Techniques for Managing Polyhedral Dataflow Graphs

No Author Given

No Institute Given

Abstract. Scientific applications, especially legacy applications, contain a wealth of scientific knowledge. As hardware changes, applications need to be ported to new architectures and extended to include scientific advances. As a result, it is common to encounter problems like performance bottlenecks and dead code. A visual representation of the dataflow can help performance experts identify and debug such problems. The Computation API of the sparse polyhedral framework (SPF) provides a single entry point for tools to generate and manipulate polyhedral data flow graphs, and transform applications. However, when viewing graphs generated for scientific applications there are several barriers. The graphs are large, and manipulating their layout to respect execution order is difficult. This paper presents a case study that uses the Computation API to represent a scientific application, *GeoAc*, in the SPF. Generated polyhedral data flow graphs were explored for optimization opportunities and limitations were addressed using several graph simplifications to improve their usability.

Keywords: Sparse Polyhedral Framework · Computation API · Polyhedral Dataflow Graph.

1 Introduction

Scientific applications, especially legacy applications, contain a wealth of scientific knowledge. However, older codes need to be ported to new architectures and new generations of computational scientists need to extend them to keep making scientific progress. As applications age and are passed from programmer to programmer, problems creep in: logic and memory bugs, performance bottlenecks, and dead code are just a few of the possibilities. A visual representation of the code will speed up the learning process for new programmers and can help identify existing issues with the code. Additionally, the right abstraction will allow performance optimizations to be performed by manipulating the visual representation rather than rewriting code manually.

Polyhedral dataflow graphs [6] highlight the dataflow, data access patterns, and execution schedule for applications visually. Originally developed to identify temporary storage reduction opportunities [12], they have proven to be useful for learning code bases and identifying opportunities for parallelism. Previous efforts used manual drawings of the graphs and then automated graph generation, running only on very small examples. Applying these techniques to real scientific applications remains a significant challenge. This paper uses a scientific application, *GeoAc*, to explore the limitations of polyhedral dataflow graphs and proposes several techniques to ensure correctness and improve their usability.

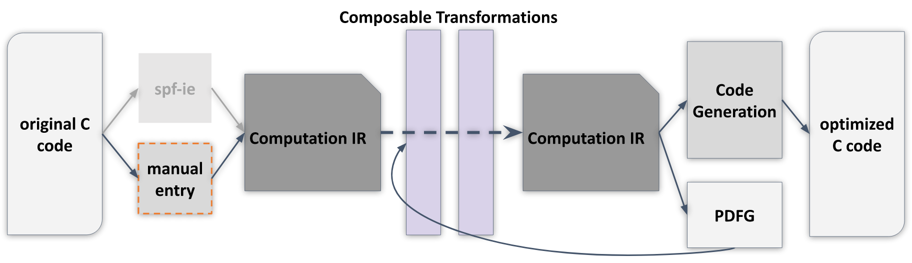


Fig. 1: Optimization Pipeline Overview [13].

Polyhedral Dataflow graphs are part of the Sparse Polyhedral Framework (SPF) and are generated automatically from the SPF intermediate representation (IR) [13]. They are referred to as polyhedral because statements are represented as a combination of statements and iteration spaces that are expressed using the polyhedral model. The polyhedral model is a mathematical representation of the source code. Transformations to the execution schedule can be applied using relations. The relations are applied to the iteration spaces, expressed as sets. The resulting code may have a different execution order or different control flow. Importantly, the transformations can be composed. This means that an arbitrarily long series of transformations can be applied to the same code base.

Figure 1 shows the anticipated workflow for human-in-the-loop optimization using polyhedral dataflow graphs (shown as PDFG in figure 1). Once an application is converted to the SPF intermediate representation, a performance expert examines the resulting graphs, indicates a series of transformations as graph operations, and repeats the process until they are satisfied. Code generation then produces the newly optimized code.

Previous work demonstrated the concept of polyhedral dataflow graphs using manually constructed graphs that represented the execution scheduling using the layout position of nodes and dataflow using edges between nodes. Notably, both dataspaces and statements are represented as nodes in the graph. Due to the limitations of this format, the execution schedule does not guide the layout of the graphs as it did in the manually produced graphs. The result is a very large graph. The goal of polyhedral dataflow graphs is to reveal dataflow optimizations and parallelism opportunities; to make this happen the graphs need to be manipulated to be smaller and communicate key information clearly.

Figure 10 shows a high-level view of the entire graph generated for a key function of *GeoAc*. The graph is not readable as it contains 4616 statement nodes and several thousand more dataspace nodes. This serves as the starting point for this work once correctness was established.

