

Exploring the Optimization Space of Dense Linear Algebra Kernels

Qing Yi¹ Apan Qasem²

¹ University of Texas at San Antonio

² Texas State University

Abstract. Dense linear algebra kernels such as matrix multiplication have been used as benchmarks to evaluate the effectiveness of many automated compiler optimizations. However, few studies have looked at collectively applying the transformations and parameterizing them for external search. In this paper, we take a detailed look at the optimization space of three dense linear algebra kernels. We use a transformation scripting language (POET) to implement each kernel-level optimization as applied by ATLAS. We then extensively parameterize these optimizations from the perspective of a general-purpose compiler and use a stand-alone empirical search engine to explore the optimization space using several different search strategies. Our exploration of the search space reveals key interaction among several transformations that must be considered by compilers to approach the level of efficiency obtained through manual tuning of kernels.

1 Introduction

Compiler optimizations have often targeted the performance of linear algebra kernels such as matrix multiplication. While issues involved in optimizing these kernels have been extensively studied, difficulties remain in terms of effectively combining and parameterizing the necessary set of optimizations to consistently achieve the level of portable high performance as achieved manually by computational specialists through low-level C or assembly programming. As a result, user applications must invoke high-performance domain-specific libraries such as ATLAS [10] to achieve a level of satisfactory efficiency. ATLAS produces extremely efficient linear algebra kernels through a combination of domain/kernel specific optimization, hand-tuned assembly, and an automated empirical tuning system that uses direct timing to select the best implementations for various performance-critical kernels. Libraries such as ATLAS are necessary because native compilers can rarely provide a similar level of efficiency, either because they lack domain-specific knowledge about the input application or because they cannot fully address the massive complexity of modern architectures.

To improve the effectiveness of conventional compilers, many empirical tuning systems have been developed in recent years [1, 4, 8, 9, 11, 14]. These systems have demonstrated that search-based tuning can significantly improve the efficacy of many compiler optimizations. However, most research in this area has

focused on individual or a relatively small set of optimizations. Few have collectively parameterized an extensive set of optimizations and investigated the interactions among them. One impediment to parameterizing a large class of transformations is that, as yet, we have no standard representation for parameterized optimizations and their search spaces. Because of this, most tuning systems consist of highly specialized code optimizers, performance evaluators and search engines. Hence, exploring a larger search space comes with the burden of implementing additional parameterized transformations. This extra overhead has, to some degree, limited the size of the search space investigated by any one tuning system. In this paper, we describe a system in which we interface a parameterized code transformation engine with an independent search engine. We show that by leveraging the complementary strengths of these two modules we are able to explore the search space of a large collection of transformations.

While most compilers understand well the collection of code optimizations that are required to achieve high performance, it is often the details in combining and collectively applying the optimizations that determine the overall efficiency of the optimized code. In our previous work [12], we have used POET, a transformation scripting language, to implement all the kernel-level optimizations as applied by ATLAS for three dense linear algebra kernels: *gemm*, *gemv*, and *ger*. Our previous work has achieved comparable, and sometimes superior, performance to those of the best hand-written assembly within ATLAS [12]. The previous work, however, utilized kernel-specific knowledge obtained from ATLAS when applying the optimizations and when searching for the best-performing kernels. In this paper, we extensively parameterize the optimization spaces from the perspective of a general-purpose compiler. We then use an independent search engine to explore this parameter space to better understand the delicate interplay between transformations. Such interactions must be considered by a compiler when combining and orchestrating different program transformations to approach a similar level of efficiency as achieved by ATLAS. The contributions of this paper include:

1. parameterization of an extensive set of optimizations that need to be considered by a general-purpose compiler to achieve high performance;
2. integration of an independent search engine and a program transformation engine;
3. empirical exploration of the search space that reveals key interaction among optimizations.

