Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms

Samuel Williams*†, Jonathan Carter*, Leonid Oliker* John Shalf*, Katherine Yelick*†
*CRD/NERSC, Lawrence Berkeley National Laboratory Berkeley, CA 94720
†CS Division, University of California at Berkeley, Berkeley, CA 94720
{SWWilliams, JTCarter, LOliker, JShalf, KAYelick}@lbl.gov

Abstract

We present an auto-tuning approach to optimize application performance on emerging multicore architectures. The methodology extends the idea of search-based performance optimizations, popular in linear algebra and FFT libraries, to application-specific computational kernels. Our work applies this strategy to a lattice Boltzmann application (LBMHD) that historically has made poor use of scalar microprocessors due to its complex data structures and memory access patterns. We explore one of the broadest sets of multicore architectures in the HPC literature, including the Intel Clovertown, AMD Opteron X2, Sun Niagara2, STI Cell, as well as the single core Intel Itanium2. Rather than hand-tuning LBMHD for each system, we develop a code generator that allows us identify a highly optimized version for each platform, while amortizing the human programming effort. Results show that our auto-tuned LBMHD application achieves up to a 14× improvement compared with the original code. Additionally, we present detailed analysis of each optimization, which reveal surprising hardware bottlenecks and software challenges for future multicore systems and applications.

1 Introduction

The computing revolution towards massive on-chip parallelism is moving forward with relatively little concrete evidence on how to best to use these technologies for real applications [1]. Future high-performance computing (HPC) machines will almost certainly contain multicore chips, likely tied together into (multi-socket) shared memory nodes as the machine building block. As a result, applications scientists must fully harness intra-node performance in order to effectively leverage the enormous computational potential of emerging multicore-based supercomputers. Thus, understanding the most efficient design and utilization of these systems, in the context of demanding numerical simulations, is of utmost priority to the HPC community.

In this paper, we present an application-centric approach for producing highly optimized multicore implementations through a study of LBMHD — a mesoscale algorithm for simulating homogeneous isotropic turbulence in dissipative magnetohydrodynamics. Although LBMHD is numerically-intensive, sustained performance is generally poor on superscalar-based microprocessors due to the complexity of the data structures and memory access patterns [11, 12]. Our work uses a novel approach to implementing LBMHD across one of the broadest sets of multicore platforms in 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 dual-socket×eight-core STI Cell QS20 Blade, as well as one of the first scientific studies of the hardware-multithreaded single-socket×eight-core×eight-thread Sun Niagara2. Additionally, we examine performance on the monolithic VLIW dual-socket×single-core Intel Itanium2 platform.

Our work explores a number of LBMHD optimization strategies, which we analyze to identify the microarchitecture bottlenecks in each system; this leads to several insights in how to build effective multicore applications, compilers, tools and hardware. In particular, we discover that, although the original LBMHD version runs poorly on all of our superscalar platforms, memory bus bandwidth is not the limiting factor on most examined systems. Instead, performance is limited by lack of resources for mapping virtual memory pages (TLB limits), insufficient cache bandwidth, high memory latency, and/or poor functional unit scheduling. Although of some these bottlenecks can be ameliorated through code optimizations, the optimizations interact in subtle

978-1-4244-1694-3/08/$25.00

ways both with each other and with the underlying hardware. We therefore create an *auto-tuning* environment for LBMHD that searches over a set of optimizations and their parameters to maximize performance. We believe such application-specific auto-tuners are the most practical near-term approach for obtaining high performance on multicore systems.

Results show that our auto-tuned optimizations achieve impressive performance gains — attaining up to 14× speedup compared with the original version. Moreover, our fully optimized LBMHD implementation sustains the highest fraction of theoretical peak performance on any superscalar platform to date. We also demonstrate that, despite the relatively weak double precision capabilities, the STI Cell provides considerable advantages in terms of raw performance and power efficiency — at the cost of increased programming complexity. Finally we present several key insights into the architectural tradeoffs of emerging multicore designs and their implications on scientific algorithm design.

2 Overview, Related Work, and Code Generation

During the past fifteen years Lattice Boltzmann methods (LBM) have emerged from the field of statistical mechanics as an alternative [15] to other numerical simulation techniques in numerous scientific disciplines. The basic idea is to develop a simplified kinetic model that incorporates the essential physics and reproduces correct macroscopic averaged properties. In the field of computational fluid dynamics LBM have grown in popularity due to their flexibility in handling irregular boundary conditions and straightforward inclusion of mesoscale effects such as porous media, or multiphase and reactive flows. More recently LBM have been applied to the field of magnetohydrodynamics [5, 10] with some success.

The LBM equations break down into two separate pieces operating on a set of distribution functions, a linear free-streaming operator and a local non-linear collision operator. The most common current form of LBM makes use of a Bhatnagar-Gross-Krook [2] (BGK) inspired collision operator — a simplified form of the exact operator that casts the effects of collisions as a relaxation to the equilibrium distribution function on a single timescale. Further implicit in the method is a discretization of velocities and space onto a lattice, where a set of mesoscopic quantities (density, momenta, etc.) and distribution functions are associated with each lattice site. In discretized form:

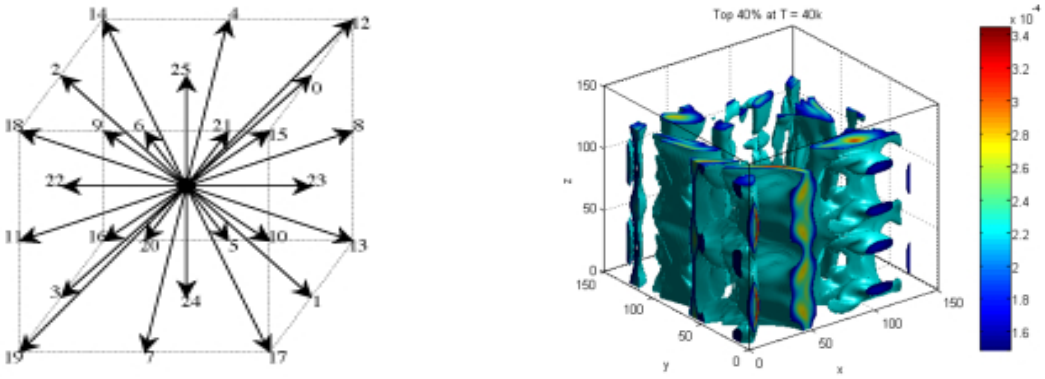
$$f_a(\mathbf{x} + \mathbf{c}_a\Delta t, t + \Delta t) = f_a(\mathbf{x}, t) - 1/\tau \left(f_a(\mathbf{x}, t) - f_a^{eq}(\mathbf{x}, t)\right) \quad (1)$$

where $f_a(\mathbf{x}, t)$ denotes the fraction of particles at time step t moving with velocity $\mathbf{c}_a$, f^{eq} is the local equilibrium distribution function constructed from the macroscopic variables to satisfy basic conservation laws and τ the relaxation time. The velocities $\mathbf{c}_a$ arise from the basic structure of the lattice and the requirement that a single time step should propagate a particle from one lattice point to another. A typical discretization in 3D is the D3Q27 model [21] which uses 27 distinct velocities (including zero velocity) is shown in Figure 1(a).

