Performance Search Engine Driven by Prior Knowledge of Optimization

Youngsung Kim
University of Colorado, Boulder and National Center for Atmospheric Research, USA
youngsung@ucar.edu

Pavol Černý
University of Colorado, Boulder, USA
pavol.cerny@colorado.edu

John Dennis
National Center for Atmospheric Research, USA
dennis@ucar.edu

Abstract

For scientific array-based programs, optimization for a particular target platform is a hard problem. There are many optimization techniques such as (semantics-preserving) source code transformations, compiler directives, environment variables, and compiler flags that influence performance. Moreover, the performance impact of (combinations of) these factors is unpredictable. This paper focuses on providing a platform for automatically searching through *search space* consisting of such optimization techniques. We provide (i) a search-space description language, which enables the user to describe optimization options to be used; (ii) search engine that enables testing the performance impact of optimization options by executing optimized programs and checking their results; and (iii) an interface for implementing various search algorithms. We evaluate our platform by using two simple search algorithms - a random search and a casetree search that heuristically learns from the already examined parts of the search space. We show that such algorithms are easily implementable in our platform, and we empirically find that the framework can be used to find useful optimized algorithms.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]; D.3.4 [*Programming Language*]: Optimization; G.1.0 [*Numerical Analysis*]: Numerical algorithms

Keywords Performance, automated optimization, code generation, scientific computing

1. Introduction

Automating the task of performance optimization has been widely recognized as a difficult task since the 1990s [11]. Most of automatic optimizations are done by compilers. However, compiler, as a general-purpose tool, has to generate correct result within limited time. Therefore, in general, it cannot search through and test results of different optimization techniques (such as *semantics-preserving source code transformations*) for the target platform. Hence, the level of optimization is restricted. Furthermore, there is an ever increasing number of available *compiler flags* to support new processors as well as advanced optimization techniques. It is generally a very time-consuming task to find a "right" set of flags that meets user's optimization goal, and furthermore, it is not always clear if the goal is achievable or not.

In order to complement the performance optimization done by compilers for scientific array-based programs, we propose an automated system that searches through (combinations of) optimization techniques and their parameters in order to find the best technique for the given architecture. There are two factors that make automation essential : (a) the *size* of the search space, which grows exponentially with the number of optimizations that can be applied, and (b) the *non-smooth*, even chaotic, nature of performance impact of various optimizations. The latter point is illustrated in [10], which states "[T]he performance dynamics of a program running on a modern computer can be complex and even chaotic."

Automated techniques for semantics-preserving program transformations can be seen as an instance of *program synthesis*. Work by Solar-Lezama, Bodík, and others [16–18] has demonstrated that synthesis can be effectively applied to problems in a wide variety of areas such as concurrent data structures [7, 18, 20, 21] spreadsheet transformations [5, 6], and many others. However, these works focus on correctness, not performance. One approach to program synthesis is superoptimization [15]. Superoptimization is the task of finding the optimal code sequence for a single, loop-free sequence of instructions. The approach in [15] applies classical algorithms (such as Metropolis-Hastings) to find the optimal sequence of instructions to replace the original straight-line code.

The goal of this paper is to *enable superoptimization techniques* for kernels of scientific array-based programs (kernels are small computation-intensive parts of scientific software). Our contributions towards this end are three-fold: First, we provide a *search-space description language (SSDL)*, a domain-specific language that enables the user to describe optimization options to be used. An SSDL description thus captures the prior knowledge of the user about which optimizations are likely to be useful. Second, we develop a *search engine* that asks a search algorithm which combination of optimization techniques to use, and then executes the resulting program, and checks its results. An efficient heuristic of checking the results of the optimized program is possible precisely because we target scientific array-based programs, where there are relatively few control branches that depend on the contents of the array. Third, we provide an *interface* of the search engine that enables different search algorithms to be plugged in with ease. In this way, we enable streamlined development of new superoptimization techniques.

In order to empirically evaluate our search engine, we develop two sample algorithms. The first algorithm chooses, at all times, the next set of optimization techniques to be examined randomly. The second algorithm, which we call casetree, learns from previ-

Array '15, June 13–14, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/2774959.2774963

ously examined cases. We evaluate whether the algorithms produce programs that are (a) correct w.r.t. the original program, (b) have improved performance, and (c) how long does the optimization process take for kernels of typical size. The correctness was simply evaluated by human experts, and it was found that the simple checking of results with respect to the original program is sufficient in practice. Methods for more extensive automated testing and verification are left for future work. The optimization process took around 78 minutes (for a relatively simple kernel), and the performance of optimized scientific kernel increased by 50%.

