Parallel Processing: Polyhedral Compilation

Justin Garrigus
University of North Texas
Department of Computer Science and Engineering
justingarrigus@my.unt.edu

Abstract

Polyhedral compilation is a unique branch of computer science and mathematics aimed at rewriting loop statements with inter-iteration dependencies to run in parallel through geometric transformations. By computing scattering matrices and applying them to a algebraic representation of code, a loop that was previously strictly sequential can be automatically transformed into one that can run across multiple cores without any data dependencies. This work implements a source-to-source compiler aimed at automating the polyhedral compilation process by performing manual transformations on code sequences with custom-designed `#pragmas` and by utilizing three popular tools built for the domain including OpenScop, Clan, and CLooG. To avoid computing complicated abstract linear algebra formulas that require additional tools to be installed, a random search of suitable transformations is performed automatically and the optimized kernel with the highest performance that preserves the semantics of the original program is chosen. Our compiler achieves better performance on two out of three input kernels compared to a non-optimized equivalent.

1. Introduction

Modern high-performance computing places an extreme emphasis on improving the level of parallelism that can be obtained from programs, including thread-level parallelism that run discrete instructions across multiple cores and instruction-level parallelism that exploit vectorized hardware units and pipelining to spread computation across a single core. Although, the benefit of these systems depend entirely on the ability of the programmer to implement them. Code which is not designed to run on multiple cores or code that has an otherwise large reliance on sequential execution would not benefit from modern hardware improvements. Thus, an automatic software-based solution is beneficial in alleviating the difficulty from the programmer.

One popular hardware pattern popularized in the 1970s was the concept of systolic arrays [12]. Featuring direct node-to-node connections rather than a shared memory space among all nodes, the hardware optimization greatly improved the amount of computing work that could be performed without operators needing to save and load data through an extensive and often unpredictable memory hierarchy. It introduced a big limitation in programming ability, as the convenience of a shared memory space among functional units needed to be removed in favor of structured parallelism, but certain computing domains like deep learning, image processing, and scientific computing still benefit from it today.

Some algorithms are naturally structured as systolic arrays, where a multi-dimensional loop has elements of an array computed in one iteration that are used across other iterations. If this array is not easily divisible across an axis of the loop, then naive processors must resort to executing them sequentially. To combat this, the domain of polyhedral compilation was developed to restructure loops so parallel axes can be exposed.

An example of polyhedral compilation taken from our compiler is given in appendix A, depicting the task of Gaussian elimination. The unoptimized code example contains a three-dimensional for-loop with two statements, where no single axis can be simply rewritten via OpenMP calls to run in parallel due to dependencies among loop iterations. Polyhedral compilation consists of a number of abstract sub-optimizations performed implicitly, including loop skewing, fusion, shifting, tiling, distribution, and more. As such, it can be optimized to run across multiple processors, to use a smaller code size or control overhead, and to prioritize local memory accesses across small time frames to exploit the cache.

Instead of representing these optimizations as a typical compiler would, polyhedral compilation is made up entirely of linear algebra operations. Code accesses and transformations are internalized as matrices and linear relationships, and a sequence of math operations can lead to the generation of different optimized kernels. Due to the regular accesses certain systolic program types exhibit on fixed-size arrays, the conversion of code to matrices and the reconversion back to optimized code can be performed easily.

This paper describes a simple approach to generating code under the polyhedral model. Instead of taking a complex, time-consuming approach involving abstract linear algebra, we implement a grid-based transformation generator that proposes a search-space and a automated executor that yields an optimized kernel from unoptimized user code. The compiler extracts code marked by C `#pragma` statements, converts it to matrix representation through OpenScop and Clan, computes a collection of candidate transformation matrices, tests the resulting program for semantics and performance, reinserts the optimal kernel translated by CLooG into the original program, and passes the entire source file through GCC for the user to execute.

The rest of the paper is organized as follows. Sec. 2 gives an overview of the polyhedral model and a step-by-step walkthrough of how it can be used to optimize systolic programs. Sec. 3 describes implementations of compilers and optimizers in the polyhedral domain since its inception in the 1960s to the present day. Sec. 4 shows a top-down view of our compiler and an abstract look at how it transforms kernels. Sec. 5 shows specific implementation details, including how it was programmed, the different tools involved, what data is communicated and translated, and more. Sec. 6 depicts the results of the compiler on a collection of short kernels and shows how it compares with sequential implementations and we conclude in Sec. 7.

2. Problem Statement

The basis of polyhedral compilation lies in structured, restrictive iteration. For a loop to be a candidate for optimization, the bounds that the loop will traverse must be known before it is entered. Generally speaking, the compilation process identifies Static Control Parts (SCoP) that feature consecutive instructions with the only allowable control structures being either for-loops or if-statements. Functions are allowed only if they are *purely functional*, meaning they do not have any side effects and will always yield the same outputs given the same set of inputs. Both scientific and signal-processing applications feature many SCoP that match this description [18], an example of which is shown in algorithm 1. The result of these restrictions is the ability to represent SCoP algebraically.

