2015 44th International Conference on Parallel Processing (ICPP)
Available at: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7349641

GLAF: A Visual Programming and Auto-Tuning Framework for Parallel Computing

Konstantinos Krommydas*
Dept. of Computer Science
Virginia Tech
kokrommy@vt.edu

Ruchira Sasanka
Intel Corporation
ruchira.sasanka@intel.com

Wu-chun Feng
Dept. of Computer Science
Virginia Tech
wfeng@vt.edu

***Abstract*—The past decade's computing revolution has delivered parallel hardware to the masses. However, the ability to exploit its capabilities and ignite scientific breakthrough at a proportionate level remains a challenge due to the lack of parallel programming expertise. Although different solutions have been proposed to facilitate harvesting the seeds of parallel computing, most target seasoned programmers and ignore the special nature of a target audience like domain experts.**

This paper addresses the challenge of realizing a programming abstraction and implementing an integrated development framework for this audience. We present GLAF — a grid-based language and auto-parallelizing, auto-tuning framework. Its key elements are its intuitive visual programming interface, which attempts to render expressing and validating an algorithm easier for domain experts, and its ability to automatically generate efficient serial and parallel Fortran and C code, including potentially beneficial code modifications (e.g., with respect to data layout). We find that the above features assist novice programmers to avoid common programming pitfalls and provide fast implementations.

I. Introduction

The ongoing parallel revolution has democratized parallel computing by making unprecedented amounts of computational power accessible to an ever increasing part of the scientific community. In contrast to what used to be the norm, more scientists, engineers, researchers (herein collectively referred to as *domain experts*) have access to at least commodity multi-core CPUs or single-node accelerator-based heterogeneous systems.

While powerful hardware is readily available, the ability to exploit it at its fullest and ignite scientific breakthrough at a proportionate level remains on the ground. One prime reason is that programming itself remains a prerogative of the few. Many domain experts possess rudimentary knowledge of at least one programming or scripting language that allows them to verify functional correctness of their algorithms, but the vast majority practically ignores *parallel* programming, an issue exacerbated by the number of parallel programming abstractions and languages. As such, domain experts typically have to resort to programming experts in order to have their algorithms coded, or optimized for *performance*. This process introduces communication overhead, errors and barriers, including the need for the programmer to obtain domain-specific knowledge and vice versa.

The question we attempt to address in this work is: "Can we realize a *programming abstraction* and implement a *development framework* for domain experts that adresses the aforementioned issues, i.e., an approach that provides a balance between performance/programmability, and renders this audience active participants of the parallelism era?"

We claim that such an abstraction should ideally be: (a) automatically parallelizable, optimizable and tunable to desired target hardware, yet platform agnostic, (b) intuitive, familiar, with minimalistic syntax, yet general, scalable and powerful enough to express real-world problems, (c) data-visual and interactive, in that code, data structures and data itself are visible simultaneously, thus facilitating algorithmic expression, understanding, and debugging, (d) able to (implicitly) integrate with existing legacy code (code used today in many sciences dates back to the early 70s).

In this work, we propose **GLAF**, a visual code-generation and auto-tuning framework for shared-memory parallel computing systems, which aspires to steer programming by domain experts with *minimal* or *basic* programming knowledge towards the general directions discussed above.

The rest of the paper is organized as follows. Section II discusses the constituent parts of the framework: the graphical user interface, GLAF as a programming language and its internal representation. Section III provides details about the proposed approach (automatic code generation, auto-parallelization, auto-tuning, and visualization). Sections IV and V present example use-cases and performance results, and Sections VI and VII discuss related work and conclude the paper, respectively.

II. GLAF Framework

A. Graphical User Interface (GUI)

Programming using GLAF differs from programming using a typical programming language (e.g., C or Java). In the latter users write code in a free textual format using the keyboard and express the algorithm using the appropriate language constructs. In GLAF, typing is kept at a minimum

*This work was conducted during an internship at Intel.

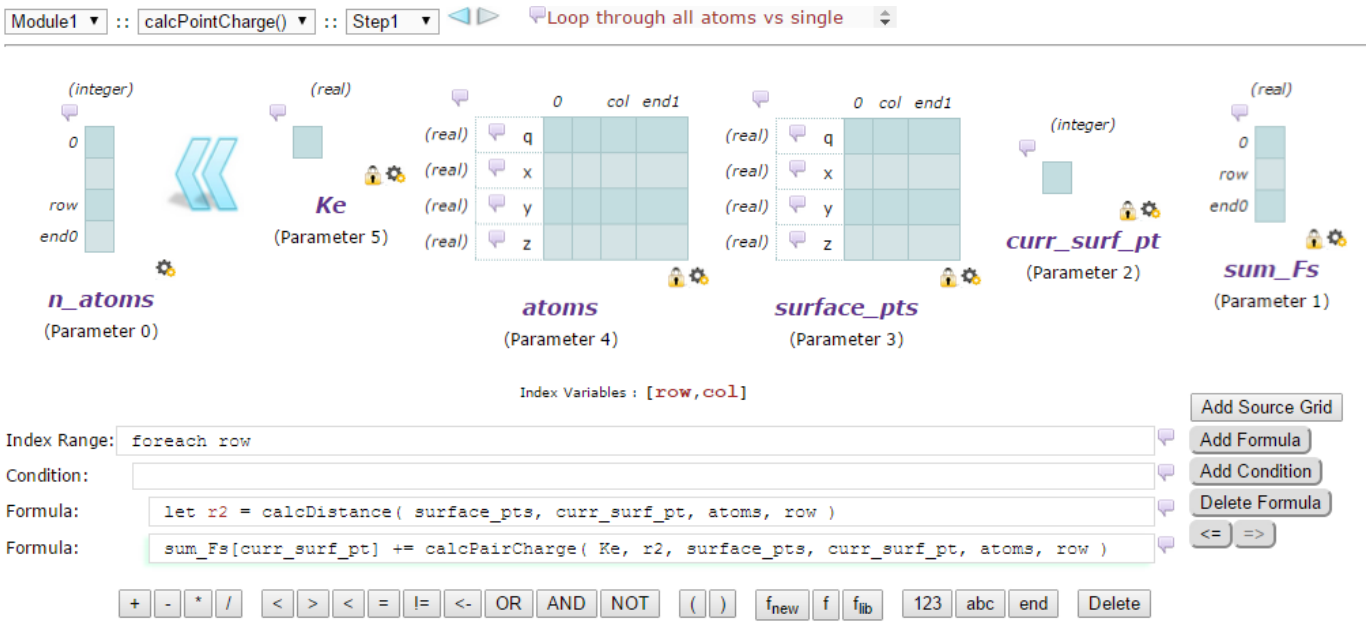


Figure 1: GLAF user interface: a GLAF step (code boxes are *automatically* filled through a point-and-click interface).

