

Speeding up Nek5000 with Autotuning and Specialization

Jaewook Shin[†], Mary W. Hall[‡], Jacqueline Chame[§],
Chun Chen[‡], Paul F. Fischer[†], Paul D. Hovland[†]

[†]{jaewook, fischer, hovland}@mcs.anl.gov [‡]{mhall, chunchen}@cs.utah.edu
Mathematics and Computer Science Division School of Computing
Argonne National Laboratory University of Utah
Argonne, IL 60439 USA Salt Lake City, UT 84112

[§]jchame@isi.edu
Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292

Abstract

Autotuning technology has emerged recently as a systematic process for evaluating alternative implementations of a computation to select the best-performing solution for a particular architecture. Specialization optimizes code customized to a particular class of input data set. In this paper, we demonstrate how compiler-based autotuning that incorporates specialization for expected data set sizes of key computations can be used to speedup up nek5000, a spectral element code. Nek5000 makes heavy use of what are effectively Basic Linear Algebra Subroutine (BLAS) calls, but for very small matrices. Through autotuning and specialization, we can achieve significant performance gains over hand-tuned libraries (e.g., Goto, ATLAS and ACML BLAS). Further, performance improves even more significantly using higher-level compiler optimizations that aggregate multiple BLAS calls. We demonstrate more than 2.3X performance gains on an Opteron over the original manually-tuned implementation, and speedups of up to 1.21X on the entire application running on 256 nodes of the Cray XT5 jaguar system at Oak Ridge.

1 Introduction

The complexity and diversity of today's parallel architectures overly burdens application programmers in porting and tuning their code. At the very high end, processor utilization is notoriously low, and the high cost of wasting these precious resources motivates application programmers to devote significant time and energy to tuning their codes.

To assist the application programmer in managing this complexity, much research in the last few years has been devoted to auto-tuning software that employs empirical techniques to evaluate a set of alternative mappings of computation kernels to an architecture and select the mapping that obtains the best performance [2, 27, 9, 21, 23]. This paper focuses on a particular role for

autotuning, used in conjunction with *specialization* for specific classes of known input sizes. The autotuner can derive highly optimized specialized versions of a computation for known input sizes, and generate a library of these specialized versions. At run time, the execution environment can invoke the appropriate specialized version from the library.

In previous work, we have presented results from using compiler-based autotuning for computational kernels, demonstrating performance sometimes comparable to manually-tuned codes [4, 6, 23]. Extending this approach to tuning scalable applications, we apply this approach to `nek5000`, a scalable spectral element code. The execution time of `nek5000` is dominated by what are essentially matrix-matrix multiplies of very small, rectangular matrices. These small matrices are of known sizes, and the size remains the same for different problem sizes and different scales. Further, while highly-optimized BLAS libraries (native, ATLAS, GOTO, etc.) are available, they are tuned for large rectangular matrices, and do not achieve as high performance as our generated library for small matrices, as reported in prior work [22]. Thus, specialization for small matrices is a particularly valuable optimization for `nek5000`, but autotuning is needed to identify the best-performing implementation of each problem size and add it to the library. Even further performance gains are possible when higher-level compiler optimization aggregates multiple BLAS calls; autotuning again identifies the best optimization strategy.

In this paper, we describe the process used to optimize `nek5000` for the Opteron processor that combines autotuning and specialization compiler technology. We use CHiLL, a polyhedral loop transformation framework with a script interface, to describe the space of specialized implementations and automatically generate the code for the optimized library [6, 23, 12]. Using heuristics to prune the search space of possible implementations, a set of variants generated by CHiLL are then measured and compared to derive the specialized library of implementations. The resulting BLAS calls are more than 2.3X faster than the original implementation, and the overall application performance is improved by 21% on 256 nodes of the Cray XT5 system at Oak Ridge. This paper makes several contributions:

- Automation of architecture-specific specialization for `nek5000` using compiler tools, demonstrating up to 21% performance improvements over on a production code that was already manually tuned for a problem size running at scale at 256 nodes (and a 19% improvement at 128 nodes). This process could be repeated to tune `nek5000` for similar architectures and native compilers.
- Demonstration that compiler-based autotuning and specialization can exceed performance gains from library-based autotuning through the use of higher-level optimization that exploit application context; that is, how the libraries are invoked and expected data sets.
- An optimization strategy that can generalize for other applications that use matrix-matrix multiply on irregular-sized matrices for which BLAS libraries do not yield high performance, or for related small matrix operations targeting high-performance for architectures supporting multimedia extensions such as SSE-3.

The remainder of the paper is organized as follows. The next section presents an overview of `nek5000` and its performance characteristics. Section 3 describes the compiler technology used in the optimization of `nek5000`. We discuss the experimental results for `nek5000` in Section 4, followed by a discussion of related work and a conclusion.

