Transformation Recipes for Code Generation and Auto-Tuning

Mary Hall*, Jacqueline Chame†, Chun Chen*, Jaewook Shin+, and Gabe Rudy*

*School of Computing, University of Utah; Salt Lake City, UT
†USC/Information Sciences Institute; Marina del Rey CA
+Argonne National Laboratory; Argonne, IL

Abstract. In this paper, we describe transformation recipes, which provide a high-level interface to the code transformation and code generation capability of a compiler. These recipes can be generated by compiler decision algorithms and savvy software developers. This interface is part of an auto-tuning framework that explores a set of different implementations of the same computation and automatically selects the implementation that best meets a set of optimization criteria. Along with the original computation, a transformation recipe specifies a range of implementations of the computation resulting from composing a set of high-level code transformations. In our system, an underlying polyhedral framework coupled with transformation algorithms takes this set of transformations, composes them and automatically generates correct code. We first describe an abstract interface for transformation recipes, which we propose to facilitate interoperability with other transformation frameworks. We then focus on the specific transformation recipe interface used in our compiler and present performance results on its application to kernel and library tuning and tuning of key computations in high-end applications. We also show how this framework can be used to generate and auto-tune parallel OpenMP or CUDA code from a high-level specification.

1 Introduction

The compiler research community has developed a significant body of work in code transformation techniques that improve performance by optimizing for specific architectural features, especially by increasing parallelism or better managing the memory hierarchy [46, 44, 45, 38, 39, 8, 28, 23, 37, 33, 5, 25, 26, 18]. However, in our work with application and library developers that are seeking very high levels of performance, we find that their interface to the commercial compiler, mostly through compile-time flags, is a stumbling block to using the code transformation capability. Many such programmers continue to manually apply the very same code transformations that their compiler can apply automatically, not only increasing programming time but also producing low-level architecture-specific code that is difficult to port and maintain.

A well-recognized challenge to the effectiveness of compiler optimizations targeting architectural features is making complex tradeoffs between different optimizations, or identifying optimal values of optimization parameters such as unroll factors or loop tile sizes. Without sufficient knowledge of the execution environment, which is extremely difficult to model statically, compilers often make suboptimal choices, sometimes even degrading performance. To address this limitation, a recent body of work on *auto-tuning* uses empirical techniques to execute code segments in representative execution environments to determine the best-performing optimization sequence and parameter values [21, 22, 43, 27, 35, 31, 32, 36, 29]. A compiler to support auto-tuning must have a different structure than a traditional compiler, as it must expose a set of different variants of a computation, with parameterized optimization variables. These parameterized variants are then evaluated empirically by an experiments engine that identifies the best implementation of the computation. In our compiler, we also must support *collaborative auto-tuning*, so that application developers can access and guide the auto-tuning framework.

As a separate concern, some but not all of the compiler community's transformation research has migrated into practice in widely-used or commercial compilers. There are some notable success stories, such as the Graphite project for Gnu compilers [30], and specific optimizations such as software pipelining and loop tiling, but it is difficult to integrate compiler decision algorithms into commercial compilers without a complete reimplementation. We make the observation that the structure of today's compilers makes it difficult to migrate new ideas into practice; retargeting optimizations and decision algorithms for a new compiler infrastructure is often simply infeasible. Every commonly-used compiler infrastructure has strengths and weaknesses as well as years of development that are costly to repeat; converging on one or a small subset of compiler infrastructures is therefore unrealistic. As an alternative viewpoint, could we possibly compose a compiler tool from the best capabilities of a collection of systems?

The focal point of this paper is a high-level interface for describing *transformation recipes*, as this interface is the mechanism by which the compiler organization meets the three driving principles: supporting auto-tuning, serving as an application-developer interface (in addition to a compiler interface), and providing a common interface for interoperability between compilers. Although other compiler interfaces exist with overlapping goals, to be discussed in Section 2, our approach is unique in trying to bring together all of these elements.

Specification of Parameterized Variants for Auto-tuning Environment. Transformation recipes describe a range of implementations, parameterized by certain optimization variables. This range of implementations can be evaluated by an external experiments engine that efficiently explores the resulting optimization search space.

