An Object-Oriented Interface to The Sparse Polyhedral Library

Tobi Popoola[1] Ravi Shankar[2] Anna Rift[3] Shivani Singh[4] Eddie C. Davis[5]
Michelle Mills Strout[6] and Catherine Olschanowsky[7]
[1,2,3,4,7]Department of Computer Science, Boise State University, Boise USA
Email: {[1]tobipopoola | [2]ravishankar | [3]annarift | [4]shivanisingh}@u.boisestate.edu, [7]catherineolschan@boisestate.edu
[5]Vulcan Inc., Seattle, USA
Email: eddied@vulcan.com
[6]Department of Computer Science, University of Arizona, Tucson, USA
Email: mstrout@cs.arizona.edu

***Abstract*—Many important applications including machine learning, molecular dynamics, and computational fluid dynamics, use sparse data. Processing sparse data leads to non-affine loop bounds and frustrates the use of the polyhedral model for code transformation. The Sparse Polyhedral Framework (SPF) addresses limitations of the Polyhedral model by supporting non-affine constraints in sets and relations using uninterpreted functions. This work contributes an object-oriented API that wraps the SPF intermediate representation (IR) and integrates the Inspector/Executor Generation Library and Omega+ for precise set and relation manipulation and code generation. The result is a well-specified definition of a full computation using the SPF IR. The API provides a single entry point for tools to interact with the SPF, generate and manipulate polyhedral data flow graphs, and transform sparse applications.**

***Index Terms*—computation api, intermediate representation, sparse polyhedral framework, polyhedral dataflow graph**

I. Introduction

Many important applications including machine learning, molecular dynamics and computational fluid dynamics use sparse data. To save memory, sparse data is often compressed by storing only the non-zero values. Index arrays are used to map the non-zero value back to their coordinates in the dense space. The index arrays are used in loop bounds while iterating over the sparse data. These non-affine loop bounds complicate the use of compiler transformation for performance optimization.

The polyhedral model is an effective optimization tool for applications with affine loop bounds. This model represents computations as sets and relations. Iteration spaces are represented with sets and data dependences are represented using relations. This combination of iteration spaces and data dependences provides a partial ordering for the target computation. Within the partial ordering, the order of execution can be altered by applying relations to the iteration space. These relations are referred to as transformations. A limitation to the polyhedral model is that the constraints within iteration spaces and transformations must be affine.

The sparse polyhedral framework extends the polyhedral model by supporting non-affine iteration spaces and transformations using *uninterpreted functions*. Uninterpreted functions are symbolic constants that represent data structures such as the index arrays in sparse data formats. The SPF provides much of the same functionality as traditional polyhedral tools: code generation with CodeGen+ [1] built on Omega [2] and precise set and relation operations in the presence of uninterpreted functions with IEGenLib [3]. However, these tools are not tightly integrated and each has their own low-level API to interact with individual components.

The contribution of this work is the Computation API. The Computation API provides a precise specification of how to combine the individual components of the SPF to create an intermediate representation. This IR can directly produce Polyhedral Dataflow Graphs (PDFGs) [4] and translates graph operations defined for PDFGs into relations used by IEGenLib to perform transformations. In this work, we extend the PDFG representation to handle non-affine loops with imperfect nesting and loop carried dependences.

Figure 1 shows an overview of our optimization framework and where the computation API fits in the process. Spf-ie highlighted grey in the figure is a tool that automates translation from source code to our API and is currently under development. In this work we focus on manually translating original source code to the computation IR. The computation IR by itself supports composable transformations by virtue of the sparse polyhedral framework. Graph operations on a PDFG generates composable transformations and are applied to statements in the IR.

Our work overcomes limitations of previous approaches through a tight integration among CodeGen+, Omega, and IEGenLib. CHiLL [5] provides a script based interface for the SPF and supports non-affine transformations by using Omega. Omega has limitations on the precision of operations involving uninterpreted functions, as well as restrictions on how they must be expressed. Our computation API depends on IEGenLib [3] for set and relation operations, thereby overcoming the precision limitations.

