size is only 16 bytes with a latency of three cycles, while the L2 latency is about 22 cycles. Since each SpMV nonzero requires two unit-stride accesses, and one indexed load, this results in between 23 and 48 cycles of memory latency per nonzero. When combined with at least an additional eight cycles for instruction execution, it becomes clear why a single thread on the Niagara2 strictly in-order cores only sustains between 50 and 90 Mflop/s for 1×1 CSR (with a sufficient number of nonzeros per row). With as much as 48 cycles of latency, and 11 instructions per thread, performance should scale well and consume the resources of a single thread group. As an additional thread group and cores are added, performance is expected to continue scaling. However, since the machine flop:byte ratio only slightly larger than the ratio in the best case (dense matrix), it is unlikely Niagara2 can saturate its available memory bandwidth for the SpMV computation. This departure from our memory-bound multicore assumption implies that search-based auto-tuning is likely necessary for Niagara2.

Although it is relatively easy to understand the performance of the Cell and Niagara2 in-order architectures, the behavior of the (out-of-order) superscalar AMD X2 and Clovertown are more difficult to predict. For instance, the full AMD X2 socket does not come close to saturating its available 10.6 GB/s bandwidth, even though a single core can use 5.24 GB/s. It is even less clear why the extremely powerful Clovertown core can only utilize 3.8 GB/s (36%) of its memory bandwidth, when the FSB can theoretically deliver 10.6 GB/s. It is interesting to note that the AMD X2 socket can utilize a slightly larger fraction of its memory bandwidth than the Clovertown MCM can of its FSB bandwidth. Performance re-

sults in Table 3 show that, for this memory bandwidth-limited application, the AMD X2 is 20% faster than a Clovertown for a full socket (even though the Clovertown socket peak flop rate is 4.2× higher than the AMD X2) As a sanity check, we also ran tests (not shown) on a small matrix amenable to register blocking that fit in the cache, and found that, as expected, the performance is very high — 23 GFlop/s on the Clovertown. Thus, performance is limited by the memory system but not by bandwidth per se, even though accesses are almost entirely unit-stride reads and are prefetched in both hardware and software.

5.3 AMD Opteron X2 Performance

Figure 4(top) presents SpMV performance of the AMD X2 platform, showing increasing degrees of single-core optimizations — naïve , prefetching (PF), register blocking (RB), and cache blocking (CB) – as well as fully-optimized parallel (Pthread) performance on both cores of a single socket, and finally full system performance (dual-socket × dual core). Additionally, comparative results are shown for both serial OSKI and parallel (MPI) OSKI-PETSc.

The effectiveness of AMD X2 optimization depends on the matrix structure. For example, the FEM-Ship matrix sees significant improvement from register blocking (due to its natural block structure) but little benefit from cache blocking, while the opposite effect holds true for the LP matrix. Generally, AMD X2 performance is tied closely with the optimized flop:byte ratio of the matrix, and suffers from short average inner loop lengths. The best performing matrices sustain a memory bandwidth of 10–13.4 GB/s, which corresponds to a high computational rate of 2–3 GFlop/s. Conversely, matrices with low flop:byte ratios show poor parallelization and cache behavior, sustaining a bandwidth significantly less than 8 GB/s, and thus achieving performance of only 0.5–1 GFlop/s.

Looking at overall SpMV behavior, serial results show that our optimizations speedup naïve runtime by a factor of 1.6× in the median case, while achieving about a 1.4× speedup (due, in part, to explicit prefetching) over the highly tuned OSKI library. For the parallel, multicore experiments, we find performance improvements of 1.5× (bus saturation) and 2.9× (multiple controllers) for the dual core and full system (dual-socket × dual-core) configurations, respectively, when compared to our optimized single-core results. More impressively, our full-system implementation runs 3.3× faster than the parallel full-system OSKI-PETSc experiments.

Note that OSKI-PETSc uses an "off-the-shelf" MPICH-shmem implementation, and generally yields only moderate (if any) speedups, due to at least two factors. First, communication time accounts on average for 30% of the total SpMV execution time and as much as 56% (LP matrix), likely due to explicit memory copies. Second, several matrices suffer from load imbalance due to the default equal-rows 1-D matrix distribution. For example, in FEM-Accel, one process has 40% of the total non-zeros in a 4-process run. It is possible to control the row distribution and improve the communication, but we leave deeper analysis and optimization of OSKI-PETSc to future work.

