Optimization of a Lattice Boltzmann Computation on State-of-the-Art Multicore Platforms

Samuel Williams[*,a,b], Jonathan Carter[a], Leonid Oliker[a], John Shalf[a], Katherine Yelick[a,b]

[a] *CRD/NERSC, Lawrence Berkeley National Laboratory Berkeley, Berkeley, CA 94720*
[b] *CS Division, University of California at Berkeley, Berkeley, CA 94720*

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 Xeon E5345 (Clovertown), AMD Opteron 2214 (Santa Rosa), AMD Opteron 2356 (Barcelona), Sun T5140 T2+ (Victoria Falls), as well as a QS20 IBM Cell Blade. Rather than hand-tuning LBMHD for each system, we develop a code generator that allows us to 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 15× improvement compared with the original code at a given concurrency. Additionally, we present detailed analysis of each optimization, which reveal surprising hardware bottlenecks and software challenges for future multicore systems and applications.

Key words: Lattice-Boltzmann, Auto-tuning, Multicore, Cell Broadband Engine, Niagara

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 [12, 13]. Our work uses a novel approach to implementing LBMHD across one of the broadest sets of multicore platforms in existing HPC literature, including the conventional homogeneous multicore designs of the dual-socket×quad-core Intel

[*] Corresponding author

Email addresses: SWWilliams@lbl.gov (Samuel Williams), JTCarter@lbl.gov (Jonathan Carter), LOliker@lbl.gov (Leonid Oliker), JShalf@lbl.gov (John Shalf), KAYelick@lbl.gov (Katherine Yelick)

Xeon E5345 (Clovertown), the dual-socket×dual-core AMD Opteron 2214 (Santa Rosa) and the dual-socket×quad-core AMD Opteron 2356 (Barcelona). Additionally, we explore performance on the heterogeneous local-store based architecture of the dual-socket×eight-core IBM Cell QS20 Blade, as well as one of the first scientific studies of the hardware-multithreaded dual-socket×eight-core×eight-thread Sun UltraSparc T5140 T2+ (Victoria Falls) — a Niagara2 SMP. These architectures are described in detail in Section 3.

In Section 4 we introduce the roofline performance model. Although it is quite easy to quantify absolute and relative performance, it can be very difficult to quantify how much further potential performance improvement is possible. We believe the roofline model allows us to effectively quantify how much performance is left on the table. Thus, we can not only quantify absolute performance but also qualify our success. i.e. Good performance leaves less than 15% on the table, where bad performance might leave 50% on the table.

In Section 5 we explore a number of LBMHD optimization strategies that 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 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.

In Section 6 wes show that our auto-tuned optimizations achieve impressive performance gains — attaining up to 15× speedup at full concurrency compared with the original version at full concurrency. 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 IBM 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 [17] 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 [11, 5] 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:

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

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 [27] which uses 27 distinct velocities (including zero velocity) is shown in Figure 1.

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.

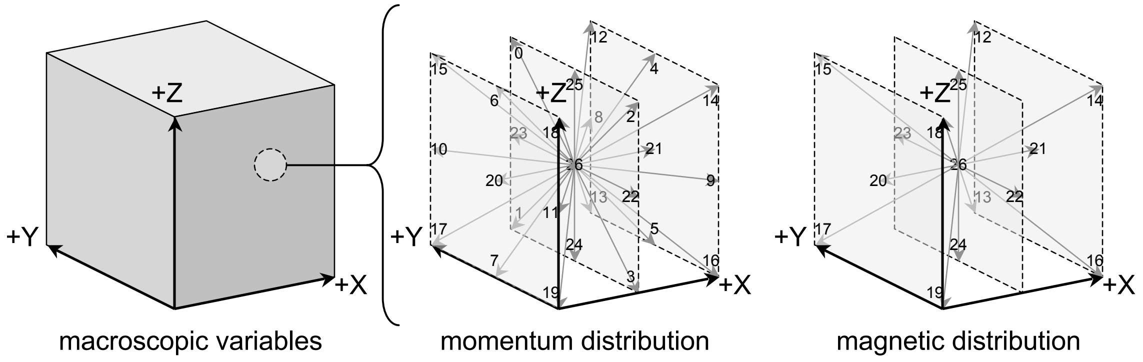


Figure 1: LBMHD simulates magnetohydrodynamics via a lattice boltzmann method using both a momentum and magnetic distribution. Note, where each velocity in the momentum distribution, each velocity in the magnetic distribution is a cartesian vector.

However, a key optimization described by Wellein and co-workers [21] 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

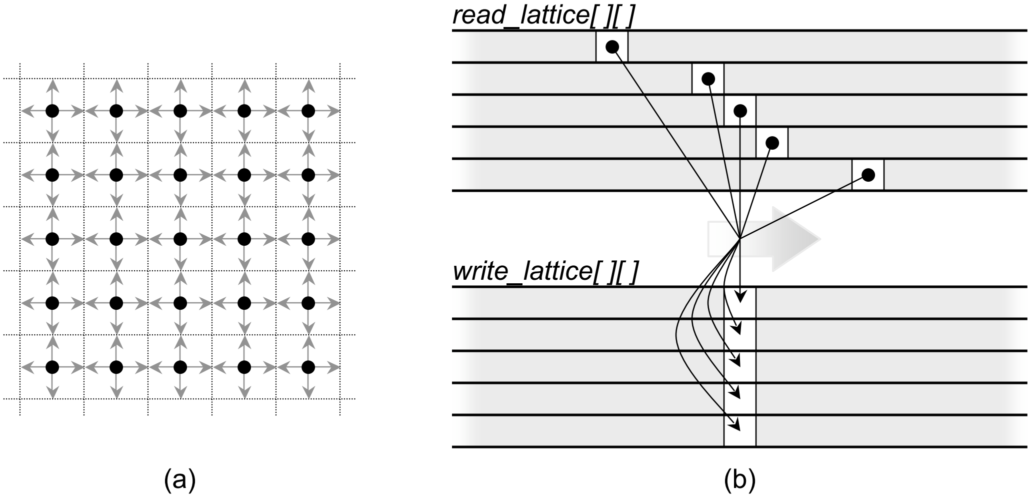


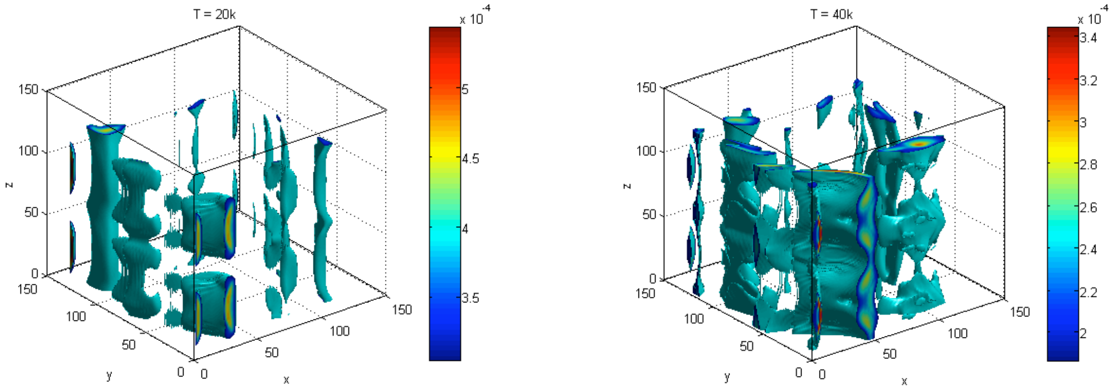
Figure 2: A Simple D2Q5 lattice. (a) Conceptualization of a grid with the lattice distribution superimposed on it as seen in 2D space. (b) mapping of the lattice-stencil from 2D space onto linear array space. The stencil sweeps through the arrays from left to right (unit stride) maintaining the spacing between different phase-space points (rows) in the stencil.

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. Figure 2 illustrates this idea for a simple 2D grid where a phase space component at one lattice site is updated with components from it's four neighbors. Figure 2(b) shows the locations of the different phase-space components in lattices for the current time step and the locations updated as time is advanced. In this formulation, 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 [21, 16], 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 co-workers [28] 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 [10] 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



(a) (b)

Figure 3: Vorticity tubes deforming near the onset of turbulence in LBMHD simulation (a) after 20K iterations, and (b) after 40K iterations.

