Identifying and Scheduling Loop Chains Using Directives

Ian J. Bertolacci
University of Arizona
Tucson, Arizona
Email: ianbertolacci@cs.arizona.edu

Michelle Mills Strout
University of Arizona
Tucson, Arizona
Email: mstrout@cs.arizona.edu

Stephen Guzik
Colorado State University
Fort Collins, Colorado
Email: Stephen.Guzik@colostate.edu

Jordan Riley
Colorado State University
Fort Collins, Colorado
Email: jriley2@rams.colostate.edu

Catherine Olschanowsky
Boise State University
Boise, Idaho
Email: catherineolschan@boisestate.edu

***Abstract*—Exposing opportunities for parallelization while explicitly managing data locality is the primary challenge to porting and optimizing existing computational science simulation codes to improve performance and accuracy. OpenMP provides many mechanisms for expressing parallelism, but it primarily remains the programmer's responsibility to group computations to improve data locality. The loopchain abstraction, where data access patterns are included with the specification of parallel loops, provides compilers with sufficient information to automate the parallelism versus data locality tradeoff. In this paper, we present a loop chain pragma and an extension to the `omp for` to enable the specification of loop chains and high-level specifications of schedules on loop chains. We show example usage of the extensions, describe their implementation, and show preliminary performance results for some simple examples.**

1. Introduction

Several programming models, languages, and abstractions have been devised to expose parallelism in applications including OpenMP [1], OpenCL [2], and OpenACC [3]. This has led to a data parallel programming style used in many large production codes where the software is "modularized" into series of parallel and reduction loops. The problem is that exploiting all possible parallelism without regard to data locality leads to insufficient arithmetic intensity and excessive memory traffic. Therefore this paper presents a set of OpenMP pragma extensions that provide a way to specify data access information and high-level schedules that incorporate both data locality and parallelism to the compiler.

Data locality and arithmetic intensity are essential to execution performance. It is important for a programming model to expose parallelism, but the reality that the memory bandwidth bottleneck is often the main obstacle to performance suggests that expressing data reuse is just as important as expressing parallelism. While existing and emerging parallel programming models provide many abstractions for exposing parallelism, abstractions for expressing locality tend to require significant work from the programmer and can lead to performance portability issues based on how well the programmer parameterizes the data and/or computation aggregation decisions they must make.

The main limitations of previous work for expressing data locality are that (1) the programmer has to aggregate computations into tasks, (2) tasks are limited to groupings of iterations within a single loop or user-defined functions, and/or (3) the programmer has to rewrite full computations in another programming model. The principal advantage of the loop chain pragmas proposed here is that they can be added to legacy applications; meaning that only the high-level scheduling directives need to be adjusted for efficient execution on various hardware.

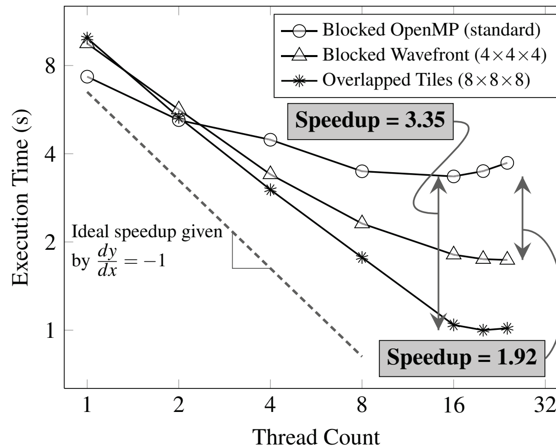
The *loop chain* abstraction represents a sequence of parallel and/or reduction loops that explicitly share data [4]. Such coding patterns are often found in stencil codes, and other kinds of buffered producer/consumer codes, which we target in particular. The loop chain abstraction requires that each loop in the chain is parallel or a reduction (typically an array reduction), has a well-defined domain, and has well-defined access functions that indicate how each iteration accesses data spaces. With these requirements, the loop chain abstraction can be used to derive a partially ordered set of iterations that makes scheduling and determining data distributions across loops possible for a compiler and/or runtime system. The flexibility to schedule across loops enables better management of the data locality and parallelism tradeoff.

Providing data access information that enables the compiler to determine dependencies and high-level schedule information in pragmas enables domain scientists to incrementally parallelize large production codes. The pragmas

978-1-5090-6152-5/16 $31.00 © 2016 IEEE
DOI 10.1109/WACCPD.2016.5

```
1   const int D = 2;
2   Real lhs[numCells];
3   Real Fx[numFaces];
4   Real Fy[numFaces];
5   ...
6   for (i=0; i<szx+1; i++)
7     for (j=0; j<szy; j++)
8       Fx[i][j] = KERNEL1(...);
9
10  for (i=0; i<szx; i++)
11    for (j=0; j<szy; j++)
12      lhs[i][j]  = KERNEL2(Fx[i][j],
13                           Fx[i+1][j]);
14  for (i=0; i<szx; i++)
15    for (j=0; j<szy+1; j++)
16      Fy[i][j] = KERNEL1(...);
17
18  for (i=0; i<szx; i++)
19    for (j=0; j<szy; j++)
20      lhs[i][j] += KERNEL2(Fy[i][j],
21                           Fy[i][j+1]);
```

(a) Pseudocode for the evaluation of diffusive fluxes in a computational combustion application. cells and their faces.



(b) In this example, shifting the F_x and F_y loops, fusing all the loops, and then using overlapped tiles result in the best performance. Results were collected on a single Cray XT6m node (AMD Magny Cours).

Figure 1. To improve performance for the pseudocode in (a), a standard technique is to use OpenMP to do blocking within each loop. With loop chain information, schedules that group iterations across loops are possible and this results in better performance than just blocking with OpenMP.

specify the loop chain abstraction and schedules for loop chains, which were developed to navigate the tradeoff between data locality and parallelism while requiring minimal extra information from programmers.