This paper documents the steps taken to process the graphs into more informative and manageable representations. While working with the graphs, we also identified operations needed for correctness. The contributions of this work include:

- A method to transform sections of code to single static assignment without requiring a control flow graph.
- Proposed alterations to single static assignment to accommodate parameters to the computation that are pointer or reference types.
- Suggested changes to arrays used to pack related variables together.
- Changes to the visualization of graphs to increase usability.

2 Background

This case study uses a portion of a scientific application to explore the capabilities of the Sparse Polyhedral Model and supporting tools. This section describes the application, *GeoAc*, and reviews important components of the Sparse Polyhedral Model.

2.1 *GeoAc*

Many earthquakes cause sudden mass displacements at the earth's surface. When this type of earthquake occurs under the ocean, is of strong enough magnitude, and meets certain other criteria, a tsunami is generated. Ground or sea-surface displacements push on the atmosphere, which in turn generates an atmospheric disturbance. This disturbance propagates upward as an acoustic wave eventually inducing a local change in the electron density of the ionosphere. Global Navigation Satellite Systems (GNSS) monitor ionospheric disturbances induced by such phenomena. Such satellite-based remote sensing methods are used to estimate the earth's surface deformation and predict the arrival time of a tsunami.

IonoSeis is a software package that combines multiple existing codebases into a single package to model GNSS-derived electron perturbations in the ionosphere due to the interaction of the neutral atmosphere and charged particles in the ionosphere. One of the pieces of *IonoSeis* is a ray-tracing package called *WASP3D* which is an older tool that does not meet the needs of the workflow. *GeoAc* [3] is a newer ray-tracing package developed at Los Alamos National Laboratory that better models the physics, and is the proposed replacement for *WASP3D*. The software is written in C++ and models the propagation of acoustic waves through the atmosphere using a fourth-order Runge–Kutta method (RK4).

A performance analysis indicates that the RK4 function is the most expensive operation in *GeoAc* and is therefore chosen for further analysis. In this case study, we consider the practical implications of viewing the full RK4 function as a Polyhedral Dataflow Graph.

2.2 SPF and the Computation API

The Sparse Polyhedral Framework extends the polyhedral model by supporting non-affine iteration spaces and transforming irregular computations using *uninterpreted functions* [8]. Uninterpreted functions are *symbolic constants* that represent data structures such as the index arrays in sparse data formats. Symbolic constants are constant values that do not change during the course of a computation. The SPF can represent computations with indirect memory accesses, relations with affine constraints, and constraints involving uninterpreted function symbols. The SPF represents *run-time reordering transformations* using integer tuple sets [18,19]. Run-time data reordering techniques attempt to improve the spatial and temporal data locality in a loop by reordering the data based on the order in which it was referenced in the loop [17].

The Computation API [13] is an object-oriented API that provides a precise specification of how to combine the individual components of the SPF to create an intermediate representation (IR). This IR can produce polyhedral dataflow graphs [6] and translates graph operations defined for polyhedral dataflow graphs into relations used by the *Inspector/Executor Generator Library (IEGenLib)* [18]. It can also be passed to Omega [14] for code generation.

IEGenLib is a C++ library with data structures and routines that represent, parse, and visit integer tuple sets and relations with affine constraints and uninterpreted function symbol equality constraints [18]. The Computation API is implemented as a C++ class in *IEGenLib* and contains all of the components required to express a Computation or a series of Computations. Dense and sparse matrix vector multiplication, shown in Figures 2 and 4, are used as examples to represent the computations in the SPF.

2.3 Polyhedral Dataflow Graphs

Polyhedral Dataflow graphs [6] represent both the dataflow and execution schedule of a computation. Initially, the graphs were manually drawn using the polyhedral representation as a guide. The current version of the graph is automatically generated. The SPF Computation IR is visited and a dot format graph is created.

Figures 3, 5 show the corresponding polyhedral dataflow graphs that are generated using the IR created in Figures 2 and 4. Multiple node types connected by edges comprise the graphs. These node type are variations of statement or dataspace nodes. Node types include: statements, data spaces, read-only parameters, parameters, active-out data spaces, read-only-active-out parameters, and active-out parameters A statement node is represented as a rectangle with rounded edges. It has an execution schedule, a statement number, and potentially a debug string. We generate the execution schedule by applying the scheduling function to the iteration space. For example, the statement node in Figure 5 executes the statement referred to using macro $S0$ with the execution schedule $\{[0, a1, 0.a3.0] : a1 \geq 0 \wedge a3 \geq 0 \wedge -a1+N-1 \geq 0 \wedge -a3+M-1 \geq 0\}$. This is generated by applying the scheduling function $\{[i, k, j]- > [0, i, 0, k, 0, j, 0]\}$ to the iteration space $\{[i, k, j] : 0 \leq i < N \wedge rowptr(i) \leq k < rowptr(i+1) \wedge j = col(k)\}$.