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. The differing components for particle and magnetic field are shown in Figure 1 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 3 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 [13]. 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 [21], except that we have the flexibility to align the components of distribution functions or macroscopic quantities without the restrictions implicit in a Fortran multidimensional array. To simplify indexing, the unused lattice elements of the magnetic component are simply NULL pointers (see Figure 4).

2.2. *Code Generation and Auto-Tuning*

To optimize LBMHD across a variety of multicore architectures, we employ the auto-tuning methodology exemplified by libraries like ATLAS [22] and OSKI [20]. We created a code generator that be configured to utilize many of the optimizations described in Section 5 including: blocking for the TLB, unrolling depth, instruction reordering, bypassing the cache, software prefetching, and explicit SIMDization. This code generator, written in Perl, 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. It should be noted that this Perl-based generator produces code solely for LBMHD. The code generator would have to be modified to implement a different lattice method. Given our level of expertise and knowledge, the existing LBMHD code generators (C, SSE, Double Hummer) each comprise less than 600 lines of Perl and each took only about a day to write. Given this is probably atypical, future work is endeavoring to make auto-tuners automatic rather than simply automated.

We use POSIX Threads API [19] to implement parallelism on the conventional microprocessor-based platforms and libspe 1.0 to launch the parallel computations on the Cell SPEs. Unlike OpenMP [14], these threading libraries afford us with close to metal control of these processors. This ensures that our results and analysis are not perturbed or clouded by the inefficiencies of a threading library.

In all variants, we use the rather conventional data structure shown in Figure 4. Future work will incorporate data structure exploration into the auto-tuning process.

For the *collision()* operation this process can generate about sixty variations of the kernel that are in turn 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 30 minute tuning benchmark is run offline to exhaustively search parameters of the optimization space; to reduce tuning overhead, the search space is pruned (telescoping vector length) to eliminate optimization parameters unlikely to improve performance. Nevertheless, on the Opterons, roughly one thousand experiments still must be performed. Our experience suggests the time required for tuning can be substantially reduced and future work will explore this. For each optimization, we measured average performance on ten iterations and report the best overall time for any configuration.

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

Figure 4: LBMHD data structure for each time step, where each pointer refers to a N^3 grid.

Core	Intel	AMD	AMD	Sun	IBM	
Architecture	Core2	Santa Rosa	Barcelona	Niagara2	PPE	SPE
Type	superscalar out-of-order	superscalar out-of-order	superscalar out-of-order	MT dual issue[†]	MT dual issue	SIMD dual issue
Clock (GHz)	2.33	2.20	2.30	1.16	3.20	3.20
DP GFlop/s	9.33	4.40	9.20	1.16	6.4	1.8
Local Store	—	—	—	—	—	256KB
L1 Data Cache per core	32KB	64KB	64KB	8KB	32KB	—
L2 Cache per core	—	1MB	512KB	—	512KB	—
L1 TLB entries	16	32	32	128	1024	256
Default Page Size	4KB	4KB	4KB	4MB	4KB	4KB

System	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	UltraSparc T5140 T2+ (Victoria Falls)	QS20 Cell Blade	
# Sockets	2	2	2	2	2	
Cores/Socket	4	2	4	8	1	8
L2/L3 Caches (shared among cores)	4×4MB (shared by 2)	—	2×2MB (shared by 4)	2×4MB (shared by 8)	—	—
max problem size w/out TLB capacity misses	3^3	4^3	4^3	76^3	N/A	
DP GFlop/s	74.66	17.6	73.6	18.7	12.8	29
DRAM Capacity	16GB	16GB	16GB	32GB	1GB	

DRAM Bandwidth (GB/s)	21.33(read)
	10.66(write)
		21.33	21.33	42.66(read)
				21.33(write)
					51.2	

DP Flop:Byte Ratio	2.33	0.83	3.45	0.29	0.25	0.57
System Power (Watts)[§]	330	300	350	610	285[‡]	
Threading	Pthreads	Pthreads	Pthreads	Pthreads	Pthreads	libspe1.0
Compiler	icc 10.0	gcc 4.1.2	gcc 4.1.2	gcc 4.0.4	xlc 8.2	xlc 8.2

Table 1: Architectural summary of evaluated platforms. Top: per core characteristics. Bottom: SMP characteristics. [†]Each of the two thread groups may issue up to one instruction. [§]All system power is measured with a digital power meter while under a full computational load. [‡]Cell BladeCenter power running SGEMM averaged per blade.

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 Setup

Our work examines several leading multicore system designs in the context of the full LBMHD application. Our architecture suite consists of the conventional homogeneous multicore designs of the dual-socket×quad-core Intel Xeon E5345 (Clovertown), the dual-socket×dual-core AMD Opteron 2214 (Santa Rosa) and the dual-socket×quad-core AMD Opteron 2356 (Barcelona). Additionally, we include the heterogeneous local-store based dual-socket×eight-core IBM Cell QS20 Blade, and the hardware-multithreaded dual-socket×eight-core×eight-thread Sun UltraSparc T5140 T2+ (Victoria Falls) — a Niagara2 SMP. An architectural overview and characteristics appear in Table 1 and Figure 5. Note, 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.

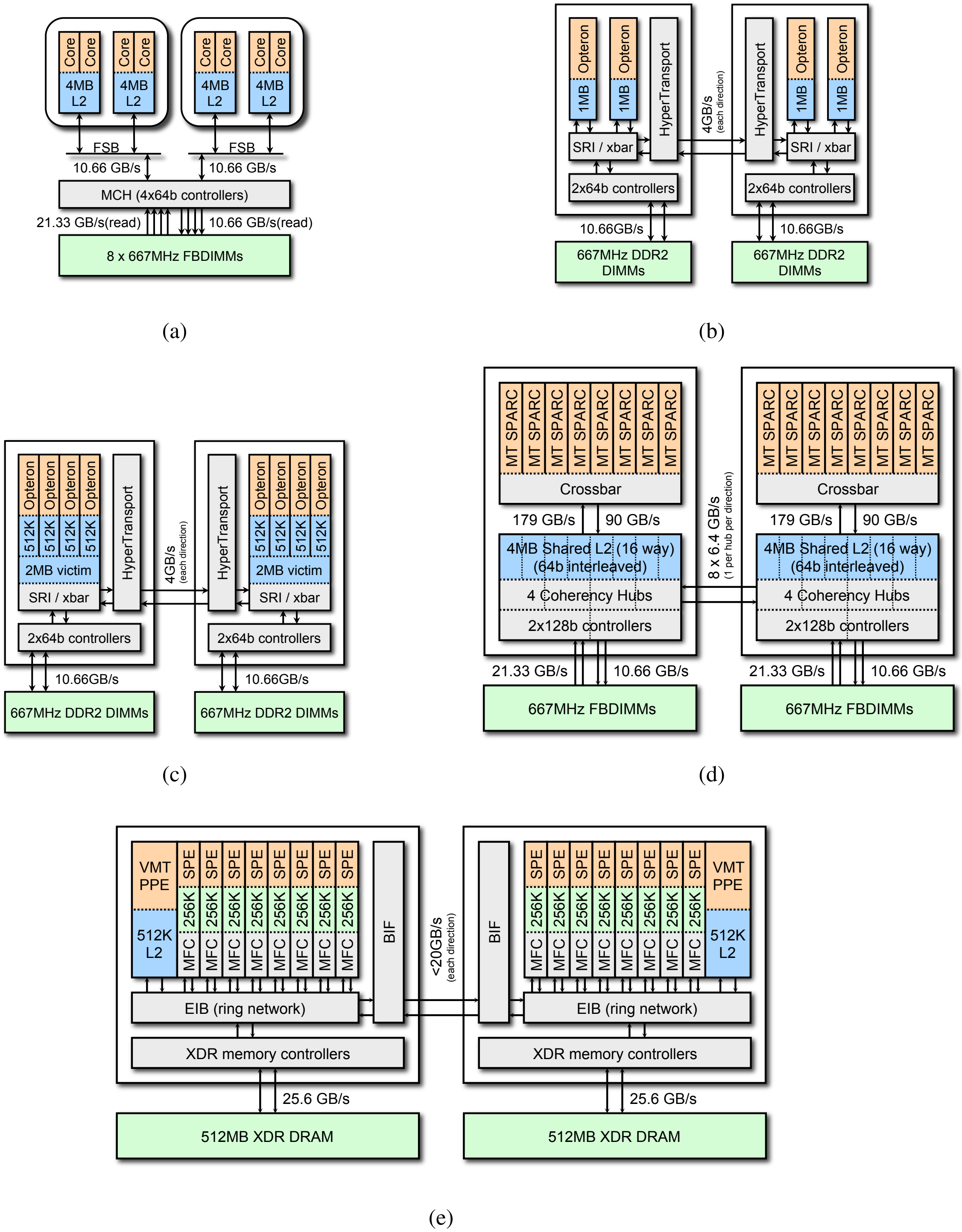


Figure 5: Architectural overview of (a) dual-socket×quad-core Intel Xeon E5345 (Clovertown), (b) dual-socket×dual-core AMD Opteron 2214 (Santa Rosa), (c) dual-socket×quad-core AMD Opteron 2214 (Santa Rosa), (d) dual-socket×eight-core Sun UltraSparc T5140 T2+ (Victoria Falls) (e) dual-socket×eight-core IBM QS20 Cell Blade.