The current implementation focuses on expressing schedules that balance data locality and parallelism for shared-memory multicore architectures. However, the information provided could be used to automate high-level schedule specification beneficial to accelerators as well. For example, Grosser et al. [5] demonstrated the performance advantages of split tiling for stencil codes on GPUs. Additionally, they presented a mechanism for automatic code generation of split tiling code. While currently beyond the scope of this work, this is precisely the type of transformation that is a candidate for inclusion in loop chains.

Manual implementations of the transformations have demonstrated their potential impact. In a previous paper, we manually applied the loop chain abstraction and explored the tradeoffs between parallelism and data locality by employing different loop chain scheduling strategies [6]. Figure 1(a) illustrates pseudocode for a small subset of the fourth-order evaluation of diffusive fluxes in a computational combustion application. There are four loops nests that share data and that we consider to be part of a loop chain. Figure 1(b) shows the execution times for three different scheduling approaches. We found schedules enabled by loop chains performed as much as $3.35\times$ faster than the original blocked OpenMP version resulting from a data parallel programming style that primarily expresses parallelism.

The specific contributions of this work include

- a pragma grammar to specify loop chains and their schedules,
- examples of how a user would annotate existing code with the pragmas,
- a prototype source-to-source translator that implements the pragmas, and
- a discussion of the current limitations in the implementation.

Our preliminary results indicate that this programming abstraction can serve as a useful tool for developers and maintainers seeking to improve the performance of their application without having to overhaul their code.

2. Specifying Loop Chains

In this work we provide developers with a set of compiler pragmas that can be inserted into application code. This enables incremental changes to be made to existing applications. For the purpose of this paper we use the pragma label *omp-lc*, indicating that the loop chain transformations have the potential for inclustion with the OpenMP standard. See Figure 5 for the full LoopChain directive grammar.

2.1. Loop Chain Directives

The loop chain directives need to communicate three categories of information to the compiler: (1) The code segment that contains all of the loop nests that should be considered part of the loop chain, (2) the domain and access patterns of each of the candidate loop nests, and (3) the transformation recipe to be applied to the loop chain. This information is passed using two different directive tags, *loopchain* and *for*.

```
#pragma omp-lc loopchain \
schedule( ... )
{
  #pragma omp-lc for domain(lb:ub) \
    with (i) write A {(i)}, \
             read  B {(i), \
                      (i-1), \
                      (i+1)}
  for( int i = lb; i <= ub; i += 1 )
    A[i] = (B[i-1] + B[i] + B[i+1])


  #pragma omp-lc for domain(lb:ub) \
    with (i) write A {(i)}, \
             read  A {(i)}
  for( int i = lb; i <= ub; i += 1 )
    A[i] = A[i] * (1.0 / 3.0)
}
```

Figure 2. Example of annotated source code (schedule hidden)

```
for( int i = lb; i <= ub; i += 1 ){
    A[i] = (B[i-1] + B[i] + B[i+1])
    A[i] = A[i] * (1.0 / 3.0)
}
```

Figure 3. Expected form of transformed code after loop fusion (*schedule(fuse)*)

The for tag is placed before each loop nest and captures the iteration domain of the loop nest and the data access patterns for data objects that are live among loop nests. The *loopchain* tag communicates the beginning and end of the code segment that should be considered for scheduling and the schedule that should be applied.

Figure 2 shows an annotated input to the source-to-source translator. Figures 3 and 4 show two possible outputs depending on the choice of schedule, the first is a fuse and the second is tiling.

2.2. Domains and Access Patterns

The domain and access pattern annotations provide key information about the iteration space of a loop nest and the accesses to data made within that space. Each nest in the chain has its own domain and access pattern annotation. The grammar specification shown in Figure 5 contains the formal grammar definitions of both *nest domain definition* and *access definition*. It is worth noting that these annotations only inform the scheduling process, and do not themselves modify or impart a scheduling on the nests they describe.

Specifying Domains. The domain clause withing the *for* tag of the pragma takes form:

```
#pragma omp parallel for
for (int tile = floord(lb,10);
         tile <= floord(ub,10);
         tile = tile + 1) {
  for (int i = max(10 * tile,lb);
           i <= min(10 * tile + 9,ub);
           i = i + 1) {
    A[i] = B[i - 1] + B[i] + B[i + 1];
    A[i] = A[i] * (1.0 / 3.0);
  }
}
```

Figure 4. Expected form of transformed code after loop fusion and tiling by 10 (*schedule(fuse, tile((10), parallel, serial))*)

```
domain (d1_lb:d1_ub,d2_lb...)
```

The domain of a loop nest is specified as a list of inclusive ranges representing the upper and lower bound of each dimension of the loop nest. The domain may be of lower dimension than the loop nest. In this case the domain can specify a subset of the actual domain that should be used for transformation. An N dimension loop nest defined by a N-k dimension domain, describes the N-k outer loop nests with the N-k+1 inner loop nests as the statement body.

While a domain can be retrieved through code analysis it may be the case that the syntax of the loop nest does not reflect the domain of interest for loop chain scheduling. For example, the inner most loop(s) may be considered to be the body of the outer most loop(s), such as when iterating within the components of an array structure. This is common in computation fluid dynamics codes, where various physical components are stored at each mesh point in an array.

Specifying Access Patterns. The data access pattern is expressed as a mapping between the iteration space and the accesses into the data space. The map can either be expressed as a read or a write access, depending on the action taken in the code.

