Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms

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, as well as the highly multithreaded Sun Niagara and heterogeneous STI Cell. 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-scale multiprocessor (CMP) system design in order to better mitigate trade-offs among performance, energy efficiency, and reliability [4, 5, 7]. 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 hardware-multithreaded single-socket eight-core Sun Niagara, and the heterogeneous local-store based architecture of the STI Cell single-socket PlayStation3 (PS3) and dual-socket QS20 Cell Blade, containing six and eight cores respectively. We show that our SpMV optimization strategies (summarized in Table 2) — and explicitly programmed and tuned for these multicore environments — attain significant performance improvements compared to existing state-of-the-art serial and parallel SpMV implementations [10]. An overview of these performance results can be found in Figures 1 and 2.

Additionally, we present key insights into the architectural trade-offs 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 in turn implies that CMP system design should emphasize, as core counts increase, bandwidth and latency tolerance over single-core performance. Finally, we show that, in spite of relatively slow double-precision arithmetic, the STI Cell provides sigificant advantages in terms absolute performance and power-efficiency, compared with the other multicore architectures in our test suite.

2 SpMV Overview

SpMV dominates the performance of diverse applications in scientific and engineering computing, economic modeling, information retrieval, among others; yet, conventional implementations of SpMV have historically been relatively poor, running at 10% or less of machine peak on single-core cache-based microprocessor-based systems [10]. 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. 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. It is clear that 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.

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 [10]. 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 [8]. 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 [2] and Trilinos [12]. This paper compares our multicore implementations against a generic "off-the-shelf" approach that uses PETSc's distributed memory SpMV with OSKI used to tune the serial component ("OSKI-PETSc"), and MPICH 1.2.7p1 configured to use the shared-memory (ch_shmem) device where message passing is replaced with memory copying. PETSc's SpMV uses a block-row partitioning, with equal numbers of rows per process by default.

Due to the importance of SpMV computation, the literature on optimization and tuning of this kernel is extensive. Optimization techniques include data structure compression [11], spatial and temporal locality enhancement [9], vectorization [3], and low-level optimization such as prefetching and software pipelining [6]. Due to space limitations, we omit a detailed discussion of related work, which will appear in the final paper version.

3 Experimental Testbed

We examine several leading CMP system designs in the context of the demanding SpMV algorithm. Here, 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 Niagara; and the heterogeneous STI Cell processor configured both as the single-socket six-core PS3 as well as the dual-socket eight-core Cell blade. An overview of the architectural configurations and characteristics can be found in Table 1. (Due to space limitations, detailed architectural schematics were omitted, and will be presented in the final paper version.)

3.1 AMD X2 Dual-core Opteron

The AMD Opteron 2214 is AMD's current dual-core processor offering. Each core operates at 2.2GHz, can fetch and decode three x86 instructions per cycle, and can 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 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 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.

Core Architecture	AMD Opteron	Intel Clovertown	Sun Niagara	STI Cell SPE	STI Cell SPE
Type	super scalar	super scalar	multi-threaded single issue	SIMD dual issue	SIMD dual issue
Clock (GHz)	2.2	2.3	1.0	3.2	3.2
L1 DCache	64KB	32KB	8KB	—	—
Local Store	—	—	—	256KB	256KB
DP flops/cycle	2	4	1 *(integer)*	4/7	4/7
DP Gflop/s	4.4	9.33	1 *(integer)*	1.83	1.83

System	AMD X2	Clovertown	Niagara	PS3	Cell Blade
Sockets on system	2	2	1	1	2
Cores per Socket	2	4	8	6(+1)	8(+1)
L2 cache	4MB *(1MB/core)*	16MB *(4MB/2cores)*	3MB *(shared)*	—	—
DP Gflop/s	17.6	74.7	8 *(integer)*	11	29
DRAM Type	DDR2-667 (2x128)	DDR2-667 (4x64)	DDR-400 (4x128)	XDR (1x128)	XDR (2x128)
System DRAM (GB/s)	21.2	21.2	25.6	25.6	51.2
System Flop:Byte ratio	0.83	3.52	0.31	0.43	0.57
Total Power in sockets (Watts)	190	160	72	$<$100	200
Total System Power (Watts)	275	333	267	200*	315

Table 1. Architectural summary of AMD Opteron X2, Intel Clovertown, Sun Niagara, and STI Cell multicore chips. *Power consumption values are based on vendor published results, except for the PS3, where power is estimated based on the Cell QS20 Blade.

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 2.33GHz, can fetch and decode four instructions per cycle, and can execute 6 micro-ops per cycle. There is both a 128b SSE adder (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 9.33 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 running at 1.33GHz (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 memory bandwidth of 21.3 GB/s. Unlike the AMD X2, each core may activate all four channels, but will likely never see the full bandwidth. The full system has 16MB of L2 cache, and 74.67 Gflop/s peak performance.