The contributions of this paper include:

- A unified standard interface to specify a computation in the SPF
- Integration with PDFGs

- Extensions to PDFGs to support non-affine loop bounds, imperfectly nested loops, and loop carried dependences
- Translations between PDFG operations and SPF relations
- Integration between CodeGen+ and IEGenLib
- Support for inlining computations
- Peripheral support for generated code

II. Computation: A C++ Class

The interface to the SPF is implemented as a C++ class in IEGenLib. The computation class contains all of the information required to express a computation or a series of computations. This includes: data spaces, statements, data dependences, and execution schedules. This section describes the design and interface for each of these elements. Matrix vector multiplication is used as a running example throughout the rest of this paper. Figures 2 & 3 show the dense and sparse versions.

A. Data Spaces

A data space represents a collection of conceptually unique memory addresses. A data space of 0 dimensions is equivalent to a scalar. Each combination of data space name and input tuple is guaranteed to map to a unique space in memory for the lifetime of the data space. For example, $D(\vec{s})$, guarantees that for each unique tuple $\vec{s}$ there will be a unique memory location, and that during the lifetime of D its memory locations will not overlap with any other live data space's memory locations.

By default all data spaces are generic. They are used with the syntax $D(\vec{s})$. For example, for a 3 parameter input tuple i, j, k the data space can be represented as $D(i, j, k)$. This data space can be written to only once but read from any number of times. The exception to this rule is for accumulation operations when a single data location within a data space can be written to multiple times ($+=, -=, *=, max, min...$).

The data spaces represented in the matrix vector multiply example in are y, A, and x. In the sparse version the index arrays $rowptr$ and col are also considered data spaces. However, since they are used within the loop bounds and to access into another data space they must be constant for the duration of this computation. Therefore, they are not required as part of the Computation's definition.

```
// dense
Computation* dsComp = new Computation();
dsComp->addDataSpace("$y$");
dsComp->addDataSpace("$A$");
dsComp->addDataSpace("$x$");

// sparse
Computation* spsComp = new Computation();
spsComp->addDataSpace("$y$");
spsComp->addDataSpace("$A$");
spsComp->addDataSpace("$x$");
```

B. Statements

Statements perform read and write operations on data spaces. We restrict the definition of statements to be basic blocks. There is a single entry and a single exit from each block of code represented.

All statements have an iteration domain associated with them. This iteration domain is a set containing every instance of the statement and has no particular order. It is typically expressed as the set of iterators that the statement runs over, subject to the constraints of their iteration (loop bounds). The following code block shows how to create a statement using the Computation API. A statement is written as a string and the names of the data spaces are delimited with \$ symbols, this can be seen on lines 2 and 5 below for the dense and sparse cases respectively. The iteration domain is specified as a set using the IEGenLib syntax, with the exception of delimiting all data spaces with \$, this can be seen on lines 3 and 6 below.

```
1 Stmt* ds0 = new Stmt(
2   "$y$(i) += $A$(i,j) * $x$(j);",
3   "{[i,j]: 0 <= i < N && 0 <= j < M}", ...
4
5 Stmt* sps0 = new Stmt(
6   "$y$(i) += $A$(k) * $x$(j)",
7   "{[i,k,j]: 0 <= i < N && rowptr(i) <= k <
      rowptr(i+1) && j = $col(k)}", ...
```

C. Data Dependence Relationships

Data dependences exist between statements. They are encoded using relations between iteration vectors and data space vectors. Calculating a closure provides the dependence relationships between statements and a partial ordering constraint on the calculation. In our running examples the data reads and writes can be specified as written below.

```
/* 4th and 5th parameters to Stmt constructor */
// dense
...
{ // reads
  {"y", "{[i,j]->[i]}"},
  {"A", "{[i,j]->[i,j]}"},
  {"x", "{[i,j]->[j]}"}
},
{ // writes
  {"y", "{[i,j]->[i]}"}
}

// sparse
...
{ // reads
  {"y", "{[i,k,j]->[i]}"},
  {"A", "{[i,k,j]->[k]}"},
  {"x", "{[i,k,j]->[j]}"}
},
{ // writes
  {"y", "{[i,k,j]->[i]}"}
}
```

D. Execution Schedules

Execution schedules are determined using scattering functions that are required to respect the data dependence relations. Scheduling functions take as input the iterators that apply to the current statement, if any, and output the schedule as an integer tuple that may be lexicographically ordered with others to determine correct execution order of a group of statements. Iterators are commonly used as part of the output tuple, representing that the value of iterators affects the ordering of the statement. For example, in the scheduling function

