Autotuning and Specialization: Speeding up Matrix Multiply for Small Matrices with Compiler Technology

Jaewook Shin[1], Mary W. Hall[2], Jacqueline Chame[3], Chun Chen[2], Paul D. Hovland[1]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439 USA, {jaewook,hovland}@mcs.anl.gov

[2] School of Computing, University of Utah, Salt Lake City, UT 84112 USA, {mhall,chunchen}@cs.utah.edu

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

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. This paper presents a compiler optimization approach that combines novel autotuning compiler technology with specialization for expected data set sizes of key computations, focused on matrix multiplication of small matrices. We describe compiler techniques developed for this approach, including the interface to a polyhedral transformation system for generating specialized code and the heuristics used to prune the enormous search space of alternative implementations. We demonstrate significantly better performance than directly using libraries such as GOTO, ATLAS and ACML BLAS that are not specifically optimized for the problem sizes on hand. In a case study of nek5000, a spectral element based code that extensively uses the specialized matrix multiply, we demonstrate a performance improvement for the full application of 36%.

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. This tuning process must be largely repeated to move from one architecture to another, as too often, a code that performs well on one architecture faces bottlenecks on another. As we enter the era of petascale systems, the challenges facing application programmers in obtaining acceptable performance on their codes will only grow.

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 [4, 18, 8, 15, 16].

In this paper, we consider *collaborative autotuning tools*, which works with application programmers or library developers to automate their performance tuning tasks

and permit them to express their algorithms in architecture-independent code. We describe a set of compiler tools that can be used by savvy programmers to express aspects of the tuning of their code that will be carried out systematically by the tools to arrive at a highly optimized implementation. 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.

For this paper, we focus this autotuning process on generating a matrix multiply library, customized for specific problem sizes. In particular, we focus on optimizing small matrices of known sizes, where specialization is applicable. While highly-optimized BLAS libraries (native, ATLAS, GOTO, etc.) are available on every platform, they are tuned for large (typically, square) matrices, and the most successful optimization strategies for small matrices are different. Thus, specialization for small matrices can yield better performance results than manually-tuned high-performance libraries, but autotuning is needed to identify the best-performing implementation of each problem size.

In this paper, we describe a process that combines autotuning and specialization compiler technology, and collaborates with the application (or library) programmer, targeting the Opteron Phenom processor. We demonstrate this approach in a case study using `nek5000`, a scalable spectral element code whose execution is dominated by what are essentially matrix multiplies of very small, rectangular matrices. We use CHiLL, a polyhedral loop transformation framework with a script interface, to describe the space of specialized implementations and automatically generate optimized code. Using heuristics to prune the search space of possible implementations, a set of variants generated by CHiLL are then measured and compared to derive a library of implementations specialized to problems sizes for a particular problem or domain. The resulting BLAS calls are up to 2.3X faster than the original implementation, and the overall `nek5000` application performance is improved by 36% on one node, as compared to an already manually-tuned version of the code. This paper makes three contributions:

- We describe an approach to collaborative autotuning combined with specialization that uses compiler tools and automation to derive highly-optimized matrix multiply libraries for expected input data sets.
- The code our compiler generates yields performance that is as much as 11.2x faster than the `icc` compiler, 3.3x faster than the hand-coded ACML library, 4.5x faster than ATLAS, and even 2.1x faster than the Goto BLAS.
- We demonstrate the impact of this approach on a case study of a production code `nek5000`, yielding up to 36% performance improvements over manual tuning. This process could be repeated to specialize matrix multiply for other applications for similar architectures supporting multimedia extensions such as SSE-3, and native compilers without any modification. Because it is based on compiler technology, it can also be applied to other computations beyond dense linear algebra.

The remainder of the paper is organized as follows. Section 2 presents the collaborative autotuning process and describes the compiler technology used in this process. We discuss the experimental results in Section 3, followed by a discussion of related work and a conclusion.