Related Work In addition to program synthesis line of work explained above, related work includes peephole optimization [1, 2] and superoptimization [15], which find the optimal code sequence for a single, loop-free sequence of instructions. Furthermore, there are many technologies that work on loops in source code, such as polytope model [9]. Olschanowsky et al recently applied polytope model to generate thirty different inter-loop optimization strategies on NUMA multicore node [12]. Pouchet et al proposed characterization of multidimensional affine transformations as a convex space and an optimization algorithm on the space [14]. As loops are generally the most time-consuming part of program, technologies in this category have strong importance in practice. However, their applications are also limited to fragments of entire program. Another approach of providing automated optimization is through specialized library. For example, Automatically Tuned Linear Algebra Software (ATLAS) [22] provides portably optimal linear algebra library, as the library finds automatically finds its own parameters that are optimal for a given architecture. There are several approaches for automating program optimization. Suda proposed online automatic tuning based on a Bayesian reasoning [19]. In this method, several program and system parameters are tuned. Based on a similar approach, IBM has developed "IBM High Productivity Computing Systems Toolkit" which optimizes performance of libraries in real time. These two approaches do not change source code, but rather tune it through parameterization. Barik et al presented an auto-vectorization framework in the back-end of a dynamic compiler that includes scalar packing and algebraic reassociation [3]. Cavazos et al applied machine learning for ordering optimization techniques during compilation [8].

2. Illustrative Example

We illustrate our approach on a small piece of computation-intensive code called DG kernel. The kernel is important enough to be optimized manually by multiple optimization experts, including engineers employed by compiler vendors. The experts optimized the code with respect to particular architectures, such as Intel Xeon Phi and Nvidia GPUs.

DG Kernel DG Kernel is derived from one of the computationally intensive parts of the High-Order Method Modeling Environment (HOMME) [4]. HOMME is a framework to investigate the utilization of high-order methods to build scalable, accurate, and conservative atmospheric general circulation models. DG Kernel calculates the gradient in the Discontinuous Galerkin (DG) formulation of the shallow water equations. There are two outer-most DO loops in DG kernel code shown in Figure 1. The first DO loop (line 5) calculates the gradient for every element and the second DO loop (line 25) updates two arrays using the calculated gradient. The term element represents a unit area in a simulation grid.

2.1 Performance Optimization Techniques

Performance optimization can be achieved at every stage of software development, from design phase to execution phase, and a wide range of optimization techniques is typically available to developers at every stage. We focus on the following optimization techniques: (a) setting environment variables, (b) applying (right combinations of) compiler flags, (c) source code transformations, including adding compiler directives in source code.

```
1  !$OMP PARALLEL DEFAULT(NONE) &
2  !$OMP SHARED(flx,fly,grad,delta,der,gw) &
3  !$OMP PRIVATE(ie,ii,i,j,k,l,s2,s1)
4  !$OMP DO
5  DO ie=1,nelem
6   DO ii=1,npts
7    k=MODULO(ii-1,nx)+1
8    l=(ii-1)/nx+1
9    s2 = 0.0_8
10   DO j = 1, nx
11    s1 = 0.0_8
12    DO i = 1, nx
13     s1 = s1+(delta(l,j)*flx(i+(j-1)*nx,ie)*&
14      der(i,k)+delta(i,k)*fly(i+(j-1)*nx,ie)*&
15      der(j,l))*gw(i)
16    END DO
17    s2 = s2+s1*gw(j)
18   END DO
19   grad(ii,ie) = s2
20  END DO
21 END DO
22 !$OMP END DO
23
24 !$OMP DO
25 DO ie=1,nelem
26  DO ii=1,npts
27   flx(ii,ie) = flx(ii,ie)+dt*grad(ii,ie)
28   fly(ii,ie) = fly(ii,ie)+dt*grad(ii,ie)
29  END DO
30 END DO
31 !$OMP END DO
32 !$OMP END PARALLEL
```

Figure 1. DG Kernel

Environmental Variables In some cases, it is important to setup environment variables correctly in order to achieve good performance. For example, the value of "OMP_NUM_THREADS" tells the OpenMP library how many threads to use. Another example of environment variable usage is that thread affinity can be configured using an environment variable such as "GOMP_CPU_AFFINITY" for a GNU compiler.