Conceptually, a LBM simulation proceeds by a sequence of *collision()* and *stream()* steps, reflecting the structure of the master equation. The *collision()* step involves data local only to that spatial point, allowing concurrent, dependence-free point updates; the mesoscopic variables at each point are calculated from the distribution functions and from them the equilibrium distribution formed through a complex algebraic expression originally derived from appropriate conservation laws. Finally the distribution functions are updated according to Equation 1. This is followed by the *stream()* step that evolves the distribution functions along the appropriate lattice velocities. For example, the distribution function with phase-space component in the $+x$ direction is sent to the lattice cell one step away in x. The *stream()* step also manages the boundary-data exchanges with neighboring processors for the parallelized implementation. This is often referred to as the "halo update" or "ghost zone exchange" in the context of general PDE solvers on block-structured grids.

However, a key optimization described by Wellein and co-workers [18] is often implemented, which incorporates the data movement of the *stream()* step directly into the *collision()* step. They noticed that the two phases of the simulation could be combined, so that either the newly calculated particle distribution function could be scattered to the correct neighbor as soon as it was calculated, or equivalently, data could be gathered from adjacent cells to calculate the updated value for the current cell. In this formulation, the collision step looks much more like a stencil kernel, in that data are accessed from multiple nearby cells. However, these data are from different phase-space as well as spatial locations. The *stream()* step is reduced to refreshing the ghost cells or enforcing boundary conditions on the faces of the lattice.

Rüde and Wellein have extensively studied optimal data structures and cache blocking strategies for BGK LBM for various problems in fluid dynamics, in the context of both single threaded and distributed memory parallel execution [14, 18], focusing on data layout issues and loop fusing and reordering. In addition, inspired by Frigo and Strumpen's [6] work on cache oblivious algorithms for a 1- and 2-dimensional stencils, Wellein and



(a) (b)

Figure 1. (a) the 27 streaming directions of the D3Q27 Lattice and (b) Vorticity tubes deforming near the onset of turbulence in LBMHD simulation.

co-workers [22] have applied cache oblivious techniques to LBM. While this has proved a successful strategy for single threaded serial performance, it is not obvious how it is amenable to distributed-memory parallelism since it would require a complex set of exchanges for the boundary values of the distribution functions and the time-skewing may be difficult to integrate into a multi-physics simulation.

2.1 LBMHD

LBMHD [9] was developed to study homogeneous isotropic turbulence in dissipative magnetohydrodynamics (MHD). MHD is the theory of the macroscopic interaction of electrically conducting fluids with a magnetic field. MHD turbulence plays an important role in many branches of physics [3]: from astrophysical phenomena in stars, accretion discs, interstellar and intergalactic media to plasma instabilities in magnetic fusion devices. The kernel of LBMHD is similar to that of the fluid flow LBM except that the regular distribution functions are augmented by magnetic field distribution functions, and the macroscopic quantities augmented by the magnetic field. Moreover, because closure for the magnetic field distribution function equations is attained at the first moment (while that for particle distribution function equations is attained at the second moment), the number of phase space velocities to recover information on the magnetic field is reduced from 27 to 15. Although a D3Q27 lattice is used throughout the simulations, only a subset of the phase-space is needed to describe the evolution of the magnetic field. While LBM methods lend themselves to easy implementation of difficult boundary geometries (e.g., by the use of bounce-back to simulate no slip wall conditions) LBMHD performs 3-dimensional simulations under periodic boundary conditions — with the spatial grid and phase space velocity grid overlaying each other on a regular three dimensional Cartesian D3Q27 lattice. Figure 1(b) is reproduced from one of the largest 3-dimensional LBMHD simulations conducted to date [4], aiming to understand better the turbulent decay mechanisms starting from a Taylor-Green vortex — a problem of relevance to astrophysical dynamos. Here we show the development of turbulent structures in the z-direction as the initially linear vorticity tubes deform.

The original Fortran implementation of the code was parallelized using MPI, partitioning the whole lattice onto a 3-dimensional processor grid, and using ghost cells to facilitate efficient communication. This achieved high sustained performance on the Earth Simulator, but a relatively low percentage of peak performance on superscalar platforms [12]. The application was rewritten, for this study, around two lattice data structures, representing the state of the system, the various distribution functions and macroscopic quantities, at time t and at time $t + 1$. At each time step one lattice is updated from the values contained in the other. The algorithm alternates between these each data structures as time is advanced. The lattice data structure is a collection of arrays of pointers to double precision arrays that contain a grid of values. This is close to the 'structure of arrays' data layout [18], except that we have the flexibility to align the components of distribution functions or macroscopic quantities without the restrictions implicit in a Fortran multi-dimensional array. To simplify indexing, the unused lattice elements of the magnetic component are simply NULL pointers (see Figure 2).

```
struct{
  // macroscopic quantities
  double * Density;
  double * Momentum[3];
  double * Magnetic[3];
  // distributions
  double * MomentumDistribution[27];
  double * MagneticDistribution[3][27];
}
```

Figure 2. LBMHD data structure for each time step, where each pointer refers to a N^3 grid.

2.2 Code Generation and Auto-Tuning

To optimize the LBMHD across a variety of multicore architectures, we employ the auto-tuning methodology exemplified by libraries like ATLAS [19] and OSKI [17]. We created a code generator that be configured to utilize many of the optimizations described in Section 4 including: blocking for the TLB, unrolling depth, instruction reordering, bypassing the cache, and software prefetching. The PERL code generator produces multithreaded C for the two primary subcomponents of the LBMHD code base: the *collision()* operation which implements the core LBM solver, and the *stream()* operation which implements the periodic boundary conditions as well as ghost-zone exchanges for the parallel implementation of the algorithm. We use POSIX Threads API to implement parallelism on the conventional microprocessor-based platforms and libspe 1.0 to launch the parallel computations on the Cell SPEs. In all variants, we use the data structure shown in Figure 2. Future work will incorporate data structure exploration into the auto-tuning process.

For the *collision()* operation this process can generate hundreds of variations that are then placed into a function table that is indexed by the optimizations. To determine the best configuration for a given problem size and thread concurrency, a 20 minute tuning benchmark is run offline to exhaustively search the space of possible code optimizations; to reduce tuning overhead the search space is pruned to eliminate optimization parameters unlikely to improve performance. Our experience suggests the time required for tuning can be substantially reduced and future work will explore this. For each optimization, we measured performance on five trials and report the best overall time. Future work will incorporate our node-centric LBMHD optimizations with explicit message-passing between nodes, allowing experiments on large-scale distributed-memory, multicore-based HPC platforms.

