HSLOT: The HERCULES Scriptable Loop Transformations Engine

Christos Kartsaklis†, Eunjung Park∗, John Cavazos∗
† Oak Ridge National Laboratory, Oak Ridge, TN 37831
kartsaklisc@ornl.gov
∗ University of Delaware, Newark, DE 19716
{epark,cavazos}@cis.udel.edu

Abstract—HSLOT arms users with a rich set of configurable transformation directives, to be used as-they-are or to be specialized and combined into powerful custom transformations. We offer a plethora of loop transformations, which includes both the classic set (unroll, fuse, fission, tile, and so on) as well as unique ones (specialize, swap nest, split, fork, and so on) that are not found in other state-of-the-art systems. We show how HSLOT enables more transformations such as merging two loops that cannot be fused because of data dependencies and how HSLOT can be used in a simple and systematic fashion to improve memory accesses and expose better parallelism. To use our system, users simply annotate loops with the transformations sequence and compile with our Open64-based HSLOT-implementing Fortran compiler, HSLF90, which produces both object files and optionally source. We describe our experiment results using a set of scientific kernels written in Fortran with HSLOT directives on AMD 32 core system.

Keywords—compiler optimization, machine learning, predictive modeling, pattern-based program characterization

I. INTRODUCTION

High-level loop transformations comprise one of the most broadly employed techniques for program optimization [1], [2], [3], [4], [5], [6]. Modern compiler frameworks provide various loop transformations and their standard compiler optimization flags such as -O3 in GCC often enable most of loop optimizations aggressively. Loops exhibit unique structure and data access patterns, making it quite hard for a single transformation strategy to work well across all conceivable loops. Suboptimal strategies are often due to compilation time constraints, but most importantly, they can be attributed to uncertainty (dependencies, aliasing, etc.). On the other hand, we have observed how quickly users adopt directive-based parallelization APIs (OpenMP, OpenACC, OpenHMPP, etc.). These APIs do code generation (threaded loops, CUDA-gridified loops) with minimum validation leaving the responsibility of confirming legality (dependency analysis) with the user. Nonetheless, such are the gains in programming effort, that users are willing to “live with it”, since the code parallelization effort gets accelerated dramatically [7].

In this paper, we introduce a directive based loop transformation system called HSLOT (the HERCULES Scriptable Loop Transformations Engine), as part of our greater HERCULES framework [8]. Using a set of HSLOT directives, users can simply tag a loop-of-interest, and then compose a set of transformations in the order they would like them applied to the loop. After users annotate a source code with HSLOT directives, our HSLF90 compiler processes the source code and generates the object file and optionally transformed source code as well. Transformed code may be reported on a step-by-step basis, which allows users to step through the transformation process and observe closely how the code is manipulated.

The advantage in using HSLOT lies in its rich set of loop transformations, which includes the standard transformations (unroll, fuse, fission, tile, and so on) as well as unique ones (specialize, swap nest, split, fork, peel, and so on) not found in other systems. For example, one such unique transformation is peel, which executes a user-provided number of a loop’s iterations before or after the loop is entered or exited. A smart use of peel can break certain dependencies and enable the merging (fusion) of loops. Detailed descriptions and advantages of using peel and other transformations are given in Section II.

The rest of this paper is organized as follows: We first introduce motivation examples to show how the unique set of loop transformations introduced in HSLOT can be beneficial in transforming code (Section II). In Section III, we briefly discuss how HSLOT works with our HSLF90 compiler and proceed with describing the operation and configuration of most HSLOT loop directives, using examples for demonstration. Section IV provides additional examples that focus on composing transformations by combining HSLOT directives together. Then, in Section V, we describe our experimental setup and results. We compare our work to other related work in Section VI and, lastly, we conclude in Section VII.

II. MOTIVATION

We introduce three motivating examples of using HSLOT, especially focusing on the unique set of loop transformations that are not found in other systems. We describe how a combination of the transformations’ unique nature and the different options that they can be fine-tuned with can be beneficial to the overall optimization task.