Application and Library Developer Interface. Transformation recipes allow application and library developers to interact directly with the compiler to transform their code, including parallelization. Through the system organization, the compiler manages the details of carrying out transformations correctly, and generates code for a range of implementations that can be compared automatically using auto-tuning technology.

Common API for Compiler Transformation Framework. Transformation recipes can also serve as a common interface for different compiler transformation frameworks. Numerous compiler transformation frameworks are capable of specific optimizations such as loop unrolling or tiling, so with the appropriate interface, the same recipe could be used by multiple compilers and their results compared. The same application could be tuned using different compiler transformation frameworks, either successively or on independent pieces of code.

In our own research, the transformation recipe interface is part of a working compiler system that uses an underlying polyhedral transformation and code generation framework to support robust code generation within its domain of applicability. This system has been used in a variety of ways: for kernel tuning [11, 9, 40], for library tuning and generation, for tuning of key computations from scientific applications, and for guiding parallel code generation for OpenMP and CUDA.

In the remainder of the paper, we describe our own working compiler framework and the transformation recipes it supports, as well as a current broad activity across a number of compiler and auto-tuning research groups to develop an infrastructure-independent *common transformation API* [1]. For the latter, we would like to engage the LCPC community to participate so that this representation can potentially interoperate with a large number of compiler infrastructures, thus moving our entire community in the direction of repeatable experimental research and easier adoption of new ideas. Therefore, in addition to presenting the design of our own compiler, we intend for this paper to initiate dialog within our community on what is the most appropriate compiler interface for supporting interoperability among compilers and user interaction for these and related transformations.

2 Related Work

Commercial and research compilers have for a long time permitted users to provide pragmas to augment the compiler's knowledge, dating back to the vectorization era and possibly beforehand. Most notably, the existence of a similar set of pragmas across different vendor compilers and a desire to standardize led to the design of OpenMP. Existing such pragmas operate at a higher level that presented here, and more often focus on augmenting compiler analysis rather than directing transformation; *e.g.,* the pragma C$IVDEP told Fortran vectorizers that the dependences it found in a loop could be ignored. Today's compilers use pragmas to describe simple code transformations like loop unrolling, but do not support the composition of several transformations.

Polyhedral loop transformation frameworks (*e.g.,* [20, 15]) are known to support composition of multiple transformations. Internally, these frameworks manipulate mathematical representations of iteration spaces and loop bounds, and expose interfaces that allow users (or compilers) to manipulate these low-level mathematical representations or individual loop or statement manipulations. Such interfaces are still too cumbersome to use when implementing a complex optimization strategy since descriptions of transformation sequences tend to be lengthy. Our own compiler uses a polyhedral transformation framework, but the transformation recipes specify high-level transformations that operate on a complete loop nest; transformation algorithms translate from the recipes to the iteration space manipulations for all statements enclosed in the loop nest [9, 10]. As compared to other polyhedral frameworks, the transformation recipes described in this paper target a higher level description of a composition of transformations that is suitable for savvy application developers in addition to compiler developers; further, the interface can be broadly applicable to non-polyhedral frameworks and polyhedral frameworks alike.

A number of interfaces to code transformation exist that are targeting a similar level to the transformation recipes presented here. These include pragma-oriented transformation specifications such as LoopTool [34], X language [13], and Orio [16]. A related tool POET uses an XML-based description of code transformation behavior to produce portable code transformation implementations [48], These tools all provide a general and flexible way to express a set of transformations on a specific code fragment and several of these generate a set of alternative implementations to support auto-tuning. Looking across the tools, the set of supported transformations is different, and ours is neither a subset or superset of other systems. Some distinguishing characteristics of our supported transformations include the OpenMP and CUDA parallel code generation, specialization and index set splitting. Further, these tools support a core set of transformations such as loop unrolling and loop tiling, and some support extension to add new transformations. In summary, each tool has its unique strengths and most suitable applications. Thus, we expect that the existence of other interfaces at this level gives promise to potential for interoperability between tools.

Our transformation recipe is separate from the code, which has several benefits: (1) the code can remain architecture independent; (2) it is a concise description of how to transform a code segment that can be saved with a program and used as documentation for how the code was transformed for a particular execution context; (3) implementations for different architectures can use separate scripts, rather than requiring multiple sets of pragmas or multiple copies of the code; and, (4) there is greater potential for reuse of scripts across different pieces of similarly-structured code. It is also mechanical to translate between recipes and pragmas in the code to support a pragma-based tool, and with appropriate naming, also to translate from a set of pragmas to a recipe.