3.1. Intel Xeon E5345 (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. 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 Opterons, 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.

3.2. AMD Opteron 2214 (Santa Rosa)

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.

3.3. AMD Opteron 2356 (Barcelona)

The Opteron 2356 (Barcelona) is AMD's newest quad-core processor offering. Each core operates at 2.3 GHz, can fetch and decode four x86 instructions per cycle, execute 6 micro-ops per cycle and fully support 128b SSE instructions, for peak double-precision performance of 9.2 GFlop/s per core or 36.8 GFlop/s per socket.

Each Opteron core contains a 64KB L1 cache, and a 512MB L2 victim cache. In addition, each chip instantiates a 2MB L3 victim cache shared among all four cores. All core prefetched data is placed in the L1 cache of the requesting core, whereas all DRAM prefetched data is placed into a dedicated buffer. Each socket includes two DDR2-667 memory controllers and a single cache-coherent HyperTransport (HT) link to access the other socket's cache and memory; thus delivering 10.66 GB/s per socket, for an aggregate non-uniform memory access (NUMA) memory bandwidth of 21.33 GB/s for the quad-core, dual-socket system examined in our study. Non-uniformity arises from the fact that DRAM is directly attached to each processor. Thus, access to DRAM attached to the other socket comes at the price of lower bandwidth and higher latency. The DRAM capacity of the tested configuration is 16 GB.

3.4. Sun Victoria Falls

The Sun "UltraSparc T2 Plus" dual-socket × 8-core processor, referred to as Victoria Falls, presents an interesting departure from mainstream multicore chip design. Rather than depending on four-way superscalar execution, each of the 16 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 a single architectural paradigm: fine-grained multithreading.

Victoria Falls instantiates one floating-point unit (FPU) per core (shared among 8 threads). Our study examines the Sun UltraSparc T5140 with two T2+ processors operating at 1.16 GHz, with a per-core and per-socket peak performance of 1.16 GFlop/s and 9.33 GFlop/s, respectively — no fused-multiply add (FMA) functionality. Each core has access to its own private 8KB write-through L1 cache, but is connected to a shared 4MB L2 cache via a 149 GB/s(read) on-chip crossbar switch. Each of the two sockets is fed by two dual channel 667 MHz FBDIMM memory controllers that deliver an aggregate bandwidth of 32 GB/s (21.33 GB/s for reads, and 10.66 GB/s for writes) to each L2, and a total DRAM capacity of 32 GB. Victoria Falls 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. IBM Cell Broadband Engine

The IBM 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 systems, the Cell blade is expected show strong variations in sustained bandwidth on memory bound applications if NUMA is not properly exploited.

3.6. Performance Measurement

In this work we examine two problem sizes: 64^3 and 128^3. 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 much of the limited DRAM available on the Cell blade, while 128^3 problem size fills a substantial 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. This tends to reduce the reported flop rates as no flops are performed during *stream()*.

In addition to problem size, we vary the thread-level concurrency per SMP in powers of two. To aide in architectural analysis, we use affinity routines to pin threads to fill a core, then a socket, then only the SMP. As all SMPs used in this work are dual-socket SMPs, the last data point is the first data point to use the second socket. Moreover, due to performance issues, we never run with less than eight threads on one core of Victoria Falls. Although there are no MPI calls, *stream()* still copies data to and from the MPI buffers.

The number of lattice updates in *collision()* is simply the product of the domain dimensions, and the number of floating-point operations per lattice update is 1300. We count the one divide as one floating-point operation. For each optimization configuration, we run ten iterations of the *collision()*/*stream()* inner loop and measure the total time using the architecture's cycle-accurate counters. We measure performance (GFlop/s) as the $10 \times 1300 \times$ the number of points divided by the time for ten iterations.

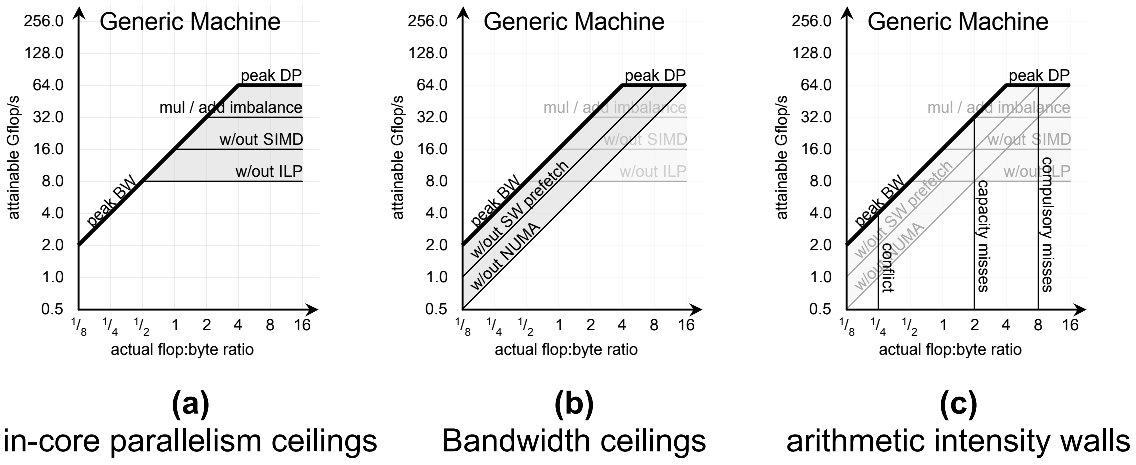
4. Roofline Performance Model

The roofline model, as detailed in [23, 26, 25] is premised on the belief that the three fundamental components of performance on single program multiple data (SPMD) kernels are communication, computation, and locality. In general, communication can be from network, disk, DRAM or even cache. In this work, the communication of interest is from DRAM and the computation of interest is floating point operations. The rates associated with these two quantities are peak bandwidth (GB/s) and peak performance (GFlop/s).

Every kernel has an associated arithmetic intensity (AI) — defined as the ratio of floating point operations to total bytes of communication, in this case DRAM bytes. This is essentially a locality metric as cache misses may significantly increase the compulsory memory traffic. One may naïvely estimate performance as:

$$\text{Attainable GFlop/s} = \min \begin{cases} \text{Peak GFlop/s} \\ \text{Peak GB/s} \times \text{AI} \end{cases} \tag{2}$$

If one were to plot attainable performance as a function of arithmetic intensity, we would see a ramp up in performance followed by a plateau at peak flops much like the sloped rooflines of many houses. Figure 6 shows a roofline



(a) in-core parallelism ceilings

(b) Bandwidth ceilings

(c) arithmetic intensity walls

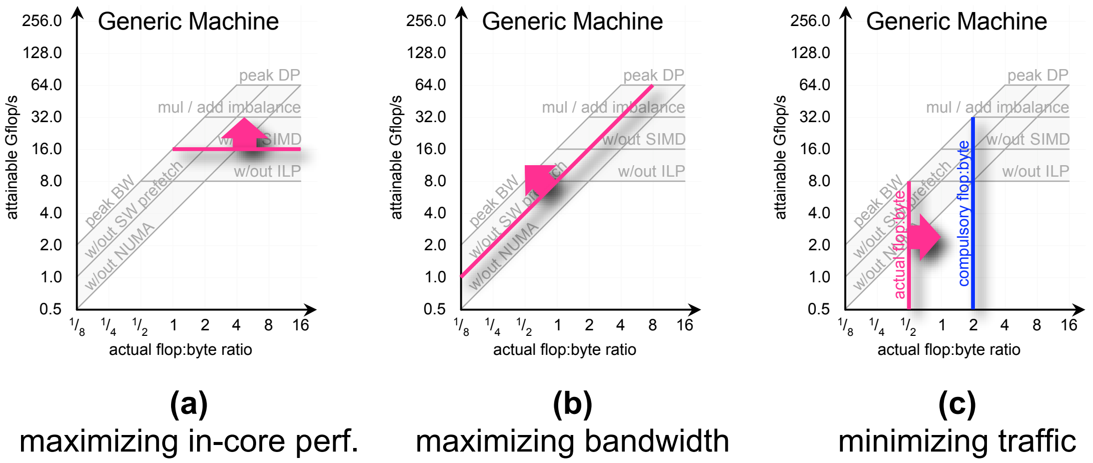
Figure 6: Construction of a DRAM based roofline model for a generic machine. Note the log-log scale. (a) Horizontal ceilings are added for 2-way SIMD and 2 cycle functional unit latency. (b) Diagonal ceilings are added for NUMA bandwidth and the fact that software prefetching is required for peak bandwidth. (c) Arithmetic intensity walls are added for the compulsory limit as well as the potential impact of capacity and conflict misses.

model for a generic machine, in this and all other roofline diagrams we use a log-log scale.

