Compilers for Regular and Irregular Stencils: Some Shared Problems and Solutions

Michelle Mills Strout
Colorado State University
Fort Collins, CO 80526
mstrout@cs.colostate.edu

***Abstract*—Solving partial differential equations results in a continuum of regular and irregular stencil computation implementations. In this paper, we use heat diffusion on a bar to show how regular and irregular stencil computations are related, and then illustrate five complicating issues that occur in implementing the continuum of regular and irregular stencil computations in full applications. These complicating issues make it difficult for compilers to discover stencil computations and to represent and generate code for the combination of regular and irregular transformations that are relevant. We overview projects at Colorado State University and elsewhere that are developing solutions to the complicating issues surrounding stencil computations in partial differential equation (PDE) solver applications.**

***Keywords*-regular and irregular stencils, sparse matrix vector multiplication (SpMV), sparse tiling, sparse polyhedral framework, heat diffusion problem, PDE solvers**

I. Introduction

Stencil computations occur in many different application areas: solving partial differential equations with finite difference, finite element methods, or finite volume methods; graphics processing pipelines; and cellular automata are common examples. A key characteristic of such computations is that there is some underlying graph and the structure of the computation derives from this graph. This computation structure often involves sweeping over data associated with edges and nodes in the graph and computing new values to store in data associated with edges or nodes in the graph. We assume that each sweep over the values associated with the graph is fully parallel (i.e., Jacobi iterations) and the computation is almost always memory bandwidth bound due to the relative amount of data being brought from memory each sweep versus the amount of computation. Although stencil computations are often memory bound they exhibit data reuse; therefore, program optimizations that reorganize the computation so that data in some level of cache are effectively reused can reduce the demands on the memory bandwidth.

A common example stencil is the heat diffusion equation, where the loss of heat in an object over time is computed. Papers such as [1]–[3] investigate methods for improving the performance of an explicit time stepping approximation of the heat diffusion equation. Here we show a derivation of the stencil and then discuss complications that occur when such stencils occur in full applications.

Assume simulation of the temperature on a bar of some material over time where the initial conditions for the bar are that it is $0\,^{\circ}\mathrm{C}$ and one end is in ice water and the other end is in boiling water. The temperature over time is assumed constant on the ends (i.e., Dirichlet boundary conditions). The partial differential equation (PDE) known as the heat diffusion equation models the temperature at each point in the bar over time:

$$u_t - \alpha u_{xx} = 0,$$

where α represents a constant based on properties of the bar material, u_t represents the first derivative of the unknown temperature in time, and u_{xx} represents the second derivative of temperature along the bar. The goal is to solve for the temperature at each point in time and in space, $u(t, x)$.

Explicit time stepping methods for solving parabolic PDEs like heat diffusion that evolve over time discretize the space being simulated (creating a mesh, or graph), use some variant of the Taylor series for approximating the value of the partial derivatives at each vertex in the graph, and then use the approximations to determine a temperature value at some point in time and space as a function of the temperature values for the same point and its neighbors at a previous time step.

Let $u(t, x)$ denote the temperature at each point in time and space. In the discretization graph, there will be vertices where the temperature is approximated at Δx spacings in simulated space. The discretization graph vertices occur at $(k\Delta t, i\Delta x)$ for k and i being integers ranging from 0 to some number T for time and n for space.

For explicit time stepping, the finite difference method based on the Taylor series approximates u_t as follows:

$$u_t(t, x) \approx \frac{u(t+1, x) - u(t, x)}{\Delta t}$$

and u_{xx} as follows:

$$u_{xx}(t, x) \approx \frac{u(t, x-1) - 2u(t, x) + u(t, x+1)}{(\Delta x)^2},$$

where the error terms of $O(\Delta t)$ and $O((\Delta x)^2)$ for both approximations have been dropped. By substituting the approximations into the PDE,

$$\frac{u(t+1, x) - u(t, x)}{\Delta t} \approx \alpha\left(\frac{u(t, x-1) - 2u(t, x) + u(t, x+1)}{(\Delta x)^2}\right)$$

```
// time steps
for (k=0; k<(T-1); k++) {
  // boundary conditions
  u[k+1][0] = 0;     // freezing water end of bar
  u[k+1][n] = 100;   // boiling water end of bar

  // iterating over discretized points in space
  for (i=1; i<n; i++) {
    u[k+1][i] = lambda (u[k][i-1] + u[k][i+1])
                + (1-2*lambda)*u[k][i];
  }
}
```

Fig. 1: Regular stencil computation for heat diffusion on a bar.

```
// time steps
for (k=0; k<(T-1); k++) {
  // boundary conditions
  u[k+1][0] = 0;     // freezing water end of bar
  u[k+1][n] = 100;   // boiling water end of bar

  //—— sparse matrix vector multiply, SpMV
  // spatial steps, or sparse matrix rows
  for (i=1; i<n; i++) {
    u[k+1][i] = 0;
    // over neighbors, or non-zeros per row
    for (p=rowptr[i]; p<rowptr[i+1]; p++) {
      u[k+1][i] += valA[p] * u[k-1][col[p]];
    }
  }

}
```

Fig. 2: Irregular stencil computation for heat diffusion.

