Liszt

A Domain Specific Language for Building Portable Mesh-based PDE Solvers

Z DeVito, N Joubert, F Palacios, S Oakley, M Medina, M Barrientos, E Elsen, F Ham, A Aiken, K Duraisamy, E Darve, J Alonso, P Hanrahan
Liszt @ LANL

Niels Joubert, Zach DeVito
Crystal Lemire, Pat Hanrahan
See more of us this week:

Today: Niels (me) presenting Liszt overview/implementation

This afternoon: Pat talking about EDSLs and the Big Picture

Tuesday: We’re running a Hackathon!

Wednesday: Zach leading design and future directions discussion
Today: Peel back the layers
Confused?
Questions Encouraged!
Science demands serious performance
The Future is Now

LANL IBM Roadrunner
Tianhe-1A
ORNL Titan
Specialization leads to efficiency

Hybrid architectures for complex workloads
How Do We Program These Machines?
The old way?

Hardware Abstraction?
Domain Specific Languages and Compilers
DSLs: Compromising generality
Bigger picture, or Sneak Peek:

My talk: Specialization

Pat and Zach: Playing nice
I’m out of pictures...
Compiling to parallel computers

Find Parallelism

Expose Data Locality

Reason about Synchronization
Analyzing data dependency

“What data does this value depend on”

Find Parallelism

- Independent data can be computed in parallel

Expose Data Locality

- Partition based on dependency

Reason about Synchronization

- Don’t compute until the dependent values are known

Can’t be done by compilers in general!

- \( A[i] = B[f(i)] \) – must compute \( f(i) \) to find dependency
Trade off generality
EDSLs solve the dependency problem

Liszt provides domain specific language features to solve the dependency problem:

- Parallelism
- Data Locality
- Synchronization

For solving PDEs on meshes:

All data accesses can be framed in terms of the mesh
Features of high performance PDE solvers

Find Parallelism

- Data-parallelism on mesh elements

Expose Data Locality

- PDE Operators have local support
- Stencil captures exact region of support

Reason about Synchronization

- Iterative solvers
- Read old values to calculate new values
Liszt Language Features

Mesh Elements
- Vertex, Edge, Face, Cell

Sets
- cells(mesh), edges(mesh), faces(mesh), ...

Topological Relationships
- head(edge), vertices(cell), ...

Fields
- val vert_position = position(v)

Parallelism
- forall statements: for( f <- faces(cell) ) { ... }
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
}
Language Features for Parallelism

```haskell
for (e <- edges(mesh)) {
    ...
}
```

Data-parallel `for`-comprehension

- Calculations are independent
- No assumptions about how it is parallelized
- Freedom of underlying runtime implementation
Language Features for Locality

Automatically infer stencil (pattern of memory accesses at element)

Restrictions:

- Mesh elements only accessed through built-in topological functions;
  \texttt{cells(mesh)}, ...

- Variable assignments to topological elements and fields are immutable;
  \texttt{val v1 = head(e)}

- Data in Fields can only be accessed using mesh elements
  \texttt{JacobiStep(v1)}

- No recursive functions
Language Features for Synchronization

Phased usage of Fields

- Fields have field phase state
  - read-only, write-only, reduce-using-operator $\text{field}(el) [\text{op}] = \text{value}$
- 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. $\text{field}(el) += \text{value}$
- Introduce write dependencies between instances of for-comprehension
Architecture

One compiler, one set of domain-specific analyses

Build AST, analyze code, extract data-dependencies

Multiple code-generators for the compiler

Generates code for each type of platform we support

Multiple run-times

Single shared base (mesh loading, partitioning, etc.)

Separate runtime API for each platform:

- pthreads-specific
- CUDA-specific
- MPI-specific
How do we infer data accesses from Liszt?

“Stencil” of a piece of code:

Captures just the memory accesses it performs

Infer stencil for each for-comprehension in Liszt

Targetting Many Architectures
Domain Specific Transform:
Inferring Stencils through Stencil Detection

Analyze code to detect memory access stencil of each top-level for-all comprehension

- Extract nested mesh element reads
- Extract field operations
- Difficult with a traditional library

```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
}
```
Domain Specific Transform: Inferring Stencils through Stencil Detection

\[ S(e_l, E) = (R, W) \]  
Read/Write Sets

\( e_l \)  
Expression

\( E \)  
Environment mapping free variables to values

\[(e_l, E) \]

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
}

\[(R, W)\]
Domain Specific Transform: Inferring Stencils through Stencil Detection

Problem: Don’t know the mesh at compile time!