A. Merging Non-Fusible Loops

Figure 1 shows an example of two loops that are not fusible in the original program. The transformed code in the left panel of Figure 1 is the result of fusing loop L1 and L2; this code produces a different output from the original program because of a(k). In the original code, $S_2$ updates

978-1-4799-7020-9 2014
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/WOLFHPC.2014.10
the value of \(a(k)\) after \(S1\) is done with reading them. However, in the transformed code, \(S2\) updates \(a(k)\) before \(S1\) reads it again in the next iteration. The transformed code in the right panel of Figure 1 is the result of using HSLOT loop transformations. We composed a set of two loop transformations using HSLOT directives. First, we use the HSLOT transformation peel to peel off a certain number of iterations. Users can specify how many iterations they want to peel off from either the beginning and/or the end of the loop. Directive peel(\(L1\), first(1)) means the first iteration of loop \(L1\) will be peeled. This transformation breaks the previously identified data dependency that would lead fusion into producing incorrect results. Then we fuse the two loops using fuse(\([L1,L2]\), different, LF) which merges \(L1\) and \(L2\) into a new loop named \(LF\) and also indicates that the two loops, \(L1\) and \(L2\), have different loop bounds (option different). The latter generates the additional loops to deal with the remaining iterations, which are only shown for \(L2\), since the one corresponding to \(L1\) was eliminated by the compiler because of a zero trip count. The code generated by HSLOT reproduces the original code’s output because the loop-carried data dependency does not exist anymore.

**B. Determining the Unroll-Factor Dynamically**

The code in the right panel of Figure 2 is generated using fork(L, multiple(4), LNOT) followed by unroll(L,4) to unroll loop L four times only if the loop bound is a multiple of four. The code in the bottom-right panel of Figure 2 is the result of first applying the previous transformations, and then fork(LNOT, multiple(3), LNOT_NOT) followed by unroll(LNOT,3): we apply an additional fork and unroll against LNOT, i.e. in the case where the original loop’s bound was not a multiple of 4. We check if the loop bound is a multiple of three, then unroll threes times the loop only if the predicate is true. As shown in this example, fork can help users in creating conditional branches and optimizing loops differently.

**C. Eliminating Unnecessary Computations**

Figure 3 describes another interesting example using transformations unique to HSLOT – specialize and nest. Transformation specialize performs either a conditional or an unconditional constant propagation. In the conditional form, it generates two branches of the code, similarly to fork, and propagates the constants to the then branch. Notice in the example that when alpha is zero, the expression “alpha*a(k,i)*b(j,k)” also becomes zero, leaving array \(c\) unchanged, i.e. \(L3\) does not need to be executed because the value of \(c(j,i)\) never changes. Therefore we need to eliminate this loop. We can achieve this by placing, or nesting, \(L3\) under a zero trip-count loop, i.e. a loop that will not execute.

**D. Summary**

In this section, we briefly introduced four transformations - peel, fork, specialize and nest. They are some of the unique transformations in HSLOT whose usage enables further loop transformations. Using peel and fuse together to remove a certain data dependency that prohibits the further fusion of two loops. A smart use of peel leads to proper manipulation of the iteration space and readies the code for parallelization, as we showed with the subsequent application of HSLOT’s openmp transformation. We also demonstrated
an example of using fork and unroll together to unroll a given loop with different unroll factors subject to the trip count. This combination of transformations is particularly beneficial when properties of the trip count are not statically known. We also demonstrated how specialize and nest can be used together to remove unnecessary computations conditionally. Other loop transformations available in HSLOT are introduced in Section III.

III. HSLOT OVERVIEW

HSLOT (HERCULES Scriptable Loop Transformations) is a collection of loop-oriented transformations, which we have implemented in Open64’s Very High WHIRL Optimizer (VHO) phase. This is generally regarded as the stage that Open64 enters right after language-specific front-ends instantiate the corresponding intermediate representation (WHIRL). Soon after WHIRL is generated, the HSLOT implementation will begin executing the directives. Once done, control is returned to Open64. We have developed some auxiliary code, such as VHO-level def-use analysis, to support helper tasks such as scalar access hoisting, which Open64 offers at stages past VHO.