The data access pattern specification follows the domain specification leading with the keyword *with*, and an ordered list of the loop nest iterators. The data access clause takes the following form, where f and g are expressions using the available loop iterators:

```
with (i,j,...) read ID{(f(i,j,...)),
                       (g(i,j,...)),...}
               read ID2{ ... }
               write ID3{ ... }
               ...
```

In Figure 2 the access pattern for the first loop states that the iteration i writes to the data space A using i, and that it reads into the data space B using $i-1$, i, and $i+1$. This example illustrates a program where the access pattern is obvious and would be easy to identify through program analysis. However, unlike the domain, program analysis will

not catch all access patterns of interest. In the following example:

```
double* ptr_1 = buffer + mk_offset( ... );
double* ptr_2 = buffer + mk_offset( ... );
```

it is impossible at compile time to know if `ptr_1` and `ptr_2` are the same or might result in overlapping accesses. For this reason, we have the programmer specify the access pattern explicitly.

2.3. Scheduling Loop-Chains

The *loopchain* tag, in addition to indicating the loop chain, communicates the scheduling transformations to be applied to the chain as a whole.

The transformations performed on an application benchmark, mini-flux-div [6], motivated our choice of loop transformations. Transformations that are currently implemented in our prototype tools are *fuse* and *tile*. Additional transformations are currently under development. The following is a short description of each transformation.

Specifying Schedule Operations. Currently there are 5 schedule operations included in the directive grammar: *serial*, *parallel*, *fuse*, *wavefront*, and *tile*. Syntactically, the schedule directive is a list of these schedule operations in the order they are to be applied. The formal grammar for this portion of the directive can be found in the *loopchain annotation* production in Figure 5.

Semantics of Schedule Operations.
fuse: All of the loops in the loop chain are fused using a loop fusion transformation. Shifts and loop peeling are done as necessary before the loop fusion to respect data dependencies. The original order of the loops within the loop chain is the order of statements is in the fused loop body.
tile: The tile scheduling function takes parameters for the tile size, the schedule over the tiles, and the schedule within the tiles. The full grammar for this operation can be found in the *schedule atom* production in Figure 5. Currently, only constant sized tiles are supported.

The size of the desired tiles must be provided for each dimension of the tile. Providing, *tile(16)*, will tile the outer loop of a loop nest in a single dimension with 16 iterations in each tile. Specifying, *tile(16,16,16)*, will create cubic tiles of size 16^3. The dimensionality of the tiling specified cannot exceed that of the domain provided to the for loop pragma.
wavefront: The inner loops of nest are skewed as necessary as a function of the outer loops and parallelized.
serial: Typically only used in the context of tiling, indicates that either the outer loop over tiles or within tiles should not be parallelized.
parallel: Typically only used in the context of tiling, indicates parallelizing the outer loop over tiles or the outer loop within a tile.

3. Compiler Infrastructure

Two main components comprise the code transformation infrastructure. The first is an internal representation specifically designed to represent and support loop chain transformation, (LoopChainIR). The second is an prototype implementation of a transformation pass written in the Rose compiler framework [7] that utilizes the LoopChain directives and LoopChainIR to perform loop chain transformations. The transformation has been designed as a single, discrete pass that could be used in composition with other transformation passes within a compiler.

3.1. LoopChainIR

LoopChainIR is a C++ library that provides abstractions, as classes, of loop nests (LoopNest), loop domains (RectangularDomain), loop chains (LoopChain), code generation ready schedules (Schedule), and transformations to apply these schedules (Transformation hierarchy). Figure 6 shows the structure of the current LoopChainIR library.

LoopChains are formed by an ordered list of LoopNests, which consist of rectangular domains.The Schedule class takes a LoopChain object and creates a mathematical representation of the domains. A Schedule object can then have transformations (in the form of Transformation objects) applied to it. At any time (even before or in between transformations) the Schedule object can generate code.

This library seeks to provide compiler developers with simple abstractions with which to represent and manipulate the loop chain, while hiding the mathematical specification of the loop chain and the transformations on it.

3.2. Integer Set Library

LoopChainIR utilizes the Integer Set library (ISL) [8], [9], [10] for code transformation and generation. When a LoopChainIR Schedule is created it creates domains in ISL representation. When the LoopChainIR Transformations are applied, they form an ordered list of transformation functions in ISL representation. When the LoopChainIR Schedule performs code generation, ISL applies those transformation functions to the union of those domains, and outputs ISL's AST representing C code.

The loop chain from Figure 2 would be represented to ISL as the following domains

```
[lb,ub]->{ stmt_0[i] : lb <= i <= ub }
[lb,ub]->{ stmt_1[i] : lb <= i <= ub }
```

All loop chains are first 'transformed' to embed their domains in a common iteration space. This is the function for Figure 2.

```
stmt_0[i] -> [0, i, 0];
stmt_1[i] -> [1, i, 0];
```

The first index is the loop ordering (0'th and 1'st loop), and the last index is the statement's position within the loop

⟨nest annotation⟩ → **for** ⟨nest domain definition⟩ ⟨access definition⟩
⟨nest domain definition⟩ → **domain (** ⟨expression⟩ **:** ⟨expression⟩ (**,** ⟨expression⟩ **:** ⟨expression⟩)* **)**
⟨access definition⟩ → **with (** ⟨id⟩ (**,** ⟨id⟩)* **)** ⟨access atom⟩ (**,** ⟨access atom⟩)*
⟨access atom⟩ → (**read** | **write**) ⟨id⟩ { ⟨iterator expression⟩ (**,** ⟨iterator expression⟩)* }
⟨iterator expression⟩ → **(** ⟨expression⟩ (**,** ⟨expression⟩)* **)**

⟨loopchain annotation⟩ → **loopchain schedule (** (⟨schedule atom⟩ (**,** ⟨schedule atom⟩)*)? **)**
⟨schedule atom⟩ → **serial** | **parallel** | **wavefront** | **fuse**
| **tile ((** ⟨int⟩ (**,** ⟨int⟩)* **) ,** ⟨schedule atom⟩ **,** ⟨schedule atom⟩ **)**

