

UC Merced

UC Merced Previously Published Works

Title

Large-scale numerical simulations on high-end computational platforms

Permalink

<https://escholarship.org/uc/item/8gq69106>

ISBN

9781439815694

Authors

Oliker, L

Carter, J

Beckner, V

et al.

Publication Date

2010

DOI

10.1201/b10509

Peer reviewed

Chapter 13

Auto-Tuning Memory-Intensive Kernels for Multicore

Samuel W. Williams

Lawrence Berkeley National Laboratory

Kaushik Datta

University of California at Berkeley

Leonid Oliker

Lawrence Berkeley National Laboratory

Jonathan Carter

Lawrence Berkeley National Laboratory

John Shalf

Lawrence Berkeley National Laboratory

Katherine Yelick

Lawrence Berkeley National Laboratory

13.1	Introduction	274
13.2	Experimental Setup	275
13.2.1	AMD Opteron 2356 (Barcelona)	275
13.2.2	Xeon E5345 (Clovertown)	276
13.2.3	IBM Blue Gene/P (Compute Node)	276
13.3	Computational Kernels	276
13.3.1	Laplacian Differential Operator (Stencil)	277
13.3.2	Lattice Boltzmann Magnetohydrodynamics (LBMHD) .	278
13.3.3	Sparse Matrix-Vector Multiplication (SpMV)	280
13.4	Optimizing Performance	282
13.4.1	Parallelism	282
13.4.2	Minimizing Memory Traffic	283
13.4.3	Maximizing Memory Bandwidth	286
13.4.4	Maximizing In-Core Performance	287
13.4.5	Interplay between Benefit and Implementation	287
13.5	Automatic Performance Tuning	287
13.5.1	Code Generation	289

13.5.2	Auto-Tuning Benchmark	290
13.5.3	Search Strategies	291
13.6	Results	291
13.6.1	Laplacian Stencil	291
13.6.2	Lattice Boltzmann Magnetohydrodynamics (LBMHD) .	292
13.6.3	Sparse Matrix-Vector Multiplication (SpMV)	294
13.7	Summary	295
13.8	Acknowledgments	296

In this, chapter, we discuss the optimization of three memory-intensive computational kernels — sparse matrix-vector multiplication, the Laplacian differential operator applied to structured grids, and the `collision()` operator with the lattice Boltzmann magnetohydrodynamics (LBMHD) application. They are all implemented using a single-process, (POSIX) threaded, SPMD model. Unlike their computationally-intense dense linear algebra cousins, performance is ultimately limited by DRAM bandwidth and the volume of data that must be transferred. To provide performance portability across current and future multicore architectures, we utilize automatic performance tuning, or auto-tuning.

The chapter is organized as follows. First, we define the memory-intensive regime and detail the machines used throughout this chapter. Next, we discuss the three memory-intensive kernels that we auto-tuned. We then proceed with a discussion of performance optimization and automatic performance tuning. Finally, we show and discuss the benefits of applying the auto-tuning technique to three memory-intensive kernels.

13.1 Introduction

Arithmetic Intensity is a particularly valuable metric in predicting the performance of many single program multiple data (SPMD) kernels. It is defined as the ratio of requisite floating-point operations to total DRAM memory traffic. Often, on cache-based architectures, one simplifies total DRAM memory traffic to include only compulsory reads, write allocations, and compulsory writes.

Memory-intensive computational kernels are characterized by those kernels with arithmetic intensities that are constant or scale slowly with data size. For example, BLAS-1 operations like the dot product of two N -element vectors perform $2 \cdot N$ floating-point operations, but must transfer $2 \cdot 8N$ bytes. This results in an arithmetic intensity (1/8) that does not depend on the size of the vectors. As this arithmetic intensity is substantially lower than most machines' Flop:Byte ratio, one generally expects such kernels to be memory-bound for

any moderately-large vector size. Performance, measured in floating-point operations per second (GFlop/s), is bound by the product of memory bandwidth and arithmetic intensity. Even computational kernels whose arithmetic intensity scales slowly with problem size, such as out-of-place complex-complex FFT's, roughly $0.16 \log(n)$, may be memory-bound for any practical size of n for which capacity misses are not an issue.

Unfortunately, arithmetic intensity (and thus performance) can be degraded if superfluous memory traffic exists (e.g., conflict misses, capacity misses, speculative traffic, or write allocations). The foremost goal in optimizing memory-intensive kernels is to eliminate as much of this superfluous memory traffic as possible. To that end, we may exploit a number of strategies that either passively or actively elicit better memory subsystem performance. Ultimately, when performance is limited by compulsory memory traffic, reorganization of data structures or algorithms is necessary to achieve superior performance.

13.2 Experimental Setup

In this section, we discuss the three multicore SMP computers used in this chapter — AMD's Opteron 2356 (Barcelona), Intel's Xeon E5345 Clovertown, and IBM's Blue Gene/P (used exclusively in SMP mode). As of 2009, these architectures dominate the top500 list of supercomputers [13]. The key features of these computers are shown in Table 13.1 and detailed in the following subsections.

13.2.1 AMD Opteron 2356 (Barcelona)

Although superseded by the more recent Shanghai and Istanbul incarnations, the Opteron 2356 (Barcelona) effectively represents the future x86 core and system architecture. The machine used in this work is a 2.3 GHz dual-socket \times quad-core SMP. As each superscalar out-of-order core may complete both a single instruction-multiple data (SIMD) floating-point add and a SIMD floating-point multiply per cycle, the peak double-precision floating-point performance (assuming balance between adds and multiplies) is 73.6 GFlop/s. Each core has both a private 64 Kbyte L1 data cache and a 512 Kbyte L2 victim cache. The four cores on a socket share a 2 Mbyte L3 cache.

Unlike Intel's older Xeon, the Opteron integrates the memory controllers on chip and provides an inter-socket network (via HyperTransport) to provide cache coherency as well as direct access to remote memory. This machine uses DDR2-667 DIMMs providing a DRAM pin bandwidth of 10.66 Gbyte/s per socket.

13.2.2 Xeon E5345 (Clovertown)

Providing an interesting comparison to Barcelona, the Xeon E5345 (Clovertown) uses a modern superscalar out-of-order core architecture coupled with an older frontside bus (FSB) architecture in which two multichip modules (MCM) are connected with an external memory controller hub (MCH) via two frontside buses. Although two FSBs allows a higher bus frequency, as these chips are regimented into a cache coherent SMP, each memory transaction on one bus requires the MCH to produce a coherency transaction on the other. In effect this eliminates the parallelism advantage in having two FSBs. To rectify this, a snoop filter was instantiated within the MCH to safely eliminate as much coherency traffic as possible. Nevertheless, the limited FSB bandwidth (10.66 Gbyte/s) bottlenecks the substantial DRAM read bandwidth of 21.33 Gbyte/s.