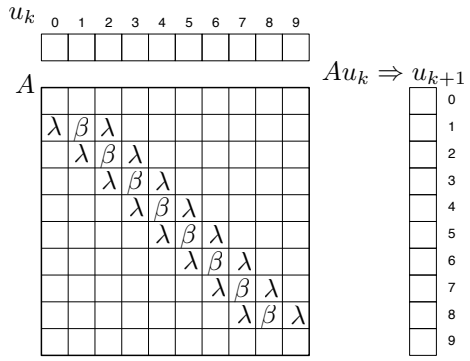

Fig. 3: The sparse matrix for an irregular implementation of heat diffusion on a bar. The computation in Figure 2 reads from the kth u vector shown above the sparse matrix based on the column of the relevant non-zeros, and the computation writes to the ith entry in the $k+1$th u vector based on the row of the sparse matrix. The coefficients for the computation are stored in a tridiagonal sparse matrix.

we obtain an equation that is only in terms of elements of the temperature unknown u. Let $\lambda = \frac{\alpha \Delta t}{(\Delta x)^2}$ and $\beta = 1 - 2\frac{\alpha \Delta t}{(\Delta x)^2}$ in the following

$$u(t+1,x) \quad \approx \quad \lambda u(t,x-1) + \lambda u(t,x+1) + \beta u(t,x).$$

We initialize all temperature values at time zero $u(0,x)$ to $0\,^{\circ}\mathrm{C}$. We also initialize boundary domain values for all time steps to 0 or 100 based on which end of the bar they represent (i.e., 0 degrees for freezing water and 100 for boiling water).

Figure 1 illustrates a *regular* implementation of the heat diffusion problem. The code in Figure 1 uses the storage mapping of $u(k*\Delta t, i*\Delta x)$ to `u[k,i]`, where the assumption is that `u[0,i]` entries hold initial guesses for temperature values and the boundary values where i is 0 or n hold the boundary constants.

The simulation of heat diffusion using finite difference approximations can also be formulated as a sparse matrix, or *irregular*, computation and when using Matlab, such an organization is typical [4]. Figure 2 contains an *irregular* version of the 1D heat diffusion problem. In the irregular version, the unknowns $u(t,x)$ are organized into $T+1$ vectors, $u_0, u_1, ..., u_T$. The coefficients for the stencil computation are now stored in a sparse matrix A, which is stored in the compressed sparse row (CSR) format. The u vector for each time step $k+1$ is computed by performing a sparse matrix vector (SpMV) computation with the sparse matrix A and the u vector from time step k. Figure 3 shows this graphically with the organization of coefficient values in the sparse matrix.

The two straight-forward implementations of heat diffusion for a 1D bar shown in Figures 1 and 2 provide a concrete example that regular and irregular stencil computations are strongly correlated. Both can implement computations over structured graphs. Regular stencil computation are common when the discretization is structured or mostly structured, because mapping the unknowns into arrays versus maintaining their structure by explicitly storing the non-zero structure of a sparse matrix reduces memory bandwidth demands and therefore improves performance. Irregular computations are more common over unstructured mesh discretizations, because it is difficult to map such meshes into arrays.

This paper presents examples of stencil computations in real applications and describes five complicating factors of such implementations from the perspective of a compiler writer (Section II). Domain-specific libraries are used to hide the complicated implementation issues, however Section III describes why such libraries – although critical for managing software complexity and enabling maintainability – make program analysis to automatically detect stencil computations more difficult. The section continues with a review of some of the declarative programming approaches that have been developed to enable complexity management and stencil detection to coexist; these include the loop chain abstraction [5] and the GridWeaver [6]. In Section IV we discuss the code representation and code generation problemand overview the Sparse Polyhedral Framework as an approach to represent program transformations for irregular stencil computations. Section V summarizes outstanding issues and promises future directions for effectively mapping regular and irregular stencil computations to current and future architectures.

II. Stencils in the Wild

In real applications, things are not as neat and simple as the regular and irregular stencil computations shown in Figures 1

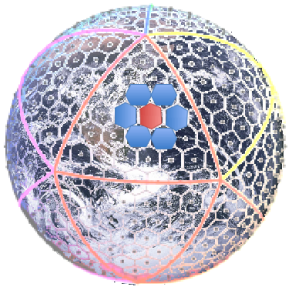

Fig. 4: Geodesic grid from SWM proxy application, where the surface of the Earth is discretized into hexagonal and occasional pentagonal cells.

```
!-------------------------------------------------
!  horizontal advection
!-------------------------------------------------
DO nsd = 1,nsdm
   CALL jacobian (
      area_inv(:,:,      nsd), &
      mss     (:,:,1,np0,nsd), &
      psi     (:,:,1,    nsd), &
      work1   (:,:,1,    nsd))
   CALL flux_divergence (
      l_weights(:,:,:,      nsd), &
      mss      (  :,:,1,np0,nsd), &
      chi      (  :,:,1,    nsd), &
      work2    (  :,:,1,    nsd))
ENDDO
```

Fig. 5: The horizontal advection operation in SWM.

and 2. Complicating issues include

1) storage allocation where memory locations are reused to reduce the number of unknown vectors stored,
2) periodic and semi-regular domain topologies,
3) finding stencil computations that have been hidden for modularity reasons,
4) computations that consist of multiple, single sweep stencils, and
5) cases where the stencil computation becomes irregular due to less structured discretization graphs.