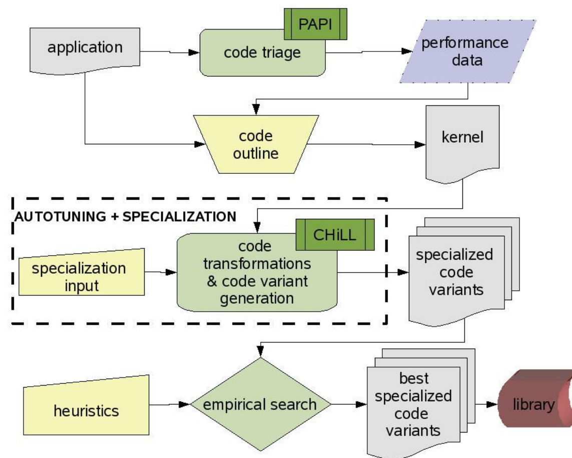


Fig. 1. Overview of our approach

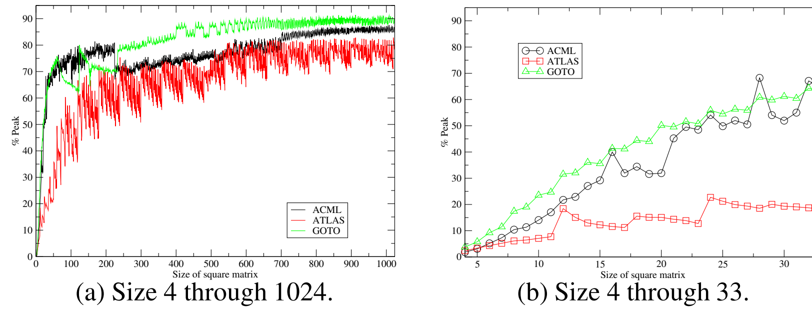
2 Compiler Technology: Autotuning and Specialization

Collaborative autotuning tools automate performance tuning tasks with assistance from the application programmer. Systematic tasks such as code triage, code transformation and code generation can be performed by tools, resulting in highly optimized code implementations. The application programmer assists the tuning process by identifying performance issues, determining optimization strategies (including specialization) and guiding the evaluation process.

In this section we describe our collaborative autotuning methodology and how a programmer might use tools to automate several aspects of performance tuning. Figure 1 shows the performance tuning steps in our methodology (*code triage*, *code outlining*, *autotuning for specialized codes* and *code generation*) and how they might be performed using autotuning tools with assistance from the application programmer.

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). The triage and autotuning steps can be automated [1], but further discussion is beyond the scope of this paper.

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.



(a) Size 4 through 1024. (b) Size 4 through 33.

Fig. 2. Performance of BLAS libraries on matrix multiplication.

2.1 Optimization Strategy

For large matrices that exceed the capacity of various levels of cache, performance BLAS libraries such as ACML (native), ATLAS, and GOTO all perform well, achieving above 70% of peak performance as seen in Figure 2(a). This is because they apply 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 [4, 18, 6, 9, 20].

However if we look closer at matrices of size 10 or smaller, those same BLAS libraries perform below 25% of peak performance as seen in Figure 2(b). This is because the optimization strategy for small matrices should be different. 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 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.

2.2 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 allows the programmer to apply complex transformation strategies to a loop nest by specifying a series of composable loop transformations using a high-level script interface [5, 7, 16].

CHiLL is a state-of-the-art polyhedral loop transformation framework designed with the autotuning environment in mind. Its high-level script interface can be used by application programmers or compilers for generating high-quality code. Transformations supported by CHiLL include loop tiling, index set splitting, loop permutation, unroll-and-jam, fission, fusion, unimodular transformations and data copying. Binding of optimization parameters (such as unroll factors in this experiment) within a CHiLL script can be performed by an external autotuning search engine, such as the one described in the next section, or as in [16].

Figure 3 illustrates the use of CHiLL for optimizing matrix multiplication with small input sizes. The loop nest in Figure 3(a) is imperfectly nested since array C is initialized before multiplication. Figures 3(b), (c) and (d) show three CHiLL scripts that correspond to different versions of the code in (a), with different loop orders.