3.3 Sun Niagara

The Sun UltraSparc T1 “Niagara” eight-core processor presents an interesting departure from mainstream multicore chip design. Rather than depending on four-way superscalar execution, each of the 8 single issue, strictly in-order, cores supports four hardware thread contexts (referred to as Chip MultiThreading or CMT) to provide a total of 32 simultaneous hardware threads per socket. The CMT approach is designed to tolerate memory latency through fine-grained multithreading, while consuming relatively little power: 72 Watts per socket at 1.0 GHz.

Unfortunately, the T1 only offers a single non-pipelined floating point unit that is shared by all of the processor cores. Thus, for the purpose of this study, we used 64b integer arithmetic to gain insight into the follow-on Niagara-2, which will have fully-pipelined double precision arithmetic units for each core. Our study examines the Sun UltraSparc T1000 with a single T1 processor operating at 1.0 GHz. Since each core is single-issue, it has a peak performance of 1Gop/s performance per core (8 GOP/s per socket) peak performance for 64-bit integer operations, which we are using as a proxy to floating point performance. Each core has access to its own private 8KB L1 caches, but is connected to a shared 12-way set-associative L2 cache via a 64 GB/s on-chip crossbar switch. The socket is fed by four dual channel DDR-400 memory controllers that deliver an aggregate of 25.6 GB/s to the L2. Niagara has no hardware prefetching and software prefetching only places data in the L2; thus attempts to hide all memory latency are via multi-threading.

3.4 STI Cell

The Sony Toshiba IBM (STI) Cell processor forms the heart of the Sony PS/3 (PS3) video game console, where its 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 together with up to eight simpler SIMD cores (Synergistic Processing Elements / SPEs) for the computationally intensive work. 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 its 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.

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. The SPE’s double precision is not as poorly supported as Niagara since 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 Opteron’s 4.4 Gflop/s

Code Optimization				Data Structure Optimization				Parallelization Optimization			
	x86	N	C		x86	N	C		x86	N	C
Pipelining			✓	BCOO	✓	✓		Threading	✓[4]	✓[4]	✓[5]
Branchless	✓[8]			16-bit indices	✓	✓	✓	Row Parallel	✓	✓	✓
SIMDization	✓	N/A	✓	32-bit indices	✓	✓		NUMA Aware	✓	✓[8]	
Pointer Arithmetic		✓[8]		Register Blocking	✓	✓		Process Affinity	✓	✓[8]	✓[6]
PF/DMA[1] Values & Indices	✓	✓	✓	Cache Blocking	✓[2]	✓[2]	✓[3]	Memory Affinity	✓	N/A	✓[7]
PF/DMA[1] Pointers/Vectors			✓	TLB Blocking	✓	✓					

Table 2. Overview of SpMV optimizations attempted in our study for the x86 (AMD X2 and Clovertown), Niagara (N), and Cell (C) architectures. Notes: [1] PF/DMA (Prefetching or Direct Memory Access), [2] sparse cache blocking, [3] dense cache blocking, [4] Pthreads, [5] libspe 1.0, [6] numactl(cpubindnode), [7]numactl(interleaved), [8] implemented, but resulted in no significant speedup.

or the Xeon's 9.33 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 dual-sockets with eight SPEs (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 blade will show strong variations in sustained bandwidth if NUMA is not properly exploited.

4 SpMV Optimizations

In this section we discuss the set of SpMV code optimization explored in our study. Our optimization effort consisted of three phases: low-level code optimization (without data structure changes), data structure optimization (with requisite code changes), and parallelization optimization. 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, reduced indices, threading, row parallelization, NUMA-aware mapping, process affinity, and memory affinity; an optimization summary can be found in Table 2.

Most of these optimizations complement those available in OSKI. In particular, OSKI includes register blocking and single-level cache/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.

We found that introducing low-level code optimizations explicitly was beneficial on each architecture. This approach helps ensure that the cores are bound by memory bandwidth, rather than by instruction latency and throughput. We devised a Perl-based code generator that produces the SpMV kernel, using the subset of optimizations appropriate for each underlying system. Our generators helped us explore the large optimization space effectively and productively.

4.1 Code Optimizations

Our first step in developing an optimized SpMV implementation was to explore efficient low-level implementation techniques. None of these change the underlying data structure — they simply examined techniques that improved its processing. A conventional CSR-based SpMV 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. The code can be further optimized via a *branchless implementation*, which is in effect a segmented scan of vector-length equal to one [3].

Prior experience has shown significant benefits from the branchless approach on several platforms, but this optimization did not improve performance on either of the x86 (AMD/Intel) machines despite several implementation attempts (cmov, SSE muxing, jumps). 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 but is of little value on the out-of-order superscalars.