Compiler Flags Most of modern compilers provide vast number of compiler flags that affect to performance. The "-O3" compiler optimization flag is a common example for high-level optimization. There are many categories of compiler flags used for performance optimizations including vectorization, parallelization, data alignment, instruction-set generation, etc. Selecting the right set of compiler flags for optimization is, in general, a challenging and time-consuming task.

Source Code Transformations The performance of a given program on a particular architecture depends on many factors, including cache/memory usage pattern, required cycles per instruction, synchronization among threads, etc. These factors can be improved by a number of *semantics-preserving* code transformations. Typically, loops are the most time-consuming parts of code. There are therefore many source code transformation techniques developed for optimization of loops, including loop unrolling and loop merging/splitting. These techniques may, for instance, achieve regular memory access pattern in order to help maximize the performance of cache and memory on processor.

2.2 Manual Optimization and Results

DG kernel shown in Figure 1 has been manually optimized for better performance by several experts. The speed-ups achieved by different experts vary largely. The best speed-up was achieved by applying a combination of multiple optimization techniques shown in

```
[Environmental variables]
    OMP_NUM_THREADS=16
[Compiler flags]
    -openmp -O3 -align array64byte -opt-prefetch=0
[Compiler directives]
    !DEC$ ATTRIBUTES ALIGN, !DEC$ ASSUME_ALIGNED
    !DEC$ vector always aligned
    !$OMP END DO nowait, and !DEC\$ noinline
[Source transformations]
    Loop unroll, loop merge, array-merge, pre-calculation
```

Figure 2. The optimization techniques that produce the best DG kernel performance on Intel SandyBridge.

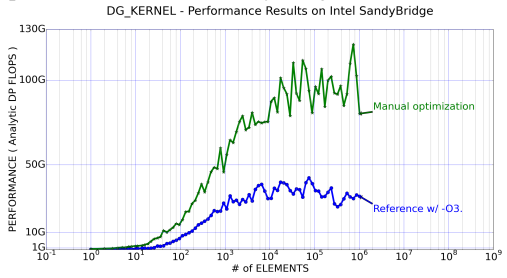

Figure 3. Performance results for original and manually-optimized code on Intel SandyBridge

Figure 2. Figure 3 shows performance results of original and manually optimized DG kernel on Intel SandyBridge. Elements in the plot represent the size of problem to solve. Analytic DP FLOPS is a relative performance metric calculated as the analytically computed number of double-precision floating-point operations from the original code divided by running time of the execution.

However, finding the right set of optimization techniques for a given program and architecture is not always a feasible option for an engineer. To them, it is not always clear which optimization techniques will work, and it is too time consuming to try many different optimization techniques and their parameters. Furthermore, it is generally getting more difficult to optimize performance on new types of micro-architectures such many-core processors and GPUs as it has deeper layers of memory hierarchy, more cores, and longer vector registers; hence, the effect of various optimization is thus more unpredictable.

2.3 Performance Optimization as a Search Problem

One can see the process of manual performance optimization as searching for an optimal set of optimizations (and their parameters) among all available ones. A search space can be created as follows. For example, if we want to select one of "-O1", "-O2", and "-O3" compiler optimization flags, it defines a search space with three options. In another example, if we want to unroll three Fortran DO loops, let's say "LOOP1", "LOOP2", "LOOP3", that defines search space, with 8 values - that is we can choose a subset of the loops to unroll. In principle, all these 8 options may have very different performance. The search space for other types of source code transformation has a more complex structure, as sometimes the order in which these transformations are applied matters. Consider the case of loop merging (we could for instance merge the first DO loop (line 5) and the second DO loop (line 25) into one loop) and loop unrolling: it matters whether we unroll the first loop and then merge, or merge and then unroll. We thus consider combinations (and permutations where appropriate) of optimization techniques to create search spaces. We then consider the product of these search spaces as the overall search space.