3 Experimental Testbed

Our work examines several leading multicore system designs in the context of the full LBMHD application. Our architecture suite consists of the dual-socket×quad-core Intel Clovertown, the dual-socket×dual-core AMD Opteron X2, the single-socket×eight-core hardware-multithreaded Sun Niagara2, and the dual-socket×eight-core STI Cell blade. Additionally, we examine the dual-socket×single-core Intel Itanium2 to explore the tradeoffs between its monolithic design and the simpler multiprocessor cores in our study. An architectural overview and characteristics appear in Table 1 and Figure 3. Note that, aside from the Itanium2, we obtained sustained system power data using an in-line digital power meter while the node was under a full computational load. We now present an overview of the examined systems.

3.1 Intel Itanium2

The Intel Itanium2 is an in-order 64-bit VLIW processor. It can issue two bundles (six instructions) per cycle, and can execute two FP fused-multiply adds (FMAs) per cycle. FP loads are directed to the 256KB L2 data cache rather than the L1; thus for FP code, the L2 is in effect the first level cache. Our evaluted system is a dual-socket×single core 1.3 GHz Madison3M incarnation with a 3MB L3 cache. Although it has 8.5 GB/s of DRAM to chipset bandwidth, the front side bus (FSB) only runs at 200MHz, thus limiting memory bandwidth to 6.4 GB/s. We include this single-core machine to explore the tradeoffs between the complex serial-performance-oriented Itanium2 core design and the simpler parallel-throughput-oriented multiprocessor cores in our study, with the forethought that multicore Itaniums will soon be available.

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 multi-chip module (MCM). Each core is based on Intel's Core2 microarchitecture, runs 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 (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 L1 cache, and each chip (two cores) has a shared 4MB L2 cache.

Core Architecture	Intel Itanium2	Intel Core2	AMD Opteron X2	Sun Niagara2	STI Cell SPE
Type	VLIW	super scalar out of order	super scalar out of order	MT dual issue†	SIMD dual issue
Clock (GHz)	1.30	2.33	2.20	1.40	3.20
DP GFlop/s	5.2	9.3	4.4	1.4	1.8
Local Store	—	—	—	—	256KB
first level Data Cache	256KB	32KB	64KB	8KB	—
first level TLB entries	128	16	32	128	256
Page Size	16KB	4KB	4KB	4MB	4KB

System	Itanium2	Clovertown	Opteron X2	Niagara2	Cell Blade
# Sockets	2	2	2	1	2
Cores/Socket	1	4	2	8	8(+1)
L2 cache	2×3MB	4×4MB(shared by 2)	4×1MB	4MB(shared by 8)	—
DP GFlop/s	10.4	74.7	17.6	11.2	29
DRAM Bandwidth (GB/s)	8.5	21.33(read) 10.66(write)	21.33	42.66(read) 21.33(write)	51.2
Flop:Byte Ratio	1.22	2.33	0.83	0.18	0.57
Max problem size without rolling the TLB	12^3	3^3	4^3	76^3	N/A
DRAM Capacity	4GB	16GB	16GB	64GB	1GB
Measured System Power (Watts)	500‡	330	300	450	285
Threading	Pthreads	Pthreads	Pthreads	Pthreads	libspe1.0
Compiler	icc 9.1	icc 10.0	gcc 4.1.2	gcc 4.0.4	xlc 8.2

Table 1. Architectural summary of Intel Itanium2, Intel Clovertown, AMD Opteron X2, Sun Niagara2, and STI Cell multicore chips. Except for the HP/Itanium2, The system power for all platforms (‡except the Itanium2) was measured using a digital power meter while under a full computational load. †Each of the two thread groups may issue up to one instruction.

Each socket has access to a 333MHz quad pumped FSB, delivering a raw bandwidth of 10.66 GB/s. In our study, we evaluate the Dell PowerEdge 1950 dual-socket platform, which contains two MCMs with dual independent busses. The chipset provides the interface to four fully buffered DDR2-667 DRAM channels that can deliver an aggregate read memory bandwidth of 21.33 GB/s. Unlike the AMD X2, each core may activate all four channels, but will likely never attain the peak bandwidth due to the limited FSB bandwidth and coherency protocol. The full system has 16MB of L2 cache and an impressive 74.7 GFlop/s peak performance.

3.3 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 L1 cache, and a 1MB 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 and a single cache-coherent HyperTransport (HT) link to access the other socket's cache and memory. Each socket can thus deliver 10.66 GB/s, for an aggregate NUMA (non-uniform memory access) memory bandwidth of 21.33 GB/s for the dual-core, dual-socket SunFire X2200 M2 examined in our study.