2 Related Work

A number of successful empirical tuning systems provide efficient library implementations for important scientific domains, such as those for dense and sparse linear algebra [5, 3], signal processing [6, 7] and tensor contraction [2]. POET [12] targets general-purpose applications beyond those targeted by domain-specific research and complements domain-specific research by providing an efficient transformation engine that can make existing libraries more readily portable to different architectures.

Several general-purpose autotuning tools can iteratively re-configure well-known optimizations according to performance feedback of the optimized code [1, 4, 8, 9, 11, 14]. Both POET and the parameterized search engine described in this paper can be easily integrated with many of these systems. POET supports existing iterative compilation frameworks by providing an output language for parameterizing code optimizations for empirical tuning.

The work by Yotov et al. [13] also studied the optimization space of the *gemm* kernel in ATLAS. However, their work used the ATLAS code generator to produce optimized code. Because the ATLAS code generator is carefully designed by computational specialists based on both architecture- and kernel-specific knowledge, the optimization space they investigated does not represent the same degrees of freedom that a general-purpose compiler must face. In contrast, we use the POET language to parameterize the different choices typically faced by general-purpose compilers and investigate the impact and interactions of the different optimization choices.

3 Orchestration of Optimizations

We used a transformation scripting language named POET to implement an extensive set of optimizations necessary to achieve the highest level of efficiency for several ATLAS kernels [12]. As shown in Fig. 1, a POET transformation engine includes three components: a language interpreter, a transformation library, and a collection of front-end definitions which specialize the transformation library for different programming languages such as C, C++, FORTRAN, or Assembly.

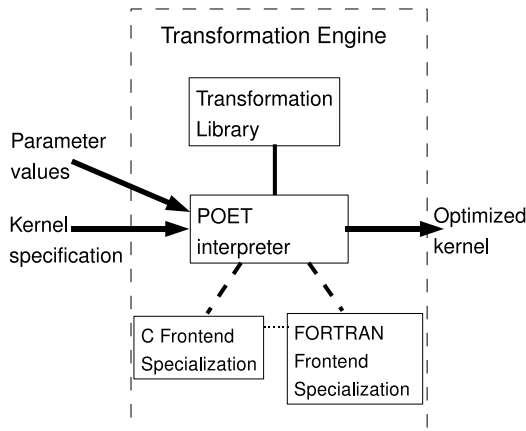


Fig. 1. POET transformation engine

The transformation engine takes as input an optimization script from an analyzer (in our case, a developer) and a set of parameter configurations from a search driver. An optimized code is output as the result, which is then empirically tested and measured by the search driver until a satisfactory implementation is found. For more details, see [12].

The optimization scripts that we developed for the ATLAS kernels have been parameterized to reflect the degrees-of-freedom that a model-driven compiler must face when orchestrating general-purpose optimizations. The optimizations focus on the efficient management of registers and arithmetic operations and are invoked by higher-level routines in ATLAS after cache-level optimizations have already been applied by ATLAS. Fig. 2 shows the reference implementations of various ATLAS kernels. The corresponding higher-level routines perform identi-

```

void ATL_USERMM(int M,
int N,int K,double alpha,
const double *A, int lda,
const double *B, int ldb,
double beta, double *C,
int ldc) {
int i, j, l;
for (j=0; j<N; j+=1) {
for (i=0; i<M; i+=1) {
C[j*ldc+i] =
beta*C[j*ldc+i];
for (l=0;l<K;l+=1) {
C[j*ldc+i]+=
A[i*lda+l]*B[j*ldb+l];}
} } }
} } }
(a) gemm kernel

void ATL_dgemvT(int M,
int N,double alpha,
const double *A, int lda,
const double *X,int incX,
double beta,double *Y,
int incY) {
int i, j;
for (i=0; i<M; i+=1) {
Y[i] = beta * Y[i];
for (j=0;j<N;j+=1) {
Y[i] +=
A[i*lda+j]*X[j];
} }
} }
(b) gemv kernel

void ATL_dger(int M,
int N, double alpha,
const double *X,int incX,
const double *Y,int incY,
double *A, int lda) {
int i, j;
for (j=0; j<N; j+=1) {
for (i=0;i<M;i+=1) {
A[j*lda+i] +=
X[i]*Y[j*incY];
}
}
}
(c) ger kernel

```

Fig. 2. Reference implementations of ATLAS kernels

cal computations but operate on larger matrices that may not fit in cache and perform cache blocking (for matrix multiplication, data copying is additionally applied to matrices A and B) before invoking the kernel implementations.