3 Rethinking the Developer and Compiler Workflows

This paper focuses on transformation recipes because they highlight requirements for: (1) programmer interaction; (2) optimization and code generation; (3) compiler development; and, (4) tool interoperability. This section describes how transformation recipes are used by application/library developers as well as compiler developers to express and document an optimization recipe. Such recipes can be later fine-tuned by other developers or modified for different architectures, while the original machine-independent code is maintained with the program. As has been proven time and again, manually modifying a piece of a highly-optimized code leads to machine-dependent code that is difficult to maintain and port. Additionally, using such a system, a compiler developer can now focus on the performance aspect of transformations on the target code, instead of the nitty-gritty details of implementation and how to generate correct code with minimal overheads.

3.1 Application and Library Developer Workflow

Application and library developers spend a significant amount of time in manual performance tuning. Many examples in the literature describe tuning using widely available loop transformations and auto-tuning to find the right optimization parameter values, for example, a computational fluid dynamics code NEK5000 [41], quantum chemistry codes [19], Cholesky factorization of a sparse matrix [17], and a sparse linear solver from LS-DYNA [24]. The tuning process typically consists of several steps: identifying performance issues, deriving optimization strategies, applying optimizations (code transformations), and evaluating the performance of the optimized code. This process is repeated several times until performance is satisfactory or no more profitable optimizations can be applied. Self-tuning libraries such as ATLAS and FFTW automate the code generation step and the search for the best performing code version. Application developers, however, still tend to apply optimizations by hand, an error-prone and time-consuming process.

As a contemporary example of this process, in [47], Wolfe writes about tuning a simple single-precision matrix multiplication kernel on a NVIDIA GeForce GTX 280. Wolfe presents several versions of the matmul code obtained by using code transformations such as loop permutation, loop strip mine and loop unroll, and caching data in local memory. Performance ranges from 1.7 to 208 GFLOPs depending on the number of threads per block (strip size), the loop(s) unrolled and unroll sizes, and the amount of data cached in local memory. Summarizing the article, Wolfe writes “Matmul is just one simple example here, three loops, three matrices, lots of parallelism, and yet I put in several days of work to get this seven line loop optimized for GPU.”

All seven versions of matmul in [47] can be derived with a combination of loop permutation, strip mine and unroll, and data copy optimization. Each combination can be expressed as a transformation recipe. The ability to express a composition of transformations as a transformation recipe allows users to specify complex optimization strategies and experiment with a large number of code versions and optimization parameter values.

Figure 1 illustrates how application or library developers may interact with tools to automate the code transformation and code generation steps.

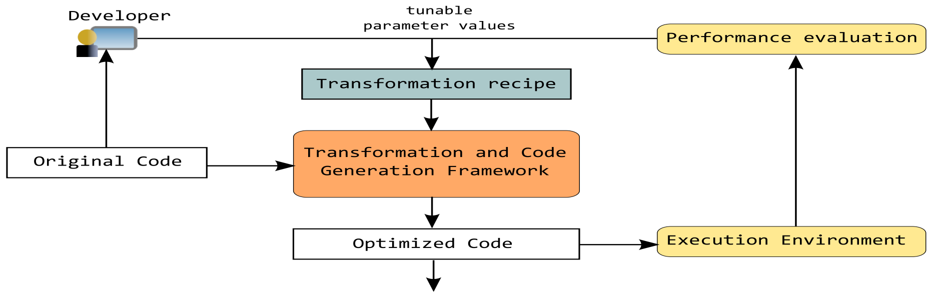

Fig. 1. Application and library developer workflow.

3.2 Compiler Developer Workflow

Traditional compilers often rely on a rigid transformation strategy that applies a fixed sequence of optimizations. [12, 6] have shown that by searching for optimization sequences compilers can achieve 15 to 25% better performance than human-designed fixed-sequence originally used in these compilers. Various performance libraries such as ATLAS [42] and FFTW [14] use custom optimization strategies to achieve significant performance gain over leading-edge compilers.

