Techniques for Managing Polyhedral Dataflow Graphs

Ravi Shankar[1], Aaron Orenstein[2], Anna Rift[3], Tobi Popoola[3], MacDonald Lowe[3], Shuai Yang[3], T.D. Mikesell[3,4], and Catherine Olschanowsky[3]

[1] Intel Corporation, Santa Clara, CA, USA
ravi.shankar@intel.com
[2] Case Western University, OH, USA
aao62@case.edu
[3] Boise State University, Boise, ID, USA
{annarift, tobipopoola, macdonaldlowe, shuaiyang}@u.boisestate.edu,
dylanmikesell@boisestate.edu, catherineolschan@boisestate.edu
[4] Norwegian Geotechnical Institute, Oslo, Norway
dylan.mikesell@ngi.no

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 dataflow 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 dataflow 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 (PDFGs) [9] highlight the dataflow, data access patterns, and execution schedule for applications visually. Originally developed to identify temporary storage reduction opportunities [15], 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 PDFGs and proposes several techniques to ensure correctness and improve their usability.

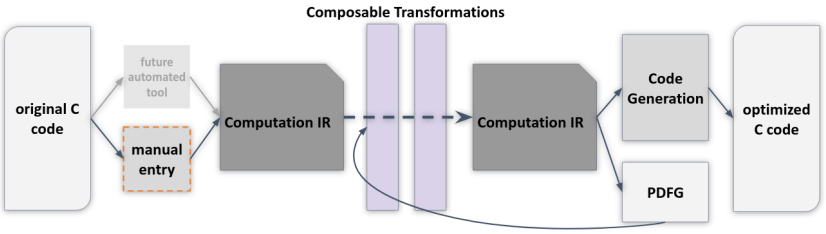


Fig. 1: Optimization Pipeline Overview [16].

Polyhedral dataflow graphs are part of the Sparse Polyhedral Framework (SPF) and are generated automatically from the SPF intermediate representation [16]. 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 PDFGs. 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. It is also possible to automate the choice of transformations. However, automation is not part of the current work.

Previous work demonstrated the concept of PDFGs using manually constructed graphs that represented the execution scheduling using the layout position of nodes and dataflow using edges between nodes. Notably, both data spaces 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 full PDFG for GeoAc is not readable as it contains 4616 statement nodes and several thousand more data space nodes. The goal of PDFGs 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.

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 static single assignment without requiring a control flow graph.
- Proposed alternatives to static single assignment for parameters to the computation that are pointers or references.
- 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, that 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 [12] 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, this is an older tool that does not meet the needs of the workflow. GeoAc [4] is a 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 thus chosen for further analysis. In this case study, we consider the practical implications of viewing the full RK4 function as a Polyhedral Dataflow Graph. It is to be noted that a newer tool called infraGA/GeoAc that includes an MPI implementation has replaced GeoAc. This work is based on the original code base.

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* [11]. 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 [22, 23]. Run-time data reordering techniques attempt to improve the spatial and temporal data locality in a loop by reordering the data based upon the order that it was referenced in the loop [21].

The Computation API [16] 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. This intermediate representation can produce PDFGs [9] and translates graph operations defined for PDFGs into relations used by the Inspector/Executor Generator Library (IEGenLib) [22]. It can also be passed to Omega [17] 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 [22]. 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 [9] 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 intermediate representation is visited and a dot format graph is created.

Figures 3, 5 show the corresponding PDFGs that are generated using the intermediate representation created in Figures 2 and 4. Multiple node types connected by edges comprise the graphs. These node type are variations of statement or data space nodes. Node types include: statements, data spaces,

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", "int*");
denseComp->addDataSpace("A", "int*");
denseComp->addDataSpace("x", "int*");
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 represented using the Computation API.

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 $\{[i, k, j] : 0 \leq i < N \wedge rowptr(i) \leq k < rowptr(i + 1) \wedge j = col(k)\}$. Code generation uses execution schedules to lexicographically order the statements.