2.4 Properties of the Search Space

There are two properties of the search space that make finding the optimal set of optimization techniques difficult: the size of the search space, and the non-smoothness of the impact on performance. First, the search space generated from combinations/permutations of optimization techniques is extremely *non-smooth*. That is, the impact on performance of one set of options may be very different from the impact of a very similar set. This makes searching through the search space time-consuming, and perhaps frustrating, to do manually. Second, the size of the search space grows exponentially with the number of optimization techniques considered. In the examples we consider, the size can easily grow beyond 2^{30}, even considering only basic source code transformations and compiler flags.

Due to the fact that search space can be very large, and performance is non-smooth as a function defined on this search space, automating the search can improve productivity of performance optimization enormously. OpenCase is a tool developed to generate search cases based on user-defined optimization schemes and actually executes each case that a search algorithm requests on a machine. In this paper, two such search algorithms, random search and casetree search, are described and evaluated.

3. OpenCase: Specification of Optimizations and Performance Search

OpenCase allows user to define a search space based on user-selected optimization techniques, described in the Search-Space Description Language (SSDL). In what follows, a *case* refers to a set of particular optimization techniques that can be applied to original Fortran program. The search space consists of a large number of cases. OpenCase executes each case on the target machine and tests the correctness of the optimized program with respect to the original program. The whole process is performed automatically.

3.1 Search Space Generation with Compiler Flags and Source Transformation Techniques

Search-Space Description Language (SSDL)

Users can define the search space of optimization techniques in SSDL. Figure 4 shows the core part of SSDL definition.

```
⟨direct-list⟩   ::=  ⟨direct⟩ | ⟨direct⟩ ⟨direct-list⟩
    ⟨direct⟩    ::=  ⟨direct-name-opt⟩ ⟨LB⟩ ⟨elem-list⟩ ⟨RB⟩⟨gen-opt⟩
⟨elem-list⟩     ::=  ⟨elem⟩ | ⟨elem⟩ ; ⟨elem-list⟩
    ⟨elem⟩      ::=  ⟨item-list⟩ : ⟨attr-list⟩
⟨item-list⟩     ::=  ⟨item⟩ | ⟨item⟩ , ⟨item-list⟩
⟨attr-list⟩     ::=  ⟨attr⟩ | ⟨attr⟩ , ⟨attr-list⟩
    ⟨item⟩      ::=  STRING | ⟨direct⟩
    ⟨attr⟩      ::=  ⟨keyword⟩ = ⟨item⟩
    ⟨LB⟩        ::=  ( | { | [ | ⟨
    ⟨RB⟩        ::=  ) | } | ] | ⟩
⟨gen-opt⟩       ::=  NEWLINE | ⟨gen⟩ | ⟨gen⟩ NEWLINE
    ⟨gen⟩       ::=  EMPTY | * | NUMBER | *NUMBER
```

Figure 4. Search-Space Description Language(SSDL)

In OpenCase, the user creates search cases using SSDL based on the user's prior-knowledge on performance optimization. $\langle direct \rangle$ symbol constructs OpenCase directives. For example, "BUILD(make: makefile=Makefile, FC=ifort, FC_FLAGS=(-O1;-O2;-O3)1)" describes that OpenCase will build the software using the make building system with Intel Fortran compiler and one of the three compiler flags:"-O1", "-O2", and"-O3". Thus, the directive creates three search cases. In the example, "make" is an instance of $\langle item \rangle$ or $\langle item\text{-}list \rangle$, and "FC=ifort" is $\langle attr \rangle$ or $\langle attr\text{-}list \rangle$.

In SSDL, each brackets represent different types of case generations: parenthesis, "()", for combination, square brackets, "[]", for permutation, braces "{}", for accumulative combination, and chevrons, "⟨⟩", for accumulative permutation. Asterisk mark in

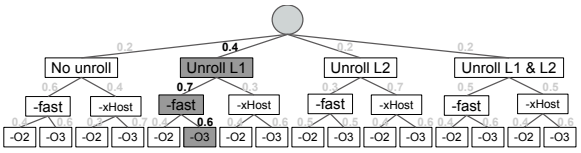

Figure 5. A tree representation of the search space generated from "((-O2;-O3)1;(-fast;-xHost)1)" and "{L1;L2}*" SSDL description. Each node of the tree has weights assigned to each sub-branch. Dark boxes have the highest probabilities to be chosen