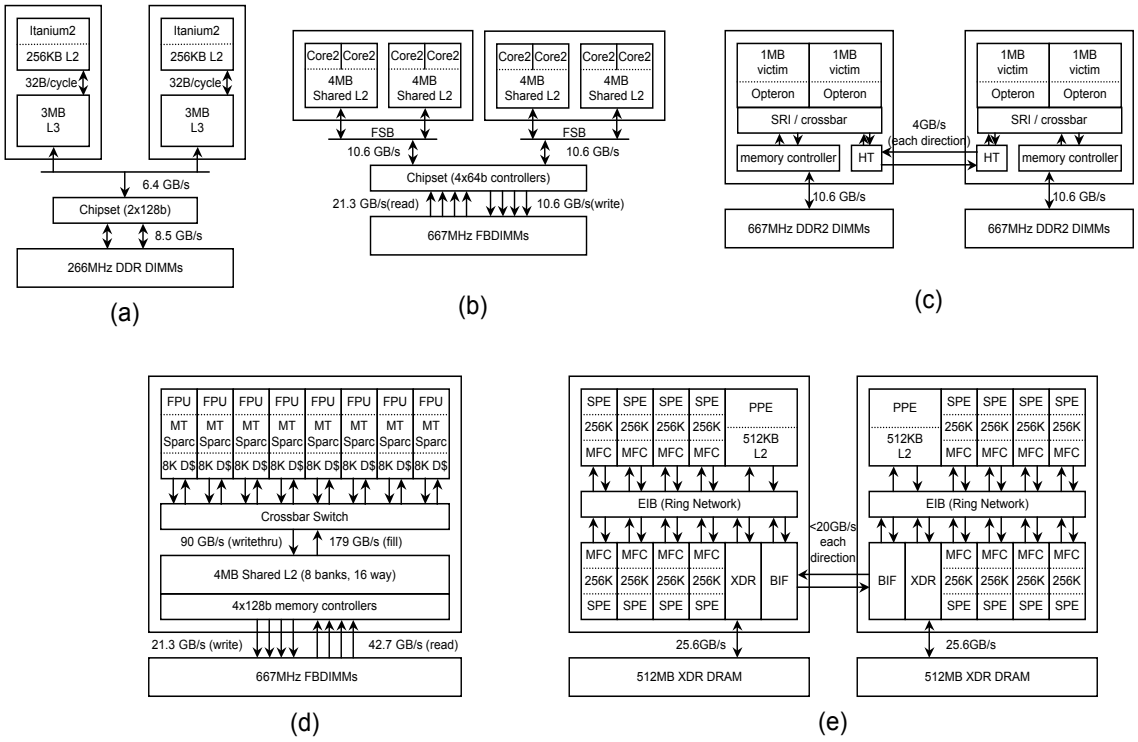


Figure 3. Architectural overview of (a) dual-socket Itanium2 (b) dual-socket×quad-core Intel Clovertown,(c) dual-socket×dual-core AMD Opteron X2 (d) single-socket×eight-core Sun Niagara2 (e) dual-socket×eight-core STI Cell.

3.4 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) — providing 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.

Niagara2 instantiates one FPU per core (shared among 8 threads). Our study examines the Sun UltraSparc T5120 with a one 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 4MB L2 cache via a 179 GB/s(read) on-chip crossbar switch. The socket is fed by four dual channel 667 MHz FB-DIMM memory controllers that deliver an impressive 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.

3.5 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]. 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.

Each SPE is a dual issue SIMD architecture which includes a half-pumped partially pipelined FPU. In effect, each SPE can execute one double-precision FMA 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 or the Xeon's 9.33 GFlop/s. In this study we utilize 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 Opteron system, the Cell blade is expected show strong variations in sustained bandwidth if NUMA is not properly exploited.

4 Multicore LBMHD Optimization

The two phases of each LBMHD time step are quite different in character. The *collision()* function has an $\mathcal{O}(n^3)$ floating-point computational cost in addtion to proportional data movement, while the lighter-weight *stream()* function performs $\mathcal{O}(n^2)$ data movement with no floating point requirements, (where n is the number of points per side side of the cubic lattice). Thus, the stream step accounts for a smaller portion of the overhead with increasing domain size. Note that when allocating the structure each 3D grid is padded to avoid cache-line aliasing.

4.1 Collision() Optimization

For each point in space, the *collision()* routine must read 73 double precision floating point values from neighboring points, perform about 1300 floating point operations, and write 79 doubles back to memory (see Figure 4). Superficially, the code requires a flop:byte ratio of approximately $2/3$ on conventional cache-based machines to attain peak performance (assuming a fill-on-write allocate cache policy). Consequently, we expect Itanium2 and Clovertown to be memory bandwidth limited, while the Niagara2 and Cell are expected to be computationally bound (given the respective flop:byte ratios of the system configurations in Table 1). Therefore, our auto-tuning optimizations target both areas given that the likely bottlenecks are system dependent.

4.1.1 Thread-Based Parallelization

The first, and most obvious step in the optimization process is to exploit thread-level parallelism. If we assume the lattice is composed of n_x, n_y, and n_z points in the x, y, and z directions, in FORTRAN's column-major layout one can view the problem as a set of $n_y n_z$ pencils of length n_x. Our implementation allows for a regular, user-specified 2D decomposition in the (y, z) plane. Load balancing is thus implicit. We explore alternate decompositions to find peak performance.

To manage the NUMA issues associated with the Opteron and Cell systems in our study, we also thread the lattice initialization routines (controlled by *collision()*'s parallelization guide) and exploit a first-touch allocation policy, to maximize the likelihood that the pencils associated with each thread are allocated on the closest DRAM interface. To correctly place threads, we use the Linux scheduler's routines for process affinity.

4.1.2 Phase-Space TLB Blocking

Given the structure-of-arrays memory layout of LBMHD's data structures, each phase-space component of the particle and magnetic-field distribution functions (that must be gathered from neighboring points in lattice space) will be widely spaced in DRAM. Thus for the page sizes used by our studied architectures and typical lattice sizes, a TLB entry is required for each of the 150 components read and written. (Note that this is significantly more demanding than typical computational fluid dynamics codes, as the MHD formulation uses an additional 15 phase-space cartesian vector components in the magnetic field distribution function.) Since the systems have relatively small L1 TLBs (16-128 entries), the original code version suffers greatly due to a lack of page locality and the resultant TLB misses.

Our next optimization (inspired by vector compilers) focuses on maximizing TLB-page locality. This is accomplished by fusing the real-space loops, strip mining into vectors, and interchanging the phase-space and strip-mined loops. For our implementation, the cache hierarchy is used to emulate a vector register file using several temporary structures. We modified our PERL code generator to create what is essentially a series of vector operations, which are called for each of the phase-space components.

On one hand, these vector-style inner loops can be extremely complex, placing high pressure on the cache bandwidth, thus favoring shorter vector lengths (VL) that enable all operands to fit within the first level cache. However, shorter VL make poor use of TLB locality (as only a few elements in a page are used before the next phase-space component is required) and access DRAM

```
for all Z{
  for all Y{
    for all X{
      // update one point in space
      recover macroscopic quantities using neighboring values: loop over phase space
      update distribution functions: loop over phase space
}}}

for all Z{
  for all vectors of points (of size VL) in XY plane{
    // simultaneously update VL points in space
    recover macroscopic quantities using neighboring values: loop over phase space
      strip-mined loop
    update distribution functions: loop over phase space
      strip-mined loop
}}}
```

Figure 4. Top: original *Collision()* pseudo code. Bottom: vectorized *Collision()* pseudo code

inefficiently for streaming loads and stores. We therefore use our auto-tuning framework to determine the optimal VL, since it is extremely difficult to predict the ideal size given these opposing constraints. To reduce the search space we observe that the maximum VL is limited by the size of the on-chip shared cache. Additionally we search only in full cache lines up to 128 elements then switch to powers of two, up to the calculated maximum VL.

4.1.3 Loop Unrolling and Reordering

Given these vector-style loops, we then modify the code generator to explicitly unroll each loop by a specified power of two. Although manual unrolling is unlikely to show any benefit for compilers that are already capable of this optimization, we have observed a broad variation in the quality of code generation on the evaluated systems. When future work adds explicit SIMDization, we expect this optimization to become essential. The most naive approach to unrolling simply replicates the body of the inner loop to amortize loop overhead. However, to get the maximum benefit of software pipelining, the inner loops must be reordered to group statements with similar addresses, or variables, together to compensate for limitations in some compiler's instruction-schedulers. The optimal reorderings are not unique to each ISA, but rather to each microarchitecture as they depend on the number of rename registers, memory queue sizes, and the functional unit latency. As such, our auto-tuning environment is well-suited for discovering the best combination of unrolling and reordering for a specific microarchitecture.

