# PERI - Auto-tuning Memory Intensive Kernels for Multicore

**Samuel Williams**[⋆†], **Kaushik Datta**[†], **Jonathan Carter**[⋆],
**Leonid Oliker**[⋆†], **John Shalf**[⋆], **Katherine Yelick**[⋆†] , **David Bailey**[⋆]

[⋆]CRD/NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
[†]Computer Science Division, University of California at Berkeley, Berkeley, CA 94720, USA

E-mail: SWWilliams@lbl.gov, kdatta@eecs.berkeley.edu, JTCarter@lbl.gov, LOliker@lbl.gov, JShalf@lbl.gov, KAYelick@lbl.gov, DHBailey@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 Sparse Matrix Vector Multiplication (SpMV), the explicit heat equation PDE on a regular grid (Stencil), and a lattice Boltzmann application (LBMHD). We explore one of the broadest sets of multicore architectures in the HPC literature, including the Intel Xeon Clovertown, AMD Opteron Barcelona, Sun Victoria Falls, and the Sony-Toshiba-IBM (STI) Cell. Rather than hand-tuning each kernel for each system, we develop a code generator for each kernel that allows us to identify a highly optimized version for each platform, while amortizing the human programming effort. Results show that our auto-tuned kernel applications often achieve a better than 4× improvement compared with the original code. Additionally, we analyze a Roofline performance model for each platform to reveal 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]. For the foreseeable 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, application 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 Sparse Matrix Vector Multiplication (SpMV), the explicit heat equation PDE on a regular grid (Stencil), and a Lattice Boltzmann application (LBMHD). Our work uses a novel approach to implementing the computations across one of the broadest sets of multicore platforms in existing HPC literature, including the conventional multicore designs of the dual-socket×quad-core Intel Xeon E5345 (Clovertown) and AMD Opteron 2356 (Barcelona), as well as the hardware multithreaded dual-socket×octal-core Niagara2 — Sun T2+ T5140 (Victoria Falls). In addition, we include the heterogeneous local-store based architecture of the dual-socket×eight-core Sony-Toshiba-IBM (STI) Cell QS20 Blade.

Our work explores a number of important optimization strategies, which we analyze to identify the microarchitecture bottlenecks in each system; this leads to several insights into how to build effective multicore applications, compilers, tools and hardware. In particular, we discover that, although the original code versions

run poorly on all of our superscalar platforms, memory bus bandwidth is often not the limiting factor on most examined systems. Instead, performance is limited by lack of resources for mapping virtual memory pages — limited translation lookaside (TLB) buffer capacity, limited cache capacity and associativity, high memory latency, voluminous cache coherency traffic, 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 the underlying hardware. We therefore create an *auto-tuning* environment for these three codes that searches through a parameter space for a set of optimizations to maximize performance. We believe such application-specific auto-tuners are the most practical near-term approach for obtaining high performance on multicore systems. Additionally, our experience offers concrete challenges to future multicore work on compilers and performance tools.

Results show that our auto-tuned optimizations achieve impressive performance gains — up to 130× speedup compared with the original version. Moreover, our fully optimized implementations sustain the highest fraction of theoretical peak performance on any superscalar platform to date. We also demonstrate that, despite the relatively slow double precision capabilities, the STI Cell provides considerable advantages in terms of raw performance and power efficiency — at the cost of increased programming complexity. Additionally, we present several key insights into the architectural tradeoffs of emerging multicore designs and their implications on scientific algorithm design.

We facilitate the visualization of the limitations to performance through a roofline model. Attained performance and requisite optimizations were well correlated to those suggested by the model. In fact, the model is a much better indicator of performance than either raw peak flops or bandwidth.

## 2. Experimental Testbed

Our experimental testbed consists of a diverse selection of multicore system implementations. A summary of key architectural features of our architectural testbed appears in Table 1 and is visualized in Figure 1. The sustained system power data was obtained using an in-line digital power meter while the node was under a full computational load. Node power under a computational load can vary dramatically from idle power and from the manufacturer's peak power specifications. Although the node architectures are diverse, they accurately represent building-blocks of current and future ultra-scale supercomputing systems. We now present a brief overview of the studied platforms.

### *2.1. Intel Clovertown*

Clovertown is Intel's first foray into the quad-core arena. Reminiscent of Intel's 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, execute 6 micro-ops per cycle, and can fully support 128b SSE, for peak double-precision performance of 10.66 GFlop/s per core.

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 front side bus (FSB), delivering a raw bandwidth of 10.66 GB/s. Our study evaluates 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, with a DRAM capacity of 16GB. The full system has 16MB of L2 cache and an impressive 85.3 GFlop/s peak performance.

### *2.2. AMD 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 the L3. Each socket includes

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

| System | Xeon E5345 (Clovertown) | Opteron 2356 (Barcelona) | UltraSparc T5140 T2+ (Victoria Falls) | QS20 Cell Blade | |
|---|---|---|---|---|---|
| # Sockets | 2 | 2 | 2 | 2 | |
| Cores/Socket | 4 | 4 | 8 | 1 | 8 |
| shared L2/L3 Cache | 4×4MB(shared by 2) | 2×2MB(shared by 4) | 2×4MB(shared by 8) | — | — |
| DP GFlop/s | 74.66 | 73.6 | 18.7 | 12.8 | 29 |
| DRAM Bandwidth (GB/s) | 21.33(read) 10.66(write) | 21.33 | 42.66(read) 21.33(write) | 51.2 | |
| DP Flop:Byte Ratio | 2.33 | 3.45 | 0.29 | 0.25 | 0.57 |
| System Power (Watts)§ | 330 | 350 | 610 | 285‡ | |
| Threading | Pthreads | Pthreads | Pthreads | Pthreads | libspe2.1 |
| Compiler | icc 10.0 | 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.

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.

