Accelerating TIME-TO-SOLUTION for Computational SCIENCE and ENGINEERING

To minimize the time-to-solution of a computational science or engineering problem, the time to write and also run the program must both be considered. Generally, there is a tradeoff: you can spend more time optimizing the program and then the computer has less time to run it, or you can spend less time optimizing and give the computer more runtime. The goal of autotuning is to avoid this tradeoff by having computers automatically optimize programs because they can tune faster — and often better — than humans, and then humans can spend more of their time on science itself.

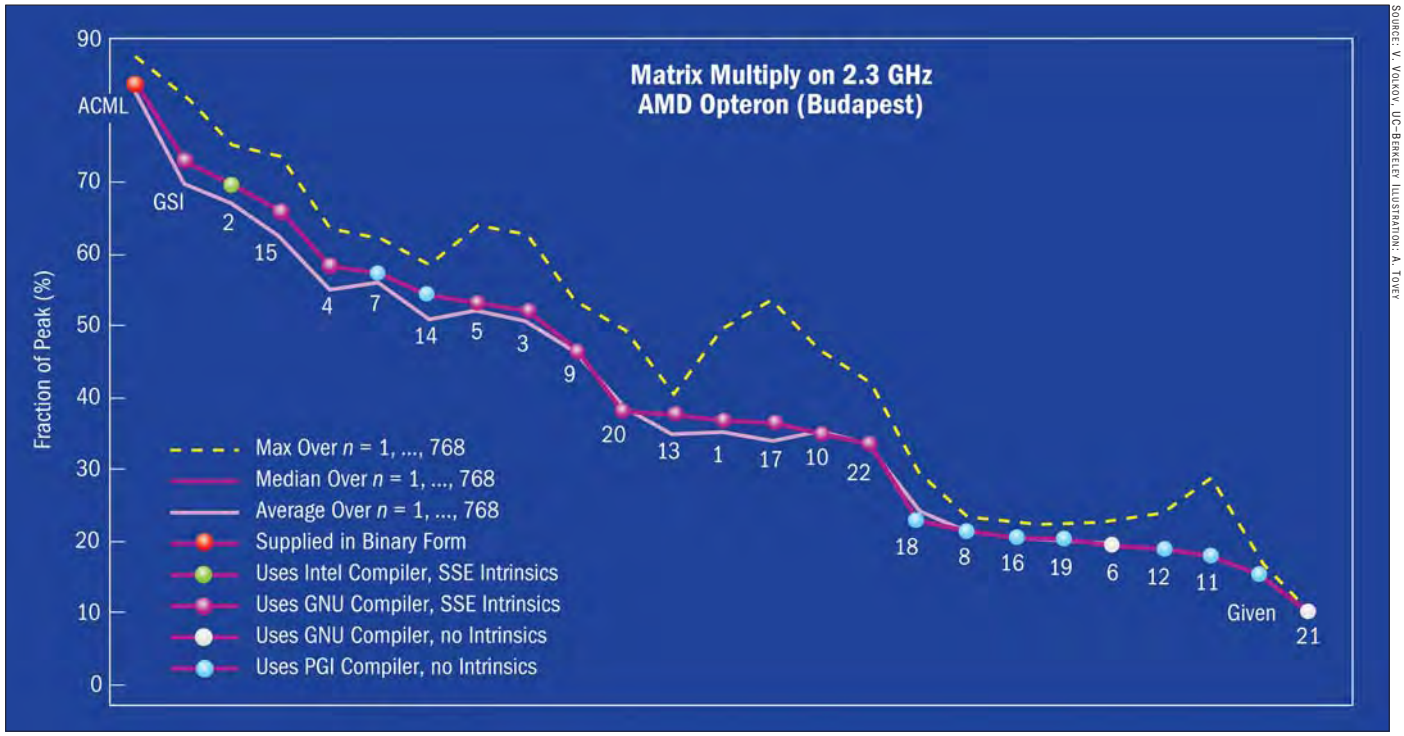
The need to automate tuning is exacerbated by both hardware and software innovations. The accelerating change in computing hardware (examples include increasing parallelism, deeper memory hierarchies, and heterogeneous processors such as general purpose graphics processing units) means that programs frequently need to be retuned to benefit from these innovations. Likewise, when software is modified to introduce a new mathematical model or algorithm, tuning often needs to reoccur to achieve a comparable level of performance.

Dense linear algebra libraries are widely used examples of autotuning. The availability of tuned Basic Linear Algebra Subroutine (BLAS) libraries across many architectures led to the incorporation of the Linear Algebra Package (LAPACK) library into MatLab, and fast Fourier transforms (FFT), where the fastest FFTs on many architectures are produced automatically by systems like FFTW and Spiral (see Further Reading, p57).

Autotuners have used diverse techniques, including the following:

- Varying code organization and generation, data structures, high-level algorithms and even combinations of hardware and software — such as the Lawrence Berkeley National Laboratory (LBNL) Green Flash project — to generate a large design space to search for the best implementation
- Measuring the performance of implementations in this design space to identify the fastest one, possibly using hardware performance counters to assess detailed hardware behavior
- Using code synthesis and correctness proofs to automatically generate complicated members of the design space (perhaps filling in difficult corner cases in a provably correct manner)
- Using statistical machine learning techniques and/or simplified performance models to more efficiently search or prune the possibly enormous design space
- Creating a database of tuning rules, growing it as more become known, and applying it automatically to search an ever larger design space (that is, avoiding the reinventing of wheels); for example, Spiral incorporates the knowledge of 50 papers describing different divide-and-conquer FFT algorithms, which it can combine in arbitrary ways to create new ones
- Creating a database of historical tuning results, so that the results of prior searches can easily be reused
- Allowing the user to give hints or otherwise steer the tuning process

The accelerating change in computing hardware and software means that programs frequently need to be re-tuned to benefit from these innovations.



SOURCE: V. VOLKOV, UC-BERKELEY ILLUSTRATION: A. TOVEY

Figure 1. *Performance of tuned matrix multiplication for student teams.*

A variety of user interfaces have also been effectively applied to autotuners; ideally, such an interface lets the user describe the desired computation as simply as possible and hides the details of how the above mechanisms accomplish tuning. The simplest user interface is to call a library routine that has been autotuned for the desired architecture. Dense linear algebra and FFTs were already mentioned as an example of this. Such autotuning can in principle occur at library install-time (or earlier), and thus it can take a long time (hours or more) to find the best implementation, without inconveniencing the user.

Not every computation can be described by a few concise parameters thereby lending itself to incorporation in such a pre-built library. Some autotuning interfaces require a search to be done at runtime, once enough information is available, and this search must obviously be done quickly. The autotuner might still provide a library interface, but now the autotuner may want to use statistical techniques, user hints, and a database of historical tuning results to limit the runtime search. An example of tuning a sparse matrix operation that requires detailed knowledge of the sparsity structure is given in the section "Tuning Sparse Matrix Computations" (p50).