4.1.4 Software Prefetching

Our previous work [8, 20] has shown that software prefetching can significantly improve performance on certain superscalar platforms. We explore a prefetching strategy that modifies the unrolled code to prefetch the entire array needed one iteration (our logical VL) ahead. This creates a *double buffering* optimization within the cache, whereby the cache needs space for two copies of the data: the one currently in use and one the being simultaneously prefetched. This resulting prefetch distance easily covers the DRAM latency, and is considerably more effective than prefetching operands for the current loop (the typical software prefetch strategy). The Cell LBMHD implementation utilizes a similar double buffering approach within the local store, utilizing DMAs instead of prefetching. To facilitate the vector prefetching, we skew the first and last point in the parallelization guide to align all vectors to cache line boundaries.

4.1.5 SIMDization

SIMD units have become an increasingly popular choice for improving peak performance, but for many codes they are difficult to exploit — lattice methods are no exception. The SIMD instructions are small data-parallel data parallel operations that perform multiple arithmetic operations on data loaded from contiguous memory locations. While loop unrolling and code-reordering described previously reveals potential opportunities for exploiting SIMD execution, SIMD implementations typically do not allow unaligned (not 128b aligned) accesses. Structured grids and lattice methods often must access the previous point in the unit stride direction resulting in an unaligned load. One solution to remedy

the misalignment is to always load the next quadword and permute it to extract the relevant double. Although this is an expensive solution on most architectures, it is highly effective on Cell because each double precision instruction is eight times the cost of a permute. Future work will will explore similar techniques to fully exploit SSE on the x86 architectures.

4.1.6 Streaming Stores

SSE2 introduced a streaming store (*movntpd*) designed to reduce cache-pollution from contiguous writes that fill an entire cache line. Normally, a write operation requires the entire cache line be read into cache then updated and written back out to memory. Therefore a write requires two times more memory traffic than a read, and consumes a cache line in the process. However, if the writes are guaranteed to update the entire cache line, the streaming-store can completely bypass the cache and output directly to the write combining buffers. This has a several advantages: useful data is not evicted from the cache, the write miss latency does not have to be hidden, and most importantly the traffic associated with a cache line fill on a write-allocate is eliminated. Theoretically, given an equal mix of loads and stores, streaming stores can decrease an application's memory bandwidth requirements by 50%. Note that Cell's DMA engines can explicitly avoid the write allocate issue and eliminate memory traffic.

4.2 Stream() Optimization:

In the original MPI version of the LBMHD code, the *stream()* function updates the ghost-zones surrounding the lattice domain held by each task. Rather than explicitly exchanging ghost-zone data with the 26 nearest neighboring subdomains, we use the shift algorithm [13], which performs the exchange in three steps involving only six neighbors. The shift method makes use of the fact that after the first exchange is completed in one direction, the ghost cells have been partially populated in other directions. The next exchange includes this data, further populating the ghost cells, and so on.

To optimize behavior, we consider that the ghost-zone data must be exchanged on the faces of a logically 3D subdomain, but are not contiguous in memory for the X and Y faces. Therefore, the ghost-zone data for each direction are packed into and unpacked out of a single buffer, resulting in three pairs of message exchanges per time step. Even on our SMP systems, we chose to perform the ghost zone exchanges by copying into intermediate buffers rather than copying directly to neighboring faces. This approach is structurally compatible with an MPI implementation, which will be beneficial when we expand our implementation to massively-parallel distributed-memory machines in future work.

4.2.1 Thread-Based Parallelization

Although the *stream()* routine typically contributes little to the overall execution time, non-parallelized code fragments can become painfully apparent on architectures such as Niagara2 that have low serial performance (Amdahl's law). Therefore it is essential to parallelize the work among threads even for code sections with trivial overheads. Given that each point on a face requires 192 bytes of communication from 24 (9 particle scalars, 5 magnetic field vectors) different arrays, we maximize sequential and page locality by parallelizing across the lattice components followed by points within each array.

5 Performance Results and Analysis

Performance on each platform is shown in Figures 5(a–e), using the 64^3 and 128^3 problem sizes for varying levels of thread concurrency. Because computations on block structure grids favor larger subdomain sizes in order to maximize the surface-to-volume ratio, the problem sizes are selected to fill local memory on the tested platforms. The 64^3 problem size fills the limited DRAM available on the Cell blade, while 128^3 problem size fills a large fraction of the available memory on the remaining platforms. Given that the surface:volume ratio decreases with problem dimension, a larger fraction of runtime is spent in *stream()* for smaller problem sizes.

The stacked bar graphs in Figures 5(a–d), show LBMHD performance contributions in GFlop/s of varying optimizations (where applicable), including: the original version (blue), auto-tuned TLB blocking (red), auto-tuned unrolling/reordering (yellow), streaming stores (green), and explicit prefetching (gray). Observe that the top (gray) bar is rarely seen, as auto-tuned TLB blocking and unrolling/reordering combined with streaming stores generally attain close to maximum performance. Additionally, Figure 5(e) shows performance of the Cell-specific implementation, which includes TLB blocking, DMA transfers, and SIMDization. An overview of the full-system per-core performance, highlighting the variation in scaling behavior can be found in Figure 5(f). Finally, Table 2 presents a several salient performance characteristics of LBMHD's execution across the architectures. We now explore these data in more detail.

5.1 Itanium2

Figure 5(a) presents Itanium2 performance results. TLB blocking shows a clear advantage on this platform,

System	64^3					128^3			
	Intel IA64	Intel Cl-town	AMD X2	Sun Niagara2	Cell Blade	Intel IA64	Intel Cl-town	AMD X2	Sun Niagara2
GFlop/s	2.4	5.1	5.7	6.2	16.7	2.7	5.5	6.3	6.3
% Peak Flops	23%	7%	32%	55%	57%	26%	7%	36%	56%
Memory Bandwidth (GB/s)	3.6	5.1	5.7	9.3	16.7	4.0	5.5	6.3	9.5
% Peak Memory Bandwidth	42%	16%	27%	14%	33%	47%	17%	29%	15%
Auto-Tuned Vector Length	512	16	256	16	64†	512	16	24	16
Auto-Tuned Unrolling/Reordering	1/1	8/8	4/4	8/1	2/2†	1/1	8/2	8/2	4/2
% Time in *stream()*	16%	13%	13%	9%	N/A	9%	8%	8%	5%

Table 2. Full-system LBMHD optimized performance characteristics, including the auto-tuned vector length and unrolling/reordering values. †The Cell code is hand optimized, not auto-tuned.

attaining an impressive 3.7x to 13.8x runtime improvement. The results also show no benefit from explicitly inserting prefetch directives as the `icc` compiler appropriately issues software prefetching instructions. Additionally, the optimal unrolling/reordering factor was found to be 1/1 (see Table 2), indicating that `icc` was effectively software pipelining the *collision()* loops without manual intervention. Attempts to explicitly unroll loops resulted in substantial performance degradation. Auto-tuning discovered the optimal vector length to be 512 grid points, the largest on any evaluated platform. This optimal VL generates a footprint (600KB) larger than the L2 cache size, but uses only 512 doubles at a time per array — or 25% of a 16KB page. This provides some insight into the relative importance of L2 and TLB misses.