This section presents code examples from proxy applications, which in turn were derived from full applications. We illustrate each of these complicating issues that make compiler analysis and optimization of stencil computations difficult.

A. Shallow Water Model (SWM) on Geodesic Grid

The Shallow Water Model (SWM) equations are solved over the geodesic grid in the SWM proxy application made available by David Randall's group at Colorado State University [7]. SWM is used as a proxy application for the much larger Global Cloud Resolution Model (GCRM) and it exemplifies how stencil operators are implemented over the geodesic grid (see Figure 4).

Figures 5 and 6 show the horizontal advection computation and one of its stencil operators, which computes the flux divergence. The horizontal advection computation in Figure 5, which calls a `jacobian` and `flux_divergence`, is contained within an outer loop stepping over simulated time. Within the time stepping loop there are other stencil operators. This code exhibits a number of complicating factors for stencil computations.

As an example of storage allocation (issue 1), note that in Figure 6 the flux values being computed during the single sweep over the mesh are not stored in an array indexed by time, but instead are stored in three copies of the `x` array: `x1`, `x2`, and `x3`. The numerical method being used to perform the explicit time stepping determines the number of copies that need to be stored.

The overriding design aspect that crosscuts all of the SWM code is the geodesic grid, which is an example of semi-regular domain topology (issue 2). The geodesic grid is modeled with

```
! w is short for l_weights
DO j = 2,jjm-1
 DO i = 2,iim-1
   x3(i,j) = 0.5_dbl_kind*                                          &
    ((x1(i,j)+x1(i+1,j   ))*(x2(i+1,j   )-x2(i,j))*w(4,i,j)+ &
     (x1(i,j)+x1(i+1,j+1))*(x2(i+1,j+1)-x2(i,j))*w(5,i,j)+ &
     (x1(i,j)+x1(i   ,j+1))*(x2(i   ,j+1)-x2(i,j))*w(6,i,j)+ &
     (x1(i,j)+x1(i-1,j   ))*(x2(i-1,j   )-x2(i,j))*w(1,i,j)+ &
     (x1(i,j)+x1(i-1,j-1))*(x2(i-1,j-1)-x2(i,j))*w(2,i,j)+ &
     (x1(i,j)+x1(i   ,j-1))*(x2(i   ,j-1)-x2(i,j))*w(3,i,j))
 ENDDO
ENDDO

!  NORTH POLE
i = 1; j = jjm-1
   x3(i,j) = 0.5_dbl_kind*                                          &
    ((x1(i,j)+x1(i+1,j   ))*(x2(i+1,j   )-x2(i,j))*w(4,i,j)+ &
     (x1(i,j)+x1(i+1,j+1))*(x2(i+1,j+1)-x2(i,j))*w(5,i,j)+ &
     (x1(i,j)+x1(i   ,j+1))*(x2(i   ,j+1)-x2(i,j))*w(6,i,j)+ &
     (x1(i,j)+x1(iim,   1))*(x2(iim,   1)-x2(i,j))*w(2,i,j)+ &
     (x1(i,j)+x1(i   ,j-1))*(x2(i   ,j-1)-x2(i,j))*w(3,i,j))

!  SOUTH POLE
i = iim-1; j = 1
   x3(i,j) = 0.5_dbl_kind*                                          &
    ((x1(i,j)+x1(i+1,j   ))*(x2(i+1,j   )-x2(i,j))*w(4,i,j)+ &
     (x1(i,j)+x1(i+1,j+1))*(x2(i+1,j+1)-x2(i,j))*w(5,i,j)+ &
     (x1(i,j)+x1(i   ,j+1))*(x2(i   ,j+1)-x2(i,j))*w(6,i,j)+ &
     (x1(i,j)+x1(iim,   1))*(x2(iim,   1)-x2(i,j))*w(2,i,j)+ &
     (x1(i,j)+x1(i   ,j-1))*(x2(i   ,j-1)-x2(i,j))*w(3,i,j))
```

Fig. 6: The code from the `flux_divergence` operator in the SWM proxy application. Each operator executes over one subdomain in the grid.

a graph where each vertex models a hexagonal or pentagonal cell. The vertices are grouped into 10 subdomains (i.e., each subdomain is two of the triangular panels from Figure 4). The focus on the geodesic grid discretization results in the horizontal advection computation in Figure 5 containing loops over the various subdomains in the geodesic grid, and the stencil pattern results in six neighbors for each vertex due to the dominant cell shape being hexagonal. Additionally, the stencil computations in the flux divergence (see Figure 6) and other operators such as the `jacobian` contain special code for the north and south poles, which are embedded in only one subdomain, but computation for each of them is done in all subdomains. The complexity the geodesic grid introduces to the stencil operators is minor compared to its effect upon the communication code. SWM uses 1,891 lines

```
do iblock=1,nblocks_tropic
    gid = blocks_tropic(iblock)

    !------ calculate (PC)r store in Z
    Z(:,:,iblock) = R(:,:,iblock)*A0R(:,:,iblock)

    !------ Compute intermediate result for dot product
    WORKN(:,:,1,iblock) = R(:,:,iblock)*Z(:,:,iblock)

    !------ update conjugate direction vector S
    S(:,:,iblock) =  Z(:,:,iblock)

    !------ compute Q = A * S
    call btrop_operator(Q,S,gid,iblock)

    !------ compute intermediate result for dot product
    WORKN(:,:,2,iblock) = S(:,:,iblock)*Q(:,:,iblock)
end do
```