HSLOT provides a rich set of loop transformations including standard loop transformations and also a unique set of transformations. HSLF90 wraps around openf90, Open64’s Fortran90 compiler, and hides the specifics of configuring the HERCULES backend. HSLF90 generates the typical object file format and then parallelize it by writing "%!hslot unroll(L), openmp(for,L)".

The transformation directives are always processed in the order that they appear in the text (left to right, top to bottom), with the lexical extent (e.g. inside a loop) having no significance. The list of transformations in HSLOT is summarized in Table I with each transformation being described with an example in the rest of this section.

### Table I. List of Transformations in HSLOT.

<table>
<thead>
<tr>
<th>Loops Processed</th>
<th>Transformations Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Loop</td>
<td>unroll, split, fork, peel, nest, openmp, fission, align, gpurr, hoistsa, ivarep, normalize, stride</td>
</tr>
<tr>
<td>Double Loops</td>
<td>collapse, fuse, unroll, interch</td>
</tr>
<tr>
<td>Multiple Loops</td>
<td>tile, specialize</td>
</tr>
</tbody>
</table>

A. Unroll

This transformation unrolls a given loop \( l \) with an unroll factor \( n \) as shown in Figure 6. The first parameter is the identifier of a loop to unroll, the second parameter is an unrolling factor, and the third, optional, parameter can be used to specify a new loop identifier for the unrolled loop.

### Format: unroll(l, integer n, \{doloop\}? )

Fig. 6. The unroll transformation.

B. Tile

The tile transformation creates blocks of loop iteration spaces to improve the data locality. Figure 7 demonstrates the format of tile directive and the example of using it to tile a double nested loop with a tiling factor \( BX \times BY \). We can specify a tiling factor with either a constant value or a variable. When we specify it as a variable, we need to make sure the variable is defined in the program and the value is already assigned by an integer larger than 0. The first parameter is the list of loop identifiers we wish to tile, while the second parameter is used to specify the corresponding tile sizes. The last parameter is optional and is used to specify identifiers for the set of outer loops created for each loop in the first parameter. Therefore all parameters should be of equal length.

### Format: tile(l, \{doloop1, ..., doloop l\}, [\{doloop1', ..., doloop l'\}] )

Fig. 7. The tile transformation.

C. Split

The split transformation, distributes the iteration space of \( l \) to two new loops based on the strategy signified by the second parameter (DESCRIPTOR). The optional bindings 11

Fig. 4. Overview of HSLOT and HSLF90.
and 12 may be assigned to the two new loops. DESCRIPTOR may be:

- `round(VALUE)`: 11 performs the first `VALUE` iterations, aligning the lower bound to `VALUE`, while 12 does the rest.
- `evenodd`: 11 performs the odd loop iterations (i.e. when the induction variable is an odd number), while 12 does the even ones; this often serves as basis for even-odd parallelization schemes [9].
- `middle`: 11 performs the first half of the iteration space of 1 and 12 performs the rest of the iterations.

Figure 8 depicts the transformation of the code with three different configurations for the DESCRIPTOR field.

**D. Collapse**

The purpose of the collapse transformation, which both of the OpenMP and OpenACC standards also support, is to flatten out the iteration space as a preparatory step for code parallelization [10], [11]. collapse, shown in Figure 9, takes a double-nested loop as input (11 is the outer while loop and 12 is the nested loop) and collapses them into a new one, 13. Omitting the optional 13 will update 11 so that it points to the new loop. Note that any non-iterative statements that may interleave 11 and 12, such as "call foo()" in the example, are permissible and will be appropriately wrapped in conditional code to execute only according to the outer loop's schedule.

**E. Fork**

Transformation fork creates a conditional branch based on predicate PROPERTY and makes the given loop l1 reachable in both of the `then` and `else` parts of the `if` statement as shown in Figure 10. The transformation itself does not perform any loop manipulation other than cloning the loop. However we can utilize this directive by optimizing a loop differently based on PROPERTY as we already shown in Section II.2. In the current system, we only support the `multiple(n)` predicate, which checks if the loop bound is a multiple of n; we plan to offer additional predicates in the future.