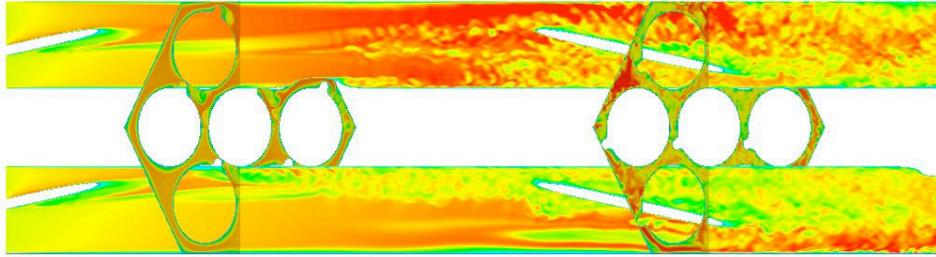


Figure 1: Turbulence in wire-wrapped subassemblies visualized by axial velocity distributions.

2 Nek5000

Nek5000 is a scalable spectral element code for the simulation of fluid flow, heat transfer, magnetohydrodynamics and electromagnetics (the latter in a separate code, NekCEM). The code is based on the spectral element method (SEM) [18], which is a hybrid of spectral and finite element methods. Spectral discretizations based on N th-order tensor product polynomial expansions in $\hat{\Omega} := [-1, 1]^d$, $d=2$ or 3 , provide rapid convergence (high accuracy per gridpoint) at low cost. In particular, operators that are nominally full with $O(N^6)$ nonzeros can be applied in only $O(N^4)$ work with only $O(N^3)$ memory accesses. Moreover, the work can be cast as dense *matrix-matrix* products. The SEM extends the geometric flexibility of spectral methods in two ways. First, the SEM employs Lagrangian interpolants based on Gauss or Gauss-Lobatto quadrature points that, in addition to ensuring stability, provide sufficiently accurate quadrature to allow pointwise evaluation of variable-coefficient integrands. Integrands involving Jacobians and metric tensors in deformed domains are thus evaluated with only N^3 storage and work complexity. In addition, the SEM allows multiple domains (deformable brick elements) to be coupled together in the traditional manner of finite elements to realize complex domain shapes. A recent example is the wire-wrapped rod-bundle flow of Figure 1, comprising 560,000 elements of order $N = 8$ [8].

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 $C \otimes B \otimes A\underline{u}$. Specifically, we require sums of the form:

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

It is clear that the first product can be cast as a matrix-matrix product if u_{ijk} is viewed as an array having N^2 columns of length N . Similarly, the last product can be expressed as $V = UC^T$, if v and u are viewed as $N^2 \times N$ matrices. The middle sum is expressed as a sequence of small products, $u(:, :, k)B^T$, $k = 1, \dots, N$ [7]. 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-matrix multiplies of known sizes over a limited range of values.

In this paper, we will use `nek5000` as an example to illustrate our autotuning process and associated tools. We evaluate performance impact of: (1) autotuning and specialization for indi-

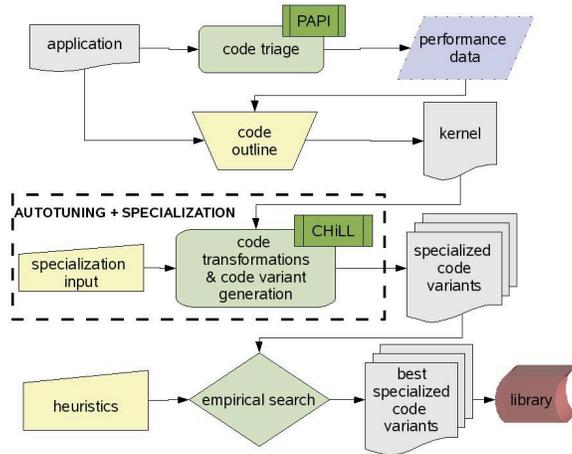


Figure 2: Overview of our approach

vidual calls to matrix multiply; and, (2) a higher-level optimization of a set of four calls to matrix multiply in a loop nest. We investigate the performance of two problems: `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.

3 Compiler Technology: Autotuning and Specialization

In this section, we describe our methodology in applying compiler technology to this autotuning and specialization process. First we describe the optimization strategy to be applied to this code. We use the CHiLL loop transformation framework to generate all desired transformed codes specialized for specific matrix sizes. Then we use empirical search to find the best optimization parameters. Here we apply a set of compiler heuristics to reduce the search space to something manageable. Finally, we create a library of specialized codes and replace invocations to the original computation with calls to the library.

To put this approach in context, we have automated much of what is difficult in manually tuning code – correct code generation, empirical measurements to explore a large set of implementations, pruning to avoid searching unprofitable implementations – but have relied on user involvement for some of the steps, particularly describing the transformation strategies to be considered. This approach can evolve towards increased automation as tool technologies improve, but the interfaces will also always leave the door open for application developers to have the control over optimization when they need it.

The tools described in this section are the middle portion of a further automated autotuning workflow described elsewhere and highlighted in Figure 2 [17, 12]. Code triage identifies computations that have optimization opportunities and performance issues. Code outlining derives a standalone kernel for the key computations discovered in the triage process, along with the kernel’s input data and parameter values collected during application runs. Once the bottlenecks of an application are identified and outlined into kernels, the extracted kernels need to be specifically tuned for the target architecture so that the best possible performance can be achieved (autotuning and code generation).