Break up inference into:

**Static part**: Abstractly reason about operators.

- generate c++ code that queries mesh to build the specific stencil for each for-comprehension

**Dynamic part**: Concretely build data structures by analyzing mesh

- run c++ code on mesh

Anything that directly depends on mesh becomes generated code, executed at runtime.

This means: It is possible to write low-level c++ code that targets our back-end. But ugly and hard!
Implementing stencil detection

\[ S(e_l, E) = (R, W) \]

Finding stencil for each for-comprehension means capturing data access pattern. This poses a set of problems:

- Cannot guarantee termination
- Can be very expensive
- Must do before parallelization in single core

```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
}
```
Implementing stencil detection

**Abstract Interpretation**

“Partial execution of a computer program to gain insight into it’s semantics”. Allows us to calculate an approximate stencil.

Apply transformation $\tau$ to Liszt code

generate code with desirable properties (terminates, fast)
Implementing stencil detection

Abstract Interpretation

“Partial execution of a computer program to gain insight into it’s semantics”. Allows us to calculate an approximate stencil

Apply transformation $\mathcal{T}$ to Liszt code

generate code with desirable properties (terminates, fast)

$S(e_l, E) \subseteq \bar{S}(e_l, E) = (R, W)$

```java
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
}
```

```java
for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)
    Flux(v1) += _
    Flux(v2) -= _
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}
```
Implementing stencil detection

Abstract Interpretation

Defining $\mathcal{T}$

$\mathcal{T}( \text{if}(e_p) \ e_t \ \text{else} \ e_e) = \mathcal{T}(e_p); \mathcal{T}(e_t); \mathcal{T}(e_e);$

Conservatively evaluate if-statements

$\mathcal{T}( \text{while}(e_p) \ e_b) = \mathcal{T}(e_p); \mathcal{T}(e_b);$ 

Single Static Assignment of Mesh Variables

$\mathcal{T}( f(a_0,...,a_n)) = f'(\mathcal{T}(a_0),...,\mathcal{T}(a_n))$

Recursively apply to functions

Everything else, recursively apply to subexpressions of expression
In pictures
Domain Specific Transform:

Stencil Detection

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}
...

...
Domain Specific Transform:
Stencil Detection

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}
...

...
Domain Specific Transform: Stencil Detection

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}

...
Domain Specific Transform: Stencil Detection

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}

...
for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)

    Flux(v1) += _
    Flux(v2) -= _
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}
Domain Specific Transform: Stencil Detection

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}

...
Domain Specific Transform: Stencil Detection

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}

...
Domain Specific Transform: Stencil Detection

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}
...

Flux(v1) +-
Flux(v2) -_
JacobiStep(v1) +-
JacobiStep(v2) +_

e in edges(mesh)

head(e) tail(e)
Read Position, Temperature
Write Flux

Read Position, Temperature

H F E C B D G 1 5 8 10 1 7 3 0 2 4 9 A
Domain Specific Transform: Stencil Detection

for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)

    Flux(v1) += _
    Flux(v2) -=_
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}
...

Diagrams of mesh and stencil operations.
Domain Specific Transform:
Stencil Detection

for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)

    Flux(v1) += _
    Flux(v2) -= _
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}
...

Read Position, Temperature
Write Flux, JacobiStep
Write Flux
Domain Specific Transform:
**Stencil Detection**

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}

...
So how do we execute your code?
Execution Strategies

Partitioning

- Assign partition to each computational unit
- Use ghost elements to coordinate cross-boundary communication.
- Ideal for single computational unit per memory space
Execution Strategies

Partitioning
- Assign partition to each computational unit
- Use *ghost* elements to coordinate cross-boundary communication.
- Ideal for single computational unit per memory space

Coloring
- Calculate interference between work items on domain
- Schedule work-items into non-interfering batches
- Ideal for many computational units per memory space
Implement Strategies using Abstract Interpretation

Track 1: Apply transformation $\mathcal{T}$ to Liszt code

Use result to generate architecture-specific datastructures

Track 2: Rewrite code for specific parallel implementation

Use generated data structures for parallel computation

for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)
    Flux(v1) += _
    Flux(v2) -= _
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}
1. Partition Mesh (ParMETIS, G. Karypis)
MPI: Partitioning with Ghosts