**F. Peel**

The peel transformation unconditionally executes n iterations at the beginning and/or the end of the loop as shown in Figure 11. When `first(n)` is given, the first n iterations of the loop will be executed outside of the loop. `last(n)` operates similarly to `first(n)` except it peels iterations off then end of the loop and, consequently, executing them after the loop terminates. This transformation can make other transformations applicable as shown in Section II.1.

**G. Fuse**

Transformation fuse merges the n loops l1, ... ln, which are specified in the first parameter, together under the constraint that all loops exist at the same nest depth. The second parameter can be either `different` or `same`, and specifies whether the trip count is the same across all n loops or not. The third parameter, loop identifier ip, is called the insertion point. It signifies where the fused loop should be inserted, which often happens to be the last, syntactically, loop in the l1, ..., ln list. If `different` is specified in the second parameter, then the n loops are fused into a single loop lnew with a trip count set to the maximum number of iterations that all n loops can safely execute, and n additional loops, l1′, ..., ln′ are inserted immediately after lnew for executing the remaining iterations. The examples in Figure 12 show the effects that specifies `same` and `different` have in the fusion process.

**H. Nest**

Subject to DESCRIPTOR, nest places a loop l in under a new loop as shown in Figure 13. DESCRIPTOR can be one of following configurations:

- `simple(IV, LB, UB, S)` where IV is a string, and LB, UB and S are either integer literals or strings, the original loop l is placed under a new loop "do IV = LB, UB, S". 
The specialize transformation is used to perform conditional or unconditional constant propagation. We can specify which loops we want to propagate constants to in the first parameter, then specify either conditional (true) or unconditional (false) constant propagation in the second parameter. The default specification is a conditional constant propagation and the example is shown in Figure 15.

Fig. 15. The specialize transformation demonstrating conditional constant propagation.

K. Fission

Figure 16 shows an example of using the fission transformation. The first parameter (1) specifies which loop to apply fission to, while the second parameter specifies the bookmarks: a list of same level loops, that are nested under 1 and that splitting will happen near which. The third parameter is the list of identifiers for the new loops that are going to be created (one for each bookmark). If the target loop’s body consisted entirely of n loop statements and these statements also happened to comprise the bookmarks list, then we would have ended with n new loops. However, since not every statement under the target loop is necessarily a loop statement, or just because we may not want each loop to act as a bookmark, we need to indicate what happens to these statements with respect to the new loop creation process. The third parameter is a list of specifiers as long as the bookmarks list, where each specifier takes one of the following values: none, before, after and both. Loop 1’s body will contain n statements, $S_1, ..., S_n$ at the top level, which is also a superset of the bookmarks list. HSLOT maintains a queue of encountered statements. Non-bookmark statements are appended to the queue and whenever a bookmark $M$ is met, the relative bookmark specifier will dictate what action is taken, summarized below as follows.

- none: the bookmark $M$ is enqueued; no loop is created.
- before: create a new loop with whatever is in the queue; clear the queue and append $M$.
- after: add $M$ to the queue, then create a new loop with whatever has been gathered in the queue; clear the queue.
- both: create a new loop with whatever is in the queue; clear the queue and create a new loop containing $M$ only; clear the queue.

If the queue is non-empty after all statement have been processed, the statements become part of a new loop. Let us elaborate on the example in Figure 16. There are three
statements: the assignment to \( DX(I) \) and the two loops \( L1 \) and \( L2 \). HSLOT starts with an empty queue and the first statement it encounters gets appended to the queue because it is not a bookmark. Then \( L1 \) is met; its specifier is \( \text{both} \). HSLOT places the accumulated statements, which comprises \( DX(I) = \ldots \) only, under a new loop, then places \( L1 \) under its own loop (\( L11 \)), and finally clears the queue. Then it encounters \( L2 \) – a bookmark too. Because the action is none, the loop gets appended to the queue. Now that all the statements have been processed, HSLOT examines the queue, and because it is not empty, it places its contents under a new loop (\( L12 \)).

L. Swap Nest

The \text{swnest} transformation is an auxiliary transformation that helps with reordering loops in a loop stack. As indicated by Figure 17, the loop trip count expressions ”\text{DO } I=\ldots” and ”\text{DO } J=\ldots” are literally swapped, and all statements (blocks 1 and 3) of the outer loop (\( LX \)) are moved into the body of the inner loop (\( LY \)).