All edges represent reads and writes to data spaces 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.

Data space nodes are drawn as rectangles with sharp corners. The representation splits data space nodes into the types listed above. In static single assignment form, every data space should have only one edge pointing into it. An exception is made for parameters to the computation that are pointers or references. Shaded rectangles surround groups of statements that are executed in a common loop nest. A partial form of polyhedral scanning is used to establish the encapsulating spaces. Visually, this helps identify sections of the code that are executed more than once, and are more important for performance.

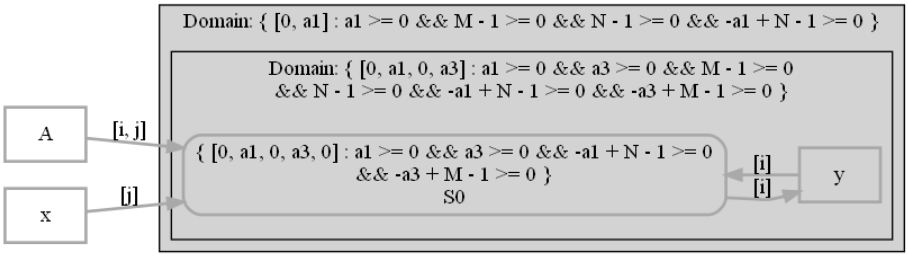


Fig. 3: PDFG for dense matrix vector multiply. `A, x, y` are data spaces, `S0` is a statement, and the `Domain:` labels indicate loop levels.

3 Case Study: Expressing GeoAc and Examining Polyhedral Dataflow Graphs

This case study mimics the behavior of a future automated tool (shown in Figure 1) that will consume existing C code and produce the SPF intermediate representation. In this work, calls to the Computation API are manually written to create the intermediate representation. The SPF tool chain generates code from the intermediate representation and generates the graphs presented in the case study. Results from executing the generated code were compared with those of the original to ensure correctness.

The case study drove the development of the Computation API and demonstrated several shortcomings of PDFGs for a full scientific application. This section overviews the challenges overcome to create accurate dataflow graphs using a sparse polyhedral representation. The first challenge was to create an approximation to static single 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. Figure 6 shows a section of a graph after applying producer-consumer reduction (Section 3.2) and dead code elimination (Section 3.5).

3.1 Approximate Static Single Assignment

The Computation API and polyhedral dataflow graphs support intrinsic types, pointers, and references. User defined types (structs and classes) are not supported. Scientific codes commonly make extensive use of user defined types. All structs and classes must be flattened. All GeoAc structs were converted to a

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", "int*");
sparseComp->addDataSpace("A", "int*");
sparseComp->addDataSpace("x", "int*");
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 represented using the Computation API..

set of data spaces: one corresponding to each member variable. 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. 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 data space names used in the representation.

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 data space written to by a statement was written to by a previous statement, the data space of the previous statement gets a revision number. Affected reads are updated as well. Importantly, the intermediate representation 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 [8] adapted for use with the polyhedral model rather than a control flow graph. If foo is a data space that requires a ϕ node as in Figure 7, 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.

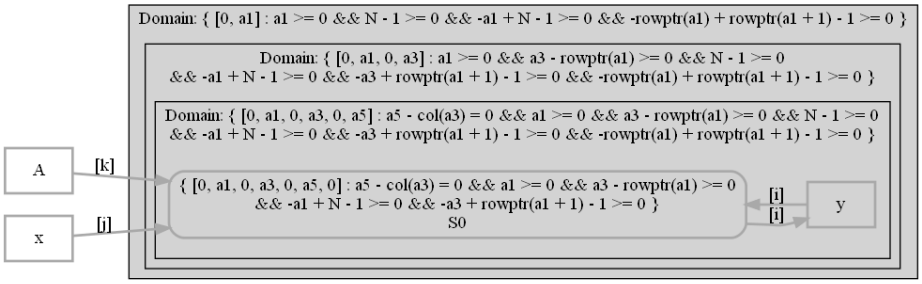


Fig. 5: PDFG for sparse matrix vector multiply. `A, x, y` are data spaces, `S0` is a statement, and the `Domain:` labels indicate loop levels.