3.1 Result of Code Triage for Nek5000

From profiling `nek5000` for the `helix2` input, we found that a procedure called `mxm44_0` takes around 60% of total execution time. This function is a manually tuned implementation of matrix multiply, which the application authors have found yields overall good performance over a wide range of architectures. `mxm44_0` takes the same set of arguments as `mxm`. The loop order is `mnk` as shown in Figure 4(a). The main loop nest is unrolled by 4 for each of `m` and `n` loop. Thus, if both `m` and `n` are multiples of 4, only this 4×4 unrolled loopbody is executed. Otherwise, clean-up loops are executed to deal with the residue iterations. By checking the value of “matrix size *mod* 4”, the clean-up loops are also unrolled as much as possible up to 4 times for each of `m` and `n` to reduce the loop overhead further.

We would like to improve on the manual optimizations of `mxm44_0` by automatically generating a library for the matrix sizes used for the `helix2` problem. To investigate the frequency of each array size, we instrumented `mxm44_0` so that it shows the number of calls for each matrix size across all of its invocations. We estimate the computational importance of each matrix size by multiplying the number of calls with the product of the sizes of the three dimensions, and select those that comprise the bulk of the calls. As we will show in Section 4, this tuned code will also be useful for other problems that use the same order for the spectral element method, (*e.g.*, the `g6a` input data set), and for similar architectures and compilers.

3.2 Optimization Strategy

Since the key computational kernel of `nek5000` is matrix-matrix multiply, why is it not sufficient to just use standard BLAS libraries for the target architecture? This point was the focus of a previous paper, which compared various hand-tuned BLAS libraries to those generated by our compiler [22]. For large square matrices, such as, for example, 1024×1024 , the ACML (native), ATLAS, and the GOTO BLAS libraries all perform well, above 70% of peak performance. This is because they incorporate aggressive memory hierarchy optimizations including code transformations such as *data copy*, *tiling* and *prefetching* to reduce memory traffic as well as to hide memory latency. Additional code transformations that can improve *instruction-level parallelism* (ILP) are performed to optimize the computation. Several examples in the literature describe this general approach [2, 27, 5, 10, 29].

However if we look closer at matrices of size 10 or smaller, those same BLAS libraries perform below 25% of peak performance. Since these 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 can use *loop permutation* and aggressive *loop unrolling* for all loops in a nest. If performed across outer loops of a loop nest, the latter optimization is often referred to as *unroll-and-jam*, indicating that outer loops are unrolled and resulting copies of inner loops are fused (jammed) together. 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 ISAs.

Thus, in our initial set of experiments, we generate code using the 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 of the three loops in the nest. However, when applied too aggressively, 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.

While effective, 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, as explained in Section 2. 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 as shown in Figure 3. Then, the inlined loopnest is used as an input to our autotuning framework.

	do iz=1,10
	do i=1,10
	do j=1,10
do iz=1,10	C(i,j,iz) = 0.0d0
call mxm(A(1,1,iz),10,B,10,C(1,1,iz),10)	do l=1,10
enddo	C(i,j,iz) = C(i,j,iz) + A(i,l,iz)*B(l,j)
	enddo
(a) original	enddo
	enddo
	enddo
	(b) inlined

Figure 3: Loop-inlining for higher-level tuning.

3.3 Specialization and Transformation Using CHiLL

One of the key challenges that a programmer faces during the tuning process is to try many different transformation strategies in order to find the best solution. This is an extremely slow and error prone process. CHiLL provides a script interface to the programmer that can be used to apply complex loop transformation strategies on a loop nest by composing a series of loop transformations [4, 6, 23]. The transformations supported include data copying, tiling, index set splitting, loop permutation, unroll-and-jam, fission, fusion and any unimodular transformation.

CHiLL's polyhedral framework provides a mathematical treatment of loop iteration spaces and array accesses. It has unique design features such as treating each statement in a unified iteration space with the same dimensionality and expressing lexicographical order among individual loops systematically in a polyhedral space. These features plus accompanied algorithms enable composing high-level loop transformations generating efficient code. Thus CHiLL provides the programmer an easy-to-use interface to the autotuning process that can express the kinds of manual optimizations that are commonly used by application programmers, generate high-quality code and facilitate systematic tuning of optimization parameters and selection of code variants.

To use CHiLL for optimizing small matrix multiply computations, consider the code in Figure 4(a). The matrix multiply code is imperfectly nested since the C array is initialized to zero before multiplication. To permute the loop in i,k,j order, the programmer only needs to specify

one permute transformation (Figure 4(c)) and CHiLL generates the correct result as shown in Figure 4(e). Further, the *unroll* command of CHiLL performs unroll-and-jam if the unrolled loop is an outer loop and inner loops can be fused together legally. Figure 4(g) shows the result of all three unroll sizes u_1, u_2 and u_3 set to 2. Such flexibility greatly helps programmers since they can now focus their effort on how the transformation affects the locality and the performance instead of the details of generating the correct code.

For the purposes of *specialization*, we have added to CHiLL the *known* command, which allows the programmer to express known loop bounds. Within CHiLL, *known* adds additional conditions to the iteration spaces extracted from the original code. These conditions subsequently affect the quality of the generated code, permitting different specialized versions and determining, for example, whether unroll factors evenly divide loop bounds, so that the compiler can avoid generating cleanup code.

