Name: Payal Guha Nandy

PhD Candidate: since Fall 2016

Institution: University of Utah

Advisor: Prof. Mary Hall

Extensions to Sparse Polyhedral Framework for Automatic Generation of Composable Inspectors/Executors

Irregular applications such as molecular dynamics simulations, finite element analysis, and big graph analysis rely on efficient computation over sparse matrices or graphs. Such sparse computations reduce data storage and computation requirements by using indirect accesses through index arrays that store only nonzero data elements. Historically, compilers have been severely limited in their ability to optimize such sparse computations due to the indirection that arises in indexing and looping over just the nonzero elements. In recognition of this property, inspector-executor strategies were developed where an inspector code traversed the index arrays to determine data access patterns at runtime and an executor code would use this information to derive transformed schedules and reordered data structures to execute the computation in a more efficient manner.

Most research that uses inspectors relies on instantiating application specific inspector templates, invoking inspector library code, or manually writing inspectors. My research is part of a project that aims to create a generalized framework for automatically generating Inspectors for a wide range of sparse matrix computations by extending the Sparse Polyhedral Framework (SPF) with composable loop and data transformations. SPF is an extension of the polyhedral framework for transformation and code generation. SPF extends the polyhedral framework to represent runtime information with un-interpreted functions and inspector computations that explicitly realize such functions at runtime. Previously SPF was used to derive inspectors for data and iteration space reordering. Data transformations will now be introduced into SPF, for example, conversions between different sparse matrix formats to leverage locality.

We have started with defining the abstractions required for creating such a framework. Non-affine iteration and data reordering transformations are defined that derive run-time relations and perform data transformations in the inspector that are used by the executor. Uninterpreted functions capture the mapping between the original iteration and data space and the transformed iteration and data space. An Inspector Dependence Graph is defined that represents the tasks the inspector must perform to generate at run time explicit versions of what are uninterpreted functions at compile time. Our compiler will generate the inspector code by walking the IDG and the original code is transformed into the executor code. The next steps will involve possible extensions to support inspector composition and incorporate other optimizations to reduce the number of passes required by the inspector to perform the transformations.

Previous work has demonstrated the automatic generation of inspector/executor code, which orchestrates code and data transformations to derive high performance representations for the Sparse Matrix Vector Multiply kernel in particular. The same transformations were integrated into sparse matrix and graph applications such as Sparse Matrix-Matrix Multiply and Stochastic Gradient Descent improving the application performance. A feasibility analysis is currently underway to determine whether efficient elimination trees can be generated as a part of Cholesky factorization of sparse matrices leading to the structural representation of the lower triangular matrix.