POSTER: A Polyhedral+Dataflow Intermediate Language for Performance Exploration

Eddie C. Davis
Computer Science
Boise State University
Boise, USA
eddiedavis@boisestate.edu

Catherine RM. Olschanowsky
Computer Science
Boise State University
Boise, USA
catherineolschan@boisestate.edu

***Abstract*—This poster introduces a compiler intermediate language designed for dataflow optimizations within a polyhedral framework. This intermediate representation describes computations at a high level, defines a set of loop and data transformations that can be applied, and provides visual feedback reflecting the expected effect of transformations on the performance model.**

Computations are represented as macro-dataflow graphs, with support for both regular and irregular scientific applications, including stencils and sparse linear algebra kernels. This layer provides optimizations such as loop transformations or temporary storage reductions. The multi-level intermediate representation enables this broad range of optimizations by allowing each layer to be transformed independently, while respecting dependences. The approach is evaluated on a computational fluid dynamics solver, sparse matrix-vector multiplication kernels, and the matrix-tensor Khatri-Rao product. The experimental results either outperform or are competitive with existing implementations.

***Index Terms*—polyhedral, dataflow, compiler, irregular, sparse, inspector, executor**

I. Introduction

The performance of many numerical applications is limited by frequent interactions with the memory subsystem. Dataflow optimizations performed across entire sections of code can provide significant reductions in memory traffic, both decreasing execution time and reducing energy consumption. Krieger et al. [1] refer to these series of loops as loop chains, and demonstrate the potential impact of these transformations in both regular and irregular applications.

Compilers leverage multiple intermediate representations and successively lower the code from a high level representation such as an abstract syntax tree (AST) to mid-level IR (e.g., LLVM or GIMPLE), to the register level. Intermediate representations increasingly incorporate the polyhedral model and dataflow representations [2]–[4]. These new layers expose optimization opportunities such as loop transformations or temporary storage reductions that are not easily expressed in existing representations.

Dataflow graphs integrated with the polyhedral model have been applied to regular applications including PDE solvers [4], and dense linear algebra [5], but to our knowledge these have not yet been applied to irregular or sparse applications.

The polyhedral dataflow graphs (PDFG) extended in this work are an intermediate representation that expresses both the execution schedule and dataflow requirement of an application section. Graph variants are produced by applying successive transformations to the graph. Optimized C code is emitted by the generator. This tool can be incorporated into other toolchains by constructing a frontend to create graph specifications. The tool can be used via a command line compiler interface, a Python API, or an online version in the form of a Jupyter notebook. The contributions of this work are summarized below.

- An implementation of a compiler internal representation based on macro-dataflow graphs [4] that represents both execution schedule and dataflow requirements of a computation.
- A specification language (eDSL) to generate the IR.
- Graph operations that encompass polyhedral or AST code transformations.
- Code generation for optimized loop nests, memory allocations, and data mappings.

II. Background

A polyhedral dataflow graph (PDFG) represents both the execution schedule and the dataflow requirements. The space required by the schedule is a component of the graph. A PDFG consists of the following components $G = (S, D, T, E)$, where S is the set of statement nodes, D the data nodes, T the transformation nodes, and E the directed edges connecting the nodes. The statement nodes and data nodes are largely based on the graphs developed by Davis et al. [4]. Statement nodes, inverted triangles in the graph, represent ordered sets of statements, and encapsulate the iteration domain, statements, global schedule location within as a scattering function, and the data mappings that reference the data spaces read and written during statement execution.

Data nodes, depicted as rectangles, abstract storage spaces and consist of the type, range of values, the domain of indices that access it, and the size. The latter can be inferred from the domain of the statement node that writes the data. The space described in the graph corresponds to local space requirements. The memory allocation and associated mapping are created during code generation.

The representation includes support for sparse data structures important in many applications, including scientific com-

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00064

puting, graph analysis, and machine learning. The code for these applications often contains multiple levels of indirection, resulting in irregular memory access patterns, e.g., for index arrays such as `A[col[i]]`). These patterns can cause poor performance due to reduced data locality or limited prefetching. Loop boundaries can also be dependent on data that are unknown until runtime, making it difficult for compilers to determine which optimizations to perform.