Figure 5. Full LoopChain directive grammar

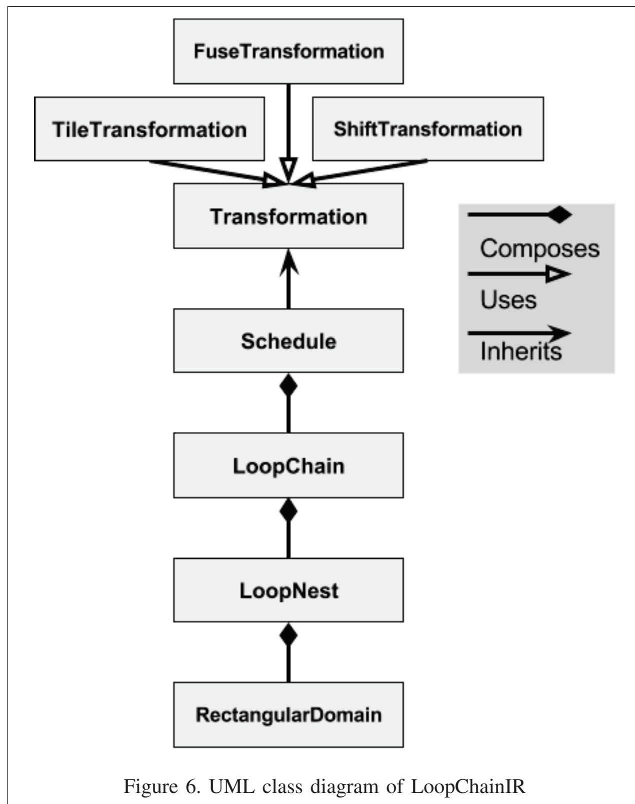


Figure 6. UML class diagram of LoopChainIR

body. This gives the statement inside a loop nest a position within the whole loop-chain iteration space.

A function can have conditional mapping. For example, a fusion between two specific loops (1 and 2 in this hypothetical case), that leaves other loops unchanged:

```
[src, i, stmt] -> [tgt, i, src] :
 (src = 1 or src = 2)
  and tgt = min(1, 2);
[src, i, stmt] -> [src, i, stmt] :
  !(src = 1 or src = 2);
```

Without the second mapping, all other source loops where $!(source = 1 \; or \; source = 2)$ are dropped from the loop chain.

```
// Parse source into Sage AST
SgProject* project = frontend( argv,
                               argc );

// Call the LoopChain transformation
TransformLoopChains( project );

// Generate code instead of
//executable binary
project->skipfinalCompileStep( true );

// Generate source code from AST
backend( project );
```

Figure 7. Calling the LoopChain transformation pass

3.3. LoopChain Transformation Pass

The transformation pass is implemented as a visitor in the Rose framework and performs the the actions required to go from LoopChain directives to transformed and integrated code. The call to transform source code based on loop chain pragmas can be inserted at any point during rose compilation when the Sage AST is stable. Figure 7 shows the interface that triggers the transformations. The transformation results in a valid AST and, therefore, can be followed by additional transformations if desired. Figure 8 shows a larger overview of process from front to back. Figure 9 gives an overview of the LoopChain transformation process and the flow of different ASTs an placement of various components. The steps are numbered here to indicate their relationship to Figure 9.

The first step (1) is to parse the LoopChain directives. Parsing the `nest annotations` produces a series of LoopNest objects, with associated iterator symbols and loop bodies, while parsing the `loopchain annotations` collects those LoopNest objects into a single LoopChain object, and retrieves the scheduling directives.

The next step (2) is to take the LoopChain object and the scheduling directives, and generate code. Once the scheduling directives are brought together with the entire loop nest, a series of LoopChainIR Transformation objects can be cre-

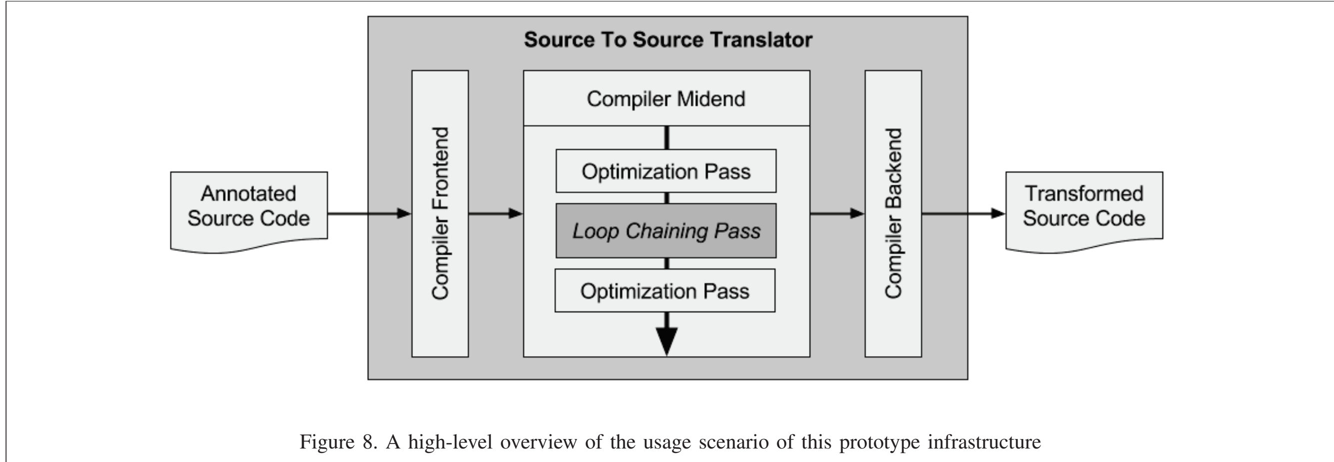


Figure 8. A high-level overview of the usage scenario of this prototype infrastructure