Fig. 7: Code from the Chronopoulos-Gear conjugate-gradient implementation in CGPOP. The `btrop_operator` routine performs a 9 point stencil in a 2D grid.

of communication code to exchange data between subdomains being computed with distributed MPI processes [6].

Another striking pattern in this example code is that there is only one sweep over the spatial domain per stencil operation (issue 4). Although the stencil pattern of nearest neighbor accesses is similar in each stencil, each stencil computation is in fact different. For modularity (issue 3), each stencil implementation is kept within its own subroutine, which forces inter-procedural programanalysis to optimize between stencil computations.

There is data sharing between the stencil sweeps and therefore optimizations that group iterations of one stencil sweep with iterations of another stencil can reduce temporary memory storage and/or improve data locality. Unfortunately time skewing/tiling techniques [8]–[10] are not directly applicable because of single sweeps (issue 4). The single sweeps of different stencils occur because the PDEs involved include more than one variable and because the time stepping approach is implicit or semi-implicit. With implicit time stepping methods, a system of linear equations that spans the whole grid needs to be solved in each simulated time step and this typically leads to some kind of reduction dependence on the full mesh. Experiments that modify the implicit time stepping scheme to an explicit one that would enable time skewing [11], but the tradeoff between the larger time step that implicit time stepping enables and the improvements in parallelism and data locality scheduling that explicit time steps provide still leans in the direction of implicit time stepping. These observations indicate that opportunities for optimizations focused on loops with multiple iterations over one stencil sweep within the same loop need to be adapted to this more complex setting. Previous research has also noted this issue [12]–[14].

B. Conjugate Gradient in Parallel Ocean Program (CGPOP)

CGPOP is a proxy application extracted from the Parallel Ocean Program [15]. Figure 7 shows a loop within the Conjugate Gradient method implemented in CGPOP. The loop iterates over blocks in a dipole discretization of the surface of the Earth (i.e., rectangular grid with periodic boundary conditions that wrap around the Earth). This one section of code illustrates modularity (issue 3) and single sweeps (issue 4). CGPOP also contains code that handles periodic boundaries (issue 2), storage allocation (issue 1) and a variant of the CG computation in CGPOP maps all of the variables to irregular/sparse data structures (issue 5).

As with SWM, CGPOP modularizes stencil operators into separate functions (issue 3). For example, in Figure 7 the `btrop_operator` routine implements a nine-point stencil in the two-dimensional sub grid `iblock`. Inter-procedural analysis would be needed to discover and/or inline stencil computations and the intervening data parallel operations such as the array assignments that make up most of the loops (issue 4), makes compiler optimization of the stencil computations difficult. Using time skewing or time tiling techniques across iterations of the Conjugate Gradient computation is in fact impossible due to the reduction that is needed at the end of each CG iteration (issue 4).

As an example of the irregular stencil (issue 5), two facets of CGPOP result in indirect referencing, or irregular data structures. First, the blocks in the dipole discretization are ordered via a space filling curve to improve the allocation of blocks to MPI processes. This maps the two-dimensional arrangement of sub grids down to a one-dimensional array and then grid neighbor ids are stored explicitly as in a sparse matrix data structure. In addition to this space filling curve, any sub grid that models land only is left out of the array of sub grids. Second, there is a variant of the CG solver that stores each sub grid in a one-dimensional sparse array instead of a two-dimensional dense array so as to avoid storing and computing on any land points. On some machines this results in improved performance [16]. Therefore, even applications such as POP where the discretization graph is mostly regular exhibit this continuum of regular and irregular stencils in the implementation.

C. Computations On Unstructured Meshes (OP2)

SWM and CGPOP have mostly regular stencils with some irregular stencil code creeping in. They are both based on finite differencing schemes. Approaches for solving PDEs that typically start with unstructured grids include finite element analysis, which is often irregular. OP2 is a library that has been developed to directly deal with irregular stencil computations (issue 5). The codes in OP2 also exhibit single sweeps (issue 4) and storage allocation complexity (issue 1). OP2 codes do not have additional complicating code to handle periodic and irregular domain topologies (issue 2), because the explicit storage of irregular graph structure handles irregular discretization domains naturally.