However, a key challenge to loop nest optimization is that it is difficult to express and compose a sequence of loop transformations. Transformation recipes provide a mechanism to describe an optimization strategy that can be used by compiler decision algorithms. Moreover, auto-tuning compilers can propose multiple code variants representing different optimization strategies that can be compared empirically. In [11], a compiler decision algorithm generates sequences of loop transformations based on data reuse available in applications. Such a transformation sequence is not fixed during the compiler design but is computation dependent, and generates different variants if static analysis cannot determine which optimization strategy is better.

As shown in Figure 2, the first step is to derive alternative code variants, where a decision algorithm such as [28, 44, 11] can be used to derive a sequence of code transformations that corresponds to a transformed version of the original code. This sequence of transformations is represented as a transformation recipe. Multiple code variants can be generated if necessary. Optionally, each transformation recipe can contain unbound parameters. In this case, a search engine can be used to find the best parameter values that are not determined statically by the decision algorithm. In the next step, the transformation recipe combined with bound parameters are given to a transformation and code generation framework to generate the transformed code.

This compiler organization enables compiler designers to prototype optimization algorithms in a convenient way. We envision that compiler designers can experiment with different ideas and check the performance results with the help of the transformation recipe and an underlying auto-tuning environment.

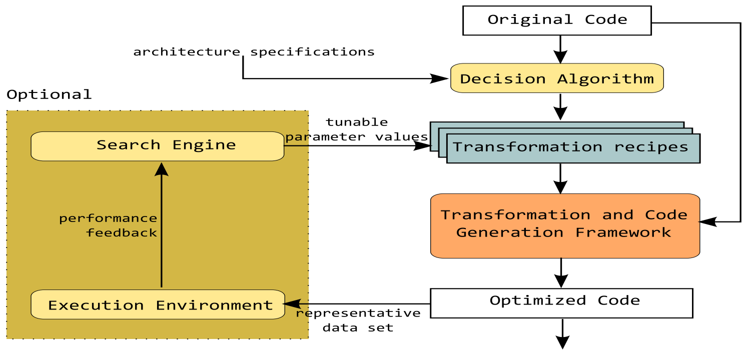

Fig. 2. Compiler Developer Workflow

4 Transformation Recipe Description

In this section we discuss the requirements of a high-level interface representing transformation recipes. We then present a set of transformations and their input and output parameters. Finally we illustrate the use of transformation recipes with a simple code example, a recipe and the corresponding output (transformed) code generated by our compiler system.

The key requirements of a high-level interface for transformation recipes in an auto-tuning environment are:

– a level of abstraction suitable to both compilers and application developers;
– a mechanism for naming language constructs;
– a mechanism to define transformations, such as *transformation rules* from the X language [13];
– support for empirical search with tunable optimization parameters.

High-level interface. A transformation recipe expresses an optimization strategy as a sequence of composable transformations. Each transformation has a set of input and output parameters. Input parameters are: (1) language constructs such as statements, loops and array references that are present in the code before the transformation is applied; (2) optimization parameters such as tile size and unroll factors, which can be integer values or unbound parameters to be instantiated during empirical search. Output parameters are specifiers to new language constructs resulting from the code transformation. Output parameters allow subsequent transformations in a transformation recipe to refer to these new constructs.

Representation of language constructs. A common interface to transformation tools or compilers requires a mechanism for naming language constructs such as loops, statements, and array references. Under the DOE SciDAC PERI project, Quinlan and Liao [2] have designed a representation based on abstract handles to language constructs, to support interoperability between compiler and tools. Abstract handles represent references to language constructs in source code and optimization phases, and allow language constructs to be specified in multiple formats, including:

– Source file position information including path, filename, line and column number as in GNU standard source position [3].
– Global or local numbering of specified language construct in source file.
– Global or local names of constructs.
– Language-specific naming mechanisms.

Interoperability is achieved by writing out abstract handles as strings and reading them with other tools to build the equivalent abstract handle. For the discussion in this paper we assume that language constructs are identified using the abstract handle mechanism in [2]. However, for simplicity of presentation, we use labels to identify loops and statements throughout the paper.

Transformations. The set of transformations listed in this section was collected from transformations frequently used for optimizing kernels and key computations of scientific applications as part of our auto-tuning work, and transformations supported by two state-of-the-art transformation tools, CHiLL [9, 10] and POET [48]. We do not include all transformations supported by these frameworks, nor all transformations in the literature. The transformation recipe interface is an open project, and new transformations can be added as needed. Table 4 shows a set of transformations and their input and output parameters. The fourth column provides a short description of each transformation.