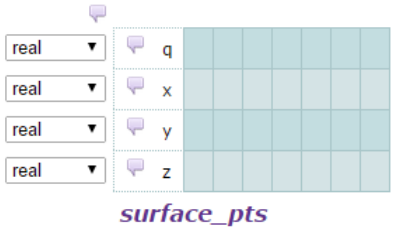
(e.g., naming grid variables) and programming is based on an intuitive point-and-click *visual interface*. GLAF's GUI is implemented in the form of a web page (Figure 1) that is written in a combination of HTML5 and JavaScript. This code drives the web interface (buttons, forms, images, menus, etc.) that facilitates GLAF code development and is responsible for populating appropriate environment variables (or *internal objects*, modeled after JavaScript functions). Modeling of such objects (Section II-C) lies at the basis of code generation and parallelism analysis algorithms.

B. GLAF as a Programming Language

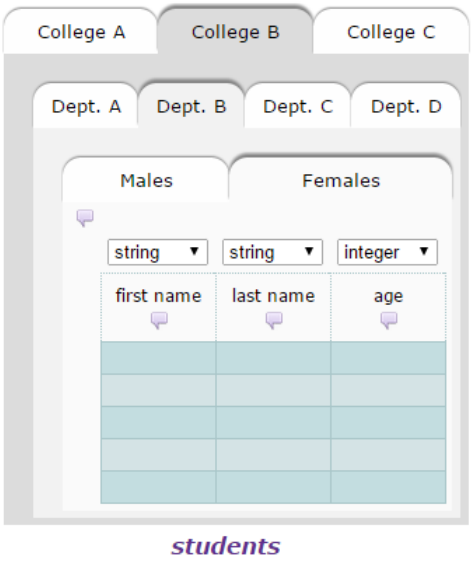
1) Data Structures: GLAF variables are based upon the concept of *grids*. Grids are simple, yet powerful data structures that can be used to represent a variety of real-world problems. A scalar variable is a 0D grid with one element and a 1D array is a 1D grid with multiple elements. Similarly, we can generalize for higher-dimensional data structures. Grids of this type may contain a single data type and be indexed by corresponding *index variables*. Allowing dimension(s) to have *titles* we can represent *tables*, in which case we may use a combination of titles and indices to address a specific grid cell. More complex structures can be described using the grid abstraction, by use of *multiple* data types across one of the dimensions with titles. Such grids can represent what would be a *struct* in C. Examples of grid declaration in GLAF are shown in Figure 2.

The *grid abstraction* has been used as the basis of programming languages or language extensions in the past ([1], [2]) due to certain advantages over using multiple, distinct data structures. In practical terms, the grid abstraction is *general* and *scalable* enough to model many real-world problems. A mathematical relation (a mapping from a domain to a range) that is discrete and finite can be represented with a grid (e.g., trees, graphs, databases). For example, a graph can be represented by using an adjacency matrix, a tree data structure can be substituted by matrices indicating the parent-child relationships and a sparse matrix can be represented in the compressed sparse row format (CSR). More importantly, the *regularity* of the grid abstraction allows for a uniform internal representation (Section II-C) that in turn renders code generation and many types of transformations and optimizations, including parallelism analysis, more straightforward. In terms of programming, coding an algorithm using the grid abstraction urges programmers to think in terms of relationships as opposed to exact data layout. For instance, representing a tree requires considering the higher-level relationships within the data (e.g., *parent_of*, *child_of*), instead of the exact tree structure and how it is coded on the lower language-level , e.g., using pointers in C and how to follow them. All things considered, the grid is a *familiar* abstraction (e.g., images, matrices, spreadsheets) to programmers and non-programmers, alike, makes visualizing certain data structures easy and inspecting results more intuitive. In some cases, however, and despite the aforementioned advantages, visualization-wise the original data structures may be more intuitive. Visualization-related issues are further discussed in Section III-D.

2) Programming Constructs: In Figure 1 we see an instance of a GLAF step. A *step* is the basic building block of a GLAF program and represents a step of computation, whereby data from input grid(s) flows after undergoing



(a) Declaration of a struct.



(b) Declaration of a simple database.

Figure 2: Examples of grid declaration in GLAF.

computation to an output grid. It may include a *loop* over zero, one or more dimensions. Such loops can be typical *foreach* loops (in the *start:end:step* format), or *forever* loops. Furthermore, a step may include *conditional statements*. Appropriate buttons insert the corresponding conditional keyword when a condition box is clicked on, and boxes are *indented* accordingly to make the order (and potential nesting) of conditionals clear. A *formula statement* can include grid cells, arithmetic/logical operations on them, as well as user-defined or library function calls (i.e., predefined sets of useful functions, like typical *Math* or *I/O* library functions). A GLAF program may include multiple steps, which belong into one (main) or more (user defined) functions. Functions, finally, can be grouped into GLAF *modules*.

C. Internal Representation

All constituent GLAF elements have a corresponding internal representation in JavaScript. As the user develops an algorithm using GUI buttons and keyboard, event listeners activate appropriate JavaScript functions to populate JavaScript objects modeled after *constructor functions* (in support of an object-oriented programming JavaScript). These are used in creating and navigating the GUI screens, in code generation, parallelism analysis, etc. They can also provide a preliminary error checking substrate (e.g., disallows declaring the same grid name twice) before code is generated, thus minimizing multiple compiler errors at the last development stage. A representative selection of the most important internal objects is outlined below:

(a) **Expression Type object:** defines the type of an *expression object* (e.g., grid, function call, string, number, conditional statement), (b) **Function object**: models a function and contains information about grids declared in a function, its arguments, its return value, etc. A function object encapsulates an array of *step objects* containing information about its steps, (c) **Step object:** models a step and contains information about all grids used in it, as well as information about the code in this step contained in arrays of box *expression objects*, (d) **Expression object:** models a line of code, can represent a *formula, a loop, or a conditional statement* and be a single object or comprise more expression objects in a tree-like structure, (e) **Grid object:** models a grid object according to the guidelines set in II-B1 (name, number of dimensions and their sizes, etc.)