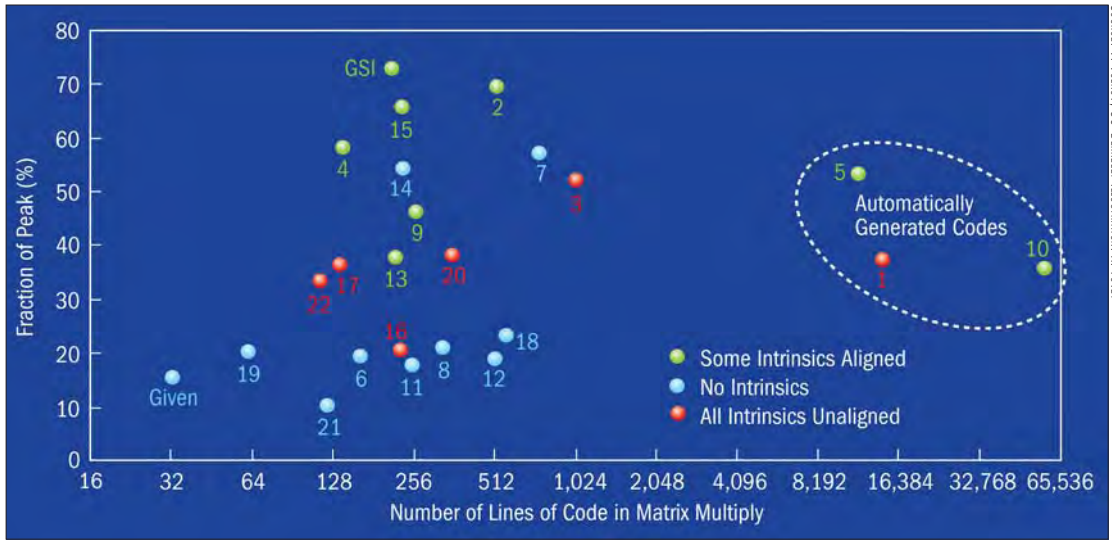
Sometimes a library interface is simply too restrictive to describe the computation, which is best described by writing a simplified version in some programming language. An example is a stencil operation, where an arbitrary but identical function of the values of the neighbors of each point in a 1D, 2D, or higher-dimensional mesh is computed. These values may actually be arrays or other data structures, as long as the same data structure is used at each point in the mesh. The simplest example is the 1D Laplacian, where one computes $A_{new}(i) = A(i\text{-}1)\text{-}2A(i)+A(i+1)$ at each mesh point i. At the other extreme is lattice-Boltzmann magnetohydrodynamics (LBMHD), where 79 values representing various physical quantities are stored at each point in a 3D mesh and combined in a complicated nonlinear function requiring hundreds of lines of code to express. The section "Tuning Stencil Operations" (p53) describes successful attempts to autotune these and similar stencil computations, with speedups of up to 132 times for LBMHD on the Cell processor.

Because complicated computations like LBMHD are best expressed in a programming language, what should be the role of compilers in the autotuning endeavor? The stencil autotuners described above have used tools like PERL scripts to generate code variants in the design space, which then need to be rewritten for each new stencil. Ideally, a domain-specific compiler would be able to take (suitably annotated) stencil code and generate the design space with much less user effort, and attempts to do this are under way.

Indeed, autotuning's high-level goal of mapping simply described computations into highly-optimized, machine-specific implementations sounds very much like the goal of compilation. Compilers have not traditionally used many of the autotuning techniques listed above, such as changing

Indeed, autotuning's high-level goal of mapping simply described computations into highly-optimized, machine-specific implementations sounds very much like the goal of compilation.

Saving energy is increasingly important, and while it is usually strongly correlated with time-to-solution, scenarios can be imagined where a different optimum is reached for reducing energy than reducing time.



SOURCE: V. VOLKOV, UC–BERKELEY ILLUSTRATION: A. TOVEY

Figure 2. *Performance and lines of code of tuned matrix multiplication for student teams.*

algorithms or using measurement to choose the best implementation. But the potential performance improvements offered by these approaches have inspired the compiler community to actively research how they could be included in compilers. Work on autotuners and compilers is becoming closely related and synergistic.

Finally, performance (time-to-solution) is not the only success metric an autotuner could have. Saving energy is increasingly important, and while it is usually strongly correlated with time-to-solution, scenarios can be imagined where a different optimum is reached for reducing energy than reducing time. For example, a less parallel algorithm that permits part of the system to be powered down might use less energy overall, or an algorithm might trade energy-hungry data movement for additional but more efficient arithmetic. Also, autotuning is beginning to be used for hardware/software co-design, where a set of algorithms and hardware parameters (such as cache sizes) is simultaneously searched to find the best overall design for special purpose processors, where the success metric might involve time-to-solution, power, and chip area.

Tuning Dense Linear Algebra

To measure the difficulty of hand-tuning code, an experiment can be run where clever but inexperienced high-performance computing programmers are trained in the basics of performance tuning and given a fixed amount of time to tune a particular piece of code. Then, the performance of the tuned code can be compared to a very highly-tuned version. At the University of California (UC)–Berkeley we run this experiment every year in the first homework assignment of our parallel computing class, CS267 (see Further Reading, p57).

The students are given a few weeks to tune a dense matrix multiplication code. They hear lectures on general low-level tuning techniques (such as use of Single Instruction, Multiple Data (SIMD) instructions and prefetching) and how to tune matrix multiplication in particular (by blocking). Then they are given a code with basic optimizations (that is, blocked for one level of memory hierarchy). The students, mostly graduate students (about half are computer scientists), are assigned to interdisciplinary teams. Figure 1 (p47) shows the results of the student teams using the Franklin Cray XT machine at the National Energy Research Scientific Computing (NERSC) Center, measured by fraction of machine peak attained and sorted in decreasing order of (median) performance attained — from the vendor-supplied tuned version (ACML), the teaching assistant's version (GSI), down to the supplied version ("given"). The data points are color coded based on compiler and hardware features. Figure 2 shows the same experiment, but where the horizontal axis is now the number of lines of code (on a log scale). The lesson from this data is that even for a well-understood kernel (sidebar "No, Not that Kind of Kernel") like matrix multiplication, tuning for performance is critical and difficult, even for clever UC–Berkeley graduate students. Therefore, it is desirable for an autotuner to do it automatically.

Why hand tuning is difficult is illustrated with a small slice of the very large parameter space that must be searched to find the best implementation of matrix multiplication. Figure 4 plots performance as a function of just two block sizes, m_0 and n_0, which determines the size of a small subblock of the matrix that must fit in the 16 floating point registers of a Sun Ultra IIi; thus we need only consider $m_0 * n_0 \leq 16$. Each such square subblock in figure 4 represents an (unrolled) implementation,

No, Not that Kind of Kernel

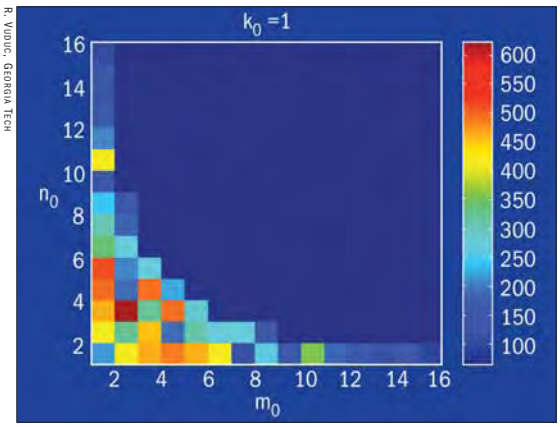
Unfortunately, the term "kernel" has become heavily overloaded even within computer science. Often in our applications the bulk of the run time is spent executing a tiny fraction of the code base. As such, we in the scientific computing community refer to these key loop nests as kernels — not to be confused with the identically named core component of an operating system. Floating-point kernels include operations like matrix–matrix multiplication (figure 3), fast Fourier transforms, stencil sweeps, interpolations, and many more complicated operations.

The un-optimized versions of these kernels can often be represented in less than a hundred lines. However, the nested loop bounds are often so large that millions, perhaps billions, of floating-point operations are performed per invocation. Moreover, the kernel may be called thousands of times in an application. As a result, optimizing this compact kernel may dramatically accelerate application performance.

```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  C[i][j] = 0.0;
  for(k=0;k<N;k++){
   C[i][j]+=A[i][k]*B[k][j];
  }
 }
}
```

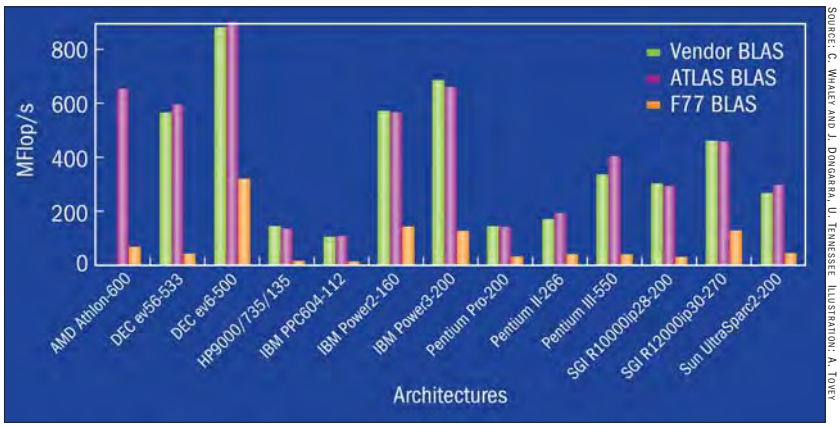
ILLUSTRATION: A. TOVEY

Figure 3. *Matrix–matrix multiplication kernel*



R. VUDUC, GEORGIA TECH

Figure 4. *Performance of matrix multiplication as a function of register block sizes on a Sun Ultra IIi.*



SOURCE: C. WHALEY AND J. DONGARRA, U. TENNESSEE ILLUSTRATION: A. TOVEY

Figure 5. *Performance of matrix multiplication using ATLAS, the vendor library, and unoptimized F77 on a variety of architectures.*

color coded by performance, from approximately 100 Mflop/s (dark blue) to approximately 600 Mflop/s (dark red). The fastest algorithm by far is for $m_0 = 2, n_0 = 3$. There is no simple explanation for why this is the needle in the haystack that we seek, but the purpose of autotuning is not to explain why, but simply to find it as quickly as possible.