ATLAS has specialized each kernel implementation to take advantage of the known blocking strategy. We used optimization scripts to similarly specialize our kernel implementations. Our future work will use POET to integrate cache-level blocking directly with low-level kernel optimizations.

Our optimizations include two groups: register reuse optimizations (loop unroll&jam and scalar replacement) and arithmetic optimizations(SSE vectorization, strength reduction, loop unrolling, and memory prefetch). Our optimizations were carefully orchestrated to avoid introducing any inefficiency.

3.1 Register Reuse Optimizations

We applied two optimizations, unroll-and-jam and scalar replacement, to promote register reuse. Fig. 3 shows the result of applying unroll-and-jam and scalar replacement to the *gemm* kernel in Fig. 2(a). For *gemv* and *ger* (shown in Fig. 2(b) and (c)), we similarly applied unroll-and-jam to the outer loops, and scalar replacement to all the matrix/vector accesses.

We parameterized the optimizations with an unroll factor for each loop and the ordering between the two optimizations. For example, the optimizations for *gemm* (Fig 2(a)) were parameterized with three parameters: *uJ* (unroll factor for loop J), *uI* (unroll factor for loop I), and *permuteReg* (the ordering between unroll-and-jam and scalar replacement for different matrices). In Fig. 3, the code on the left shows the result of applying unroll-and-jam followed by scalar replacement to all matrices; the code on the right shows the result of reverting the ordering of unroll-and-jam with scalar replacement for matrices A and B.

Most tuning systems treat loop unroll factors as part of the empirical search space. However, few systems (including ATLAS) have used empirical tuning to determine the ordering between optimizations. Most model-driven compilers perform unroll-and-jam before scalar-replacement, which is the right decision in a majority of situations. However, as shown in Fig. 3, unroll-and-jam significantly increased the number of different memory locations accessed at each iteration of

```

void ATL_USERMM(const int M, const int N, const int K,
               const double alpha, const double *A, const int lda,
               const double *B, const int ldb, const double beta,
               double *C, const int ldc)
{
    int i, j, l;    double *pA0, *pA00, *pB0, *pB00, *pC0, *pC00;
    pB0=B; pC0=C;
    for (j = 0; j < NB; j += 2) {
        pA0 = A; pC00=pC0;
        for (i = 0; i < MB; i += 2) {
            c_buf_0_0=*pC00; c_buf_1_0=(pC00+ldc); c_buf_0_1=(pC00+1); c_buf_1_1=(pC00+ldc+1);
            c_buf_0_0 = beta * c_buf_0_0; c_buf_1_0 = beta * c_buf_1_0;
            c_buf_0_1 = beta * c_buf_0_1; c_buf_1_1 = beta * c_buf_1_1;
            pA00=pA0; pB00=pB0;
            for (l = 0; l < KB; l +=1) {
                ----- | if A,B scalarRepl is done first -----
                | a_buf_0 = *pA00; | | a_buf_0 = *pA00; b_buf_0=*pB00; | |
                | a_buf_1 = *(pA00+KB); | | c_buf_0_0 += a_buf_0 * b_buf_0; | |
                | b_buf_0 = *pB00; | | a_buf_0 = *pA00; b_buf_0=(pB00+KB); | |
                | b_buf_1 = *(pB00+KB); | | c_buf_1_0 += a_buf_0 * b_buf_0; | |
                | c_buf_0_0 += a_buf_0 * b_buf_0; | ==>| a_buf_0 = *(pA00+KB); b_buf_0=*pB00; | |
                | c_buf_1_0 += a_buf_0 * b_buf_1; | | c_buf_0_1 += a_buf_0 * b_buf_0; | |
                | c_buf_0_1 += a_buf_1 * b_buf_0; | | a_buf_0 = *(pA00+KB); b_buf_0=(pB00+KB); | |
                | c_buf_1_1 += a_buf_1 * b_buf_1; | | c_buf_1_1 += a_buf_0 * b_buf_0; | |
                -----
                pA00+=1; pB00+=1;
            }
            *pC00=c_buf_0_0; *(pC00+ldc)=c_buf_1_0; *(pC00+1)=c_buf_0_1; *(pC00+ldc+1)=c_buf_1_1;
            pA0 += KB; pC00 += 1;
        }
        pB0 += KB; pC00+=ldc;
    }
}