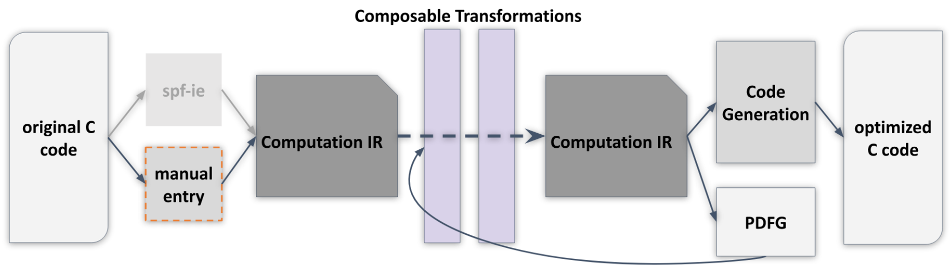

Fig. 1: Optimization Pipeline Overview

Dense matrix vector multiply

```c
for (i = 0; i < N; i++) {
    for (j=0; j<M; j++) {
        y[i] += A[i][j] * x[j];
    }
}
```

Fig. 2: Dense Matrix Vector Multiply

CSR Sparse matrix vector multiply

```c
for (i = 0; i < N; i++) {
    for (k=rowptr[i]; k<rowptr[i+1]; k++) {
        j = col[k];
        y[i] += A[k] * x[j];
    }
}
```

Fig. 3: Sparse Matrix Vector Multiply.

$\{[i,j]->[0,i,0,j,0]\}$, the position of i before j signifies that the corresponding statement is within a loop over j, which in turn is within a loop over i. Additionally, in a lexicographical ordering, all instances of the statement with $i = 1$ will precede all instances with $i = 2$, regardless of the value of j.

```c
/* 3rd parameter to the Stmt constructor */
// dense
 "{[i,j] ->[0,i,0,j,0]}"

 // sparse
"{[i,k,j]->[0,i,0,k,0,j,0]}"
```

Figure 4 shows the complete specification of two computation, first dense matrix vector multiply followed by sparse matrix vector multiply.

E. Code Generation

The Computation class interfaces with CodeGen+ [1] for code generation. CodeGen+ uses Omega sets and relations for polyhedra scanning. Omega sets and relations have limitations in the presence of uninterpreted functions. Uninterpreted functions are limited by the prefix rule. This rule states that an uninterpreted function must be a prefix of the tuple declaration. Uninterpreted functions cannot have expressions as parameters. Code generation overcomes this limitation by modifying uninterpreted functions in IEGenLib to be Omega compliant, while storing a mapping of the original uninterpreted function to its modified uninterpreted function. The separation of representations for transformations and code generation allows precise operations during transformations while still leveraging the functionality of CodeGen+ for polyhedra scanning.

Figure 5 shows the results of code generation for the sparse matrix vector multiplication computation defined in Figure 4. Line 2 of Figure 5 defines a macro for the statement s0, lines 9 - 13 remap the Omega compliant uninterpreted function back to its original. Lines 15 - 20 are a direct result of polyhedra scanning from CodeGen+. The Computation implementation provides all of the supporting definitions for fully functional code.

III. Visualizing a Computation on a Graph

Visualizing computations as a data flow graph gives performance experts a suitable view to reason about transformations for optimization opportunities. To this end, PDFGs express regular computations using a combination of the polyhedral model and dataflow graphs [4]. The original graphs have certain limitations: loop carried dependences were not expressed, and imperfect loop nests were not supported. This section describes how these limitations were overcome. Additionally, the creation of PDFGs by traversing the SPF IR is integrated with the Computation API. After a Computation is created such as the one in Figure 4 a function can be called that outputs the PDFG as a dot file.

In the original PDFG, shaded rectangular boxes represent data spaces and inverted triangles represent statements. In the extended PDFG, shaded rectangular boxes represent domains, transparent rectangular boxes represent data spaces and rounded rectangular boxes represent statements. Edges represent reads and writes in both the original and extended PDFG. The extended PDFG does not currently express the type and size of data spaces in a computation.