The Itanium2 is able to sustain over 4GB/s of DRAM bandwidth (63% of FSB peak) for LBMHD. However, the tested system shows sublinear speedup (only 25%) when moving from one to two sockets for the larger problem size, as can clearly by the steep drop off of Itanium2 per-core performance in Figure 5(f). This is not surprising considering that both processors share a common 6.4 GB/s front side bus for both memory traffic and coherency (see Table 1).

5.2 Clovertown

The data in Figure 5(b) clearly indicates that TLB blocking benefits Clovertown less as as the number of cores increase. The auto-tuned VL (seen in Table 2) is 16 points (19KB), which represents only 3% of the page size. Thus it is evident that the L1 cache effect is dominant on this processor. The auto-tuned unrolling matches well with the cache line size (8 doubles), although *icc* appeared to deliver nearly the same performance without explicit unrolling. Additionally, Clovertown benefits from streaming stores — 33% and 21% for the single- and dual-socket configurations respectively. This indicates that Clovertown is running into a bandwidth limit that can be forestalled by decreasing the memory traffic.

In the multithreaded experiments, the optimally auto-tuned TLB-blocking results in approximately 4.0 GB/s and 4.4 GB/s of memory bandwidth utilization for two and four cores (respectively), which is close to the practical limits of a single FSB [20]. The quad pumping a dual FSB architecture has reduced data transfer cycles to the point where they are on parity with coherency cycles. Surprisingly, performance only improves by 43% in the eight-core experiment when both FSBs are engaged, despite the fact that the aggregate FSB bandwidth doubled. Note, however, that although each socket cannot consume 100% of the DRAM bandwidth, each socket can activate all of the DIMMs in the memory subsystem. The declining per-core performance, clearly seen in Figure 5(f), suggests the chipset's capabilities limit multi-socket scaling on memory intensive applications.

5.3 Opteron

The performance data in Figure 5(c) shows that the TLB blocking and streaming stores optimizations increase Opteron performance by approximately 1.7x. Similar to the Clovertown, the auto-tuner chose a vector length for the 128^3 problem that fits within the 64KB L1 cache, while the optimal unrolling also matches the cache line size — indicating the limitations of the *gcc* compiler. A vector length of 256 provided a very slight performance boost over 24 on the 64^3 problem. Virtually no benefit was provided from explicit software prefetching.

Results also show that the Opteron only consumes 1.6 GB/s per core (Table 2) for the optimized LBMHD algorithm. Thus memory bandwidth is not an impediment to performance, allowing the Opteron to achieve