SCoP are optimized best by the polyhedral model if they take the form of systolic operations, with values from one iteration feeding directly into the next. Algorithm 1 matches this description, and its *index space* and *dependency graph* is given in figure 1 to demonstrate this. The algorithm cannot be simply parallelized on a single axis due to the inner-loop changing values that are needed across iterations; algoithms which are parallelizable across a loop axis would feature only horizontal or vertical lines in their dependency graph, like matrix multiplication or 2D convolution.

Algorithm 1 SCoP pseudocode

```
Require: $n, a[n], b[n], c[n]$
1: for $i \leftarrow 0, n$ do
2:     for $j \leftarrow n, 0$ do
3:         if $i == 0 \parallel j == n$ then
4:             $c[i+j] \leftarrow a[i] \times b[j]$                ▷ S1
5:         else
6:             $c[i+j] \leftarrow c[i+j] + a[i] \times b[j]$      ▷ S2
7:         end if
8:     end for
9: end for
```

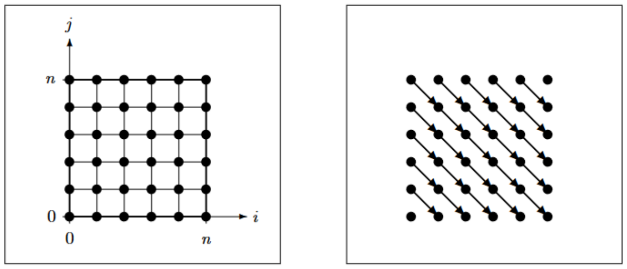

Figure 1. Index space (left) and dependency graph (right) of algorithm 1.

Once a SCoP is identified, algebraic features can be extracted from it. The first is the *iteration domain*–a collection of integral points that a loop visits–which is the basis for representing an index space. For algorithm 1, this would be:

$$\{D \in Z^2 | 0 \le i \le n \parallel 0 \le j \le n\}$$

It is important to note that polyhedral optimization only works on *affine* expressions, which are the result of a skew and a translation on the original set. As such, functions are not allowed as the iterator or condition in target for-loops, except for the special integral functions `min`, `max`, `ceil`, and `floor`. An iteration domain whose condition, iterator, and shape match these descriptions is known as a `Z-polyhedron`.

The ultimate goal is to shift the iteration domain so that the same points are visited but in a different order. The final dependency graph for algorithm 1 should be completely vertical or horizontal, which means a single transformed axis or loop could run in parallel, for instance by adding a tag such as `#pragma omp parallel for`. Other target polyhedral graphs could be obtained, such as graphs that minimimize the local access scope of index spaces rather than to maximize parallelism in order to support cache blocking, but this paper focuses specifically on simple parallelism.

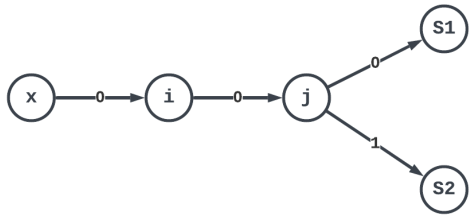

Figure 2. Abstract Syntax Tree (AST) for algorithm 1.

The iteration domain shown previously does not specify an ordering of the points, which is shown by the iterators for i and j looking similar despite iterating in opposite directions. To remedy this, a *scattering* is created that attaches an ordering to each individual statement as a function of the iterators. Scatterings yield a vector such that the statement *instance* that runs first precedes those that come later based on a lexicographic order. A simple range-based for-loop would have a single-dimensional scattering created from the loop's iterator, but more complicated multidimensional loops would likely require creative multidimensional scattering functions. The main point of research among the polyhedral community is finding optimal scattering functions.

The easiest way to create a scattering that properly represents the original program is to use the program's abstract syntax tree (AST). For algorithm 1, this would look like figure 2. A statement's scattering vector is obtained by traversing the tree: for instance, `S2`'s scattering vector would be "`0i0j1`". AST's were used as a test for our implementation to see if scattering generation worked and their implementation is described in section 5, but they do not yield new transformations. To generate optimal code, scatterings need to be found algebraically.

Scattering functions are not concretely defined in any code sense, and they reflect abstract ideas of ordering more than they represent manifestations of computation concepts like parallelism and blocking. Two common terms used to characterize scatterings are *schedules* (scatterings which give each statement instance an "execution date") and *allocations* (scatterings which divide statements among processors). A scattering which combines schedules and allocations is a *space-time mapping*, and is the focus of this paper.

Once these items are determined, they can be translated into different forms for easy manipulation. Iteration domains are essentially a set of inequalities that define a traversal bounds, so they can in turn become *domain matrices*, where algorithm 1's relational set and domain matrix is shown below.

$$\begin{cases} i \geq 0 \\ -i+n \geq 0 \\ j \geq 0 \\ -j+n \geq 0 \end{cases} \iff \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Since scattering functions naturally yield vectors, they can interface neatly with iteration domains. Although, the typical method for generating a scattering matrix, or *transformation* as its referred to throughout this paper, can become very complex. The common method is to apply the affine form of the Farkas Lemma [21] to yield special multipliers, then create a dependence matrix from another set of relations with the multipliers as unknowns, and iteratively satisfying relations until a collection of valid transformations can be obtained [18].