Each core runs at 2.4 GHz, has a private 32 KB L1 data cache, and, like the Opteron, may complete one SIMD floating-point add and one SIMD floating-point multiply per cycle. Unlike the Opteron, the two cores on a chip share a 4 Mbyte L2 and may only communicate with the other two cores of this nominal quad-core MCM via the shared frontside bus.

13.2.3 IBM Blue Gene/P (Compute Node)

IBM's Blue Gene/P (BGP) takes a radically different approach to ultra-scale system performance compared to traditional superscalar processors, as it relies more heavily on power efficiency to deliver strength in numbers instead of maximizing performance per node. To that end, the compute node instantiates four PowerPC 450 embedded cores in its one chip. These cores are dual-issue, in-order, SIMD enabled cores that run at a mere 850 MHz. As such, each node's peak performance is only 13.6 GFlop/s — a far cry from the x86 superscalar performance. However, the order of magnitude reduction in node power results in superior power efficiency.

Each of the four cores on a BGP compute chip has a highly associative 32 Kbyte L1 data cache, and they collectively share an 8 Mbyte L3. As it is a single chip solution, cache-coherency is substantially simpler, as all snoops and probes are on chip. The chip has two 128-bit DDR2-425 DRAM channels providing 13.6 Gbyte/s of bandwidth to 4 Gbyte of DRAM capacity. Like Opterons and Xeons, Blue Gene/P has hardware prefetch capabilities.

13.3 Computational Kernels

In this section, we introduce the three memory-intensive kernels used as exemplars throughout the rest of the chapter: the Laplacian stencil,

TABLE 13.1: Architectural summary of evaluated platforms.

Core Architecture	AMD Barcelona	Intel Core2	IBM PowerPC 450
Type	superscalar out-of-order	superscalar out-of-order	dual issue in-order
Clock (GHz)	2.3	2.40	0.85
DP Peak (GFlop/s)	9.2	9.60	3.4
Private L1 Data Cache	64 Kbyte	32 Kbyte	32 Kbyte
Private L2 Data Cache	512 Kbyte	—	—

Socket Architecture	Opteron 2356 Barcelona	Xeon E5345 Clovertown	Blue Gene/P Chip
Cores per Socket	4	4 (MCM)	4
Shared Cache	2 Mbyte L3	2×4 Mbyte L2	8 Mbyte L2
Memory Parallelism Paradigm	HW prefetch	HW prefetch	HW prefetch

Node Architecture	Opteron 2356 Barcelona	Xeon E5345 Clovertown	Blue Gene/P Node
Sockets per SMP	2	2	1
DP Peak (GFlop/s)	73.69	76.80	13.60
DRAM Bandwidth (GB/s)	21.33	21.33 (read) 10.66 (write)	13.60
DP Flop:Byte Ratio	3.45	2.40	1.00

the `collision()-stream()` operators extracted from Lattice Boltzmann Magnetohydrodynamics (LBMHD), and sparse matrix-vector multiplication (SpMV).

13.3.1 Laplacian Differential Operator (Stencil)

Partial differential equation (PDE) solvers constitute a large fraction of scientific applications in such diverse areas as heat diffusion, electromagnetics, and fluid dynamics. Solutions to these problems are often implemented using an explicit method via iterative finite-difference techniques. Computationally, these approaches sweep over a spatial grid performing *stencils* — a linear combinations of each a point’s nearest neighbor. Our first kernel is the quintessential finite difference operator found in many partial difference equations — the 7-point Laplacian stencil.

This kernel is implemented as an out-of-place Laplacian stencil and is visualized in Figure 13.1. Since it uses Jacobi’s method, we maintain a copy of the grid for both the current and next time steps and thereby avoid any data hazards. Conceptually, the stencil operator in Figure 13.1(b) is simultaneously

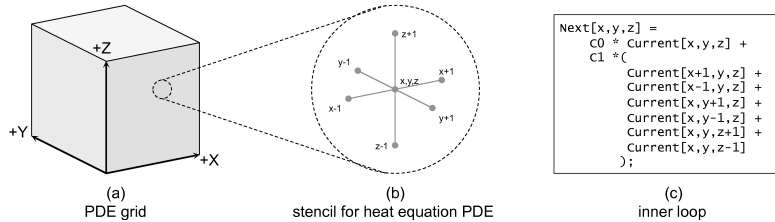


FIGURE 13.1: Visualization of the data structures associated with the heat equation stencil. (a) The 3D temperature grid. (b) The stencil operator performed at each point in the grid. (c) Pseudocode for stencil operator.

applied to every point in the 256^3 scalar grid shown in Figure 13.1(a). This allows an implementation to select any traversal of the points.

This kernel is exemplified by an interesting memory access pattern with seven reads and one write presented to the cache hierarchy. However, there is possibility of 6-fold reuse of the read data. Unfortunately this requires substantial per-thread cache capacity. Much of the auto-tuning effort for this kernel is aimed at eliciting this ideal cache utilization through the elimination of cache capacity misses. Secondary efforts are geared toward the elimination of conflict misses and write allocation traffic. Thus, with appropriate optimization, memory bandwidth and compulsory memory traffic provide the ultimate performance impediment. To that end, in-core performance must be improved through various techniques only to the point where it is not the bottleneck. For further details on the heat equation and auto-tuning approaches, we direct the reader to [108].

13.3.2 Lattice Boltzmann Magnetohydrodynamics (LBMHD)

The second kernel examined in this chapter is the inner loop from the Lattice Boltzmann Magnetohydrodynamics (LBMHD) application [223]. LBMHD was developed to study homogeneous isotropic turbulence in dissipative magnetohydrodynamics (MHD) — the theory pertaining to the macroscopic behavior of electrically conducting fluids interacting with a magnetic field. The study of MHD turbulence is important in the physics of stellar phenomena, accretion discs, interstellar and intergalactic media, and plasma instabilities in magnetic fusion devices [56].

In Lattice methods, the macroscopic quantities (like density or momentum) at each point in space are reconstructed through operations on a momentum lattice — a discretization of momentum along 27 vectors. As LBMHD couples computational fluid dynamics with Maxwell’s equations, the momentum lattice is augmented with a 15-velocity (cartesian vectors) magnetic lattice as shown in Figure 13.2. Clearly, this creates very high memory capacity requirements — over 1 Kbyte per point in space.

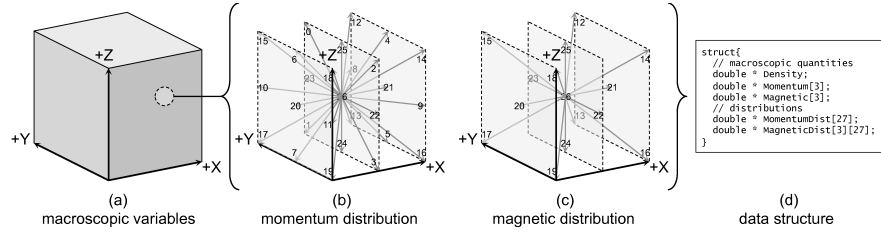


FIGURE 13.2: Visualization of the data structures associated with LBMHD. (a) The 3D macroscopic grid. (b) The D3Q27 momentum scalar velocities. (c) D3Q15 magnetic vector velocities. (d) C structure of arrays datastructure. Note: each pointer refers to a N^3 grid, and X is the unit stride dimension.