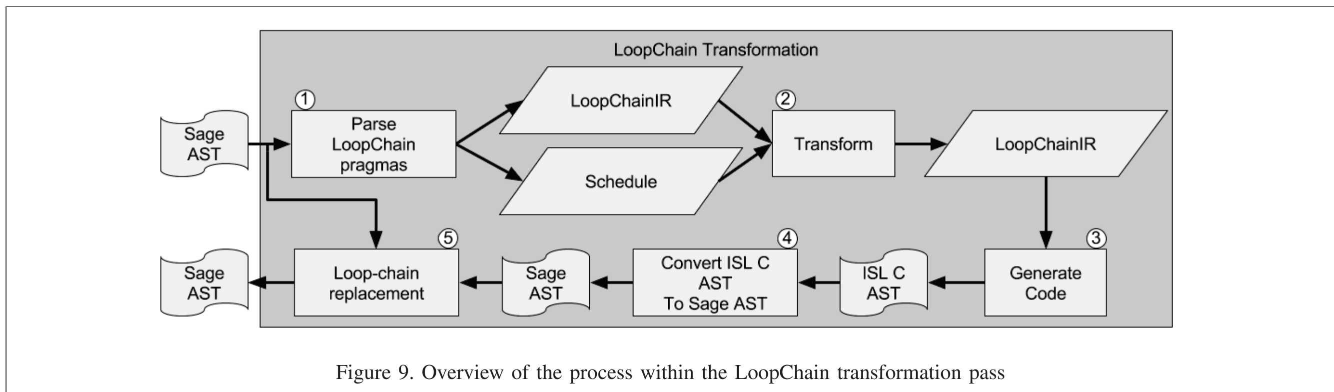


Figure 9. Overview of the process within the LoopChain transformation pass

ated. A Schedule object is constructed from the LoopChain, and the Transformation objects are added to it. The Schedule object then applies the transformation functions (supplied by the Transformation objects) over the domains (specified by the LoopChain and its LoopNests). Now ISL can generate its C AST representing the transformed loop chain (3).

The last step is to take that AST, and inject it back into the source AST. The ISL To Sage translator (described in Section 3.4) is responsible for the translation of ISL's C AST into Sage AST and injecting the resulting dangling tree back into the original AST (4). The transformer then maps the new iterator expression to their original iterator symbols, and replaces the statement macros from ISL with the original loop bodies. The original loop chain code is then removed from the AST entirely, becoming replaced by the new, transformed code (5).

The transformation does this for all annotated loop chains in the input code. Once all annotated loop chains have been transformed, the compiler can continue applying other optimization and transformation passes.

3.4. ISL To Sage Translation

An critical component of the LoopChain Transformation pass is the translation of ISL's C AST into Rose's Sage AST. This is performed by a simple recursive traversal of the ISL AST, and creating an image of it using the Rose SageBuilder tools.

Importantly, in addition to the ISL AST, the ISL to Sage translator must also take in the parent scope to the site in the original Sage AST where the resulting new Sage AST will be injected. This is required to match symbols that are defined in or above the injection site that are referenced (in name only) by the ISL AST, such as loop bounds and other symbolic constants.

In addition to constructing the new Sage AST the ISL to Sage translator also keeps track of the Sage AST nodes it creates that are representations of the new loop bodies. Because ISL does not know what the loop bodies are when it creates its AST, it uses statement macros (`stmt_0(c1,c2)`, for example). These are collected and passed upward to ease their replacement by the original loop bodies later.

4. Preliminary Results

The performance (in speedup relative to the unmodified serial implementation) of two benchmarks, StreamTriad were recorded for the original code and the code output from

our transformation tool. The results are encouraging, demonstrating a performance improvement with some schedules.

4.1. Experimental Setup

Benchmarks. StreamTriad is a simple array addition with scalar multiplication benchmark: $(A_i + B_i) \cdot \alpha$. To view the effects of the transformations on this simple benchmark, we artificially inflated the reuse distance and created additional loops by splitting the addition and multiplication into two loops. Blur9 is the average of an element with its 8 surrounding neighbors: $A_{i,j} = (A_{i-1,j-1} + \cdots + A_{i,j} + \cdots + A_{i+1,j+1}) * (1/9)$. Again, we artificially inflated the reuse and created additional loops by splitting the addition and multiplication into two loops.

Compilers and Compiler Options. GCC g++ version 4.9.2 was used to compile all the benchmarks. The `-O3` flag was used to optimize.

Hardware. The test machines is single node, 2 socket Intel Xeon E5-2620 v2 @ 2.10GHz with 6 cores (12 threads), for a total of 12 cores (24 threads), with 32KB L1, 256KB L2, and 15360KB L3 caches, 2 NUMA domains, and 63GB of RAM.

Problem Size and Tiling Parameters. Double precision values were used in both benchmarks. The StreamTriad benchmark was tested on an input of $2 * 10^8$ (1.49GB) totalling $6 * 10^8$ FLOPS, and the blur benchmark was tested on an input of $(2 * 10^4 + 1)^2$ (2.98GB) computing $(2 * 10^4)^2$ values totalling $9 * (2 * 10^4)^2$ FLOPS.

Tile sizes for blur were chosen chosen to fit well within L1 (20x20 or 3.125KB), tightly within L1 (60x60 or 28.125KB), and outside L1 (120x120 or 112.5KB). We also compared parallelism over the tiles vs parallelism within the tiles.

4.2. Speedup Results

The serial runtime of the naive implementation for StreamTriad was 1.16 seconds ($1.17 * 10^8$ FLOPS/second), and for Blur9 it was 3.07 seconds ($1.72 * 10^9$ FLOPS/second). Figures 10(a) and 10(b) show speedup relative to the naive serial implementation of the StreamTriad and Blur9 benchmarks. We can clearly see that in both StreamTriad and Blur9 benchmarks, fused loops outperform the naive implementation, typically by an increase of 1x speedup.