The final step in the compilation process is to translate a source program with a computed transformation into a new representation. The transformation should essentially be an affine expression applied to the original statements, shifting loop bounds or creating new loops when necessary, and can be performed automatically with specialized tools. The result of this process is a source-to-source compiler that takes inefficient systolic loops and removes the dependencies that occur between iterations to allow for potentially massive performance gains.

3. Related Work

The polyhedral model was originally outlined in [11], which described how uniform recurrence equations created from the dependency graph of lattice points in geometric space can be scheduled in parallel. The idea was first applied to the domain of compilation in [13], implementing the *hyperplane method*–where hyperplanes describe edges of the polytope or iteration domain–for Fortran and Algol compilers.

The concept of systolic arrays as a hardware optimization was developed in [12] and several additional contributions came after, with [16] mapping cyclic loop algorithms to VLSI systems, [9] representing array computation as geometric and using linear transformations to change their shape, [15] embedding space-time computation graphs with dependency graphs to map systolic arrays to physical arrays, and [19] recognizing the need for finding schedule and allocation scatterings that map one iteration domain to another to create a fully-automated translation method. Each of these papers and more are organized in [20] for applying the method in signal processing, graph theory, numerical linear algebra, and other domains.

Several more seminal papers influenced our efforts. [14] gives an overview of the polyhedral model from a theoretical and practical standpoint, and compiles together much of the introductory material for the polyhedral domain. [18] and [17] show how to iteratively create candidate one-dimensional and two-dimensional scatterings, respectively, and how the scatterings can yield a traversable search-space of candidate optimizations. [2] brings the polyhedral domain to the GPU and optimizes affine loop nests represented in CUDA code. [22] and [1] use the polyhedral representation to improve tensor operator scheduling for deep learning systems on GPUs and other accelerators.

There are many tools available that are designed for polyhedral compilation specifically, several of which are used as core parts of our compiler's pipeline. OpenScop [5, 7] is a specification of data structures meant to unify all other polyhedral tools, and an included API gives ways of representing SCoP, iteration domains and polyhedra, scattering relations, and transformation matrices. Clan [6, 8] gives a few methods for scraping the polyhedral sections from a given source code file and turning it into OpenScop data structures. CLooG [3, 4] is a code generator that takes SCoP and scatterings to yield equivalent C or Fortran code, and is used to create the resulting transformed code after optimization.

Other papers that are not strictly related to polyhedral compilation but were still necessary for creating this paper include the TVM deep-learning compiler [10], which lets users define abstract requirements for deep-learning modules in order to create a search-space that is traversed on hardware to choose an optimal compiled output, and [23], which describes bottlenecks for graph applications on GPU systems and gives a helpful methodology for identifying areas for improvement in software domains. While this paper is about simple array-based polyhedral compilation on the CPU, taking inspiration from other sources like these was necessary to complete the project.

4. Proposed Method

The main difficulty of our implementation was following along with the math described in previous work. Many papers are involved with scattering-generation in the polyhedral model, but few of these approaches define the computation in terms of concrete operations. Steps are instead described through lemmas, axioms, relations, and equations involving abstract symbols; without a working prototype to view the concrete details of, it was difficult for us to make sense of the real inner-workings of the approaches on a non-theoretical basis. As such, due to time constraints, we instead opted for a different approach for transformation creation.

Since scatterings are easily defined in terms of systems of inequalities, and furthermore as matrices of limited size with values of definite bounds, we can perform a random search of naively generated candidate transformations without performing the "provably-correct" approaches. Although, the size of the search spaces grows exponentially with program size leading to a combinatorial explosion of viable searches, only few of which are *semantically correct*. Coupled with our desire for automatic parallelization, this leads to a compiler which is not guaranteed to produce a correct, efficient program output in any reasonable amount of time. Regardless, it was the most viable solution to our project constraints, so we will leave a deterministic solution up to future work. A complete pipeline of our approach is shown in figure 3. Our compiler is essentially a wrapper to the GCC compiler. It takes source code with added `#pragma` statements with other files, applies transformations where applicable, and forwards the result to GCC.

The user should edit their source code to indicate areas they want to have parallelized through the polyhedral model, shown in listing 1. These sections must represent SCoP, which incurs the limitations of loop iteration and boundings, but users should also specify a `setup` and `teardown` subsection. Our compiler will *test* if a generated transformation is viable by creating and executing only the polyhedral code, with the number of tests executed on any transformation equal to the value specified in the `#pragma polyhedral` header's `test` argument. This shortened test file will contain the `setup` and `teardown` code, which generates dummy variables for the optimized code to operate on. The `teardown` section should also include a float *output*, giving a simple way for the compiler to tell if a transformed program is semantically equivalent to the "baseline" unoptimized program. This entire polyhedral section is embedded directly within the code of a user's program; in this case, the matrix multiplication algorithm specified (between `#pragma endsetup` and `#pragma teardown`) was originally in a program's main method.

