Designing the Language Liszt
for Building Portable Mesh-based PDE Solvers

Zachary DeVito
Niels Joubert
Crystal Lemire
Pat Hanrahan
Recap

✓ Niels: Implementation of the Liszt compiler on clusters, SMPs, and GPUs
✓ Pat: history of co-design in graphics
✓ Crystal and Niels: Hands-on with Liszt at the Hackathon!

o Today: Design of the Liszt language
  1. Insight into key design decisions we made
  2. Planned designs for new features and interoperability
Why use a DSL?

- Platform Independent
- More productive programming environment like R and MATLAB
- Specialize software like we specialize hardware
Successful Languages (portable)
Successful Languages (portable)
Challenges

• Expressing the important problems
• Elegant and simple design
• Possible to implement efficiently and portably
Liszt: Heat Conduction on Grid

```scala
val Position = FieldWithLabel[Vertex,Float3]("position")
val Temperature = FieldWithConst[Vertex,Float](0.0f)
val Flux = FieldWithConst[Vertex,Float](0.0f)
val JacobiStep = FieldWithConst[Vertex,Float](0.0f)
var i = 0;
while (i < 1000) {
  for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)
    val step = 1.0f/(length(dP))
    Flux(v1) += dT*step
    Flux(v2) -= dT*step
    JacobiStep(v1) += step
    JacobiStep(v2) += step
  }
  for (p <- vertices(mesh)) {
    Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
  }
  for (p <- vertices(mesh)) {
    Flux(p) = 0.f; JacobiStep(p) = 0.f;
  }
  i += 1
}
```

Mesh Elements

Topology Functions

Fields (Data storage)

Parallelizable for
Design

1. Level of abstraction
   – Element-wise formulation
2. Representing the domain
   – Topological functions and fields
3. Finding data-dependencies
   – Inferring the data-dependencies from the stencil
4. Parallel Language Semantics
Design Decision: Levels of Abstraction
Level of Abstraction

- Continuous PDE

\[
\frac{\partial \rho U}{\partial t} + \nabla \cdot \phi U - \nabla \cdot \mu \nabla U = -\nabla p
\]

solve

(\text{fvm::ddt(rho, U)}
+ \text{fvm::div(phi, U)}
- \text{fvm::laplacian(mu, U)}
==
- \text{fvc::grad(p)}
)

[OpenFOAM User Guide]
Level of Abstraction

- Element-wise Formulation: Liszt, OP2, Scout
Level of Abstraction

- Explicit Parallel Programming

\[
\begin{align*}
\text{head}_b[] &= \{F, G, E, F, H, G\} \\
\text{tail}_b[] &= \{D, D, F, G, F, H\}
\end{align*}
\]

\[
\begin{align*}
\text{head}_o[] &= \{A, C, C, D, C, E\} \\
\text{tail}_o[] &= \{B, A, B, B, D, C\}
\end{align*}
\]
Level of Abstraction

• Continuous PDE
  – ideal for already established numeric methods

• Explicit parallel programming
  – most flexibility, but limited portability to diverse architectures

• Element-wise formulation
  – much of the flexibility of explicit parallel programming
  – ability to run on multiple architectures
Design Decision: Representing the Domain
Representing the Domain

- Ad-hoc Storage

```
for(int v = 0; v < N; v++) {
    int v_start = xadj[v];
    int v_end = xadj[v+1];
    for(int i = v_start; i < v_end; i++) {
        int v2 = adjncy[i];
        //...
    }
}
```
Representing the Domain

• Relations $r(x) = \{y_0, y_1, \ldots, y_N\}$
  
  – 1 to N mapping from user-defined element of type $x$ to another type $y$:

  [Giles et al.]
Representing the Domain

- Relations \( r(x) = \{ y_0, y_1, \ldots, y_N \} \)

```c
// create a relation from an edge to each of its vertices
int edge_map[20] = {1,2,2,3,3,4,3,6,2,4,1,4,4,5,6,5,4,6,1,5};
op_map pedge;
#define edge_map pedge
```
Representing the Domain

- Mesh Topology

```latex
for(v <- vertices(mesh)) {
    for(v2 <- vertices(v)) {
        //...
    }
}
```
Why Meshes?

- Follows the way we visualize the domain
Why Meshes?

• More specific than a graph
  – we can use this for optimized data structures [Laszlo and Dobkin]
Why Meshes?