### *2.3. 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 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

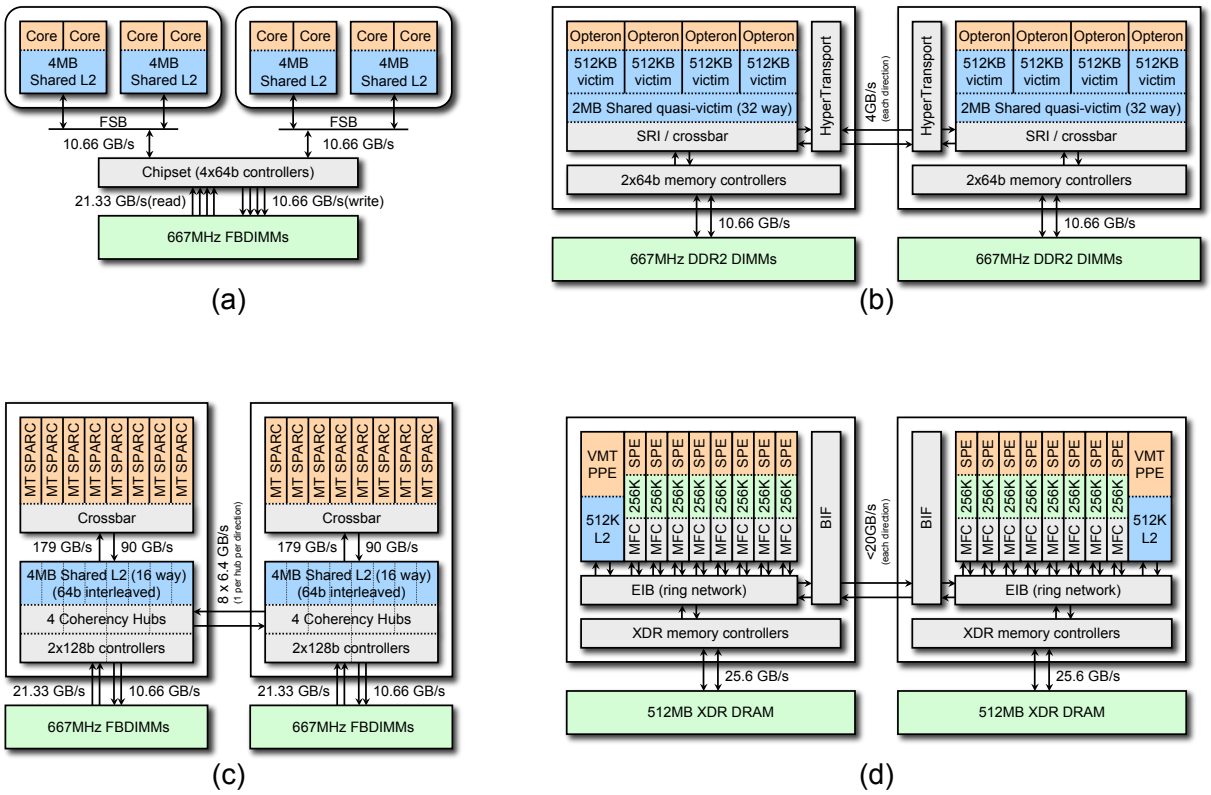



**Figure 1.** Architectural overview of (a) dual-socket×quad-core Intel Xeon E5345 (Clovertown), (b) dual-socket×quad-core AMD Opteron 2356 (Barcelona), (c) dual-socket×eight-core Sun Niagara2 T2+ T5140 (Victoria Falls), (d) dual-socket×eight-core STI QS20 Cell Blade,

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.

### *2.4. STI Cell*

The Sony Toshiba IBM (STI) Cell processor is the heart of the Sony PlayStation 3 (PS3) video game console, whose aggressive design is intended to meet the demanding computational requirements of video games. Cell adopts a heterogeneous approach to multicore, with one conventional processor core (Power Processing Element / PPE) to handle OS and control functions, combined with up to eight simpler single instruction multiple data (SIMD) cores (Synergistic Processing Elements / SPEs) for the computationally intensive work [6].

At first glance, each PPE appears to be a conventional 64b dual-issue, cache-based PowerPC core. In fact, each PPE is 2-way vertically-multithreaded. Thus similar to Victoria Falls, multiple thread contexts are maintained in hardware on each core. On any given cycle the core may fetch and issue up to two instructions from one thread or the other. There are no hardware prefetchers and the cores are in-order. Thus one expects the PPEs to be low performing, especially on memory bound kernels. Thus one should consider their value to be compatibility and productivity, not performance.

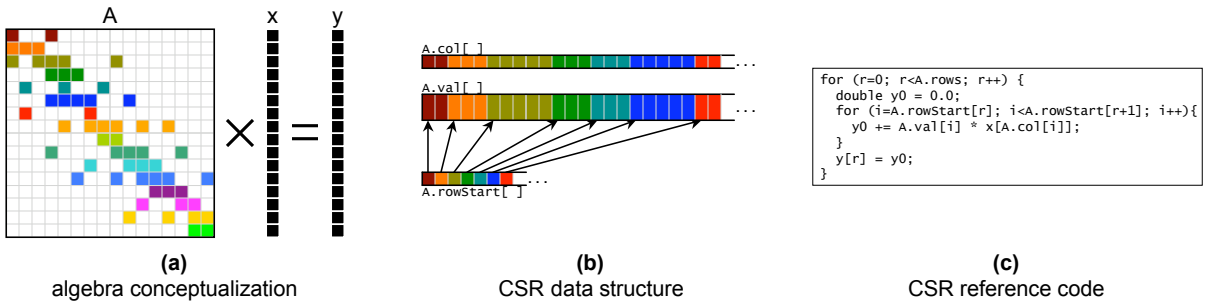



**Figure 2.** Sparse Matrix Vector Multiplication (SpMV). (a) visualization of the algebra: $y \leftarrow Ax$, where $A$ is a sparse matrix. (b) Standard compressed sparse row (CSR) representation of the matrix. This structure of arrays implementation is favored on most architectures. (c) The standard implementation of SpMV for a matrix stored in CSR. The outer loop is trivially parallelized without any data dependencies.