First, the compiler scans through each input file for sections marked for polyhedral compilation, and extracts header values and sub-sections (including `setup`, `body`, and `teardown`), from each area, raising compilation errors whenever missing or unexpected tokens are detected. Drivers and stubs are created as well: optimized code will be placed in their own function, and test-cases will be generated with only a main method and the polyhedral code. This also requires adding `#include` and `#define` directories and function declarations to the test-case basis when applicable. Identifiers must be renamed in certain cases–like with preprocessor directives defining constant values, which disallows using those same definitions as the names of parameters–so mappings are created.

Next, an abstract syntax tree is obtained from polyhedral bodies. This is used as a scattering for the baseline configuration, since an AST scattering is guaranteed to return the original code after code transformation through CLooG.

Afterwards, the polyhedral body is translated into an OpenScop representation through Clan. Although, Clan has

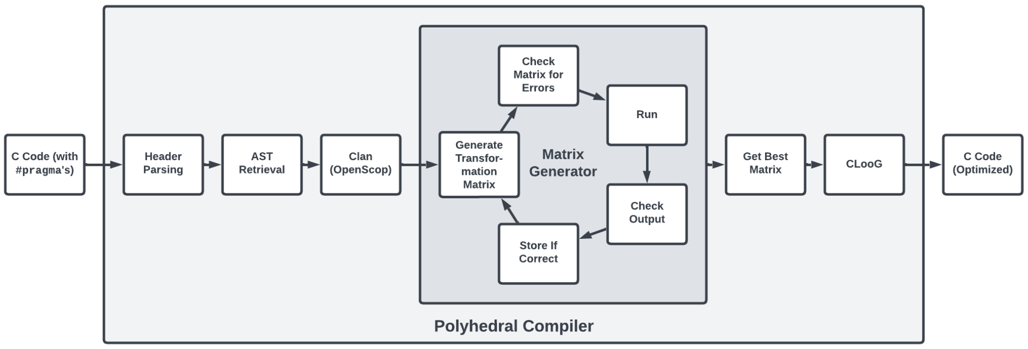

Figure 3. Proposed polyhedral compiler pipeline.

a limitation in that it can only parse code that (1) comes from a file, and (2) is surrounded by special `#pragma scop` headers, which requires more preprocessing. This generates a unified object containing statement mappings used by CLooG for re-translation and scattering stubs we can edit to represent transformations. An example output of the translation process for a Gaussian elimination kernel is shown in appendix B.

Now we can proceed to the main contribution of our work with transformation generation. At this point, Clan returned a stub for scattering inequality relations which can be forwarded to CLooG for code generation. These relations are better visualized as a matrix, and the dimensionality of the matrix can be user-defined. For simplicity, we make our matrix have the same dimensions as the longest statement path in the baseline AST, which makes testing easy as the baseline's transformation matrix is the same dimensionality of all other tests.

As mentioned previously, the transformation matrix is an abstract concept not concretely defined in terms of code generation. As such, without a proper mathematical perspective to view the values through, we resorted to identifying certain patterns in their representation. Matrix rows corresponded to coefficients to target iterators, array accessors, and conditional bounds, and all programs used between 5 and 8 coefficients. An example of this can be seen in appendix B. We noticed the following patterns held:

1. Matrix values needed to be integers in the set $x \in \{-1, 0, 1\}$. Values which were not in this set never generated valid transformations.

2. At least one value in each column needed to be non-zero.

3. High-entropy matrices (with lots of values set to non-zero numbers) tended to result in unpredictable programs with extreme outputs, while low-entropy matrices were usually invalid.

We use these patterns as a basis for generating our own matrices while forming our search space. Additionally, to automatically parallelize the final program, we add `#pragma omp parallel for` to random generated loop statements. Since we cannot know if a loop axis is truly independent of other iterations, the random placement of parallel indicators would need to suffice.

Our compiler continuously generates candidate transformations and tests them in short code segments by placing them in new temporary files with the preprocessor directives and header fields obtained in an earlier stage. After creating a test, it is passed through GCC, executed a number of times, and its returned results are compared with the baseline. If the program outputs are always equivalent to the baseline's then it is considered to be semantically equivalent and it is stored; otherwise, it is removed.

The user passes their source code files through our compiler just as they would with GCC, and they specify arguments that determine how many tests to execute and whether or not to clean temporary intermediate files. Once the search concludes, the list of semantically-correct programs is searched and the one with the best performance is passed along to GCC for final compilation.

5. Implementation

The main difficulty of the project was learning every tool and connecting them together. The tools presuppose that programmers have a high knowledge of polyhedral compilation already, but they come equipped with a Doxygen interface for exploring the code. Besides this, there are three main areas the implementation focused that will be described: I/O, transformation organization, and transformation checking.