This is different from a mere interchange (transformation \text{interch}), which only swaps the loop trip count expressions. Figure 18 highlights the differences in the two when combined with tiling: Figure 18 shows how to optimize 2MM. Our strategy for this kernel is to peel the first iteration off the first loop then fuse with the second loop. If we simply fuse the two loops without peeling, we will encounter data dependency problems and the kernel will not produce the correct output. After we fuse the two loops together, we openmp-parallelized the resulting loop. In this figure, we show the transformed code at the end of each step as well. Each transformed code is readable to users, and we can compile and execute them as well. In a similar fashion, we also optimized \text{Jacobi-2D} computation as shown in Figure 20. An additional optimization that we performed was the unrolling of the outer loop to read more columns before fuse. We continued with a strategy similar to that chosen for the 2MM kernel. By performing these transformations, we achieved 4.0× speedup for 2MM, and 2.2× speedup for \text{Jacobi-2D} over −0.3 in openf90.

IV. EXAMPLES

This section demonstrates how the different directives we described in Section III can be combined for optimizing two different computational kernels.

We optimized two scientific kernels: 2MM (Two Matrix Multiplications) and a \text{Jacobi-2D} stencil computation. Figure 19 shows how to optimize 2MM. Our strategy for this kernel is to peel the first iteration off the first loop then fuse with the second loop. If we simply fuse the two loops without peeling, we will encounter data dependency problems and the kernel will not produce the correct output. After we fuse the two loops together, we openmp-parallelized the resulting loop. In this figure, we show the transformed code at the end of each step as well. Each transformed code is readable to users, and we can compile and execute them as well. In a similar fashion, we also optimized \text{Jacobi-2D} computation as shown in Figure 20. An additional optimization that we performed was the unrolling of the outer loop to read more columns before fuse. We continued with a strategy similar to that chosen for the 2MM kernel. By performing these transformations, we achieved 4.0× speedup for 2MM, and 2.2× speedup for \text{Jacobi-2D} over −0.3 in openf90.

V. EXPERIMENTAL METHODOLOGY

This section describes the setup including hardware, and benchmarks used for our experiment. We used a dual Intel Xeon E5-2650 system, which comprises 2 chips with 8/16 core/threads each, for a total of 32 HW threads; each chip also features a 20MB Intel smart cache.

We optimized 11 scientific programs written in Fortran taken from Polybench v2.1 [12] with standard problem
size. We applied a set of transformations to improve data reuse and reveal more parallelism for these programs and then compared the performance improvements to openf90 -O3 – our baseline. HSLF90 was configured with -O3 too, in order to test whether HSLOT transformations can coexist with the compiler’s default optimization strategies.

A. Experiment

We optimized a set of programs using HSLOT and the results that we achieved for the best of our optimization strategies compared to -O3 in openf90 are given in Table II. The main strategy of applying transformations is to improve data reuse or remove dependencies so that we expose more parallelism or enable time skew (e.g. fuse two loops that are not fusible originally, slightly skew the original loop then fuse them). For the transformation using OpenMP, we used the five different HW thread count - 2, 4, 8, 16, and 32.

The PolyBench programs we transformed are shown in Table II, which also cites the different directives employed in each case as well as the speedup achieved. Note that due to space limitations the last column mentions the transformations used in each program rather than the exact transformation sequence and actual parameters. We also provide the number of times that a particular transformation was employed in the same program.

Program 2mm has two matrix multiplications having dependencies between two loops. We peeled the first iteration of the first loop to remove these dependencies and then fused two loops. This gave us a 1.2× speedup over -O3. Using parallelization with 32 threads led to additional improvements and the achievement of a 4.0× speedup over -O3. Program 3mm has a structure very similar to 2mm but also features an additional matrix multiplication, so we have three triple-nested loops in total. We applied the same set of transformations that we used for 2mm for the first two triple-nested loops and achieved 1.15× for a sequential version and 2.1× with 32 threads over -O3. We did not fuse the last loop because peeling the first iteration of the second loop makes the fusion of the first loop and the second loop illegal. We also applied the same strategy to stencil computational kernels - ffdt-2D, jacobi-1D and jacobii-2D and achieved speedups of 1.1×, 1.1× and 2.2× respectively.