Transformation	Input Params	Output Params	Description
permute	`l1` ... `ln`		permute loops to new order `l1` ... `ln`
stripmine	`l`, `ssz`	`sl`	strip mine `l` by `ssz`
tile	`l`, `tsz`	`tl`	tile `l` by `tsz`
unroll	`l`, `usz`		unroll `l` by `usz`
unroll_&_jam	`l`, `usz`		unroll `l` and recursively fuse inner loops
fuse	`l1`, `l2`	`fl`	fuse `l1` and `l2`
distribute	`l`, `stmt_list`	`dl`	distribute statements in list into new loop `dl`
split	`l`, `cond_exp`	`sl`	split index set of `l` given conditional expression
shift	`l`, `offset`		shift induction variable of `l` by `offset`
scale	`l`, `f`		scale induction variable of `l` by factor `f`
data_copy	`a`, `l`, `ta`		copy data referenced by `a` within `l` to `ta` and replace references to `a` within `l` with references to `ta`
scalar_repl	`a`, `l`, `ts`		copy data referenced by `a` to scalar `s` and replace references `a` within `l` with references to `s`
stmt_split	`stmt`, `v`	`stmt_list`	split operations on variable `v` into separate statements

Table 1. Transformations.

The transformations in Table 4 are included in the common transformation interface being developed by the PERI auto-tuning team. This common interface will allow developers to use the same transformation recipe with PERI compilers and tools, including CHiLL and POET. POET supports all transformations listed, and CHiLL supports most transformations (exceptions are scalar replacement and statement spliting). CHiLL also supports specialization and directives for OpenMP and CUDA code generation (to be discussed next).

5 Example Applications of this Tool

This compiler framework has been applied to the challenge of tuning a variety of code examples. We have used transformation recipes extensively to tune performance of computational kernels, generate libraries, tune applications, and generate optimized parallel (OpenMP and CUDA) code. This section presents highlights of this work.

5.1 Kernel Tuning

We use LU factorization to demonstrate a very complex sequence of transformations that can be expressed using transformation recipes. From data reuse analysis, a compiler decision algorithm [9] can derive a transformation sequence very similar to the optimization strategy in LAPACK [7]. Previously such a strategy could only be generated by a domain expert and no compiler-optimized code could approach the performance of manually-tuned versions.

Figure 3 shows the original LU code and a transformation recipe [9, 10]. First, loop `J` is tiled and the loops inside the tile controlling loop can be split. This results in separating an imperfect loop nest (a mini-LU) from a perfect loop nest with the heuristic that a perfectly nested loop renders more optimization opportunities. Next, the perfect loop nest is split so that one loop nest has non-overlapping array accesses (a GEMM kernel) and the other has overlapping array accesses (a TRSM kernel). The goal is to expose more optimization opportunities in a perfect loop nest with non-overlapping data accesses. Then, each sub loop nest can be independently analyzed and further transformed according to its data reuse and data dependence patterns. Figure 3(c) shows the code generated from the transformation recipe in (b) with the best parameter values found on the Intel Pentium M. Figure 4 shows the performance of the final code, which achieves an average speedup of 16.29x over the native compiler.

This example shows how transformation recipes can facilitate automatic generation of highly optimized code. In [11], we combine a compiler locality optimization algorithm for multi-level memory hierarchies with empirical tuning. The models and compiler heuristics limit the search space to a few candidate transformation recipes and a small range of parameter values. An empirical search can then use heuristics and constraints on parameter values to find the best implementation, searching just a few dozen points. The results for Matrix Multiply and Jacobi Relaxation on two architectures, MIPS R10K and Sun UltraSparc II, shows substantial performance improvements over native compilers and performance comparable and sometimes better than manually-tuned code.