On modern architectures this naïve roofline performance model (guaranteeing either peak flops or peak bandwidth) is wholly inadequate. Achieving peak in-core flops is premised on fully exploiting every architectural innovation of the last 15 years. For example, to achieve peak flops on a AMD Opteron (Barcelona), one must fully exploit all forms of in-core parallelism: balance between the number of multiplies and adds, fully exploiting SSE, unrolling by enough to cover the functional unit latency (4 cycles), and ensuring that loops are sufficiently long that startup overhead is amortized — see Figure 6(a). For architectures bound by instruction fetch bandwidth (e.g. Niagara), attainable performance is dictated by the fraction of code that is floating-point. This metric is essentially the loop balance. Thus, even if the entire code fits in the instruction cache, performance is proportional to the product of instruction bandwidth and the floating point fraction of the dynamic instruction mix. Extrapolating to memory bandwidth, for an Opteron to achieve peak bandwidth, one must ensure that the memory accesses are long unit strides, memory allocation must be NUMA-aware, and software prefetching may be needed — Figure 6(b). Failing to employ even one of these will diminish the attainable performance and form ceilings below the roofline. These ceilings constrain performance to be below them.

As mentioned, each kernel has an associated flop:compulsory byte arithmetic intensity — vertical lines in Figure 6(c). Ideally, all cache misses would be compulsory [8]. Unfortunately, capacity and conflict misses are possible. As their presence increases memory traffic, capacity and conflict misses will significantly reduce arithmetic intensity — creating a true arithmetic intensity to the left of the compulsory arithmetic intensity. For each kernel, we may scan upward along the specified arithmetic intensity line to determine which optimizations are required to achieve better performance — the ceilings that we must punch through.

Figure 7 visualizes three general optimizations — maximizing in-core performance, maximizing memory bandwidth, and minimizing memory traffic — by overlaying them on the roofline model for a generic machine. Improving in-core performance is valuable when performance is not bandwidth limited. Conversely, improving bandwidth or



(a)
maximizing in-core perf.

(b)
maximizing bandwidth

(c)
minimizing traffic

Figure 7: The three general optimizations associated with the roofline model. (a) Maximizing in-core performance is appropriate when performance at a given arithmetic intensity is not bandwidth limited. (b) Maximizing memory bandwidth is valued on this machine for arithmetic intensities less than 16. (c) Improving arithmetic intensity by minimizing conflict and capacity misses can significantly improve performance when bandwidth limited.

increasing arithmetic intensity is valuable when not limited by in-core performance.

We may now construct a roofline model for each microarchitecture explored in this paper, as shown in Figure 8. The model is based on microbenchmarks for bandwidth and optimization manuals for computation. The order of optimizations shown in the roofline diagrams is based on the likelihood of compiler exploitation. For example, fused multiply add (FMA) and multiply/add balance are inherent in linear algebra kernels, but nearly impossible to fully exploit in structured grid codes. Thus, this optimization is the roofline for LBMHD, but would be the lowest ceiling for linear algebra routines. Similarly, compiler unrolling is more likely and easier than SIMDizing a kernel, and NUMA optimizations are an easier optimization to employ than effective manual software prefetching.

Two architectures require some commentary. First, unlike the other architectures, the Clovertown is not a NUMA architecture and does not have a dedicated coherency network. Thus, all coherency traffic appears on the front side bus (FSB) in addition to the data transfers. This can significantly reduce the useful bandwidth. To minimize the coherency traffic, a snoop filter is present within the memory controller hub. It is tasked with tracking which caches may contain certain cache lines. If the request for a cache line appears on one FSB, and it is not in either of the caches on the other FSB, then no coherency traffic needs to be generated. Unfortunately, the snoop filter only tracks a 16MB working set. Thus, when drawing the bandwidth diagonals for the Clovertown, there is no effect from NUMA optimizations, but bandwidth can vary due to the effectiveness of the snoop filter.

Second, Victoria Falls is highly multithreaded, and the SPARC ISA has no double precision SIMD, fused multiply add, or separate add or multiply datapaths. Thus, in-core parallelism is easily hidden. However, the single issue per thread group architecture is very sensitive to loop balance. Thus, when drawing the in-core performance ceilings, we choose to draw loop balance ceilings rather than in-core parallelism ceilings.

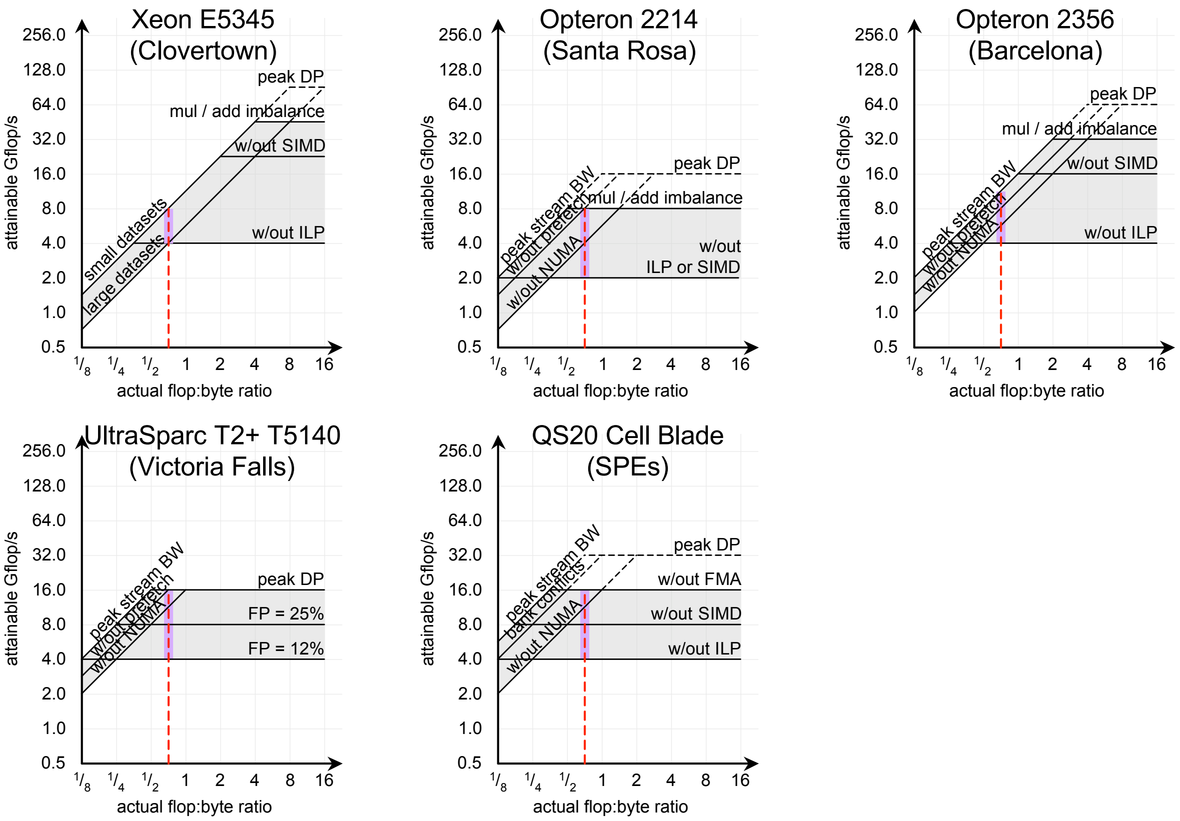


Figure 8: Bandwidth oriented roofline models for the architectures presented here. Note, on Victoria Falls, all forms of in-core parallelism are easily covered, but performance may suffer if floating point instructions don't dominate the instruction mix.