In summary, these results indicate tremendous potential in leveraging multicore resources — even for memory bandwidth-limited computations such as SpMV — if optimizations and programming methodologies are employed in the context of the available memory resources. Additionally, our data show that significantly higher performance improvements could be attained through multicore parallelizations, rather than serial code or data structure transformations. This an encouraging outcome since the number of cores per chip is expected to continue increasing rapidly, while core performance is predicted to stay relatively flat [1].

5.4 Intel Clovertown Performance

The quad-core Clovertown SpMV data appears in Figure 4 (second from top), showing both naïve and optimized serial performance, as well as parallel (Pthread) dual-core, quad-core, and full system (dual-core, quad-core) results. Serial OSKI and parallel (MPI) OSKI-PETSc results are also presented as a baseline for comparison. Unlike the AMD X2, Clovertown performance is more difficult to predict based on the matrix structure. The sustained bandwidth is generally less than 9 GB/s, but does not degrade as profoundly for the difficult matrices as on the AMD X2: no doubt due to the large (16MB) aggregate L2 cache and the larger total number of threads to utilize the available bandwidth. This cache effect can be clearly seen on the Economics matrix, which contains 1.3M nonzeros and requires less than 15MB of storage. As a result, superlinear (4×) improvements are seen when comparing a single-socket × quad-core with 8MB of L2 cache, against the dual-socket full system, which contains 16MB of L2 cache across the eight cores. Despite fitting in cache, the few nonzeros per row significantly limit performance — when compared to matrices that fit in cache, have large numbers of nonzeros per row, and exhibit good register blocking.

Examining the median Clovertown performance, shows that our single-core SpMV optimization resulted in only a 1.4× improvement compared with the naïve case. This is due, in part, to the Xeon's superior hardware prefetching capabilities compared with the Opteron, as there was only moderate benefit from software prefetching. Additionally, register blocking was useful on less than half of the matrices, while cache blocking held little benefit due to the large L2 cache of the Clovertown.

Surprisingly, only a paltry speedup of 7% is seen (between the optimized versions) when comparing one and two cores of the Clovertown system. Performance only increases slightly (an additional 21%) when four cores (within a single socket) are employed, since the two-core experiment usually attains a significant fraction of the sustainable FSB bandwidth. Examining the full system (dual-socket × quad-core), shows performance that exceeds the optimized serial case by only 2.2×; this was not surprising as the aggregate FSB bandwidth was doubled in this configuration. Finally, comparing results with the OSKI autotuner, we see a serial improvement of 1.7× and a parallel full-system speedup of 2.2× compared with OSKI and OSKI-PETSc respectively. These results highlight the effectiveness of our explicitly programmed, multicore-specific optimization and parallelization schemes.

5.5 Sun Niagara2 Performance

Figure 4 (third from top) presents SpMV performance of the Niagara2 system, showing increasing levels of optimizations for the single-threaded (single core) test case, as well as optimized performance using all eight threads per cores on increasing numbers of cores. The full system configuration utilizes eight cores and all eight hardware-supported CMT contexts. As discussed in Section 5.2, Niagara2's performance is bound by the interaction between the L1, L2, lack of L1 prefetching, and the strictly in-order cores rather than limits in DRAM bandwidth.

Results show that, as expected, single thread results are extremely poor, achieving only 75 Mflop/s for the median matrix in the naïve case, with about 10% speedup from our suite of optimizations (86 Mflop/s). Eight-way multithreading on a single core yielded a 5× increase in performance, and significant performance improvement is achieved as the number of fully populated cores increases, achieving a 1.9×, 3.8×, and 6.8× speedup for 16 threads (2 cores), 32 threads (4 cores), and 64 threads (8 cores) respectively — when compared with the optimized single core (1 core × 8 threads). As-