Loop carried dependences and imperfect loop nests are important patterns to consider when deciding which optimizations to apply. Loop carried dependences refer to coding patterns where one iteration of a loop reads or writes data produced by another iteration of the same loop. An imperfect

Full Computation Specification

```
// dense mvm
Computation* dsComp = new Computation();

// add data spaces
dsComp->addDataSpace("$y$");
dsComp->addDataSpace("$A$");
dsComp->addDataSpace("$x$");

Stmt* ds0 = new Stmt(
  // source code
  "$y$(i) += $A$(i,j) * $x$(j);",
  // iter domain
  "{[i,j]: 0 <= i < $N$ && 0 <= j < $M$}",
  // scheduling function
  "{[i,j] ->[0,i,0,j,0]}",
  { // data reads
    {"y", "{[i,j]->[i]}"},
    {"A", "{[i,j]->[i,j]}"},
    {"x", "{[i,j]->[j]}"}
  },
  { // data writes
    {"y", "{[i,j]->[i]}"}
  }
);
dsComp->addStmt(ds0);

// sparse mvm
Computation* spsComp = new Computation();

// add data spaces
spsComp->addDataSpace("$y$");
spsComp->addDataSpace("$A$");
spsComp->addDataSpace("$x$");

Stmt* sps0 = new Stmt(
  "$y$(i) += $A$(k) * $x$(j)",
  "{[i,k,j]: 0 <= i < N && rowptr(i) <= k < rowptr(i+1) && j = col(k)}",
  "{[i,k,j]->[0,i,0,k,0,j,0]}",
  {
    {"y", "{[i,k,j]->[i]}"},
    {"A", "{[i,k,j]->[k]}"},
    {"x", "{[i,k,j]->[j]}"}
  },
  {
    {"y", "{[i,k,j]->[i]}"}
  }
);
spsComp->addStmt(sps0);
```

Fig. 4: Computation API specification for dense and sparse matrix vector multiply

SPMV codegen

```
#undef s0
#define s0(__x0, i, __x2, k, __x4, j, __x6)   y(i) += A(k) * x(j)

#undef col(t0)
#undef col_0(__tv0, __tv1, __tv2, __tv3)
#undef rowptr(t0)
#undef rowptr_1(__tv0, __tv1)
#undef rowptr_2(__tv0, __tv1)
#define col(t0) col[t0]
#define col_0(__tv0, __tv1, __tv2, __tv3) col(__tv3)
#define rowptr(t0) rowptr[t0]
#define rowptr_1(__tv0, __tv1) rowptr(__tv1)
#define rowptr_2(__tv0, __tv1) rowptr(__tv1 + 1)

for(t2 = 0; t2 <= N-1; t2++) {
  for(t4 = rowptr_1(t1,t2); t4 <= rowptr_2(t1,t2)-1; t4++) {
    t6=col_0(t1,t2,t3,t4);
    s0(0,t2,0,t4,0,t6,0);
  }
}

#undef s0
#undef col(t0)
#undef col_0(__tv0, __tv1, __tv2, __tv3)
#undef rowptr(t0)
#undef rowptr_1(__tv0, __tv1)
#undef rowptr_2(__tv0, __tv1)
```

Fig. 5: SPMV codegen

The forward solve example code

```
for (i=0; i<N; i++){
   S0: tmp(i) = f(i);
   for(j=0; j<i; j++){
      S1: tmp(i) -= A(i, j)*u(j);
   }
   S2: u(i) = tmp(i)/A(i, i);
}
```

Fig. 6: Forward Solve

loop nest is one that has statements at multiple levels as shown in Figure 6.

A. Loop Carried Dependences

The existing representation is not capable of visualizing the presence of a loop carried dependence. Figure 8 shows the PDFG for the forward solve example in Figure 6. In the example, the code contains a loop carried dependence for the data space u in statements $S1$ and $S2$. However, the original PDFG graph in figure 8 does not visualize the loop carried dependence.

Figure 7 shows the extended PDFG. The outgoing edge from the data node u to the statement node $S1$ at index j shows the read access in the j-loop. The incoming edge from the statement node $S2$ to the data node u at index i shows the write access in the i-loop. This shows a loop carried dependence over i.

B. Imperfect loop nests

The forward solve example also exhibits an imperfect loop nest pattern: statements $S0$ and $S2$ are in the outer i-loop while statement $S1$ is in the inner j loop. The original PDFG in Figure 8 does not exhibit this pattern. The updated design in the extended PDFG show in Figure 7 visualizes the imperfect loop pattern.