Moreover, transformation recipes allow us to leverage other tools to work with compilers. In [40], Active Harmony, an auto-tuning tool supporting parallel search of optimization parameters, is integrated with CHiLL to generate alternative implementations of computational kernels and automatically select the best-performing implementation. Performance results on three kernels, Matrix Multiply, Triangular Solver and Jacobi Relaxation, are 1.4 to 3.6 times faster than the native Intel compiler.

```
DO K=1,N-1
 DO I=K+1,N
  A(I,K)=A(I,K)/A(K,K)
 DO I=K+1,N
  DO J=K+1,N
   A(I,J)=A(I,J)-
          A(I,K)*A(K,J)
```

(a) Original code

permute([1,2,3])
tile(1,3,TJ,1)
split(1,2,[L2≤L1-2])
permute(2,[1,2,4,3])
permute(1,[1,3,4,2])
split(1,2,[L2≥L1-1])
tile(3,2,TI_1,3)
split(3,3,[L5≤L2-1])
tile(3,5,TK_1,5)
tile(3,5,TJ_1,7)
datacopy(3,4,[1],1)
datacopy(3,5,[2])
unroll(3,5,UI_1)
unroll(3,7,UJ_1)
datacopy(4,3,[1],1)
tile(1,4,TK_2,3)
tile(1,3,TI_2,5)
tile(1,5,TJ_2,7)
datacopy(1,4,[1],1)
datacopy(1,5,[2])
unroll(1,5,UI_2)
unroll(1,6,UJ_2)

(b) Recipe

```
DO T2=2,N,64
 DO T4=2,T2-64,256
  DO T6=1,T4-1,256
   DO T8=T6,MIN(T4-1,T6+255)
    DO T10=T4,MIN(T2-2,T4+255)
     P1(T8-T6+1,T10-T4+1)=A(T10,T8)
   DO T8=T2,MIN(T2+56,N),8
    DO T10=T8,MIN(N,T8+7)
     DO T12=T6,MIN(T6+255,T4-1)
      P2(T12-T6+1,T10-T8+1)=A(T12,T10)
    DO T10=T4,MIN(T2-2,T4+255)
     DO T12=T8,MIN(N,T8+7)
      DO T14=T6,MIN(T6+255,T4-1)
       A(T10,T12)=A(T10,T12)-P1(T14-T6+1,T10-T4+1)*P2(T14-T6+1,T12-T8+1)
  DO T6=T4,MIN(T4+254,T2-3)
   DO T8=T6+1,MIN(T4+255,T2-2)
    P3(T6-T4+1,T8-(T4+1)+1)=A(T8,T6)
  DO T6=T4+1,MIN(T4+255,T2-2)
   DO T8=T2,MIN(N,T2+63)
    DO T10=T4,T6-1
     A(T6,T8)=A(T6,T8)-P3(T10-T4+1,T6-T4+2)*A(T10,T8)
 DO T4=1,T2-65,256
  DO T6=T2-1,N,256
   DO T8=T4,MIN(T4+255,T2-2)
    DO T10=T6,MIN(T6+255,N)
     P4(T8-T4+1,T10-T6+1)=A(T10,T8)
   DO T8=T2,MIN(T2+56,N),8
    DO T10=T8,MIN(N,T8+7)
     DO T12=T4,MIN(T4+255,T2-2)
      P5(T12-T4+1,T10-T8+1)=A(T12,T10)
    DO T10=T6,MIN(T6+255,N)
     DO T12=T8,MIN(N,T8+7)
      DO T14=T4,MIN(T2-2,T4+255)
       A(T10,T12)=A(T10,T12)-P4(T14-T4+1,T10-T6+1)*P5(T14-T4+1,T12-T8+1)
 DO T4=T2-1,MIN(N-1,T2+62)
  DO T8=T4+1,N
   A(T8,T4)=A(T8,T4)/A(T4,T4)
  DO T6=T4+1,MIN(N,T2+63)
   DO T8=T4+1,N
    A(T8,T6)=A(T8,T6)-A(T8,T4)*A(T4,T6)
```

(c) Generated code from (b) with bound parameters

Fig. 3. LU Factorization code and transformation recipe

5.2 Library and Application Tuning

We have used transformation recipes to generate a library of optimized routines for Nek5000 [4], where the core computation is small, highly rectangular matrix-matrix multiply. While vendor BLAS routines achieve very high performance for large square matrices, their optimization strategies are not suitable for small rectangular matrices. Further, an optimized code version for a small matrix size does not always perform the best for other small matrices [41]. To optimize for multiple matrix shapes and sizes, we use *specialization*, where matrix sizes are fixed to the eight most frequent sizes that comprise about 74% of the matrix multiply computation time in Nek5000. We then use CHiLL's transformation recipes to specify constant values for the matrix sizes. Figure 5(a) shows an example recipe used to specialize the small matrix multiply code in (b). For the three loop bounds, constants are specified in the recipe with a *known* command. Specialization allows CHiLL to generate more efficient code with fewer checks