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 that includes a partially pipelined FPU. Although the SPEs can execute two single precision fused-multiply adds (FMAs) per cycle, they can only execute one double-precision FMA SIMD instruction every 7 cycles, for a peak of 1.8 GFlop/s per SPE. This study utilizes the QS20 Cell blade comprised of two sockets with eight SPEs each (29.2 GFlop/s double-precision peak). Each socket has its own dual channel XDR memory controller delivering 25.6 GB/s, with a DRAM capacity of 512 MB per socket (1 GB total). The Cell blade connects the chips with a separate coherent interface delivering up to 20 GB/s, resulting in NUMA characteristics (as with Barcelona and Victoria Falls).

## 3. Kernels and Applications

In this work, we examine the performance of three memory intensive kernels: SpMV, stencils, and LBMHD. They are all $O(1)$ arithmetic intensity, varying from as little as 0.16 to as high as 1.0 flops per byte. We detail each of them in the following sections.

### *3.1. Sparse Matrix Vector Multiplication*

Sparse Matrix Vector Multiplication (SpMV) dominates the performance of diverse applications in scientific and engineering computing, economic modeling and information retrieval; yet, conventional implementations have historically been relatively poor, running at 10% or less of machine peak on single-core cache-based microprocessor systems [11]. Compared to dense linear algebra kernels, sparse kernels suffer from higher instruction and storage overheads per flop, an inherent lack of instruction- and data-level parallelism in the reference implementations, as well as indirect and irregular memory access patterns. Achieving higher performance on these platforms requires choosing a compact data structure and code transformations that best exploit properties of both the sparse matrix — which may be known only at run-time — *and* the underlying machine architecture. This need for optimization and tuning at run-time is a major distinction from both the dense case as well as many structured grid codes. We reuse the suite of 14 matrices used in our SC07 [15] paper. These matrices run the gambit of structure, density, and apsect ratio. None should fit in cache.

We consider the SpMV operation $y \leftarrow Ax$, where $A$ is a sparse matrix, and $x, y$ are dense vectors. The most common data structure used to store a sparse matrix for SpMV-heavy computations is compressed sparse row (CSR) format, illustrated in Figure 2. SpMV has very low arithmetic intensity, performing only 2 flops for

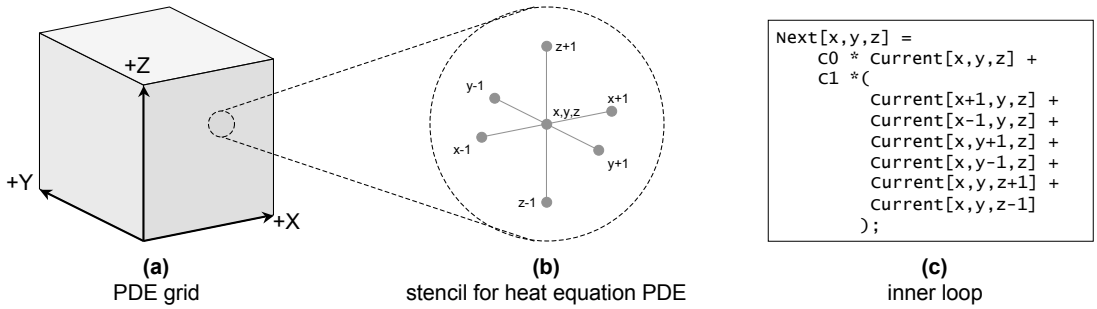



**Figure 3.** Visualization of the data structures associated with the heat equation stencil. (a) the 3D temperature grid. (b) the stencil operator performed at each point in the grid. (c) pseudocode for stencil operator.

every double and integer loaded from DRAM. We define the compulsory miss limited arithmetic intensity to be the ratio of the total number of floating point operations to the compulsory cache misses. True arithmetic intensity includes all cache misses, not just the compulsory ones. With a compulsory miss limited arithmetic intensity of 0.166, one expects most architectures to be heavily memory bound.

For further details on the sparse matrix vector multiplication and our auto-tuning approach, we direct the reader to our SC07 paper [15].

### *3.2. The Heat Equation Stencil*

The second kernel examined in this work is the explicit heat equation PDE applied to a uniform 3D grid. Partial differential equation solvers constitute a large fraction of scientific applications in such diverse areas as heat diffusion, electromagnetics, and fluid dynamics. These applications are often implemented using iterative finite-difference techniques that sweep over a spatial grid, performing nearest neighbor computations called *stencils*. In a stencil operation, each point in a multidimensional grid is updated with weighted contributions from a subset of its neighbors in both time and space — thereby representing the coefficients of the PDE for that data element. In this work, we examine performance of the 3D heat equation, naïvely expressed as triply nested loops over $xyz$ and shown in Figure 3

This seven-point stencil performs a single Jacobi iteration, meaning that the calculation is out-of-place; thus reads and writes occur in two distinct arrays. As a result the compulsory limited arithmetic intensity is 0.33 on write allocate architectures. We use a $256^3$ problem when auto-tuning and benchmarking.

For further details on the heat equation and our auto-tuning approach, we direct the reader to our paper submitted to SC08 [4].

### *3.3. Lattice Boltzmann Magnetohydrodynamics*

The final application studied here is Lattice Boltzmann Magnetohydrodynamics (LBMHD) [8]. LBMHD 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. In 3-dimensional LBMHD, the 27 velocity momentum lattice is augmented by a 15 (cartesian vector) velocity magnetic lattice — shown in Figure 4. This creates tremendous memory capacity requirements — over 1KB per point in space. LBM methods iterate on two phases: a *collision()* operator, where the grid is evolved on each timestep, and a *stream()* operator that exchanges data with neighboring processors. Typically, *collision()* constitutes 90% or more of the total runtime. Thus, although we parallelize *stream()*, we concentrate on auto-tuning *collision()*. On most architectures, we use a $128^3$ problem when auto-tuning and benchmarking. However, this exceeds the Cell Blade's 1GB memory capacity requirements. As such, we run only ran a $64^3$ problem on Cell.