To permute the loop order the programmer only needs to specify one `permute` command, where the loop numbers indicate the loop order after permutation. Loops are numbered starting at 1 from the outermost loop and increasing as we move inward. The scripts in (c) and (d) permute the loop order to j,k,i and k,i,j, respectively, while the script in (b) maintains the original loop order i,j,k. Figures 3(e) and (f) show the transformed code versions after permuting the loops as specified in (c) and (d), respectively. For brevity, in (e) and (f) we show the simplest versions where all unroll amounts are 1.

The command `unroll` performs unroll-and-jam if the target loop is an outer loop and its inner loops can be fused together. 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 scripts in Figures 3(b), (c) and (d) are different in the number of `unroll` commands; this is necessary because the number of loops are different after permutation, as shown in (a), (e) and (f), depending on the loop order.

The *known* command provides support for *specialization*, allowing the programmer to express known values such as input sizes and loop bounds. Within CHiLL, *known* adds additional conditions to the iteration spaces extracted from the original code. These conditions improve 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. From the scripts in (b), (c) and (d), CHiLL automatically generates correct transformed code. Figure 3(g) shows the result of script (c) when all unroll sizes u1, u2 and u3 are set to 2.

The flexibility provided by CHiLL's script interface greatly helps programmers, since they can now focus on how the transformations affect performance instead of the details of generating correct transformed code. The above scripts plus additional ones for different loop orders combined with eight different `known` statements were what we used to tune `nek5000`. The next section discusses the heuristics used to reduce the number of scripts and their parameter space for larger specialized matrix sizes.

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

(b) loop order i,j,k

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

(c) loop order j,k,i

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

(d) loop order k,i,j

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

(e) After loop permutation in (c)

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

(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

Fig. 3. Example of CHiLL scripts and the generated codes.

2.3 Autotuning: Heuristics for 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 describe the pruning heuristics we used, and then describe how they are extracted from the results of an exhaustive search on a particular matrix size in the next section. We generate only the code variants that satisfy all four heuristics.

Heuristic 1: Loop order. Certain loop orders that lead to lower-performance code variants are pruned from the search. We select the loop orders based on the performance evaluations for a particular matrix size as described in Section 3.

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. With this limit, the search space size is restricted to a number of tuples (U_m, U_k, U_n) where U_m, U_k and U_n are the unroll amounts for m, k and n-loop respectively, and satisfy $U_m \times U_k \times U_n < C$.

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. With this heuristic, the search space is pruned by the ratio of $\frac{mk+kn+mn}{mkn} = \frac{1}{n} + \frac{1}{m} + \frac{1}{k}$.

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 mod 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. The search space is pruned by a factor of $\frac{\sigma_0(m)\sigma_0(k)\sigma_0(n)}{mkn}$ where $\sigma_0(m)$ is the number of divisors of m.

The four heuristics described above are shown to be effective in finding high performance code for small-matrix multiplication on a particular architecture within a reasonable search time as shown in Section 3. However, at a higher level, the framework we describe can be used for other application-architecture pairs just by replacing heuristics and code transformations. As for the four heuristics in our example, the heuristics for other settings can reflect the features of the architecture, compiler and/or the application. Since the knowledge about a particular architecture or application is enclosed in a set of code transformations and heuristics as components, it is straightforward to adapt our framework to another application and/or architecture. Such heuristics on compilers, architectures and application kernels can be shared among users by collecting them in a 3D-table that shows architectures, kernels and compilers on each of the three axes.

2.4 Building the Library

After each code variant is generated, it is compiled and linked with the driver that measures and records the performance. A careful measurement process must be used to