Dense Matrix vector multiply

```
/* Dense vector multiply
for (i = 0; i < N; i++) {
    for (j=0; j<M; j++) {
        y[i] += A[i][j] * x[j];
}}*/
Computation* denseComp = new Computation();
denseComp->addDataSpace("y");
denseComp->addDataSpace("A");
denseComp->addDataSpace("x");
Stmt* denseS0 = new Stmt(
  "$y$(i) += $A$(i,j) * $x$(j);", // Source code
  "{[i,j]: 0 <= i < $N$ && 0 <= j < $M$}", // Iteration domain
  "{[i,j] ->[0,i,0,j,0]}", // Scheduling Function
  { {"y", "{[i,j]->[i]}"}, {"A", "{[i,j]->[i,j]}"},
    {"x", "{[i,j]->[j]}"} }, // Data reads
  { {"y", "{[i,j]->[i]}"} } ); // Data writes
denseComp->addStmt(denseS0);
```

Fig. 2: Dense Matrix Vector Multiply.

Code generation uses execution schedules to lexicographically order the statements.

Dataspace nodes are drawn as rectangles with sharp corners. The representation splits dataspace nodes into the types listed above. In static single assignment form, every dataspace should have only one edge pointing into it. A form of liveness analysis will be used to minimize the amount of memory used during execution.

All edges represent reads and writes to dataspaces by statement nodes. The labels on edges refer to the access parameters. All scalar values are read and written using 0 as an access parameter. Arrays can be read and written using any combination of constants or iterators from the iteration space.

3 Case Study: Expressing *GeoAc* and Examining Polyhedral Data Flow Graphs

This case study uses a manual implementation of the SPF IR. The implementation was written based on the *GeoAc* code base and testing of the generated code demonstrated correctness of the implementation. The planned use of the Computation API is through a tool that is currently under development that traverses the Clang AST and produces the IR. Figure 1 shows the anticipated workflow. Legacy applications will be parsed by Clang, the AST traversed to produce a Computation IR, and the IR outputted either as a graph to be ma-

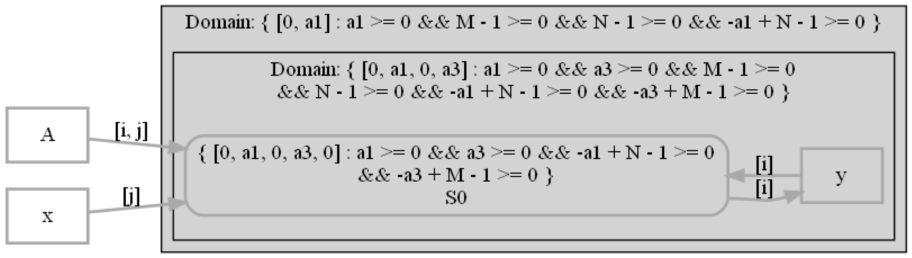

Domain: { [0, a1] : a1 >= 0 && M - 1 >= 0 && N - 1 >= 0 && -a1 + N - 1 >= 0 }
Domain: { [0, a1, 0, a3] : a1 >= 0 && a3 >= 0 && M - 1 >= 0 && N - 1 >= 0 && -a1 + N - 1 >= 0 && -a3 + M - 1 >= 0 }
{ [0, a1, 0, a3, 0] : a1 >= 0 && a3 >= 0 && -a1 + N - 1 >= 0 && -a3 + M - 1 >= 0 }
S0
A
[i, j]
x
[j]
[i]
[i]
y

Fig. 3: Polyhedral Data Flow Graph for Dense matrix vector multiply. Rounded rectangles represent statements, square rectangles represent data spaces and shaded square rectangles represent nested loops.

nipulated or as C code. This case study drove the development of the API and demonstrated several shortcomings of current polyhedral data flow graphs.

This section overviews the challenges overcome to create accurate dataflow graphs using a polyhedral representation. The first challenge was to create an approximation to single static assignment in the absence of a control flow graph. Special handling of structs, pass-by-reference or pointer parameters, and some arrays was required. The size of the graphs make them almost impossible to view. To circumvent this we minimize all statements that are not within loops and propose future analysis to further simplify the graphs. The polyhedral model and the SPF require constraints to be affine based on constants. Scientific codes often use data in control flow. We expanded our representation to handle data in constraints in limited circumstances. Finally, we implemented a debugging interface that can be used to map graph nodes to a location in the generated code.