The OP2 library [17] provides an interface for computations over unstructured meshes. A Rolls Royce engine simulation uses OP2. Figure 8 shows some example code. Once again we see that different kernels, or stencil computations, are being performed within one time step (issue 4). Additionally, each stencil computation is within a function (issue 3).

```
// loop over edges
op_par_loop (edges, kernel1,
  op_arg_dat (x, -1, OP_ID, OP_READ),
  op_arg_dat (vert, 0, edges2vertices, OP_INC),
  op_arg_dat (vert, 1, edges2vertices, OP_INC))

// loop over cells
op_par_loop (cells, kernel2,
  op_arg_dat (vert, 0, cells2vertices, OP_INC),
  op_arg_dat (vert, 1, cells2vertices, OP_INC),
  op_arg_dat (vert, 2, cells2vertices, OP_INC),
  op_arg_dat (res, -1, OP_ID, OP_READ))

// loop over edges
op_par_loop (edges, kernel3,
  op_arg_dat (vert, 0, edges2vertices, OP_INC),
  op_arg_dat (vert, 1, edges2vertices, OP_INC))
```

Fig. 8: Example OP2 code. Loops over sets of mesh elements such as cells and edges are hidden behind `op_par_loop` function calls. However, the parameters provided to the `op_par_loop` function calls provide data access information.

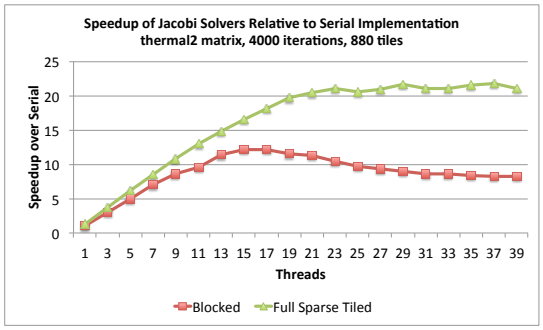

Fig. 9: Speedup of a row-blocked Jacobi solver implementation and a full sparse tiled implementation compared with a serial version of the same algorithm. The FST version is consistently faster than the blocked version and scales better on the Intel MTL 40 core system.

III. Analysis Problem and Solutions for Regular and Irregular Stencils

The complicating issues in real stencil codes all contribute to making program analysis that recognizes stencils quite difficult. Storage allocation complexity (issue 1) results in different time steps of a variable being stored in disparate arrays. Periodic and semi-regular domains (issue 2) along with the typical MPI distributed memory parallelism in these codes result in complex communications between each stencil application. Approaches for maintaining modularity and hiding parallelization implementation details that include wrapping stencil operators and communication in functions (issue 3) require the use of inter-procedural analysis in situations where it may not be possible to provide the whole program due to library usage. Due to the multitude of code idioms for storing sparse matrices, irregular stencil computations (issue 5) are even more difficult to discover.

Many projects avoid the analysis problem by providing ways to declaratively specify the stencil computations [14], [18]–[20]. These all work toward developing abstractions for specifying these stencil computations; however there is no single specification mechanism that handles all of the complicating issues yet. We have developed declarative abstractions such as the loop chain, described in Section III-A, for dealing with the modularity (issue 3) and the GridWeaver active library, described in Section III-B, for dealing with semi-regular grid topologies (issue 2).

A. Loop Chain for Annotating Existing Codes

The loop chain abstraction represents sequence of loops that share data and specifies how each iteration in the loops read and write data. A loop chain can be expressed to the compiler and/or runtime system using pragmas, a library interface, or new programming language constructs [5]. The loop chain abstraction provides a way to avoid complex, inter-procedural program analysis while requiring minimal extra information from the programmer. The loop chain abstraction requires that each loop in the chain is parallel or a reduction, has a well-defined domain, and has *well-defined access functions that indicate how each iteration accesses data spaces*. With these requirements, the loop chain abstraction provides enough information to a compiler and/or a runtime system to determine the partially ordered set of iterations that makes scheduling and determining data distributions across loops possible for a compiler and/or run-time system. The flexibility to schedule across loops enables better management of the data locality and parallelism tradeoff.

We have developed a data structure for representing loop chains *at runtime* for sparse, or irregular, applications [21]. The runtime data structure is an object oriented implementation including a single object for each loop representing the bounds of the loop, an object to represent each data array an its domain, and edges connecting iterations and data that represent data access functions. A programmer could use the data structure we have developed although it is somewhat cumbersome. We are investigating how to provide the information in pragmas or to derive the loop chain information from domain-specific library function calls like the `op_par_loop` calls in Figure 8 that provide data access information. For example, `OP_READ` indicates the `x` array is being read in the loop. The `OP_ID` tag indicates that the read is done with an identity function, each edge reads a corresponding `x` entry. The `OP_INC` tag indicates that an associative and commutative operation is being performed on entries in the `vert` array. Entries in the `vert` array are being accessed through integer index arrays, `edges2vertices`.

Our research group and others have found that scheduling across loops improves data locality and overall performance. For irregular codes, there have been various inspector/executor strategies [22], [23] that reschedule across loops to improve data locality while still providing parallelism. Examples include fully sparse tiling [5], [24]–[26] from our group, and parallel shrinking sparse tiles with a cleanup tile [27] and overlapped sparse tiles [28], [29] from other groups. Figure 9

(from our recent loop chain paper [5]) shows that scheduling across loops in the case of a sparse Jacobi computation can improve the scalability of a computation over the blocking done within a loop by OpenMP significantly. Scheduling strategies like this have been termed communication avoiding by Demmel et al. [28], [29]. The loop chain abstraction provides an intermediate representation for applying these scheduling strategies in a more automated fashion while side stepping some of the difficult program analysis issues.