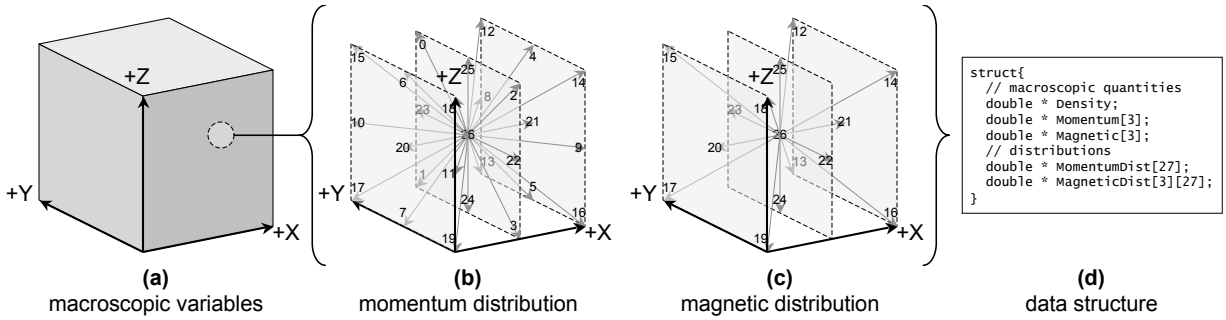



**Figure 4.** Visualization of the datastructures associated with LBMHD. (a) the 3D macroscopic grid. (b) the D3Q27 momentum scalar velocities. (c) D3Q15 magnetic vector velocities. (d) C structure of arrays datastructure. Note, each pointer refers t a $N^3$ grid, and $X$ is the unit stride dimension.

The code is far too complex to duplicate here, although a conceptualization of the lattice method and the data structure itself is shown in Figure 4. Nevertheless, the *collision()* operator must read 73 doubles, write 79 doubles, and perform 1300 floating point operations per lattice update. This results in a compulsory limited arithmetic intensity of about 0.7 on write allocate architectures.

For further details on LBMHD and our auto-tuning approach, we direct the reader to our IPDPS08 paper [14].

## 4. Roofline Performance Model

The roofline model, as detailed in [13, 16, 17] 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 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 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 the $min(peakGFlop/s, AI \times peakGB/s)$. 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 slopped rooflines of many houses.

On modern architectures this naïve Roofline performance model (guaranteed 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 have a balance between the number of multiplies and adds, fully exploit SSE, unroll by enough to cover the functional unit latency (4 cycles), and ensure that loops are sufficiently long that startup overhead is amortized — see Figure 5(a). For instruction bandwidth bound architectures (e.g. Niagara), attainable performance is dictated by the fraction of code that is floating-point. 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 5(b). Failing to employ even one of these will diminish the attainable performance and form ceilings below the roofline.

We may now construct a roofline model for each microarchitecture explored in this paper, as shown in Figure 8 — note the log-log scale. The model is based on microbenchmarks for bandwidth and optimization manuals for computation. The order of optimizations 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 lowest of the computational ceilings for SpMV (from sparse linear algebra), but is the roofline for Stencil and LBMHD. Furthermore,

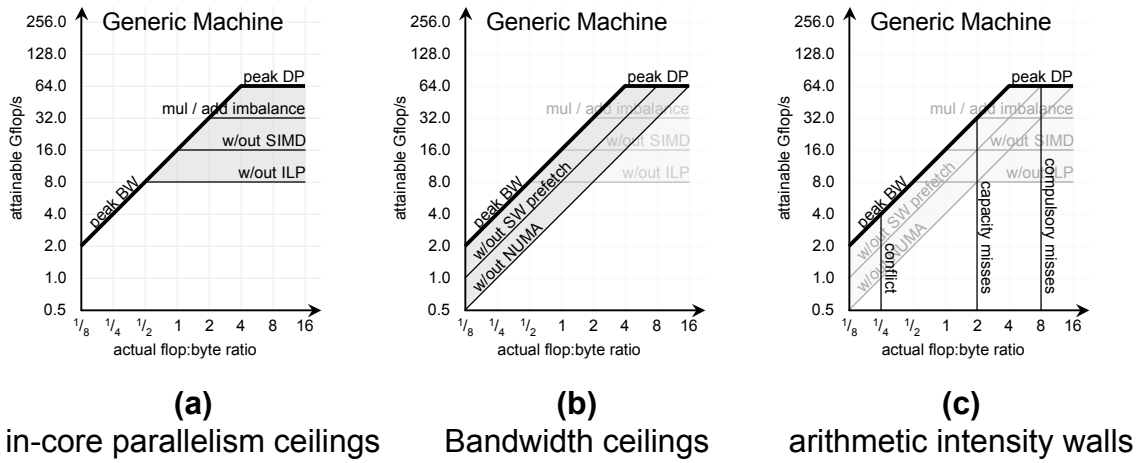



**Figure 5.** Construction of a DRAM based roofline model for a generic machine. Note the log-log scale (a) Horizontal ceilings are added for 2-way SIMD and 2 cycle functional unit latency. (b) Diagonal ceilings are added for NUMA bandwidth and the fact that software prefetching is required for peak bandwidth. (c) Arithmetic intensity walls are added for the compulsory limit as well as the potential impact of capacity and conflict misses.

compiler unrolling is more likely and easier than SIMDizing a kernel. Similarly, NUMA optimizations are an easier optimization to employ than manually software prefetching.

As mentioned, each kernel has an associated flop:compulsory byte arithmetic intensity — vertical lines in Figure 5(c) and Figure 8. This only counts memory traffic (bytes) from compulsory cache misses. Of course, capacity and conflict misses will significantly reduce this 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.

## 5. Optimizations and Auto-tuning

Principally, there are three categories of optimizations that must be performed on these memory intensive SPMD kernels: maximizing memory bandwidth, minimizing memory traffic, and ensuring that there is enough instruction and data level parallelism to keep the machine memory bound.

### *5.1. Auto-tuning and Code Generation Background*

Optimizing compilers are data and algorithm agnostic and rely on heuristics to optimize code. However, it has been shown that not only do search based methods improve both dense linear algebra [2, 12] and spectral methods [5, 9], but data and algorithm foreknowledge allows for complex restructuring of both implementation and data structure [11]. These automatically tuned libraries — auto-tuners — deliver significant performance boosts on existing architectures. They also provide a productive solution to optimizing future architectures.