Figure 4(b), (c) and (d) show three sample CHiLL scripts used to generate three different loop orders in (a) (actual output is similar but with loop upper bounds fixed at 10), (e) and (f), respectively. For brevity, we show the simplest versions where all unroll amounts are 1. Loops are numbered starting 1 for the outermost loop and increasing as we move inward. In *permute*, the loop numbers are used to indicate the loop order after the permutation. The meaning of `unroll(stmt,loop,factor)` is to unroll the individual statement *stmt* (numbered from 0) within *loop* by unroll factor *factor* (shown as unbound variables). The three scripts shown in Figure 4(b), (c) and (d) are different in the number of `unroll` commands; this is necessary because the number of loops are different after loop permutation as shown in (a), (e) and (f) depending on the loop order given. The above scripts plus additional ones for different loop orders combined with ten different *known* statements were used to generate the specialized BLAS library for nek5000.

These CHiLL scripts, as well as the ones for higher-level optimization, are automatically generated. Given a loop order and an unroll factor for each loop, first, a *permute* command is generated with the loop order, and then an `unroll` command is generated for each loop for statement *s1* in Figure 4(a). If all loops in which statement *s0* is not enclosed are inside of all loops in which it is enclosed as in the loop order of 123, no other `unroll` commands are necessary. Otherwise, an `unroll` command should be generated for each loop in which statement *s1* is enclosed but is itself enclosed within the loop in which statement *s0* is not enclosed. For example, *i* and *j*-loops are such loops when the loop order is 312, i.e., *kij* as in Figure 4(d). Since statement *s0* is not inside the *k*-loop but both *i* and *j*-loops are within *k*-loop after loop permutation, separate `unroll` commands are generated for statement *s0* for each of *i* and *j*-loop. This same approach can be applied to the higher-level loopnest in Figure 3(b). The only difference is that it is a 4-deep loop nest, and so four loops are considered for both the *permute* and `unroll` transformations.

3.4 Autotuning: Pruning the Search Space

With all the transformation scripts ready, the next step is to search for the best optimization parameters by invoking CHiLL to generate actual code variants and measure their performance on the target machine. For some of the smaller matrix sizes, the search space is sufficiently small that exhaustive search is feasible, but for the larger matrices, we must develop heuristics to prune the search space to complete the experiments. We extract the heuristics from the exhaustive search results of small matrices, and use them in pruning the space of larger matrices. In this section, we briefly describe the pruning heuristics we used, with a more detailed presentation in [22].

```

do 10, i=1,M
  do 20, j=1,N
s0:    c(i,j) = 0.0d0
      do 30, k=1,K
s1:    c(i,j) = c(i,j) + a(i,k)*b(k,j)
30    continue
20    continue
10    continue

```

(a) original.f

		permute([3,1,2])
	permute([2,3,1])	known(M=N=K=10)
permute([1,2,3])	known(M=N=K=10)	unroll(1,1,u1)
known(M=N=K=10)	unroll(1,1,u1)	unroll(1,2,u2)
unroll(1,1,u1)	unroll(1,2,u2)	unroll(1,3,u3)
unroll(1,2,u2)	unroll(1,3,u3)	unroll(0,2,u2)
unroll(1,3,u3)	unroll(0,3,u3)	unroll(0,3,u3)

(b) loop order i,j,k

(c) loop order j,k,i

(d) loop order k,i,j

		do 2, t4 = 1, 10, 1
		do 4, t6 = 1, 10, 1
		c(t4, t6) = 0.0d0
		4 continue
		2 continue
do 2, t2 = 1, 10, 1		do 6, t2 = 1, 10, 1
do 4, t6 = 1, 10, 1		do 8, t4 = 1, 10, 1
c(t6, t2) = 0.0d0		do 10, t6 = 1, 10, 1
4 continue		c(t4,t6)=c(t4,t6)+a(t4,t2)*b(t2,t6)
do 6, t4 = 1, 10, 1		10 continue
do 8, t6 = 1, 10, 1		8 continue
c(t6,t2)=c(t6,t2)+a(t6,t4)*b(t4,t2)		6 continue
8 continue		
6 continue		
2 continue		

(e) After loop permutation in (c) (f) After loop permutation in (d)

```

do 2, t2 = 1, 9, 2
  do 4, t6 = 1, 9, 2
    c(t6, t2) = 0.0d0
    c(t6, t2+1) = 0.0d0
    c(t6+1, t2) = 0.0d0
    c(t6+1, t2+1) = 0.0d0
4    continue
  do 6, t4 = 1, 9, 2
    do 8, t6 = 1, 9, 2
      c(t6, t2) = c(t6, t2) + a(t6, t4) * b(t4, t2)
      c(t6, t2+1) = c(t6, t2+1) + a(t6, t4) * b(t4, t2+1)
      c(t6, t2) = c(t6, t2) + a(t6, t4+1) * b(t4+1, t2)
      c(t6, t2+1) = c(t6, t2+1) + a(t6, t4+1) * b(t4+1, t2+1)
      c(t6+1, t2) = c(t6+1, t2) + a(t6+1, t4) * b(t4, t2)
      c(t6+1, t2+1) = c(t6+1, t2+1) + a(t6+1, t4) * b(t4, t2+1)
      c(t6+1, t2) = c(t6+1, t2) + a(t6+1, t4+1) * b(t4+1, t2)
      c(t6+1, t2+1) = c(t6+1, t2+1) + a(t6+1, t4+1) * b(t4+1, t2+1)
8    continue
6    continue
2    continue

```

(g) A complete example of script in (c) with u1=u2=u3=2

Figure 4: Example of CHiLL scripts and the generated codes.