Homework assignments can sometimes have unintended side-effects. When the assignment above was used in a CS267 class in the mid-1990s, two student teams beat the vendor code. Intrigued by the results, the teaching assistant (Jeff Bilmes) and another graduate student (Krste Asanovic) studied the students' work and built the first prototype autotuner for matrix multiplication called Portable High Performance ANSI C (PHiPAC). In turn, Clint Whaley and Jack Dongarra were inspired to produce a more complete and portable autotuner for matrix multiplication and eventually all the BLAS, called ATLAS. ATLAS begins by detecting specific hardware properties: the cache sizes, the floating point pipeline length, and so on. Then it systematically explores the different possible implementations (say of matrix–matrix multiplication), of which there can be hundreds of thousands of variations. After eliminating unlikely candidates by using heuristics based on gathered information, ATLAS generates code to implement the remaining alternatives, and then compiles, executes, and times them all to choose the fastest.

Figure 5 compares the performance of matrix multiplication from ATLAS, the vendor library, and the un-optimized F77 version across a wide variety of architectures. ATLAS is about as fast as the vendor library (when it exists), sometimes faster, and much faster than the un-optimized code.

Before ATLAS, vendors charged significant prices for their tuned libraries, which discouraged some independent software vendors from using them in their products. ATLAS removed this obstacle, which had significant implications for commercial software. For example, the developers of MATLAB,

ATLAS generates code to implement the remaining alternatives, and then compiles, executes, and times them all to choose the fastest.

R. Vuduc, Georgia Tech

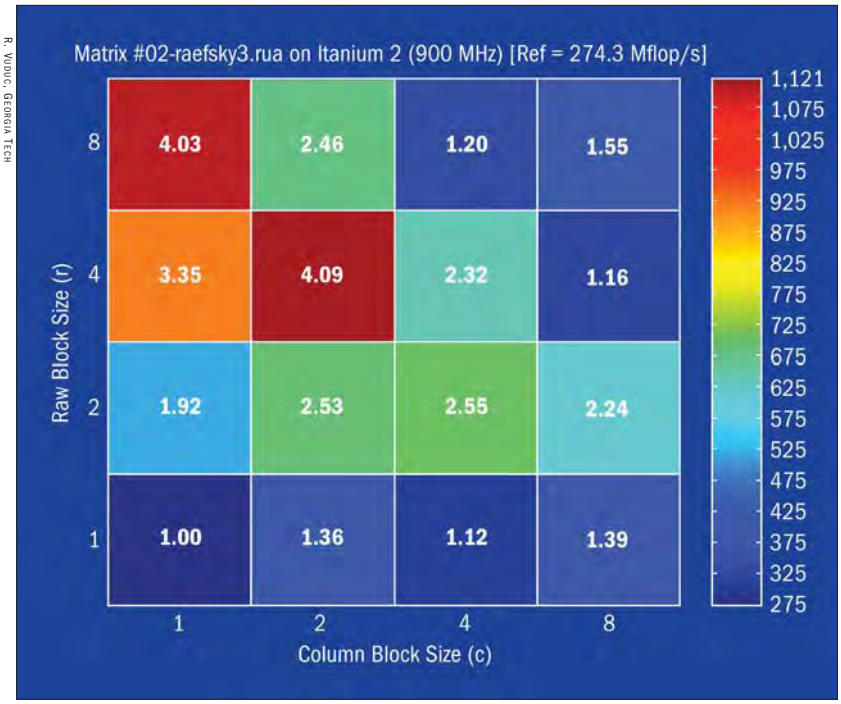


Figure 6. *Performance of Blocked SpMV for different block sizes on the raefsky matrix on Itanium 2.*

Source: B. Lee, UC-Berkeley; and R. Vuduc, Georgia Tech Illustration: A. Tovey

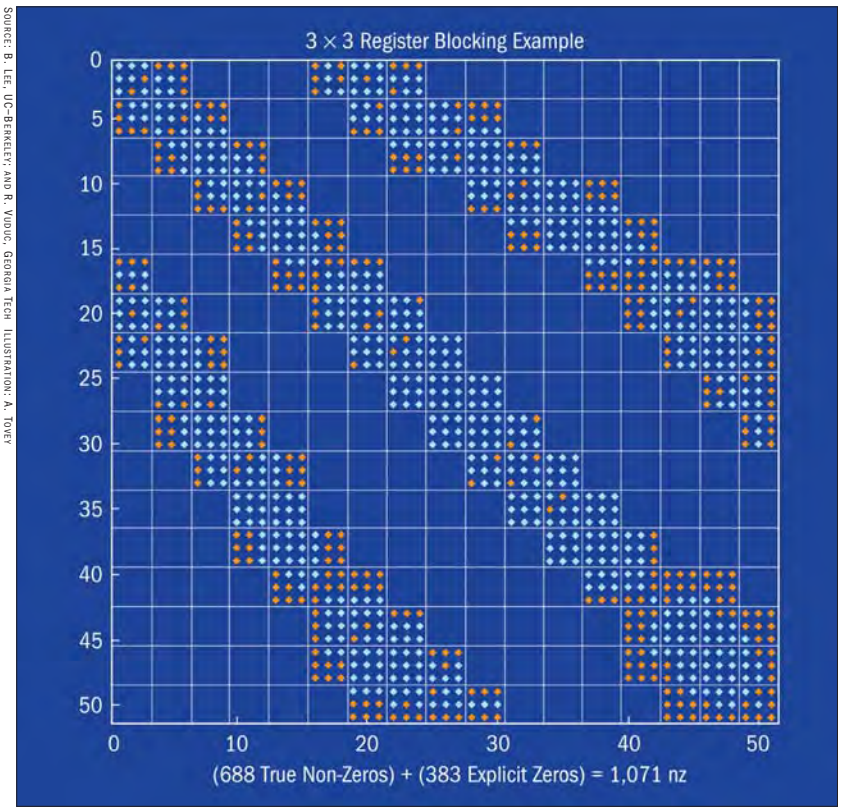


Figure 7. *Sparsity pattern (blue) of the leading part of ex11 matrix.*

MathWorks, incorporated the LAPACK linear algebra library into their product. Now, concise MatLab statements like $x = A\backslash b$ for solving the linear system $Ax = b$ uses LAPACK's solvers and benefits from the tuned BLAS on whose performance LAPACK depends. LAPACK and its associated libraries ScaLAPACK, CLAPACK, and LAPACK95 are the most widely used dense linear algebra libraries, with more than 100 million hits at www.netlib.org. They have been adopted for use by Advanced Micro Devices, Apple (under Mac OS X), Cray, Fujitsu, Hewlett Packard, IBM, Intel, NEC, SGI, several Linux distributions (such as Debian), Numerical Algorithms Group, IMSL, Interactive Supercomputing, and Portland Group.

Ultimately, some vendors have also adopted autotuning as a tool to produce their own libraries. Greg Henry of the Intel Math Kernel Library team says they use autotuning as a tool now and are looking for more ways to take advantage of it.

LAPACK and ScaLAPACK based on tuned matrix multiplication are not the end of the story, because neither one works as well as possible (and sometimes not well at all) on emerging multicore, GPU, and heterogeneous architectures. New ways of organizing the algorithms, new data structures, and even new algorithms with different numerical convergence and stability properties are emerging as the algorithms of choice on these architectures. Indeed, a new theory of communication-avoiding algorithms shows how to construct algorithms that do asymptotically less data movement than the algorithms in LAPACK and ScaLAPACK. As communication costs continue to grow exponentially more expensive relative to floating point costs, these new communication-avoiding algorithms may well become the algorithms of choice. As the set of possible algorithms and implementations continues to grow, tools for more easily generating members of this set become important. The sidebar "Optimization versus Autotuning: A Visual Guide" discusses performance on emerging architectures.