3.1 Approximate Single Static Assignment

The Computation API and polyhedral dataflow graphs support intrinsic types, pointers, and references. User defined types (structs and classes) are not supported. These restrictions allow memory allocation to be delayed until after code generation. The memory allocation is preprended to the source code. Macros map between the actual memory and the dataspace names used in the representation. Scientific codes commonly make extensive use of user defined types. All structs and classes must be flattened. All *GeoAc* structs were converted to a set of dataspaces: one corresponding to each member type. This alteration was done before using the computation API. The consequence is that any tool generating calls to the API is responsible for object flattening.

Other changes required are made within the intermediate representation, as it is built. The computation is converted to SSA form as it is built. As each statement is added, the reads and writes are inspected and stored. If the dataspace written to by a statement was also written to by a previous statement, the

Sparse matrix vector multiply

```
/*Sparse vector multiply
for (i = 0; i < N; i++) {
    for (k=rowptr[i]; k<rowptr[i+1]; k++) {
        j = col[k];
        y[i] += A[k] * x[j];
}}*/
Computation* sparseComp = new Computation();
sparseComp->addDataSpace("y");
sparseComp->addDataSpace("A");
sparseComp->addDataSpace("x");
Stmt* sparseS0 = new Stmt(
  "$y$(i) += $A$(k) * $x$(j)", // Source code
  // iteration domain
  "{[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]}", // Scheduling Function
  { {"y", "{[i,k,j]->[i]}"},{"A", "{[i,k,j]->[k]}"},
    {"x", "{[i,k,j]->[j]}"}}, // Data reads
  { {"y", "{[i,k,j]->[i]}"} } // Data writes
);
sparseComp->addStmt(sparseS0);
```

Fig. 4: Sparse Matrix Vector Multiply.

dataspace of the previous statement gets a revision number. Affected reads are updated as well. Importantly, the IR does not keep a control flow graph and ϕ or join nodes must be added to ensure proper versioning.

To generate ϕ nodes in the absence of a control flow graph we use the constraints on iteration spaces. We use a dominance frontier method [5] adapted for use with the polyhedral model rather than a control flow graph. Suppose foo is a dataspace that requires a ϕ node as in Figure 6. We must locate three statements:

1. **read statement** - the statement that reads from foo
2. **first write** - the most recent write to foo under any constraints
3. **guaranteed write** - The most recent write to foo whose constraints also apply to the read statement.

We begin with the read statement and move backwards through our statements, identifying the first and guaranteed writes. We construct a phi node if the first and guaranteed writes are distinct statements. We extract all conditions which apply to the first write but not to the read statement. The phi node takes the general form: *foo = conditions ? first write : guaranteed write*. Due to SSA, guaranteed write is versioned while first write is unversioned. The addition of

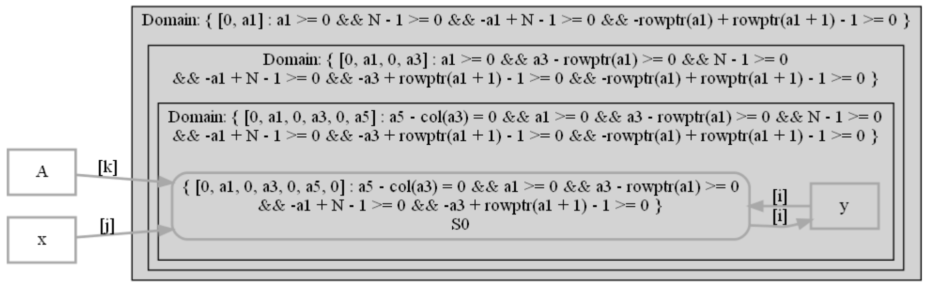

Domain: { [0, a1] : a1 >= 0 && N - 1 >= 0 && -a1 + N - 1 >= 0 && -rowptr(a1) + rowptr(a1 + 1) - 1 >= 0 }
Domain: { [0, a1, 0, a3] : a1 >= 0 && a3 - rowptr(a1) >= 0 && N - 1 >= 0 && -a1 + N - 1 >= 0 && -a3 + rowptr(a1 + 1) - 1 >= 0 && -rowptr(a1) + rowptr(a1 + 1) - 1 >= 0 }
Domain: { [0, a1, 0, a3, 0, a5] : a5 - col(a3) = 0 && a1 >= 0 && a3 - rowptr(a1) >= 0 && N - 1 >= 0 && -a1 + N - 1 >= 0 && -a3 + rowptr(a1 + 1) - 1 >= 0 && -rowptr(a1) + rowptr(a1 + 1) - 1 >= 0 }
A
[k]
x
[j]
{ [0, a1, 0, a3, 0, a5, 0] : a5 - col(a3) = 0 && a1 >= 0 && a3 - rowptr(a1) >= 0 && -a1 + N - 1 >= 0 && -a3 + rowptr(a1 + 1) - 1 >= 0 }
S0
[i]
[i]
y

Fig. 5: Polyhedral Data Flow Graph for Sparse matrix vector multiply. Rounded rectangles represent statements, square rectangles represent data spaces and shaded square rectangles represent nested loops.