```

Fig. 3. *gemm* optimized with unroll&jam, scalar replacement, and strength reduction

loop l . These memory references subsequently require a large number of scalar variables during scalar replacement. For example, if we set the unroll factors to be 12 for loop I in Fig. 3 then a total of 12 scalar variables will be required for matrix A. The large number of scalar variables could potentially disrupt register allocation at a later stage and cause performance break-down. In contrast, if scalar replacement for matrices A and B are applied before unroll-and-jam, scalar variables are reused across different iterations of the original loop, reducing register pressure. However, if scalar replacement is always performed first, the reuse of scalar variables could create artificial dependences that disrupt instruction scheduling at a later stage. Scalar replacement can also disable subsequent application of unroll-and-jam, and the repetitive load of the scalar variables can be a source of inefficiency. We believe that performing unroll-and-jam before scalar replacement is the correct decision in most cases. However, we parameterized their ordering for the purpose of empirically investigating their interaction.

3.2 SSE Vectorization

Some Intel and AMD processors provide specialized floating point SSE registers that allow two double-precision or four single-precision floating point values to be operated simultaneously, potentially doubling or quadrupling the peak MFLOPS of a computer. SSE vectorization therefore could significantly boost the performance of user applications.

We implemented SSE vectorization support in POET and applied it to all three kernels where architecture allows. Before applying the optimization, all floating-point operations need to be translated into three-address code, a SSE register must be allocated to each floating-point scalar variable, and an innermost loop must be identified to be vectorized. Our POET optimization scripts applies SSE vectorization to all kernels based on kernel-specific knowledge about their dependence constraints. All required knowledge, however, can be automatically discovered by a compiler via loop dependence analysis. We parameterized SSE vectorization with two parameters: the number of available SSE registers and the size (length) of each SSE register. These parameters can be automatically determined when an application is ported to a new architecture. The optimization is automatically turned off if there are not enough SSE registers to vectorize the given code.

3.3 Strength Reduction and Loop Unrolling

To promote efficiency of arithmetic operations, two additional optimizations, strength reduction and loop unrolling, need to be applied to the result of SSE vectorization. Strength reduction significantly reduces the cost of matrix/vector references inside loops. Loop unrolling ensures that sufficient number of instructions are available to support pipelining of different functional units. Both optimizations need to be applied after SSE vectorization because both could disable the vectorization of loops. Fig. 3 shows the result of applying strength-reduction to *gemm*. For microprocessors that offer hardware support for dynamic scheduling of instructions, strength reduction combined with loop unrolling could be sufficient for satisfactory instruction-level efficiency.

We parameterized both optimizations with two parameters: the unroll factor for the innermost loop, and the ordering between the two transformations. Using Fig. 3 as example, if unrolling of loop l is performed after strength reduction, the pointer induction variables (e.g., pA00,pB00) introduced by strength reduction are incremented more frequently (every iteration of the original loop) than necessary (every iteration of the unrolled loop). In contrast, applying loop unrolling before strength reduction may cause the pointer induction variables (e.g., pA00) to be incremented only once in the unrolled loop but by a much larger offset (e.g., by 288 instead of by 1 or 2). We have parameterized the ordering of these two optimizations to study their interactions.