Lattice Boltzmann Methods (LBM) iterate through time calling two functions per time step: a `collision()` operator, where the grid is evolved one timestep, and a `stream()` operator that exchanges data with neighboring processors. In a shared memory, threaded implementation, `stream()` degenerates into a function designed to maintain periodic boundary conditions.

In serial implementations, `collision()` typically dominates the run time. To ensure that an auto-tuned `collision()` continues to dominate runtime in a threaded environment, we also thread-parallelize `stream()`. We restrict our exploration to a 128^3 problem on the x86 architectures, but only 64^3 on Blue Gene as it lacks sufficient main memory. For further details on LBMHD and previous auto-tuning approaches, we direct the reader to the following papers [363, 364].

The `collision()` code is far too large to duplicate here. Superficially, the `collision()` operator must read the lattice velocities from the current time step, reconstruct the macroscopic quantities of momentum, magnetic field, and density, and create the lattice velocities for the next time step. When distilled, this involves reading 73 doubles, performing 1300 floating point operations, and writing 79 doubles per lattice update. This results in a compulsory-limited arithmetic intensity of about 0.7 Flops per byte on write-allocate architectures, but may be improved to about 1.07 through the use of cache bypass instructions.

Conceptually, the `collision()` operator within LBMHD comprises both a 15 and a 27 point stencil similar to the previously discussed Laplacian Stencil. However, as lattice methods utilize an auxiliary grid that stores a distribution of velocities at each point, these stencil operators are different in that they reference a different velocity from each neighbor. As such, there is no inter-lattice update reuse. Proper selection of data layout (structure-of-arrays) transforms the principal optimization challenge from cache blocking to translation lookaside buffer (TLB) blocking. When coupled with a code transformation, one may reap the benefits of good cache and TLB locality simultaneously with effective SIMDization.

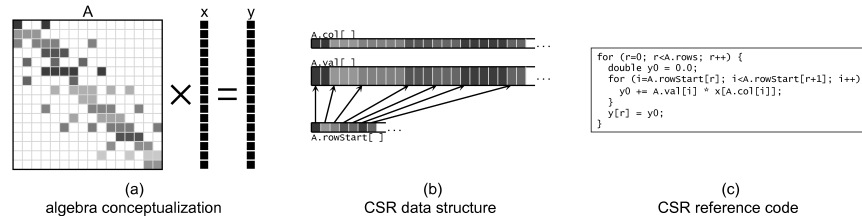









FIGURE 13.3: Sparse matrix-vector multiplication (SpMV). (a) Visualization of the algebra: $y \leftarrow Ax$, where A is a sparse matrix. (b) Standard compressed sparse row (CSR) representation of the matrix. This structure of arrays implementation is favored on most architectures. (c) The standard implementation of SpMV for a matrix stored in CSR. The outer loop is trivially parallelized without any data dependencies.

As this application was designed for a weak-scaled message passing (MPI), single program, multiple data (SPMD) environment, we may simply tune to optimize single node performance, then integrate the resultant optimized implementation back into the MPI version.

13.3.3 Sparse Matrix-Vector Multiplication (SpMV)

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 performed poorly on single-core cache-based microprocessor systems [346]. Compared to dense linear algebra kernels, sparse kernels like SpMV suffer from high instruction and storage overhead per floating-point operations, and a lack of instruction- and data-level parallelism in the reference implementations. Even worse, unlike the implicit (arithmetic) addressing possible in dense linear algebra and structured grid calculations (stencils and lattice methods), indexing neighboring points in a sparse matrix requires an indirect access. This can result in potentially irregular memory access patterns (jumps and discontinuities). As such, achieving good performance on these kernels often requires selection of a compact data structure, reordering of the computations to favor regular memory access patterns, and code transformations based on runtime knowledge of the sparse matrix. This need for run-time optimization and tuning is a major distinction from most other computational methods.

In this chapter, we consider the SpMV operation $y \leftarrow Ax$, where A is a sparse matrix, and x, y are dense vectors. A sparse matrix is a special case of the matrices found in linear algebra in which most of the matrix entries are zero. In a matrix-vector multiplication, computation on zeros does not change the result. As such, they may be eliminated from both the representation

	Dense	Protein	Spheres	Cantilever	Wind Tunnel	Harbor	QCD
Spyplot (Sparsity)							
Rows	2K	36K	83K	62K	218K	47K	49K
Columns	2K	36K	83K	62K	218K	47K	49K
Nonzeros (NNZ)	4.0M	4.3M	6.0M	4.0M	11.6M	2.4M	1.9M








	Ship	Economics	Epidemiology	Accelerator	Circuit	Webbase	LP
Spyplot (Sparsity)							
Rows	141K	207K	526K	121K	171K	1M	4K
Columns	141K	207K	526K	121K	171K	1M	1M
Nonzeros (NNZ)	4.0M	1.3M	2.1M	2.6M	0.9M	3.1M	11.3M

FIGURE 13.4: Benchmark matrices used as inputs to our auto-tuned SpMV library framework.

and the computation, leaving only the *nonzeros*. Although the most common data structure used to store a sparse matrix for SpMV-heavy computations is compressed sparse row (CSR) format, illustrated with the corresponding kernel in Figure 13.3, we will explore alternate representations of the compute kernel. CSR requires a minimum overhead of 4 bytes (column index) per 8 byte nonzero. As microprocessors only have sufficient cache capacity to cache the vectors in their entirety, we may define the compulsory memory traffic as 12 bytes per nonzero. SpMV will perform 2 Flops per nonzero. As such, the ideal CSR arithmetic is only 0.166 Flops per byte; making SpMV heavily memory-bound. Capacity misses and sub-optimal bandwidth will substantially impair performance.

Unlike most of dense linear algebra, stencils on structured grids, and Fourier transforms, matrices used in sparse linear algebra are not only characterized by their dimensions but also by their *sparsity* pattern — a scatter plot of nonzeros. Figure 13.4 presents the spyplot and the key characteristics associated with each matrix used in this chapter. Observe that for the most part, the vectors are small, but the matrices (in terms of nonzeros) are large. Remember, 12 bytes are required per nonzero. As such, a matrix with four million nonzeros requires at least 48 Mbyte of storage — far larger than most caches. We selected a set of matrices that would exhibit several classes of sparsity: dense, low bandwidth (principally finite element method), unstructured, and extreme aspect ratio. Such matrices will see differing cache capacity issues on multicore SMPs. In addition, we ensured the matrices would run the gambit of nonzeros per row — a key component in CSR performance. Finally, although

some matrices are symmetric, we convert all of them to non-symmetric format and do not exploit this characteristic.

For further details on the sparse matrix-vector multiplication and previous auto-tuning efforts, we direct the reader to [362, 365, 366].