For dynprog, gemm, and seidel-2D, we used the strategy of unrolling the outer loop then merging the inner loops to access more columns in each iteration. We unrolled the outer loops instead of the inner loops to access more columns rather than rows since Fortran uses column-major array layout. We achieved performance speedups of 1.2×, 1.1× and 1.1× respectively. In each case we parallelized one of the fused loops and achieved speedups of 1.2× for dynprog, 14.6× for gemm, and 27.0× for seidel-2D using 32 threads. We applied a relatively simple combination of transformations for mvt and doitgen using the unroll and OpenMP parallelization, which yielded speedups of 8.3× and 16.5× respectively with 32 threads.

The transformations we applied to correlation was different from the transformations we applied to the other 10 programs. We applied fission to one of the loops in order to create a perfectly nested loop, then we unrolled the outer loop and finally fused the inner loops. However, we could not achieve good performance using this set of transformations. Although we transformed part of one loop, we could not transform rest of the loops efficiently because of heavy dependencies between loops.

In overall, we achieved an average serial speedup of 1.2×, while applying the openmp transformation for certain programs, resulted in additional average speedups of up to 3.5× with 32 threads.

VI. RELATED WORK

OpenHMPP [7] is a directives API that users can employ to quickly port their codes to accelerators, such as GPUs, while remaining target-agnostic as the accelerators are an active area of research. They also provide a limited set of loop transformations for optimizing code and mostly for mapping parallelism to the right unit (e.g. a mid loop to the y-th CUDA grid dimension). However HSLOT provides a larger set of transformations than OpenHMPP.

Another similar related work is the CHiLL framework [13]. CHiLL is a polyhedral loop transformation based framework and allows users to specify a set of high-level loop transformations in a script to process them with a given loop identifiers (the target loops). HSLOT is implemented in Open64 VHO level and also provides a standard loop transformations and a unique set of transformations that CHiLL does not provide such as peel, swnest, fork, and so on. There are also some transformations in CHiLL, such as datacopy, that HSLOT does not implement. However, these are not loop transformations and thus remain the subject of future work.

Marker et al. [14] proposed a way for encoding domain expert knowledge and automatically generating optimized code for dense linear algebra algorithms on distributed-memory systems as opposed to hand-tuning. HSLOT follows a similar approach in terms of allowing users (or experts) to specify their knowledge of optimizing a code using a sequence of directives instead of hand tuning the code.

Franchetti et al. [15] presents a rewriting system to automatically vectorize signal transform algorithms at a high-level mathematical abstract level. They incorporated their rewriting system into the Spiral code generator to automatically generate a vector code. HSLOT performs more wider range of loop transformations rather than only vectorization. HSLOT does not support vectorization currently but can be extended to generate a vectorized code as well in the future work.

Bernhloodt et al. [16] introduces the KNOT system providing three components to annotate a block of code, specify a transformable section of code, transform it, and perform optimization/code generation. Sottile et al. [17] uses a rewriting approach for deriving transformations by examining code before and after the transformation. Their systems focus on the mechanics of term rewriting and its utilization as a vehicle for implementing custom transformations while we focus more on identifying, and directly implementing in HSLOT, the transformations that users may find practical.

Bodin et al. [18] presents a new program transformation infrastructure named TSF and its transformation script system.
TABLE II. This table summarizes the performance improvements and the list of transformations used for each program we evaluated. The baseline for the speedup is `$-O3$ in `openmp`. The performance for programs transformed using the `openmp` parallelization is obtained by comparing them against the baseline, which is serial; for scalability analysis purposes we also show results for the following thread counts: 2, 4, 8, 16, and 32.