For parallelism analysis the above objects are used in a single pass to formulate an *additional collection* of objects that store information about *scalar* and *non-scalar* grids (Figure 3). Finally, for non-parallelizable steps, detailed information is stored in appropriate objects that may be used in a feedback functionality. Such objects collect information about the name of grid, its type (scalar/non-scalar), error type (e.g., RAW dependency), and the name of function(s), steps and box numbers ("line of code").

IndVarsWrittenInStep[M][F][S]: 3D structure, elements correspond to a step (in a given module/function) containing info on the index variables iterated over on this step's loop.
FuncsFromCaller[M][F][S]: 3D structure, elements correspond to a step (in a given module/function) containing info (like name, ID, arguments per each call within step) about each function called from within this step.
NonScalarGridsInFunc[M][F]: 2D structure, elements correspond to a function (in a given module) containing info (like name, written/read, indices written/read for each dimension) about non-scalar grid arguments of function.
NonScalarGridInstances[M][F][S]: 3D structure, elements correspond to a step (in a given module/function) containing info (like name, written/read, indices written/read per dimension) on non-scalar grids written/read in step.
ScalarGridInstances[M][F][S]: 3D structure, elements correspond to a step (in a given module/function) containing info on scalar grids written/read in this step.

Figure 3: Internal representation objects for parallelism analysis back-end.

III. Capabilities

A. Code Generation

To support GLAF data visualization within the GUI, JavaScript code generation is *de-facto* supported. However, to broaden GLAF's utility we have implemented support of code generation and optimizations for *Fortran* and *C*,

```
int ft_calcPointCharge(int *ft_n_atoms, double *ft_sum_Fs,
int ft_curr_surf_pt, struct TYP_surface_pts typvar_surface_pts,
struct TYP_surface_pts typvar_atoms, double ft_Ke) {
    int fun_curr_surf_pt;
    double fun_Ke;
    int ft_ReturnValue;
    int ft_row;
    int ft_end0;
    int atoms_q=0, atoms_x=1, atoms_y=2, atoms_z=3;
    int surface_pts_q=0, surface_pts_x=1, surface_pts_y=2,
        surface_pts_z=3;
    double ft_r2;
    fun_curr_surf_pt = ft_curr_surf_pt;
    fun_Ke = ft_Ke;
    // Loop through all atoms vs single surf pt
    ft_end0 = 4-1;
    double tmp_sum_Fs=ft_sum_Fs[fun_curr_surf_pt];
    #pragma omp parallel for collapse(1) private(ft_r2) \
    reduction(+: tmp_sum_Fs)
    for (ft_row = 0; ft_row <= ft_end0; ft_row += 1) {
        // Calculating the distance between a surface point
        // and each atom
        ft_r2 = ft_calcDistance(typvar_surface_pts,
                fun_curr_surf_pt,typvar_atoms,ft_row);
        // Add current pairs charge to total
        tmp_sum_Fs = tmp_sum_Fs + ft_calcPairCharge(fun_Ke,
                     ft_r2,typvar_surface_pts,fun_curr_surf_pt,
                     typvar_atoms,ft_row);
    }

    ft_sum_Fs[fun_curr_surf_pt]=tmp_sum_Fs;
}
```

Figure 4: Example of automatically generated C code.

```
struct TYP_surface_pts {
  float dim0;
  float dim1;
  float dim2;
  float dim3;
};
// Surface points outside the biomolecule
struct TYP surface pts *typvar surface pts;
// Initialization of array of structures (AoS)
typvar_surface_pts = (struct TYP_surface_pts *)malloc(
                     sizeof(struct TYP_surface_pts)*3);
// Value assignment
typvar_surface_pts[ft_col].dim0 = ft_col * 3;
```

(a) Code for AoS.

```
struct TYP_surface_pts {
  float *dim0;
  float *dim1;
  float *dim2;
  float *dim3;
};
// Surface points outside the biomolecule
struct TYP surface pts typvar surface pts;
// Allocation of each array within the struct (SoA)
typvar_surface_pts.dim0=(float *)malloc(sizeof(float )*3);
// Value assignment
typvar_surface_pts.dim0[ft_col] = ft_col * 3;
```

(b) Code for SoA.

Figure 5: Generated code for declaration and accessing different data layouts.

languages that have been extensively used in technical/ scientific computing. Users need *not* write a single line of Fortran- or C-specific code, since GLAF follows a paradigm of *writing an algorithm in one language (GLAF) and getting source code in many*. Much of the complexity and peculiarities of languages are concealed from the user, thus facilitating code development for domain experts.

Code generated for a target language by GLAF falls under *two* broad categories, i.e., *serial* and *parallel*. The former is a one-on-one mapping of the GLAF programming constructs, as represented internally, to the target language's constructs, while the latter decorates parallelizable regions with appropriate *OpenMP directives* so that code can run on any device supporting OpenMP (e.g., multi-core CPUs, Intel Xeon Phi). Detecting parallelism is a task undertaken by the *GLAF parallelization analysis back-end* (Section III-B). Following those two code categories (*serial, parallel*) we provide a *secondary level* of auto-tuning options (Section III-C).

Figure 4 shows generated C code for the parallel implementation of a GLAF step (Figure 1) (Fortran look is similar). One can notice the *readability* of the automatically-generated code: code is *indented* and contains appropriate *comments*, as inserted by the user using the GUI. Depending on the user, the above features may vary on the importance scale: a non-programmer domain expert may never need to see the actual code. On the other hand, users that know programming (although *not necessarily* parallel), may want to inspect the code, apply further optimizations, or write a GLAF module, auto-tune taking into account legacy code's requirements (e.g., data types and layout of a function's arguments) and use the resulting parallel generated code as part of that larger unoptimized or serial legacy code.