13.4 Optimizing Performance

Broadly speaking, we may either classify optimizations by their impact on implementation and usage, or by the performance bottleneck they attempt to eliminate. For example, an implementation-based categorization may delineate optimizations into four groups based on what changes are required: only code structure, data structures, the style of parallelism, or algorithms. On the other hand, if we categorize optimizations by bottleneck, we may create groups that more efficiently exploit parallelism, minimize memory traffic, maximize memory bandwidth, or maximize in-core performance. That being said, in this section, we describe the optimizations employed by our three auto-tuners grouped using the bottleneck-oriented taxonomy. Moreover, as we're focused on memory-intensive kernels, we will prioritize the optimizations accordingly.

13.4.1 Parallelism

Broadly speaking parallelism encompasses approaches to synchronization, communication, use of threads or processes, and problem decomposition.

Synchronization and Communication: Although a number of alternate strategies are possible (including DAG-based schedulers [278]), we adopted a POSIX thread-based, SPMD, bulk-synchronous approach to exploit multicore parallelism. Unlike process-based, shared memory-optimized message passing approaches, we exploit the ever-present cache coherency mechanisms for both efficient communication as well as to eliminate system calls. We enforce bulk synchronous semantics via a shared memory spin barrier.

Problem Decomposition: We utilize two different approaches to problem decomposition. First, the structured grid codes spatially decompose the stencil sweep into subdomains by partitioning the problem in two dimensions (not the unit stride). We ensure there are at least as many subdomains as there are threads. Subdomains may then be assigned to threads in chunks in a round-robin ordering. For LBMHD, the subdomains are not perfect rectangular volumes. Rather, within each plane the boundaries are aligned to cache lines. In effect this performs only loop parallelization through blocking. No part of the data structure is changed.

Conversely, we apply a very different technique when parallelizing sparse matrix-vector multiplication. To ensure there are no data dependencies, we only parallelize by rows, creating a number of submatrices that contain roughly

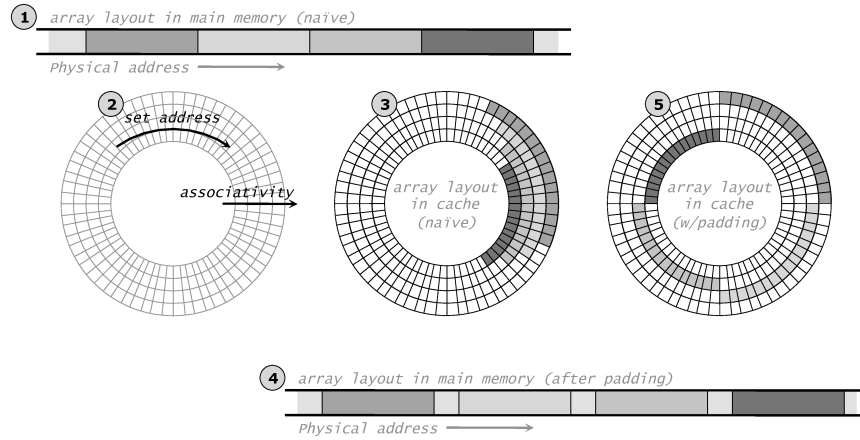


FIGURE 13.5: Conceptualization of array padding in both the physical (linear) and cache (periodic) address spaces. (See color insert.)

the same number of nonzeros, but may span wildly different numbers of rows. Each of these submatrices is as if it were its own matrix (hierarchical storage). We may now optimize each submatrix with a unique register blocking (see below). That is, some submatrices may be encoded using 1×1 COO while others are encoded with 8×4 CSR. Clearly, SpMV optimization went well beyond simple loop parallelization and subsumes data structure transformations as well.

13.4.2 Minimizing Memory Traffic

When a kernel is memory-bound, there are two principal optimizations: reduce the volume of memory traffic or increase the attained memory bandwidth. For simple memory access patterns, modern superscalar processors often achieve a high fraction of memory bandwidth. As such, our primary focus should be on techniques that minimize the volume of memory traffic. Broadly speaking, we may classify memory traffic using the Three C cache model's compulsory, conflict, and capacity misses [165] augmented with speculative (prefetch) and write-allocate traffic. We implemented a set of optimizations that attempt to minimize each class of traffic. Not all optimizations are applicable to all kernels.

Array Padding: Caches have limited associativity. When too many memory references map to the same set in the cache, a conflict miss will occur and useful data will be evicted. These conflicts may arise from intra-thread conflicts or, when shared caches are in play, from inter-thread conflicts.

An example of this complex behavior may be illustrative. In physical memory, subsets of an array assigned to different threads (red, orange, green, blue)

are disjoint: Figure 13.5(1). Unfortunately, a given set in a cache may map many different addresses in physical memory (there is a 1 to many relationship). As such, in order to properly visualize caches, we must abandon the linear memory concept in favor of the periodic coordinate system shown in Figure 13.5(2). In this case, every 64^{th} physical address maps to the same cache set. Figure 13.5(3) shows that when those disjoint physical addresses are mapped to the cache set address space, the segments overlap. This can put substantial pressure on cache associativity (number of elements mapped to the same set address). Execution on any cache less than 4-way associative will generate conflict misses and limit the working set size to 2 elements. Array padding is the simplest and most common remediation strategy. Simply put, dummy elements (placeholders) are inserted between each thread's subset of the array (4). As a result, when mapped to the cache coordinate system, Figure 13.5(5), we see there is no longer any overlap of data at a given set address, and the demands on cache associativity have been reduced from 4-way to 1-way (direct mapped).

Due to the limited number of memory streams and reuse, inter-thread conflict misses predominate on SpMV and the Laplacian stencil. On LBMHD, where the `collision()` operator attempts to keep elements from 150 different arrays in the cache, eliminating intra-thread conflict misses is key. As such, we implemented two different array padding strategies to mitigate cache conflict misses. For SpMV and stencils, we pad each array so that the address of the first element maps to a unique set in the cache. Moreover, the padding is selected to ensure that the set addresses of the threads' first elements are equally spaced (by set address) in the last level cache. The ideal padding may be either calculated arithmetically or obtained experimentally. For SpMV, we simply `malloc` each thread block independently with enough space to pad by the cache size. We then align to a 4 MB boundary and pad by the thread's fraction of the cache. Array padding for LBMHD is somewhat more complex. We pad each velocity's array so that when referenced with the corresponding stencil offset (and corresponding address offset) the resultant physical address maps to a unique, equally-spaced cache set. Although this sounds complicated in practice, it's relatively easy to implement. For further details on how these kernels exploit array padding, we direct the reader to [108, 362–364].

