

Compiler-Directed Automatic Performance Tuning

Mary Hall (PI), University of Utah

Jacqueline Chame (co-PI), USC/ISI

Jaewook Shin (co-PI), Paul Hovland (co-PI), Argonne National Laboratory

Project Summary

This project has developed compiler-directed performance tuning technology targeting the Cray XT4 Jaguar system at Oak Ridge, which has multi-core Opteron nodes with SSE-3 SIMD extensions, and the Cray XE6 Hopper system at NERSC. To achieve this goal, we combined compiler technology for model-guided empirical optimization for memory hierarchies with SIMD code generation, which have been developed by the PIs over the past several years. We examined DOE Office of Science applications to identify performance bottlenecks and apply our system to computational kernels that operate on dense arrays. Our goal for this performance-tuning technology has been to yield hand-tuned levels of performance on DOE Office of Science computational kernels, while allowing application programmers to specify their computations at a high level without requiring manual optimization. Overall, we aim to make our technology for SIMD code generation and memory hierarchy optimization a crucial component of high-productivity Petaflops computing through a close collaboration with the scientists in national laboratories.

1. Code Transformation Framework

This project has funded significant development of CHiLL, a framework for composing high-level loop transformations designed to generate efficient code for complex loop nests [1]. It supports an extensive collection of loop transformations for perfect and imperfect loop nests, including tiling, permutation and unroll-and-jam, thus lifting the burden of generating multiple intermediate steps from compilers or optimization tools. CHiLL uses the Omega test to manipulate integer arithmetic and relies on polyhedral scanning provided by Omega's code generator. Recently we have integrated new features into CHiLL, including support for user input to the tool. Users can now relay information about the code and input data that could not be derived by static analysis alone, and this information often results in more efficient code generation.

We have publicly released CHiLL. We have also updated the Omega Library and the associated code generator, released as Omega+ and Codegen+, which is a main component of CHiLL. All have been released and can be found on our research page, http://ctop.cs.utah.edu/ctop/?page_id=21

We have worked extensively to increase the capability of CHiLL. A major focus of this work is on providing the appropriate interface to application and library developers to

increase their productivity. In working with application and library developers, we have discovered that many optimizations that are well-known and available in most compilers are performed manually by the application developers. The reason behind this is the application developers do not have sufficient control over how the optimizations are performed or the parameters that are used by the compiler, and they cannot see the results of the optimization. Our goal in building the interface to CHiLL was to expose the transformation capability to the programmer, and then rely on the compiler and auto-tuning framework to automate code generation and parameter selection. Over the past year, we have broadened the capabilities of the interface in CHiLL. We now support OpenMP and CUDA code generation through high-level directives provided by the programmer, and work is underway for OpenCL code generation. We have also raised the level of abstraction for this interface so that programmers have a higher level of abstraction when interacting with the system [1][2].

There are also two major internal improvements in CHiLL. Currently, CHiLL uses the Stanford SUIF compiler to provide a frontend, intermediate representation (IR), and backend code generation from the IR. However, the majority of CHiLL is independent of its IR. We have built it in this way because we planned to replace SUIF, an infrastructure that is over 15 years old, with more modern compiler platforms. To this end, we have improved CHiLL so that it no longer uses SUIF's control structures directly when initializing a loop, but rather uses an infrastructure-independent abstraction in C++ classes. Combined with previous data type abstraction, CHiLL now has an independent abstract layer over the underlying compiler IR, paving the way to other compiler infrastructure. Migrations to ROSE and Clang infrastructures are underway (ROSE completed after the grant ended and now released).

The other internal improvement to CHiLL is dependence graph updating for all loop transformations. After each loop transformation, CHiLL updates its internal dependence graph to maintain a consistent view of the loop structure and data accesses in the (partially) transformed loop. This is critical in a polyhedral framework to allow unrestrictive composition of multiple loop transformations, some of which might affect loops in a complicated way. New relations of dependences are not calculated from recomputing dependences among array references under the new iteration spaces. They are deduced from the semantics of the transformation itself. Incremental modification to the dependence graph is more efficient than simply rerunning dependence analysis on the modified loop nest. By maintaining an accurate dependence graph during the composition of transformations, CHiLL provides a robust loop transformation framework that can handle complicated and unpredictable usage situations.

2. Nek5k tuning results

Nek5000 is a scalable code for simulating fluid flow, heat transfer, and magnetohydrodynamics as well as electromagnetics (in a separate code, NekCEM). The

code is based on the spectral element method (SEM), a hybrid of spectral and finite-element methods.