B. GridWeaver Active Library for Sem-Regular Grids

The GridWeaver tool [6] is part of the SAIMI project (Separating Algorithms from Implementation through program Model Injection) [30]. The problem that GridWeaver deals with are semi-regular grids (issue 2) that occur in many atmospheric science computations such as CGPOP and SWM. In a semi-regular grid (also called irregularly coupled regular mesh [31] and irregular block structured applications [32]) the majority of the domain is broken into regular subdomains and then these subdomains are connected to other subdomains in irregular patterns (see Figure 4 for an example of a semi-regular grid). This is common in climate codes, because discretization occurs over the surface of a sphere.

CGPOP and SWM are example codes that operate on such semi-regular grids. For SWM, Figure 6 illustrates that characteristics of the grid, in this case the north and south poles, show up in every stencil operator in the code. The communication code for SWM is thousands of lines of code, and it is completely coupled with the semi-regular grid connectivity.

In the GridWeaver tool, a Fortran 90+ library interface is provided to enable the orthogonal specification of semi-regular grids, decomposition on those grids for SPMD-MPI parallelism, and stencil specifications. This provides a way to separate these concerns. To recover the performance, a source-to source compiler inlines the user provided stencil functions and GridWeaver iteration functions. To generalize this approach, abstractions would be needed to orthogonally specify storage mapping (issue 1) and to determine when the stencil computation should be implemented in an irregular/sparse fashion (issue 5). By providing orthogonal specifications of stencil computations, domain-specific program analysis can discover where single sweep stencils are applied (issue 4) and then adapt existing program optimizations to group computation across such stencil applications.

IV. Code Representation and Generation for Irregular Stencils

A significant amount of research has been done to optimize the regular stencil implementation of computations like the heat diffusion example in Section I. Storage allocation optimizations [33], [34], tiling through time [8]–[10], overlapped tiling [35], and diamond tiling [36] are just some examples. These optimizations focus on improving data locality while exposing sufficient parallelism. Reducing the amount of storage needed and/or reorganizing the computation to reduce memory accesses can improve the data locality in the computation. Exposing parallelism amongst aggregations of computations (or tiles) provides coarse-grain parallelism.

As shown in Section I, the heat diffusion stencil computation has a corresponding irregular stencil implementation that involves sparse matrix vector multiplications. Since all stencil computations are essentially sparse matrix vector multiply (SpMV) in an efficient disguise, it follows that loop transformations such as blocking and time tiling that improve the performance in regular stencil computations can provide benefits in irregular stencil implementations. Although there are a number of loop transformation frameworks and code generators for codes with compile-time analyzable dependence patterns like regular stencils, the code generation for irregular stencils has been more focused on selecting sparse matrix formats [37]–[39] and generating transformed variants of critical kernels like SpMV [40], [41]. This section reviews sparse tiling techniques, illustrates how full sparse tiling modifies the schedule of the irregular stencil implementation of heat diffusion, and overviews the sparse polyhedral framework as a candidate solution for representing and transforming stencil implementations that lie within the regular and irregular continuum.

A. Sparse Tiling Program Optimization

One example where optimizations for regular stencil computations have crossed over to unstructured meshes and sparse matrix computations are techniques that schedule across iterations of a sparse matrix computation or between loops where at least one loop has an indirect reference. These sparse tiling techniques partition the discreitzation graph at inspector time and then use various algorithms to aggregate groups of iterations into tiles that can be executed atomically (i.e., all the input data needed for a tile is available before the tile starts execution and there are no cycles between tiles).

Sparse tiling approaches include the so called cache blocking methodology developed by Douglas et al. [27], full sparse tiling [25], [42], [43], Adam's approach to parallelizing Gauss-Seidel [44] and communication avoiding strategies, which include variants of overlapped sparse tiles [28], [45]. These sparse tiling techniques are often done in concert with other run-time reordering techniques such as reordering the rows and columns in a sparse matrix.

B. Example Application of Sparse Tiling

One key question is how can we automate the representation and code generation of irregular transformations like sparse tiling. Figure 2 shows the original code, and Figure 10 shows the transformed code once full sparse tiling has been applied (note there is some code not shown referred to as inspector code that computes the tiling function `tileptr` and `row` arrays at runtime). Recall that Figure 9 showed how a full sparse tiling approach can improve the parallel scalability of such computations.

C. Sparse Polyhedral Framework

The Sparse Polyhedral Framework is being developed to answer the questions of how to represent irregular com-

```
for (t=1; t<=Nt; t++) {
  for (k=0; k<(T-1); k++) {
    for (v=tileptr[t][k]; v<tileptr[t][k+1]; v++) {
      i = row[v];
      for (p=rowptr[r]; p<rowptr[r+1]; p++) {
        u[k+1][i] += valA[p] * u[k-1][col[p]];
    }
  }
}
```

Fig. 10: Full sparse tiled heat diffusion kernel.