We know that we will have an extremely large code optimization parameter space to explore. As such, it is imperative to create tools that generate the hundreds of different inner loop variants. We chose to write a parameterized Perl script for each kernel (SpMV, Stencil, LBMHD) capable of producing all the desired variants. The functions are then packed into a pointer to function table that may be indexed by the desired optimizations.

We then create an auto-tuning benchmark for each kernel. It first creates n-1 additional threads that will call a SPMD auto-tuning benchmark functions to create, initialize, and load the datasets. Then, the threads explore

the parameter space by applying optimizations in an order appropriate for the given platform. The optimal configuration and performance of these runs is then reported.

The following sections group all the auto-tuned optimizations into the three basic categories and details each of them.

### *5.2. Threading and Parallelization*

We selected Pthreads as the threading library for all cache-based microarchitectures. The Cell SPEs use IBM's libspe threading library. On top of Pthreads we implement a shared memory SPMD-like model where all threads execute the same function for all iterations of the kernel. We implemented a flat shared memory barrier for each threading library.

For SpMV, we restricted ourselves to a deterministic 1D decomposition of the matrix (by rows). This ensures that no inter-thread reductions are necessary. Generally this approach works well when the number of rows is at least an order of magnitude larger than the number of threads.

For LBMHD, we primarily implement a deterministic 1D decomposition of the grid (into slabs or eventually planes). This is acceptable until we have more threads than planes in $Z$ — the least contiguous dimension. At that point we transition, and begin to parallelize in the $Y$ dimension.

The heat equation employs an even more complex parallelization scheme. Not only does it parallelize in all three dimensions, but it auto-tunes to determine the optimal parallelization in each dimension. We simultaneously search for the best cache blocking and thread parallelization.

### *5.3. Maximizing in-core performance*

We must first attempt to ensure that the kernels expected to be memory-bound are, in fact, bound by memory. Essentially, the optimizations detailed here boost in-core performance by ensuring the data and instruction level parallelism inherent in a kernel is expressed in its implementation. The principal optimizations are unrolling, reordering, and explicit SIMDization via intrinsics. This last optimization is a clear departure from portable C code as it requires an optimized SSE implementation, an optimized SPE implementation, an optimized double hummer implementation, etc... for each SIMD ISA. The out-of-the-box SpMV CSR implementation has no instruction-level parallelism (ILP) or data-level parallelism (DLP). Thus these optimizations could not be applied until register blocking (see below) was applied. Nevertheless, the heat equation and LBMHD (after loop restructuring) have ample ILP and DLP. We auto-tune unrolling and reordering in powers of two to determine how much ILP and DLP should be expressed in the implementation.

### *5.4. Maximizing Memory Bandwidth*

As suggested in the roofline model, many optimizations are required to maximize memory bandwidth. These include limiting the number of memory streams, exploiting the NUMA nature of these machines, and proper use of software and hardware prefetching.

The most basic optimization is to properly collocate data with the threads tasked to process it. We rely on an OS process affinity routine and a first touch policy. We then bind threads to cores and allow each thread, rather than only the master, to initialize its own data. This ensures all memory controllers may be engaged, and minimizes inter-socket communication.

Although SpMV and stencil only require two memory streams per thread, LBMHD generates over 150. In this case, we restructured the loop to avoid TLB misses and also properly engaged the hardware prefetchers [14]. However, this resulted in an unknown quantity — vector length, for which we must auto-tune.

Finally, on some architectures, hardware prefetchers can be easily confused. On others, they don't exist. As a result, we instrumented our code generators to produce variants with software prefetches inserted into the inner loops. Our auto-tuning framework then tunes for the appropriate prefetch distance.

*5.5. Minimizing Memory Traffic*

When we have optimized our code to the point where it is using nearly all of a machine's sustainable memory bandwidth, the only option for improving performance is to reduce total memory traffic. Basically, this boils down to addressing the 3C's [7] of caches. Essentially any cache miss can be categorized as a conflict, capacity, or compulsory miss. Conflict misses arise due to limited associativity in set associative caches, and capacity misses occur due to limited cache capacity — a bigger cache would eliminate them. However, compulsory misses are ever present. Data starts in DRAM. To read it, a cache miss always must occur. In this category we also include data that is hardware prefetched into the cache that would normally generate a compulsory miss if the prefetcher wasn't present.

For all three kernels, the data must start in DRAM, and the result must be returned to DRAM. As a result, for each matrix multiplication or grid sweep we will generate a large number of compulsory misses. We can calculate an upper limit to arithmetic intensity based on the total number of floating point operations and compulsory cache misses. For SpMV, two floating point operations (flops) are performed for every nonzero (a double and an integer) loaded from DRAM. The heat equation stencil must read every point in the grid (one double each), perform the stencil for each (8 flops), and write the updated point back to DRAM (one double for every point in the grid). LBMHD is similar, but the balance is 73 reads, 1300 flops, and 79 writes per point. On write-allocate cache architectures, each write actually generates 16 bytes of memory traffic — 8 bytes of write-miss fill, and 8 bytes of writeback. Thus we expect flop:compulsory byte ratios of 0.33 and 0.7 respectively for the heat equation and LBMHD.

A flop:byte ratio of 0.166 is not the ultimate limit for SpMV. In fact we may exploit register blocking [11] techniques to improve the flop:byte ratio to 0.25 by eliminating redundant column indices. For the structured grid codes we may exploit an x86 instruction — *movntpd* — and force writes to bypass the cache. The result is that the fill memory traffic may be eliminated. For architectures that allow this optimization, the highest flop:byte ratios of 0.5 for stencil and 1.0 for LBMHD can be achieved.

Conflict misses arise due to the relatively limited associativity of the caches. Normally, these misses are small, but when power of two problem sizes are coupled with power of two numbers of threads on shared caches with low associativity, the number of conflict misses skyrockets. For SpMV, we address this by skewing the first element of each thread's piece of the matrix to uniformly spaced cache sets. The heat equation only has two arrays total, shared by all threads. As a result, each 3D array is padded in the unit stride dimension to ensure collectively, all the points of the thread's stencils do not hit the same set. The stencil for LBMHD touches one point in each of 150 separate 3D grids. As a result, we pad each grid to ensure no points within a stencil hits the same set.