5. 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()* phase 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.

5.1. 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 9). Superficially, the code requires a flop:byte ratio of approximately $2/3$ on conventional cache-based machines to attain peak performance (assuming an allocate on a write-miss cache policy). Consequently, we expect Clovertown and Barcelona to be heavily memory bandwidth limited, while the Victoria Falls and Cell are expected to be heavily 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. That is, we must create code that is computationally efficient as well as code that exploits the available memory bandwidth.

5.1.1. 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. Rather than attempting to decompose and load balance a large grid, our implementation composes the single monolithic larger grid by specifying the subdomain size and the number of subdomains in y, and z. Load balancing is thus implicit. We explore alternate decompositions (number of subdomains in each dimension) 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 or Solaris scheduler's routines for process affinity.

From the perspective of the roofline model, NUMA is a bandwidth diagonal oriented optimization designed to improve the attained memory bandwidth. Thus, we may eliminate the NUMA diagonal as it is implicit in our code. However, the lowest drawn diagonal, unit stride streams, is not. To remove this ceiling we must restructure our code.

```
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 9: Top: original *Collision()* pseudo code. Bottom: vectorized *Collision()* pseudo code

5.1.2. 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 3D-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 capacity 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-register bandwidth, thus favoring shorter vector lengths (VL) that enable all operands to fit within the first level cache. However, shorter VL makes 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 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 capacity per thread. Additionally we search only in full cache lines up to 128 elements then switch to powers of two, up to the calculated maximum VL. This telescoping approach allows for a rapid search over the range of possible vector lengths.

Figure 10 attempts to visualize the two phases of *collision()* before and after vectorization. In the standard implementation (a and b), the neighboring velocities are read one at a time, and the macroscopic quantities are reconstructed. Then, using the newly calculated macroscopics, the lattice update proceeds by updating the velocities for the current point. Clearly, in this approach, although spatial locality may be good, TLB locality can be poor as each velocity will reside on a different page. In the vectorized version (c and d), the velocities are gathered VL at a time to reconstruct the macroscopics for VL points at a time. Thus, VL of the down velocities are accessed, then VL of the left velocities, then VL of the right, and so on through all the directions. Once the macroscopics have been reconstructured, *collision()* proceeds and updates the VL down velocities for the current point, then the VL left velocities, and so on. As VL adjacent points are accessed successively without any intervening accesses, both spatial cache and TLB locality are preserved.

As drawn, the lowest bandwidth diagonal of the roofline model (Figure 8) is associated with a small number of unit-stride streams. In the out-of-the box LBMHD implementation, there are over 150 unit-stride streams per thread — essentially randomly accessing full cache lines. To show this case, one could expand the roofline by drawing random access bandwidth diagonals below the naïve unit stride diagonal. However, vectorization transforms these essentially random accesses into a finite number — less than a dozen — of unit stride streams per thread. Furthermore, as the implementation is NUMA aware, vectorization allows us to punch through two of the bandwidth ceilings.

5.1.3. Loop Unrolling and Reordering

As noted, it is quite possible that some of our architectures will be processor bound. Thus maximizing in-core performance is essential. Given TLB blocking produces vector-style loops, we modified 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. In addition, many compilers will not similarly optimize explicitly SIMDized code. Thus, we

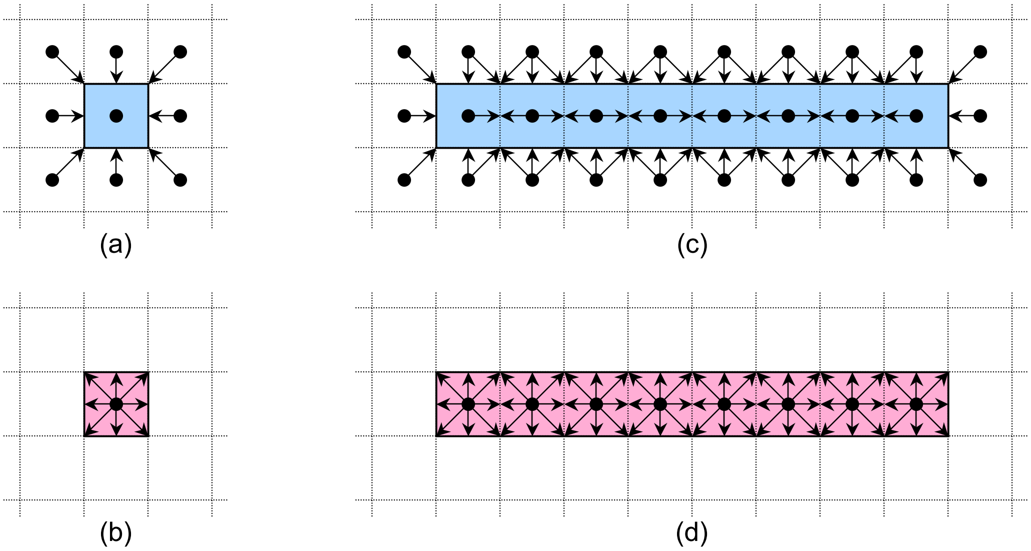


Figure 10: Comparison of traditional (a and b) LBMHD implementation with a vectorized version(c and d). In the first phase of *collision()*(a and c) data is gathered, and the macroscopics are reconstructed. In the second phase (b and d), the local velocity distributions are updated. The difference is that the vectorized version updates VL points at a time.

expect this optimization to become essential later on.

The most naïve 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.

In-core explicit parallelism optimizations such as unrolling and reordering allows one to attempt to exploit instruction-level parallelism, and possibly data-level parallelism for capable compilers. Thus, in the context of the roofline model (Figure 8), at the very least, we've punched through the instruction-level parallelism (ILP), and possibly the data-level parallelism (DLP) ceilings. When combined with the previous optimizations, it is clear that for the out-of-the-box arithmetic intensity, the only remaining ceilings are the ones associated with explicit software prefetching and SIMDization.

5.1.4. Software Prefetching

Our previous work [9, 24] 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 on some architectures is 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.

Examining the roofline model in Figure 8, its clear that LBMHD's arithmetic intensity is sufficiently large — the arithmetic intensity wall intercepts the final in-core parallelism ceiling before intercepting the software prefetch optimized bandwidth ceiling — that on most architectures that software prefetching will be of little value. Nevertheless, we include it in the auto-tuning framework to attain even a small increase in performance.

5.1.5. 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. On x86 architectures, SSE provides an unaligned load that performs this operation with some performance penalty.

We modified the code generator to produce x86 specific SSE2 instrinsics for the vectorized kernels that are unrolled by two or more. Although this seems like a substantial modification, some implementation forethought ensured it would be little more than replacing C operators like '+' with SSE intrinsics like *_mm_add_pd()*. Thus an entire complementary set of optimization variations are produced for x86. The tuning time required to explore this new space was less than doubled as they run much faster.

In the context of the roofline model (Figure 8), explicit SIMDization punches through the final in-core parallelism ceiling — clearly, the imbalance in multiplies and adds ensures that ceiling may never be conquered.