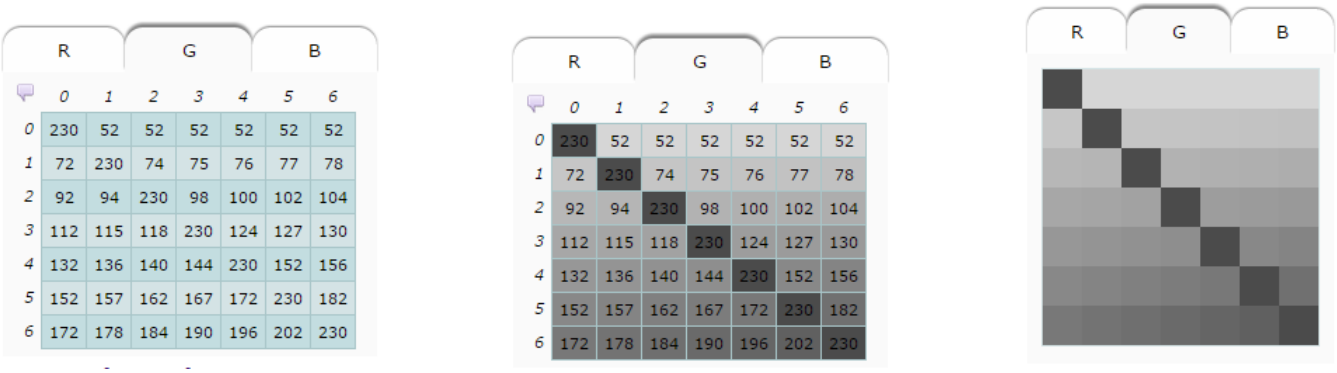
B. Auto-Parallelization

An important part of GLAF is its *parallelism analysis back-end*. Parallelism is analyzed at the granularity of a GLAF step and the result of this analysis is two-fold: (a) *information about the loop index variables whose loops are parallelizable*, (b) *information about the reasons parallelization is not possible*. Our analysis and resulting code transformations act in a complementary manner to a compiler providing code more favorable to compiler optimizations and do not claim to surpass its well-established capabilities.

For (a), a form of cross-iteration loop dependence analysis is carried out in a pre-code-generation pass to identify dependencies within a GLAF step. High-level pseudocode is given in Figure 6. This analysis includes: *non-scalar* grid variables (includes grids passed in functions called from within a step), and *scalar* grid variables (for which we build a control flow graph that is subsequently used to identify dependencies). More details about information collected in this pre-pass stage are given in Section II-C. Relevant parallelism information is used in parallel code generation and displayed in a *parallelism meter* on each step's header indicating the index variable name and the number of iterations that can be parallelized. For (b), each dependency is recorded and GLAF can provide feedback about module name(s), function name(s), step(s) and line(s), as well as a description of the reason parallelization fails.

```
Function findParallelismInProgram()
For each module M
  For each function F of M
    For each non-scalar grid argument G
      Record name of G in NonScalarGridsInFunc[M][F]
    Call recordGridsAndIndicesOfStepBoxes(M, F)
Call findParallelismInFunction(MainModule, MainFunc)

Function findParallelismInFunction(M: <module ID>, F: <function ID>)
For each step S of F of M
  Parse Index Range box, record index var. in IndVarsWrittenInStep[M][F][S]
  Call buildCFG() (builds control flow graph used in scalar grid dep. analysis
  For each box B after the Index Range box
    If B is of type Mask Statement (if/elseif)
      Add non-scalar/scalar grids in Mask to NonScalarGridInstances[M][F][S]
    Else if B is a Formula of the form G = RHS_expr
      Call addGrid(G) and parseRightHandSide(RHS_expr)
    Else if B is a formula of the form LET N = RHS_expr,
      Call parseRightHandSide(RHS_expr)
    Else if B is a stand-alone function call of the form func(args)
      Add func(arg) info to FuncsFromCaller[M][F][S]
  // Do cross-iteration dependency analysis among non-scalar grids, in two
  // steps: (i) For each function in FuncsFromCaller[M][F][S], detect and
  // record iteration dependencies among its arguments
  Call doDependAnalForNonScalarGridsPassedToAllFuncs(M, F, S)
  // (ii) For each grid in this step, find iteration dependencies ignoring any
  // function calls (since they were processed in (i))
  Call doDependAnalForNonScalarGrids(M, F, S)
  Call analyzeConstantIndices(M, F, S)
  Call doScalarDependAnalysis(CFG)
  Call estimateOverallParallelization(M, F, S)

Function addGrid(G: <grid to be added>)
If G scalar and entry for G does not exist in ScalarGridInstances[M][F][S],
add entry E for G to ScalarGridInstances[M][F][S] or if G non-scalar do so
for NonScalarGridInstances[M][F][S]
Init./update E to indicate if G is read/written and record attributes of G (e.g.,
box id, box type where G is found if scalar, or dimension read/written and the
index (location) from/to which each dimension is read/written if non-scalar)

Function addNonScalarForFunc(NS: <non-scalar grid to be added>, M: <mo-
dule ID>, F: <function ID>)
If NS is in NonScalarGridsInFunc[M][F], update information of corresponding
element E named after NS as read/written in NonScalarGridsInFunc[M][F] and
store current instance (i.e., index) for each of the dimensions

Function doDependAnalForNonScalarGrids (M: <module ID>, F: <function
ID>, S: <step ID>)
gridsInstances = NonScalarGridInstances  or NonScalarGridsInFunc depen-
ding if called from MainFunc or not)
For all elements E of gridsInstances[M][F][S] whose grid G is written
  For each dimension D of E
    For all combinations of write instances I1,I2 in D
      If I1,I2 write locations are different and contain index variable I from
      IndVarsWrittenInStep[M][F][S], mark I as non-parallelizable in G
    For each combination of write & read instances I1, I2
      If I1, I2 write/read locations are different & contain index variable I from
      IndVarsWrittenInStep[M][F][S], mark I as non-parallelizable in G

Function parseRightHandSide(RHS_expr: <expression appearing as right-hand
side of assignment)
For each element E in RHS_expr, call addGrid(G) if E is a grid G or add F
call's information to FuncsFromCaller[M][F][S], if E is a function call

Function recordGridsAndIndicesOfStepBoxes(M: <module ID>, F: <function
ID>)
For each step S in function F, record (in NonScalarGridsInFunc[M][F])  each
grid G that appears in boxes of S, if G is an incoming argument to F. For each
function-call foo(args) in S that passes G as an argument, record  foo and its
relevant info in FuncsFromCaller[M][f], and call this method recursively.

Function doDependAnalForNonScalarGridsPassedToAllFuncs(M: <module
ID>, F: <function ID>, S: <step ID>)
For each function-call FC in FuncsFromCaller[M][F][S] and for each non-
scalar grid G passed as an argument in FC's call
  Find corresponding name G of FC's argument GC in F's argument list
  If G present and read-only in S
    If GC read-only in FC, do nothing
      Else if GC is written or both read and written in FC
        Analyze all valid pairs of indices across all dimensions D that G is read in
        F and GC written (or written and read) in FC and if indices differ and
        contain index variable I from IndVarsWritenInStep[M][F][S], stop check-
        ing for I and mark I on G as non-parallelizable
  Else if G present and written (or written and read) in S and written or read (or
  written and read) in FC
    Analyze all valid pairs of indices across all dimensions D that G is written (or
    written and read) in FC and written (or written and read) in FC and if indices
    differ and contain index variable I from IndVarsWrittenInStep[M][F][S], stop
    checking for I and mark I on G as non-parallelizable
  Else if G not present in current step of F
    Call nonScalarGridDependencyAnalysis(MC, FC, SC)
    Call analyzeConstantIndices(MC, FC, SC) for all steps SC of FC of MC

Function estimateOverallParallelization (M: <module ID>, F: <function ID>, S:
<step ID>)
Check and mark if parallelism is broken for each index variable I in
IndVarsWrittenInStep[M][F][S], because of:
  - start/end/ step scalar variable for I being written in S
  - a scalar grid  (using info from scalar grid dependency analysis)
  - a non-scalar grid (using info from non-scalar grid dependency analysis)
For each func. called from S of F of M, call findParallelismInFunction(M, F)

Function analyzeConstantIndices(M: <module ID>, F: <function ID>, S: <step
ID>)
gridsInstances = NonScalarGridInstances or NonScalarGridsInFunc (depending
if called from MainFunc or not)
For each G in gridsInstances[M][F][S] that is written
  Check the indices of the instances where G is written and identify the cases
  where a dimension's index is always a constant as non-parallelizable

Function doScalarDependAnalysis(CFG: <control flow graph for a step>)
For each scalar grid occurrence X that is written at least once
  Push CFG root node in stack
  While stack not empty
    N = pop last element from stack
    If X is written in current node N
      If there is a read of X in N
        Identify write after read condition, reduction, break
      Else do nothing, break
    Else if X is not written in N
      If there is a read of X in N
        Identify write after read condition, reduction, break
      Else push each child of N to stack
  If reduction found, mark as a reduction clause
```