Tuning Sparse Matrix Computations

With dense matrices, only a few parameters (the dimensions) are needed to define a problem, such as multiplication, that permits offline autotuning. In principle, this autotuning takes as much time as needed — whether hours or weeks — to search for the best implementation for each dimension (or range of dimensions), which can then be packaged in a library for use at runtime.

In contrast, to tune a sparse algorithm like sparse-matrix (dense) vector multiplication (SpMV), the sparsity pattern of the matrix needs be to known, but it generally is not until runtime. Even if many SpMVs are done with a given matrix (pattern), too much time cannot be spent at runtime to tune. The approach taken by Optimized Sparse Kernel Interface (OSKI) (Further Reading, p57), an autotuner for SpMV, is used to illustrate.

Optimization versus Autotuning: A Visual Guide

We may define the quality of performance based on a fraction of algorithmic peak — a speed of light model that sets an upper bound to performance. Given a reference implementation of a computational kernel, the processors in supercomputers today (spanning nearly a decade of processor architectures) may achieve dramatically different levels of performance across a range of computational kernels. Qualitatively speaking, we observe a general decline in performance on ever-newer architectures. In other words, it is becoming increasingly difficult to achieve quality performance on novel architectures. We may qualitatively visualize this trend in figure 8(a). Here we see 18 nondescript computational kernels from three domains ("dwarfs" or "motifs" in *Berkeley View* parlance). The quality of performance is visualized as a heat map, where red is ideal performance.

To rectify this performance disparity, individual groups within our community may optimize their application or kernel of interest for the processor available to them at the time. In doing so, they may achieve ideal performance for that particular processor-kernel combination (figure 8(b)). This optimized implementation may boost performance on some existing architectures in the DOE inventory as well as on some future machines, but, unfortunately, as architectures and even instruction set architectures continue to evolve (such as streaming SIMD extensions, multicore, Advanced Vector Extensions, and so on), there is a half-life associated with these optimizations. As a result, eventually the performance advantage will be lost.

Automatic performance tuning attempts to generalize the optimization process on a kernel-by-kernel basis by first parameterizing the optimizations, then searching for the appropriate parameters for each processor. The benefit is that for all the autotuned kernels, we may achieve near ideal performance on all existing architectures (figure 8(c)). Although the autotuner may improve performance on future architectures, it will need periodical refurbishing as processor designers introduce new architectural paradigms. Nevertheless, applications that use the same computational kernels may productively leverage the benefit of autotuned kernels to accelerate their own performance.

Although either hand optimization or autotuning of a particular kernel will boost the performance on a range of architectures (vertical structures in figure 8), it will not boost performance on unrelated kernels.

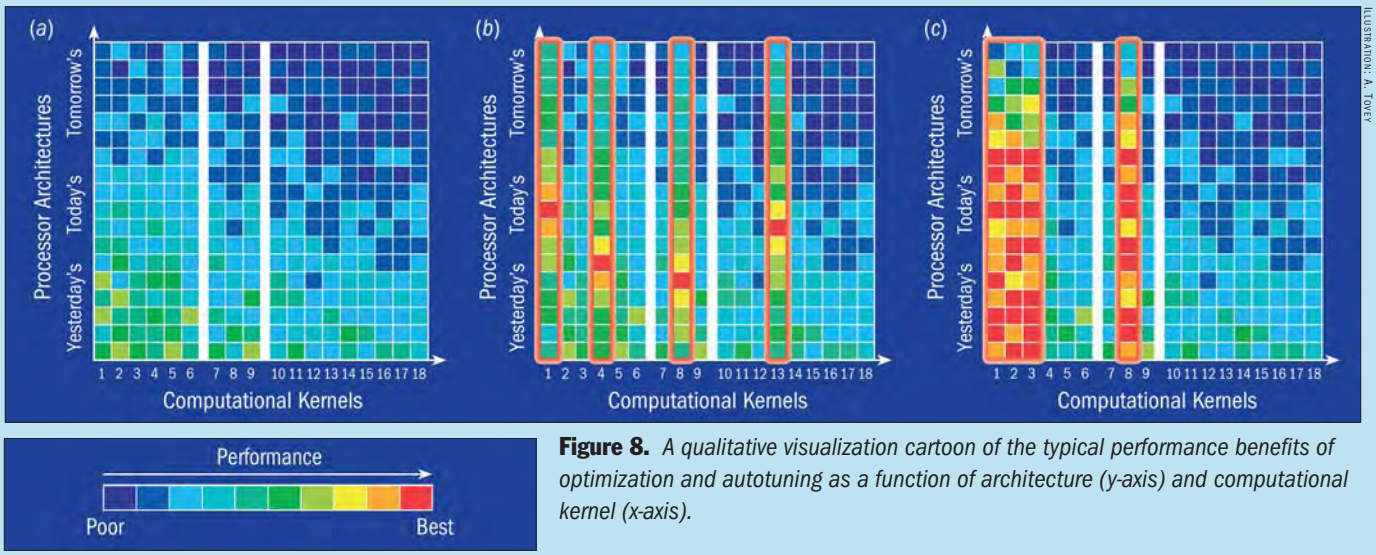


ILLUSTRATION: A. TOVEY

Figure 8. *A qualitative visualization cartoon of the typical performance benefits of optimization and autotuning as a function of architecture (y-axis) and computational kernel (x-axis).*

Because SpMV performs only two arithmetic operations per nonzero matrix entry, performance is memory-bound. A critical goal of tuning then is to choose a sparse matrix data structure that is as compressed as possible to minimize memory traffic. The conventional data structure stores the value of each nonzero entry, its column (or row) index, and where each row (resp. column) begins and ends. The nonzero values are unavoidable, but clever data structures can eliminate most of the indices. For example, a matrix from a finite-element model typically consists of many small dense $r \times c$ blocks (one per element), so storing one index per block instead of one index per nonzero reduces the number of indices required by a factor $r \cdot c$ — a great improvement.

On a typical structural analysis matrix (raefsky, from the University of Florida sparse matrix collection) that consists entirely of 8×8 dense blocks, can the number of indices be reduced by a factor of 64? Each 8×8 block can be stored as a collection of 4×4 blocks, 2×8 blocks, and so on, with 16 possibilities in all (for all possible values of r and c chosen from {1, 2, 4, 8}). By implementing SpMV with raefsky in all 16 possible ways, the performance is measured (on an Itanium 2). The results are shown in figure 6,