select the best-performing code variant. We consider measurement overhead and performance fluctuations in deriving our measurements. Measurement overhead is significant for the short execution times of matrix multiplication for small matrices; we reduce this overhead by increasing the number of executions of the code per measurement (500 times, as described in the next section). Because of performance fluctuations, such as servicing interrupts and varying cache states, which occur somewhat randomly, we execute a large number of independent measurements (100 as in the next section), and take the minimum execution time. As a second consideration, the compiler flags and array declarations have to be carefully selected so that the backend compiler can generate the most efficient code whenever possible. For example, alignment information of arrays is important for the backend compiler to generate efficient SIMD instructions. Such information can be delivered to the backend compiler by making the code variants and the driver as simple as possible and by using language extensions supported by the compiler and the command line options.

```
(1)   mxm(a, M, b, K, c, N){
(2)      if (all a, b and c are aligned to the SIMD register width){
(3)         if (M == 10){
(4)            if (K == 8){
(5)               if (N == 10){ mxm10810(a,b,c); return;}
(6)               if (N == 64){ mxm10864(a,b,c); return;}
(7)            } else if (K == 10){
(8)               if (N == 10){ mxm101010(a,b,c); return;}
(9)               if (N == 100){ mxm1010100(a,b,c); return;}
(10)        } else if (M == 100 && N == 10){
(11)           if (K == 8){ mxm100810(a,b,c); return;}
(12)           if (K == 10){ mxm1001010(a,b,c); return;}
(13)        } else if (M == 8 && K == 10){
(14)           if (N == 8){ mxm8108(a,b,c); return;}
(15)           if (N == 100){ mxm810100(a,b,c); return;}
(16)        }}
(17)     mxm44_0(a, M, b, K, c, N);}
```

Fig. 4. A template of the wrapper code for specialized matrix multiplication routines.

When the evaluation is complete for the generated code variants, the best performing variant is selected for each matrix size. In addition to the object files of the code variants, a wrapper code is necessary to call the specialized matrix multiplication routines depending on the matrix sizes. This wrapper code 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`. Figure 4 shows the wrapper code that is used to call eight specialized routines for the sizes in Table 1. First, at line 2, the array addresses are checked for alignment to the SIMD register size. If all arrays are aligned, the three arguments for matrix size are examined and a specialized code is invoked if one is available that matches the three parameters; otherwise, the original manually-tuned

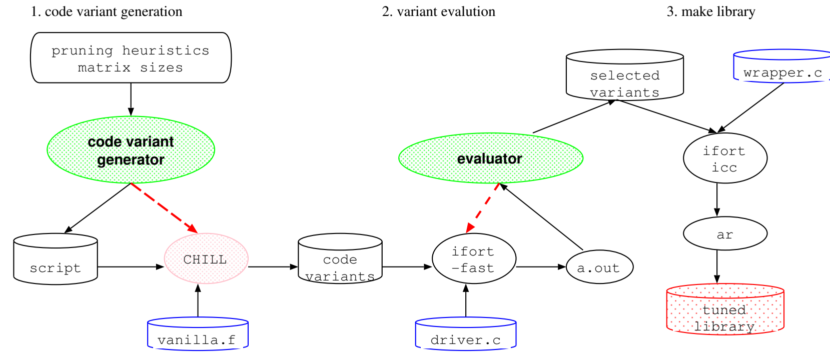


Fig. 5. Block diagram describing autotuning tools and experiment.

version is invoked. Finally, a library is generated from the wrapper code and the tuned object codes.

3 Experiments

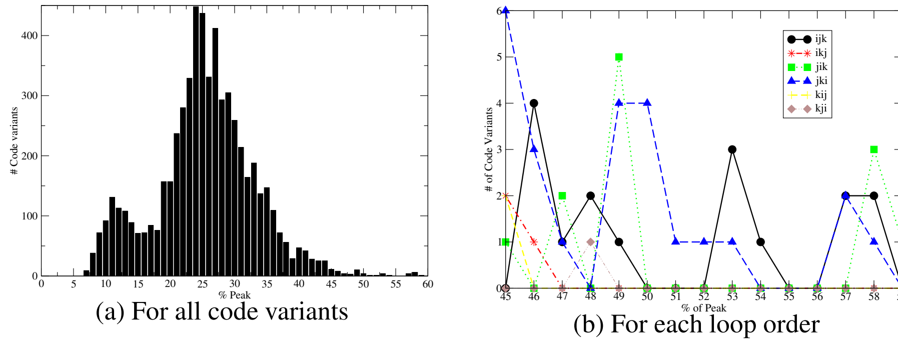
This section demonstrates the performance impact of the approach and set of compiler tools described in the previous two sections, using `nek5000`.