Figure 6: Parallelism back-end pseudocode (related internal objects used are described in Figure 3).



(a) Values only. (b) Values and colorization. (c) Image map.

Figure 7: Examples of data visualization methods in GLAF.

C. Auto-Tuning

The auto-tuning menu screen includes options regarding *source code, binaries, compilation and execution scripts generation*, as well as *execution times presentation* for the resulting implementations. At the **first level** the user selects the *target platform* (currently *CPU* and *Xeon Phi*). This option identifies the appropriate compilation flags to be passed to the generated platform-specific auto-tuning script. In the future, this option will allow *platform-specific* optimizations. The **second level** of auto-tuning concerns the *target language* (currently *Fortran* or *C*). The **third level** offers three different basic implementations *within* a selected language and platform: (a) *serial:* serial code, as automatically generated by GLAF, (b) *GLAF-parallelized:* parallel implementation of the code, in which appropriate OpenMP directives precede parallelizable code sections, as identified by the auto-parallelization back-end, (c) *compiler-parallelized:* with this option serial code in (a) is to be compiled with the auto-parallel compiler flag (*-parallel*) - this flag is not used in (b) where the code includes explicit OpenMP pragmas and is compiled with the *-openmp* flag. The **fourth level** expands our choices in code generation. Options at this level can be *combined*. The first option showcases the flexibility in *expressing data structures*, and creates implementations where grids are expressed as either *Structures of Arrays* (SoA) or *Arrays of Structures* (AoS), if applicable (i.e., in the form of Figure 2a). Figure 5 shows code automatically generated in C for declaring, initializing and using such data structures. The second option, tunes the level of *loop collapsing* for nested parallelizable loops and the third concerns *loop interchange*, when applicable. Particular options from this category are more suitable for specific algorithmic patterns and their effect is reflected in the execution time of an application written in a specific way. We discuss such concerns in more detail in Section V, where we present results for representative cases. Users who are familiar with the options' meaning and implications may select to prune the auto-tuning space by deselecting certain options. Non-programmers may simply let the auto-tuner generate code using all applicable permutations and present the best combination (and corresponding code).

D. Visualization

Data visualization facilitates understanding the algorithms being developed, as well as revealing bugs, in an *intuitive, visual way*. Currently, GLAF supports *two* ways of visualizing data. By selecting the *"Show Data"* menu item in a step, data for each grid cell are calculated up to that step by evaluating automatically generated JavaScript code on-the-fly and using the internal representation of data structures, which is by design in JavaScript (Section II-C). The user can navigate in multi-dimensional grids by clicking on the appropriate dimension. For grids whose dimensions are larger than the screen space, appropriate arrows enable navigating within grid contents. The *"Colorize"* menu option paints each grid cell on the greyscale color spectrum according to the magnitude of the corresponding cell value, together with its value. Finally, the *"Image Map"* option, a straightforwad extension of the previous two, contains only color (i.e., no data values). Techniques based on color make it easy to spot *outlier* values or specific *patterns* in relative problems, especially in the presence of huge amounts of data, thus facilitating result observation and interpretation. They are especially useful in the image processing field, as well. Figure 7 shows three simple examples of data visualization functionalities. In the future, more *complex* visualization schemes can be built upon the unified and regular internal representation of the grid abstraction by use of specialized visualization libraries for frequently-used data structures. For example, sparse matrices in CSR format could be visualized automatically as single (sparse) matrices, rather than the collection of the helping 1D arrays that represent such matrices in CSR format. Similarly, graphs could be visualized as such, while internally represented via adjacency matrices.

IV. Example Applications

To assess GLAF's code generation, parallelism analysis and auto-tuning back-ends in diverse situations, we resort to applications that fall under the *dwarfs classification* [3]. We focus on the *structured grids* and *n-body methods* dwarfs to illustrate how easy it is for novice programmers using typical programming languages to fall into *coding pitfalls*, their effect in performance, and how GLAF addresses these problems in an automated way thus providing a fair trade-off between performance and programmability.