• Specialize code for different mesh types
  - Structured meshes:

  ```
  int x0 = v0 % 5;
  int y0 = v0 / 5;
  for(int x = v0-1; x <= x0+1; x += 2) {
    for(int y = y0-1; y <= y0+1; y += 2) {
      int v = NC * y + x;
    }
  }
  ```

for(v <- vertices(v0)) {
  //...
}
Field-based Data Storage

• Mesh-based representation leads naturally to Liszt’s fields
• Conceptually store the degrees of freedom for basis functions used to construct a continuous field

\[\begin{align*}
dof_f & : \text{Field[Face, Float]} \\
\text{val} \quad dof & : \text{Field[Vertex, Float]} \\
dof_v & : \text{Field[Vertex, Float]} \\
dof_e & : \text{Field[Edge, Float]} \\
\end{align*}\]

Constant \quad Linear \quad Quadratic
Design Decision: Expressing Data Dependencies
Data Dependencies

• Crucial to finding parallelism, exposing locality, and minimizing synchronization

• Determining data-dependencies in a general-purpose language is difficult and inaccurate

\[ A[i] = B[f(i)] \] – must compute \( f(i) \) to find dependency
Data Dependencies

• Explicitly provided by the user (OP2) [Giles et al.]

```c
//create a relation from an edge to each of its vertices
int edge_map[20] = {1,2,2,3,3,4,3,6,2,4,1,4,4,5,6,5,4,6,1,5};
op_map pedge;
op_decl_map(
    edges /*set of edge objects*/,
    nodes /*set of node objects*/,
    2, /*each edge has 2 nodes*/
    edge_map,/* the data */
    pedge /* pedge holds the relation*/
);
//run a function that specifies our relation as a dependency
op_par_loop_2(
    kernel, /* run function kernel */
    edges, /* on all edges */
    temperature, 1,pedge, 2,"float", OP_READ,
         /* each edge with read the temperature data from both vertices*/
    flux, 0,pedge, 2,"float", OP_INC /* and update the flux data */
);
```
Data Dependencies

• Affine Partitioning
  – Addresses are affine transformations of variables \( x, y, \) and \( v0 \)
  – Determine the stencil analytically [Lim and Lam]

```c
int x0 = v0 % 5;
int y0 = v0 / 5;
for(int x = v0-1; x <= x0+1; x += 2) {
    for(int y = y0-1; y <= y0+1; y += 2) {
        A[v0] += B[NC * y + x];
    }
}
```
Data Dependencies

- Inferring Dependencies from semantics of the mesh (Liszt)
  – can work for relations as well!

```scala
for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)
  val step = 1.0f/(length(dP))
  Flux(v1) += dT*step
  Flux(v2) -= dT*step
  JacobiStep(v1) += step
  JacobiStep(v2) += step
}
```
Data Dependencies

• Explicitly provided dependencies are verbose
• Affine partitioning only works for structured meshes
• Inferring dependencies based on mesh semantics address both issues
Parallel Language Semantics

Minimizing false dependencies

• Phased usage of Fields
  – Fields have field phase state
    • read-only, write-only, reduce-using-operator field(el) [op]= value
  – Fields cannot change phase within for-comprehension

• Associative Operators
  – Allow single expensive calculation to write data to multiple elements
  – Provide atomic scatter operations to fields
    • e.g. field(el) += value
  – Introduce write dependencies between instances of for-comprehension
Design

• Features
  – meshes, fields, topology functions, sets, for-comprehension

• Restrictions
  – parallel-for, associative reductions, field phases, immutable mesh variables

• Simplicity
  – built-in topological operators, simple fields
The Future...
Support More Applications
Interoperate with existing code
More Apps: Sparse Matrices

• Simple Example: Graph Layout

[Images from Friedel et al.]
More Apps: Sparse Matrices

- Simple Example: Graph Layout
  - Tutte’s spring embedding theorem:
    - Planar graph embedded in a convex polygon can be drawn without edge crossings by solving a linear system that places each vertex at the center of its neighbors

\[ P_v - \frac{1}{N} \sum_{i=0}^{N} P_{v_i} = 0 \]
More Apps: Sparse Matrices

object MyLS extends LinearSystem {
    def row(v : Vertex) : Vec[_2,RowIndex] = AutoIndex
    def col(v : Vertex) : Vec[_2,ColIndex] = AutoIndex
}
val A = MyLS.A(); val x = MyLS.x(); val b = MyLS.b()
<initialize boundary>
<initialize interior>
MyLS.solve(A,x,b)
More Apps: Sparse Matrices

object MyLS extends LinearSystem {
  def row(v : Vertex) : Vec[_2,RowIndex] = AutoIndex
  def col(v : Vertex) : Vec[_2,ColIndex] = AutoIndex
}
val A = MyLS.A(); val x = MyLS.x(); val b = MyLS.b()
<initialize boundary>
<initialize interior>
MyLS.solve(A,x,b)