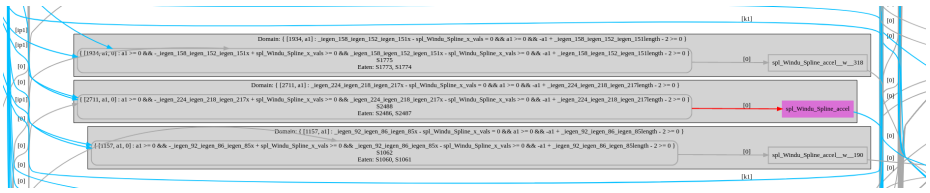


Fig. 6: A subsection of the graph with producer-consumer reductions and dead code elimination applied. The pink node is a parameter that is written to within the loop level.

We begin with the read statement and move backwards through our statements, identifying the first and guaranteed writes. We construct a ϕ node if the first and guaranteed writes are distinct statements. We extract all conditions that apply to the first write but not to the read statement. The ϕ node takes the general form: `foo = conditions ? first write : guaranteed write;`. Due to SSA, the guaranteed write is versioned while first write is left alone. The addition of 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. Figure 8 shows the dataflow graph generated from the ϕ node example in Figure 7.

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.

ϕ 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. 7: ϕ Node Example.

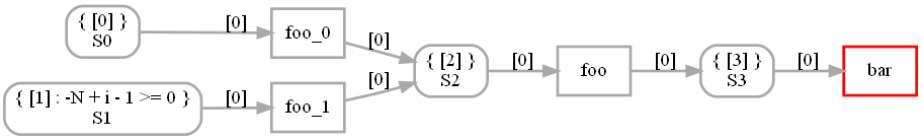


Fig. 8: Graph generated from Figure 7. `foo` is written to once as `foo_0` (S0) and once as `foo_1`(S1). `foo` then chooses between these values based on the given condition (S2) and is subsequently read by `bar` (S3).

3.2 Producer Consumer Reductions

A producer-consumer relationship that can be safely excluded from the visual representation of the graph exists between some statements $S0$ and $S1$ through a data space $D0$ if:

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

An example is shown in Figure 9a.

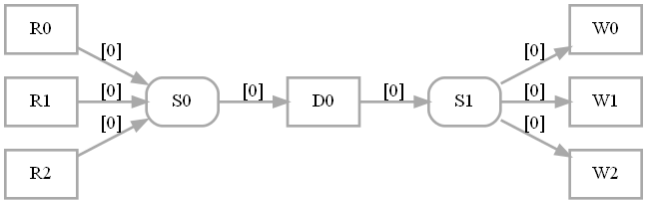
Hiding these simple chains of statements in the graph is similar to grouping statements into basic blocks. This operation should not be mistaken for a producer-consumer fusion. The reduction improves graph readability but does not change the actual computation. The steps of the reduction process are:

- $S0$'s reads are assigned to $S1$.
- $D0$ is removed as a read from $S1$.
- $S0$ and $D0$ are removed from the graph.

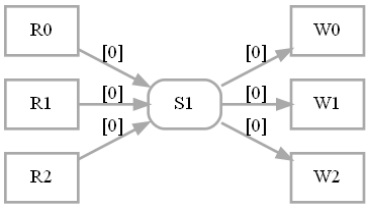
An example result is shown in Figure 9b.

3.3 Graph Components

Within a dataflow graph, there are multiple types of nodes that require distinction. Statements and data spaces are the primary types, denoted by curved and



(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: reads and W's: writes.