2. Find used mesh elements and field entries using stencil data and duplicate locally into “ghost” elements

Implementation directly depends on algorithm’s access patterns
3. Annotate for-comprehensions with field preparation statements

```java
Flux.ensureState<LISZT_SUM>();
JacobiStep.ensureState<LISZT_SUM>();
Position.ensureState<LISZT_READ>();
Temperature.ensureState<LISZT_READ>();
for (e <- edges(mesh)) {
    val dP = Position(v1) - Position(v2)
    ...
    Flux(v1) += dT*step
    JacobiStep(v1) += step
}
Temperature.ensureState<LISZT_SUM>();
Flux.ensureState<LISZT_READ>();
JacobiStep.ensureState<LISZT_READ>();
for (p <- vertices(mesh)) {
    Temperature(p) += 0.01f * Flux(p)/JacobiStep(p)
}
...
MPI: Partitioning with Ghosts

4. MPI communication is batched during for-comprehensions and only transferred when necessary.
Applying Program Analysis: Results

```plaintext
for(f <- faces(mesh)) {
    rhoOutside(f) :=
        calc_flux( f,rho(outside(f) ) )
    + calc_flux( f,rho(inside(f) ) )
}
```
for (f faces(mesh)) {
    rhoOutside(f) := calc_flux(f, rho(outside(f))) + calc_flux(f, rho(inside(f)))
}

Ghost Cells
Applying Program Analysis: Results

for (f <- faces(mesh)) {
  rhoOutside(f) := calc_flux(f, rho(outside(f))) + calc_flux(f, rho(inside(f)))
}

Ghost Cells
GPU: Schedule threads with coloring

- Shared Memory
- Field updates need to be atomic
- Concerns about MPI approach – volume vs surface area

Build a graph of interfering writes:

```
\begin{array}{c}
\text{head(e)} \\
\text{tail(e)} \\
\end{array}
```

```
\begin{array}{c}
\text{e in edges(mesh)} \\
\text{vertices(mesh)} \\
\end{array}
```

- Read Position, Temperature
- Write Flux, JacobiStep
- Read/Write Flux
- Read/Write JacobiStep
- Write Temperature
GPU: Schedule threads with coloring

Compile time: Generate code to create field write structure

```
FORALL_SET(e, edges(mesh))
  Vertex v_1 = head(e);
  igraph->addEdge(thread(e).ID(), v_1.ID(), 18886);
  Vertex v_2 = tail(e);
  igraph->addEdge(thread(e).ID(), v_2.ID(), 18886);
ENDSET
```

Runtime: Build this structure using mesh and stencil
GPU: Schedule threads with coloring

Threads 1 edge assigned to 1 thread

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
</table>

Force Field:

A B C D E F G H

Memory
GPU: Schedule threads with coloring

Threads 1 edge assigned to 1 thread

Force Field:

Memory

A  B  C  D  E  F  G  H

0  1  2  3  4  5  6  7  8  9  10  11
Convert each for-comprehension into a GPU kernel, launched multiple times for sets of non-interfering iterations.

```c
__global__ for_01(ColorBatch batch, Force, Position, Velocity, Maxforce) {
  val dT = Position(v1) - Position(v2)
  ...
  Flux(v1) += dT*step
  Flux(v2) -= springForce
  ...
}
WorkgroupLauncher launcher = WorkgroupLauncher_forWorkgroup(001);
ColorBatch colorBatch;
while(launcher.nextBatch(&colorBatch)) {
  Maxforce.ensureSize(colorBatch.kernel_size());
  GlobalContext_copyToGPU();
  for_01<<<batch.blocks(),batch.threads()>>>(
    batch, Force, Position,
    Velocity, Maxforce);
}
```

Schedule

- Set of nonconflicting threads per color