The core computation in Nek5000 calls for repeated function evaluations either for explicit substeps of the time advance or for iterations in implicit substeps. Within each element, each evaluation entails matrix-vector products of the form $C \times B \times A \bar{u}$. Specifically, we require sums of the following form:

$$v_{ijk} = \sum_{p=1, N} A_{ip} u_{pjk}, \quad v_{ijk} = \sum_{p=1, N} B_{jp} u_{ipk}, \quad v_{ijk} = \sum_{p=1, N} C_{kp} u_{ijp}, \quad i, j, k \in \{1, \dots, N\}^3$$

The first product can be cast as a matrix multiply if u_{ijk} is viewed as an array having N^2 columns of length N . Similarly, the last product can be expressed as $V = UC^T$. The middle sum is expressed as a sequence of small products, $u(:, :, k)B^T$, $k = 1, \dots, N$. Because the approximation order of the pressure and velocity spaces differ by 2, the above sums also appear with permutations in which index ranges may be replaced by $M = N - 2$. Thus, Nek5000 requires numerous calls to small, dense matrix multiplies of known sizes over a limited range of values.

We investigated the performance impact of autotuning and specialization for two Nek5000 data sets: Helix2, which is helical pipe flow, similar to that found in certain vascular flows, and G6a, which is turbulent flow in a channel that is partially blocked by a cylinder. We used PAPI to collect hardware performance metrics and observed that, with the Helix2 input, the application spends approximately 60% of the time on a particular function, `mxm44_0`. This function is a manually tuned implementation of matrix multiply, which yields overall good performance over a wide range of architectures. The main loop nest is unrolled by 4 for each of the i and j loops of the original loop nest. If either M or N is not a multiple of 4, clean-up loops execute the residual iterations. To investigate the frequency of each array size, we instrumented `mxm44_0` so that it captures the number of calls for each matrix size across all of its invocations for each of the two problems. We use these call frequencies to select sizes for specialization and optimize the conditional checks for matrix size.

The methodology for optimizing Nek5000 consists of three steps. We first use CHiLL to generate code versions specialized for specific matrix sizes. An automated empirical search then finds the best optimization parameters, using a set of compiler heuristics to keep the search space manageable. Finally, we create a library of specialized code versions and replace the original computation with calls to the library. At run time, the matrix size determines which of the tuned versions will be executed.

Specialization information allows the autotuning tools to derive highly optimized specialized versions of a computation for known input sizes, which is particularly valuable for Nek5k. We used CHiLL to automatically generate the specialized versions for the library. Because these small matrices fit within even small L1 caches, the focus of optimization should be on managing registers, exploiting ILP in its various forms, and

reducing loop overhead. For these purposes, we use *loop permutation* and aggressive *loop unrolling* for all loops in a nest. To the backend compiler, unrolling exposes opportunities for instruction scheduling, scalar replacement, and eliminating redundant computations. Loop permutation may enable the backend compiler to generate more efficient *single-instruction multiple-data* (SIMD) instructions by bringing a loop with unit stride access in memory to the innermost position, as required for utilization of multimedia-extension instruction set architectures. Thus we generate specialized versions using a combination of loop permutation and unroll-and-jam. In some cases, where the matrices are small, we obtain the best performance by coming close to fully unrolling all the three loops in the nest. When applied too aggressively, however, loop unrolling can generate code that exceeds the instruction cache or register file capacity. Therefore, we use autotuning to identify the unroll factors that navigate the tradeoff between increased ILP and exceeding capacity of the instruction cache and registers. We rely on the native backend compiler for the architecture to identify the SIMD instructions, and simply expose code to the backend that will be optimized most effectively. Even better performance can be obtained by aggregating multiple calls to matrix-matrix multiply and optimizing the code to exploit reuse in registers and cache. Being compiler-based, our approach can optimize the middle loop that contains multiple calls to matrix-matrix multiply. To do this, we inline the matrix-multiply function into the loop, and use the inlined loop nest as the input to the autotuning framework.

Performance improvements for the full Nek5k application running on the Cray XT5 *jaguar* system at Oak Ridge are 38% on 4 nodes for input helix2, and up to 26% on 256 nodes for input g6a, as illustrated in Figure 1 and Figure 2.