5.1.6. 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 twice the memory traffic as a read, and consumes a cache line in the process. However, if the writes are guaranteed to update the entire cache line without ever reading the original contents, 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.

We modified the SSE code generators by replacing all stores that update the velocities to use the streaming store intrinsic: *_mm_stream_pd()*. As previously mentioned, a little forethought ensured these optimizations were easily implemented.

In Figure 8 we note the value of streaming stores by noting an increased arithmetic intensity (to 1.07) on the relevant architectures. We highlight the resultant performance region — bounded by the out-of-the-box and optimized arithmetic intensities as well as the performance ceilings. Clearly, on many memory bound architectures, performance will slide up and to the right along the uppermost bandwidth ceiling.

5.2. 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, which performs the exchange in three steps involving only six neighbors. The shift algorithm makes use of the fact that after the first exchange between processors along one cartesian coordinate, the ghost cells along the border of the other two directions can be partially populated. The exchange in the next coordinate direction includes this data, further populating the ghost cells, and so on. Palmer and Nieplocha provide a recent summary [15] of the shift algorithm and compare the trade-offs with explicitly exchanging data with all neighbors using different parallel programming models.

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.

5.2.1. Thread-Based Parallelization

Although the *stream()* routine typically contributes little to the overall out-of-the-box execution time, non-parallelized code fragments can become painfully apparent on architectures such as Victoria Falls that have low serial performance (Amdahl's law). In addition, order of magnitude reductions in the time spent in *collision()* can drastically increase the fraction of application time spent in *stream()*. 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. In the future, we believe even more optimization work will be necessary to keep *stream()* a minor part in overall application runtime.

	64^3				
System	Intel Xeon E5345 (Clovertown)	AMD Opteron 2214 (Santa Rosa)	AMD Opteron 2356 (Barcelona)	Sun T5140 T2+ (Victoria Falls)	IBM QS20 Cell Blade
GFlop/s (% of peak)	5.1 (6.8%)	6.7 (38%)	12.4 (17%)	9.8 (53%)	16.7 (57%)
DRAM GB/s (% of peak)	5.5 (17%)	7.2 (34%)	13.3 (62%)	13.9 (22%)	16.7 (33%)
Auto-Tuned Vector Length	64	112	104	16	64[†]
Auto-Tuned Unrolling/Reordering	8/8	8/8	8/4	4/2	2/2[†]
SIMDized	✓	✓	✓	N/A	✓
% Time in *stream()*	12.8%	15.6%	17.5%	7.6%	N/A

	128^3				
System	Intel Xeon E5345 (Clovertown)	AMD Opteron 2214 (Santa Rosa)	AMD Opteron 2356 (Barcelona)	Sun T5140 T2+ (Victoria Falls)	IBM QS20 Cell Blade[‡]
GFlop/s (% of peak)	5.6 (7.5%)	7.4 (42%)	14.1 (19%)	10.5 (56%)	-
DRAM GB/s (% of peak)	6.0 (19%)	7.9 (37%)	15.1 (71%)	14.9 (23%)	-
Auto-Tuned Vector Length	128	128	120	24	-
Auto-Tuned Unrolling/Reordering	8/8	8/8	8/4	4/1	-
SIMDized	✓	✓	✓	N/A	-
% Time in *stream()*	7.7%	9.0%	10.8%	4.8%	-

Table 2: Full-system LBMHD optimized performance characteristics, including the auto-tuned vector length and unrolling/reordering values for both the 64^3 (top) and 128^3 (bottom) problems. DRAM bandwidth is a lower bound based on compulsory memory traffic and excludes coherency traffic. [†]The Cell code is hand optimized, not auto-tuned. [‡]Cell blade has insufficient DRAM capacity to simulate the 128^3 problem.

6. Performance Results and Analysis

Performance on each platform is shown in Figures 11(a–e), using the 64^3 and 128^3 problem sizes for varying levels of thread concurrency.

The stacked bar graphs in Figures 11(a–d), show LBMHD performance contributions in GFlop/s of varying optimizations (where applicable), including from bottom up: the original version (blue), lattice-aware padding (red), auto-tuned TLB blocking (yellow), auto-tuned unrolling/reordering (green), explicit prefetching (purple), explicit SIMDization and cache bypass (orange), and finally smaller pages (maroon). Observe that the prefetching and unrolling bars are rarely seen, as prefetching is of lesser value after vectorization, and compilers are capable of unrolling. Additionally, Figure 11(e) shows performance of the PPE optimized version and a unified, SPE-specific implementation, which includes TLB blocking, DMA transfers, and SIMDization (brown). An overview of the full-system per-core performance, highlighting the variation in scaling behavior can be found in Figure 11(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.

6.1. Intel Xeon E5345 (Clovertown)

The data in Figure 11(b) clearly indicates that TLB blocking benefits Clovertown less as as the number of cores increase. At full concurrency, lattice-aware padding improved performance by 1.3×. In addition, the auto-tuned VL

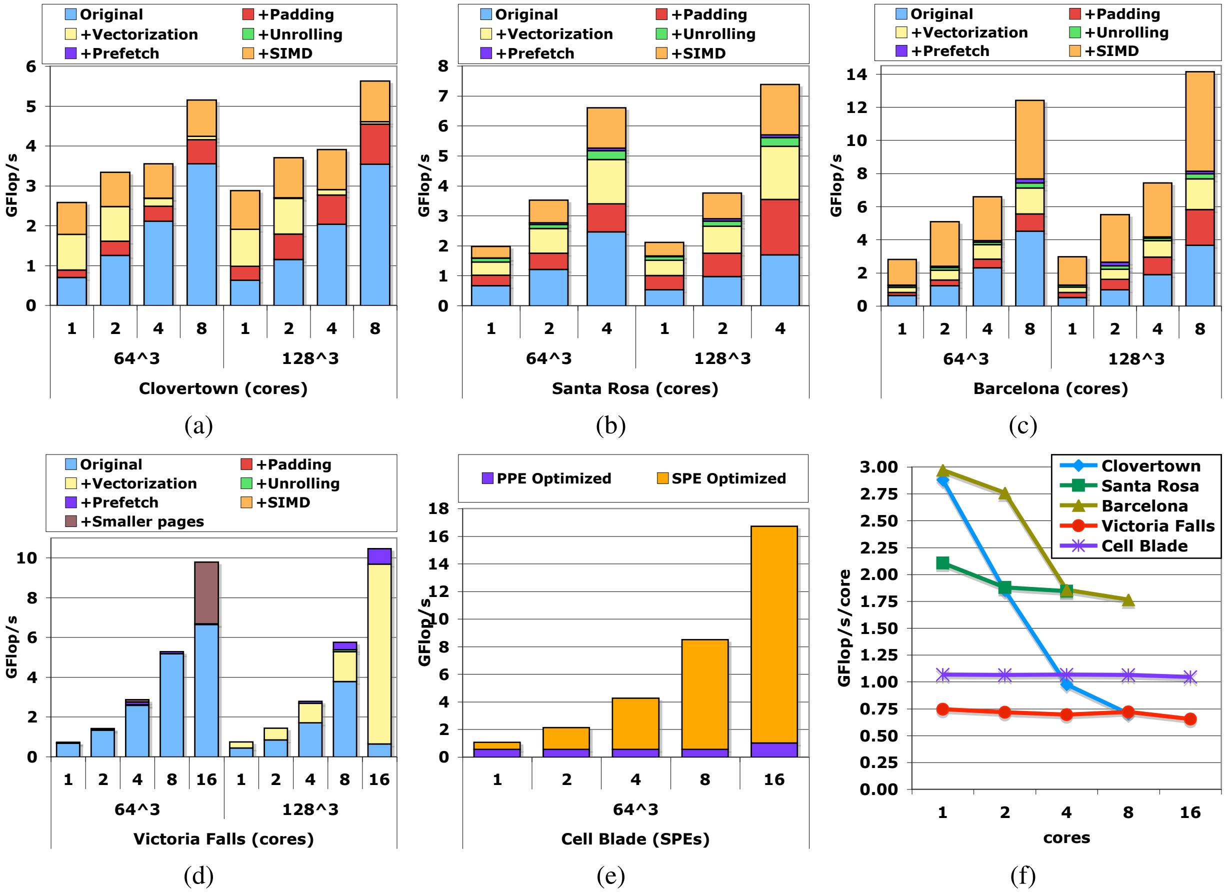


Figure 11: Contributions of explored optimizations of LBMHD on an (a) Intel Xeon E5345 (Clovertown) (b) AMD Opteron 2214 (Santa Rosa) (c) AMD Opteron 2356 (Barcelona) (d) Sun T5140 T2+ (Victoria Falls), as well as (e) performance of the Cell-specific implementation on a IBM QS20 Cell blade. Full-system per-core performance across all platforms is shown in (f).

(seen in Table 2) is 128 points — corresponding to a cache working set size of 80KB when using cache bypass. A VL of 128 only constitutes 25% of a TLB page. Thus it is evident that the L1 cache effect is significant 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 clearly benefits from SIMDization primarily thru the use of streaming stores. 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 closing in on the practical limits of a single FSB [24]. The quad-pumping 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 11(f), suggests the chipset's capabilities limit multi-socket scaling on memory intensive applications.

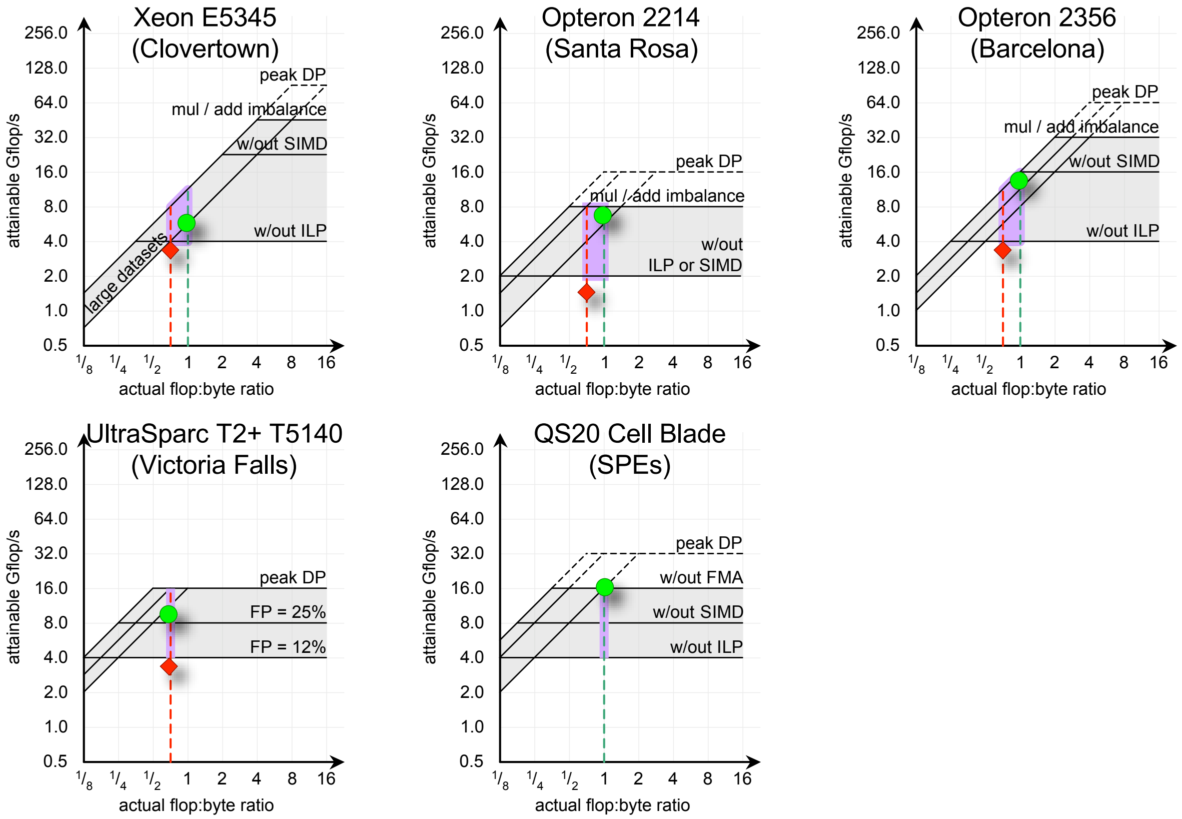


Figure 12: Actual LBMHD performance imposed over a roofline model of LBMHD. Note the lowest bandwidth diagonal assumes a unit-stride access pattern. Red diamonds denote untuned out-of-the-box performance, where green circles mark fully tuned performance for the largest problem attempted. Note, on Victoria Falls, in-core parallelism is easily satisfied, but performance may suffer if floating point instructions don't dominate the instruction mix.

Figure 12 overlays LBMHD performance before (red diamond) and after (green dot) tuning onto a roofline model for the Clovertown. Recall from Section 4, the Clovertown has very good prefetchers, no NUMA issues, but a front side bus on which bandwidth is shared with coherency traffic. To avoid superfluous snoop traffic, a snoop filter is used. The result of using a snoop filter is that bandwidth is dependent on problem size. As a result, the Clovertown bandwidth rooflines are bounded by an effective snoop filter(high roofline) and an ineffective snoop filter(low roofline). As LBMHD has huge streaming working sets, the snoop filter is ineffective. Thus, Clovertown is typically bound to the lowermost bandwidth diagonal. Furthermore, the expected range in performance must only exploit a paltry degree of in-core parallelism to reach the bandwidth ceiling. It is clear why array padding and streaming stores were the only optimizations of value — they increase arithmetic intensity.