Cache Blocking: The reference implementations of many kernels demand substantial cache working sets. In practice, as processor architects cannot implement caches that are large enough to avoid capacity misses. The volume of memory traffic is increased. LBMHD does not exhibit any inter-stencil reuse. That is, there is no two stencils reuse the same values. As such, cache capacity misses are nonexistent. However, the Laplacian stencil shows substantial reuse. Like dense matrix-vector multiplication, SpMV will also show reuse on vector accesses. In either case, we must restructure code, and possibly data, to eliminate capacity miss traffic. Like cache blocking in dense linear algebra, we may apply a simple loop blocking technique to the Laplacian stencil to ensure an implementation generates relatively few capacity misses. In practice, this is

implemented the same as problem decomposition for parallelization. However, defining dense blocks (of source vectors) for SpMV often yields a dramatically suboptimal solution. As such, we employ a novel sparse blocking technique that ensures that each cache block touches the same number of cache lines regardless of how many rows or columns the block spans. In practice, for a given thread's block of the matrix, we cache block it by simply adding columns of the sparse matrix until the number of unique cache lines touched reaches a preset number. Clearly, this requires substantial data structure changes.

Cache Bypass: Based on consumer applications, most cache architectures implement a *write-allocate* protocol. That is, if a store (write) misses in the cache, an existing line will be selected for eviction, the target line will be loaded into the cache, and the target word will be written to the line in the cache. Such an approach is based on the implicit assumption that if data is written, it will be promptly read and modified many times. Note, usage of such a policy is orthogonal to the *write-back* or *write-through* choice. Unfortunately, many computational kernels found in HPC read and write to separate arrays or data structures. As such, most writes allocate a line in the cache, completely obliterate its previous contents, and eventually write it back to DRAM. This makes write-allocate not only superfluous, but expensive as it will generate twice the memory traffic as a read — an obvious target for optimization when memory-bound.

Modern write-allocate cache architectures provide a means of eliminating this superfluous memory traffic via a special store instruction that bypasses the cache hierarchy. In the x86 ISA, this is implemented with the `movntpd` instruction. Unfortunately, most compilers cannot resolve the complex decision as to when to use this instruction; improper usage can reduce performance by an order of magnitude, where correct usage can improve performance by 50%. As such, in practice, we may only exploit this functionality through the use of SIMD *intrinsics* — a language construct with the interface of a function that the compiler will map directly to one instruction.

Often, the vectors used in SpMV are small enough to fit in cache. As such, the totality of DRAM memory traffic is reads and there is no need to use cache bypass. However, Jacobi stencils and lattice methods read and write to separate arrays. For other finite-difference operators like gradient or divergence, cache bypass may only improve performance by 75% or 25% respectively. Given this variability in benefit and the human effort required to implement this optimization, one should analyze the code before proceeding with this optimization. Nevertheless, usage of cache bypass on the Laplacian stencil or LBMHD can reduce the total memory traffic by 33% and improve performance by 50%.

Register blocking for SpMV: For most matrices, SpMV is dominated by compulsory misses. As such, neither cache blocking nor cache bypass will provide substantial benefits. That is not to say nothing can be done. Rather, a radical solution has emerged that eliminates compulsory miss traffic. Broadly speaking, sparse matrices require substantial meta data per nonzero — per-

haps a 50% overhead. However, we observe that many nonzeros are clustered in relatively small regions. As such, the optimization known as *register blocking* reorganizes the sparse matrix of nonzeros into a sparse matrix of small $R \times C$ dense matrices. Meta data is now needed only for each register block rather than each nonzero. If the zero fill required to make those $R \times C$ register blocks dense is less than the reduction in meta data, then the total memory traffic has been reduced — a clear win for a memory-bound kernel. Similarly, we may note that a range of column or row indices can be represented by a 16-bit integer instead of a 32-bit integer. This can save 2 bytes (or 17%) per nonzero. In this chapter we explore the product of 16 different register blockings, two matrix formats: compressed sparse row (CSR) and coordinate (COO), and the two index sizes. Register blocked CSR and COO are noted as BCSR and BCOO, respectively.

Please note, the term register blocking, when applied to sparse linear algebra refers to a hierarchical restructuring of the data, but when applied to dense linear algebra refers to a unroll and jam technique. In this chapter, our SpMV code heuristically explores these matrix compression techniques to find the combination that minimizes the matrix footprint.

13.4.3 Maximizing Memory Bandwidth

Now that we've discussed optimizations designed to minimize the volume of memory traffic, we may examine optimizations that maximize the rate at which said volume of data can be streamed into the processor. Basically, these optimizations aim to either avoiding memory latency or hide memory latency.

Blocking for the Translation Lookaside Buffer: All modern microprocessors use virtual memory. To translate the virtual address produced by the program's execution into the physical address required to access the cache or DRAM, the processor must inspect the page table to determine the mapping. As this is a slow process and page table entries rarely change, page table entries may be placed in a very fast, specialized (page) cache on chip — the *translation lookaside buffer* or TLB. Unfortunately, TLBs are small and thus may not be able to cache all the pages referenced by an application (regardless of page size). As such, it is possible to generate TLB capacity misses. These typically don't generate superfluous DRAM traffic like normal cache capacity misses because evicted page table entries may land in the L2 or L3 cache. The performance difference (resulting from an increase in average memory latency) between translations that hit in the TLB and those that hit in the L3 is substantial. We may recast the cache blocking technique (which eliminated cache capacity misses) to eliminate TLB capacity misses and avoid memory latency. In LBMHD, we used a loop interchange technique coupled with an auxiliary data structure. This allowed us to trade cache capacity for increased page locality (and reduced TLB capacity misses). This technique is detailed in [362–364].

Prefetching: Memory latency is high. To satisfy Little's Law [41]

and maximize memory bandwidth, the processor must express substantial memory-level parallelism. Unfortunately, superscalar execution may be insufficient. As such, hardware designers have incorporated both hardware and software prefetching techniques into their processors. The goal for either is to hide memory latency.

A software prefetch is an instruction like a load without a target address. As such, the processor will not stall waiting for it to complete. The user simply prefetches one element from each cache line to initiate the entire line's load. Unfortunately, such a practice requires the programmer to tune for the optimal "prefetch distance" — how far ahead prefetch addresses should be from load addresses. If he aims too low, latency will not be completely hidden. If he aims too high, cache capacity will be exhausted. More recently, a hardware prefetchers have begun to supplant software prefetching. Typically, they detect a series of cache misses, speculate as to future addresses, and prefetch them into the cache without requiring any user interaction.

In this chapter, we structure our auto-tuned codes to synergize with hardware prefetchers (long unit-stride accesses) but supplement this with software prefetching. This general approach provides performance portability as we make no assumptions as to whether a processor implements software prefetching, hardware prefetching, or both.

13.4.4 Maximizing In-Core Performance

For memory-intensive computations our primary focus should be on minimizing memory traffic and maximizing memory bandwidth. However, it is important not to overlook in-core(cache) performance. Code written without thought to the forms of parallelism required to attain good in-core performance may actually be compute-bound rather than memory-bound. The most common techniques are unroll and jam, permuting or reordering the computation given an unrolling, and SIMDization. We explored all of these via auto-tuning on all three kernels.