We also consider *explicit prefetching*, using our code generator to tune the prefetch distance from 0 (no prefetching) to 512 doubles (one page). Architectures like the AMD X2 rely on hardware prefetchers to overcome memory latency, however, prefetched data is placed in the L2 cache — as a result, the L2 latency must still be hidden. 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 Niagara 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.

Finally, our code generators can produce both conventional array indexing as well as *pointer arithmetic*, since certain instruction sets are particularly adept at complex array indexing. The pointer arithmetic form was moderately beneficial on Niagara for matrices with poor register blocking, but showed little benefit on other systems. Future work will examine this issue in more detail.

4.2 Data Structure Optimizations

The next phase of our implementation was to optimize the SpMV data structures. 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 approach, 16 bytes of storage are required for each matrix nonzero: 8 bytes for the double precision nonzero, plus 4 bytes for each row and each column coordinate. Our data structure transformations can cut these storage requirements in half.

Register blocking groups adjacent nonzeros into rectangular tiles, with only one coordinate index per tile [8]. 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 4×4, to enable *SIMDization* and minimize register pressure. Additionally, when the matrix dimension is less than 64k, we represent the coordinates with *memory efficient indices* of 2 bytes, saving up to 4 bytes per tile. This technique is less general but simpler than another recent index compression approach [11].

To further reduce memory bandwidth requirements, we use *block coordinate* (BCOO) storage in the presence of empty rows. Otherwise, empty rows waste row pointer storage and time executing zero-length loops. One solution is to use a generalized CSR format that stores only non-empty rows while associating explicit indices with either each row or with groups of consecutive rows, as is available in OSKI. In these experiments, we alternatively use BCOO format, storing both a row and column index with each block. Rather than tuning via search, our implementation performs one pass over the nonzeros to determine the combination of register blocking, index size, first/last row, and format that minimizes the matrix footprint. Future work will parallelize this routine to reduce optimization overhead.

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 x 1K) can improve SpMV performance [8, 9]. We take this idea a step further by accounting for cache utilization, as opposed to simply spanning a fixed number of columns. Specifically, we first specify the number of cache lines available for blocking, and divide them to cache elements of the source and destination vectors. For the resulting cache blocked row, we span enough columns such that the number of touched cache lines corresponding to the source vector 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 a unique number of columns. We refer to this optimization as *sparse cache blocking*, in contrast to the classical dense cache blocking approach. Unlike OSKI, where the cache blocking must be specified or searched for, we use a heuristic to cache block.

We apply the aforementioned register blocking heuristic individually to each cache block. Thus, it is possible for some cache blocks to be stored in 1x4 BCOO with 32-bit indices, and others in 4x1 BCSR with 16-bit indices. This approach often works well in selecting a cache blocking that improves performance.

Finally, we explore a data structure optimization that minimizes TLB misses. Previous work showed that TLB misses can vary by an order of magnitude depending on the blocking strategy, highlighting the importance of *TLB blocking* [9]. Similar to our cache blocking strategy, for each given row we determine the maximum number of columns based on the number of unique pages touched. We actually perform this heuristic between cache blocking rows and cache blocking columns. Rather than searching for the correct blocking (an expensive task as it requires re-encoding the matrix), we use these heuristics to find a well performing blocking. In the case of the Opteron we found it beneficial to block for the L1 TLB.

4.3 Parallelization Optimizations

Following code and data optimizations, we then perform parallelization optimizations to achieve scalable performance across multiple processing cores. We consider three possible approaches for exploiting SpMV *thread level parallelism*: column partitioning, row partitioning, and segmented scan. Column partitioning clearly requires explicitly blocking the matrix. Explicit blocking can be used during row partitioning as well, to facilitate implementations on NUMA machines. Our implementation attempts to statically load balance the matrix by balancing the number of nonzeros, as the transfer of this data accounts for the majority of time on matrices whose vectors fit in cache. A thread based segmented scan would allow dynamic parallelization (by nonzeros) within a sub-block of the matrix. This is conceptually similar to utilizing a segmented scan on a single processor, but would be implemented very differently. In this paper, we only exploit *row partitioning*; future work will examine column partitioning and segmented scan.

Finally, for the two NUMA architectures in our study, Cell and AMD X2, we apply *NUMA-aware* optimizations in which we explicitly assign each matrix block to a specific core and node. Each of these thread blocks may in turn be cache and TLB blocked, and each of the resulting cache blocks may be further register blocked. Depending on the OS and library support we either use `libnuma`, Linux scheduling, or Solaris scheduling to ensure that both the thread block and the process assigned to it are mapped to a the core (*process affinity*) and DRAM (*memory affinity*) proximal to it. Although it may be seem obvious that ignoring memory affinity can limit the AMD X2's performance, we found to our surprise that thread mapping can also affect the Clovertown's performance, no doubt due to the shared L2 cache.