3.4 Memory Prefetch

The last optimization that we performed on the kernels is memory prefetch, which loads data ahead of time so that the latency of memory operations can be hidden when possible. Memory prefetch can reduce the cost of memory accesses only if the memory bandwidth is not already saturated. It is therefore necessary to selectively prefetch only those data that will be used in the near future just enough time ahead.

The POET library allows us to insert prefetch instructions in a variety of locations. We have thus parameterized each prefetch optimization with two decisions: where to insert prefetch instructions, and what data to prefetch. For example, for *gemm*, prefetch is parameterized with three configurations: prefetch at each iteration of loop I the next cache block of matrix A; prefetch at each iteration of loop J the vector of matrix B accessed in the next iteration of loop J; prefetch at each iteration of loop J the vector of matrix C accessed in the next iteration of loop J. Different prefetch configurations can also be combined to accomplish the best effect.

4 A Parameterized Search Engine for Automatic Tuning

We designed and implemented a parameterized search engine (PSEAT) to navigate optimization search spaces with greater flexibility and efficiency. In most existing autotuning systems the search module is tightly coupled with the transformation engine. PSEAT is designed to work as an independent search engine and provides a search API that can be used by other autotuning frameworks. This section discusses design features of PSEAT and its integration with POET.

100	# maximum number of program evaluations
3	# number of dimensions in the search space
R 1 16	# range : 1 .. 16
P 4	# permutation : sequence length 4
E 2 8 16	# enumerated : two possible value 8 and 16

Fig. 4. Example configuration file for PSEAT

Input Input to PSEAT is a configuration file that describes the search space of optimization parameters. Fig. 4 shows an example configuration file. The syntax for describing a search space is fairly simple. Each line in the configuration file describes one search dimension. A dimension can be one of three types: *range* (R), *permutation* (P) or *enumerated* (E). *range* is used to specify numeric transformation parameters such as tile sizes and unroll factors. *permutation* specifies a transformation sequence and is useful when searching for the best phase-ordering. An *enumerated* type is a special case of the *range* type. It can be used to describe a dimension where only a subset of points are feasible within a given range. An example of an *enumerated* type is the prefetch distance in software prefetching. In addition, PSEAT supports inter-dimensional constraints for all three dimension types. For example, if the unroll factor of an inner loop needs to be smaller than the tile size of an outer loop then this constraint is specified using a simple inequality within the configuration file.

Information specific to a search algorithm is specified elsewhere. For example, for simulated annealing the *alpha* and *beta* factors for each dimension is specified in a separate file. The parameters for the search algorithm have been deliberately kept separate to make the search space representation more general. Both the configuration file and the search parameter file can be written by hand or automatically generated by a transformation engine. This feature facilitates the use of PSEAT with model-based search strategies.

	Core 2 Duo	Opteron
Proc.	2.2 GHz Intel Core 2 Duo	2.2 GHz AMD Dual Core
SSE#	16	16
L1	32 KB, 2-way 64 Byte/line	64 KB, 2-way 64 Byte/line
L2	4 MB, 4-way 64 Byte/line	1 MB, 2-way 64 Byte/line
Com- piler	gcc 4.0.1	gcc 4.0.1

(a) Experimental platforms

Parameter	Type	Description	Values
MU	R	unroll factor for loop M	1-M/2
NU	R	unroll factor for loop N	1-N/2
KU	R	unroll factor for loop K	1-K/2
PF	R	memory prefetch	1-5
SSENO	E	number of SSE registers	(0,8,16)
SSELEN	E	length of SSE registers	(8,16)
PERM1	P	ordering of unroll&jam and scalar replacement for A,B/X,C/Y	0-23
PERM2	P	ordering of strength reduction and inner-loop unrolling	0-1

(b) Search spaces

Table 1. Experimental design

Program Evaluation The procedure for program evaluation (*compile-run-measure*) depends on a number of factors and is usually different for different platforms and applications. Hence, incorporating a module for program evaluation makes the search engine less portable. To address this problem, we moved the task of program evaluation away from the search engine and into a set of scripts that are bundled together with PSEAT. These scripts are, to a great degree, self-customizing. They probe a particular machine, *a la* ATLAS, to gather necessary information for program evaluation. The scripts then interpret the output from the search module and deliver the data in the desired format to the tool responsible for applying the transformations. Once the program has been evaluated, a script gathers the performance data and feeds it to the search engine. In integrating PSEAT with POET we used the scripts to transform the search engine output to a command line configuration that POET can recognize. We used ATLAS timers to measure the performance of the kernels and PSEAT scripts to extract the relevant performance data.