Phi Node Example

```
foo = 0; // SSA: foo_0 = 0; (guaranteed write)
if (i - 1 >= N) {
    foo = 1; // SSA: foo_1 = 1; (first write)
}
// SSA: add phi node foo = -N + i - 1 >= 0 ? foo_1 : foo_0;
bar = foo; // read statement
```

Fig. 6: Phi Node Example.

the ϕ node provides a new write to foo and versions the first write. This means all three statements write to different versions of foo.

Parameters that are pointer or reference types have to be handled differently. Any parameters of those types can be rewritten multiple times. As part of SSA, the final write to a data space remains unversioned — only previous writes are versioned. Thus at the end of their dataflow, these parameters retain their original names and are then correctly recognized as active-out data spaces. It is important to consider the execution schedule when examining these nodes in the dataflow graphs as this could allow for illegal schedule transformations to be applied.

One coding pattern observed in scientific applications is packing individual, but related scalar variables, into constant sized arrays. The code then accesses the variables using constants or iterators — the latter often occurring in loops with small domains. Often, this domain is the number of dimensions being simulated. In our SSA form, arrays are versioned as a whole, meaning that writing to a single index of an array versions the entire array. If a loop writes to the elements of an array the array is only versioned once for the entire loop. When dealing with arrays that are a small number of packed variables this causes extra

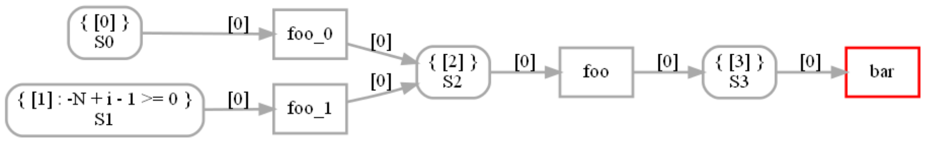


Fig. 7: Graph generated from Phi Node Example. `foo` is written to once as `foo_0` (S0) and then again as `foo_1` if $-N + j - 1 >= 0$ (S1). `foo` then chooses between these values based on this condition (S2) and is subsequently read by `bar` (S3). Rounded rectangles represent statements, square rectangles represent data spaces and red square rectangles represent returned dataspaces.

versioning and complicates the view of the dataflow. Our solution is to detect arrays that are only accessed using literals and replace them with individual variables. Array accesses using variables whose values can be determined at compile time are also replaced.

3.2 Data Dependant Control Flow

Each statement in the Computation IR stores constraints on its execution in its iteration space. The SPF requires that those constraints each be affine or be affine expressions using constant uninterpreted functions. However, scientific applications often define control flow using data. An example of this is a Riemann solve where the computation used depends on the value at that iteration [2]. We support constraints on data by requiring that they are constant for the duration of the loop nest and treating them as uninterpreted functions. Transformations can be performed in the presence of these constraints, but cannot use them. This is a feature we will explore in the future. One limitation of this approach is that IEGenLib will not support $\neq$ constraints as this would create a non-convex iteration space.

3.3 Debugging Information

While examining the graphs it is necessary to understand the connection between the nodes in the graph and the original source code. However, there is a disconnect between statement nodes on the graph and statement objects from the Computation API due to function inlining and ϕ nodes. This is important because object in the Computation API directly correlate to lines in the original source code. Function inlining causes the same statement object to generate in the graph multiple times each as a different node. Programmatic addition of ϕ statements and array access statements further changes the graph statement order from the API statement order. To overcome this limitation we added a debugging interface that allows us to tag a statement object with a string. This string shows up on all statement nodes generated from that object, as shown in Figure 8. Each statement node directly maps to a line in the generated code.

With a debug statement, we can connect a statement object to a set of lines in the generated code. Eventually, this interface will be used by automated tools to provide filename and line number information from the original input code parsed by Clang.

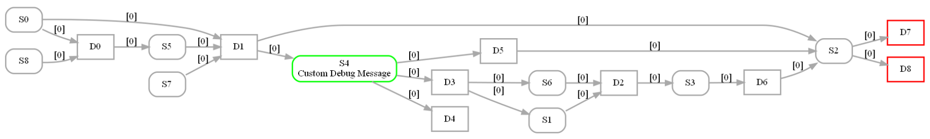


Fig. 8: An example graph that includes a debug statement at $S4$ (highlighted green for easy identification). In the scalable vector graphic (SVG) format, the user-defined debug string is searchable for easy node identification.