Heuristic 1: Loop order. Certain loop orders that lead to lower-performance code variants are pruned from the search.

Heuristic 2: Instruction cache. The total unroll amount for all three loops is limited by a constant C that is likely to fill the L1 instruction cache.

Heuristic 3: Unit stride on one loop. Related to heuristic 1, the search is restricted to only those tuples (U_m, U_k, U_n) where at least one of U_m , U_k or U_n is 1, to achieve spatial locality within a SIMD register between instances of an array access from consecutive iterations.

Heuristic 4: Unroll factor divides iteration space evenly. When a loop of iteration count m is unrolled by a factor of u , the last ' $m \bmod u$ ' iterations have to be executed in a clean-up loop that is not unrolled. While the cost of executing in the clean-up loop is negligible when the iteration count is large, it is significant when the matrices are small.

For the four-loop loopnest of Figure 3(b), exhaustive search is not possible even for the smallest input size. We use the last three heuristics, but now for a four-dimensional loop nest.

While we presented specific heuristics tailored to a set of transformation scripts, at a higher level, the framework we describe can be used for other application-architecture pairs just by replacing heuristics and code transformations to reflect the features of the architecture, compiler and/or the application.

3.5 Building the Library

After each code variant is generated, it is compiled and linked with the driver that measures and records the performance. The careful measurement process is described in Section 4.

When the evaluation is complete for the generated code variants, the best performing variant is selected for each matrix size, and these are aggregated into a library. The library includes a wrapper code that takes the same number of arguments and has the same name as the existing default implementation to provide the same interface to the rest of `nek5000`, as in Figure ???. A specialized code is invoked if one is available that matches the three size parameters; otherwise, the original manually-tuned version is invoked.

4 Experiments

This section demonstrates the performance impact of optimizing `nek5000` using the approach and set of compiler tools described in the previous two sections. We begin with details of the two Opteron architectures we used for the set of experiments; the code was tuned for an Opteron Phenom processor, but additional experiments were conducted on a supercomputer of newer Opterons to demonstrate that performance gains continue when the tuned code is run on similar architectures and different data sets of similar order. Subsequently, we present specifics on how we generated the specialized code for `helix2`, an example input for `nek5000` described in Section 2. Finally, we present the performance results for first the optimized higher-level kernels and then the full `nek5000` codes invoking the specialized libraries.

For the small data set `helix2`, we measured performance on a 2.5 GHz AMD Opteron Phenom workstation that has four cores. The machine has separate 64KB L1 instruction and data caches, an integrated 512 KB L2 cache, 2 MB L3 cache and 4 GB of memory. Since it runs 64-bit Linux (Ubuntu_8.04-x86_64), all 16 XMM registers are available for use. To allow an access to

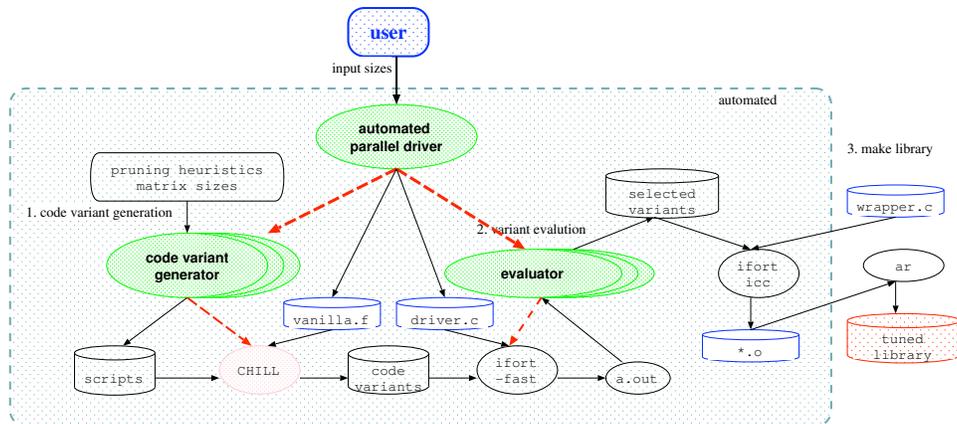


Figure 5: Block diagram describing autotuning tools and experiment.

hardware performance counters, the Linux kernel (version 2.6.25.4) is patched with `perfctr` distributed in PAPI version 3.6.0 [3]. CHiLL version 0.1.5 [6] and the Intel compiler version 10.1 [15] are used to transform and compile the code variants. For scaled up performance on 256 nodes, we measured performance on the `jaguarpf` at Oak Ridge National Laboratory, a Cray XT5 supercomputer with new six-core Opteron Istanbul processors. The architecture is similar to Phenom but there are 12 cores per node, each core runs at 2.6 GHz and has 2 GB of memory per core. The compiler we use is PGI `pgcc` and `pgf90` version 9.0-2, which is the default compiler on the machine. For these experiments, we used the `g6a` data set.