5.1. I/O

The I/O module aimed to both extract relevant features from C-source inputs and to output organized represen-

Listing 1. Example code input to polyhedral compiler featuring matrix multiplication.

```
#pragma polyhedral \
  array(float** a, float** b, float** c) \
  iter(int i, int j, int k) \
  global(int N) test(5)
{
  #pragma setup
  srand(13);
  float a[N][N];
  float c[N][N];
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      a[i][j] = (float)rand() /
        (float)(RAND_MAX / 10);
      c[i][j] = (float)rand() /
        (float)(RAND_MAX / 10);
    }
  }
  #pragma endsetup

  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      c[i][j] = 0;
      for (int k = 0; k < N; k++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }

  #pragma teardown
  float norm = 0;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      norm += c[i][j];
    }
  }
  printf("%f", norm);
  #pragma endteardown
}
#pragma endpolyhedral
```

tations of the polyhedral model. This included parsing the AST from the polyhedral body and outputting test sequences from a given transformation.

The AST performs a line-by-line reading of C source code and generates a mapping of the relationships between containers and their contained values (e.g., loop statements and their associated blocks). In order to simplify the parsing process, Clang was originally used as a backend for parsing programs for their AST, but we ran into a number of difficulties in parsing program segments as opposed to full source files and in handling `#pragma` statements. As such, we perform a manual reading of the input string in plain C instead.

An example of the output of the AST module is shown in figure 4. With it generated, a path is collected from the root node to each statement, and the size of each statement's transformation matrix is normalized to the largest AST path. For example, statement S3's path would be "`0i2j0k0`".

Once created, each statement can fill in their respective matrices according to the path sequence. An example of S3's transformation matrix is:

$$\begin{cases} S1(i,j) & = (0,i,0,j,0,0,0) \\ S2(i) & = (0,i,1,0,0,0,0) \\ S3(i,j,k) & = (0,i,2,j,0,k,0) \\ S4(i,j) & = (0,i,2,j,1,0,0) \end{cases}$$

$$\left[\begin{array}{ccccccccccccc} eq & p1 & p2 & p3 & p4 & p5 & p6 & p7 & i & j & k & n & 1 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

Since matrices can be visualized as collections of linear relations, we assign a diagonal of values corresponding to iteration multipliers and match their associated mapped parameters (e.g., iterators `i`, `j`, and `k`; global parameter `n`; and constant `1`).

5.2. Transformation Organization

AST's form the basis of transformation generation, as all transformations build off it as a starting point. The baseline configuration without any transformation is a special case where no modification is performed to the matrix, but all other tests modify their matrix in a random way following the patterns denoted earlier.

At compilation time, the user specifies how many tests to search through and how many executions to perform for each test in order to confirm an optimization is semantically correct. For transformation generation, random values are seeded with each test's index. This has a benefit for allowing us to store only the seed of tests that produce viable results, since the seed is all that is needed in order to recreate an optimized kernel.

Along with needing random numbers to fill in transformation matrices, more random numbers are required to indicate which loops in the generated code should run in parallel. Since we are unable to check if a loop contains code that can run independent of other iterations, the random values ensure that at least a fraction of properly-optimized kernels can become automatically parallelized without worrying about (1) completely removing multithreading or (2) parallelizing a loop with inter-iteration dependencies.

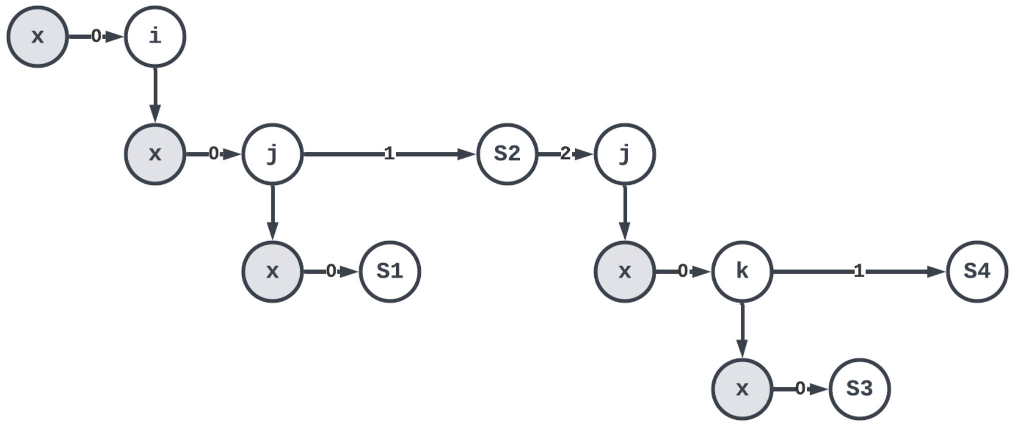

Figure 4. Example AST for a Cholesky factorization kernel.