Search Algorithm PSEAT implements a number of search strategies including genetic algorithm, direct search, window search, taboo search, simulated annealing and random search. We include *random* in our framework as a benchmark search strategy. A search algorithm is considered effective only if it does better than random on a given search space.

5 Experimental Results

Our previous work has already compared the performance of POET-optimized kernels with those of ATLAS and the Intel `icc` compiler [12]. The goal of this study is to take a more extensive look at the optimization space of dense linear algebra kernels (*dgemm*, *dgemv* and *dger*). We ran experiments on two platforms. The configurations for these two machines are presented in Table 1(a). Each experiment is repeated three times and the mean from these three runs is reported.

Table 1(b) describes the optimization search space for the three kernels. Number of search dimensions is eight for *dgemm* and seven for *dgemv* and *dger* (*dgemv* and *dger* have one fewer loop to unroll than *dgemm*). PERM1 covers the ordering of four transformations: unroll-and-jam and scalar replacement of

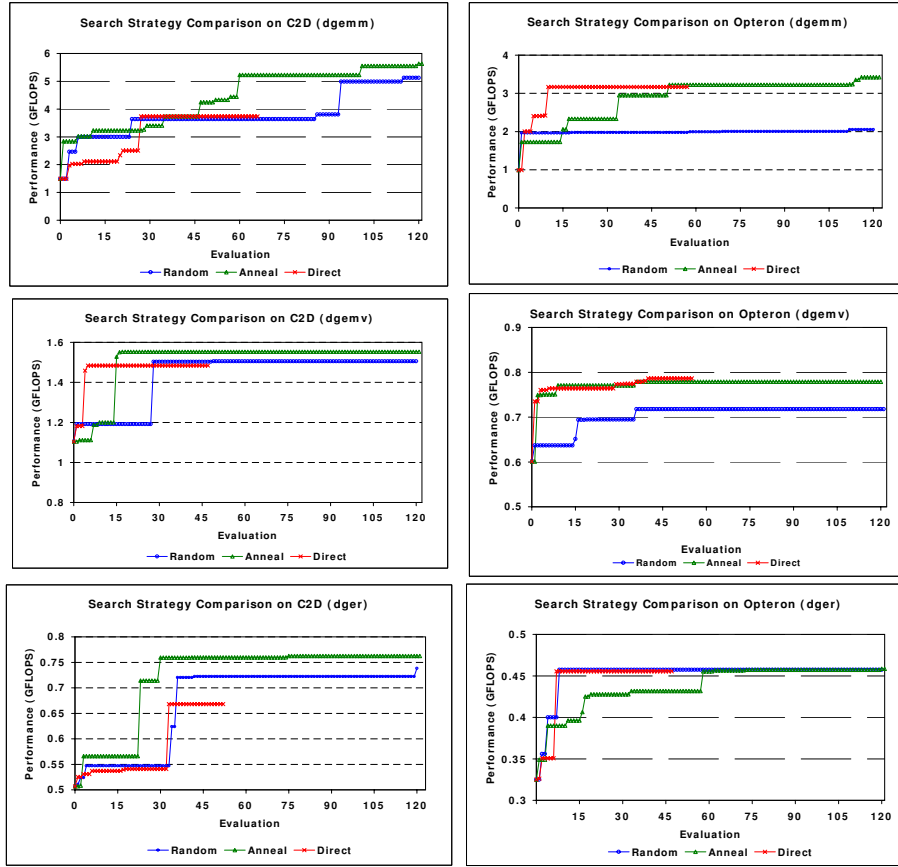


Fig. 5. Search strategy comparison

three different matrices/vectors (A, B, C for *gemm*; A, X, Y for *dgemv/dger*). The range of feasible values for PERM1 is between 0 and 23 ($4! - 1$), which covers all valid permutations of the four transformations. PERM2 covers the ordering of two transformations and hence works as a binary dimension. The values for the unroll factors are between 1 and one-half of the loop iteration numbers. The software prefetching parameter (PF) has five different values that determine which and where array references are prefetched. The search spaces for all three kernels are fairly large even by empirical tuning standards. For example, for a 100x100 matrix the 8-dimensional search space for *dgemm* will have over 120 million feasible points!