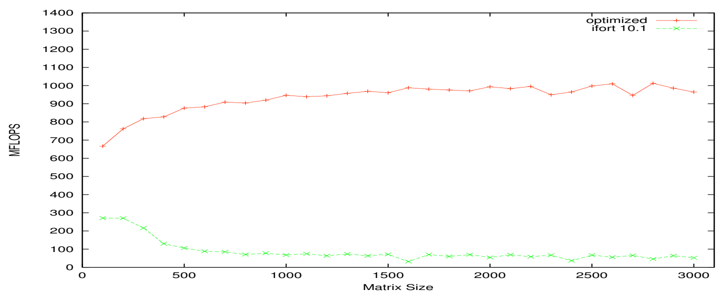

Fig. 4. Performance from Figure 3(d) on Pentium M

and the backend compiler to apply more aggressive optimizations. The eight best performing variants selected for each of the eight matrix sizes have different transformation parameter values. Finally, a wrapper provides the same interface as the default function for NEK5000, and is combined with the eight optimized matrix multiply routines into a library. Upon a call, the wrapper checks the three input values and calls an optimized code version if there is a match. The library is more than 2.3 times faster than the default code in Nek50000 for the eight most frequent sizes, and up to 2.1X times faster than the hand-coded Goto BLAS linear algebra library.

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

(a) loop order i,j,k

```
     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
```

(b) `original.f`

Fig. 5. Example of a CHiLL script and a specialized matrix multiply code.

With collaborators, we also have recent experiences and results applying this framework to other applications SMG2000, and a quantum mechanical code with stencil computations, which we plan to summarize in the final paper.

5.3 Optimizing and Generating Parallel Code

We are using the transformation recipes as a mechanism to generate parallel OpenMP and CUDA code, taking the sequential code and transformation recipe as input. The transformation recipes express the composition of parallelization and a set of code transformations for these two different explicitly parallel programming models. Thus, they can be used to support the CUDA code generation examples from Section 3 and similar OpenMP examples. We add two constructs to the transformation interface to support this parallel code generation.

Transformation	Input Params	Output Params	Description
OMPize	`l`, `sched`, `chksz`		parallelize `l`, with `sched` and `chksz` as variables
CUDAize	`l`, TI, TJ	`kernnm`	parallelize `l` and create CUDA kernel `kernnm`

Table 2. Recipe entries for parallel code generation.

For OMPize, the `sched` parameter allows the script to customize the OpenMP behavior for scheduling the distribution of loop iterations over threads (*runtime, static or dynamic*), while `chksz` specifies the scheduling unit of loop iterations to threads. The result of the code generation is insertion of the appropriate OpenMP pragmas in either C or Fortran syntax, as requested by the user. This set of parameters, along with the previously-described loop transformations, allows the application developer or compiler to trade off the impact of locality, multimedia extensions, parallelism granularities, and scheduling strategies.

The CUDAize transform parses out two loop levels and performs a tiling of loop dimensions to a width of TI and TJ. The remaining inner procedure is refactored into a GPU kernel function with the name `kernnm`. The proper scaffolding code is then substituted in place of the original loop to properly set up, copy input, make the kernel call and copy out results. This set of parameters assists with the challenging problems of decomposition of parallelism into blocks and threads, appropriate indexing given a specific decomposition and managing a heterogeneous memory hierarchy including explicit copying. Different solutions impact locality in registers within a thread, register usage, coalescing of global memory accesses, and amount of parallelism needed to hide memory latency.

6 Summary and Conclusion

This paper focuses on transformation recipes, a high-level description of code transformations that serve as an interface to describe the composition of complex code transformations. Within the context of our own work on compiler-based auto-tuning of kernels, libraries and key computations from high-end applications, this interface and the CHiLL transformation framework that supports it have proven very useful at obtaining high levels of performance, sometimes comparable or even better than manually-tuned code. We discuss how this interface is designed for both compiler developers and application/library developers. This interface relates to many other tools developed by researchers in the LCPC community. We feel the time is right to converge on a common interface for these transformations that would support interoperability across compiler frameworks. The specific interface described here is just a starting point for these discussions.