In Blur9, tiling does not seem to improve performance over the naive implementation, and parallelism within a tile performs much worse that parallelism within a tile. However, this is probably an issue more to do with the tile size, with which we did not extensively experiment.

5. Related Work

Application optimization with an awareness of data locality is an active research area with a rich history. This section briefly highlights projects with similar aims and approaches. Loop chaining differentiates itself from most of the previous work by removing complex tasks from the domain of the user. Specifically, *task aggregation* requires users to rewrite large portions of the code. Loop chaining depends only on the user annotating existing code. Other *directive- or annotation-based approaches* do not provide a data access pattern interface. This interface allows the optimizing compiler to make decisions not always possible through data analysis. The same is true for *optimization scripting languages*. The advantage of loop chaining is that it hides this complexity from the user, making optimizing transformations more practical.

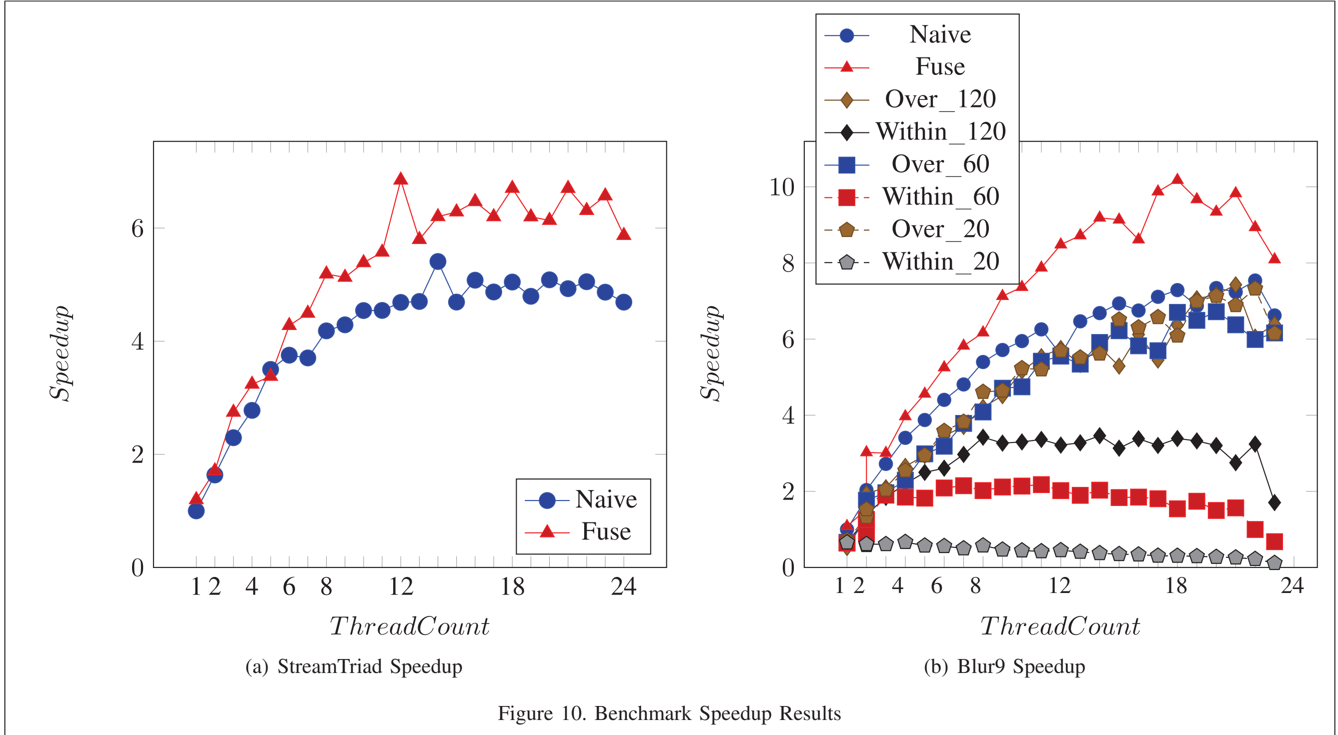
5.1. Improving Data Locality by Aggregating Computation into Tasks

Various approaches have been developed to navigate the tradeoff between parallelism and locality. We leverage the concept that developing a static aggregation or tiling strategy followed by dynamic execution of a task graph results in improved data locality within each tile and concurrency, load balancing, and memory latency tolerance between tiles. The key difference between previous work and loop chaining is that the programmer's responsibilities are less while still having feasible program analysis requirements.

The problem with having the programmer aggregate computations into tasks is that the programmer has to make some decision about task granularity across loops, and that decision might not be portable. There are various ways to aggregate computations into tasks: using an OpenMP pragma and specifying the chunk size [11], tiling the loop and having tile iterations be tasks [12], iteration space slicing [13], [14], and encapsulating tasks within functions that have parameters indicating the task granularity. The OmpSs work [15], [16] has the programmer indicate tasks by placing pragmas on C function definitions with in/out information about parameters and whether a function should be considered higher priority. Many new programming models [17], [18], [19], [20], [21], [22], [23] provide a task graph abstraction and suggest that programmers rewrite existing code with sequences of parallel loops in the form of task graphs instead. Iteration space slicing techniques [13], [14] that find sets of iterations across loops by doing transitive closure with data dependence relation information help automate task aggregation but depend heavily on precise and interprocedural data dependence analysis.

Once a task graph has been created, there are various ways of optimizing the performance of the task graph. In [24], the authors provide algorithms for scheduling task graphs using a mix of task and data parallelism. Within each task have data parallelism. In [25] the authors propose a new tag for OpenMP that allows the user to provide locality information through an affinity tag. Other work determines the data locality between threads when scheduling tasks and uses this to control thread affinity [26], [27], [28].