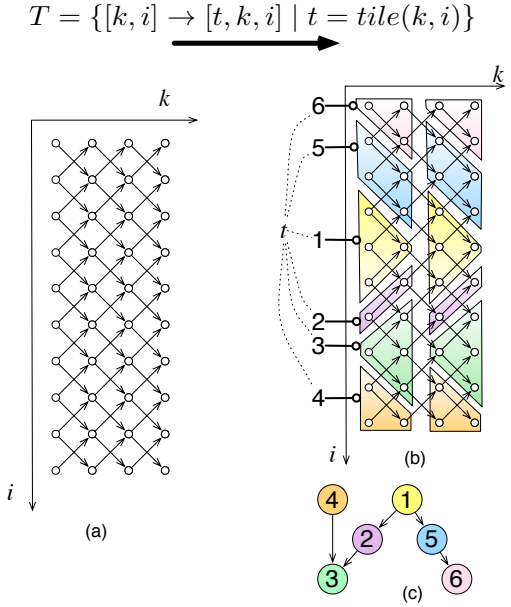

Fig. 11: (a) An iteration space representation of the computation in Figure 3. Each point represents an iteration (k, i) of the outer two loops. The arrows represent data flow dependencies between the iteration points (i.e., the source of the arrow must be computed before the target of the arrow). (b) A full sparse tiling of the iteration space. The transformation relation T represents the transformation. (c) To expose parallelism between tiles it is possible to explicitly create a task graph at runtime and either schedule wavefronts of independent tiles.

putations and transformations for such computations in a compiler. Figure 11(a) shows how the irregular heat diffusion implementation from Figure 2 can be represented as a set of iteration points (k, i). The transformation relation $T = \{[k,i] \rightarrow [t,k,i] \mid t = tile(k,i)\}$ describes how a user-defined inspector that creates the uninterpreted function $tile()$ at runtime, can transform the original computation into a full sparse tiled computation illustrated in Figure 11(b).

SPF was originally presented (although not so named) in the context of molecular dynamics and sparse iterative (i.e., irregular stencil) benchmarks [25]. More recent work has been developing techniques for manipulating the sets and relations in SPF, which include uninterpreted function symbols [46]. Additional work by Venkat et al. [47] modifies a common code generation interface used in polyhedral loop transformation frameworks (specifically CHILL [48]) for the incorporation of irregular transformations (i.e., commonly called inspector/executor strategies).

Remaining questions we are investigating within the sparse polyhedral framework research include (1) Code generation: What part of the inspector needs to be provided by the user and which can be generated by a compiler? (2) Transformation correctness: how do we automate the correctness check for user-defined inspectors? and (3) Guidance: How do we automate the selection of inspector/executor transformations and their parameterization?

V. Summary

The continuum of regular and irregular stencil computations in real codes present complex obstacles in terms of applying program optimizations to improve data locality while still exposing parallelism. Declarative approaches for specifying stencil computations such as the loop chain abstraction and GridWeaver help deal with the analysis problem. The analogy of regular stencil computations to sparse matrix and irregular problems in general can be leveraged to enable the specification of transformations for irregular stencils within the context of extended loop transformation frameworks like the Sparse Polyhedral Framework.

VI. Acknowledgments

This project is supported by the Department of Energy CACHE Institute grant DE-SC04030, DOE grant DE-SC0003956, and NSF grant CCF 0746693. I would like to thank all of my collaborators for contributing to the ideas and some of the figures in this paper especially Christopher Krieger for the loop chain abstraction work and Andrew Stone for the GridWeaver work. I also would like to thank Larry Carter and Jeanne Ferrante for providing feedback on this paper.

References

[1] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explict optimizations for stencil computations," in *Memory Systems Performance and Correctness*, 2006.

[2] K. Datta, M. Murphy, V. Volkov, S. W. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick, "Stencil computation optimization and autotuning on state-of-the-art multicore architectures," in *In Proceedings of the ACM/IEEE Conference on Supercomputing*, 2008.

[3] J. Holewinski, L. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 311–320.

[4] J. N. Kutz, *Data-driven modeling and scientific computing: Methods for Integrating Dynamics of Complex Systems and Big Data*. Oxford University Press, 2013.

[5] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. H. Kelly, G. Mudalige, B. V. Straalen, and S. Williams, "Loop chaining: A programming abstraction for balancing locality and parallelism," in *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, May 2013.

[6] A. Stone and M. M. Strout, "Programming abstractions to separate concerns in semi-regular grids," in *In Proceedings of the 27th International Conference on Supercomputing (ICS)*, June 2013.

[7] "Spherical geodesic grids: A new approach to modeling the climate," http://kiwi.atmos.colostate.edu/BUGS/geodesic/, November 2013.

[8] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Programming Language Design and Implementation*. ACM, 1991.

[9] F. Bassetti, K. Davis, and D. Quinlan, "Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures," *Lecture Notes in Computer Science*, vol. 1505, 1998.

[10] D. Wonnacott, "Achieving scalable locality with time skewing," *International Journal of Parallel Programming*, vol. 30, no. 3, pp. 181–221, 2002.

[11] C. S. Konor and A. Arakawa, "Multipoint explicit differencing (med) for time integrations of the wave equation," *Monthly Weather Review*, vol. 135, no. 11, pp. 3862–3875, 2013/11/26 2007.