5.1 Search Strategy Comparison

We explore the search spaces using three different search strategies: simulated annealing (`anneal`), direct search (`direct`) and random search (`random`). Fig. 5 shows performance of the three strategies on two platforms. Each figure shows the

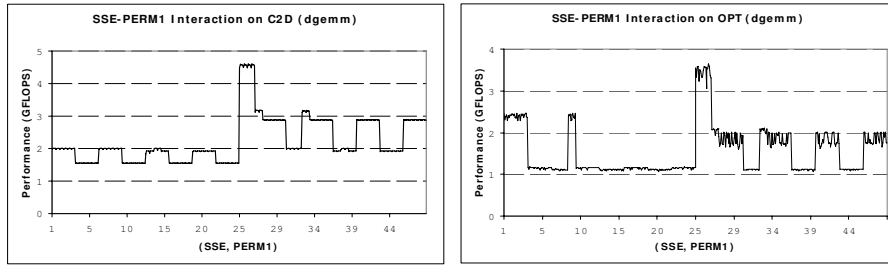


Fig. 6. SSE-PERM1 interaction for *dgemm*

best performance achieved as we progressively increase the number of program evaluations. To make the comparison fair, the initial point is picked randomly and the same initial point is used for all search strategies.

From these figures, **anneal** has a clear advantage over the other two search strategies. It discovers the best value in all but one of the six cases. Because **direct** converges to a local minima much sooner than **anneal**, it often does not discover the best value, but it does discover *good* values more quickly. Given that reducing the number of program evaluations is a key consideration for effective autotuning, **direct** might well be the search strategy of choice for these kernels. The performance of all three strategies also tend to level-off after a certain number of evaluations. This indicates that large regions in the search space may have very little performance variation. Therefore, model-guided pruning can help make the searches more efficient.

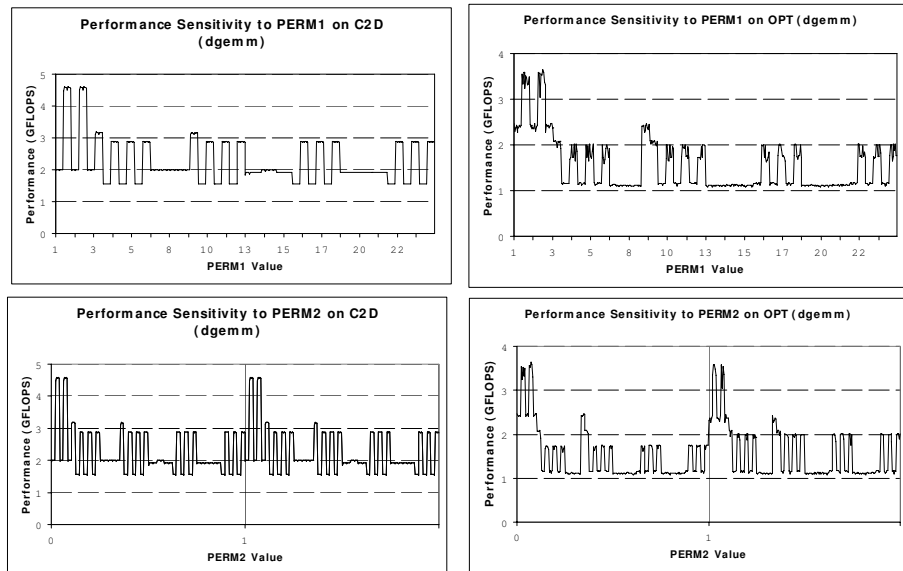


Fig. 7. *dgemm* performance sensitivity

5.2 Search Space Exploration

We performed exhaustive search on various cross-sections of each search space to gain insight into their characteristics. This section summarizes our key findings.