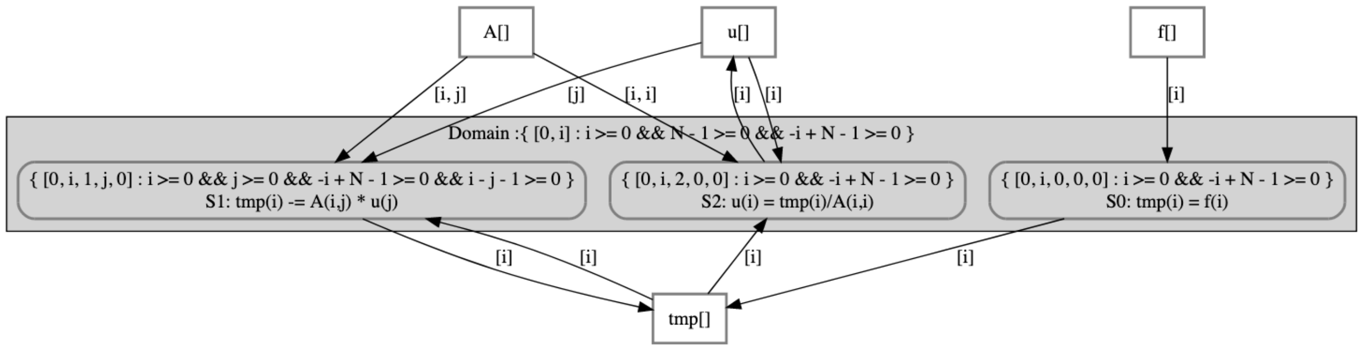
A[]
u[]
f[]
[i, j]
[j]
[i, i]
[i]
[i]
[i]
Domain :{ [0, i] : i >= 0 && N - 1 >= 0 && -i + N - 1 >= 0 }
{ [0, i, 1, j, 0] : i >= 0 && j >= 0 && -i + N - 1 >= 0 && i - j - 1 >= 0 }
S1: tmp(i) -= A(i,j) * u(j)
{ [0, i, 2, 0, 0] : i >= 0 && -i + N - 1 >= 0 }
S2: u(i) = tmp(i)/A(i,i)
{ [0, i, 0, 0, 0] : i >= 0 && -i + N - 1 >= 0 }
S0: tmp(i) = f(i)
[i]
[i]
[i]
[i]
tmp[]

Fig. 7: Extended PDFG. This graph is automatically generated from the Computation class and includes loop carried dependences and irregular loop nests. The lexicographical ordering of the tuples in the set attached to each statement informs its execution order.

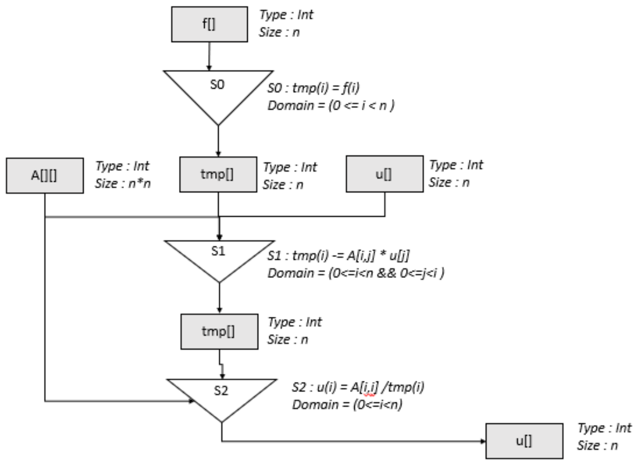

Fig. 8: The Original PDFG for the Forward Solve example.

C. Dot File Graph

The computation API is traversed to generate the extended PDFG as a directed graph. Loop nests are expressed using a modified version of polyhedra scanning. Polyhedra scanning is a top down recursive algorithm for generating code from a set of iteration domains of statements. Our modified version of polyhedra scanning uses the same approach but generates nodes of a directed graph for statements and generate clusters for statements within the same loop nest.

IV. Operations on Computations

Computations are composed of IEGenLib Sets and Relations. IEGenLib's functionality is used directly to manipulate the Computations' components. The core operations implemented in IEGenLib include inversion, compose, apply, and related supporting functions. Using these operations we are able to perform function inlining and loop transformations within the Computation class.