5.3. Transformation Checking

The random nature of our searching procedure in regards to the size of transformation matrices and output code size leads to a combinatorial explosion, which is why typical polyhedral compilers employ deterministic approaches to generate output code that is sure to be semantically correct. For instance, if a typical program has three iterators, one global variable, and a constant field, all with a longest-path AST that is 8 statements long, then this leads to a matrix with $path_{longest} \times (n_{iterator} + n_{global} + n_{constant}) = 8 \times (3 + 1 + 1) = 40$ values to set. Furthermore, if we follow our aforementioned pattern to assign a random integer value $x \in \{-1, 0, 1\}$ to each matrix element, then there would be $3^{size} = 3^{40} = 1.2 \times 10^{19}$ possible combinations of matrices.

In addition, not all of these matrices are valid. We have already discussed the problem of generating semantically incorrect code–fixed by having polyhedral test code return a single value to denote the "correctness" of an execution–but CLooG may yield a segmentation fault when given certain matrices. The reasons for these faults are likely due to the contents of the matrix itself, as CLooG may not be capable of generating output code for a transformation that is completely incorrect. To fix this, we changed CLooG's source code and added a short error-checking routine before output code is generated that scans their internal AST data structure for iteration variables which are `NULL`.

This fixes the issue most of the time, but certain cases may still result in segmentation faults. As a final resort, we created a bash script that acts as a wrapper to our compiler; in the case of a segmentation fault, it restarts the test routine to the index succeeding the failed case. This fixes any issues we received, and allowed the compilation process to continue without interruption.

6. Performance Evaluation

Our excessive usage of random values has some benefits in that it allows us to traverse a search space without requiring complex algebra to do so, but it results in a compiler that takes very long to run, is not guaranteed to converge, and will not always generate semantically correct code output. As such, the results of our compilation tests are volatile.

We tested our compiler on three input kernels: matrix multiplication, Cholesky factorization, and Gaussian elimination. The results are shown in table 1, along with the size of the kernel in lines of code, the runtime of the baseline, the runtime of the best result, and how many tests were required to get the best result. Each baseline ran sequentially with no OpenMP calls, while the resulting kernels possibly contained parallel loops. Programs were executed on a 2.30GHz Intel i7 CPU. Also, GCC had no optimization flags enabled in any case, as to demonstrate the performance benefits from polyhedral optimization only.

A consequence of our method was the length of time required to generate valid tests. Most tests were completely invalid: Cholesky factorization and Gaussian elimination only had around 10 semantically-valid generated kernels out of 10,000 attempts. We attempted to implement a procedure to improve this by reducing the problem size of input kernels, since large problem sizes (e.g., input matrix sizes for matrix-multiplication) lead to noticeable time differences between unoptimized and optimized code but overall slower execution. Our procedure replaces global definitions with smaller values in order to generate a set of kernels that lead to semantically-correct results, and then re-runs this culled set on the original problem size to get better timing statistics, which is the approach we used to run more tests in a short amount of time.

It is important to note that our matrix-generation method

Kernel	Size (LOC)	Baseline time (s)	Optimized time (s)	Tests performed
Matrix Multiplication	5	0.381	0.310	1,510
Cholesky Factorization	8	0.462	0.295	10,572
Gaussian Elimination	5	0.489	0.554	12,143

Table 1. Execution results for each compiled kernel.

will lead to optimized kernels eventually. Our culled-pattern approach of generating specific values within the range $x \in \{-1, 0, 1\}$ could be changed for a larger range if applicable, and we could entirely remove the restrictions that all columns of the relation matrix must have at least one nonzero value within it. This would guarantee our results eventually converge with existing techniques, but at the cost of navigating more search transformations than feasibly possible.

An example of optimized code outputted from our polyhedral compiler is given in appendix A.

7. Conclusion

The polyhedral model can be used to remove inter-iteration dependencies, allowing an axis of a multidimensional loop to be parallelized. Although making an automatic parallelizer through the polyhedral model requires proficient understanding of abstract linear algebra, we created a source-to-source compiler that randomly generates candidate transformation matrices, runs each resulting test, and selects the highest-performing optimized kernel automatically. While our results do not indicate that a random search of transformation matrices would lead to an efficient or correct compiler, we hope to develop our understanding of compilers more in the future through similar projects.