3.4 Layout Considerations and Limitations

Polyhedral dataflow graphs are generated using the dot format. Dot provides little control over the layout of the graph because it depends almost entirely on the connectivity of the components. This makes the layout almost entirely dependant on dataflow rather than execution order. The original graphs were laid out by hand and execution order determined the placement of the nodes.

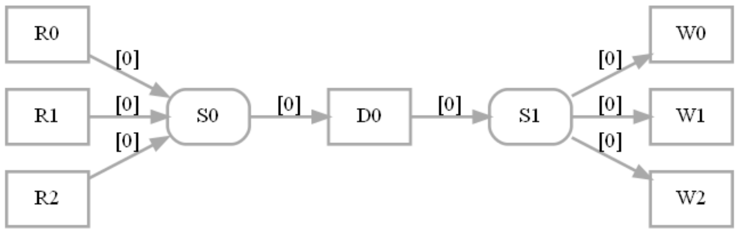
Due to its large size, displaying the graph left-to-right instead of the typical top-to-bottom made viewing simpler. Statement and data space nodes can be reduced to points to further simplify the graph and better expose loop-subgraphs.

Producer-consumer fusion eliminates unnecessary dataflow thereby reducing temporary storage requirements [6]. A producer-consumer relationship exists between some statements $S0$ and $S1$ through a dataspace $D0$ if:

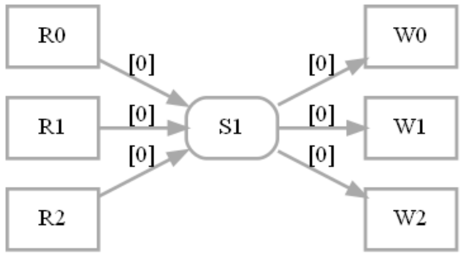
- $S0$ only writes to $D0$
- $D0$ is only written to by $S0$
- $D0$ is only read by $S1$

This is shown in Figure 9a. To remove this unnecessary dataflow, $S0$ and $D0$ are removed from the graph. $S0$'s reads are assigned to $S1$ and $D0$ is removed as a read from $S1$. An example result is shown in Figure 9b. The reads and writes must be constant accesses (as opposed to iterator access for arrays). Producer-consumer relationships are unnecessary for correct dataflow as they contain only a single path for the data to flow. Removing them improves readability.

Colors are used to indicate different types of dataspaces. This is important information because data spaces that are pointer or reference types must be treated differently than normal dataspaces. Colors also assist with debugging and ensuring graph correctness. Filling versus outlining nodes provides an additional layer of distinction between node types — especially similar node types.



(a) A dataflow graph with a producer-consumer relationship between $S0$ and $S1$.



(b) The dataflow graph after producer-consumer fusion.

Fig. 9: Producer-consumer fusion example. R's represent reads and W's represent writes.

The size of the graphs presents challenges beyond viewing the graph. A combination of size and graph complexity causes tools that render the dot input as an SVG file to fail. Portions of the graphs have to be created separately to work around this. This and the desire to use the graphs interactively will drive our future work, including identifying a different format to express the graphs.

4 Related Work

Polyhedral dataflow graphs are a compiler intermediate representation that exposes optimization opportunities such as loop transformation and temporary storage reductions. To implement polyhedral data flow graphs, the framework introduces a specification language called Polyhedral Dataflow Language. This specification language can be written directly, derived from existing codes, or lifted from another intermediate representation.

Existing work demonstrates the benefit of polyhedral data flow optimizations. Olschanowsky et al. demonstrated this benefit on a computational fluid dynamic benchmark [12]. Davis et al. automated the experiments from the previous work using modified macro dataflow graphs [6]. Strout et al. extended the polyhedral model for sparse computations allowing indirect array accesses [18]. This research distinguishes itself by being applied to a full application in a different domain.

Tools such as *Polly* [7], *Pluto* [4], *Loopy* [11], *PolyMage* [10] and *Halide* [15,16,9] use the polyhedral model to transform regular codes. *PolyMage* and *Halide* are two domain specific languages and compilers for optimizing parallelism, locality, and recomputation in image processing pipelines. *Halide* separates the definition of algorithms from the concerns of optimization making them simpler, more modular and more portable. This lets us play around with different optimizations with a guarantee not to change the result. *PolyMage's* optimization strategy relies primarily on the transformation and code generation capabilities of the polyhedral compiler framework and performs complex fusion, tiling, and storage optimization automatically.