6.2. AMD Opteron 2214 (Santa Rosa)

The performance data in Figure 11(c) shows that lattice-aware padding improved out-of-the box performance by 2.1×. The TLB blocking and SIMDization optimizations further increased Opteron performance by a further 2.1× — a total of 4.4×. Similar to the Clovertown, the auto-tuner chose a vector length for the 128^3 problem that corresponds to a working set slightly larger than the 64KB L1 cache, while the optimal unrolling also matches the cache line size

— indicating the limitations of the *gcc* compiler. Virtually no benefit was provided from explicit software prefetching.

Results also show that the Opteron only consumes 2 GB/s per core (Table 2) for the optimized LBMHD algorithm. Thus memory bandwidth is likely not an impediment to performance, allowing this Opteron to achieve nearly linear scaling for both the multicore and multi-socket experiments, as seen in Figure 11(f).

In the context of the roofline model in Figure 12, the first optimization — array padding — improves the arithmetic intensity to close to 0.7, with a bandwidth less than half of the NUMA optimized, unit-stride number. Vectorization further increases performance — nearly reaching both the NUMA optimized unit-stride ceiling as well as the final in-core performance ceiling. The combination of explicit SIMDization with streaming stores slightly improved performance, but as suggested by the roofline model, performance has nearly reached the algorithmic limit defined by the multiply/add imbalance ceiling.

6.3. AMD Opteron 2356 (Barcelona)

Barcelona not only doubles the number of cores but also doubles the per-core peak flops compared to the Santa Rosa Opteron. This will likely provide enough flops to ensure LBMHD is memory bound. When examining performance, we see lattice-aware padding improved performance by 1.6×, but vectorization by only a further 1.3×. Disturbingly, explicit SIMDization and cache bypass provided an additional 1.7×. Thus, without this significant user work, nearly half the performance would have been lost. Interestingly, Barcelona spends nearly 11% of its total runtime in *stream()* — more than any other cache-based architecture. This is because *collision()* saw much better speedups when optimized as compared to the other architectures. A corollary of this is that the *collision()*-only performance is about 12% higher than the overall LBMHD performance. It is interesting to note that the auto-tuner selected different vector lengths and reorderings for the two Opterons perhaps due to differences in the cache and core microarchitectures.

After array padding and vectorization, the Barcelona performance is right on the NUMA aware unit stride bandwidth diagonal of Figure 12. The combination of software prefetching, SIMDization and cache bypass improves performance to the point where it is ultimately bandwidth limited — the green dot is close to the upper bandwidth diagonal.

6.4. Sun T5140 T2+ (Victoria Falls)

The Victoria Falls experiments in Figure 11(d) show several interesting trends. Note, there are always 8 threads per core. For the small (64^3) problem size, almost no benefit is seen from our optimization strategies because, the entire problem can be mapped by each core's 128-entry TLB, when using Solaris' default 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 40% on a single socket. When using the second socket, the limited associativity of the cache was exposed and out-of-the box performance plummeted. Auto-tuning finds a vector length that avoids this pitfall and improves performance at full concurrency by 15×.

Since the shared L1 cache is only 8KB, L1 capacity misses are guaranteed and L1 conflicts are a secondary concern. The L2 cache can only hold vectors of 53 points per thread — far less than the full locality. Nevertheless, auto-tuning found a balance between L1 misses and TLB locality with a vector length of 24 elements while unrolling by 4 without reordering (see Table 2). Clearly cache misses are weighted more heavily than TLB misses.

Performance (as expected) is extremely low for a single thread and is not shown. Performance increases by 7.7× when using all 64 threads of a chip compared to using eight threads within a single core (multicore scaling), and another 1.8× when using the 64 threads on the second socket (multisocket scaling) — the near linear scaling can be seen in Figure 11(f). As a result, performance on the large problem size achieve an impressive 56% of peak while utilizing only 16% of the available memory bandwidth. This suggests further multicore scaling can easily be exploited on future systems of this kind.

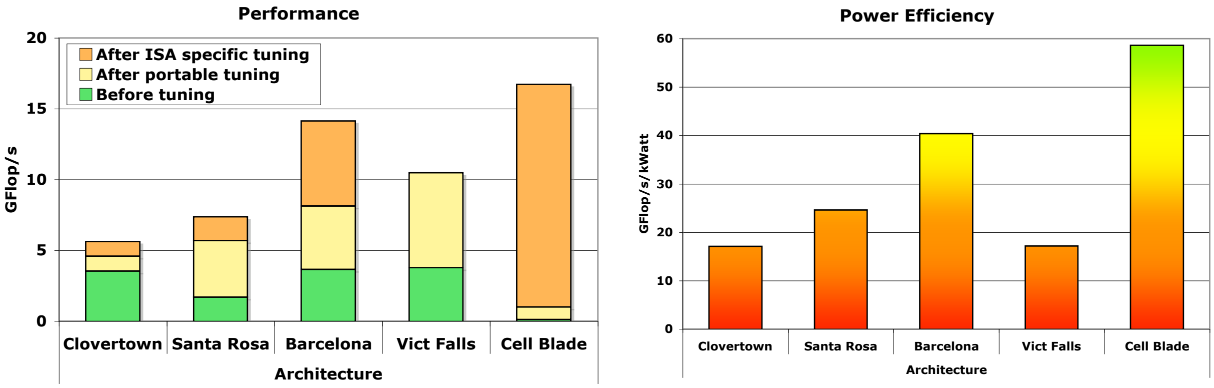
It should be noted that Solaris' default 4MB pages can be a hindrance as they are allocated in their entirety onto one socket or the other. As a result, if the concurrency calls for array parallelization on granularities finer than 4MB, NUMA effects will be present. To quantify this effect, we overrode the default page size and used 64KB pages on the heap — maroon bar in Figure 11(d). Clearly, the effect is seen only on the smaller pages when using the second socket. A 1.5× increase in performance should not be overlooked when it required nothing more than how the program is specified on the command line.

Figure 12 presents a roofline model for Victoria Falls running the 128^3 LBMHD problem. Correlating with the scalability seen in Figure 11(d), it is clear that the red diamond corresponding to out-of-the-box performance only represents one socket. However, as drawn, the roofline represents the full capabilities of the dual-socket SMP. Thus, one should expect performance to be less than half the roofline performance. Vectorization allowed better scalability, however vectorized LBMHD load balance is relatively low. As a result it is extremely likely far less than 50% of the dynamic instruction mix are floating-point instructions. We performed several small experiments on a higher frequency (1.4GHz) Victoria Falls with exactly the same bandwidth. We saw linear increases in performance adding credibility to our postulation that instruction fetch bandwidth was the limiting factor.

6.5. IBM QS20 Cell Blade

Figure 11(e) shows both the auto-tuned cache implementation running on the PPEs as well as the local store implementation running on the SPEs. Note, there is no auto-tuning performed on the SPEs, and the performance using 4 PPE threads is only shown under the 16 SPE (two socket) bar.

Clearly, we see that a single SPE is faster than a fully threaded and auto-tuned PPE core, and also that the 16 SPEs are about 16× faster than the two PPEs each with two threads. Cell achieves near perfect linear scaling across the 16 threads, as evident from Figure 11(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



(a) (b)