A. Inlining

Computation inlining handles the complexity of creating the SPF representation of functions that call other functions. Instances of Computations need to be reusable in the same way that functions are reusable.

Each function in the original source code is represented as a Computation. When a function call is first encountered in the source, a Computation must be created for it. Then, the contents of that Computation are inserted (inlined) into the caller's Computation that is being constructed. If the same function is called multiple times, the Computation that has been generated for it will be reused. The inlining process can continue to any nesting depth, if a function being called also calls other functions.

The inlining function is responsible for:

- avoiding naming conflicts between variables in the caller and callee,
- generating assignment statements for function parameters,
- providing callee return values to the caller and
- updating iteration domains, execution schedules and data dependences in the inlined statements.

Before inlining, the names of data spaces and iterators in the callee are prefixed with a unique string to avoid collisions with the caller's data spaces, or with other instances of the same inlined Computation within the same scope. This change is reflected in the stored source code string of the statement, as well as other parts of its representation that involve these names.

To keep things simple, the return values and arguments to an inlined function are restricted to either names of data spaces or literals. To pass a more complicated expression into a function, like `A[0]` or `x+y`, it must first be assigned to a temporary variable which can then be passed in. To preserve the original calling semantics of the program, when arguments are passed to a function, statements are generated to declare each of its parameters equal to the passed in values. No equivalent process occurs with return values, because they could potentially be used in a larger variety of contexts (assigned to variables, used immediately in an expression, or ignored entirely). Therefore, the inlining process returns the values that are returned by the inlined function, as strings, to be used however the caller sees fit.

Iteration domains, execution schedules, and data access relations are updated to reflect the surrounding context the statements have been inserted into. For example, if a function is called within a loop, and the callee itself also contains a loop, the representation of the innermost statements are adjusted to reflect that they are now nested under both loops.

B. Loop Transformations as Graph Operations

In this work, we provide fusion and reschedule operations on the graph. The operations on the graph translates to transformations performed by our interface API to manipulate the SPF components.

1) Fusion: This is a transformation that joins two statements from separate loops into a single loop. There are various categories of loop fusion, including *read reduction fusion* and *producer-consumer fusion*. Read reduction involves fusing loops that read from the same memory location while producer consumer fusion involves merging loops where one loop writes a variable that is then read by the second loop.

Graph operation for fusing two statements together at a particular level of the execution schedule is denoted as $fuse(S1, S2, level)$ in our IR. $S1$ and $S2$ are the statements to be fused and $level$ indicates what depth to fuse at. Specifying the depth to fuse at allows for more flexibility in a fusion operation. After fusion, $S2$ will be ordered immediately after $S1$.

2) Reschedule: The reschedule operation involves moving a statement to a new location in the graph and consequently changing its execution schedule. Reschedule by itself is not an optimization, however, it exposes optimization opportunities. The reschedule graph operation is denoted as $reschedule(S1, S2)$ in our IR. This will cause statement $S1$ to be rescheduled to appear before $S2$.

V. Related Work

Our work differs from other work on optimization using the polyhedral model because of our focus and support for sparse or irregular applications. We build on previous work of the Omega [1], CHiLL [6], and sparse polyhedral framework [7] projects. This interface provides a new mechanism to optimize applications using the sparse polyhedral framework and generate fully functional code incrementally within a legacy scientific application. The structure of the Computation API lays the groundwork for manipulating data spaces, including memory layout in the future.

A. Polyhedral Model Tools

Tools such as Polly [8], Pluto [9], Loopy [10], PolyMage [11] use the polyhedral model to transform regular codes. Polly automatically detects and transforms important code sections in the LLVM IR, breaking the limitations of most tools limited to a single source language. PolyMage is a domain specific language that automates the generation of efficient implementations of image processing pipelines. Halide [12] is another compiler for generating code for image computing algorithms. Halide separates algorithm and scheduling specification, thereby allowing optimization engineers to write different schedules for optimum performance. Their work also uses autotuning to generate efficient code by performing a stochastic search to find good schedules for the algorithm.