Capacity misses occur when the desired working set size exceeds the cache capacity per thread. Depending on how much inherent reuse a kernel has, this will cut the arithmetic intensity in half or more. This can be devastating on memory bound kernels. Each dataset for each kernel has a different capacity requirement, and when coupled with the fact that each architecture has a different cache size, it becomes evident that we must auto-tune each to eliminate as many cache misses as possible. For SpMV, capacity misses occur when the working set of the source vector is too big. To rectify this, we cache block the matrix [15] into sub-matrices known not to touch more data than the cache can hold. A similar technique is applied to the heat equation stencil. Here, to generate solely compulsory misses, we must maintain a working set of 4 planes in the cache. Thus, we auto-tune the loop structure to implicitly block the 3D grid into columns for which this is true. The structure of arrays data structure used in LBMHD was explicitly chosen as it is expected to be free of capacity misses.

*5.6. Summary of Techniques*

Figure 6 visualizes the 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 increasing arithmetic intensity is valuable when not limited by in-core performance.

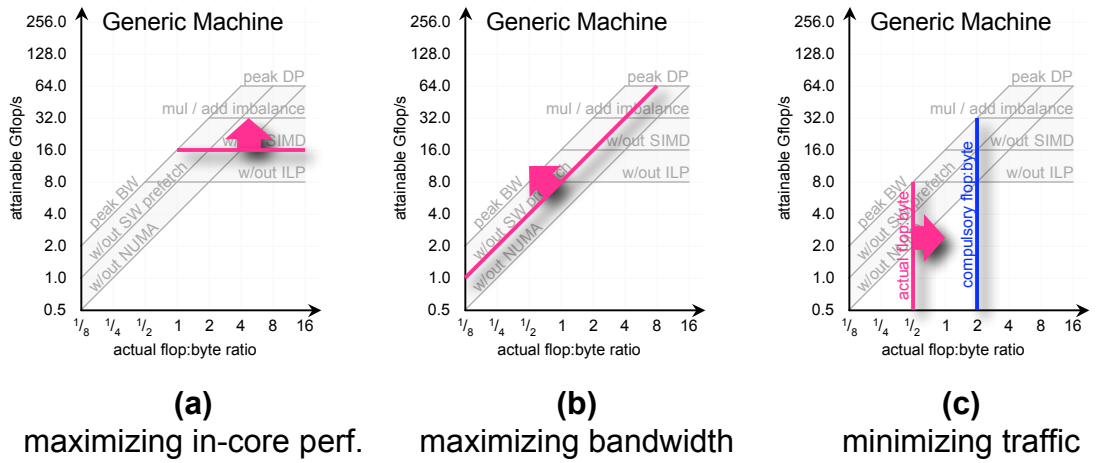


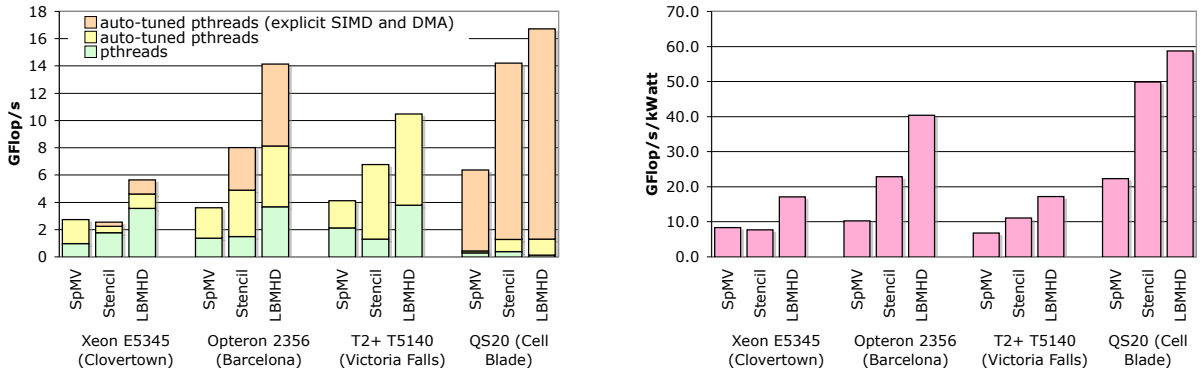

**Figure 6.** The three general optimizations associated with the Roofline model. (a) Maximizing in-core performance is appropriate when performance at a given arithmetic intensity is not bandwidth limited. (b) Maximizing memory bandwidth is valued on this machine for arithmetic intensities less than 16. (c) Improving arithmetic intensity by minimizing conflict and capacity misses can significantly improve performance when bandwidth limited.

## 6. Performance Results and Analysis

Figure 7(a) shows SpMV(suite median), stencil, and LBMHD performance before and after auto-tuning. We've also explicitly shown the benefits of exploiting architecture or instruction set architecture (ISA) specific optimizations including SSE for x86 or heterogenity (SPEs) on the Cell blade. On Cell, only the orange, architecture specific, bar represents code running on the SPEs. All other results were obtained on the PPEs. Clearly all architectures benefit from auto-tuning on all kernels, however Barcelona gets much of its performance on stencils and LBMHD from x86 optimized code — perhaps acceptable given the ubiquitous x86 ISA. However, Cell gets almost entirely all of its performance from SIMDized and local store implementations running on the SPEs. Neither of these would be considered productive. In Figure 8, we overlay kernel performance onto a roofline model constructed for each architecture. This allows us to visually interpret the fundamental architectural limitations and their impact on kernel performance. For simplicity, we only show SpMV for the dense matrix stored in sparse format. Red diamonds denote untuned performance, where green dots denote auto-tuned performance. The purple regions highlight the expected range of compulsory arithmetic intensities. Clearly, on most architectures, performance is limited by the sustainable memory bandwidth (diagonal lines). We should reiterate that both stencils and LBMHD have disproportionally few multiply instructions. As such, the inherent imbalance between multiplies and adds will limit performance to 50% of peak on all but the SPARC architecture.