Uninterpreted functions can represent data dependent loop bounds or other constraints that are unknown at compile time, such as index arrays in sparse structures. Support for these functions is provided by provided by the SPF [6] and Omega+ [7] code generator. They are realized as explicit functions that satisfy the associated constraints at runtime. The computation that requires the explicit function is known as an *executor*, while the *inspector* produces the data.

III. Polyhedral+Dataflow Language

The polyhedral+dataflow language (PDFL) expresses both regular (structured), or irregular (sparse) computations such as those in scientific or other numerical applications, including stencils in PDE solvers, or sparse linear and multilinear algebra kernels. A domain is bounded by a set of constraints. Each space can describe either an iteration or data space. Iteration spaces are associated with statements that define the computations performed at each point.

The underlying data structure for the IR is the polyhedral+dataflow graph. The domains of data and statement nodes are defined as integer sets. Input nodes are fixed-size and immutable, while output nodes are written, but not resized. The compiler has complete control over intermediate, temporary storage allocation. Initial sizes are inferred from the data access patterns, extracted from the right and left hand sides of assignment statements.

A statement node also has read and write data mappings defined to represent the data locations read from and written to during execution. Statements are defined as expressions of functions, constants, and literals. Each statement node represents a single loop nest. Transformation nodes contain relations that are applied to iteration or data spaces.

A. Graph Operations

Iteration and data domains can be transformed by graph operations such as node fusion, splitting, reordering, tiling, or unrolling. The liveness analysis algorithm will reduce storage on an entire graph. The scheduling algorithm produces a global schedule by traversing the graph.

1) Split, Fuse, and Tile: A statement node can be *split* partitions the domain on a given iterator by a split factor, *f*. The polyhedral+dataflow graphs also support the fusion of statement nodes (loop nests). Tiling can be applied to improve both temporal and data locality of a loop nest.

2) Rescheduling: Rescheduling refers to moving a statement node earlier or later in the execution. This can be useful when preparing to fuse nodes by moving them adjacent to one another in the schedule.

B. Memory Allocation

The memory allocation algorithm traverses the graph in reverse order, i.e., bottom to top, right to left. Temporary data spaces are stored in a reference table. An entry is marked as inactive if no longer being read from or written to at the current execution stage. If an existing, inactive space of adequate size is not found, a smaller inactive space will be resized. If no inactive spaces are available, a new active space will be allocated.

IV. Experimental Evaluation

This IR was evaluated against implementations of the Mini-FluxDiv CFD benchmark from [4], sparse matrix-vector multiplication (SpMV) inspector/executor kernels from [8], and the Khatri-Rao product (MTTKRP) kernel [9]. The generated code either outperformed or matched existing implementations.

V. Conclusion

The polyhedral+dataflow language and intermediate representation introduced here combines execution schedule transformations with dataflow optimizations. The language can be derived from a high-level programming language or other intermediate representation. Support for sparse data structures allows the optimizations to be applied to non-affine or regular codes, including PDE solvers, stencils, or sparse linear algebra kernels such as SpMV or MTTKRP.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1563818 and by the U.S. Department of Energy.

References

[1] M. M. Strout, F. Luporini, C. D. Krieger, C. Bertolli, G.-T. Bercea, C. Olschanowsky, J. Ramanujam, and P. H. Kelly, "Generalizing run-time tiling with the loop chain abstraction," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1136–1145.

[2] I. GitHub, "Multi-level intermediate representation compiler infrastructure," https://github.com/tensorflow/mlir, 2019.

[3] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 2019, pp. 193–205.

[4] E. C. Davis, M. M. Strout, and C. Olschanowsky, "Transforming loop chains via macro dataflow graphs," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 265–277.

[5] A. Sbîrlea, J. Shirako, L.-N. Pouchet, and V. Sarkar, "Polyhedral optimizations for a data-flow graph language," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2015, pp. 57–72.

[6] M. M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky, "An approach for code generation in the sparse polyhedral framework," *Parallel Computing*, vol. 53, pp. 32–57, 2016.

[7] C. Chen, "Polyhedra scanning revisited," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 499–508, 2012.

[8] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 521–532.

[9] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 77, 2017.