Figure 13: Comparison of (a) runtime performance and (b) power efficiency across all studied architectures for the 64^3 problem. ISA specific tuning includes explicitly SIMDized, and local store versions.

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.

NUMA optimizations were not implemented on Cell. When examining the Cell roofline model in Figure 12, it was clear that at an arithmetic intensity of 1.0, Cell is limited by in-core performance — i.e. the required bandwidth to achieve peak in-core performance at an arithmetic intensity of 1.0 is comparable to the no-NUMA unit-stride bandwidth. As future enhanced double precision implementations will raise the in-core performance ceilings by a factor of 7×, but leave the bandwidth ceilings intact, it is obvious that at an arithmetic intensity of 1.0, the requisite bandwidth will be significantly greater than the no-NUMA unit-stride bandwidth. This implies not only will bandwidth optimizations become necessary, but the code will likely be bandwidth limited.

6.6. Architectural Comparison

Figure 13(a) compares LBMHD performance across our suite of architectures for the largest problem run on each architecture problem at the concurrency delivering the best performance. Note, ISA specific tuning includes explicit SIMDization and the Cell local store implementation. When examining out-of-the box performance (green bar), it is clear the Santa Rosa Opteron is somewhat slower than most architectures, but that the Cell PPE performance is abysmal. Auto-tuning provides a significant speedup across all architectures — 1.6× on Clovertown, 4.4× on Santa Rosa, 3.9× on Barcelona, 2.8× on Victoria Falls, and 130× on the Cell blade (over the PPEs). In terms of raw performance, Cell has lost much of its luster over the last three years, although it will gain a significant boost with the release of the QS22 blades. Nevertheless, Cell still delivers 3.0×, 2.3×, 1.2×, 1.6× speedups compared with Clovertown, Santa Rosa, Barcelona, and Victoria Falls respectively. In today's competitive multicore arms race, a factor of two advantage in performance may be short lived.

Of the microprocessors that use a conventional cache hierarchy and programming model, the Victoria Falls provides the highest productivity as it delivers good performance without relying on hand crafted SSE intrinsics. Although the benefit from ISA specific tuning is relatively small on Clovertown and Santa Rosa, Barcelona gets nearly half its performance from explicit SIMDization. Productivity is even worse on Cell; The local store, explicitly SIMDized, SPE implementation is roughly 15 times faster than the portable tuned version running on the PPEs.

Comparing the x86 architectures, results shows that the dual-core Opteron (Santa Rosa) attains slightly better performance than the quad-core Clovertown, but the quad-core Opteron (Barcelona) achieves better than 2.5× the Clovertown performance. Interestingly, Clovertown and Barcelona provide nearly identical auto-tuned single thread performance. However, as concurrency is scaled up the limitations of the FSB become undeniable.

Figure 13(b) compares LBMHD power efficiency (GFlop/s/kWatt) on our evaluated testbed (see Table 1) — one of the most important considerations in HPC acquisitions today. Results show that the Cell blade leads in power efficiency, attaining an impressive advantage of 3.4×, 2.4×, 1.5×, and 3.4×, compared with the Clovertown, Santa Rosa, Barcelona, and Victoria Falls (respectively) for the full-system experiments. Although the Victoria Falls system attains high LBMHD performance, the eight channels of FBDIMM (with 16 DIMMs) drove sustained power to 610W, causing the power efficiency to fall below the x86 platforms — calling into question the value of FBDIMM in a power conscious industry.

7. 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 the explicit 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 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. However, the Barcelona Opteron was a close competitor. Both architectures required a vectorization technique and explicit SIMDization. Nevertheless, Cell's weak double precision implementation ultimately limits performance, where on Barcelona, SIMDization improved performance to the point where it became memory bound. The myriad of requisite optimizations implies one must sacrifice as much productivity to get good performance on the cache-based machines as one does on Cell.

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 on-chip performance. While prior reseach has shown that these design philosophies offer substantial benefits for peak computational rates [18], 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 15× relative to the original code at maximum concurrency. 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.

8. Acknowledgments

We would like to express our gratitude to Forschungszentrum Jülich for access to their Cell blades as well as Brian Waldecker for access to the Barcelona Opterons. 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. Authors from UCB were also supported by Microsoft (Award #024263), Intel (Award #024894), and by matching funding by U.C. Discovery (Award #DIG07-10227).

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)*, pages 361–366, 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] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[9] 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)*, pages 51–60, 2006.

[10] 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.

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

[12] L. Oliker, R. Biswas, J. Borrill, A. Canning, J. Carter, et al. A performance evaluation of the Cray X1 for scientific applications. In *VECPAR: 6th International Meeting on High Performance Computing for Computational Science*, pages 51–65, Valencia, Spain, June 28-30, 2004.

[13] 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*, page 62, Seattle, WA, 2005.

[14] OpenMP. `http://openmp.org`, 1997.

[15] 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*, pages 192–197, 2002.

[16] 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.

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

[18] 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.

[19] The IEEE and The Open Group. The Open Group Base Specifications Issue 6, 2004.

[20] 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*, page 521530. Institute of Physics Publishing, June 2005.

[21] 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.

[22] 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.

[23] S. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, December 2008.

[24] 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*, pages 1–12, 2007.

[25] S. Williams, D. Patterson, L. Oliker, J. Shalf, and K. Yelick. The roofline model: A pedagogical tool for auto-tuning kernels on multicore architectures. In *IEEE HotChips Symposium on High-Performance Chips (HotChips 2008)*, August 2008.

[26] S. Williams, A. Watterman, and D. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the ACM*, April 2009.

[27] 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.

[28] 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*, pages 179–188, 2006.