<table>
<thead>
<tr>
<th>Program</th>
<th>Problem Size</th>
<th>Serial Speedup</th>
<th>Speedup with Different Thread Numbers</th>
<th>Set of Transformations Used with Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm</td>
<td>NL,NJ,NK,NL=1024</td>
<td>1.21×</td>
<td>0.62×</td>
<td>1.16×</td>
</tr>
<tr>
<td>3mm</td>
<td>NL,NJ,NK,NL,NM=1024</td>
<td>1.15×</td>
<td>0.48×</td>
<td>0.85×</td>
</tr>
<tr>
<td>correlation</td>
<td>N,M=102×</td>
<td>0.6×</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>dolgen</td>
<td>NQ,NK,NP=128</td>
<td>1.04×</td>
<td>2.1×</td>
<td>4.26×</td>
</tr>
<tr>
<td>dynprog</td>
<td>LEN=50, TSTEPS=10K</td>
<td>1.17×</td>
<td>1.19×</td>
<td>1.66×</td>
</tr>
<tr>
<td>ffdl-2D</td>
<td>NX,NX=10000</td>
<td>1.1×</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>gomm</td>
<td>NL,NJ=1024</td>
<td>1.05×</td>
<td>1.65×</td>
<td>3.28×</td>
</tr>
<tr>
<td>jacob1-1D</td>
<td>N=100000, TSTEPS=100</td>
<td>1.1×</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>jacob2-2D</td>
<td>N=90000, TSTEPS=10</td>
<td>1.32×</td>
<td>1.5×</td>
<td>2.95×</td>
</tr>
<tr>
<td>mvt</td>
<td>N=4000</td>
<td>2.48×</td>
<td>1.22×</td>
<td>2.38×</td>
</tr>
<tr>
<td>seidel-2D</td>
<td>N=32000, TSTEPS=4</td>
<td>1.06×</td>
<td>1.86×</td>
<td>3.71×</td>
</tr>
<tr>
<td>GMEAN</td>
<td>1.15×</td>
<td>1.11×</td>
<td>1.74×</td>
<td>2.47×</td>
</tr>
</tbody>
</table>

TSF is built on the top of FORESYS (Fortran Engineering System) and the script system acts as an interface between external tools and FORESYS while assisting users in constructing simple program transformations. TSF does provide a number of loop transformations, such as loop normalization, but not as many as HSLOT.

PoCC [19] is a source-to-source C compiler framework supporting polyhedral based loop transformations including parallelization and vectorization (via PLUTO [20]) as well. The loop transformations supported by PoCC and HSLOT are slightly different. HSLOT provides a greater variety of loop transformations while some in PoCC are not supported by HSLOT, such as vectorization. Another difference between HSLOT and PoCC is the manner by which transformations are applied. PoCC applies loop transformations on a per-SCOP (Static Control Part) granularity (a `#pragma`-enclosed region) while the order of transformations cannot be set by users. HSLOT allows users to tag loops and optimize them individually in the desired order.

VII. Conclusions

In this paper, we introduced a directive-based loop transformation system called HSLOT. HSLOT provides a rich set of high-level loop transformations including standard loop transformations and a unique set that are not found in other systems as well. HSLOT enables loop transformations that the compiler would otherwise be hesitant in performing due to dependencies that the user (domain expert) easily demystifies.

We optimized a set of scientific computation kernels from PolyBench, and optimized each of them achieving speedups in the range $1.1 \times - 26.9 \times$. Most importantly, we did not have to manipulate our sources in any way (modify, isolate or normalize) for HSLOT as inserting the directives sufficed.

Our results also showed that HSLOT transformations generally coexist well with the `openf90` compiler’s `-O3` optimizations. We still had a program achieving a poor performance because we could not find the right transformations for the program. This is one of the well known challenges in optimization and can be improved by involving more analysis or machine learning techniques as well. Nonetheless, HSLOT enabled testing an intricate transformation with minimal programming effort.

In the future, we plan to implement a macro-like functionality for defining new transformations in terms of existing ones and making the new primitive available at the directives API level. For example, `funroll2(A,B)` can be defined as `"funroll2(A,B)\"` and invoked as `#hslot funroll2(L1,L2)` for two loops $L_1$ and $L_2$.

ACKNOWLEDGMENTS

The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-000R22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-000R22725.

REFERENCES


Fig. 20. Jacobi-2D stencil computation.