A cursory examination of the green bars (out-of-the box pthreads) in Figure 7(a) shows that Clovertown, Barcelona, and Victoria Falls all deliver comparable un-tuned performance for each kernel. The Cell PPE un-tuned performance is so poor, we can barely see it at this scale. We also see that the benefits of auto-tuning (yellow bars) can significantly improve all machines. When using explicitly SIMDized and auto-tuned code on the cache-based machines, and fully exploiting the heterogeneity of the Cell blade (orange bars), we see that for most kernels, the Cell SPEs deliver considerably better performance than even Barcelona. However, for the most computationally intensive kernel, LBMHD, Cell's weak double precision implementation is now the bottleneck. Thus, Barcelona has nearly reached parity with Cell.

Simply put, the out-of-the box pthreads code (green bars) represent maximum productivity, while auto-tuned pthreads (yellow bars) makes a one time sacrifice in productivity for performance portability across most architectures. The orange bars represent an almost complete loss of productivity in favor of great ISA-specific




**(a)** Performance **(b)** Power Efficiency

**Figure 7.** Comparison of performance (left) and power efficiency (right) across architectures and kernels with increasing optimization. Sustained power was measured under full load with a digital power meter.

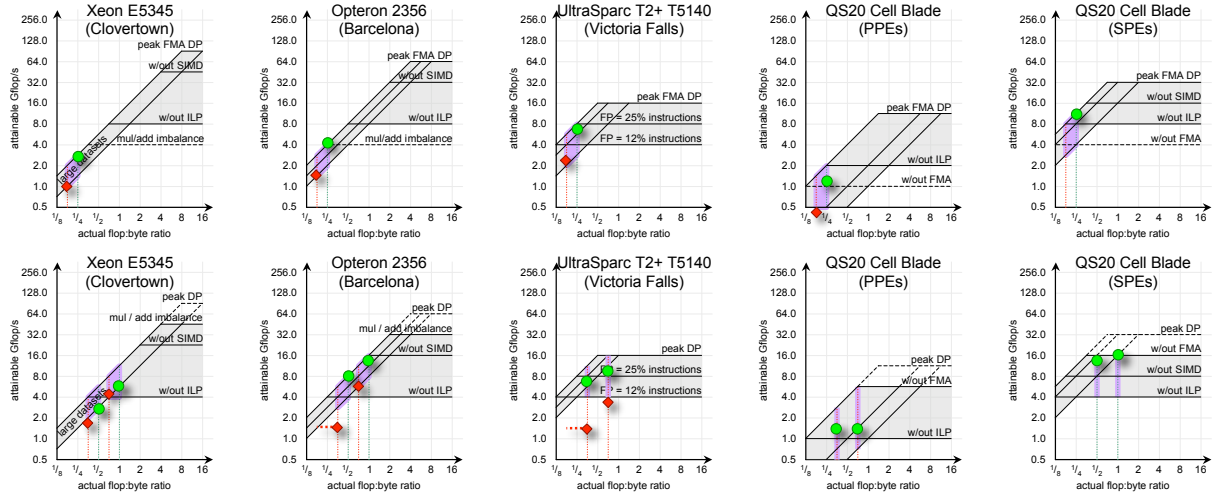



**Figure 8.** Roofline models for SpMV (top) as well as a combined model (bottom) for both the heat equation ($ArithmeticIntensity \leq 0.5$) and LBMHD ($ArithmeticIntensity \leq 1.0$).

auto-tuned performance.

Figure 7(b) shows power efficiency after full optimization. We used a digital power meter to measure sustained system power and quote it in Table 1. Clearly, Victoria Falls' high sustained power — 610W — significantly reduces its power efficiency, while Cell's relatively low 285W improved its relative power efficiency.

### *6.1. Clovertown*

Although auto-tuning provided nearly a 3× increase in median performance for SpMV, we see only about a 50% increase in performance for Stencil and LBMHD. Clovertown has no NUMA issues, has numerous hardware prefetchers but relatively poor memory bandwidth, and has giant caches with ample associativity and capacity per thread. As a result, Clovertown started out nearly memory bound, and suffered from few conflict or capacity misses. Thus, auto-tuning provides relatively little benefit on this architecture for these kernels. When referring to the roofline models for the Clovertown in Figure 8, we see that Stencil and LBMHD performance seemed to be pegged to the lower bandwidth diagonal associated with large data sets, while the smaller SpMV problems reach the higher optimized bandwidth diagonal. The relatively small snoop filter in the memory controller hub

can eliminate the front side bus (FSB) snoop traffic that saps useful bandwidth. Thus, smaller problems will likely see better bandwidth. In either case, it is clear that Clovertown is heavily memory bandwidth starved. In fact scalability studies have shown that quad-core is of little value on SpMV, and surprisingly, dual core is of little value on Stencil and LBMHD. On both Stencil and LBMHD, Clovertown achieves relatively little speedup from explicit SIMDization.

*6.2. Barcelona*

Although Barcelona has the same number of cores, sockets and DRAM frequency as Clovertown, we see considerably better performance after auto-tuning. The Opteron is a dual socket NUMA machine. When properly managed, this provides significantly more memory bandwidth per core than the FSB used on the Clovertown. The caches are much smaller making cache blocking a necessity. We also observed that software prefetching could significantly improve memory bandwidth. When bandwidth is maximized and traffic minimized, we must also ensure that in-core performance is not the bottleneck. Explicit SIMDization via SSE intrinsics nearly doubled both LBMHD and Stencil performance. Part of this speedup is from the use of an x86 only cache-bypass instruction, but the other part is from explicitly exploiting the 128b floating point units. As noted, the order of optimizations in the roofline is based on which is easiest for a compiler to generate. After explicitly SIMDizing the code, perhaps we should have redrawn the roofline explicitly interchanging SIMD and ILP. One should be mindful of this optimization as it is difficult for most programmers to SIMDize even small kernels. As the roofline figure suggests, after auto-tuning, Barcelona is totally memory bound. Auto-tuning provided between a 2.6× and 5.4× speedup.