SSE and PERM1: Our experiments reveal a strong interaction between SSE vectorization and the order in which unroll-and-jam and scalar replacement are applied. Fig. 6 shows the effect on *dgemm* performance on the Core2Duo and OPteron as we vary the SSE and PERM1 parameters. Each figure displays the linearized value of SSE and PERM1 along the x-axis (i.e., each x-value corresponds to a pair of values: one for SSE and one for PERM1). There is a clear indication that irrespective of the value of PERM1, vectorization has a positive effect on performance. The best performance on both platforms is achieved when the value along the x-axis is between 25-26, which correspond to the cases where SSE is activated and PERM1 is [SA,UJ,SB,SC] and [SA,UJ,SC,SB], respectively. On these values, the relative ordering of unroll-and-jam and scalar replacement opens up additional opportunities for vectorization. This interplay is somewhat subtle and may not be apparent to a general-purpose compiler that uses only static heuristics to analyze the program. Picking the right values for the two parameters in question can lead to almost a factor of two performance improvement for *dgemm* on the Core2Duo.

PERM1 and PERM2: Fig. 7 shows performance sensitivity as the relative ordering of unroll-and-jam and scalar replacement for different matrices is changed. As expected, the ordering has a significant impact on performance. The best ordering occurs when unroll-and-jam is performed after scalar replacement of references in matrix *A* but before scalar replacement of matrix *C* and matrix *B*. This ordering is different from the one that we manually picked in our previous work [12], where we speculated that applying scalar-replacement of *B* after unroll-and-jam may increase the register pressure too much to create problems on some architectures. However, as it turns out, both these machines are capable of handling the excess register pressure. These results point out the limitations of static analysis and reiterate the need for empirical tuning.

Our experiments also show that the ordering of strength reduction and inner loop unrolling has very little impact on performance. On both platforms we observe very similar performance for both orderings of transformations.

6 Conclusions

This paper examines the optimization space of three linear algebra kernels by combining a pair of independent program transformation and empirical search engine. Our system is flexible and portable and is a small step towards better integration of existing autotuning systems. Our exploration of the search space show significant interaction among several transformations. In particular, the interaction between SSE vectorization and the ordering of unroll-and-jam and scalar replacement had not been revealed in any of the previous studies. The results of this study can be utilized by general-purpose compilers to orchestrate the set of transformations discussed in this paper to achieve improved performance.

References

1. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization, 2006. (CGO 2006)*., New York, NY, 2006.
2. G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Ciorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of *ab initio* quantum chemistry models. *Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.
3. P. Bientinesi, J. A. Gunnels, M. E. Myers, E. Quintana-Orti, and R. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.
4. C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO*, San Jose, CA, USA, March 2005.
5. J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.
6. M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.
7. M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.
8. A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing*, 36(2):183–196, 2006.
9. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, San Jose, CA, USA, March 2005.
10. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
11. R. C. Whaley and D. B. Whalley. Tuning high performance kernels through empirical compilation. In *The 2005 International Conference on Parallel Processing*, June 2005.
12. Q. Yi and C. Whaley. Automated transformation for performance-critical kernels. In *ACM SIGPLAN Symposium on Library-Centric Software Design*, Montreal, Canada, Oct. 2007.
13. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
14. Y. Zhao, Q. Yi, K. Kennedy, D. Quinlan, and R. Vuduc. Parameterizing loop fusion for automated empirical tuning. Technical Report UCRL-TR-217808, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, December 2005.