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://www.peri-scidac.org/wiki/index.php/Main_Page`.
2. `http://rosecompiler.org/`.
3. `http://www.gnu.org/prep/standards/html_node/Errors.html`.
4. http://nek5000.mcs.anl.gov/index.php/Main_Page.
5. Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, May 2000.
6. L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, June 2004.
7. Ed Anderson, Danny Sorensen, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, Alan McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, and Christian H. Bischof. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing '90*, November 1990.
8. Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, November 1994.
9. Chun Chen. *Model-Guided Empirical Optimization for Memory Hierarchy*. PhD thesis, University of Southern California, May 2007.
10. Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, June 2008.
11. Chun Chen, Jacqueline Chame, and Mary W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2005.
12. Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, August 2002.
13. Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David Padua, and Keshav Pingali. A language for the compact representation of multiple program versions. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.
14. Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):216–231, February 2005.
15. Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.
16. Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, May 2009.
17. José R. Herrero and Juan J. Navarro. Improving performance of hypermatrix cholesky factorization. In *9th International Euro-Par Conference*, pages 461–469, 2003.

18. Marta Jiménez, José M. Llabería, and Agustín Fernández. Register tiling in nonrectangular iteration spaces. *ACM Transactions on Programming Languages and Systems*, 24(4):409–453, July 2002.
19. 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*, pages 120–130. Springer, 2008.
20. Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Department of Computer Science, University of Maryland, 1993.
21. Toru Kisuki, Peter M. W. Knijnenburg, and Michael F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 2000.
22. Peter M. W. Knijnenburg, Toru Kisuki, Kyle Gallivan, and Michael F. P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurrency and Computation: Practice and Experience*, 16(2–3):247–270, March 2004.
23. Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multilevel blocking. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.
24. Y. Lee, P. Diniz, M. Hall, and R. Lucas. Empirical optimization for a sparse linear solver: A case study. *International Journal of Parallel Programming*, 33, 2005.
25. Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitioning. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, January 1997.
26. Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2001.
27. Qingda Lu, Sriram Krishnamoorthy, and P. Sadaypppan. Combining analytical and empirical approaches in tuning matrix transposition. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2006.
28. Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
29. Boyana Norris, Albert Hartono, Elizabeth Jessup, and Jeremy Siek. Generating empirically optimized composed matrix kernels from matlab prototypes. In *Proceedings of the International Conference on Computational Science*, May 2009.
30. Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. GRAPHITE: Polyhedral analyses and optimizations for GCC. In *Proceedings of the 4th GCC Developers' Summit*, June 2006.
31. Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2007.
32. Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part II, multi-dimensional time. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2008.

33. William Pugh and Evan Rosser. Iteration space slicing for locality. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, August 1999.
34. Apan Qasem, Guohua Jin, and John Mellor-Crummey. Improving performance with integrated program transformations. Technical Report TR03-419, Rice University, October 2003.
35. Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 2006 ACM International Conference on Supercomputing*, June 2006.
36. Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. A tuning framework for software-managed memory hierarchies. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 2008.
37. Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.
38. Vivek Sarkar and Radhika Thekkath. A general framework for iteration-reordering loop transformations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1992.
39. Olivier Temam, Elana D. Granston, and William Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, November 1993.
40. Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffery K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium*, April 2009.
41. Henry M. Tufo and Paul F. Fischer. Terascale spectral element algorithms and implementations. In *ACM/IEEE conference on Supercomputing*, Portland, OR, 1999.
42. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, January 2001.
43. R. Clint Whaley and David B. Whaley. Tuning high performance kernels through empirical compilation. In *Proceedings of the 34 International Conference on Parallel Processing*, June 2005.
44. Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1991.
45. Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.
46. Michael Wolfe. Data dependence and program restructuring. *The Journal of Supercomputing*, 4(4):321–344, January 1991.
47. Michael Wolfe. Compilers and more: Optimizing gpu kernels. `http://www.hpcwire.com/features/Compilers_and_More_Optimizing_GPU_Kernels.html`, October 2008.
48. Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. POET: parameterized optimizations for empirical tuning. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, March 2007.