*6.3. Victoria Falls*

Victoria Falls is a heavily multithreaded architecture with relatively small shared caches, and even smaller working sets per thread — 1KB L1 per thread, 64KB L2 per thread. The more challenging issue is the low associativity of the caches — 16 way L2 for 64 threads. As a result most of the optimizations that showed benefit were ones that addressed the 3C's of caches. Nevertheless, the portable auto-tuned C code for all three kernels runs better on Victoria Falls than any other architecture despite having the lowest peak floating point capability. When examining the roofline model, we conclude that Victoria Falls is bandwidth bound for SpMV, and possibly stencil, but has transitioned to being processor bound for LBMHD. Without accurate performance counter data we cannot accurately determine the x-coordinates of the dots and diamonds in the roofline figure. Thus, we cannot conclude with certainty when small changes in arithmetic intensity allow one conclusion or the other. Auto-tuning provided between a 1.9× and 5.2× speedup.

*6.4. Cell Blade (PPEs)*

Out-of-the box Cell PPE performance is abysmal on all three kernels. Auto-tuning provided significant speedups for Stencil and LBMHD — 3.3× and 10× respectively. Nevertheless, using both threads on both cores delivers far less performance than any other machine. This has serious ramifications as a heterogeneity version of Amdahl's law suggests that the vast majority of the code must be amenable to running on the SPEs to see significant performance speedups. When coupled when the immaturity of the Cell SDK, productivity will suffer greatly.

*6.5. Cell Blade (SPEs)*

The 3C's model is loosely applicable to local store architectures. First, compulsory misses are universal for all architectures. Second, there are no conflict misses. However, some padding is still required for efficient direct memory access (DMA) operation in much the same way padding is required for SIMDization. Finally, all capacity misses must be handled in software. Programs whose working set size exceeds the local store capacity must be blocked in software. Thus, the program, rather than the hardware, must find all spatial and temporal locality either when the problem is specified or during execution. The advantage of this approach is that the arithmetic intensity can be calculated easily and exactly. There is no need for performance counters to cope

with the uncertainty in the number of conflict or capacity misses. Furthermore, the decoupling of DMA from execution in conjunction with double buffering is a perfect match for the overlapping of communication and computation found in the Roofline model.

Despite the SPE's weak double precision implementation, we see that their use of DMA ensures good memory bandwidth performance coupled with a minimization of memory traffic. As a result, we see more than a $10\times$ improvement over auto-tuned C code on the PPE. When examining the roofline, we see that SpMV is totally memory bandwidth bound, although the choice of a minimum register blocking of $2\times1$ means the implementation compulsory misses are higher than the algorithmic compulsory misses. Stencil is likely barely compute bound (remember FMA's cannot be exploited), while LBMHD is strongly compute bound. Thus one expects the forthcoming QS22 blades to significantly improve LBMHD performance, slightly improve Stencil performance, but do nothing for SpMV.

## 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 discussed a set of multicore optimizations for Sparse Matrix Vector Multiplication (SpMV), the explicit heat equation PDE on a regular grid (Stencil), and a lattice Boltzmann application (LBMHD). We detailed and presented the results of 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: TLB and cache blocking, loop unrolling, code reordering, software prefetching, streaming stores, and use of SIMD instructions. The impact of each optimization varies significantly across architecture and kernel, necessitating a machine-dependent approach to automatic tuning. In addition, our detailed analysis reveals the performance bottlenecks for each computation on our evaluated system.

Results show that the Cell processor still offers the highest raw performance and power efficiency for these computations, despite having peak double-precision performance and memory bandwidth that is lower than many of the 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. However, much of this work must be duplicated on cache-based architectures to achieve good performance. Nevertheless, these performance disparities point to the deficiencies of existing automatically-managed cache hierarchies, even for architectures with sophisticated hardware and software prefetch capabilities. Thus there is considerable room for improvements in the latency tolerance techniques of microprocessor core designs.

Our study has demonstrated that — for the evaluated class of algorithms — auto-tuning is essential in achieving good performance. Moreover, in the highly multithreaded/multicore world, DRAM bandwidth is the preeminent factor in sustained performance. In this world, there is little need for processor designs that emphasize high single thread throughput. Rather, high throughput via large numbers of simpler cores is likely to deliver higher efficiency and vastly simpler implementation. While prior reseach has shown that these design philosophies offer substantial benefits for peak computational rates [10], our work quantifies that this approach can offer significant performance benefits on real scientific applications.

Overall the auto-tuned codes achieved sustained superscalar performance that is substantially higher than any published results to date, with speedups often in excess of $5\times$ 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.

Finally, we analyzed the performance using the roofline model. The roofline makes it clear which architectural paradigms must be exploited to improve performance. Fraction of peak raw bandwidth or flops is an inappropriate metric of performance. The Roofline model presents realistic ceilings to performance. After auto-tuning is applied to the kernels we see that performance is very near the roofline ceilings of each architecture despite potentially low percent of peak raw flop or bandwidth numbers.

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. We hope the roofline model will provide an architecture specific optimization search path that will significantly reduce the time required for auto-tuning.

## 8. Acknowledgments

We would like to express our gratitude to Forschungszentrum Jülich for access to their Cell blades, as well Brian Waldecker of AMD for Barcelona access. This work was supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231, by NSF contract CNS-0325873, and by Microsoft and Intel Funding under award #20080469.

## 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] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC.

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

[4] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *(submitted to) Proc. SC2008: High performance computing, networking, and storage conference*, 2008.

[5] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

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

[7] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[8] 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.

[9] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. K. Prasanna, M. Püschel, and M. Veloso. SPIRAL: Automatic implementation of signal processing algorithms. In *High Performance Embedded Computing (HPEC)*, 2000.

[10] 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.

[11] 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.

[12] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.

[13] S. Williams. *Autotuning Performance on Multicore Computers*. PhD thesis, University of California, Berkeley, 2008.

[14] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*, Miami, Florida, 2008.

[15] 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.

[16] S. Williams and D. Patterson. The roofline model: An insightful multicore performance model. *(submitted to) Communications of the ACM*, 2008.

[17] 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 *Hot Chips 20: Stanford University, Stanford, California, August 24–26, 2008 (to appear)*, 2008.