Figure 5 shows the block diagram of the experimental flow. Given input matrix sizes for which specialization is desired, a matrix multiplication kernel (`vanilla.f`), shown in Figure 4(a) and a driver (`driver.c`), are used to measure and collect the performance of code variants. The driver measures the number of clock cycles using `PAPL_TOT_CYC`, and to obtain accurate measurements, we execute a variant 500 and 10 times per measurement for matrix multiply and higher-level kernel respectively, collect 100 such measurements and record the minimum of the 100 measurements as the final performance of the variant. We produce as output a high-performance library of specialized matrix multiplication routines. Parameters m , k and n are used in defining matrix sizes as in $C(m, n) = A(m, k) \times B(k, n)$. Code variants are generated in Fortran but the driver is a C function. To generate aligned SIMD instructions, `_attribute__((aligned(16)))` qualifier is added to array declarations and the interprocedural optimization feature was used with `-fast` for `ifort` and `-O3 -ipo` for `icc`. Currently, all components inside the dotted box of Figure 5 are automated. Also, *automated parallel driver* can invoke multiple copies of *code variant generator* and *evaluator* to exploit task parallelism when multiple cores are available.

We optimize `nek5000` in the context of the `helix2` input data set. We chose the fastest variant for each matrix size to create a library of ten specialized matrix multiplication routines and the wrapper routine. A standard question that arises with this work is whether simply using highly-tuned BLAS libraries for the Opteron would be sufficient to tune `nek5000`. In fact, the use of specialization, autotuning and optimizations focused on small matrices yields significant gains over even the best manually-tuned libraries. A full comparison with other BLAS libraries was the focus of prior work [22], summarized here for completeness. We compared our generated BLAS codes to the manually-tuned code that is included with the `nek5000` distribution, the native ACML BLAS

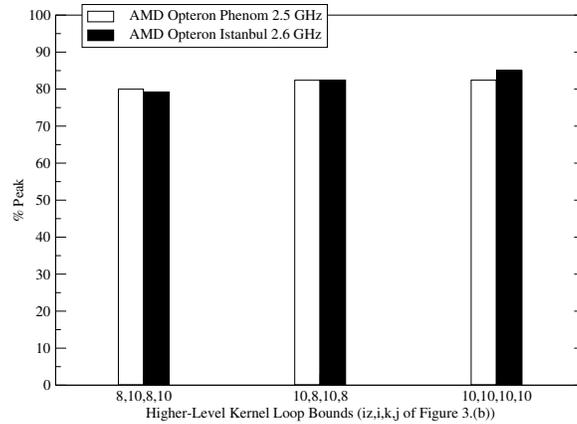


Figure 6: Performance of the tuned higher-level kernel of Figure 3(b).

(version 4.1.0), GOTO BLAS (goto_barcelone-r1.26) and ATLAS (version 3.8.2 with architectural default AMD64K10h64SSE3). The manually-tuned code that is part of `nek5000` performs at roughly 23% of peak across all sizes, which is more than 3X faster than using the native compiler. By comparison, ACML, GOTO and ATLAS perform slightly better, but still less than 30% of peak for the smallest matrices. Our automatically-generated codes yield performance that is up to 74% of peak, a 2.3X improvement over the manually-tuned code.

The remainder of this section presents the performance gains obtained by using the two versions of our generated library.

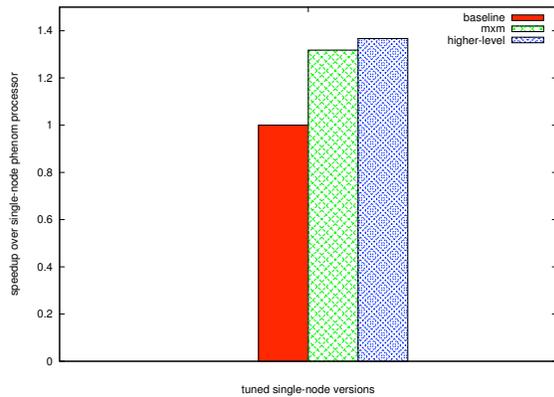
4.1 Performance of Higher-Level Kernels

Figure 6 shows the performance of the tuned versions of the higher-level kernels, shown in Figure 3(b), for three input sizes. Each label in X-axis of the graph represents the four loop bounds of iz , i , k and j -loops, and Y-axis is the percentage of machine's peak. The performance shown in the graph is extremely high as a compiler generated code, reaching as high as 82% of peak on a Phenom processor. Recall these kernels are the loops with a call to dense matrix multiply. When the matrix multiply kernels are tuned for these small sizes of (10,8,10), (8,10,8) and (10,10,10), the performance was shown to be between 57% and 58% of peak [22]. By tuning the outer loop together with the dense matrix multiply call, we could not only reduce the function call overheads but also achieve much higher performance.