R. Vuduc, Georgia Tech

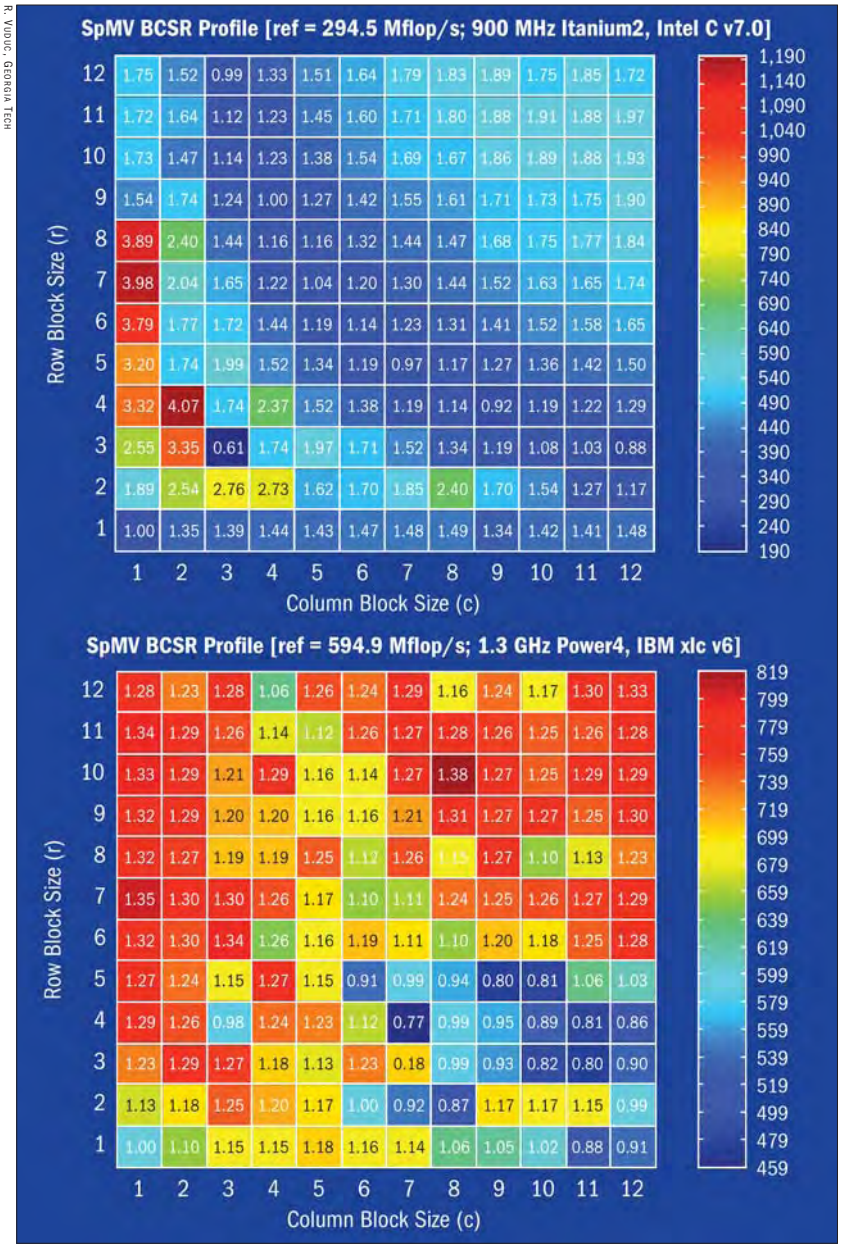


Figure 9. *(a) Performance of blocked SpMV for various block sizes on an Intel Itanium 2; (b) performance of blocked SpMV for various block sizes on an IBM Power 4.*

where each square represents an implementation, color coded by speed in Mflop/s, and labeled by speedup versus the 1 × 1 case in the lower left. The 8 × 8 case goes 1.55 times faster, but the fastest is 4 × 2, which goes 4.09 times faster. As before, why this is true is a complicated story, but from the user's point of view all that matters is that the autotuner can figure this out and reorganize the data structure and SpMV implementation accordingly.

To pick the best data structure we must sometimes go beyond storing the nonzero entries in blocks. Consider the ex11 test matrix from the same collection as raefsky; a "spyplot" zoomed in to the top-left 100 × 100 corner shows the nonzero entries as blue dots in figure 7 (p50). No obvious block structure is evident; if one were to insist on using, say, 3 × 3 blocks as shown in figure 7 (p50), then all the zero entries shown in red would have to be stored as explicit zeros within those blocks. The effect would be an increase in the number of explicit matrix entries stored by 1.5 times and also the number of floating point operations needed to perform SpMV by 1.5 times. This hardly seems like a good idea, but in fact it speeds up SpMV by a factor 1.5, because the overall floating point rate increases a factor 2.25 times, more than overcoming the increase in the number of floating point operations.

How does the autotuner quickly figure this out at runtime? OSKI begins by running an offline benchmark to characterize how fast SpMV runs for all possible different block sizes $r \times c$, from 1×1 to 12×12. This benchmark leads to an image similar to figure 6 (p50), which can differ dramatically from machine to machine. Two examples are shown in figure 9, an Intel Itanium 2 at the top and an IBM Power 4 at the bottom. At runtime, OSKI does a quick statistical sampling of the user's matrix to estimate the number of extra zero entries that would be stored using any $r \times c$ block structure. By combining for each $r \times c$ the estimated speed with the estimated number of floating point operations, OSKI can estimate the runtime of SpMV and pick the value of $r \times c$ that minimizes it.

For this run-time optimization, OSKI may need 5–40 times the cost of a single un-optimized SpMV. Most of this is the cost of copying the matrix from the old to the new data structure. It is only worthwhile if the user intends to do a fairly large number of SpMVs, which is frequently the case in practice. But because of this overhead and also because OSKI can exploit tuning hints only the user can supply — examples include "my matrix is symmetric," or "it is OK to reorder the rows and columns of my matrix to accelerate SpMV, because I can adjust the rest of my algorithm accordingly," or "this is the same as the matrix I used last week, called raefsky, so please just use the same optimizations" — OSKI's interface may require more user effort than just calling a library.

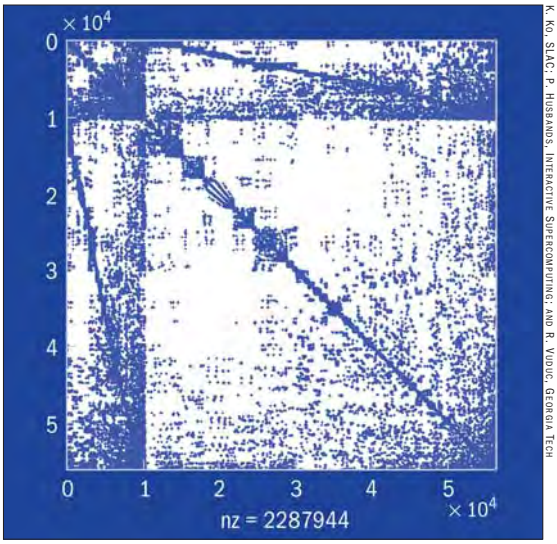
We illustrate this by applying OSKI to a matrix arising in accelerator cavity design, which was supplied by Kwok Ko and Parry Husbands. Figure 10 shows the original symmetric matrix, and figure 11 shows it after reordering its rows and columns to "push" its nonzero entries toward the diagonal, in order to create more dense blocks. Figure 12 zooms in to the leading 100 × 100 block and shows the locations of the original matrix entries before reordering (red and green dots) and after reordering (blue and green dots). It is evident that the reordering created much larger blocks for OSKI to exploit and leads to speedups of 1.7 times on Itanium 2, 2.1 times on Pentium 4, 2.1 times on Power 4, and 3.3 times on Ultra 3.

Lastly, we illustrate some optimizations for multicore platforms to be included in a future version of OSKI. SpMV seems straightforward to parallelize because each group of matrix rows may be handled as an independent SpMV, but in fact more care is needed to attain high performance. Figure 13 (p54) shows the speedups of SpMV on three different multicore platforms, for 13 different sparse matrices, and for a variety of different optimization techniques. Each vertical bar shows speed of SpMV for a particular matrix on a particular platform (so up is good), color-coded by the extra speed resulting from the labeled optimization technique. Optimizations referred to are Parallel (using PThreads to evenly divide the work, by rows, over the available hardware threads: eight hardware threads on Xeon, eight on Opteron, and 128 on Ultrasparc); NUMA (to account for non-uniform memory access on Opteron and Ultrasparc); Prefetch (to overlap communication and computation); and Compression (including $r \times c$ blocking as described above). The best speedups over the parallelized naïve code were 2.7 times on Xeon, 4 times on Opteron, and 2.9 times on Ultrasparc, and required all these optimizations to be used. This complexity should be hidden from the user in an autotuner.

Tuning SpMV is not the end of the story for sparse linear algebra, just as tuning matrix-multiplication was not the end of the story for dense linear algebra. Just as in the dense case, by looking at an entire algorithm rather than just the kernel, it is possible to find communication-avoiding algorithms that minimize data movement. This search is even more important in the sparse case than the dense case, because sparse algorithms are naturally dominated by their data movement costs. Briefly, a typical iterative method for solving $Ax = b$ seeks to find an "optimal solution" (in some sense) lying in the so-called Krylov subspace spanned by the vectors $W = [b, Ab, A^2b, A^3b, \ldots, A^kb]$. This class includes well-known algorithms like conjugate-gradients and generalized minimum residuals. The conventional implementations form a basis for the space spanned by W by calling SpMV k times, so that the communication cost grows proportionally to k. The new method computes a different basis of the same subspace for (roughly) the communication cost of just one call to SpMV, under reasonable assumptions about the sparsity pattern of A. This new set of algorithms greatly expands the tuning space and requires the autotuning of algorithms consisting of multiple interacting kernels that cannot be tuned independently.