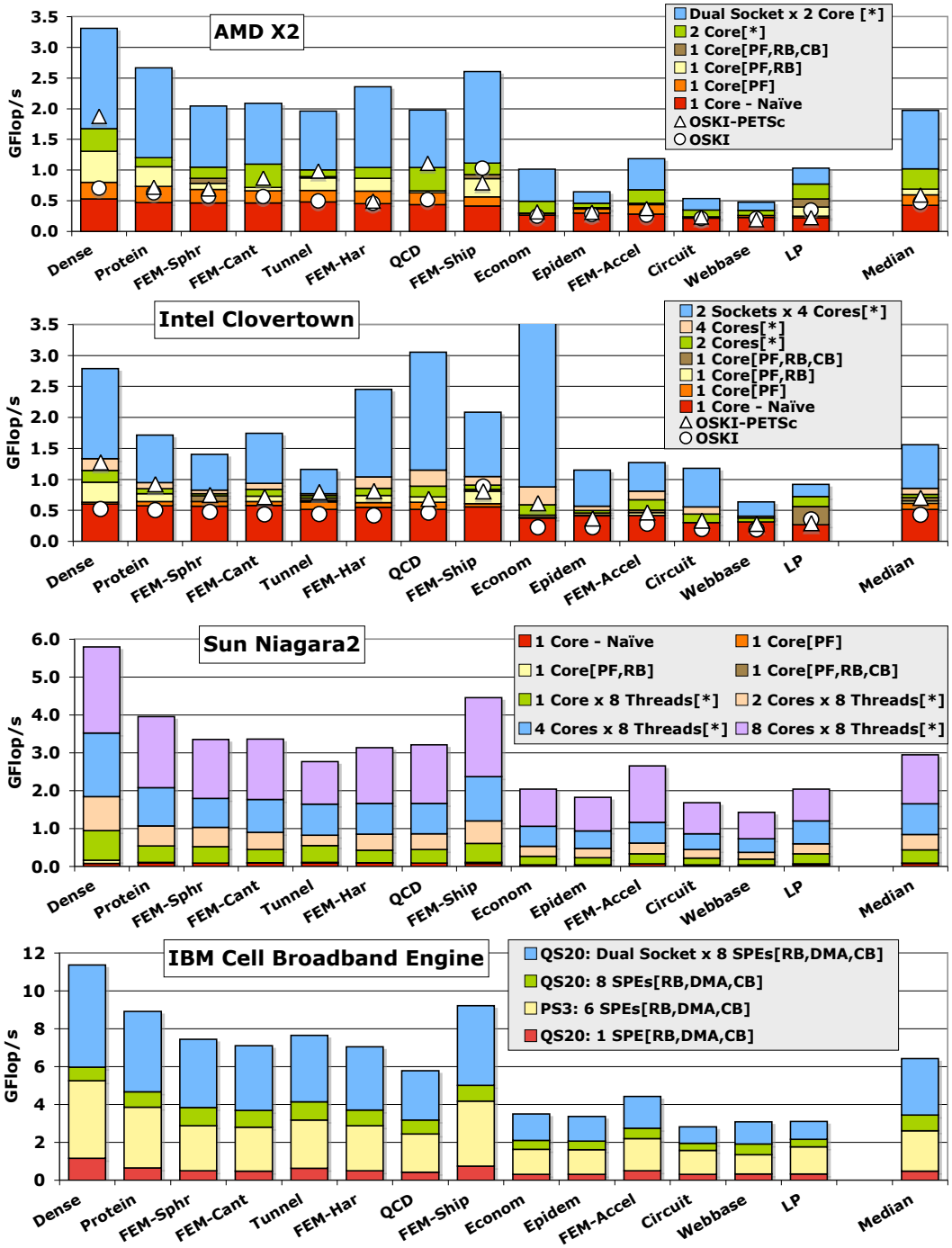


Figure 4: Effective SpMV performance (not raw flop rate) on (from top to bottom) AMD X2, Clovertown, Niagara, and Cell, showing increasing degrees of single-core optimizations — prefetching (PF), register blocking (RB) and cache-blocking (CB) (denoted as *) — as well as performance on increasing numbers of cores, and multiple-socket (full system) optimized results. OSKI and OSKI-PETSc results are denoted with circles and triangles. Note: Bars show the best performance for the current subset of optimizations/parallelism. The LP matrix could not be run on the PS3; the 6 SPE QS20 data is shown instead.

toundingly, the full system single socket (64 thread) median results achieve 3 GFlop/s, more than 3× the performance of a single socket of the x86 machines.

Although Niagara2 achieves high performance, we believe that such massive thread-level parallelism is a less efficient use of hardware resources than a combination of intelligent prefetching and larger L1 cache lines. Nevertheless, Niagara2 performance may be a harbinger of things to come in the multi- and many-core era, where high performance will depend on effectively utilizing a large number of participating threads.

5.6 STI CELL Performance

Finally, Cell results appear in Figure 4 (bottom). Although the Cell platform is often considered poor at double-precision arith-