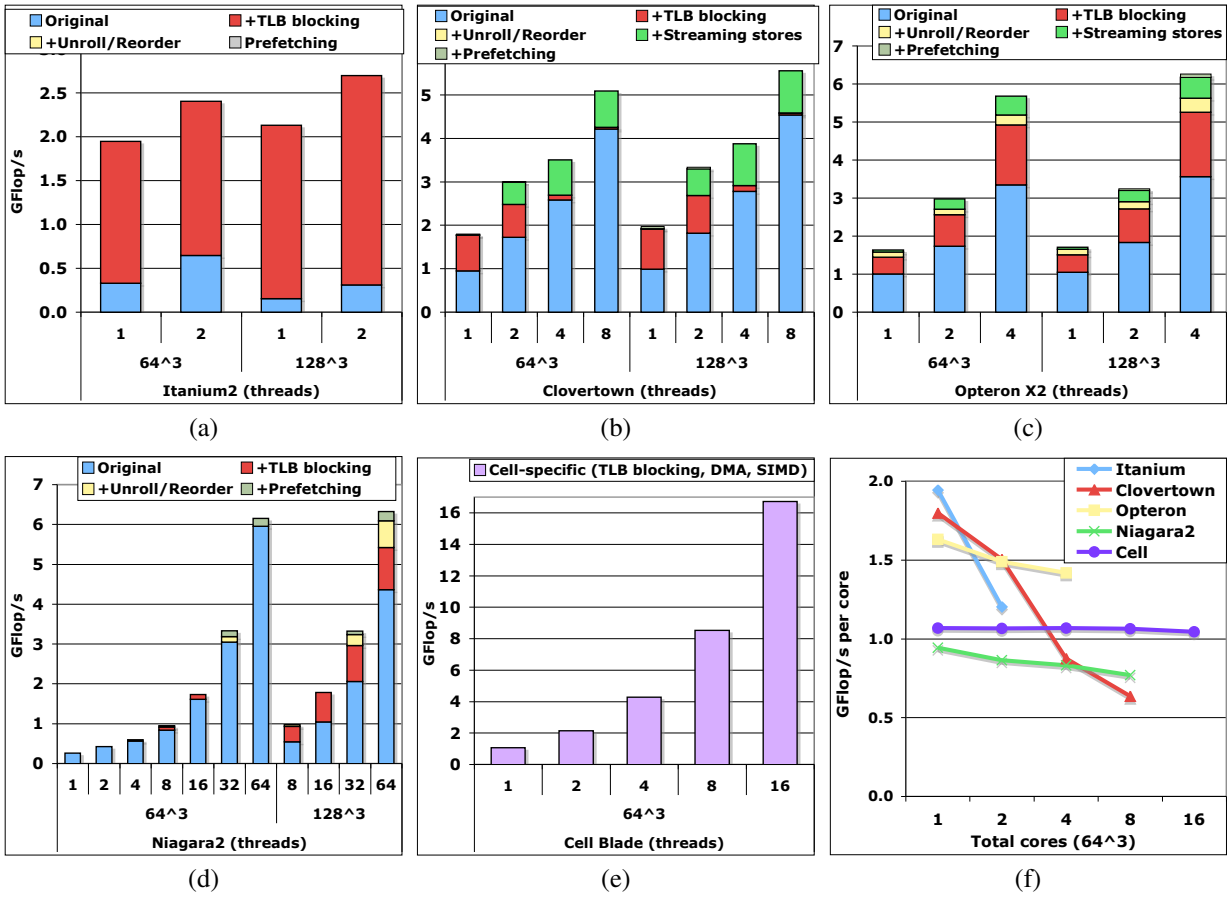


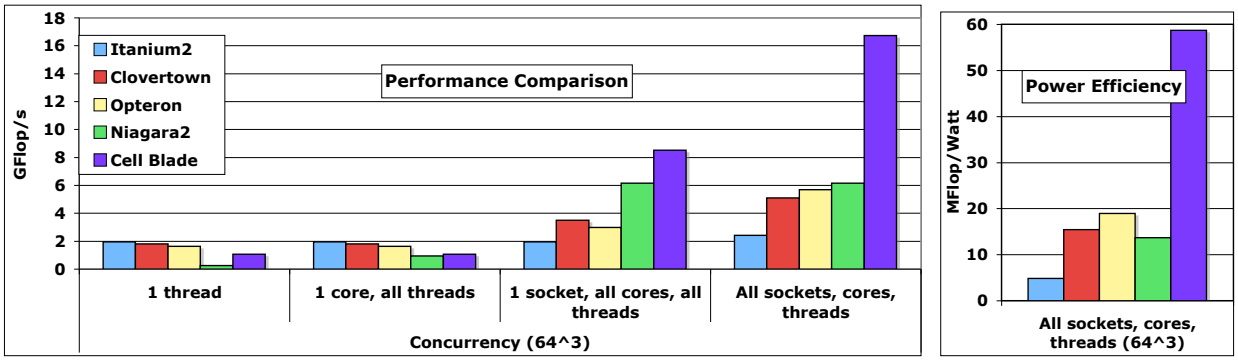
Figure 5. Contributions of explored optimizations of LBMHD on (a) Itanium2 (b) Clovertown (c) Opteron (d) Niagara2, as well as (e) performance of the Cell-specific implementation. Full-system per-core performance across all platforms is shown in (f).

nearly linear scaling for both the multicore and multi-socket experiments, as seen in Figure 5(f). Given the limited bandwidth requirements, we expect the recently-released quad-core Barcelona processor to continue linear performance scaling with the doubling of the cores. Future experiments will extend our study to the latest generation of multicore platforms.

5.4 Niagara2

The Niagara2 experiments in Figure 5(d) show several interesting trends. For the small (64^3) problem size, almost no benefit is seen from our optimization strategies because, the entire problem can be mapped by Niagara's 128-entry TLB, given Solaris' 4MB pages. For the larger problem case (128^3), the working set can no longer be mapped by the TLB, causing our auto-tuned TLB blocking approach to improve performance by 25%. Since the Niagara2's shared L1 cache is only 8KB, each computational thread is only provided a working set of about 1 point. Given each L1 cache line is 16 bytes, each core will have to rely on its L2 working set (54 points). As a result, the auto-tuned vector length is only 16 (see Table 2), because the cache has a more substantial influence on performance than the comparatively large TLB.

Performance (as expected) is extremely low for a single thread, and increases by 4x when using all eight threads within a single core (multithreaded scaling), while improving more than 24x when engaging all 64 threads across the eight cores (multicore scaling) — the near linear multicore scaling can be seen in Figure 5(f). As a result, performance on the large problem size achieve an impressive 56% of peak while utilizing only 15% of the available memory bandwidth. This suggests further multicore scaling can easily be exploited on future Niagara systems.



(a) (b)

Figure 6. Comparison of (a) runtime performance and (b) power efficiency across all studied architectures for the 64^3 problem.

5.5 Cell Blade

Figure 5(e) shows that Cell has a number of unique features compared with the other platforms in our study. First observe that there is no *original* performance baseline. This is because the generic microprocessor-targeted source code cannot be naively compiled and executed on the Cell SPEs, due to its software controlled cache that requires explicit DMAs to manage memory movement. Therefore a Cell-specific implementation must be created in order to perform any meaningful experiments. The explicit DMAs obviate the need for streaming stores and our preliminary Cell version does not utilize auto-tuned blocking.

Looking at the performance behavior, the Cell achieves near perfect linear scaling across the 16 threads, as evident from Figure 5(f). Thus, even though each individual SPE is slower than any other core in our study (due to extremely weak double precision), the linear scaling coupled with the large number of cores results in the fastest aggregate LBMHD execution. Each Cell core delivers just over 1 GFlop/s, which translates to an impressive 56% of peak — despite the inability to fully exploit FMA (due to the lack of potential FMAs in the LBMHD algorithm). In terms of memory bandwidth utilization, it can be seen in Table 2 that Cell achieves approximately 17 GB/s or 33% of theoretical peak. This indicates that Cell can readily exploit more cores or enhanced double precision for the LBMHD algorithm. Note that the Cell *stream()* function has not yet been implemented and will presented in future work; however, we do not expect a dramatic change in overall performance as *stream()* typically constitutes a small fraction of the application's running time.

5.6 Architectural Comparison

Figure 6(a) compares LBMHD performance across our suite of architectures for the 64^3 problem, using a single thread, 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 a 7×, 3.3×, 2.9×, and 2.7× speedups compared with the Itanium2, Clovertown, Opteron, and Niagara2. Although the Cell platform is often considered poor at double-precision arithmetic, results show the Cell's LBMHD execution times are dramatically faster than all other multicore nodes in our study. However, the high performance is comes with the significant overhead of additional programming complexity.

Of the microprocessors that use a conventional cache hierarchy and programming model, the Niagara2 demonstrates the highest aggregate single-socket performance for both problem sizes. Niagara2's single thread performance is extremely poor, but per-core performance improves quickly with increasing thread parallelism. The overall performance, although lower than a single Cell socket, is significantly faster than the Itanium2 and x86 systems, albeit the tested system has dramatically higher memory bandwidth.

Comparing the x86 architectures, results shows that the dual-core Opteron attains comparable performance with the quad-core Clovertown. This is somewhat surprising as the Clovertown's per-socket computational peak is 4.2× higher than the Opteron and has no NUMA constraints. Finally, the monolithic VLIW Itanium2 system achieves the highest single-core performance across our architecture suite. However, little improvement is gained when running across both (single core) sockets due to its limited memory bandwidth. Thus aggregate

Itanium2 system performance is substantially lower than the fully-packed *single* socket rate of all the multicore platforms in our study. These results portend a trend toward simpler high-throughput cores that offer higher aggregate performance at the expense of the per-core serial performance.

Figure 6(b) compares LBMHD power efficiency (MFlop/s/Watt) on our evaluated testbed (see Table 1) — one of today's most important considerations in HPC acquisition. Results show that the Cell blade leads in power efficiency, attaining an impressive advantage of 12.2×, 3.8×, 3.1×, and 4.3×, compared with the Itanium2, Clovertown, Opteron, and Niagara2 (respectively) for the full-system experiments. Although the Niagara2 system attains high LBMHD performance, the eight channels of FBDIMM (with 16 DIMMs) drove sustained power to 450W, causing the power efficiency to fall below the x86 platforms. For the problem sizes in question, one could easily remove eight DIMMs and thus cut the power by more than 100W (and improve efficiency by 30%) without significantly reducing performance. Finally, the power hungry Itanium2 looses ground against the rest of the architectures, achieving less than a $1/12$ of the Cell's power efficiency.