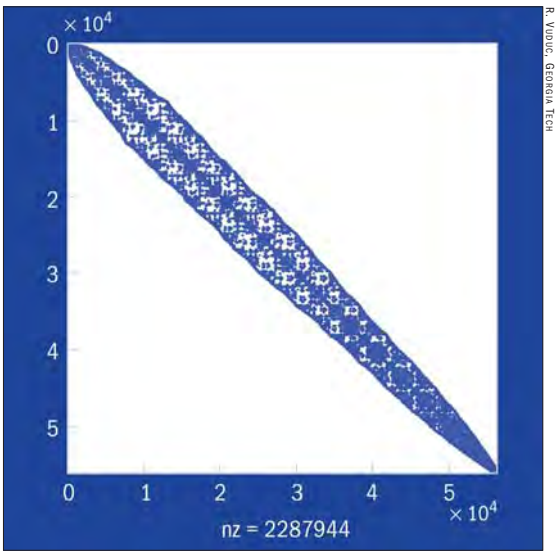
Tuning Stencil Operations

A stencil operation refers to a data structure with a common data format at each point of a 1D, 2D, or higher-dimensional mesh, not necessarily rectangular, and the computation at each mesh point



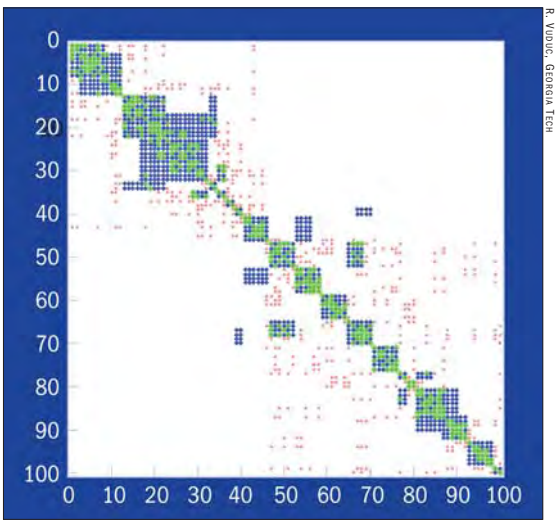
K. Ko, SLAC; P. Husbands, Interactive Supercomputing; and R. Vuduc, Georgia Tech

Figure 10. *The sparsity pattern of an accelerator cavity design matrix.*



R. Vuduc, Georgia Tech

Figure 11. *The sparsity pattern of an accelerator cavity design matrix after reordering.*



R. Vuduc, Georgia Tech

Figure 12. *The sparsity pattern of an accelerator cavity design matrix after reordering, zoomed into top right corner.*

Each of these optimizations is critical for performance on at least one machine. Trying them all is a job to be automated by an autotuner.

SOURCE: S. WILLIAMS, LBNL ILLUSTRATION: A. TOVEY

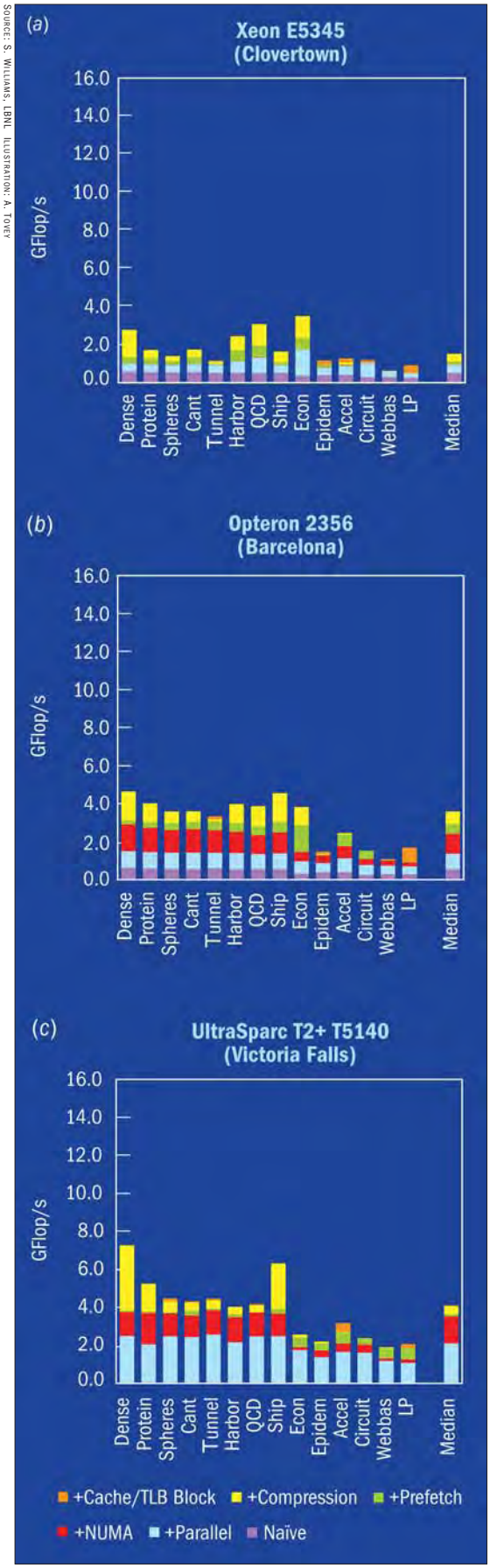


Figure 13. *Autotuning SpMV on multicore (performance versus matrix).*

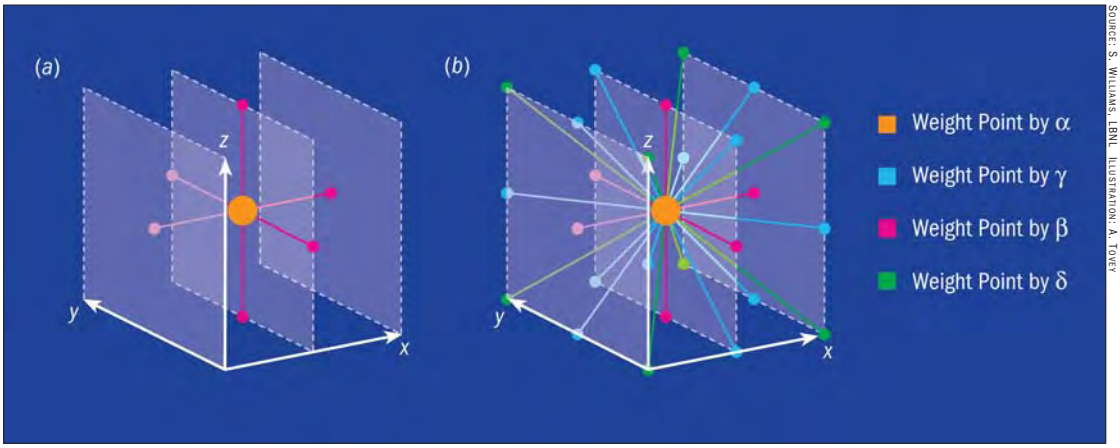
of the same function *F*(.) of the data at the mesh point and selected neighbors. The data format at each mesh point and the function *F*(.) can be arbitrarily complicated (for example, LBMHD) or as simple as computing some weighted average of scalar values at the neighbors (such as the Laplacian). When the function is linear, this is mathematically equivalent to SpMV, but the special structure (same nonzero entries in each row) allows us to perform even more aggressive optimizations than in sparse matrix computations.

Optimizations for Stencils

Consider two different 3D stencils on a cube of scalar data: the seven-point stencil computes a weighted average of the data at the mesh point and its six nearest neighbors, and the 27-point stencil uses the $3^3 - 1 = 26$ nearest neighbors (figure 14). In the seven-point case, instead of seven different coefficients to weight the values, there is only *X*, and in the 27-point case, there is only *Y*. This arrangement permits the use of common subexpression elimination as an optimization (that is, factoring out the common coefficient).