4.4 Cell-Specific Optimizations

For the Cell processor we implemented a SpMV routine similar to [13], which uses only dense cache blocks and virtually no other optimization aside from the mandatory DMAs and compressed 2 byte indices. As a result, Cell is doubly handicapped in our benchmark: The hardware's double precision is fairly slow, and our code is far from optimized. To improve performance when using 8 or fewer SPEs on the Cell blade, we used Cell command `numactl` to bind both SPEs and memory to node 0; when running with 16 SPEs (both Cell sockets), we employed `numactl` to interleave pages. Although this is a suboptimal solution, it is superior to simply binding all data to a single node. However, as Section 6 shows, our partially-optimized Cell implementation achieves extremely impressive performance. The final paper version will include all relevant optimizations on Cell, including register blocking and NUMA optimizations, which will allow us to evaluate performance on the full set of test matrices.

5 Evaluated Sparse Matrices

To evaluate the performance of SpMV our 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, as well as a connectivity graph collected from a web crawl. 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 Table 3.

5.1 Performance Impact of Matrix Structure

Before presenting experimental results, we first explore the structure of several matrices in our test suite, and consider their expected affect 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 2 flops for 8 bytes, 0.25), 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 very 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 implements a simplistic version of cache blocking, this problem can be exacerbated on that platform.

Additionally, simply quantifying that a matrix has a few nonzeros per row, is insufficient in determining performance; we must also determine the average number of nonzeros per row for each cache block. For example, with a

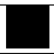
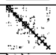
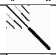
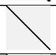
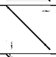
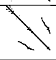
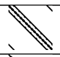
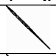
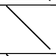
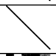
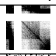
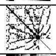
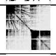
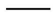
Matrix	Filename	Rows	Columns	NNZ	NNZ/row	Spyplot	Notes
Dense	dense2.pua	2K	2K	4M	2K		Dense matrix in sparse format
Protein	pdb1HYS.rsa	36K	36K	4.3M	119		Protein data bank 1HYS
FEM/Spheres	consph.rsa	83K	83K	6M	72.2		FEM Concentric spheres
FEM/Cantilever	cant.rsa	62K	62K	4M	64.5		FEM cantilever
Wind Tunnel	pwtk.rsa	218K	218K	11.6M	53.2		Pressurized wind tunnel
FEM/Harbor	rma10.pua	47K	47K	2.37M	50.4		3D CFD of Charleston harbor
QCD	qcd5-4.pua	49K	49K	1.9M	38.8		Quark propagators (QCD/LGT)
FEM/Ship	shipsec1.rsa	141K	141K	3.98M	28.2		Ship section/detail
Economics	mac-econ.rua	207K	207K	1.27M	6.1		Macroeconomic model
Epidemiology	mc2depi.rua	526K	526K	2.10M	4.0		2D Markov model of epidemic
FEM/Accelerator	cop20k-A.rsa	121K	121K	2.62M	21.7		Accelerator cavity design
Circuit	scircuit.rua	171K	171K	959K	5.6		Motorola Circuit Simulation
webbase	webbase-1M.rua	1M	1M	3.1M	3.1		Web connectivity matrix
LP	rail4284.pua	4K	1.1M	11.3M	2825		Railways set cover constraint matrix

Table 3. Overview of sparse matrices used in multicore evaluation.

fixed 17K columns per cache block, the ostensibly random matrix, FEM/Accelerator, should require 7 cache blocks. Thus its 2.6M nonzeros are divided into 121K rows, for a total of 3 nonzeros/row/cacheblock. This would indicate FEM/Accelerator would perform poorly on both the Cell as well as cache-blocked versions on the other multicore platforms. Our experimental data in Figure 1 confirm this expectation.

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 memory bandwidths 12.5 GB/s and 8.6 GB/s respectively, we don't expect the performance of Epidemiology to exceed 1.39 Gflop/s and 0.98 Gflop/s (respectively), regardless of CSR performance. The results of Figure 1 confirm this prediction.