The *isl* (integer set library) is a thread-safe C library for manipulating sets and relations of integer tuples bounded by affine constraints [20]. This library is used in the polyhedral model for program analysis and transformations and forms the basis for affine transformations used in all the tools discussed previously in this section. Stateful dataFlow multigraphs (SDFGs) are a data-centric IR that enables separating code definition from its optimization [1]. Polyhedral dataflow graphs differ from SDFGs because of their dependence on the polyhedral model. The graphs are not the intermediate representation, but a view of that representation. Any graph operations performed to transform the graph are translated to relations and applied to the underlying polyhedral representation.

5 Conclusion

This paper presents a case study that uses the Computation API to represent a scientific application, *GeoAc*, in the sparse polyhedral framework. Polyhedral dataflow graphs were generated from the SPF IR and measures were taken to make the graphs more readable and informative. The computation is converted to SSA form as it is built to simplify and enable optimizations like redundancy elimination. In the absence of control flow, constraints on the iteration space were used to generate *phi* nodes for dataspace versioning. The large size of the generated graph is made more manageable by minimizing non-loop statements to keep the graphs simple and easy to read. Function inlining by the Computation API makes it difficult to map the source code to the generated code. This limitation is overcome by adding a debugging interface that can tag statement objects created from the source code with a string that is then searchable in the graph. Polyhedral dataflow graphs are generated using the dot format, making the layout of the graph dependant on the dataflow rather than execution order. The graph is displayed left to right and different colors are used to distinguish various graphical elements.