The simplest implementation of a stencil would simply loop over the mesh points (in the order in which they are stored in memory) and apply the function. The simplest parallel implementation would assign disjoint parts of the mesh (say slabs) to different processors. This "naïve" code is shown as the purple bar at the bottom of each performance plot in figure 15 (p56) for seven-point stencils and in figure 16 (p57) for 27-point stencils. The other colored bars show the importance of an entire sequence of other autotuning optimizations, including NUMA awareness (so that processors work on data in their local memories), padding (to keep data aligned on cache boundaries), core blocking and register blocking (to use the fast memory hierarchy levels most effectively), software prefetching (to overlap communication and computation), SIMDization (to use the fastest floating point units), cache bypass (to avoid unnecessary cache traffic), and common subexpression elimination. Finally, "two-pass greedy search" refers to searching for the best parameters of each optimization one at a time instead of searching all possible combinations, to reduce search time. Each of these optimizations is critical for performance on at least one machine. Trying them all is a job to be automated by an autotuner.

Different derivations of finite difference methods can result in substantially more points in the stencil. If properly optimized, the memory traffic will be the same as a seven-point stencil. As a result, the code may be substantially more computationally intense. To that end, many of the register blocking optimizations used in dense linear algebra may be applied to improve performance.



SOURCE: S. WILLIAMS, LBNL ILLUSTRATION: A. TOVEY

Figure 14. *Conceptualization of (a) seven- and (b) 27-point stencil operators.*

The stencil operators can be substantially more complex than simple linear operators. For example, in lattice Boltzmann methods — in addition to macroscopic quantities like density, momentum, and magnetic field — a distribution of velocities is maintained at each point in space. The corresponding stencil gathers a particular velocity from each neighbor without reuse to evolve both the macroscopic quantities as well as the velocities one step in time. Although structurally similar to finite difference methods, the operator's computation is dramatically different.

Because the stencil in lattice methods touches so many different velocities, finite translation lookaside buffer (TLB) capacity becomes a performance impediment. To that end, we may block for the TLB instead of the cache.

Autotuning LBMHD

The LBMHD application was developed to study homogenous isotropic in turbulence MHD, the macroscopic behavior of an electrically conducting fluid interacting with a magnetic field. The study of MHD turbulence plays an important role in the physics of steller phenomena, accretion discs, interstellar medium, and magnetic fusion devices.

At each point, one calculates macroscopic quantities like density, momentum, and magnetic field, and being a lattice-Boltzmann method, a distribution of 27 velocities must be maintained. However, as LBMHD couples computational fluid dynamics (CFD) with Maxwell's equations, a 15-velocity magnetic field distribution is also maintained. The result is that each point in space stores three macroscopic quantities and two velocity distributions — a total of 79 doubles (figure 17, p57).

LBMHD iterates through time performing a series of *collision*( ) operators — a nonlinear stencil operator. Although this results in a conceptually simple memory access pattern, the sheer scale of it can severely impair performance on modern cache-based microprocessors.

In addition to blocking for the TLB, data structure changes are required to achieve optimal performance. As the optimal parameterizations of the optimizations varied from one architecture to the next, we choose to construct an application-specific autotuner for LBMHD's *collision*( ) operator that explores changes to loop structure (blocking for the TLB), data structure (array padding), and approaches to exploit multicore parallelism (skewed loop parallelization).

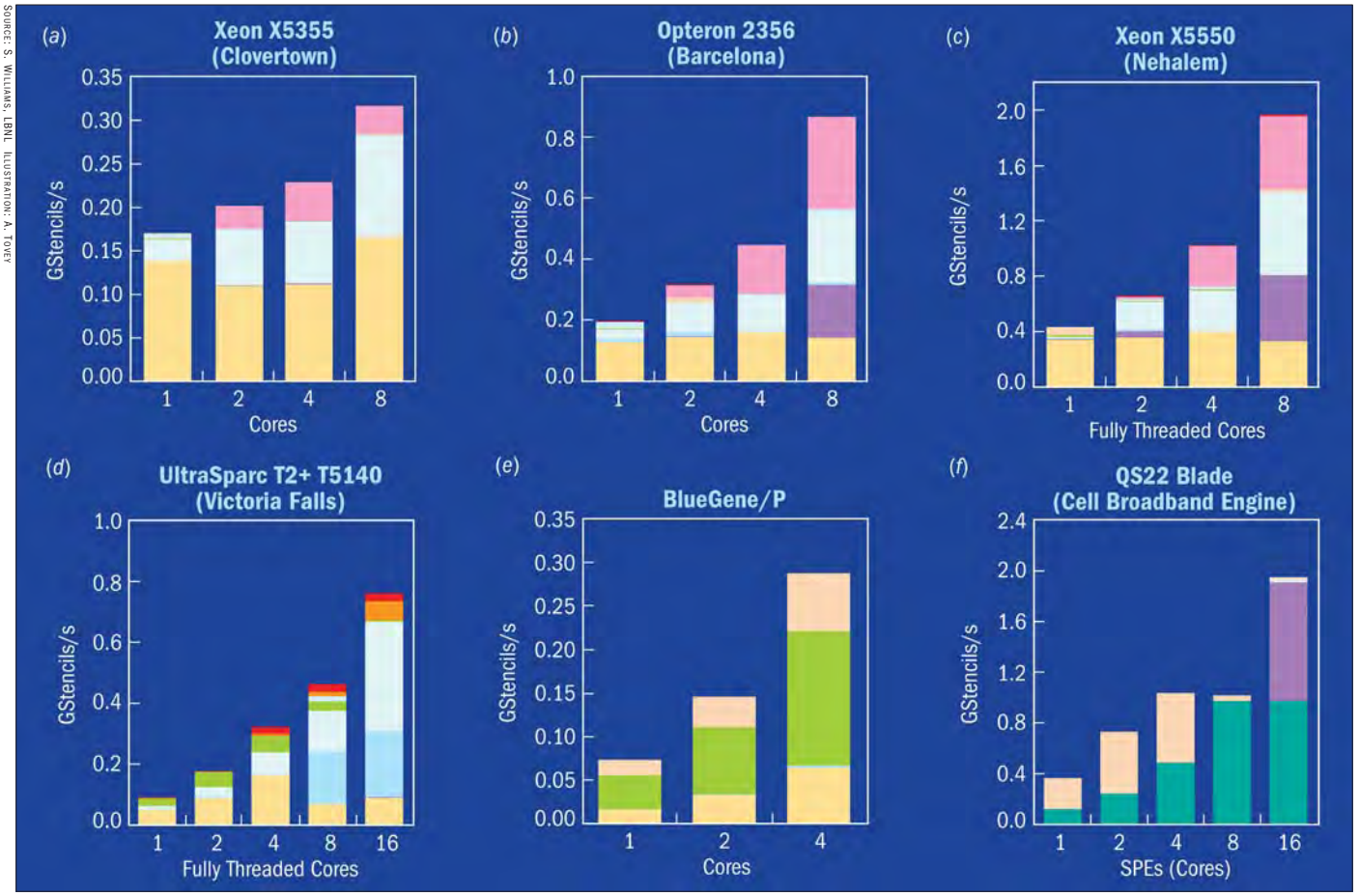
When running on an SMP, the common code base could improve performance at full concurrency by four times on a dual-socket, quad-core Opteron; 16 times on a dual-socket, 128-thread Niagara2; and 132 times on a dual-socket QS20 Cell Blade.

A distributed memory autotuner was created that allowed the exploration of alternate Message Passing Interface (MPI) decompositions as well as different balances between MPI processes and threads per MPI process (hybrid programming model). This autotuner allowed the fusion of the traditional SMP autotuning technology with the desired distributed memory application. When integrated into the application, autotuning improved performance by 2.5 times on a 512-core simulation (single-socket, quad-core Opteron nodes) compared with the existing, Gordon Bell-nominated, flat MPI implementation; and performance improved by three times when including autotuning of MPI and thread decomposition.

The Future of Structured Grid Autotuning

Typically, autotuners for structured grids are constructed with Perl scripts. Unfortunately, this is not a particularly productive solution because there is a myriad of structured grid kernels and limited retargetability of autotuners. To that end, current research is examining the prospect of integrating compilation techniques into autotuners. As we have observed

Current research is examining the prospect of integrating compilation techniques into autotuners.



SOURCE: S. WILLIAMS, LBNL ILLUSTRATION: A. TOVEY

Figure 15. *Auto-tuning the seven-point stencil for multicore (performance versus concurrency).*

with autotuners for stencils and lattice methods, this technique must evolve beyond simple autotuner compilers and integrate the ability to explore alternative data structures. Because of the sheer number of structured grid-based applications, if autotuning of structured grids is made productive, the technology will have a much larger impact on the community than autotuning linear algebra or FFTs.