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

	Sustained Memory Bandwidth: GB/s (%peak)						Sustained Computational Rate: Gflop/s (% peak)					
Machine	one core		1 full socket		full system		one core		1 full socket		full system	
Niagara	0.26	(1%)	2.06	(8%)	5.02	(20%)	0.065	(6.5%)	0.51	(6.4%)	1.24	(16%)
Clovertown	3.62	(34%)	6.56	(62%)	8.86	(42%)	0.89	(9.5%)	1.62	(4.3%)	2.18	(2.9%)
AMD X2	5.40	(51%)	6.61	(62%)	12.55	(59%)	1.33	(30%)	1.63	(19%)	3.09	(18%)
Cell(PS3)	3.25	(13%)	18.35	(72%)	18.35	(72%)	0.65	(35%)	3.67	(33%)	3.67	(33%)
Cell(Blade)	3.25	(13%)	23.20	(91%)	31.50	(62%)	0.65	(35%)	4.64	(32%)	6.30	(22%)

Table 4. Sustained bandwidth and computational rate for the *dense2.pua* dense matrix problem (stored in sparse format), in GB/s (and percentage of peak bandwidth) and Gflop/s (and percentage of peak performance).

the other hand, this matrix structure is amenable to effective cache-blocking as there are plenty of nonzeros per row. As a result, LP should show significant benefits to cache blocking on both the AMD X2 and Clovertown (recall that Niagara performance is heavily limited by L2 sustained bandwidth). Figure 1 confirms this prediction.

6 Performance Results and Analysis

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. We ran PETSc with up to 8 tasks, but only present the fastest results for the case where fewer tasks achieved higher performance. We compiled OSKI with both `gcc` and `icc`, and present only the best results here. For our code we used `gcc` on Niagara, Opteron, and Clovertown, and `xlc` on the Cell. We start with a detailed analysis of the dense matrix case followed by a broader discussion across the whole matrix suite.

6.1 Peak Effective Bandwidth

On any modern machine, SpMV should be limited by memory system performance, so we 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. Results confirm that the dense matrix reaches a flop:byte ratio close to upper bound of 0.25 (0.20 for Cell). Since all the multicore systems in our study have a flop:byte ratio greater than 0.25, we expect systems 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 show in Table 4.

As seen from the data, 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 91% of the theoretical potential. This is due, in part, to the explicit local store architecture of the Cell, which allows double-buffered DMA's to hide the majority of memory latency. This high memory bandwidth translates to high sustained performance, where over 6 Gflop/s (but only 62% of theoretical bandwidth) is attained on the dual-socket Cell blade. Sub-linear Cell scaling was due to page interleaving between nodes; expect that a NUMA aware version will significantly improve this. Results also show that the PS3 is actually not memory bound, as full socket (6 SPE) performance increases by almost 6x compared with single core results, and does not saturate the bandwidth.

Looking at the other end of the spectrum, the data in Table 4 shows that the Niagara system sustains only 1% of its memory bandwidth when utilizing a single thread on a single core. There are numerous reasons for this poor performance. First, the Niagara 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 3 cycles, while the L2 latency is about 22 cycles. Each SpMV nonzero requires two unit stride accesses, and one indexed load, resulting in between 23 and 48 cycles of memory latency per nonzero. When this is combined with an additional 10 cycles for instruction execution, and 10 cycles for multiply latency, it quickly becomes apparent why a single thread on these strictly in-order cores only sustains between 29 and 46 Mflop/s for 1x1 CSR. Recall that all prefetches on Niagara place data in the L2, making explicit prefetching of little benefit on this platform. However, Niagara's architectural characteristics also indicate that the bandwidth behavior should scale to multiple cores and threads-per-core, which is validated in our results.

Although it is relatively easy to understand the performance trends of the Cell and Niagara in-order architectures, the behavior of the (out-of-order) superscalar AMD X2 and Clovertown are more difficult to predict. Results surprisingly show that the full AMD X2 socket cannot saturate its available 10.6 GB/s bandwidth, even though a single core can use 5.4 GB/s. It is even less clear why the extremely powerful Clovertown core can only utilize 3.6 GB/s (34%) of its memory bandwidth, when the front side bus (FSB) can theoretically deliver 10.6 GB/s. It is interesting to note that a Clovertown MCM can utilize the same fraction of FSB bandwidth as the AMD X2's sustained memory bandwidth. Performance results in Table 4 show that, for this memory bandwidth limited application, the AMD X2 and Clovertown achieve almost identical computational rates for a full socket (about 1.6 Gflop/s), even though the peak rate of the Clovertown socket is 4.2x higher than the AMD X2 — this accounts for the significant disparity in the percentage of computational peak (19% on the AMD X2 vs. 4.3% on the Clovertown). As a sanity check, we also ran tests (not shown) on a small matrix that fit in the cache, and found that, as expected, the performance is very high — 12 Gflop/s on the Clovertown. Thus, performance is limited by the memory system but not by bandwidth per se, even though accesses are mostly unit stride and are prefetched.