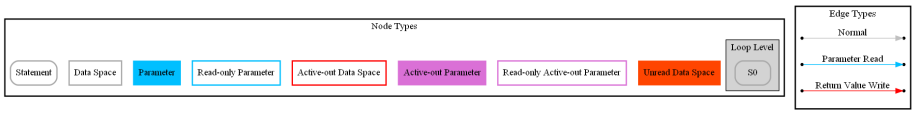


Fig. 10: The different types of components and their colorings

sharp-cornered rectangles, respectively. Statements do not require further classification as they only serve to connect data spaces with a line of code. Data spaces have multiple types: normal, parameter, and active-out. Normal is equivalent to neither parameter nor active-out. From the latter two, we derive active-out parameter, read-only parameter, and read-only active-out parameter. Note that pass-by-reference parameters are active-out parameters and returned values are active-out non-parameters. If a data space is never read from and not active-out, we call it an unread data space. Such data spaces are the target of dead code elimination. Finally, we organize statements into loop levels, corresponding to loops in the source code. Each data space is placed in the loop level of the statement that write to it. We encapsulate grouped nodes in a filled-in gray box with black edges. See Figure 10 for the graph components.

3.4 Data Dependent Control Flow

Each statement in the Computation intermediate representation 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 [3]. 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.

3.5 Dead Code Elimination

Eliminating redundant computations is an optimization technique that improves the performance of an application. In a graph, a statement node is dead if it writes to a data node that is never read from. In our work, we provide this transformation as an option to the Computation intermediate representation. It should be noted that this operation removes dead statements from the Computation intermediate representation. We implement this functionality by performing a breadth first search from dead data nodes in the graph, we keep removing statement nodes recursively until we reach data nodes that are read from by other statements. A breadth first approach allows our algorithm to remove dead statements per each level when traversing the graph backwards. This operation results in a significantly smaller dataflow graph.

3.6 Subgraphs

Due to the immense size of the dataflow graph it is very difficult to visually inspect the dependencies of a single node. To aid in this, it is helpful to remove edges and nodes not connected to the target node. More formally, we define a dependency relationship between node A and node B if B can reach A through either only in-edges or only out-edges. B may reach A through a combination of in and out-edges, however this only implies that B and A share read/write dependencies, not that one depends on the output of the other. We say A has a write dependency with B if B can reach A using only in-edges. We say A has a read dependency with B if B can reach A using only out-edges. Practically, a read dependency indicates that A uses B's output and a write dependency indicates B uses A's output. Figure 11 shows the read dependencies for data space `_iegen39_iegen_31X`.

3.7 Constant Size Arrays

One coding pattern observed in scientific applications such as GeoAc 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.

In our SSA form, arrays are versioned as a whole, meaning that writing to a single index versions the entire array. This causes extra versioning that complicates the view of the dataflow and generates incorrect code. One solution is to detect arrays that are only accessed using literals and replace them

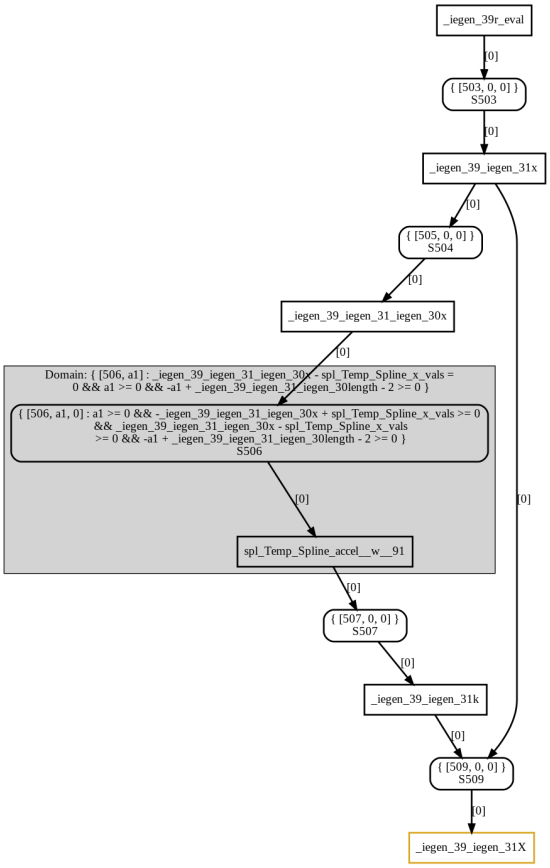


Fig. 11: The read dependencies for `_iegen39_iegen_31X` (highlighted in yellow).