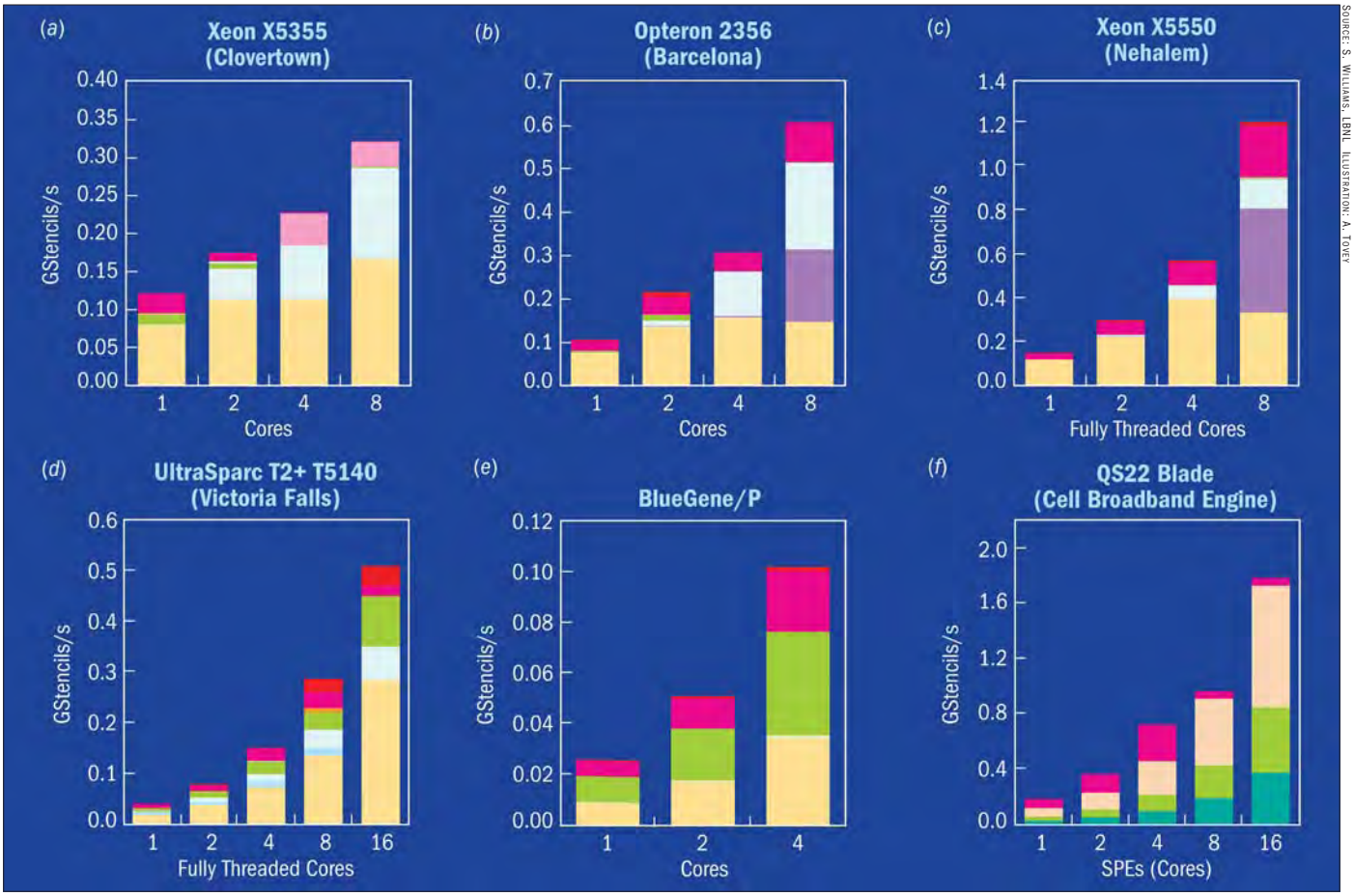
Machine Learning to Automate Autotuning

A major challenge of autotuning is the size of the parameter space to explore: state-of-the-art autotuners that consider only a single application, a single compiler, a specific set of compiler flags, and homogeneous cores may explore a search space of over 40 million configurations. An exhaustive search would take about 180 days to complete on a single machine. If the autotuner considers alternative compilers, multichip NUMA (non-uniform memory architecture) systems, or heterogeneous hardware, the search becomes prohibitively expensive. Even parallel exploration of multiple configurations (such as in a supercomputing environment) achieves only linear speedup in the search, so most autotuners prune the space by using heuristics of varying effectiveness.

To address this challenge, researchers have begun turning to statistical machine learning (SML) algorithms that can draw inferences from automatically constructed models of large quantities of data. SML-based autotuning does not require knowledge of the application or the microarchitecture. In addition, some SML algorithms even allow simultaneously tuning for multiple metrics of success.

We are able to reduce the half-year long search to two hours while achieving performance at least within 1% of and up to 18% better than that achieved by a human expert.

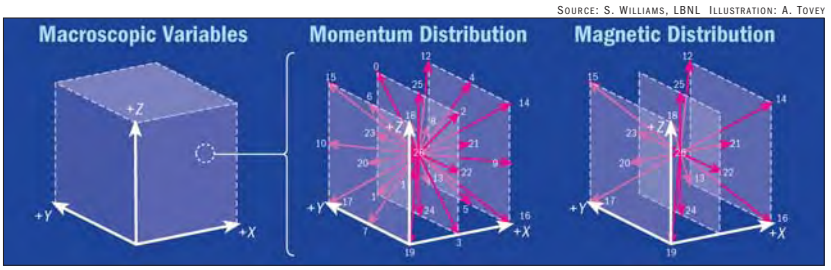
As an example, Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson of UC–Berkeley applied a state-of-the-art SML technique, Kernel Canonical Correlation Analysis (KCCA), to guide an autotuner's search through the parameter space of optimizing two stencil codes on two different multicore architectures. Compared to a human expert hand-optimizing the codes given extensive knowledge of the microarchitecture, the autotuned codes matched or outperformed the human expert by up to 18%. The autotuner took about two hours to explore a space that would take weeks to explore exhaustively on a single computer



SOURCE: S. WILLIAMS, LBNL ILLUSTRATION: A. TOVEY

Figure 16. *Autotuning the 27-point stencil for multicore (performance versus concurrency).*

by conventional techniques because the search guided by KCCA navigates a narrow path through the high-dimensional space of optimization parameters, avoiding combinatorial explosion. SML-guided search therefore opens new autotuning research directions that were previously intractable, including optimizing jointly for power and performance, optimizing the composition of kernels rather than considering only each kernel in isolation, and tuning for multichip architectures by optimizing both computation on individual (possibly heterogeneous) nodes as well as communication efficiency across the network. ●



SOURCE: S. WILLIAMS, LBNL ILLUSTRATION: A. TOVEY

Figure 17. *Conceptualization of the data structures used in LBMHD.*

Contributors James Demmel, UC–Berkeley and LBNL; Jack Dongarra, University of Tennessee; Armando Fox, UC–Berkeley; Sam Williams, LBNL; Vasily Volkov, UC–Berkeley; Katherine Yelick, UC–Berkeley and LBNL

Further Reading

FFTW
www.fftw.org
Spiral
www.spiral.net
UC–Berkeley CS267: Applications of Parallel Computers
www.cs.berkeley.edu/~demmel/cs267_Spr09
OSKI
http://bebop.cs.berkeley.edu/
R. Vuduc, J. Demmel, and K. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *J. Phys.: Conf. Ser.* **16**: 521-530.
http://www.iop.org/EJ/abstract/1742-6596/16/1/071
S. Kamil et al. 2009. A Generalized Framework for Auto-tuning Stencil Computations. Cray User Group Conference (Winner, Best Paper). Atlanta, GA.
http://cug.org/7-archives/previous_conferences/CUG2009/index.php
R. Nishtala and K. Yelick. 2009. Optimizing Collective Communication on Multicores. HotPar 2009. Berkeley, CA.
http://www.usenix.org/event/hotpar09/tech/full_papers/nishtala/nishtala.pdf