6.2 AMD X2

Figure 1 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

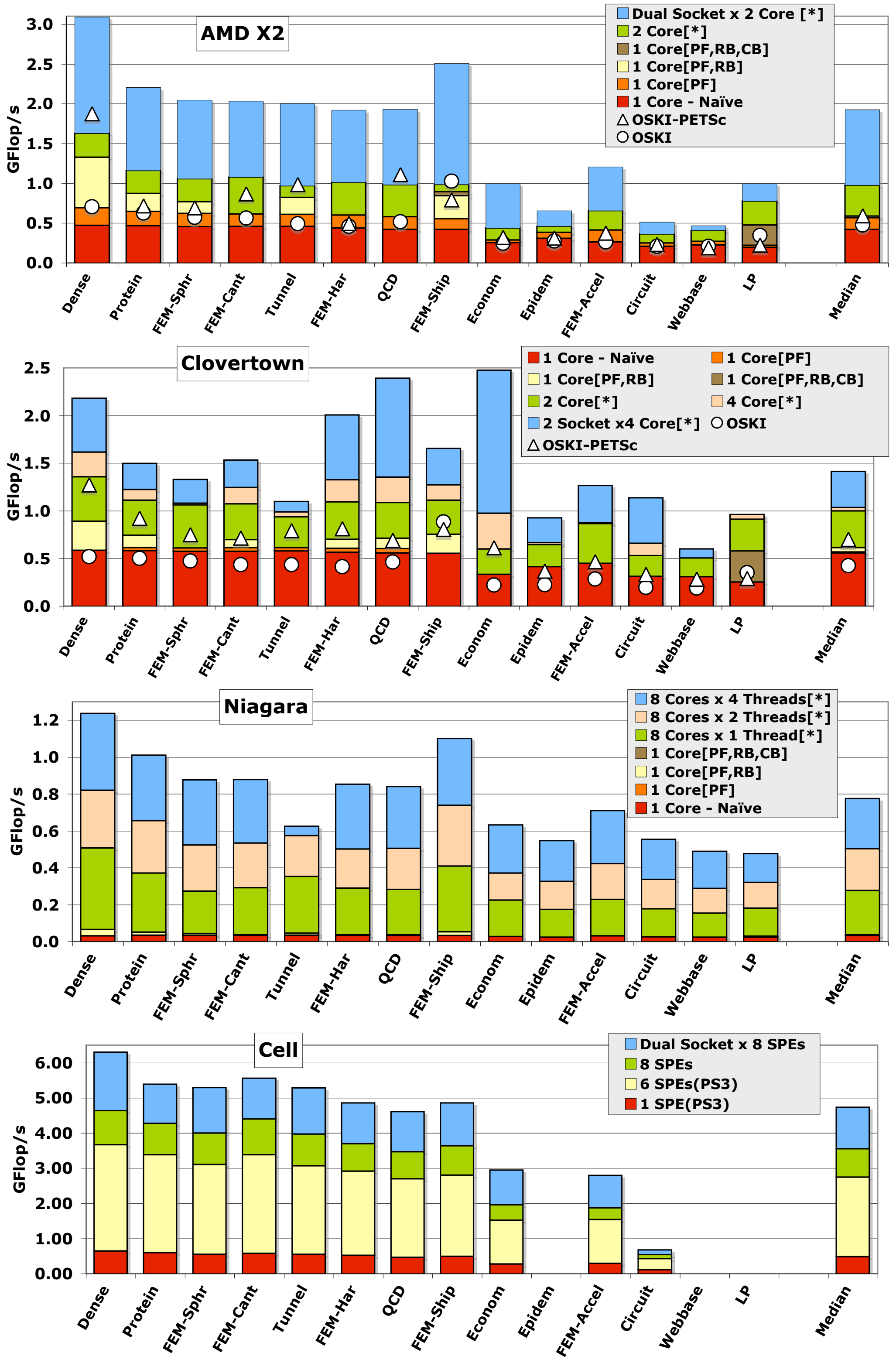


Figure 1. Effective SPMV performance (not raw flop rate) on (from top to bottom) AMD X2, Clovertown, Niagara, and Cell, showing increasing degrees of single-core optimizations — prefetching (PF), register blocking (RB) and cache-blocking (CB) — as well as performance on increasing numbers of cores with all optimizations (denoted as *), 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.

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 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 can be seen 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 10–12.5 GB/s, which corresponds to a high computational rate of 2–3 Gflop/s. On the other hand, matrices with low flop:byte ratios generally show poor parallelization and cache behavior, sustaining a bandwidth significantly less than 8 GB/s, and thus achieve performance in the range 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 1.4x in the median case, while achieving about a 1.2x speedup over the highly tuned OSKI library (where prefetching undoubtedly helped). For the parallel, multicore experiments, we find performance improvements of 1.7x (saturation) and 3.3x (multiple controllers) for the dual core and full system (dual-core dual-socket) configurations, respectively, when compared to our optimized single-core results. More impressively, our full-system implementation runs almost 3.2x 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. Secondly, 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].

6.3 Intel Clovertown

The quad-core Clovertown SpMV data can be found in Figure 1 (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 it does on the AMD X2 — no doubt due to the very large (16 MB) aggregate L2 cache, and the larger 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 (2.5x) 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.