The loop chain abstraction can complement any and all of these approaches by providing the needed information for



(a) StreamTriad Speedup

(b) Blur9 Speedup

Figure 10. Benchmark Speedup Results

creating tasks to the compiler and then using the appropriate task graph based system as a backend. An issue we do not address in the proposed work is generating distributed memory code. Some aggregation approaches do provide distributed memory implementations [22], [29]. We are tackling the problem of providing effective shared memory parallelism for individual MPI processes.

5.2. Programmer Guided Code Transformation

The idea of providing optimization hints to the compiler through directives is not a new one. The Intel compiler (among others) offers a range of pragmas to aid in optimizing applications. For example, the following Fortran code uses the *loop count* directive.

```
!DIR\$ LOOP COUNT (10000)
do i =1,m
b(i) = a(i) +1
enddo
```

It is likely that with this information the compiler will schedule the code differently than it would without. The directives available through the Intel compiler that are most related to our work are the loop optimization pragmas: nofusion, unroll, and nounroll. There is not, however, a pragma available that will simplify the data flow analysis necessary for loop fusion and shifting in complex scientific applications.

Other frameworks have been developed that allow the programmer to apply more complex optimizations. Frameworks such as Orio [30] involve annotating the source code with instructions for optimizations. POET [31] and CHiLL [32] each provide a scripting language for optimization. The optimization scripts (or recipes) can be placed within the source code or associated with the source code from an external file.

6. Future Work

Gathering data using the current transformation infrastructure revealed several opportunities for improvement and expansion. The primary improvements to be made include altering the LoopChainIR data structure in order to improve the composability of transformations, integrating the use of data access information, parameterizing transformations, and expanding the set of transformations available.

Transformation Composability. It is often the case that a segment of code benefits from a series of transformations being performed. For instance, one may fuse two loop nests, causing a loss of parallelism by introducing loop carried dependencies, tile the fused code, and apply a wavefront to the result. During a transform, such as tiling the arity of the integer set representing the iteration space changes. The current LoopChainIR data structure does not contain the necessary information to perform further transformations after an arity change.

Unlocking Data Access Information. The expression of data access information is a key defining feature of loop chaining. This information will greatly expand the capabil-

```
for( int t = 1; t <= ub_T/2; ++t ){
#pragma omp-lc loopchain schedule(fuse)
  #pragma omp-lc for domain(lb_x,ub_x)\
    with (i) write A{(i)}, \
             read B{(i-1),(i),(i+1)};
  for( i = lb_x; i <= ub_x; ++i )
    A[i] = (B[i-1]+B[i]+B[i+1])/3;
  #pragma omp-lc for domain(lb_x,ub_x)\
    with (i) write B{(i)}, \
             read A{(i-1),(i),(i+1)}
  for( i = lb_x; i <= ub_x; ++i )
    B[i] = (A[i-1]+A[i]+A[i+1])/3;
}
```

Figure 11. Jacobi (1D) Annotated Source Code

```
for( int t = 1; t <= ub_T/2; ++t ){
  int i = lb_x-i;
  A[i] = (B[i-1]+B[i]+B[i+1])/3;
  #pragma omp parallel for
  for( i = lb_x; i <= ub_x-1; ++i ){
    A[i+1] = (B[i]+B[i+1]+B[i+2])/3;
    B[i] = (A[i-1]+A[i]+A[i+1])/3;
  }
  B[i] = (A[i-1]+A[i]+A[i+1])/3;
}
```

Figure 12. Jacobi (1D) Transformed Source Code

ities of the transformations. For instance, without this information, fusing subsequent loops in a stencil computation requires that the user input the shift information. It is our goal to remove that responsibility from the user and insert any shifts necessary for legal transformation.

Figure 11 illustrates the planned format for pragmas in this situation. The programmer simply identifies the domain, the access patterns which most programmers understand well, and states that the desired schedule is a fuse. The LoopChainIR software should use the access information to detect the need for a shift and add the shift to the fuse transformation.

Beyond this simple case the inclusion of access patterns enables a variety of other planned features. For instance, transformations requested by the user that result in breaking data dependencies or potentially introduce race conditions can be flagged and the user warned. We plan to warn the user and allow the transformation. This is because there are cases where introducing this type of issue is not wrong; a specific example of this is the Floyd-Warshall algorithm. A long-term goal is to use these access information to suggest transformations to the user as well.

Parameterizing Transformations. A significant amount of effort has gone into parameterizing code generation for transformations such as tiling. The foundation has been set to be able to move forward with this work, however, it is not supported in our current tool chain. A specific example of this is tiling. It is common to sweep through a set of tile sizes to determine the best performing configuration. However, with our current configuration each tile size needs to be determined at compilation time. It is preferable to change this at runtime.

Expansion of Available Transformation. Our initial development goal in this project was to get the entire path from annotated code to transformed code functional on only one or two transformations. With this achieved we were able to test the design and identify deficiencies in the design before investing a large amount of time in more complex transformations. We will continue to add transformations and update the infrastructure in response to our findings. The first set of transformations that will be added include all of those used in our previously published work [6] (shift, fuse, tile, wavefront, and overlapped tiling). We hope to expand that set in the future to include more advanced tiling including: diamond tiling, split tiling and overlapped tiling.

Interaction With Existing Optimizations. The current implementation of our work does not utilize other optimizations that are commonly implemented in compilers, such as unrolling and vectorization. It is important to understand how our work would interact with these existing optimizations. Specifically, we would like to know how they change the optimization process, and how they can affect code performance. Loop unrolling, for instance, exposes loop chains that could be optimized by our work, and we would expect to improve performance. However, loop chain optimizations could make vectorization optimizations more challenging, and we may lose the ability to perform that optimization. We would like to explore these interactions in detail, and have experimentally backed predictions of their effects.