13.4.5 Interplay between Benefit and Implementation

The bottleneck that an optimization attempts to alleviate is orthogonal to the scope of the software implementation effort that is required to achieve it. For example, Table 13.2 lists the optimizations used when auto-tuning our three memory-intensive kernels. Loop or code structure transformations have perennially been the only changes allowed by an auto-tuner as they preserve the input and output semantics. Nevertheless, we see many optimizations require an abrogation of this convention as changes to data structures are required for ideal performance.

TABLE 13.2: Interplay between the bottleneck each optimization addresses (parallelism, memory traffic, memory bandwidth, in-core performance) and the primary impact on implementation (code-only, data structures, styles of parallelism). Obviously, changing data or parallelism structure will mandate some code changes. †Efficient SIMD requires data structures be aligned to 128-byte boundaries.

Optimization	Loop/Code Structure	Data Structure	Style of Parallelism
BS SPMD (pthreads)			✓
Decomposition (loop-based)	✓		
		(hierarchical)	✓
Array Padding		✓	
Cache Blocking (loop-based)	✓		
		(sparse)	✓
Cache Bypass (movntpd)	✓		
Reg. Blocking (sparse)		✓	
TLB Blocking (loop-based)	✓		
		(sparse)	✓
Prefetching (software)	✓		
Unroll and Jam	✓		
Reordering	✓		
SIMDization	✓		†

13.5 Automatic Performance Tuning

Given this diversity of computer architectures, performance optimization has become a challenge as optimizing an application for one microarchitecture may result in a substantial performance loss on another. When coupled with the demands to optimize performance in a shorter timeframe than architectural evolution (several new variants of the x86 processor lines appear every year), hand optimizing for each is not practical. To that end, automatic performance tuning, or *auto-tuning* has emerged as a productive approach to tune key computational kernels and even full applications in minutes instead of months [140, 314, 346, 361]. In essence, auto-tuning is built on the premise that if one can enumerate all possible implementations of a kernel, the performance of modern computers allows for the exploration of these variants in less time than a human would require to optimize for one. Moreover, once this auto-tuner has been constructed it can be reused on any evolution of these architectures. The best choice or parameterization for the optimizations in question may be either architecture-dependent, input-dependent, or both. If it is neither, simple optimization will suffice, and auto-tuning is not needed.

Typically, auto-tuning a kernel is divided into three phases: enumeration

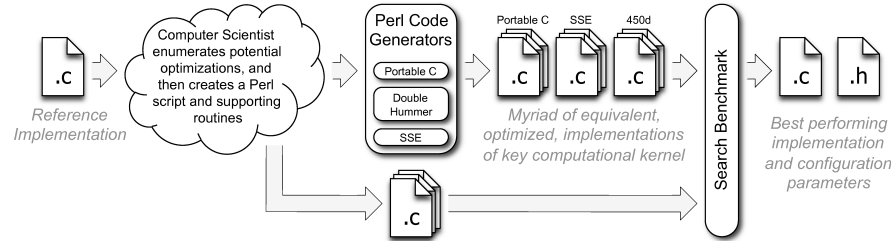


FIGURE 13.6: Generic visualization of the auto-tuning flow.

of potentially valuable optimizations, implementation of a code generator to produce functionally equivalent implementations of said kernel using different combinations of the enumerated optimization space, and implementation of a search component that will benchmark these variants (perhaps using real problem data) in an attempt to find the fastest possible implementation. We may visualize the auto-tuning flow in Figure 13.6, and will discuss the principal components in the following sections.

13.5.1 Code Generation

For purposes of this chapter, we use a simple auto-tuning methodology in which we use a Perl script to generate a few hundred potentially viable parameterizable implementations of a particular kernel. An implementation is a unique code representation that may be parameterized with a run time configuration. For example, cache blocking transforms a naive three nested loop implementation of matrix-matrix multiplication into a six nested loop implementation that is parameterized at runtime with the sizes of the cache blocks (the range of the inner loops). This is still just one variant. However, when one register blocks matrix multiplication, the inner three nested loops are so small (less than 16) it is common to simply fully unroll all loops and create perhaps a few thousand different code variants. When combined with cache blocking, we may have hundreds of individually parameterizable code variants.

Code variants are also needed when dealing with different data structures (i.e., hierarchical instead of flat), styles of parallelism (dataflow instead of bulk synchronous), or even algorithms. Each of these may in turn be parameterized.

As the differences between clusters of code variants may easily be expressed algorithmically, the Perl scripting language provides a pragmatic and productive means of tackling the intellectually-uninspiring task of producing the tens of thousands of lines of code. In essence, the simplest Perl code generation techniques are nothing more than a for loop over a series of `printf`'s. Every line in the resultant C code (function declarations, variables, statements, etc.) maps to a corresponding `printf` in the Perl script.

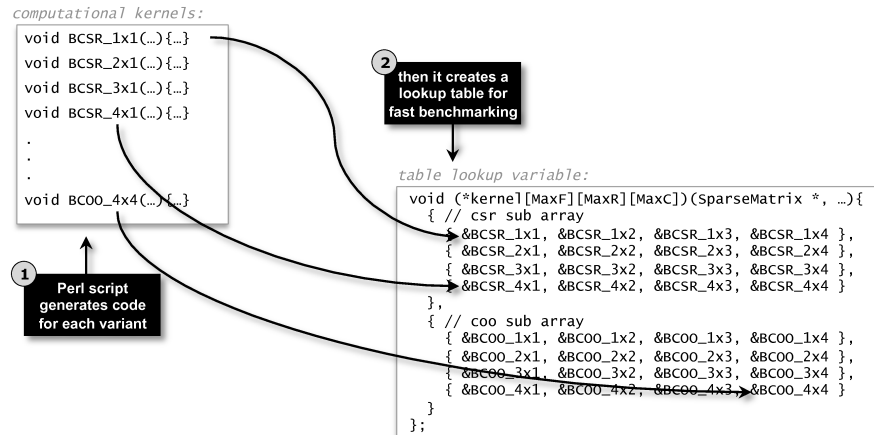


FIGURE 13.7: Using a pointer-to-function table to accelerate auto-tuning search.

13.5.2 Auto-Tuning Benchmark

A Perl script may generate thousands of code variants. Rather than trying to compile an auto-tuning benchmark for each, we integrate and compile all of them into one auto-tuning benchmark. This creates the challenge of selecting the appropriate variant without substantial overhead. To that end, we create an N-dimensional pointer-to-function table indexed by the code variants and possible parameters.

For example, in SpMV we use a 3-dimensional table indexed by kernel type (BCSR, BCOO, etc.) and the register block sizes as measured in rows and columns. As shown in Figure 13.7, the Perl script first generates the code for each kernel variant. It then creates a pointer-to-function table that provides a very fast means of executing any kernel. During execution, one simply calls `kernel1[BCSR][4-1][3-1](...)` to execute the 4×3 BCSR kernel. The auto-tuning benchmark can be constructed to sweep through all possible formats and register blockings (nested for loops). For each combination, the matrix is reformatted and blocked, and the SpMV is benchmarked through a simple function call via the table lookup. This provides a substantial tuning time advantage over the naïve approach of compiling and executing one benchmark for every possible combination. Moreover, it provides a fast runtime solution as well as easy library integration. We’ve demonstrated this technique when auto-tuning dense linear algebra, sparse linear algebra, stencils, and lattice methods.