<initialize boundary> ===
for(v <- boundary) {
  val row = MyLS.row(v)
  val col = MyLS.col(v)
  val sz = size(vertices(v))
  A(row,col) = Mat(Vec(1.0,0.0), /* For brevity: Identity2x2 */
                    Vec(0.0,1.0))
  val pos : Double2 = position(v)
  b(row) = pos
}
object MyLS extends LinearSystem {
    def row(v : Vertex) : Vec[2,RowIndex] = AutoIndex
    def col(v : Vertex) : Vec[2,ColIndex] = AutoIndex
}
val A = MyLS.A(); val x = MyLS.x(); val b = MyLS.b()
<initialize boundary>
<initialize interior>
MyLS.solve(A,x,b)

<initialize interior> ===
for(v <- interior) {
    val row = MyLS.row(v)
    val col = MyLS.col(v)
    val N = size(vertices(v))
    A(row,col) = -Identity2x2 /* − P_v */
    for(v2 <- vertices(v)) {
        A(row,MyLS.col(v2)) = Identity2x2 / N /* + P_v2 / N */
    }
    b(row) = Vec(0.0,0.0) /* = 0 */
}
Interoperate: Embedded DSLs
Interoperate: Embedded DSLs

- SQL

```c
mysql_query(conn, "SELECT data FROM images WHERE id=1");
MYSQL_RES * result = mysql_store_result(conn);
int num_fields = mysql_num_fields(result);
MYSQL_ROW row;
while (row = mysql_fetch_row(result)) {
    for(int i = 0; i < num_fields; i++) {
        printf("%s\t", row[i]);
        printf("\n");
    }
}
```
Embedded DSLs

• OpenGL

```c
char * shader = "void main()" 
{" 
  gl_FragColor = " 
  vec4(0.4,0.4,0.8,1.0);" 
}";

GLuint f = 
glCreateShader(GL_FRAGMENT_SHADER); 
glShaderSource(v, 1, &shader,NULL); 
glCompileShader(f);

GLuint p = glCreateProgram(); 
glAttachShader(p,f); 
glLinkProgram(p); 
glUseProgram(p);

glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glEnd();
```
Embedded DSLs

- Lua – Game Scripting

```lua
if (IsPartyLeader()) then
    ChatFrame1:AddMessage("I am the leader of my party!")
end
```
Embedded DSLs

- Lua – Embedded by design
  - Lua: “Passing a Language through the Eye of a Needle” [Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes]

```c
lua_State *L = luaL_newstate(); /* create a new state */
luaL_openlibs(L); /* open the standard libraries */

/* reads lines and executes them */
while (fgets(line, sizeof(line), stdin) != NULL) {
    luaL_loadstring(L, line); /* compile line to a function */
    lua_pcall(L, 0, 0, 0); /* call the function */
}
```
Liszt as an Embedded DSL

Scala Compiler
- Scala frontend
- Liszt plugin

Liszt Compiler
- platform-independent analysis

Native Compiler
- mpicxx
- c++
- nvcc

Runtime
- MPI app
- pthreads app
- CUDA app
- partitioning
- coloring

Platform
- Cluster
- SMP
- GPU

Runtime platform
- pthreads
- CUDA
Liszt as an Embedded DSL

Liszt compiler will generate a library for your Liszt code which you can then link in a general-purpose language.
val Position = Field[Vertex,Float3] /* value is not specified! */
val Temperature = Field[Vertex,Float](0.0f) /* value is not specified! */
val Flux = FieldWithConst[Vertex,Float](0.0f)
val JacobiStep = FieldWithConst[Vertex,Float](0.0f)

@entry def calculate_temperature() {
  var i = 0;
  while (i < 1000) {
    for (e <- edges(mesh)) {
      val v1 = head(e)
      val v2 = tail(e)
      val dP = Position(v1) - Position(v2)
      val dT = Temperature(v1) - Temperature(v2)
      val step = 1.0f/(length(dP))
      Flux(v1) += dT*step
      Flux(v2) -= dT*step
      JacobiStep(v1) += step
      JacobiStep(v2) += step
    }
    for (p <- vertices(mesh)) {
      Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
    }
    for (p <- vertices(mesh)) {
      Flux(p) = 0.f; JacobiStep(p) = 0.f;
    }
    i += 1
  }
}
Liszt as an Embedded DSL

/* recall: Temperature and Position weren't initialized in Liszt */

double * temperature_data = MyLoadData("my_initial_temperatures.dat")
LisztMesh mesh;
Liszt_load_fluent_mesh("myfluentfile.msh", &mesh);
Liszt_bind_mesh(mesh);
Liszt_load_field(temperature_data,"Temperature"); //load temperature
Liszt_load_field_as_position("Position"); //load position
Liszt_execute("calculate_temperature");

double * calculated_temperature;
Liszt_read_field(calculated_temperature,"Temperature");
Summary

• Balance features, restrictions, and simplicity
  – Element-wise formulation
  – Mesh-based topological functions and storage
  – Inferred dependencies

• Future:
  – expand the language to new problems
  – interface with existing languages