6 Summary and Conclusions

The computing industry is moving rapidly away from exponential scaling of clock frequency toward chip multiprocessors in order to better manage trade-offs among performance, energy efficiency, and reliability. Understanding the most effective hardware design choices and code optimizations strategies to enable efficient utilization of these systems is one of the key open questions facing the computational community today.

In this paper we developed a set of multicore optimizations for LBMHD, a lattice Boltzmann method for modeling turbulence in magnetohydrodynamics simulations. We presented an auto-tuning approach, which employs a code generator that produces multiple versions of the computational kernels using a set of optimizations with varying parameter settings. The optimizations include: an innovative approach of phase-space TLB blocking for lattice Boltzmann computations, loop unrolling, code reordering, software prefetching, streaming stores, and use of SIMD instructions. The impact of each optimization varies significantly across architectures, making a machine-independent approach to tuning infeasible. In addition, our detailed analysis reveals the performance bottlenecks for LBMHD in each system.

Results show that the Cell processor offered (by far) the highest raw performance and power efficiency for LBMHD, despite having peak double-precision performance, memory bandwidth, and sustained system power that is comparable to other platforms in our study. The key architectural feature of Cell is explicit software control of data movement between the local store (cache) and main memory. However, this impressive computational efficiency comes with a high price — a difficult programming environment that is a major departure from conventional programming. Nonetheless, these performance disparities point to the deficiencies of existing automatically-managed coherent cache hierarchies, even for architectures with sophisticated hardware and software prefetch capabilities. The programming effort required to compensate for these deficiencies demolishes their initial productivity advantage.

Our study has demonstrated that — for the evaluated class of algorithms — processor designs that emphasize high throughput via sustainable memory bandwidth and large numbers of simpler cores are more effective than complex, monolithic cores that emphasize sequential performance. While prior reseach has shown that these design philosophies offer substantial benefits for peak computational rates [16], our work quantifies that this approach can offer significant performance benefits on real scientific applications.

Overall the auto-tuned LBMHD code achieved sustained superscalar performance that is substantially higher than any published results to date — over 50% of peak flops on two of our studied architectures, with speedups of up to 14× relative to the original code. Auto-tuning amortizes tuning effort across machines by building software to generate tuned code and using computer time rather than human time to search over versions. It can alleviate some of compilation problems with rapidly-changing microarchitectures, since the code generator can produce compiler-friendly versions and can incorporate small amounts of compiler- or machine-specific code. We therefore believe that auto-tuning will be an important tool in making use of multicore-based HPC systems of the future. Future work will continue exploring auto-tuning optimization strategies for important numerical kernels on the latest generation of multicore systems, while making these tuning packages publicly available.

7 Acknowledgments

We would like to express our gratitude to Forschungszentrum Jülich for access to their Cell blades. We would also like to thank George Vahala and his research group for the original version of the LBMHD code. All authors from LBNL were supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231.

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, University of California, Berkeley, 2006.

[2] P. Bhantnagar, E. Gross, and M. Krook. A model for collisional processes in gases i: small amplitude processes in charged and neutral one-component systems. *Phys. Rev.*, 94:511, 1954.

[3] D. Biskamp. *Magnetohydrodynamic Turbulence*. Cambridge University Press, 2003.

[4] J. Carter, M. Soe, L. Oliker, Y. Tsuda, G. Vahala, L. Vahala, and A. Macnab. Magnetohydrodynamic turbulence simulations on the Earth Simulator using the lattice Boltzmann method. In *Proc. SC2005: High performance computing, networking, and storage conference*, 2005.

[5] P. Dellar. Lattice kinetic schemes for magnetohydrodynamics. *J. Comput. Phys.*, 79, 2002.

[6] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th ACM International Conference on Supercomputing (ICS05)*, 2005.

[7] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Computing Fontiers*, pages 1–8, New York, NY, USA, 2006.

[8] S. Kamil, K. Datta, S. Williams, L. O. J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Memory Systems Performance and Correctness (MSPC)*, 2006.

[9] A. Macnab, G. Vahala, L. Vahala, and P. Pavlo. Lattice boltzmann model for dissipative MHD. In *Proc. 29th EPS Conference on Controlled Fusion and Plasma Physics*, volume 26B, Montreux, Switzerland, June 17-21, 2002.

[10] D. Martinez, S. Chen, and W. Matthaeus. Lattice Boltzmann magnetohydrodynamics. *Phys. Plasmas*, 1, 1994.

[11] L. Oliker, A. Canning, J. Carter, and J. Shalf. Scientific computations on modern parallel vector systems. In *Proc. SC2004: High performance computing, networking, and storage conference*, Pittsburgh, PA, Nov6-12, 2004.

[12] L. Oliker, J. Carter, M. Wehner, A. Canning, S. Ethier, et al. Leading computational methods on scalar and vector HEC platforms. In *Proc. SC2005: High performance computing, networking, and storage conference*, Seattle, WA, 2005.

[13] B. Palmer and J. Nieplocha. Efficient algorithms for ghost cell updates on two classes of MPP architectures. In *Proc. PDCS International Conference on Parallel and Distributed Computing Systems*, volume 192, 2002.

[14] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde. Optimization and profiling of the cache performance of parallel lattice Boltzmann codes. *Parallel Processing Letters*, 13(4):S:549, 2003.

[15] S. Succi. The Lattice Boltzmann equation for fluids and beyond. *Oxford Science Publ.*, 2001.

[16] D. Sylvester and K. Keutzer. Microarchitectures for systems on a chip in small process geometries. In *Proceedings of the IEEE*, pages 467–489, Apr. 2001.

[17] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series*. Institute of Physics Publishing, June 2005.

[18] G. Wellein, T. Zeiser, S. Donath, and G. Hager. On the single processor performance of simple lattice Boltzmann kernels. *Computers and Fluids*, 35(910), 2005.

[19] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimization of Software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[20] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. SC2007: High performance computing, networking, and storage conference*, 2007.

[21] D. Yu, R. Mei, W. Shyy, and L. Luo. Lattice Boltzmann method for 3d flows with curved boundary. *Journal of Comp. Physics*, 161:680–699, 2000.

[22] T. Zeiser, G. Wellein, G. Hager, A. Nitsure, K. Iglberger, and G. Hager. Introducing a parallel cache-oblivious blocking approach for the lattice boltzmann method. In *ICMMES-2006 Proceedings*, 2006.