A. 3D Finite Difference Calculation (3DFD)

In structured grids algorithms computation proceeds as a series of grid update steps. In the 3D case data is arranged in a regular 3D grid. For each step iteration each point is updated as a function of its neighboring points' values across each of the 3 dimensions. In our example each point for step $n+1$ is the value at step n plus the sum of 8 neighboring points across each dimension (Listing 1). We define a source and destination grid, which are used interchangeably at each computation iteration. In 3DFD, we have ample parallelism within a step iteration: within each dimension summation of all 8 neighboring values for each point can be seen as a reduction and on the outer level (3 nested *for* loops) each grid cell update can be calculated independently.

```
for number of steps/iterations S
 for each point[i,j,k]
 (incl. vertical boundary condition)
   sum[i,j,k]+= 8pt stencil across horizontal dim
 for each  point[i,j,k]
 (incl. horizontal boundary condition)
   sum[i,j,k]+= 8pt stencil across vertical dim
 for each  point[i,j,k]
 (incl. 3rd dim. boundary condition)
   sum[i,j,k]+= 8pt stencil across 3rd dim
```

Listing 1: 3DFD pseudocode

B. Electrostatic Surface Potential Calculation (ESP)

In ESP we calculate the electrostatic potential at a collection of points on the surface of a biomolecule. The potential at each point is the sum of charges contributed by its interaction with all atoms within the biomolecule (Listing 2). Each surface point and atom is represented as a structure containing information about its electric charge and its coordinates in the 3D space (Figure 2a). In ESP we have *two levels* of parallelization opportunities for the two nested loops described above (the latter being a *reduction*). Computations are *independent* for each surface point and for each surface point/atom pair.

```
for all surface points sp[i]
 for all atoms at[j]
   sq_dist = (x[i]-x[j])^2 + (y[i]-y[j])^2 + (z[i]-z[j])^2
   sum_esp[i]+= K_e * (q[i] * q[j])/sq_dist
```

Listing 2: ESP pseudocode

V. Results

Our experiments were run on: (a) a dual-socket **Intel Xeon E5-2697 CPU** (each with 12 cores at 2.6GHz, 256KB of L2 and 30MB of L3 cache) , and (b) an **Intel Xeon Phi (XP) 7120** co-processor (61 cores at 1.238GHz, 30.5MB cache). We used the Intel set of compilers (ifort for Fortran, icc for C, in Intel Composer XE 2015 v.15.0.1), Intel Vtune Performance Analyzer and compiler optimization/parallelization reports to identify performance issues.

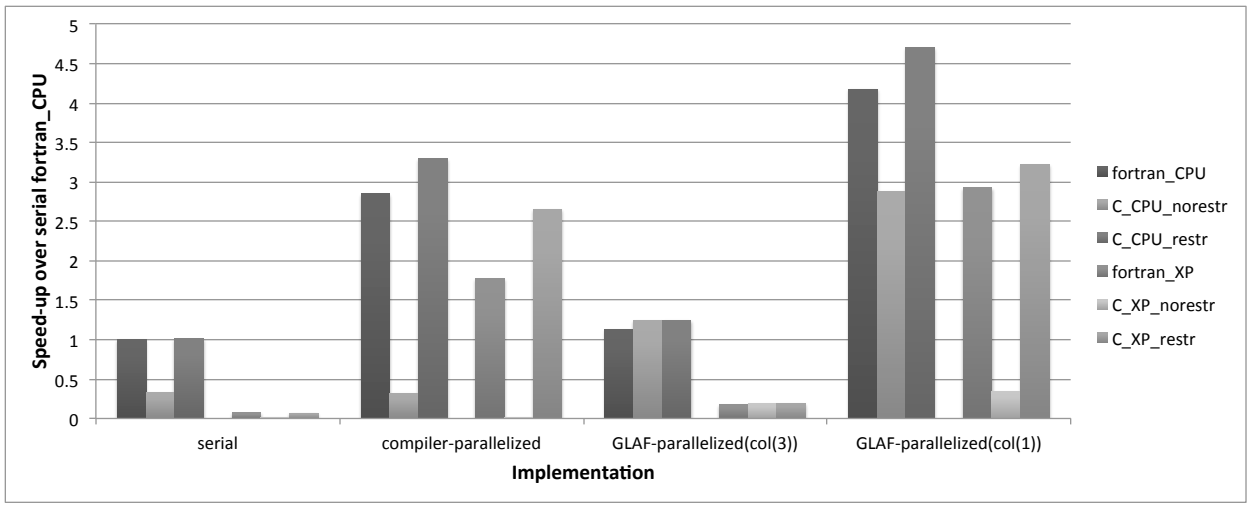
In our example applications, we generate *serial*, *compiler-parallelized* and *GLAF-parallelized*, implementations (see Section III-C for details) for both Fortran and C, with appropriate optimizations applied by the auto-tuning back-end. For explicit performance comparisons graphs for our experiments show speed-up (y-axis) over a single baseline implementation (Fortran CPU serial) for the most relevant implementations (x-axis) as created by the auto-tuning framework. The serial baseline code highly resembles code that a novice programmer would write (e.g., Figure 4).

A. 3DFD Results

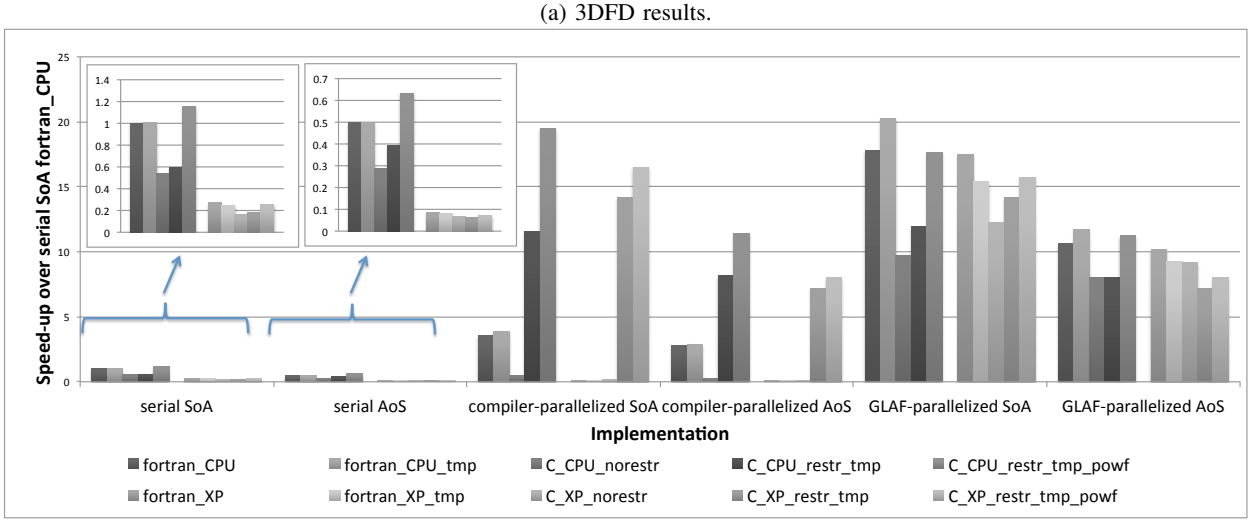
For the 3DFD experiment we run the algorithm for 20 steps and a 512x512x512 input grid. There are no *data layout* concerns (i.e., SoA, AoS) in 3DFD, however, *loop collapsing* and *loop interchange* auto-tuning options are applicable. For the former, we present results for *collapse(1)* (i.e., no collapsing), and *collapse(3)*. For the latter, we auto-generate the fastest loop ordering, according to the target language's way of arranging multi-dimensional arrays in memory (*row-major* for C, *column-major* for Fortran).