with individual variables. Array accesses using variables whose values can be determined at compile time are also replaced. This process replaces each array access with a data space, combining the array name and indices as follows: $arr[i_0][i_1]...[i_n] \rightarrow arr\_ati_1\_ati_2...\_ati_n$. If reading from the array, unpacking adds a statement of the form "`arr_at0_at1 = arr[0][1];`". SSA safely renames the new data spaces. At the end of the computation, repacking generates statements of the form `arr[0][1] = arr_at0_at1;`. With SSA, the last write to a data space is unversioned so repack statements need not consider renamed data spaces.

Complications arise between constant-access and iterator-access arrays. Iterator accesses do not explicitly write to specific indices. Thus an array must be repacked, undergo iterator access, and be unpacked. A similar issue occurs when an array is copied. In the example of `arr = arr2;` where `arr2` has been previously unpacked, `arr2` must be repacked, copied to `arr`, and unpacked. Currently, unpacking/repacking only acts on constant-access arrays. This addresses

the motivating case of data placed in an array for convenience, not iteration. Handling mixed constant/iterator-access arrays and array copying provide motivating cases for future development.

3.8 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 12. 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.

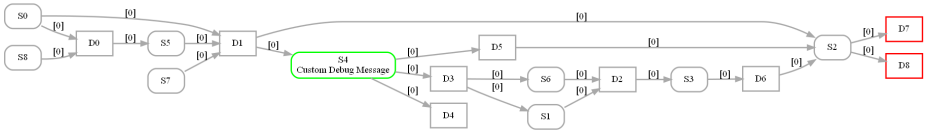


Fig. 12: An example graph that includes a debug statement at $S4$ (highlighted green for easy identification). In the scalable vector graphic format, the user-defined debug string is searchable for easy node identification.

4 Related Work

This work builds on previous work that can be divided into 3 categories: polyhedral tools, sparse polyhedral tools, and similar macro dataflow graphs. Because this work is based on the sparse polyhedral model, the graphs are capable of representing both regular and irregular computations. Additionally, using a scientific application in a use case reveals many of the issues related to graph size. The graphs and associated tool chain build on previous work building the Computation API [16], designing polyhedral dataflow graphs [9], and underlying tools CHiLL [7], IEGenLib [23], and Omega [17].

Polyhedral Tools such as PolyMage [14], Halide [18, 19, 13] and AlphaZ [24] use the polyhedral model to transform regular codes. PolyMage and Halide are

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. 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. AlphaZ [24] expresses transformations using the Alpha equational language and allows for complex memory remapping. Our work differs from the aforementioned work due to our support for non affine polyhedral spaces characteristic with sparse computations.

Work done on representing indirect memory accesses in a computation using the polyhedral model has seen the development of tools such as Omega [10], Chill [7] and the Computation API [16]. Omega [10] is a C++ library for manipulating integer tuple relations and sets. Codegen+ [6] is built on omega and generates code with polyhedral scanning in the presence of uninterpreted functions. Chill [7] is a polyhedral compiler transformation and code generation framework that uses Codegen+ for code generation. It allows users to specify transformation sequences through scripts. Our work differs from this work as we represent a holistic view of a computation and we support more precise transformations in the presence of sparse computations.