Examining the median Clovertown performance, shows that our single-core SpMV optimization resulted in only 1.1x 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 is rarely any benefit from software prefetching. Additionally, register blocking was only useful on less than half of the matrices, while cache blocking held little benefit due to the large L2 cache of the Clovertown.

However, a significant speedup of 1.6x is seen (between the optimized versions) when comparing one and two cores of the Clovertown system. Performance only increases slightly when four cores (within a single socket) are employed, since the two-core experiment usually attains the maximum sustained FSB bandwidth. Examining the full system (dual-socket, quad-core), shows performance that exceeds the optimized serial case by only 2.3x; this is somewhat disappointing as the Clovertown achieved a 1.7x speedup on only 4 cores total. Moreover — unlike the AMD X2 — performance rarely increases when aggregate system bandwidth doubled in dual-socket experiments. These results were confirmed during MPI stream benchmarking (which will be presented in the final paper version). Finally, comparing results with the OSKI autotuner, we see a serial improvement of 1.4x and a parallel full-system speedup of 2x compared with OSKI and OSKI-PETSc respectively. These results once again highlight the effectiveness of our explicitly programmed, multicore-specific optimization and parallelization schemes.

6.4 Sun Niagara

Figure 1 (third from top) presents SpMV performance of the Niagara system, showing increasing levels of optimizations for the single-threaded (single core) test case, as well as optimized performance using all eight cores on increasing numbers of threads. The full system configuration utilizes eight cores and all four hardware-supported CMT contexts. Recall, that our examined Niagara T1 only offers a single non-pipelined floating point for the entire system. Thus, for the purpose of this study, we used 64-bit integer arithmetic to approximate the throughput of the follow-on Niagara-2, which will have fully-pipelined double precision arithmetic units for each core. As discussed in

Section 6.1, Niagara's performance is heavily bound by the interaction between the L1, L2, lack of L1 prefetching, and the strictly in-order cores.

Results show that, as expected, single thread results are extremely poor, achieving only 32 Mflop/s for the median matrix in the naïve case, with about 15% speedup from our suite of optimizations (37 Mflop/s). Significant performance improvement are achieved as the number of cores and threads increases, achieving a 7.6x, 13.8x, and 21.2x speedup for 8 threads (8 cores, 1 thread), 16 threads (8 cores, 2 threads), and 32 threads (8 cores, 4 threads) respectively — when compared with optimized single thread performance. However, the full system (32 thread) median results only achieve 0.8 Gflop/s, significantly less than the other platforms in our study.

We believe that Niagara-2 performance, with twice as many threads (8 cores with 8 threads each) running at 40% higher frequency, will significantly improve performance. Nevertheless, increasing hardware threading would probably be less beneficial than intelligent prefetching, larger L1 cache lines, or improved L2 latency. Niagara performance may be a harbinger of things to come in the multi- and many-core era, where high performance will depend on a large number of participating threads.

6.5 STI CELL

Finally, Cell results can be seen in Figure 1 (bottom). Although the Cell platform is often considered poor at double-precision arithmetic, results show the contrary — as the Cell's SpMV runtimes are dramatically faster than all other multi-core SMP's in our study. Cell performance is highly correlated with two related parameters: The average number of nonzeros per row per non-empty cache block, and the actual flop:byte ratio of the cache block. Without perfect branch prediction or a branchless implementation, we see that matrices with few nonzeros per row (Economics and Circuit) are heavily penalized by the loop overhead including the branch misprediction penalty. Matrices with a large number of nonzeros per row, run with enough cores, can actually saturate the system's memory bandwidth. The PS3, with only 6 cores, is not capable of saturating its 25 GB/s peak bandwidth on even the dense matrix testcase — indicating that it is bound by inner kernel performance. On the other hand, a single socket on the Cell blade is memory bound, achieving up to 91% of the theoretical memory bandwidth (see Table 4).

Examing multicore behavior, we see speedups of 5.7x, 7.4x, and 9.9x when utilizing 6 cores (PS3), 8 cores (single blade socket), and 16 cores (full blade system), compared with a single-core PS3. These results show almost perfect scaling on the PS3, while the Cell blade still has headroom for scaling improvements. It is important to highlight that our current SpMV Cell version is not fully optimized, lacking the NUMA features, which undoubtedly hamper blade performance and scalability. The final version of the paper will contain the fully-optimized Cell implementation, which will provide an additional performance boost, while allowing us to evaluate the full set of experimental matrices.