[12] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," *ACM SIGPLAN Notices (PLDI)*, vol. 34, no. 5, pp. 215–228, May 1999.

[13] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimizations and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, vol. 51, no. 1, pp. 129–159, 2009.

[14] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 519–530.

[15] A. Stone, J. Dennis, and M. M. Strout, "The CGPOP miniapp, version 1.0," Colorado State University, Tech. Rep. Technical Report CS-11-103, July 1 2011.

[16] J. M. Dennis and E. R. Jessup, "Applying automated memory analysis to improve iterative algorithms," *SIAM Journal of Scientific Computing*, vol. 29, pp. 2210–2223, 2007.

[17] G. Mudalige, M. Giles, J. Thiyagalingam, I. Reguly, C. Bertolli, P. Kelly, and A. Trefethen, "Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems," *Parallel Computing*, vol. 39, no. 11, pp. 669 – 692, 2013.

[18] T. Yuki, V. Basupalli, G. Gupta, G. Iooss, D. Kim, T. Pathan, P. Srinivasa, Y. Zou, and S. Rajopadhye, "Alphaz: A system for analysis, transformation, and code generation i n the polyhedral equational model," Technical Report CS-12-101, Colorado State University, Tech. Rep., 2012.

[19] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 117–128.

[20] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, 2013, pp. 13–24.

[21] C. Krieger, "Generalized full sparse tiling of loop chains," Ph.D. dissertation, Colorado State University, 2013.

[22] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley, "Principles of runtime support for parallel processors," in *Proceedings of the 2nd International Conference on Supercomputing*, 1988, pp. 140–152.

[23] J. Saltz, C. Chang, G. Edjlali, Y.-S. Hwang, B. Moon, R. Ponnusamy, S. Sharma, A. Sussman, M. Uysal, G. Agrawal, R. Das, and P. Havlak, "Programming irregular applications: Runtime support, compilation and tools," *Advances in Computers*, vol. 45, pp. 105–153, 1997.

[24] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck, "Sparse tiling for stationary iterative methods," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 95–114, February 2004.

[25] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reorderings," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[26] C. D. Krieger and M. M. Strout, "Executing optimized irregular applications using task graphs within existing parallel models," in *Proceedings of the Second Workshop on Irregular Applications: Architectures and Algorithms (IA^3) held in conjunction with SC12*, November 11, 2012.

[27] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß., "Cache Optimization for Structured and Unstructured Grid Multigrid," *Electronic Transaction on Numerical Analysis*, pp. 21–40, February 2000.

[28] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2008.

[29] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Supercomputing*. New York, NY, USA: ACM, 2009.

[30] M. M. Strout, "SAIMI: Separating algorithm and implementation via programming model injection," http://www.cs.colostate.edu/hpc/SAIMI/, 2013.

[31] A. Sussman, G. Agrawal, and J. Saltz, "A manual for the multiblock parti runtime primitives, revision 5," University of Maryland, Tech. Rep., January 1994.

[32] S. J. Fink, S. B. Baden, and S. R. Kohn, "Efficient run-time support for irregular block-structured applications," *Journal of Parallel and Distributed Computing*, vol. 50, no. 1-2, pp. 61–82, 1998.

[33] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe, "A unified framework for schedule and storage optimization," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2001, pp. 232–242.

[34] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1242–1257, 2005.

[35] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007, pp. 235–244.

[36] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2012.

[37] A. J. C. Bik and H. A. G. Wijshoff, "Automatic data structure selection and transformation for sparse matrix computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 2, pp. 109–126, 1996.

[38] W. Pugh and T. Shpeisman, "Sipr: A new framework for generating efficient code for sparse matrix computations," in *Proceedings of the Eleventh International Workshop on Languages and Compilers for Parallel Computing*, Chapel Hill, North Carolina, August 1998.

[39] N. Ahmed, N. Mateev, and K. Pingali, "A framework for sparse matrix code synthesis from high-level specifications," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 2000.

[40] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in sparsity," in *Proceedings of the International Conference on Computational Science (ICCS)*, ser. Lecture Notes in Computer Science, V.N.Alexandrov, J. Dongarra, and C.J.K.Tan, Eds., vol. 2073. Berlin / Heidelberg: Springer, May 28-30, 2001, pp. 127–136.

[41] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 1–12.

[42] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck, "Sparse tiling for stationary iterative methods," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 95–113, 2004.

[43] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck, "Combining performance aspects of irregular Gauss-Seidel via sparse tiling," in *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, July 2002.

[44] M. F. Adams, "A distributed memory unstructured Gauss-Seidel algorithm for multigrid smoothers," in *SC2001: High Performance Networking and Computing. Denver, CO*, ACM, Ed., 2001.

[45] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. ACM, 2009, pp. 36:1–36:12.

[46] M. M. Strout, G. George, and C. Olschanowsky, "Set and relation manipulation for the sparse polyhedral framework," in *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, September 2012.

[47] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," in *To be published in: Proceedings of International Symposium on Code Generation and Optimization CGO)*, 2014.

[48] M. Hall, J. Chame, J. Shin, C. Chen, G. Rudy, and M. M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2009.