13.5.3 Search Strategies

Given an auto-tuning benchmark, we must select a traversal of the optimization–parameter space that finds good performance quickly. Over the years, a number of strategies have emerged. In this chapter, we employ three different auto-tuning strategies: exhaustive, greedy, and heuristic. When the optimization–parameter space is small, an exhaustive search implemented as a series of nested loops is often acceptably fast. However, in recent years we’ve observed a combinatoric explosion in the size of the space. As a result, exhaustive search is no longer time- and resource-efficient. As a result, a number of new strategies have emerged designed to efficiently search the space. Greedy algorithms assume the locally optimal choice is also globally optimal. As such, it assumes the best parameterization for each optimization may be explored independently. As such, they may transform a N^D optimization–parameter space of D optimizations each of N possible parameters into a sequential search through D optimizations each of N parameters ($N \times D$ points). Often, with substantial architectural intuition, we may express the best (or very close to best) combination in $O(1)$ time through an arithmetic approach that combines machine parameters and kernel characteristics.

Due to the size of the search space for the Laplacian stencil, we were forced to perform a greedy search algorithm after ordering the optimizations with some architectural intuition. This reduced the predicted tuning time from three months to 10 minutes. Conversely, LBMHD almost essentially uses an exhaustive strategy across seven code variants, each of which could accept over one hundred different parameter combinations. Typical tuning time was less than 30 minutes. SpMV used a combination of heuristics and exhaustive search. The none/cache/TLB blocking variant space was search exhaustively. That is, we benchmarked performance not blocking for either the cache or TLB, blocking for just the cache, and blocking for both the cache and TLB. Unlike the typical dense approach, the parameterization for cache and TLB blocking was obtained heuristically. Similarly, unlike the approach used by Berkeley’s Optimized Sparse Kernel Interface (OSKI) [346], the register blocking was obtained heuristically by examining the resultant memory footprint size for each power-of-two register blocking. However, like LBMHD, the prefetch distance was obtained through an exhaustive search.

13.6 Results

In this section, we present and discuss the results from the application of three different custom auto-tuners to the three benchmarks used in this chapter. Previous papers have performed a detailed performance analysis for these three kernels [108, 363–366].

13.6.1 Laplacian Stencil

Figure 13.8 shows the benefits of auto-tuning the 7-point Laplacian stencil on a 256^3 grid on our three computers as a function of thread concurrency and increasing optimization. Threads are ordered to fully exploit all the cores within a socket before utilizing the second socket. We have condensed all optimizations into two categories: those that may be expressed in a portable C manner, and those that are ISA-specific. The former is a common code base that may be used on any cache-based architecture, not just these three. The latter includes optimizations like explicit SIMDization, and cache bypass. Thus Barcelona and Clovertown use the same x86 ISA-specific auto-tuner, and Blue Gene/P uses a different one. If an architecture with a third ISA were introduced (e.g., SPARC or Cell) it too would need its own ISA-specific auto-tuner. The auto-tuning search strategy uses a problem size-aware, greedy search algorithm in which the optimizations are searched one a time for the best parameterizations.

Clearly, the reference implementation delivers substantially suboptimal performance. As expected on the bandwidth-starved x86 processors, we see the reference implementation shows no scalability as one core may come close to fully saturating the available memory bandwidth.

When the portable C auto-tuner is applied to this kernel, we see that optimizations like cache blocking dramatically reduce superfluous memory traffic allowing substantially better performance. In general, on the x86 processors, we see a saturation of performance at around four cores (one socket), but a jump in performance at eight cores as using the second socket doubles the useable memory bandwidth. However, on NUMA architectures, like Barcelona, this boost is only possible if data is allocated in a NUMA-aware manner.

Rather than hoping the compiler, the non-portable, ISA-specific auto-tuner explicitly SIMDizes the kernel via intrinsics. Unfortunately, this is only useful on compute-bound platforms like Blue Gene. Unfortunately, despite the simplicity of this kernel, the lack of unaligned SIMD loads in the ISA results in less than perfect ($2\times$) benefit. Although explicit SIMDization was not beneficial on the x86 architectures, a different ISA-specific optimization, cache bypass, was useful as it reduces the memory-traffic on a memory-bound kernel. Doing so can substantially improve performance. Unfortunately, compilers will likely never be able to determine when this instruction is useful as it requires run-time knowledge.

In the end, auto-tuning improved the performance at full concurrency by $6.1\times$, $1.9\times$, and $4.5\times$, for Barcelona, Clovertown, and Blue Gene/P respectively. Moreover, thread-parallelizing and auto-tuning the 7-point stencil improved performance by $6.8\times$, $2.3\times$, and $17.8\times$, for Barcelona, Clovertown, and Blue Gene/P respectively. Although substantial effort was required in implementing a x86-specific auto-tuner, it may be reused on all subsequent x86 architectures like Nehalem or Magny-Cours thus amortizing the up front productivity cost.

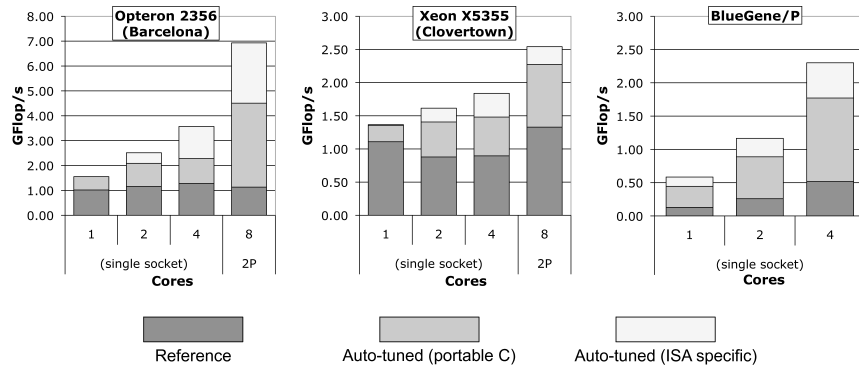


FIGURE 13.8: Benefits of auto-tuning the 7-point Laplacian stencil. Note: this Clovertown is actually the 2.66 GHz E5355 incarnation. As it has the same memory subsystem as the E5345, we expect nearly identical performance on this memory-bound kernel.

13.6.2 Lattice Boltzmann Magnetohydrodynamics (LBMHD)

Figure 13.9 shows the benefits of auto-tuning LBMHD on our three computers as a function of thread concurrency and increasing optimization. Unfortunately, Blue Gene/P does not have enough DRAM to simulate the desired 128^3 problem. As such, it only simulates a 64^3 problem. Once again, we have condensed all optimizations into two categories: those that may be expressed in a portable C manner, and those that are ISA-specific. The auto-tuning search strategy is exhaustive although vectorization is quantized into cache lines.