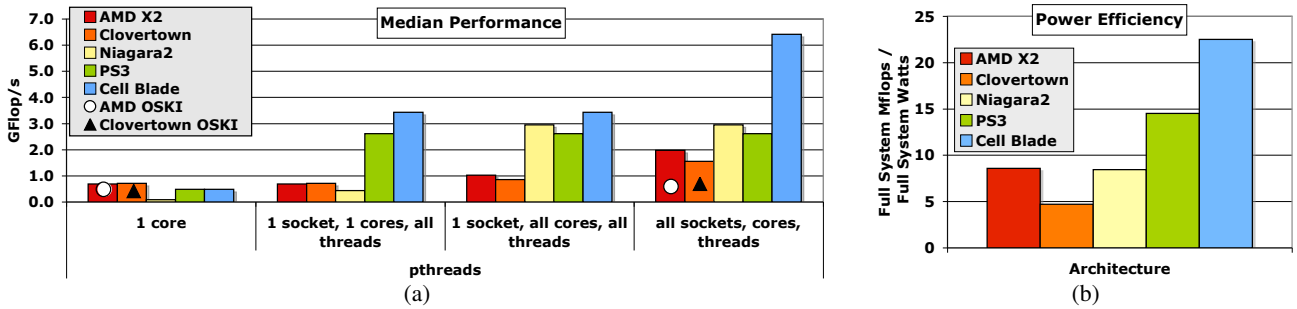


Figure 5: Architectural comparison of the median matrix performance showing (a) GFlop/s rates of OSKI and optimized SpMV on single-core, full socket, and full system and (b) relative power efficiency computed as total full system Mflop/s divided by sustained full system Watts (see Table 1).

metic, results show the contrary — the Cell's SpMV execution times are dramatically faster than all other multi-core SMP's in our study. Cell performance is highly correlated with a single parameter: the actual flop:byte ratio of the matrix. In fact, since DMA makes all transfers explicit, including bytes unused due to a lack of spatial locality, it is possible to show that for virtually every matrix, Cell sustains about 90% of its peak memory bandwidth. On the more difficult matrices, the get list DMA command can effectively satisfy Little's Law [2] on a stanza gather; as a result, there is no reduction in actual sustained bandwidth. The PS3, with only 6 cores, is not capable of saturating its 25 GB/s peak bandwidth even on the dense matrix test case — indicating that it is bound by inner kernel performance (every Cell double-precision instruction stalls all subsequent issues for 7 cycles).

Examing multicore behavior, we see speedups of 5.4×, 7.1×, and 13.4× when utilizing 6 cores (PS3), 8 cores (single blade socket), and 16 cores (full blade system), compared with a single-core Cell blade. These results show impressive scaling on a single socket, however, the lack of sharing of data between local stores can result in a non-trivial increase in redundant total memory traffic. Recall that this implementation is sub-optimal as it requires 2×1 or larger BCOO; thus, it may require twice the memory traffic of a CSR implementation, especially for matrices that exhibit poor register blocking, This version was not implemented due to the inefficiency of scalar double-precision on Cell.

5.7 Architectural Comparison

Figure 5(a) presents compares median matrix results, showing the optimized performance of our SpMV implementation, as well as OSKI, using a single-core, fully-packed single socket, and full system configuration. Results clearly indicate that the Cell blade significantly outperforms all other platforms in our study, achieving 3.3×, 4.1×, and 2.2× speedups compared with the AMD X2, Clovertown, and Niagara2 despite its poor double-precision and sub-optimal register blocking implementation. Cell's explicitly programmed local store allows for user-controlled DMAs, which effectively hide latency and utilize a high fraction of available memory bandwidth (at the cost of increased programming complexity).

Looking at the Niagara2 system, results are extremely poor for a single core/thread, but improve quickly with increasing thread parallelism. The overall performance is on par with a single Cell socket, and significantly faster than the x86 machines, albeit with significantly more memory bandwidth. Finally, the dual core AMD X2 attains better performance than a quad core Clovertown both within a single-socket, as well as in full-system (dual-socket) experiments. This is somewhat surprising as the Clovertown's per-socket computational peak is 4.2× higher than the AMD X2, has no NUMA affects, and uses FBDIMMs.

Next we compare power efficiency — one of today's most important considerations in HPC acquisition — across our evaluated suite of multicore platforms. Figure 5(b) shows the Mflop-to-Watt ratio based on the median matrix performance and the actual (sustained) full-system power consumption (Table 1). Results show that the Cell blade leads in power efficiency, followed by the single socket PS3. The single socket Niagara2 delivered the same efficiency as a dual socket Opteron. Although the Niagara2 system attains high performance and productivity, it comes with a price — power. Eight channels of FBDIMM drove sustained power to 350W for the best performing matrix. Thus, Niagara2's system efficiency for the SpMV kernel was marginal. The Clovertown delivered the poorest power efficiency because its high power and low (relative) performance. The Cell blade attains an approximate advantage of 2.6×, 4.8×, and 2.7× compared with the AMD X2, Clovertown, and Niagara2 (respectively).