References

[1] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S.V. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 138–149. 4

[2] M.M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS'08)*, pages 225–234, 2008. 4

[3] C. Bastoul. Code generation in the polyhedral model is easier than you think. *IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT'13)*, pages 7–16, 2004. 4

[4] C. Bastoul. Cloog documentation, 2007. 4, 9

[5] C. Bastoul. Openscop: A specification and a library for data exchange in polyhedral compilation tools. 2011. 4

[6] C. Bastoul. Clan documentation, 2014. 4

[7] C. Bastoul. Openscop documentation, 2014. 4

[8] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. *International Workshop on Languages and Compilers for Parallel Computers (LCPC'16)*, pages 209–225, 2003. 4

[9] P.R. Cappello and K. Steiglitz. Unifying vlsi array designs with geometric transformations. *International Conference on Parallel Processing (ICPP'83)*, 1983. 3

[10] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. *USEIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, 2018. 4

[11] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 1967. 3

[12] H.T. Kung and C. E. Leiserson. Algorithms for vlsi processor arrays. *Addison-Wesley*, pages 245–282, 1978. 1, 3

[13] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, 1974. 3

[14] C. Lengauer. Loop parallelization in the polytope model. *4th International Conference on Concurrency Theory (CONCUR'93)*, pages 398–416, 1993. 3

[15] W. M. Miranker and A. Winkler. Spacetime representations of computational structures. *Computing*, 2:93–114, 1984. 3

[16] D.I. Moldovan. On the design of algorithms for vlsi systolic arrays. *Proceedings of the IEEE*, 71(1), 1983. 3

[17] L.N. Pouchet, C. Bastoul, A. Cohen, and J. Cavanos. Iterative optimization in the polyhedral model: Part ii, two-dimensional time. *ACM SIGPLAN Notices*, 2008. 4

[18] L.N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. *IEEE/ACM International Symposium on Code Generation and Optimization (CGO'07)*, 2007. 2, 3, 4

[19] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA'84)*, pages 208–214, 1984. 3

[20] S.K. Rao. Regular iterative algorithms and their implementations on processor arrays. *Ph.D. Thesis*, 1986. 3

[21] A. Schrijver. Theory of linear and integer programming. *John Wiley & Sons*, 1986. 3

[22] S. Verdoolaege, J.C. Juega, A. Cohen, J.I. Gomez, C. Tenilado, and F. Catthoor. Polyhedral parallel code generation

for cuda. *ACM Transactions on Architecture and Code Optimization*, 9(4):1–23, 2013. 4

[23] Q. Xu, H. Jeon, and M. Annavaram. Graph processing on gpus: Where are the bottlenecks? *IEEE International Symposium on Workload Characterization (IISWC)*, 2014. 4

A. Optimized Polyhedral Code

Our initial tests revolved around replicating the results found in the CLooG documentation [4], so our compiler is biased around generating transformation matrices that are similar to the one they provide as an example. The original unoptimized Gaussian elimination kernel is shown in listing 2, and the optimized version is given in listing 3.

Listing 2. Gaussian elimination baseline kernel

```
for (int i = 0; i < N;i++) {
  for (int j = i + 1; j <= N; j++) {
    c[i][j] = a[j][i] / a[i][i];
    for (int k = i + 1; k <= N; k++) {
      a[j][k] -= c[i][j] * a[i][k]
    }
  }
}
```

Listing 3. Gaussian elimination kernel optimized through polyhedral compiler

```
if (n >= 2) {
  for (int b2 = 2; b2 <= n; b2++) {
    c[1][b2] = a[b2][1] / a[1][1];
  }
}
#pragma omp parallel for
for (int b0 = 2; b0 <= n - 1; b0++) {
  for (int b1 = 1; b1 <= b0 - 1; b1++) {
    for (int b2 = b1 + 1; b2 <= n; b2++) {
      a[b2][b0] -= c[b1][b2] * a[b1][b0];
    }
  }
  for (int b2 = b0 + 1; b2 <= n; b2++) {
    c[b0][b2] = a[b2][b0] / a[b0][b0];
  }
  if (n >= 2) {
    for (int b1 = 1; b1 <= n - 1; b1++) {
      for (int b2 = b1 + 1; b2 <= n; b2++) {
        a[b2][n] -= c[b1][b2] * a[b1][n];
      }
    }
  }
}
```

B. OpenScop Output

Listing 4 shows the format that the polyhedral tools used in this project communicate in. Of particular interest are the Scattering section showing the different relations and iteration parameters, and the fact that each statement has its own unique scattering.