Pluto [9] is a fully automatic source to source transformation tool that optimizes programs for parallelism and locality. Pluto uses integer linear programming to decide on optimal code using parallelism and locality as part of its cost functions. Loopy, is a tool that allows a programmer to describe loop transformation at high level and verifies the transformation for correctness. Loopy, like Polly is implemented in LLVM. *isl* [13] is a tool for manipulating sets and relations in the polyhedral model. This tool forms the basis for affine transformations used in all the tools earlier discussed in this section [12], [9], [10], [11].

B. Program Dependence Graphs

PDFGs are based on a line of research started by Ferrante et. al, [14] with their work on program dependence graphs. Polyhedral optimizations are proposed in the work on Data Flow Graph Representations (DFGR) [15], [16]. Our work is closely related to this project, but polyhedral dataflow graphs are a view of the polyhedral representation rather than being a separate IR. [17]

C. Sparse Polyhedral Model Tools

This work takes a slightly different approach than CHiLL [18]. In CHiLL the performance expert writes a transformation recipe directed at a section of code. The script can include several operations that are a combination of abstract syntax tree and polyhedral transformations. Our work provides an interface to build a dataflow graph and control flow graph representation of the code and then provide an interface to transform the graph. This provides a visual representation of the transformations as well as making the polyhedral transformations more approachable.

VI. Conclusion

This work presents an object oriented API to the sparse polyhedral library. The API provides a standard interface to fully specify a computation in the Sparse Polyhedral framework from a single entry point. The single entry point is enabled by tight integration among IEGenLib, CodeGen+, and PDFGs. PDFGs have been expanded to represent non-affine loop boundaries, imperfect loop nests, and loop carried dependences. The API does not currently support early exits, loops where the bounds are modified within the loop nests, and loops without bounds such as while loops. In the future we hope explore techniques such as those from [19] to overcome limitations of representing unbounded loops and exit predicates.

We provide support for combining Computations through inlining. This increases reusability and reliability. The code generated by a computation encompasses statement macros and, when needed, variable declarations. This means that in the future, memory layout decisions for temporary storage can be made entirely within the SPF.

Future tools will use this API to generate the SPF IR, manipulate it, and produce optimized code from legacy applications and custom high-level or domain specific languages. Additionally, this API will be used when iteratively transforming a computation; the PDFG will be displayed at every step to guide the performance experts decisions.

REFERENCES

[1] Chun Chen. Polyhedra scanning revisited. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 499–508, 2012.

[2] Pugh W Rosser E Shpeisman T Wonnacott D Kelly W, Maslov V. The Omega Library interface guide. *University of Maryland at College Park Mar 1995*, 1995.

[3] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. Set and relation manipulation for the Sparse Polyhedral Framework. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7760 LNCS:61–75, 2013.

[4] Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. Transforming loop chains via macro dataflow graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 265277, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Mary Hall, Jacqueline Chame, Jaewook Shin, Chun Chen, Gabe Rudy, and Malik Murtaza Khan. Loop transformation recipes for code generation and auto-tuning. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2009.

[6] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 192–202, Los Alamitos, CA, USA, October 1999. IEEE Computer Society.

[7] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. An approach for code generation in the sparse polyhedral framework. *Parallel Computing*, 53(C):32–57, April 2016.

[8] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly - Polyhedral optimization in LLVM. *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT '11)*, page None, 2011.

[9] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System. *PLDI 2008 - 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–15, 2008.

[10] Kedar S. Namjoshi and Nimit Singhania. Loopy: Programmable and formally verified loop transformations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9837 LNCS(July):383–402, 2016.

[11] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage. *ACM SIGPLAN Notices*, 50(4):429–443, mar 2015.

[12] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Frédo Durand, Connelly Barnes, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 519–530, 2013.

[13] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Lecture Notes in Computer Science,*, pages 299–302. Springer, September 2010.

[14] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[15] Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. Polyhedral optimizations for a data-flow graph language. In Xipeng Shen, Frank Mueller, and James Tuck, editors, *Languages and Compilers for Parallel Computing*, pages 57–72, Cham, 2016. Springer International Publishing.

[16] Alina Sbirlea, Louis-Noel Pouchet, and Vivek Sarkar. DFGR an Intermediate Graph Representation for Macro-Dataflow Programs. In *2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, pages 38–45, Edmonton, AB, Canada, August 2014. IEEE.

[17] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[18] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, June 2008.

[19] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, volume LNCS 6011, Berlin, Heidelberg, 2010. Springer-Verlag.