The best performance is obtained using the C_CPU_restr GLAF-parallelized(col(1)) implementation (4.7x), followed by the corresponding implementation in Fortran (4.17x). Despite this performance difference *across* languages, *within* each language and platform, C achieves *better* speed-up (over serial) than Fortran (e.g., 38.37x for compiler-parallelized C_XP_restr versus 24.64x Fortran_XP). Moreover, a non-negligible 1.49x performance gain (on XP, compiler-parallelized implementation) can be obtained by switching from Fortran to C. GLAF renders this trivial by its ability to automatically generate code in both languages.

In Figure 8a, we observe that C_CPU_restr and C_XP_restr perform overall better than their *norestr* counterparts in all implementations. *Norestr* corresponds to an earlier stage of C code generation in GLAF. In the case of C *norestr* implementations, we observe that compiler-parallelized code fails to provide any speed-up versus the corresponding serial. C enforces strict aliasing rules, and as such the compiler acts conservatively by not parallelizing the three nested loops (Listing 1) and assuming the existence of dependence between the pointer variables for each grid in a function call. To indicate no aliasing a C programmer would need to use the *restrict* keyword and *-restrict* compiler flag. In Fortran, grids (i.e., Fortran arrays) are always passed by



(a) 3DFD results.



(b) ESP results

Figure 8: Performance results for the example applications developed, auto-tuned by the GLAF framework.

reference and are assumed to not alias by default. Strict aliasing rules, when unnecessarily enforced, do *not* only affect auto-parallelization, but vectorization, too. Namely, it seriously hinders serial C performance, as well. Note the huge difference (*restr* vs. *norestr*) in XP (16.04x), as opposed to 3.07x in CPU, owing to the wider vector units in XP compared to the CPU. Novice programmers are typically *not* aware of issues like aliasing and related solutions. This is another point in favor of automatic code generation by GLAF, doing so in multiple languages, as well as GLAF's simplicity in not allowing aliasing by default. As such, the above remedies can be taken care of by the framework at code-generation and compilation script creation time.

In some cases, where nested loops are parallelizable, it may make sense to collapse them. A novice programmer with rudimentary OpenMP knowledge and lack of architecture knowledge could have coded these nested loops with nested OpenMP pragmas, spawning an increasing number of threads with the associated overheads, or with a *collapse(3)* OpenMP clause. Using *collapse(1)* (i.e., parallelizing only across one dimension) allows the compiler to generate *vector code* for the remaining two (parallelizable with *unit stride*) loops. This increases performance as a function of the vector unit's width. As such the relevant performance gain between those two options is higher (16.8x) in the case of Xeon Phi (512-bit wide vector units), than in the lower (3.7x) CPU case (256-bit AVX) in *both* Fortran and C implementations.

B. ESP Results

For ESP we use a problem size of 128000 surface points and 256000 atoms. ESP illustrates the utility of GLAF's *data layout transformations*. *Structure-of-arrays* and *array-of-structures* implementations can be automatically generated by GLAF for the surface point and atom grids (Figure 5).

The best overall performance for ESP is obtained using the Fortran GLAF-parallelized SoA implementation in the CPU, followed by the corresponding C CPU implementation. The fact that the language of choice for EPS is *different* than that of 3DFD emphasizes the utility of automatic code-generation in multiple target languages. In many instances Xeon Phi execution proves slower than the corresponding CPU C and Fortran implementations. We expect to be able to achieve better speed-ups for the Xeon Phi case, once GLAF provides MIC architecture-specific hints to the compiler (e.g., alignment attributes that facilitate more efficient vectorization, aligned loads, cache miss reduction).

As in the case of 3DFD we observe compiler-parallelized C implementations (*norestr*) to initially exhibit subpar performance, equal to the serial. According to the compiler parallelization report the reason for failing to parallelize the outer loop was "insufficient work". This was an erroneous assessment by the compiler, given the available amount of work. In fact, the 2013 (v.13.1.3) version of icc *did* parallelize the same loop when the number of loop iterations was larger than the number of (logical) processors. The latest compiler was eventually able to automatically parallelize this loop once we performed (via GLAF) another optimization that the compiler was not performing itself. In particular (*restr_tmp*), we introduce a temporary variable in the nested loop of Listing 2 in place of the *sum_[i]* array, which further allows the loop to be identified as a reduction (C does not otherwise allow reductions on dynamically allocated arrays). GLAF applies the same code transformation for Fortran.

ESP is an illustrative case for the utility of *data layout* transformations. Results highlight how a bad choice for structs declaration (AoS here) account for twice worse performance. SoA layouts are amenable to efficient vectorization and offer the possibility for unmasked *unit stride* loads, as opposed to more expensive *strided* loads, and *gather* operations employed with AoS. Profiling data across our implementations validate AoS detrimental effect, especially on cache, as CPU and XP speed-ups denote. Especially for the serial fast implementations performance degradation ranges between 1.52 to 2x (C, Fortran in CPU) and 2.89 to 3.15x (C, Fortran in XP). In SoA cases, all vector loads in Fortran are aligned, whereas in C the majority is unaligned, resulting in higher load latencies. In both SoA and AoS CPU cases C is better in its compiler-parallelized and worse in the GLAF-parallelized implementations, while in XP it far surpasses the corresponding Fortran compiler-parallelized implementations and performs similarly for the GLAF-parallelized. While GLAF provides the answer to what the best implementation is, we can imagine the pitfalls a novice user would fall into if he were to write a serial version in the "wrong" language and/or in the "wrong" struct format (i.e., failure of compiler auto-parallelization).