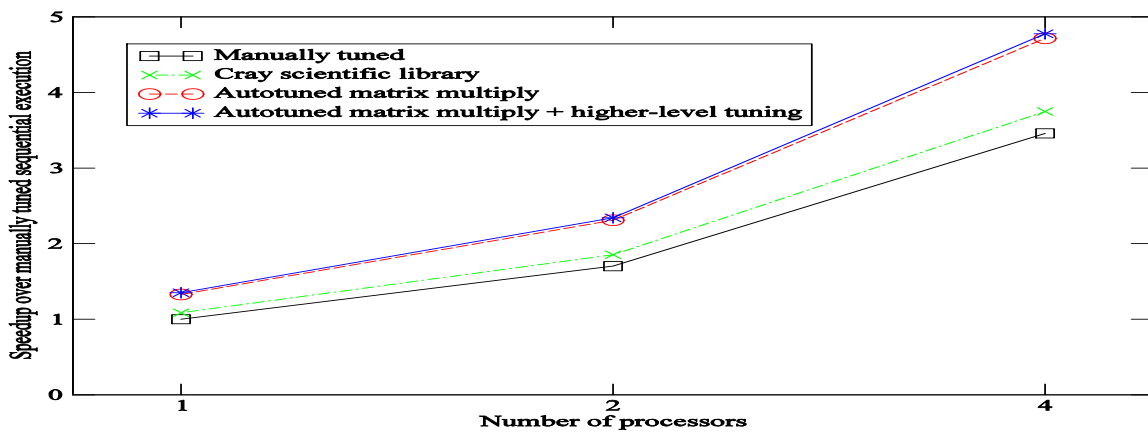


Figure 1. Nek5000 with input helix2 on jaguar using 1 core per node.

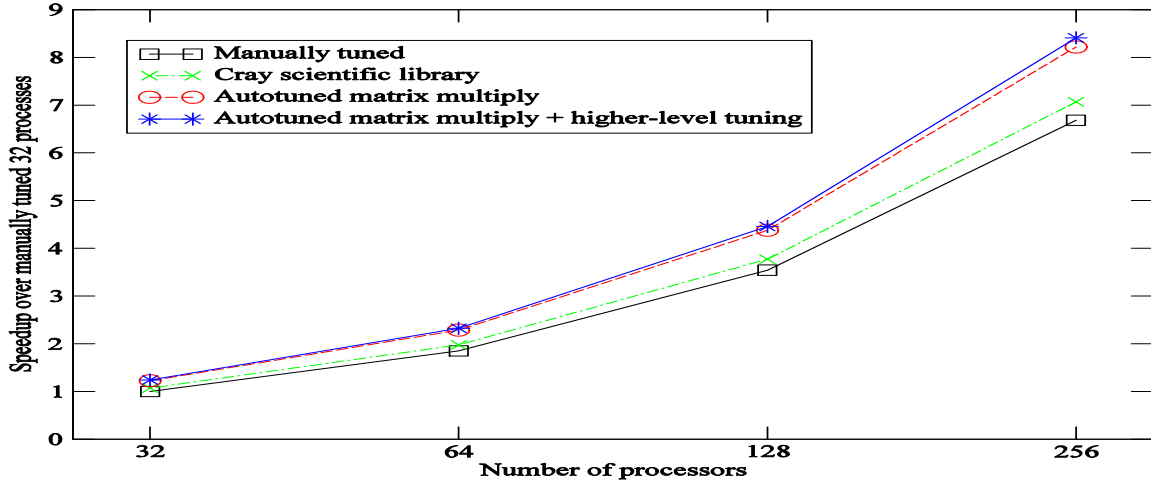


Figure 2. Nek5000 with input g6a on jaguar using 1 core per node.

3. Auto-tuning for instruction selection and scheduling

With a faculty and student team from Chicago State University, we investigated techniques to improve the performance of small matrix-matrix multiply through instruction selection and empirical instruction scheduling. As part of this research, we identified an opportunity to replace two instructions (MOVSD and UNPCKLPD) generated by the compiler with a single instruction (MOVDDUP). In addition, we developed a binary instruction rescheduler and a binary instruction generator that employs very aggressive unrolling. Together, these techniques raise the performance of (10,10,10) matrix-matrix multiply from 57% of peak to nearly 70% of peak on an AMD K10 processor. Our preliminary analysis suggests that 70% is close to the maximum achievable performance on this processor family.

4. Analysis of PETSc

An important lesson from optimizing nek5000 is the performance gain that can be achieved by specializing library code for its execution context. Libraries are written in a very general way to anticipate a wide variety of ways in which they may be used. This generality may lead to extensive control flow tests or other overheads, and reduces optimization opportunities, especially when the libraries are used in ways that differ from the common case (such as the small matrices used in nek5000). Through instrumentation, we may be able to identify common use cases within a specific application, and improve optimization effectiveness when such information is available. For example, selecting unroll factors and tile sizes for loop nests benefits from information about the iteration count and memory accesses within the nest.