$\langle gen \rangle$ symbol represents an empty set case and number in the symbol represents the number of elements to select. If number is omitted, it is interpreted as the number of all elements. "Accumulation" of braces and chevrons has meaning of summing up all the cases up to the specified number in $\langle gen \rangle$ symbol. Some examples of selected usages are below. "=>" in the examples means "generates."

```
(a;b;c)2 => { (a,b), (a,c), (b,c) }
[a;b;c]2 => {(a,b),(b,a),(a,c),(c,a),(b,c),(c,b)}
{a;b;c}*2 => {(),(a),(b),(c),(a,b),(a,c),(b,c)}
<a;b;c>* => {(),(a),(b),(c),(a,b),(a,c),(b,c),(a,b,c),
             (a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a)}
```

A more complicated search space can be modeled as a tree. Assume that we want to create search cases with unrolling some or none of two loops, applying one of "-O2" and "-O3" compiler flags, and one of "-fast" and "-xHost" compiler flags. Figure 5 shows the search space in a tree model. In this model, finding a right set of optimization techniques is formulated as search for a case that comprises of the "right" decision nodes from the root to a leaf.

3.2 Implementation of Optimization Techniques

While SSDL does not limit the types of optimization techniques to be expressed, current OpenCase implementation supports three types of optimization techniques: environmental variables, compiler flags, and source code transformations.

Environmental Variables With "PRERUN" OpenCase directive, user can specify search cases with a command that will be executed before compiling source code. SSDL syntax and an example are shown below.

```
SSDL SYNTAX :PRERUN <direct>
EXAMPLE:PRERUN('export␣OMP_NUM_THREADS=8';
  'export␣OMP_NUM_THREADS=16')1
```

Compiler Flags Among various software building system, OpenCase currently supports only "make" building system. Two attributes, FC and FC_FLAGS, are used for specifying a compiler command and compiler flags as shown below.

```
SSDL SYNTAX :BUILD <direct>
EXAMPLE:BUILD( make:makefile=Makefile,
  FC=ifort,FC_FLAGS=((-O2;-O3)1;(-fast;-xHost)1))
```

Source Code Transformations There are various source code transformations implemented in OpenCase. The exact syntax is different for each transformation. However, all of them follows general syntax: name of transformation followed by parameters of the transformation as shown below.

```
SSDL SYNTAX :SRCGEN <direct>
EXAMPLE:SRCGEN(loop_unroll:target={L1;L2}*)
```

The following list shows the names of source code transformations implemented in OpenCase. Unroll repeats the body of a loop to reduce the iteration of the loop. Merge combines multiple loops into one. Split seperates a body of a loop into multiple ones. Interchange switches the loop-control of the inner loop with the loop-control of the outer loop.

```
Loop transformations: Unroll, Merge, Split, Interchange
Compiler directives: OpenMP, Any directive
Primitives: Statement insert/delete, Name modify/switch
```

3.3 Performance Search Engine

OpenCase modifies source codes and generates a shell script according to the case number of the search space generated from SSDL description. It then executes the shell script, collects output data, and checks its correctness. The process stops if the performance result meets user's goal, or it continues if it does not. Figure 6 shows the process in a block diagram.

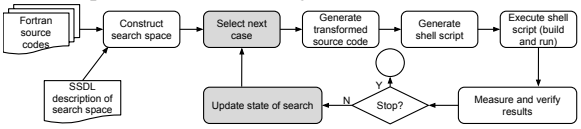

Figure 6. OpenCase block diagram. Search algorithm provides OpenCase with functions for the two gray blocks

Source Code Generation First, OpenCase transforms source files according to information collected from OpenCase "SRCGEN" directives. If there are multiple transformations defined, each of them is applied in order. If two transformations are not compatible, latter transformation is not applied.

Shell Script Generation In this stage, OpenCase utilizes several OpenCase directives including "PRERUN", "BUILD", "EXECUTE", and "POSTRUN" and populates them into a shell script in order. This is a conventional shell script file.

Shell Script Execution The generated shell script contains commands required to execute a search case. OpenCase creates a separate process to run the shell script. The compiled program is run multiple times to improve the accuracy of performance measurement according to user-provided configuration through "EXECUTE" OpenCase directive.