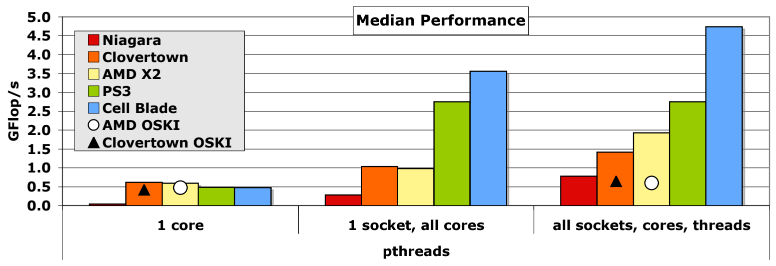
6.6 Architectural Comparison

Figure 2(a) presents a comparison summary of the 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 3.4x, 3.6x and 12.8x single-socket speedups compared with the Clovertown, AMD X2, and Niagara (respectively). The Cell's explicitly programmed local store architecture 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). We expect even higher performance when the Cell SpMV code is fully optimized for the final paper version.

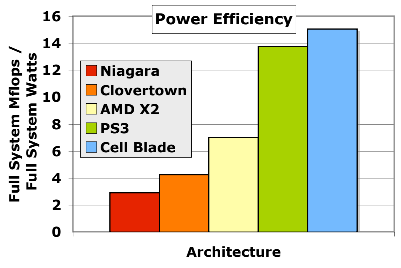
Looking at the Niagara system, results are extremely poor for a single core/thread, but grow quickly with increasing thread parallelism. Unfortunately, the overall performance is still quite a bit lower than the other platforms. We expect the next generation of the Niagara processor to further performance gains, and provide a reasonable platform for double-precision computations. To truly improve performance, the Niagara system must improve sustainable memory bandwidth as the current implementation lacks to ability to effectively amortize L2 latency.

Finally we see that the Clovertown attains comparable performance to the AMD X2 within a single-socket, but actually performs slower in full-system (dual-socket) experiments. This is somewhat surprising as the Clovertown's per-socket computational peak is 4.2x higher than the AMD X2. Experiments show that the Clovertown's sustainable memory bandwidth did not scale in the multi-socket tests, despite attaining approximately the same single socket bandwidth as the AMD X2.

Next we compare power efficiency — one of today's most important considerations in HPC acquisition — across our evaluated suite of multicore platforms. Figure 2(b) shows the Mflop-to-Watt ratio based on the matrix performance and the full-system power consumption (Table 1). Results show that the Cell blade leads in power efficiency, while the single-core PS3 attains near comparable performance when taking power into consideration, both attain an



(a)



(b)

Figure 2. 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 maximum full system Watts (see Table 1).

approximate advantage of 2.1x, 3.5x, and 5.2x compared with the AMD X2, Clovertown, and Niagara (respectively). Although the Niagara chip consumes less power than the other chips in our study, the system power is only marginally less than the other platforms; thus Niagara's power efficiency is the lowest of our evaluated architectures.

7 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 improvement from one to two cores; which should not be surprising as the multicore designs of these architectures provide approximately twice the bandwidth of their single-core counterparts. However, little additional gain was seen on the quad-core Clovertown, since FSB bandwidth did not scale further for quad core. This indicates that memory bandwidth may become a significant bottleneck as core count increases, and software designers should consider bandwidth reduction as a key algorithmic optimization (*e.g.*, symmetry, advanced register blocking, A^k methods [10]).

On the other hand, the two in-order "light-weight" cores in our study, Niagara and Cell — although showing significant differences in architectural design and absolute performance — achieved high scalability across numerous cores at reasonable power demands. This 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 indicate multicore systems should be designed to maximize sustained bandwidth and tolerate latency with increasing core count, even at the cost of single core performance — effective prefetching is an example.

Results also showed that explicit DMA transfers can be a very effective tool, allowing Cell to sustain a much higher fraction of the available memory bandwidth, compared with the alternative techniques of our other architectures: 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 the that Pthreads strategy resulted in runtimes more than twice as fast as the message passing implementation. 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 [10], and that optimizations like the ones we have described should be made generally available in a system like OSKI. Future work will continue exploring optimization for SpMV and other important numerical kernels on the latest generation of multicore systems, while making these tuning packages publicly available.

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] Satish Balay, William D. Gropp, and Lois Curfman McInnesand Barry 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. Birkhauser Press, 1997.

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

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

[5] P.P Gelsinger. Microprocessors for the new millennium: Challenges, opportunities, and new frontiers. In *Proc. In International Solid State Circuits Conference, (ISSCC)*, San Francisco, CA, 2001.

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

[8] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[9] Rajesh Nishtala, Richard Vuduc, James W. Demmel, and Katherine A. Yelick. When cache blocking sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication, and Computing: Special Issue on Computational Linear Algebra and Sparse Matrix Computations*, March 2007.

[10] Richard Vuduc, James W. Demmel, and Katherine 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.

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

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

[13] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.