Existing work demonstrates the benefit of polyhedral dataflow optimizations. Olschanowsky et al. demonstrated this benefit on a computational fluid dynamic benchmark [15]. Davis et al. automated the experiments from the previous work using modified macro dataflow graphs [9]. This work distinguishes itself by being applied to a full application in a different domain. The Concurrent Collections (CnC) programming model [5] is a dataflow and stream-processing language where a program is a graph of computation nodes that communicate with each other. DFGR [20] is based on CnC and Habanero-C [1] programming models and allows developers to express programs at a high level with dataflow graphs as an intermediate representation. Our work uses the dataflow graph to focus on serial code optimization while DFGR and CnC explores parallelism. Stateful dataflow multigraphs (SDFGs) [2] are a data-centric intermediate representation that enables separating code definition from its optimization. Our work differ from SDFGs due to the use of the polyhedral model. The graphs are not the intermediate representation, but a view of that representation. Any graph operation performed to transform the graph is 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 Computation intermediate representation 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 ϕ nodes for data space 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. PDFGs are generated using the dot format, making the layout of the graph dependent on the dataflow rather than execution order. The graph is displayed left to right and different colors are used to distinguish various graphical elements.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant Numbers 1849463 and 1563818. This research utilized the high-performance computing support of the R2 compute cluster (DOI: 10.18122/B2S41H) provided by Boise State University's Research Computing Department.

References

1. Barik, R., Budimlic, Z., Cavè, V., Chatterjee, S., Guo, Y., Peixotto, D., Raman, R., Shirako, J., Taşırlar, S., Yan, Y., Zhao, Y., Sarkar, V.: The habanero multicore software research project. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. p. 735–736. OOPSLA '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1639950.1639989, https://doi.org/10.1145/1639950.1639989
2. 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
3. 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
4. Blom, P.: Geoac: Numerical tools to model acoustic propagation in the geometric limit. https://github.com/LANL-Seismoacoustics/GeoAc (2014)
5. Budimlić, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., Schlimbach, F., Taşırlar, S.: Concurrent collections. Sci. Program. **18**(3–4), 203–217 (Aug 2010). https://doi.org/10.1155/2010/521797, https://doi.org/10.1155/2010/521797
6. Chen, C.: Polyhedra scanning revisited. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 499–508. PLDI '12, Association for Computing Machinery, New York, NY,

USA (2012). https://doi.org/10.1145/2254064.2254123, https://doi.org/10.1145/2254064.2254123
7. Chen, C., Chame, J., Hall, M.: Chill: A framework for composing high-level loop transformations. Tech. rep. (2008)
8. 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
9. 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)
10. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega library interface guide (1995)
11. LaMielle, A., Strout, M.M.: Enabling code generation within the sparse polyhedral framework. Technical report, Technical Report CS-10-102 (2010)
12. Mikesell, T., Rolland, L., Lee, R., Zedek, F., Coïsson, P., Dessa, J.X.: Ionoseis: A package to model coseismic ionospheric disturbances. Atmosphere **10**(8), 443 (Aug 2019). https://doi.org/10.3390/atmos10080443, http://dx.doi.org/10.3390/atmos10080443
13. 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)
14. 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)
15. 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)
16. Popoola, T., Shankar, R., Rift, A., Singh, S., Davis, E.C., Strout, M.M., Olschanowsky, C.: An object-oriented interface to the sparse polyhedral library. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC). pp. 1825–1831 (2021). https://doi.org/10.1109/COMPSAC51774.2021.00275
17. 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
18. 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)
19. 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)

20. Sbirlea, A., Pouchet, L.N., Sarkar, V.: Dfgr an intermediate graph representation for macro-dataflow programs. In: 2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing. pp. 38–45 (2014). https://doi.org/10.1109/DFM.2014.9
21. 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
22. 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)
23. 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)
24. Yuki, T., Gupta, G., Kim, D., Pathan, T., Rajopadhye, S.: AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In: Kasahara, H., Kimura, K. (eds.) Languages and Compilers for Parallel Computing. pp. 17–31. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)