Listing 4. OpenScop output generated via Clan from a source-code file containing Gaussian elimination.

```
# [File generated by the OpenScop Library 0.9.5]

<OpenScop>

# =============================================== Global
# Language
C

# Context
CONTEXT
0 3 0 0 0 1

# Parameters are provided
1
<strings>
N
</strings>

# Number of statements
2

# =============================================== Statement 1
# Number of relations describing the statement:
5

# ----------------------------------------------- 1.1 Domain
DOMAIN
5 5 2 0 0 1
# e/i|  i    j |  N |  1
   1    1    0    0    0    ## i >= 0
   1   -1    0    1   -2    ## -i+N-2 >= 0
   1    0    0    1   -2    ## N-2 >= 0
   1   -1    1    0   -1    ## -i+j-1 >= 0
   1    0   -1    1   -1    ## -j+N-1 >= 0

# ----------------------------------------------- 1.2 Scattering
SCATTERING
5 10 5 2 0 1
# e/i| c1   c2   c3   c4   c5 |  i    j |  N |  1
   0   -1    0    0    0    0    0    0    0    0    ## c1 == 0
   0    0   -1    0    0    0    1    0    0    0    ## c2 == i
   0    0    0   -1    0    0    0    0    0    0    ## c3 == 0
   0    0    0    0   -1    0    0    1    0    0    ## c4 == j
   0    0    0    0    0   -1    0    0    0    0    ## c5 == 0

# ----------------------------------------------- 1.3 Access
WRITE
3 8 3 2 0 1
# e/i| Arr  [1]  [2]|  i    j |  N |  1
   0   -1    0    0    0    0    0    4    ## Arr == c
   0    0   -1    0    1    0    0    0    ## [1] == i
   0    0    0   -1    0    1    0    0    ## [2] == j

READ
3 8 3 2 0 1
# e/i| Arr  [1]  [2]|  i    j |  N |  1
   0   -1    0    0    0    0    0    5    ## Arr == a
   0    0   -1    0    0    1    0    0    ## [1] == j
   0    0    0   -1    1    0    0    0    ## [2] == i

READ
3 8 3 2 0 1
# e/i| Arr  [1]  [2]|  i    j |  N |  1
   0   -1    0    0    0    0    0    5    ## Arr == a
   0    0   -1    0    1    0    0    0    ## [1] == i
   0    0    0   -1    1    0    0    0    ## [2] == i

# ----------------------------------------------- 1.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
2
# List of original iterators
i j
# Statement body expression
c[i][j] = a[j][i] / a[i][i];
</body>

# =============================================== Statement 2
# Number of relations describing the statement:
6

# ----------------------------------------------- 2.1 Domain
DOMAIN
7 6 3 0 0 1
# e/i|  i    j    k |  N |  1
   1    1    0    0    0    0    ## i >= 0
   1   -1    0    0    1   -2    ## -i+N-2 >= 0
   1    0    0    0    1   -2    ## N-2 >= 0
   1   -1    1    0    0   -1    ## -i+j-1 >= 0
   1    0   -1    0    1   -1    ## -j+N-1 >= 0
   1   -1    0    1    0   -1    ## -i+k-1 >= 0
   1    0    0   -1    1   -1    ## -k+N-1 >= 0

# ----------------------------------------------- 2.2 Scattering
SCATTERING
```

```
7 13 7 3 0 1
# e/i| c1   c2   c3   c4   c5   c6   c7 |  i    j    k |  N |  1
   0   -1    0    0    0    0    0    0    0    0    0    0    0    ## c1 == 0
   0    0   -1    0    0    0    0    0    1    0    0    0    0    ## c2 == i
   0    0    0   -1    0    0    0    0    0    0    0    0    0    ## c3 == 0
   0    0    0    0   -1    0    0    0    0    1    0    0    0    ## c4 == j
   0    0    0    0    0   -1    0    0    0    0    0    0    1    ## c5 == 1
   0    0    0    0    0    0   -1    0    0    0    1    0    0    ## c6 == k
   0    0    0    0    0    0    0   -1    0    0    0    0    0    ## c7 == 0

# ---------------------------------------------  2.3 Access
READ
3 9 3 3 0 1
# e/i| Arr  [1]  [2]|  i    j    k |  N |  1
   0   -1    0    0    0    0    0    0    5    ## Arr == a
   0    0   -1    0    0    1    0    0    0    ## [1] == j
   0    0    0   -1    0    0    1    0    0    ## [2] == k

WRITE
3 9 3 3 0 1
# e/i| Arr  [1]  [2]|  i    j    k |  N |  1
   0   -1    0    0    0    0    0    0    5    ## Arr == a
   0    0   -1    0    0    1    0    0    0    ## [1] == j
   0    0    0   -1    0    0    1    0    0    ## [2] == k

READ
3 9 3 3 0 1
# e/i| Arr  [1]  [2]|  i    j    k |  N |  1
   0   -1    0    0    0    0    0    0    4    ## Arr == c
   0    0   -1    0    1    0    0    0    0    ## [1] == i
   0    0    0   -1    0    1    0    0    0    ## [2] == j

READ
3 9 3 3 0 1
# e/i| Arr  [1]  [2]|  i    j    k |  N |  1
   0   -1    0    0    0    0    0    0    5    ## Arr == a
   0    0   -1    0    1    0    0    0    0    ## [1] == i
   0    0    0   -1    0    0    1    0    0    ## [2] == k

# ---------------------------------------------  2.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
3
# List of original iterators
i j k
# Statement body expression
a[j][k] -= c[i][j] * a[i][k];
</body>

# =============================================== Extensions
<scatnames>
b0 i b1 j b2 k b3
</scatnames>

<arrays>
# Number of arrays
6
# Mapping array-identifiers/array-names
1 i
2 N
3 j
4 c
5 a
6 k
</arrays>

<coordinates>
# File name
/tmp/tmp.bW0YPvqFx4.c
# Starting line and column
20 0
# Ending line and column
30 0
# Indentation
8
</coordinates>

</OpenScop>
```