Measurement and Checking In this stage, OpenCase utilizes two OpenCase directives, "MEASURE" and "VERIFY". Current implementation of OpenCase collects kernel output from standard output during kernel execution and applies verfication method statically specified in "VERIFY" OpenCase directive to the collected kernel output. User can specify different methods to capture the results of the optimized case and to check the correctness of the measured results according to the "VERIFY" directive. An efficient heuristic of checking the results of the optimized program is possible for scientific array-based programs, where there are relatively few control branches that depend on the contents of the array. Hence, if the results of the optimized and unoptimized version are the same for a small number of input arrays, it is an indication that the result would be the same for all input arrays. Empirically, this type of simple checking mechanism was validated for array-based programs.

4. Search Algorithms

In practice, it is essential to devise a search algorithm that examines a smaller number of sets of optimizations until OpenCase finds a case that meets the user-specified goal. To make it possible to develop various search algorithms, OpenCase provides a simple interface that enables the user to implement a new algorithm with ease. The interface has two functions: one for selecting next case number, and the other to update the internal information on the search space. These functions are called at every iteration of the search process. Two sample algorithms, random search and case-tree search, are implemented in order to illustrate the simplicity of using the OpenCase interface.

Random Search Random search algorithm generates randomly the next case to be examined. The chance to find the cases that meets user's goal is proportional to the ratio of the number of "good" cases to the total number of cases in the search space.

Casetree Search Casetree search algorithm generates a case number for the next case to be examined according to the probability distribution among optional choices on each decision nodes. Figure 5 shows an example tree representation of a search space with weights assigned to each sub-branches. The main idea of the algorithm is to increase the weight of a branch if a decision represented by the branch in the tree had better performance impact, and to decrease it for negative impact on performance. The updated weights of a node, in turn, form a probability distributions that a next case is selected from. In this way, the algorithm uses knowledge learned from previous decisions and their results. Algorithm 1 shows the pseudo code for the "update" function which updates weights of sub-branches. There are two failure cases in the algorithm: compilation error and checking error. A weight of a choice is decreased in a failure case except when the choice was used for generating the current best case. By having this exception, the algorithm gives more chances to the choices that contributed to the best case.

Algorithm 1 update(case,ranking,num_success,num_failure)

```
if execution is failed then
    if current choice did not contribute for current best case then
        decrease weight
else if execution is successful then
    if ranking is better than median then
        increase weight
```

5. Results

Two kernels are automatically optimized using OpenCase on Intel SandyBridge. DG Kernel is described in Section 2. Taumol03 kernel is extracted from PSRad [13]. PSrad is a new radiation package designed for use in models of the atmosphere. We evaluate whether the algorithms produce programs that are (a) correct w.r.t. the original program, (b) have improved performance, and (c) how long does the optimization process take for kernels of typical size. The correctness is checked as explained in Section 3 and finally confirmed by human experts.

An experiment is setup with random search and casetree search algorithms. In one experiment, 700 cases are evaluated in DG kernel and 1000 cases are in Taumol03 kernel. On average, evaluating one case by OpenCase took 6.7 seconds for DG kernel and around 8.6 seconds for Taumol03 kernel. The whole optimization process took around 78 minutes. A kernel generated in one search case is executed five times for improving accuracy of timing measurement. Each experiment is tagged as RAND n (random search) or CTREE n (casetree search) in Figure 8 and Figure 9.

Figure 7 shows the SSDL description used to specify a search space for automated DG kernel optimization using OpenCase. PRERUN, SRCGEN, and BUILD OpenCase directives are explained in Section 3. The other OpenCase directives do not contribute for increasing the search space, but provide OpenCase with operational information. REFCASE provides a sequence of command that generate the reference performance output. CLEAN tells OpenCase how to initialize the kernel before executing next modified kernel. EXECUTE contains information to run the kernel for each search case. MEASURE defines what to measure. VERIFY tells OpenCase how to check the correctness of an output from executing a search case. RANK selects a measurement for ranking. CASEGEN selects a search algorithm for the experiment.