7. Conclusions

There exist programming models, languages, and abstractions that can expose and exploit parallelism in applications. However, exploiting maximum parallelism without respecting data locality results in poor performance through excessive memory traffic. We have presented (1) a novel programming abstraction though OpenMP style pragmas, and (2) a software framework to describe and transform loop chains. These tools can provide developers of new applications, and maintainers of legacy applications, with the ability to identify and transform loop chains in order to increase arithmetic intensity by simultaneously increasing both parallelism *and* data locality.

Further, we have created a prototype code transformation pass, and used it to demonstrate the potential of these tools

to effectively transform simple benchmarks. Our performance results are encouraging. We believe that this programming abstraction can work reasonably well for developers looking to increase the performance of their application without requiring them to overhaul their codes.

8. Acknowledgments

This work was supported by a National Science Foundation grant NSF CCF-1422725.

References

[1] O. A. R. Board, "Openmp application program interface, version 4.0," July 2015. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[2] K. O. W. Group, "The opencl specification, version: 2.2," March 2016. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf

[3] OpenACC-Standard.org, "The openacc aplication programming interface, version 2.5," October 2015. [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf

[4] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. H. Kelly, G. Mudalige, B. V. Straalen, and S. Williams, "Loop chaining: A programming abstraction for balancing locality and parallelism," in *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, May 2013.

[5] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for gpus: automatic parallelization using trapezoidal tiles," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 2013, pp. 24–31.

[6] C. Olschanowsky, M. M. Strout, S. Guzik, J. Loffeld, and J. Hittinger, "A study on balancing parallelism, data locality, and recomputation in existing pde solvers," in *In The IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2014.

[7] D. Quinlan and C. Liao, "The rose source-to-source compiler infrastructure," in *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[8] S. Verdoolaege, "isl: An integer set library for the polyhedral model," *Mathematical SoftwareICMS 2010*, pp. 299–302, 2010. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-15582-6_49

[9] ——, *barvinok: User Guide*, 2015. [Online]. Available: http://compsys-tools.ens-lyon.fr/iscc/barvinok.pdf

[10] ——, "Integer Set Library," 2016. [Online]. Available: http://isl.gforge.inria.fr/

[11] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[12] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors," *PPOPP*, vol. 44, no. 4, pp. 219–228, 2009.

[13] W. Pugh and E. Rosser, "Iteration space slicing for locality," in *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, vol. LNCS 1863. London, UK: Springer-Verlag, August 1999, pp. 164–184.

[14] A. Beletska, W. Bielecki, A. Cohen, M. Palkowski, and K. Siedlecki, "Coarse-grained loop parallelization: Iteration space slicing vs affine transformations," *Parallel Computing*, vol. 37, no. 8, pp. 479 – 497, 2011.

[15] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *In Proceedings of the IEEE International Conference on Cluster Computing*, 2008.

[16] A. DURAN, E. AYGUADÉ, R. M. BADIA, J. LABARTA, L. MARTINELL, X. MARTORELL, and J. PLANAS, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[17] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta, "Extending the openmp tasking model to allow dependent tasks," in *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, ser. IWOMP'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 111–122.

[18] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 123–132.

[19] D. Andrade, B. B. Fraguela, J. Brodman, and D. Padua, "Task-parallel versus data-parallel library-based programming in multicore systems," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 101–110.

[20] M. Huang, V. K. Narayana, H. Simmler, O. Serres, and T. El-Ghazawi, "Reconfiguration and communication-aware task scheduling for high-performance reconfigurable computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, pp. 20:1–20:25, Nov. 2010.

[21] A. Chandramowlishwaran, K. Knobe, and R. W. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[22] P. Cicotti and S. Baden, "Latency hiding and performance tuning with graph-based execution," in *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on*, oct. 2011, pp. 28 –37.

[23] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[24] N. Vydyanathan, S. Krishnamoorthy, G. M. Sabin, U. V. Catalyurek, T. Kurc, P. Sadayappan, and J. H. Saltz, "An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 1158–1172, 2009.

[25] P. Virouleau, A. Roussel, F. Broquedis, T. Gautier, F. Rastello, and J.-M. Gratien, "Description, Implementation and Evaluation of an Affinity Clause for Task Directives," in *IWOMP 2016*, ser. IWOMP 2016 - LLCS 9903, Nara, Japan, Oct. 2016. [Online]. Available: https://hal.inria.fr/hal-01343442

[26] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and thread affinity in openmp programs," in *Proceedings of the 2008 Workshop on Memory Access on Future Processors (MAW)*. New York, NY, USA: ACM, 2008, pp. 377–384.

[27] F. Song, S. Moore, and J. Dongarra, "Analytical modeling and optimization for affinity based thread scheduling on multicore systems," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, September 2009, pp. 1 –10.

[28] J. Meng, J. Sheaffer, and K. Skadron, "Exploiting inter-thread temporal locality for chip multithreading," in *IPDPS*, 2010.

[29] F. Schlimbach, J. Brodman, and K. Knobe, “Concurrent collections on distributed memory theory put into practice,” in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, 2013, pp. 225–232.

[30] B. Norris, A. Hartono, and W. Gropp, “Annotations for productivity and performance portability,” in *Petascale Computing: Algorithms and Applications*, ser. Computational Science. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007, pp. 443–462, preprint ANL/MCS-P1392-0107. [Online]. Available: http://www.mcs.anl.gov/uploads/cels/papers/P1392.pdf

[31] Q. Yi, “Poet: a scripting language for applying parameterized source-to-source program transformations,” *Software: Practice and Experience*, vol. 42, no. 6, pp. 675–706, 2012. [Online]. Available: http://dx.doi.org/10.1002/spe.1089

[32] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, “Loop transformation recipes for code generation and auto-tuning,” in *Languages and Compilers for Parallel Computing*, vol. 5898. Springer Berlin Heidelberg, 2010, pp. 50–64.