```
Batch 1  Batch 2  Batch 3  Batch 4
1  3  8  11  0  5  7  10  4  9  2
```
How fast do we go?
Results

4 example codes with Liszt and C++ implementations:

- Euler solver from Joe
- Navier-Stokes solver from Joe
- Shallow Water simulator
  - Free-surface simulation on globe as per Drake et al.
  - Second order accurate spatial scheme
- Linear FEM
  - Hexahedral mesh
  - Trilinear basis functions with support at vertices
  - CG solver
Real applications: Joe in Liszt

- Navier-Stokes and Euler equations (unstructured solver).
- Explicit time integration using local time stepping.
- Second order upwind scheme via variable extrapolation using gradient calculation.
- Convective terms evaluated using a HHLC approximate nonlinear Riemann solver.
- Fully-viscous approximation of the viscous stresses.
- Gradient calculation using a Green-Gauss scheme.
- New boundary conditions for adiabatic solid walls.
Real applications
Scalar Performance Comparisons

Runtime comparisons between hand-tuned C++ and Liszt
Liszt performance within 12% of C++

<table>
<thead>
<tr>
<th></th>
<th>Euler</th>
<th>Navier-Stokes</th>
<th>FEM</th>
<th>Shallow Water</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh size</td>
<td>367k</td>
<td>668k</td>
<td>216k</td>
<td>327k</td>
</tr>
<tr>
<td>Liszt</td>
<td>0.37s</td>
<td>1.31s</td>
<td>0.22s</td>
<td>3.30s</td>
</tr>
<tr>
<td>C++</td>
<td>0.39s</td>
<td>1.55s</td>
<td>0.19s</td>
<td>3.34s</td>
</tr>
</tbody>
</table>
MPI Performance

4-socket 6-core 2.66Ghz Xeon CPU per node (24 cores), 16GB RAM per node. 256 nodes, 8 cores per node.

---

**Euler**

- 23M cell mesh
- Liszt
- C++

**Navier-Stokes**

- 21M cell mesh
- Liszt
- C++
GPU Performance

Tesla C2050, Double Precision vs single core, Nehalem E5520 2.26Ghz
Portability

Tested both pthreads (coloring) and MPI (partitioning) runtime on:

- 8-core Nehalem E5520 2.26Ghz, 8GB RAM
- 32-core Nehalem-EX X7560 2.26GHz, 128GB RAM

Comparison between Liszt runtimes
Vectorization (in progress)

Consider n-wide set of elements at once

Use SSE/AVX/NEON vector lanes

Serialize only conflicting writes

Preliminary results: 30% speedup
Aubrey Gress, masters student:

Use a two-level staged coloring approach.

- Arrange data in global memory for ideal coalescing
- Stage data into shared memory, color shared memory

Preliminary results: another ~30% speedup
Aside: Improved debugging!
Aside: Improved debugging: visualization
Aside: Improved debugging: inspection
Future work – so much cool stuff

This talk:

- Low-level implementation flexibility
- Deeper in-situ visualization and performance analysis
- More runtimes, improved back-end runtime interface
- More applications

Other talks:

- Language extensions for FEM, sparse matrices
- Evolving to become a library (we have some tricks)
- Embedding Liszt directly into another language
Future work: flexibility allows optimizations

Multiple mesh representations and mesh specialization

- We don’t change mesh layout or assume a type of mesh
- But we can optimize for this!
  
  - Automatically specialize for grids, structured meshes, etc.
  
- Multiple representations of the same mesh for performance?

Data layout optimizations

- We can automatically do AoS-SoA transformations
- Modify data layout dynamically
- Smart staging and streaming of data (already partially for CUDA)
Future work: more debugging and analysis

In-situ LisztVis-like analysis

Cluster-wide visualization of simulations, in-situ

Interact with simulation as it progresses

eg. Track specific data representations as it cascades

Visual performance analysis

We can deeply instrument specific parts of your code, so…

Present caching behaviour and interaction of code sections

Analyze how you use instruction set, bottleneck instructions

Do it visually.
Future work: runtimes!

Fast GPUs

Staged, hierarchical GPU runtime.

There’s another 30% on the table (Aubrey Gress)

Hierarchical runtimes

MPI+GPU

More exotic things?

Fully Heterogeneous

We can combine approaches, MPI+GPU+SMP
Future work: runtimes!

Coherent back-end interface

Originally the Liszt back-end was for application programmes (!)

We’re converging to a new design:

Single compiler, single code-generator, targetting a single API

Stream Programming-like interface*

Multiple back-ends for multiple execution strategies

Easier and cleaner to add more runtimes

* Gummarahu and Rosenblum. “Stream Programming on General-Purpose Processors” 2005, MICRO 38
Current/Future work: applications

More applications makes for a better language.

Hello LANL!

* Gummarahu and Rosenblum. “Stream Programming on General-Purpose Processors” 2005, MICRO 38
That's all Folks!
Interested in more?

To appear in Supercomputing 2011

- “Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers”, submitted to Supercomputing 2011

Visit our website, it’s all there.

- http://liszt.stanford.edu

Talk to us this week!

- Drop by, we’re in building 200 (training room)

Other talks

- Pat at 1.30pm today
- Zach at 10am Wednesday