Figure 8 and 9 show the best performances found by OpenCase for the DG kernel and the Taumol03 kernel. Horizontal axis contains experiments and vertical axis shows performance. For the DG kernel, the performance is represented by Analytic Giga-Floating-Point Operations Per Second (GFLOPS). This metric is obtained by dividing the total number of analytically counted FLOPs in the

```
REFCASE('export␣OMP_NUM_THREADS=16;
␣␣make␣clean;␣make␣ref_recover;␣make
␣␣build␣FC=ifort␣FC_FLAGS="-O3␣-openmp";␣make␣run')
PRERUN('export␣OMP_NUM_THREADS=16';
   'export␣OMP_NUM_THREADS=32')1
SRCGEN(loop_unroll: target={80;100;230}*,
   factor=full, method=('inc';'const')1)
SRCGEN(loop_merge: from=220, to=30)*
SRCGEN(loop_merge: from=230, to=40)*
SRCGEN(loop_split: before=(250)*)
SRCGEN(loop_interchange: outer=220, inner=230)*
SRCGEN(remove_stmt: target=(200;210;280;290))*
SRCGEN(name_change: target=(240;250), switch='ii:ie')*
CLEAN(make: makefile=Makefile, target=clean)
BUILD(make: makefile=Makefile, target=build,
   FC=ifort, FC_FLAGS=('-openmp'; (-O2;-O3;-fast)1;
   ('-opt-assume-safe-padding')*; ('-opt-prefetch=0';
   '-opt-prefetch=3')1;{'-no-prec-sqrt';'-no-prec-div'}*))
EXECUTE(make: makefile=Makefile, target=run,repeat=5)
MEASURE(gflops: prefix='Gflops␣␣␣=')
MEASURE(flx_diff: prefix='SUM(flx)␣=')
VERIFY(flx_diff: method='diff', refval='0.1113450E+02',
   maxdiff='1.0E-15')
RANK(gflops: sort=descend)
CASEGEN(ctree)
```

Figure 7. SSDL description for DG kernel search space

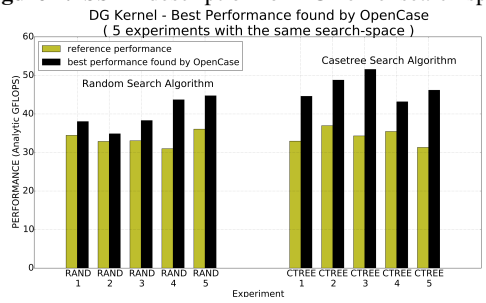

Figure 8. DG kernel best performances found by OpenCase on Intel SandyBridge

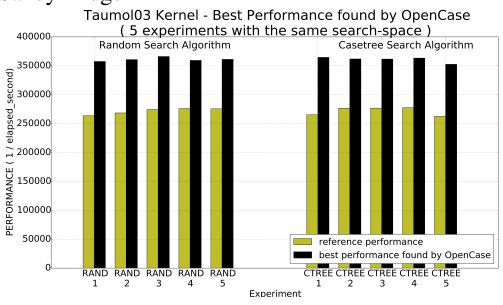

Figure 9. Taumol03 best performances found by Opencase on Intel SandyBridge

original source code by elapsed execution time of each executed case. Each darker bar in the plots is the best performance found by OpenCase among all the cases executed in the experiment.

The first observation is that OpenCase found better performance than reference performance in all experiments of both of two kernels regardless of algorithm used. This proves that the search-space specified by OpenCase directives in Figure 7 contains some cases that generate better performance than reference performance, and OpenCase can find those "good" cases. Casetree algorithm consistently found higher performance results than random algorithm in DG kernel experiments. There is not a significant difference between two algorithms for Taumol03.

While finding the best-performing case is the most interesting result in performance optimization, OpenCase enables to collect

Table 1. Compiler flags and their usage count for top30 cases of DG kernel found by OpenCase

compiler flag	count
-fast	25
-opt-assume-safe-padding	21
-opt-prefetch=3	21
-no-prec-sqrt	14
-no-prec-div	10

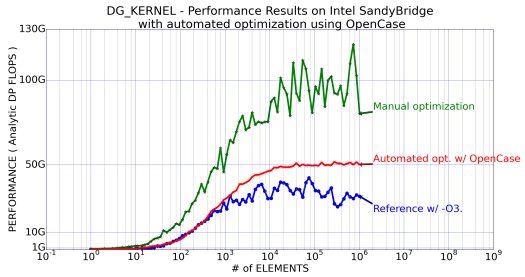

Figure 10. Performance results for original, manually optimized code, and OpenCase-optimized code on Intel SandyBridge

additional information that provides user with better understanding of performance optimization. As an example, we shows that it is possible to find out "the right" set of optimization techniques. Table 1 shows the compiler flags that are used for top 30 cases in DG kernel experiments and the counts of each flags. As all of compiler flags in Table 1 come from top 30 cases, they are considered as "good" compiler flags for performance on Intel SandyBridge. Especially "-opt-assume-safe-padding", "-fast", and "-opt-prefetch=3" worked better than the other flags.