3.1 Experimental Environment

The platform for these experiments is a 2.5 GHz AMD Opteron 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. CHiLL version 0.1.5 [7] and the Intel compiler version 10.1 [12] are used to transform and compile the code variants. All performance measurements are on single core unless mentioned otherwise.

3.2 Generating the Library

Figure 5 shows the block diagram of the experimental flow. It takes as input matrix sizes for which specialization is desired, a matrix multiplication kernel (`vanilla.f`), shown in Figure 3(a) and a driver (`driver.c`) that measures the performance using PAPI_TOT_CYC. The driver executes a variant 500 times per measurement warming up the L1 data cache, collects 100 such measurements and records 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. The 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 have the compiler to generate aligned SIMD instructions, `__attribute__((aligned (16)))` qualifier



(a) For all code variants

(b) For each loop order

Fig. 6. Number of code variants across performance in % of peak.

is added to array declarations and the interprocedural optimization feature was used with `-fast` for `ifort` and `-O3 -ipo` for `icc`.

The AMD Phenom processor's peak double-precision floating point operations per cycle is 4, and the machine's peak is 10 GFlops. We use *percentage of machine's peak* as the main metric:

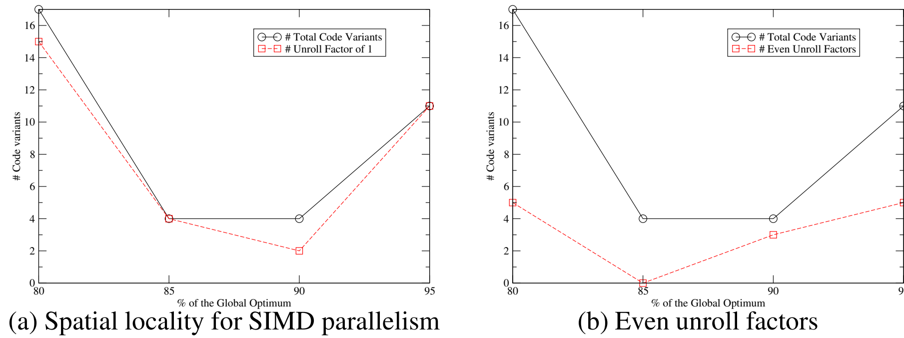
$$\text{percentage of machine's peak} = \frac{\frac{m \times k \times n \times 2}{\text{measured run time in cycles}} \times 100}{\text{machine's peak floating point operations per cycle}}$$

We optimize `nek5000` in the context of the `helix2` input data set. 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. `mxm44_0` takes the same set of arguments as `mxm` of Figure 4. The main loopnest is unrolled by 4 for each of `m` and `n` loop. By instrumenting `mxm44_0`, we found 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.

Table 1. Pruning of search space by heuristics(% of remaining search space).

`m,k,n`	Total	Loop Order	SIMD	EvenUnl	I-Cache	All heuristics	LO	U_i	U_k	U_j
8,10,8	3840	1920(50.0)	1194(31.1)	384(10.)	3840(100.0)	111(2.9)	ijk	8	10	4
10,8,10	4800	2400(50.0)	1398(29.1)	384(8.0)	4800(100.0)	111(2.3)	ijk	1	8	5
10,10,10	6000	3000(50.0)	1626(27.1)	384(6.4)	6000(100.0)	111(1.9)	jik	1	9	5
10,8,64	30720	15360(50.0)	6906(22.5)	672(2.2)	28170(91.7)	174(0.6)	ijk	1	8	4
8,10,100	48000	24000(50.0)	10578(22.0)	864(1.8)	38940(81.1)	216(0.5)	ijk	1	10	4
100,8,10	48000	24000(50.0)	10578(22.0)	864(1.8)	38940(81.1)	216(0.5)	jki	1	8	5
10,10,100	60000	30000(50.0)	11886(19.8)	864(1.4)	45264(75.4)	216(0.4)	jik	1	10	4
100,10,10	60000	30000(50.0)	11886(19.8)	864(1.4)	45264(75.4)	216(0.4)	jik	1	10	10

The first column in Table 1 shows the top eight matrix sizes chosen from the list sorted in decreasing order of importance, comprising about 74% of the overall estimated computation for `mxm44_0`. The second column of Table 1 shows the size of the search



(a) Spatial locality for SIMD parallelism (b) Even unroll factors

Fig. 7. Supporting graphs for the two heuristics.

space, the product of the number of all loop orders and unroll factors for each of the three loops. For example, for a size (8,10,8), the size of the search space is $3840 = 6 \times 8 \times 10 \times 8$.