Although the reference implementation delivers good scalability, simple performance modeling using the Roofline Model (see Chapter 9) suggests it was delivering substantially suboptimal performance. Such a model also explains why even after auto-tuning the bandwidth-starved Clovertown shows poor scalability despite the performance boosts.

The biggest boosts derived from the portable C auto-tuner are NUMA-aware allocation, lattice-aware array padding, and vectorization (to eliminate TLB capacity misses). The Opteron and Blue Gene/P, with moderate machine balance (Flops:bandwidth), see good scaling on this kernel. Conversely, Clovertown, with a high machine balance, sees poor multicore scalability. Whether compute-bound or memory-bound, the non-portable, ISA-specific auto-tuner provided tremendous performance boosts either through explicit SIMDization or cache bypass.

In the end, auto-tuning improved the full concurrency performance by $3.9\times$, $1.6\times$, and $2.4\times$, for Barcelona, Clovertown, and Blue Gene/P respectively. Moreover, the coupling of thread-parallelization and auto-tuning improved LBMHD performance by an impressive $28.3\times$, $8.9\times$, and $9.2\times$,

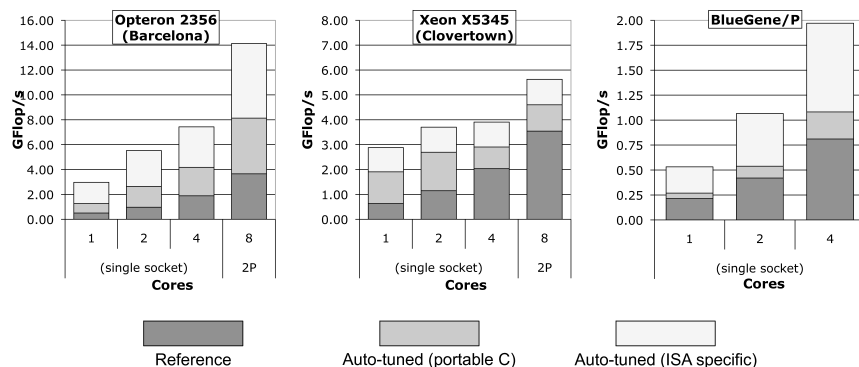


FIGURE 13.9: Benefits of auto-tuning the lattice Boltzmann Magnetohydrodynamics (LBMHD) application.

for Barcelona, Clovertown, and Blue Gene/P respectively. Clearly, the ISA-specific auto-tuners were critical in achieving these speedups.

13.6.3 Sparse Matrix-Vector Multiplication (SpMV)

Figure 13.10 shows the benefits of auto-tuning SpMV on our three computers. Unlike the previous two figures, where optimization and benefit is basically independent of problem size, the horizontal axis in Figure 13.10 represents different problems (matrices). The ordering preserves that in Figure 13.4. The lowest bar is the untuned serial performance, while the middle bar represents untuned performance using the maximum number of cores. The top bar is the tuned (portable C) performance using the maximum number of cores. Unlike the other kernels, a non-portable ISA-specific auto-tuner was not implemented for SpMV.

The auto-tuning search strategy is somewhat more complex. Register, cache and TLB blocking use a footprint minimization heuristic based on cache and TLB topology, where prefetching is based on an exhaustive search quantized into cache lines.

We observe a trimodal performance classification: problems where both the vectors and matrix fit in cache, problems where only the vectors can be kept in cache, and problems where neither the matrix nor the vectors can be kept in cache. Clearly, on Barcelona, no matrix ever fits in the relatively small cache, but the performance differences between the problems where the vectors fit can be clearly seen. On Clovertown, where the cache grows from 4 Mbyte to 16 Mbyte using all eight cores, we can see the three matrices that get a substantial performance boost through utilization of all the cache and compression of the matrices. Blue Gene/P, like Barcelona can never fit any matrix in cache, but we can see the matrices (*Economics* through *Linear Programming*) where the vectors don't fit.

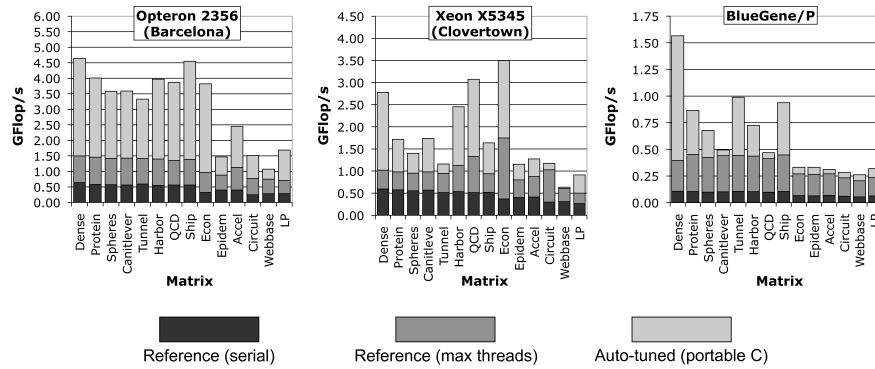


FIGURE 13.10: Auto-tuning Sparse Matrix-Vector Multiplication. Note: horizontal axis is the matrix (problem) and multicore scalability is not shown.

As it turns out, on Barcelona, NUMA-aware allocation was an essential optimization across all matrices. Across all architectures, matrix compression delivered substantial performance boosts on certain matrices. Interestingly, on Blue Gene/P, matrix compression improved performance by a degree greater than the reduction in memory traffic — an effect attributable to an initially compute-bound reference implementation. Interestingly, TLB blocking delivered substantial performance boosts on only one matrix, the extreme aspect ratio linear programming problem.

We observe that threading alone provided median speedups of $1.9\times$, $2.5\times$, and $4.2\times$ on Barcelona, Clovertown, and Blue Gene/P. Clearly, only Blue Gene/P showed reasonable scalability. However, when coupled with auto-tuning, we observe median speedups of $3.1\times$, $6.3\times$, and $5.1\times$ and maximum speedups of $9.5\times$, $11.9\times$, and $14.6\times$. Ultimately, performance is hampered by memory bandwidth on both Barcelona and Clovertown, leading to sublinear scaling. As a result, auto-tuning strategies targeted at reducing memory traffic are critical.

13.7 Summary

Naïvely, one might expect that nothing can be done to improve the performance of memory-intensive or memory-bound kernels like stencils, LBMHD, or SpMV. However, in this chapter, we discussed a breadth of useful optimizations applicable not only to our three example kernels but to many others domains. Unfortunately, no human could explore all the parameterizations of these optimizations by hand. To that end, we showed how automatic performance tuning, or *auto-tuning*, can productively tune code and thereby dramat-

ically improve performance across the breadth of architectures that currently dominate the top500 list. Unfortunately, to achieve the best performance, non-portable ISA-specific auto-tuners that generate explicitly SIMDized code are required.

13.8 Acknowledgments

We would like to express our gratitude to Forschungszentrum Jülich for access to their Blue Gene machine. 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.