In the related SciDAC project PERI, we applied CHiLL to optimize PFLOTRAN, a DOE application developed at LANL that models multiscale-multiphase-multicomponent subsurface reactive flows. PFLOTRAN uses the PETSc library as the basis of its parallel framework. The PERI team identified three main computations in PFLOTRAN as candidates for optimization: a PETSc routine that computes a matrix-vector multiplication (MatMul_SeqBAIJ_N); a PETSc function that solves the system $Ax = b$, given a factored matrix A, (MatSolve_SeqBAIJ_N); and a routine that calculates the contribution of aqueous equilibrium complexity to the residual and Jacobian functions for Newton-Raphson (RTOTAL).

Instead of writing manually optimized versions of PETSc library calls and testing for different unroll/tiling factors, the kernel along with its known parameters were provided as inputs to CHiLL. CHiLL was used to generate different code variants according to the parameter values and transformation factors. A heuristic based search was then performed by Active Harmony to find the best performing variant for each permutation of values a set of parameters/variables can possess. The experiment was performed swiftly and the overall applications performance was improved by 5%.

Writing specialized code is a technique often used by library developers to optimize applications. However, manually-written code has several disadvantages when compared to our framework.

- The library developer will not possess the values of parameters along with their frequencies at design time. Hence, he would be able to write specialized functions only for specific values.
- It gets a little difficult for the developer to reason about the best implementation when the number of variables/parameter along with each of their possible values is more than a few.
- It is not feasible to expect the application programmer to write specialized code.
- Performance of implementations vary according to the architecture.

PETSc alone has 29 functions which have been specialized, amounting to a total of 242 manually-written functions. For the last few months we have been investigating how to combine CHiLL with PETSC such that applications can use the CHiLL framework to generate these specialized versions to reduce the amount of code that is provided by the library and, using auto-tuning and specialization, derive more highly optimized versions of the library functions. This code generation could be deferred until the build of the application so that the code can be specialized for the application and execution context. Such an integration would allow users to study the hotspots in the PETSC

library and extract frequent values if possible. The best performing code would then be automatically generated and included in the PETSC library.

Figure 3 shows results from a study using CHILL to specialize PETSc code for three large-scale applications: PFLOTRAN (as previously described), the Uintah Problem Solving Framework and UNIC, a 3D unstructured deterministic neutron transport code. This work demonstrated significant performance improvements of more than 1.8X on the library functions and overall gains of 9 to 24% on the overall applications. A full report of this experiment and methodology can be found elsewhere [8][9].

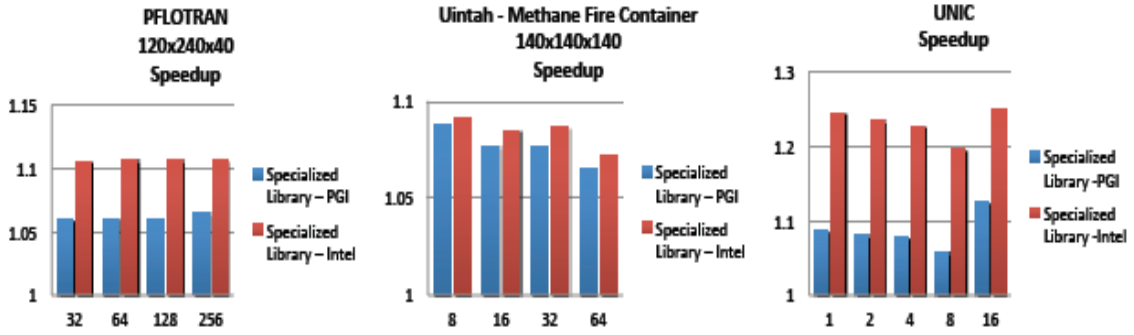


Figure 3. Impact of PETSc specialization on application performance.