3.3 Evaluating the Pruning Heuristics

The set of heuristics used to prune the space of matrix multiply implementations is based on data collected during the exhaustive search for a particular matrix size, $10 \times 10 \times 10$. The assumption is that the search space of implementations for a small matrix size (such that data fit in the L1 data cache) shares characteristics with the search space of other small matrix sizes. Good candidates for building pruning heuristics for other input sizes are characteristics that affect performance in predictable patterns independently of input sizes. The characteristics observed in this experiment are loop order, spatial locality across loop iterations, and instruction cache misses due to code size. We do not include data cache behavior that is dependent on matrix size, and consider only sizes such that all data fit in the L1 data cache.

For matrix size (10,10,10) the exhaustive search takes about 7 hours when 6 search processes are run in parallel, one for each loop order. Figure 6(a) shows the distribution of code variants for size (10,10,10) along the X-axis as a percentage of peak. The fastest runs at 59.29% and the slowest at 7.14% of peak.

Figure 6(b) shows the number of code variants that perform higher than 45% of machine's peak for each loop order. Each line in the graph represents a different loop order, where ijk represents the original loop order as in Figure 3(a). Only three loop orders, ijk, jik and jki, result in performance higher than 49% of peak for size (10,10,10). Therefore the first heuristic prunes all other loop orders (ikj, kij and kji) from the search space of other matrix sizes.

The solid line of Figure 7(a) shows the number of code variants as the percentage of the highest performance. For example, 4 at 90% means that there are 4 code variants that perform higher than or equal to 90% but lower than 95% of the highest performance. The dashed line shows the number of code variants with at least one unroll factor of 1 (no unroll). As discussed in Section 2.3, this heuristic is used to filter out code variants without spatial locality across adjacent iterations.

Figure 7(b) is the same as (a) except that the dashed line represents the number of code variants whose three unroll factors evenly divide the corresponding iteration

count. For example, an iteration count of 10 has four even unroll factors, 1, 2, 5 and 10. If the iteration count is 10 for all three loops, there are 64 ($= 4^3$) variants that satisfy this heuristic for each loop order.

The heuristic concerning L1 instruction cache misses was obtained by increasing the square matrix size and executing the fully unrolled loops for each size. The instruction cache misses, measured after warming up the cache, start increasing when the matrix size is 13 or larger. Since the total unroll factor when N=13 is 2197($= 13^3$), this heuristic limits the total unroll factor to less than or equal to 2197, for all matrix sizes.

Columns 3-6 of Table 1 show the size of the remaining search space when each of the four heuristics is used in isolation to prune the search space. The seventh column of Table 1 shows the size and the ratio of the remaining space when all four heuristics are used. As the size of the search space increases, the pruning heuristics reduce the percentage of the remaining search space down to less than 1%. We used these heuristics to select the best variants for the five larger sizes. When these heuristics are used, the tuning time for each matrix size is less than an hour from beginning of code generation to the completion of evaluation. The four columns on the right show the loop order and the three unroll factors for *i*, *k* and *j*-loop for the selected eight code variants.

3.4 Performance Results

We chose the fastest variant for each matrix size to create a library of eight specialized matrix multiplication routines and the wrapper routine. This section presents the performance results obtained by using this specialized library.