Finally, C implementations with *powf* suffix (Figure 8b) highlight how a simple oversight can affect performance. In particular, our code-generation back-end initially produced a *pow()* function call for the corresponding function in GLAF. While in Fortran the exponentiation intrinsic function (**) is overloaded with the appropriate version according to the arguments, in C *pow()* emits the *double* version. Ignoring calling appropriate versions of a function in C is common among non-programmers. Use of *powf()* instead of *pow()* increased C serial CPU performance by 1.94x for the SoA and 1.61x for the AoS case. Argument type detection is inherent within the framework (Section II-C) and detection of such cases of potential unwanted performance degradation is important in steering users away from such pitfalls.

VI. Related Work

In an effort to achieve parallelism gains a programmer has an array of tools at his disposal, some of which attempt to automate the conversion of sequential code to parallel in a *transparent* way and others that require the user to *explicitly* identify and indicate available parallelism by hinting the compiler through directives or other special notation. The former category includes modern compilers, like GCC, Intel/Cray/PGI Compilers, which perform loop parallelization and code vectorization, among a wider collection of optimizations. The latter includes languages, extensions, libraries, such as the well known Pthreads, OpenMP, OpenACC, Intel Cilk Plus [4], Chapel [5]. Exploiting the above solutions requires programming knowledge, either sequential in the former case or a certain understanding of parallel in the latter. This is a task that domain experts are often reluctant to undertake. But, even in the case of novice programmers' attempting to exploit the auto-parallelization features of compilers failure may ensue due to restrictions enforced by the *conservative* nature of compiler optimization algorithms. We discuss such examples in Section V. Our work *complements* the first category, by generating code that is *more amenable* to auto-parallelizing compilers and that *automatically* takes advantage of extensions like OpenMP.

A different research path attempts to target domain experts with problem solving environments (PSEs), with examples from the computational biology, physics, or even more specific subareas within a class of problems ([6], [7], [8]). Such solutions offer the added benefit of high-performance, often a result of optimizations based on domain knowledge. This specific nature of PSEs and auto-tuning frameworks, however, is a double-edged sword, in that it also introduces the problem of lack of generality of applicability. Such solutions focus to very function-specific codes, like dense linear algebra code [9] or FFT [10]. Irrespective of that, most are restrictive in terms of target language and/or architecture (e.g., GPU, CUDA, OpenCL [11], [12]). PSEs and auto-tuning frameworks, like the ones mentioned above have their indisputable merits and strengths, based on their intended purpose. Our work attempts to address the need for a programming abstraction and framework that is *general* enough

to be of use across domains, as problems in engineering and sciences may be composed by multiple different parts that the restrictive nature of auto-tuners may not be able to directly address. *Generality* also refers to multiple target languages and architectures. Grid-based data structures, which lie at the basis of GLAF, have also been the central datatype of languages/extensions (APL [1], NumPy [2]). In GLAF the grid data structure is – among others – meant to support a programming paradigm that resembles the familiar *spreadsheet* workflow (where cells/tables undergo transformations based on formulas/macros). GLAF extends this familiar paradigm in ways to enable complex, general-purpose program development, and addresses the need for performance via parallelism support and other optimizations in automatically generated code. Last, we enhance upon existing work by *integrating* the visualization aspect in programming, where data and the operations on data (i.e., code) coexist during the development stage.

VII. Conclusions and Future Work

In this work we presented GLAF, an all-encompassing code development environment for non-programmers or novice programmers, like the majority of *domain experts*. Its key elements are its *intuitive* visual programming interface that aspires to make it easy for them to express and validate their algorithms and its ability to *automatically* generate efficient serial and parallel Fortran and C code, including an array of potentially beneficial code modifications with respect to data layout, loop ordering, etc. for *fast* execution.

We showed illustrative example applications and presented results that corroborate that GLAF and its associated approach that attempts to bridge the gap between performance and programmability may be a useful addition in the array of programming tools at the disposal of domain experts. We showed how automatically generating multiple versions of code (derived from the *same* GLAF code) in different languages and using different data layouts and optimizations can facilitate obtaining the best performing code. This systematic code generation and auto-tuning approach extends beyond standard optimizations and tuning that typically target a single language and is especially useful for novice programmers, where certain mistakes in a language may result in code that the compiler fails to auto-parallelize. Our findings reveal that the traditional coding paradigm, where a single implementation is written, can be sub-optimal for novice (or even average) programmers. Rather, GLAF allows multiple starting points (analogous to different seeds in a state-space search algorithm) for different optimizations, which lead to overall better performance (analogous to the global as opposed to local minimum).

Our GLAF prototype incorporates core functionalities that form the *substrate* for an extensible set of capabilities. For future work we plan to enable support for more languages and extensions (e.g., OpenACC), distributed-memory systems via MPI, and dynamic and *user-friendly* feedback concerning issues that limit parallelism. Finally, we seek to improve the robustness of the auto-parallelization back-end and implement additional auto-tuning options.

Acknowledgment

This work was supported in part by the Institute for Critical and Applied Sciences (ICTAS) at Virginia Tech.

References

[1] K. E. Iverson, "A Programming Language," in *Spring Joint Computer Conference*. ACM, 1962, pp. 345–351.

[2] S. van der Walt, S. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[3] K. Asanovic et al., "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Dept., U. of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[4] A. D. Robison, "Composable Parallel Patterns with Intel Cilk Plus," *Computing in Science and Engineering*, vol. 15, no. 2, pp. 66–71, 2013.

[5] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[6] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf, "The Cactus Framework and Toolkit: Design and Applications," in *High Performance Computing for Computational Science VECPAR 2002*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2565, pp. 197–227.

[7] J. Davison de St.Germain, J. McCorquodale, S. Parker, and C. Johnson, "Uintah: a Massively Parallel Problem Solving Environment," in *The 9th International Symposium on High-Performance Distributed Computing*, 2000, pp. 33–41.

[8] A. Dubey, K. Antypas, M. K. Ganapathy, L. B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide, "Extensible Component-Based Architecture for FLASH, a Massively Parallel, Multiphysics Simulation Code," *Parallel Computing*, vol. 35, no. 10, pp. 512–522, 2009.

[9] R. Clint Whaley, A. Petitet, and J. J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Parallel Computing*, vol. 27, no. 1, pp. 3–35, 2001.

[10] M. Frigo and S. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[11] A. Davidson and J. Owens, "Toward Techniques for Auto-tuning GPU Algorithms," in *Applied Parallel and Scientific Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7134, pp. 110–119.

[12] Y. Li, Y. Zhang, H. Jia, G. Long, and K. Wang, "Automatic FFT Performance Tuning on OpenCL GPUs," in *IEEE 17th International Conference on Parallel and Distributed Systems*, 2011, pp. 228–235.