5. Other benchmark: MADNESS

We have also applied the combined autotuning and specialization technology to the core computation of MADNESS. As in Nek5000, computational kernels performing matrix multiplications of small matrices are responsible for large fractions of execution time. On MADNESS, typical matrix sizes are in the range of 2 to 30. We used the same autotuning and specialization methodology to derive specialized code versions for MTXMQ, a matrix-transpose matrix multiplication routine. Our results show our automatically-generated library yields better performance than the hand-coded assembly library for MADNESS on small problem sizes, but not yet for larger problem sizes. Our auto-tuned library is also sometimes better than the ACML library due to specialization. Table 1 shows the number of FLOPS per cycle of MTXMQ, ACML and our TUNE versions of the original MTXMQ routine.

ni	nj	nk	<i>MTXMQ</i>	<i>ACML</i>	<i>TUNE</i>	<i>TUNE %peak</i>
4	2	2	0.10	0.07	1.50	37.66
16	4	4	1.04	0.51	1.63	40.84
36	6	6	1.74	0.99	1.89	47.47
64	8	8	2.33	1.56	1.96	49.10
100	10	10	2.61	1.80	2.26	56.00
144	12	12	2.69	2.12	2.42	60.72

Table 1. Performance of MTXMQ (from MADNESS's implementation) versus ACML and TUNE. ni , nj and nk are matrix dimensions.

We are currently investigating the performance of MADNESS on multiple nodes and

multiple cores per node on *jaguar*, the Cray XT5 at ORNL.

Publications

- [1] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, M. Khan, Loop Transformation Recipes for Code Generation and Auto-Tuning, *The 22nd International Workshop on Languages and Compilers for Parallel Computing*, October 8-10, 2009.
- [2] G. Rudy, M. Hall, C. Chen, J. Chame, M. Khan, "A Programming Language Interface to Describe Transformations and Code Generation," *The 23rd International Workshop on Languages and Compilers for Parallel Computing*, October, 2010.
- [3] J. Shin, M. Hall, J. Chame, C. Chen, P. Fisher and P. Hovland. "Speeding up Nek5000 with Autotuning and Specialization. *The 24th International Conference in Supercomputing (ICS 2010)*, June 2010.
- [4] J. Shin, M. W. Hall, J. Chame, C. Chen, P. D. Hovland, "Autotuning and Specialization: Speeding up Matrix Multiply for Small Matrices with Compiler Technology," International Workshop on Automatic Performance Tuning, October, 2009.
- [5] M. Khan, C. Chen, M. Hall, J. Chame, "CUDA-CHiLL: Using Compiler-Based Autotuning to Generate High-Performance GPU Libraries," Poster presentation, SC 2010, November, 2010.
- [6] J. Shin, M. W. Hall, J. Chame, C. Chen, P. D. Hovland, "Autotuning and Specialization: Speeding up Matrix Multiply for Small Matrices with Compiler Technology, *Software Automatic Tuning: from concepts to state-of-the-art results*, edited by Keita Teranishi, John Cavazos, Ken Naono and Reiji Suda, to appear 2010.
- [7] M.W. Hall and J. Chame, "Languages and Compilers for Autotuning," In *Scientific Computer Performance*, edited by David Bailey and Robert F. Lucas. Taylor and Francis publishers, to appear 2011.
- [8] S. Ramalingam, M. Hall, and C. Chen; "Improving High-Performance Sparse Libraries Using Compiler-Assisted Specialization: A PETSc Case Study," *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, May 2012.
- [9] S. Ramalingam, "Improving High-Performance Sparse Libraries Using Compiler-Assisted Specialization: A PETSc (Portable Extensible Toolkit for Scientific Computation) Case Study," May 2012.

Presentations

- "Tools for High Productivity HPC Software Development", Paul Hovland, Aachen University, June 2010.
- "Collaborative Autotuning of Scientific Applications," Mary Hall, SIAM Parallel Processing Symposium, Feb. 2010.
- "Paving the Way for Programming Extreme Scale Systems," DOE Institute for Computing in Science, Future of the Field Workshop, Jul. 2010.
- "Compiler-Based Auto-tuning for Application and Library Code," Mary Hall, Invited talk, DOE SciDAC Center for Scalable Application Development Software Workshop on Libraries and Autotuning for Petascale Applications, August, 2010.

“Next Generation Compiler”, Mary Hall, Panelist, DOE SciDAC Center for Scalable Application Development Software Workshop on Libraries and Autotuning for Petascale Applications, August, 2010.

“Compiler-Based Autotuning of Energy Applications,” USC-DOE Conference on Materials for Energy Applications: Experiment, Modeling and Simulations, March, 2011.

“Autotuning Compilers: Paving the Way to Exascale”, Invited Talk, Joint DOE ASCR and NNSA Exascale PI meeting, Annapolis, MD, October 2011.

“Automating Application Mapping with Autotuning: Paving the Way to Exascale,” Salishan Conference on High-Speed Computing, April 2012.

“Autotuning Compiler and Language Technology and its Role in Exascale Systems,” Invited speaker, 6th International Conference on Automatic Differentiation, July 2012.