6. SUMMARY AND CONCLUSIONS

We are witnessing a sea change in computer architectures due to the impending ubiquity of multicore processors. Understanding the most effective design and utilization of these system, in the context of demanding scientific computations, is of utmost priority to the HPC community. In this work we examine SpMV, an important and highly- demanding numerical kernel, on one of the most diverse sets of multicore configurations in existing literature.

Overall our study points to several interesting multicore observations. First, the "heavy-weight" out-of-order cores of the AMD X2 and Clovertown showed sub-linear improvement from one to two cores. This finding is surprising because the multicore designs of these architectures provide approximately twice the bandwidth of their single-core counterparts. However, significant additional performance was seen on the dual socket configurations. This indicates that sustainable memory bandwidth may become a significant bottleneck as core count increases, and software designers should consider bandwidth reduction (*e.g.*, symmetry, advanced register blocking, A^k methods [20]) as a key algorithmic optimization .

On the other hand, the two in-order "light-weight" cores in our study, Niagara2 and Cell — although showing significant differences in architectural design and absolute performance — achieved high scalability across numerous cores at reasonable power demands. This observation is also consistent with the gains seen from each of our optimization classes; overall, the parallelization strategies provided significantly higher speedups than either code or data-structure optimizations. These results suggest that multicore systems should be designed to maximize sustained bandwidth and tolerate latency with increasing core count, even at the cost of single core performance.

Our results also show that explicit DMA transfers can be an

effective tool, allowing Cell to sustain a much higher fraction of the available memory bandwidth, compared with the alternative techniques on our other platforms: out-of-order execution, hardware prefetching, explicit software prefetching, and hardware multithreading.

Finally, our work compares a multicore-specific Pthreads implementation with a traditional MPI approach to parallelization across the cores. Results show that the Pthreads strategy resulted in runtimes more than twice as fast as the message passing strategy. Although the details of the algorithmic and implementation differences must be taken into consideration, our study strongly points to the potential advantages of explicit multicore programming within and across SMP sockets.

In summary, our results show that matrix and platform dependent tuning of SpMV for multicore is at least as important as suggested in prior work [20]. Future work will include the integration of these optimizations into OSKI, as well as continued exploration of optimizations for SpMV and other important numerical kernels on the latest multicore systems.

7. ACKNOWLEDGMENTS

We would like to express our gratitude to both Forschungszentrum Jülich in der Helmholtz-Gemeinschaft for access to their 3.2GHz Cell blades, Sun Microsystems for access to their Niagara2 systems, and David Patterson and the University of California Berkeley for access to their x86 machines. All authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231.

8. REFERENCES

[1] K. Asanovic, R. Bodik, B. Catanzaro, et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[2] D. Bailey. Little's law and high performance computing. In *RNR Technical Report*, 1997.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202, 1997.

[4] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computations on vector multiprocessors. Technical Report CMU-CS-93-173, Department of Computer Science, CMU, 1993.

[5] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, Jul-Aug, 1999.

[6] R. Geus and S. Röllin. Towards a fast parallel sparse matrix-vector multiplication. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Proceedings of the International Conference on Parallel Computing (ParCo)*, pages 308–315. Imperial College Press, 1999.

[7] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006.

[8] M. Gschwind, H. P. Hofstee, B. K. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach; fourth edition*. Morgan Kaufmann, San Francisco, 2006.

[10] E. J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[11] B. C. Lee, R. Vuduc, J. Demmel, and K. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *Proceedings of the International Conference on Parallel Processing*, Montreal, Canada, August 2004.

[12] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix vector multiply using unroll-and-jam. In *Proc. LACSI Symposium*, Santa Fe, NM, USA, October 2002.

[13] R. Nishtala, R. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication, and Computing*, March 2007.

[14] A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proc. Supercomputing*, 1999.

[15] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. *Graph Theory and Computing*, pages 183–217, 1973.

[16] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proc. Supercomputing*, 1992.

[17] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[18] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[19] R. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, December 2003.

[20] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, June 2005.

[21] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick. Memory hierarchy optimizations and bounds for sparse $A^T Ax$. In *Proceedings of the ICCS Workshop on Parallel Linear Algebra*, volume LNCS, Melbourne, Australia, June 2003. Springer.

[22] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries*, New York, USA, June 2002.

[23] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proc. International Conference on Supercomputing (ICS)*, Cairns, Australia, June 2006.

[24] J. W. Willenbring, A. A. Anda, and M. Heroux. Improving sparse matrix-vector product kernel performance and availabillity. In *Proc. Midwest Instruction and Computing Symposium*, Mt. Pleasant, IA, 2006.

[25] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the Cell processor. *International Journal of Parallel Programming*, 35(3):263–298, 2007.