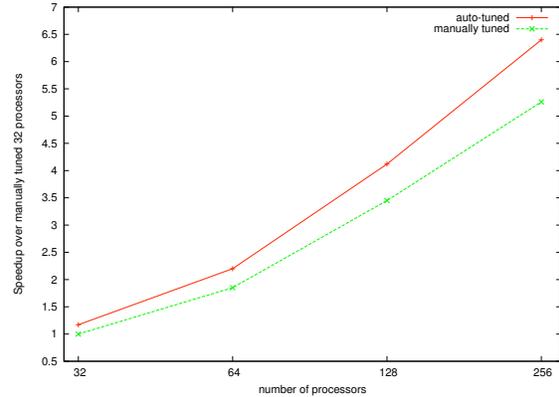
As a preparation to use the tuned library on a large parallel machine, the binary library was transferred to `jaguarpf` at the Oak Ridge National Laboratory to see if the binary library tuned on a different machine can be used on a large supercomputer based on a similar architecture. The black bars show the performance of the binary library on `jaguarpf`. As can be seen from the graph, the two different machines are comparable in the performance.

4.2 Performance of Nek5000 on Helix2 on Phenom Workstation

For this experiment, we used the `time` tool in Linux to measure the wall clock time. We ran the whole program 10 times for 50 time steps and took the median run time as the measurement for each of the baseline that uses `mxm44_0`, MXM that uses the tuned library of matrix multiplies and



(a) helix2 on a Phenom workstation



(b) g6a on jaguarpf

Figure 7: Speedups of the library generated by our approach

TUNE that uses the specialized library of both matrix multiplies and higher-level kernels. The bar chart in Figure 7(a) show the result of this experiment. When the tuned library of matrix multiplies is used, the run time drops significantly from the baseline of 207 seconds to 157 seconds achieving a 31% speedup. When the higher-level kernel is used in addition, the execution time drops even further to 151, achieving an additional 3% speedup. Overall, this is a speedup of 1.36X over the baseline.

4.3 Performance of Nek5000 on G6a on Jaguar Supercomputer

We next wanted to evaluate whether this significant performance gain for a single core would continue as we scaled up the computation to larger configurations and scaled-up problem sizes running on the jaguarpf supercomputer. We perform these experiments with a different input data set g6a, one designed to run on 32 to 256 cores, and compare the performance of the original nek5000 with the two versions that uses the specialized libraries, one with matrix multiplies and the other with higher-level kernels in addition to matrix multiply kernels. The g6a input was run for 1000 time steps, and we use the median execution time for each version of the code across five runs and for 32 to 256 nodes in a single-core mode, i.e., a process per 12-core node.

The speedup curve in Figure 7(b) show that performance gains are achieved with autotuning and specialization even at this larger scale, and for a different input data set and target platform than for which the code was tuned. The performance gains are significant, ranging from 17% on 32 nodes up to 21% on 256 nodes.

5 Related Work

Much prior work has been documented on automatically tuning systems. PHiPAC and ATLAS tune matrix multiplication code automatically for many target machines [2, 27]. Both ATLAS and PHiPAC use heuristics in guiding the tuning process to reduce the search time. FFTW is a self-tuning high performance library for discrete Fourier transform. The best factorization of the input size N is selected by dynamic programming based on empirical performance data [9]. SPIRAL is a high-performance code generation system for digital signal processing transforms [21]. Recently,

they show how to automatically generate high-performance code from a domain specific language for the domain of linear transforms [25]. OSKI combines install-time evaluations with run-time models to tune sparse-matrix vector multiplication and other solvers such as triangular solver [26].

Compiler assisted autotuning and tools facilitating code transformations have also been extensively studied. Chen et. al. combine compiler models and heuristics with guided empirical evaluations to take advantage of their complementary strengths [5]. Tiwari et. al. combine Active Harmony and CHiLL to generate and evaluate code variants. They use a search strategy similar to the Nelder-Mead algorithm [23]. Hartono et. al. used annotations in the code to describe performance improving transformations for C programs [13]. POET is a scripting language for parameterizing complex code transformations [28], which can be used in an autotuning process as well. Pouchet et. al. embedded legality of affine transformations as linear constraints in polyhedral space, thereby combining the code transformation steps and the legality checking step [19, 20]. Kulkani et. al. described a tool called VISTA which allowed the users to interactively select the order and scope of optimization phases in compiler [16].

Herrero and Navarro described specializing matrix multiplication for small matrices [14]. However, their code variants were generated manually. For tuning `nek5000`, the most closely related is Paul Fischer et. al. 's work on scaling `nek5000` on Bluegene [8] and ASCI Red [24]. Our approach is to improve the single node performance of `nek5000` based on compiler technology. Gunnels et. al. provide insights and strategies for blocking matrices for matrix multiplication at each level of hierarchical memories [11]. However this does not apply to the small matrices we encounter in `nek5000`. Barthou et. al. reduce the search space by separating optimizations for in-cache computation kernels from those for memory hierarchy [1]. Similar to our approach, they obtain high performance kernels from the vendor compiler by providing it with simplified kernels. To generate code variants, they use X Language that is controlled by user provided pragmas. In our earlier work, we described a compiler-based technique that combines *specialization* with auto-tuning for matrix multiply of small, rectangular matrices [22]. In this paper, we demonstrate that the same auto-tuning strategy can be used to tune higher-level kernels that are application specific. Further, the strategy is now automated in an auto-tuning system that can exploit multiple cores in a system. In addition, we experimentally verified that this auto-tuning strategy enables high speedups even on a large supercomputer system.