Comparison with other BLAS implementations We compare this library against the naive `vanilla.f` matrix multiply version from Figure 3(a), the manually optimized `mxm44_0` and `mxf8/10` implementations, and for completeness, several BLAS performance libraries, as shown in Figure 8. The comparison is performed for the eight matrix sizes we selected for tuning. The X-axis represents matrix sizes in the form of "m,k,n" where the two input matrices are $m \times k$ and $k \times n$ in size. The Y-axis represents the percentage of machine's peak. Although BLAS libraries are highly tuned for large matrix multiply, they are not optimized for small matrix multiply because small matrix multiply requires aggressive unrolling and different loop orders that vary for the slightest change in matrix sizes, a completely different optimization strategy from the large matrices. For ATLAS, we use the architectural default for `AMD64K10h64SSE3` in version 3.8.2. The versions for ACML and GOTO BLAS are 4.1.0 and `goto_barcelona-r1.26`, respectively.

The performance of `mxm44_0` is roughly flat around 23% of peak across all sizes. `mxf8/10` is also included in the NEK5000 code although it is not used when the `K` dimension is greater than or equal to 4. For the eight small matrices, however, `mxf8/10` is faster than `mxm44_0` achieving close to 40% of peak. We include `vanilla.f` to show the quality of the code generated by the native compiler from the naive implementation. The performance is around 7% of peak. ACML and GOTO BLAS are manually implemented BLAS routines. While their performance reaches 90% of peak and higher for large square matrices, the performance for the three smallest matrices is still less

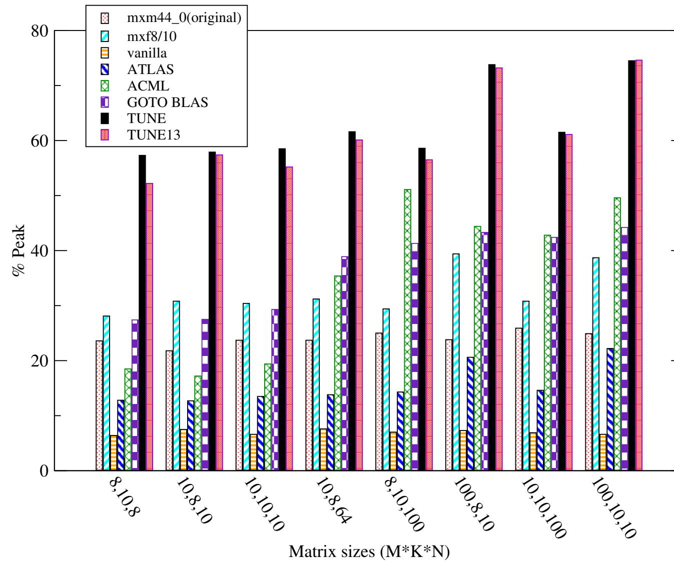


Fig. 8. Speedups of the library generated by our approach

than 30% of peak due to the reason we mentioned above. `TUNE` represents the code tuned with our approach for the eight matrix sizes in the X-axis. Compared to the three BLAS libraries that take 13 parameters, `TUNE` takes only 6 parameters as in Figure 4. For the eight sizes, `TUNE` is the fastest reaching as high as 74% of peak. Compared to the default implementation `mxm44_0`, `TUNE` is more than 2.3 times faster. To show the effect of fewer function parameters of `TUNE`, `TUNE13` implements the wrapper in Figure 4 within a `dgemm` reference implementation [2], and thus supports not only the standard 13-parameter interface but also the full semantics. For the eight sizes, the performance drop caused by the larger number of parameters is not significant.

Performance of Nek5000 on Helix2 on Phenom Next, we measured the sequential performance improvement for `nek5000` for the `helix2` problem. 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 smallest run time as the measurement for each of the baseline that uses `mxm44_0` and the run that uses the specialized library. When the tuned library is used, the run time drops significantly from 190 seconds to 140 seconds. This is a speedup of 1.36X over the baseline. Moreover, this speedup is achieved through a simple empirical tuning approach and replacing the hand-tuned code. This experiment demonstrates the power of empirical performance tuning and code specialization.

4 Related Work

There have been much prior work on automatically tuning libraries. PHiPAC and ATLAS tune matrix multiplication code automatically for many target machines [4, 18].

FFTW is a self-tuning library designed to generate high performance code for Discrete Fourier Transforms [8]. SPIRAL is a high-performance code generation system for digital signal processing transforms [15]. OSKI combines install-time evaluations with run-time models to tune sparse-matrix vector multiplication and other solvers such as triangular solver [17].