Finally, Figure 10 shows the performance results for the original, manually optimized, and automatically optimized DG kernels in one plot to compare the level of optimizations. Automated performance optimization has achieved around 1.5X speed-ups from the original performance and marked around 50% performance compared to the performance from manual optimization. Array-merge and pre-calculation source transformations are key techniques for speed-ups in manual optimization, and their implementation is left for future work.

6. Conclusion and Future Work

In this paper, we showed that automated optimization using OpenCase can explore a large number of different optimizations. This would be hard to achieve through manual optimization in practice. We showed that OpenCase can achieve better performance than a conventional compiler optimization, because it can generate search space based on user's prior knowledge on performance optimization, and search efficiently Two examples of automated optimization on OpenCase are presented: DG kernel and Taumol03 kernel. In both cases, OpenCase found an optimized case that shows better performance than each of reference cases. In addition, it is shown that the output from the automated searching on OpenCase could generate valuable information that user can get insights on performance optimization for the target architecture.

Currently, there are two directions for future work. First, advanced source transformations could be added to OpenCase. By having them OpenCase can get closer to performance results achieved by manual optimization. Second, better search algorithm could be added to OpenCase. We plan to investigate uses of classical and custom optimization algorithms (such as Metropolis-Hastings, simulated annealing) or machine learning algorithms.

Acknowledgments

This work was supported in part by a gift from the Intel Corporation, through NSF Cooperative Grant NSF01 which funds the National Center for Atmospheric Research (NCAR), and through an Intel Parallel Computing Center grant from the Intel Corporation.

References

[1] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, pages 394–403, 2006.

[2] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *OSDI*, pages 177–192, 2008.

[3] R. Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. In *MICRO 2010*, pages 201–212, 2010.

[4] J. Dennis, J. Edwards, K. Evans, O. Guba, A. Mirin, M. Taylor, and P. Worley. Cam-se: A scalable spectral element dynamical core for the community atmosphere model. In *The International Journal of High Performance Computing Applications*, pages 26, 74–89, 2012.

[5] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[6] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *CACM*, 55(8):97–105, Aug. 2012.

[7] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Concurrent data representation synthesis. In *PLDI*, pages 417–428, 2012.

[8] S. Kulkarni and J. Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *OOPSLA*, pages 147–162, 2012.

[9] C. Lengauer. Loop parallelization in the polytope model. In *Proceedings of the 4th International Conference on Concurrency Theory*, pages 398–416, 1993.

[10] T. Mytkowicz, A. Diwan, and E. Bradley. Computer systems are dynamical systems. In *CHAOS*, 2009.

[11] K. Naono, K. Teranishi, J. Cavazos, R. Suda, and Editors. Chapter 1. In *Software Automatic Tuning*, pages 3–3, 2010.

[12] C. Olschanowsky, M. Strout, S. Guzik, J. Loffeld, and J. Hittinger. A study on balancing parallelism, data locality, and recomputation in existing pde solvers. In *SC14*, pages 793–804, 2014.

[13] R. Pincus and B. Stevens. Paths to accuracy for radiation parameterizations in atmospheric models. In *J. Adv. Model. Earth Syst.*, pages 5, 225–233, 2013.

[14] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: convexity, pruning and optimization. In *POPL 2011*, pages 549–562, 2011.

[15] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, pages 305–316, 2013.

[16] A. Solar-Lezama, L. Tancau, R. Bodík, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[17] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[18] A. Solar-Lezama, C. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.

[19] R. Suda. Chapter 16: A bayesian method of online automatic tuning. In *Software Automatic Tuning*, pages 275–293, 2010.

[20] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, pages 125–135, 2008.

[21] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338, 2010.

[22] R. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. In *Software: Practice and Experience*, volume 35, pages 101–121, February 2005.