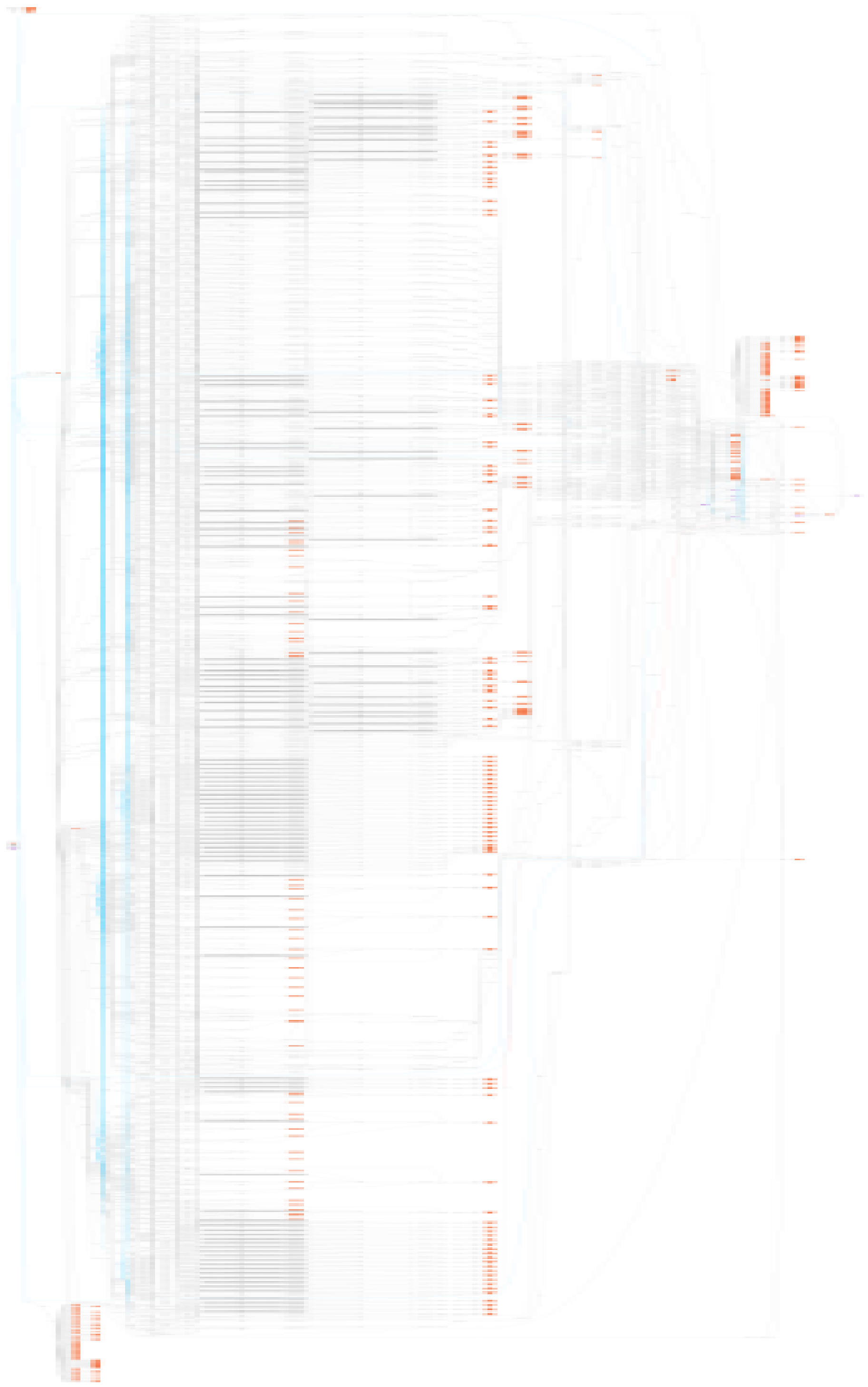


Fig. 10: Full dataflow graph.

References

1. Ben-Nun, T., de Fine Licht, J., Ziogas, A.N., Schneider, T., Hoefler, T.: 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. SC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3295500.3356173
2. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction. p. 283–303. CC'10/ETAPS'10, Springer-Verlag, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11970-5$_1$6
3. Blom, P.: Geoac: Numerical tools to model acoustic propagation in the geometric limit. https://github.com/LANL-Seismoacoustics/GeoAc (2014)
4. Bondhugula, U., Ramanujam, J., Sadayappan, P.: PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System. PLDI 2008 - 29th ACM SIGPLAN Conference on Programming Language Design and Implementation pp. 1–15 (2008)
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 25–35. POPL '89, Association for Computing Machinery, New York, NY, USA (1989). https://doi.org/10.1145/75277.75280, https://doi.org/10.1145/75277.75280
6. Davis, E.C., Strout, M.M., Olschanowsky, C.: Transforming loop chains via macro dataflow graphs. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization. pp. 265–277. ACM (2018)
7. Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., Pouchet, L.N.: Polly - Polyhedral optimization in LLVM. Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT '11) p. None (2011), http://perso.ens-lyon.fr/christophe.alias/impact2011/impact-07.pdf
8. LaMielle, A., Strout, M.M.: Enabling code generation within the sparse polyhedral framework. Technical report, Technical Report CS-10-102 (2010)
9. Mullapudi, R.T., Adams, A., Sharlet, D., Ragan-Kelley, J., Fatahalian, K.: Automatically scheduling halide image processing pipelines. ACM Transactions on Graphics (TOG) **35**(4), 83 (2016)
10. Mullapudi, R.T., Vasista, V., Bondhugula, U.: Polymage: Automatic optimization for image processing pipelines. In: ACM SIGARCH Computer Architecture News. vol. 43, pp. 429–443. ACM (2015)
11. Namjoshi, K.S., Singhania, N.: 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). https://doi.org/10.1007/978-3-662-53413-7$_1$9
12. Olschanowsky, C., Strout, M.M., Guzik, S., Loffeld, J., Hittinger, J.: A study on balancing parallelism, data locality, and recomputation in existing pde solvers. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 793–804. IEEE Press, IEEE Press, 3 Park Ave, New York, NY, USA (2014)

13. Popoola, T., Shankar, R., Rift, A., Singh, S., Davis, E., Strout, M., Olschanowsky, C.: An object-oriented interface to the sparse polyhedral library. In: 2021 Ninth Workshop on Data-Flow Execution Models for Extreme Scale Computing (Accepted)
14. Pugh, W., Wonnacott, D.: Eliminating false data dependences using the omega test. In: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation. p. 140–151. PLDI '92, Association for Computing Machinery, New York, NY, USA (1992). https://doi.org/10.1145/143095.143129
15. Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., Durand, F.: Decoupling algorithms from schedules for easy optimization of image processing pipelines (2012)
16. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Notices **48**(6), 519–530 (2013)
17. Strout, M.M., Carter, L., Ferrante, J.: Compile-time composition of run-time data and iteration reorderings. SIGPLAN Not. **38**(5), 91–102 (May 2003). https://doi.org/10.1145/780822.781142
18. Strout, M.M., Georg, G., Olschanowsky, C.: Set and relation manipulation for the sparse polyhedral framework. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 61–75. Springer (2012)
19. Strout, M.M., LaMielle, A., Carter, L., Ferrante, J., Kreaseck, B., Olschanowsky, C.: An approach for code generation in the sparse polyhedral framework. Parallel Computing **53**, 32–57 (2016)
20. Verdoolaege, S.: isl: An integer set library for the polyhedral model. In: International Congress on Mathematical Software. pp. 299–302. Springer (2010)