Compiler assisted autotuning and tools facilitating code transformations have also been extensively studied. Knijnenburg et. al. compared various search algorithms in the space of tiling two dimensions and unrolling one dimension for multiple loop orders of matrix multiplication [14]. Chen et. al. combine compiler models and heuristics with guided empirical evaluations to take advantage of their complementary strengths [6]. 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 [16]. Hartono et. al. use annotations in the code to describe performance improving transformations for C programs [10]. POET is a scripting language for parameterizing complex code transformations [19], which can be used in an autotuning process as well.

For tuning matrix multiply for small matrices, the work most closely related to ours is Herrero and Navarro's, which focus on specializing matrix multiplication for small matrices [11]. However, their code variants were generated manually and it's not clear how many code variants in the parameter space were evaluated. In contrast, our heuristic-based parameter space pruning is automated, and thus can select the best code from a larger set of code variants. Kaushik et. al. compared a hand-optimized tensor matrix vector multiplication routine with `mxm` in `nek5000` and the `dgemm` of Intel's MKL, and showed the hand-optimized routine performed the best for small, highly rectangular matrices [13]. Barthou et. al. reduce the search space by separating optimizations for in-cache computation kernels from those for memory hierarchy [3].

5 Conclusion

This paper describes an autotuning approach for specializing matrix multiply according to problem size, applied to case study `nek5000`. The tuning process involves providing a set of parameterized optimization scripts to the CHiLL polyhedral framework that generates specialized code. A set of heuristics prune the space of parameter values and variants as part of autotuning the implementation. Specialization for matrix size and a focus on optimization strategies for small matrices enables our automated approach to obtain better performance than carefully hand-coded BLAS libraries. The specialized code our compiler generates yields performance that is as much as 11.2x faster than the `icc` compiler, 3.3x faster than the hand-coded ACM L library, 4.5x faster than ATLAS, and even 2.1x faster than the Goto BLAS. We show speedups of more than 2.3X on the core computation as compared to already manually tuned matrix multiply in `nek5000`, and performance improvement for the full application is 36% on one node. Looking forward, this paper shows an approach to tuning code that is repeatable, and permits the application to maintain high-level, architecture-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. http://rosecompiler.org/.
2. http://www.netlib.org/blas/.
3. Denis Barthou, Sebastien Donadio, Patrick Carribault, Alexandre Duchateau, and William Jalby. Loop optimization using hierarchical compilation and kernel decomposition. In *International Symposium on Code Generation and Optimization*, San Jose, CA, 2007.
4. 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.
5. Chun Chen. *Model-Guided Empirical Optimization for Memory Hierarchy*. PhD thesis, University of Southern California, 2007.
6. 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.
7. 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.
8. Matteo Frigo and Steven G. Johnson. The fastest fourier transform in the west. Technical Report MIT-LCS-TR728, MIT Lab for Computer Science, 1997.
9. John A. Gunnels, Robert A. Van De Geijn, and Greg M. Henry. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27, 2001.
10. 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.
11. José R. Herrero and Juan J. Navarro. Improving performance of hypermatrix cholesky factorization. In *9th International Euro-Par Conference*, pages 461–469, 2003.
12. Intel. *Intel Fortran Compiler User and Reference Guides*, 2008. http://www.intel.com/cd/software/products/asmo-na/eng/406088.htm.
13. Dinesh K. Kaushik, William Gropp, Michael Minkoff, and Barry Smith. Improving the performance of tensor matrix vector multiplication in cumulative reaction probability based quantum chemistry codes. In *15th International Conference on High Performance Computing (HiPC 2008)*, volume 5374 of *Lecture Notes in Computer Science*. Springer, 2008.
14. P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1):43–67, 2003.
15. 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.
16. Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable autotuning framework for compiler optimization. In *IPDPS*, Rome, Italy, 2009.
17. 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.
18. R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing*, 1998.
19. Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. POET: Parameterized Optimizations for Empirical Tuning. In *IPDPS*, Long Beach, CA, March 2007.
20. 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.