6 Conclusion

This paper described an autotuning and specialization methodology applied to `nek5000`. The tuning process involves identifying data set sizes for the core computation (matrix multiply of small matrices), and providing a set of parameterized optimization scripts to the CHiLL polyhedral framework that generate specialized code. A set of heuristics prune the space of parameter values and variants as part of autotuning the implementation. We show speedups of up to 2.3X on the core computation as compared to already manually tuned code, and also much better performance than standard BLAS libraries. Performance improvements for the full `nek5000` application are 36% on one node, and up to 21% on 256 nodes. *These results are significant, as they demonstrate speedup due to compiler optimizations running at scale on a supercomputer for a code that was already manually tuned.*

Beyond tuning `nek5000`, this paper shows an approach to tuning code that is repeatable,

and permits the application to maintain high-level, architecture-independent code. We can view the CHiLL script, search-space pruning heuristics and generated library as documentation of the tuning process for a particular platform and native compiler. Over time, as this code is tuned for multiple architectures, this prior tuning work can potentially be reused, thus systematically evolving application code for new architectures from machine-independent code.

Acknowledgment This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

- [1] Denis Barthou, Sebastien Donadio, Alexandre Duchateau, William Jalby, and Eric Courtois. Iterative compilation by exploration of kernel decomposition. In *The 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2006)*, New Orleans, LA, 2006.
- [2] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing*, pages 340–347, Vienna, Austria, 1997.
- [3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
- [4] Chun Chen. *Model-Guided Empirical Optimization for Memory Hierarchy*. PhD thesis, University of Southern California, 2007.
- [5] Chun Chen, Jacqueline Chame, and Mary Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, March 2005.
- [6] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, Computer Science Department, 2008.
- [7] M.O. Deville, P.F. Fischer, and E.H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge, 2002.
- [8] Paul Fischer, James Lottes, David Pointer, and Andrew Siegel. Petascale algorithms for reactor hydrodynamics. *Journal of Physics: Conference Series*, 125, 2008.
- [9] Matteo Frigo and Steven G. Johnson. The fastest fourier transform in the west. Technical Report MIT-LCS-TR728, MIT Lab for Computer Science, 1997.
- [10] John A. Gunnels, Robert A. Van De Geijn, and Greg M. Henry. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27:422–455, 2001.
- [11] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. High-performance matrix multiplication algorithms for architectures with hierarchical memories. Technical Report CS-TR-01-22, University of Texas at Austin, 2001.
- [12] Mary W. Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, and Gabe Rudy. Transformation recipes for code generation and auto-tuning. In *International Workshop on Languages and Compilers for Parallel Computing*, October 2009.
- [13] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, 2009.

- [14] José R. Herrero and Juan J. Navarro. Improving performance of hypermatrix cholesky factorization. In *9th International Euro-Par Conference*, pages 461–469, 2003.
- [15] Intel. *Intel Fortran Compiler User and Reference Guides*, 2008. <http://www.intel.com/cd/software/products/asm-na/eng/406088.htm>.
- [16] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *ACM SIGPLAN conference on Language, compiler, and tool support for embedded systems*, San Diego, CA, 2003.
- [17] Chunhua Liao and Dan Quinlan. A rose-based end-to-end empirical tuning system for whole applications. Technical Report UCRL-SM-210032-DRAFT, Lawrence Livermore National Laboratory, August 2009.
- [18] A. T. Patera. A spectral element method for fluid dynamics - laminar flow in a channel expansion. *Journal of Computational Physics*, 54:468–488, 1984.
- [19] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Fifth International Symposium on Code Generation and Optimization (CGO'07)*, San Jose, CA, 2007.
- [20] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part ii, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, Tucson, AZ, 2008.
- [21] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.
- [22] Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, and Paul D. Hovland. Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. In *The Fourth International Workshop on Automatic Performance Tuning*, October 2009.
- [23] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable autotuning framework for compiler optimization. In *IPDPS*, Rome, Italy, May 2009.
- [24] Henry M. Tufo and Paul F. Fischer. Terascale spectral element algorithms and implementations. In *ACM/IEEE conference on Supercomputing*, Portland, OR, 1999.
- [25] Yevgen Voronenko, Frédéric de Mesmay, and Markus Püschel. Computer generation of general size linear transform libraries. In *International Symposium on Code Generation and Optimization (CGO)*, Seattle, WA, 2009.
- [26] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521–530, 2005.
- [27] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing*, 1998.
- [28] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. POET: Parameterized Optimizations for Empirical Tuning. In *IPDPS*, Long Beach, CA, March 2007.
- [29] Kamen Yotov, Xiaoming Li, Gang Ren, María Jesús Garzarán, David Padua, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, 2005.