Liszt
Programming Mesh-based PDEs on Heterogeneous Parallel Platforms

Z. DeVito, N. Joubert, M. Medina, M. Barrientos, S. Oakley, J. Alonso, E. Darve, F. Ham, P. Hanrahan
Era of Power-Constrained Computing

Mobile devices
- Battery life

Supercomputers
- Total energy requirement
Exascale: 100:1 Improvement Needed

Source: DARPA Exascale Hardware and Software Studies
Special Hardware 100x More Efficient

Anton

Molecular dynamics computer

D. E. Shaw et al. Supercomputer 2009, Best Paper and Gordon Bell Prize
### GPUs 10x More Efficient

20 times greater throughput for same area and power
½ the sequential performance

<table>
<thead>
<tr>
<th></th>
<th># CPU cores</th>
<th>Instructions per issue</th>
<th>VPU lanes per core</th>
<th>L2 cache size</th>
<th>Single-stream</th>
<th>Vector throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 out of order</td>
<td>4 per clock</td>
<td>4-wide SSE</td>
<td>4 MB</td>
<td>4 per clock</td>
<td>8 per clock</td>
</tr>
<tr>
<td></td>
<td>10 in-order</td>
<td>2 per clock</td>
<td>16-wide</td>
<td>4 MB</td>
<td>2 per clock</td>
<td>160 per clock</td>
</tr>
</tbody>
</table>

Specialization Leads to Efficiency
Apple A4 Processor

Contains CPU and GPU and …
Specialization Leads to Heterogeneity
Complex Heterogeneous Platforms

LANL IBM Roadrunner
    (Opteron + Cell)

ORNL Cray 20 PFLOPs
    (Opteron + Fermi)

No agreed upon programming paradigm
Multiple Parallel Platforms

Cluster
- Distributed memory over system area network

Many-core GPU (e.g. Cell, Fermi)
- SIMD / SIMT architecture
- Local memory on chip / Separate GPU memory
- Accelerator connected via PCI-E

Multi-core SMP (e.g. 32 core, 4-socket systems)
- Multi-threaded
- Wider vector units
- Complex memory hierarchy, consistency protocols, …
Multiple Parallel Programming Models

Cluster
- MPI

Multi-core SMP (e.g. 32 core, 4-socket systems)
- Threads/locks (pthreads)
- OpenMP
- Thread building blocks, Grand Central
- Transactional memory, …

Many-core GPU (e.g. Cell, Fermi)
- CUDA, OpenCL, Compute Shader
- Ct, Copperhead, …
Should Scientists
Deal with this Complexity?
Can We Buffer
Innovative Software-Hardware Designs from the Demands of Applications?
Possible Solution:

Domain-Specific Frameworks and Languages
Definition: Domain-Specific

Definition: A language or library that exploits domain knowledge for **productivity** and **efficiency**

Widely used in many application areas

- matlab / R
- SQL / map-reduce / Microsoft’s LINQ
- OpenGL/D3D and Cg/HLSL
- …
Four Advantages

Productivity

- Separate computational science expertise from computer science expertise

Portability

- Run on wide range of platforms

Performance

- Super-optimize using a combination of domain knowledge and platform knowledge

Innovation

- Allows vendors to change architecture and programming models in revolutionary ways
Liszt DSL

Domain Specific Language

SC.scala

Compiler
applying Domain Transformations

Domain Knowledge

Architecture Knowledge

Massive MPI Performance

Huge GPU Speedups

31.5x

Visual Debugging

Future Platforms

Presented by N. Joubert
Status

Applications
- Joe (Forward & Backward Euler), Turbulence & Viscosity (in progress)
- Static 1st-order FEM

Automatic Optimizations
- Automatic partitioning
- Automatic ghost cell discovery
- Memory access stencil analysis
- Coloring, static thread scheduling

Runtimes
- MPI
  - Parallel Mesh Loading
- GPU
- Naïve OpenMP (fine grain locking)
Using Domain Knowledge

Knowledge about topological relationships on the mesh are built into the compiler

We can use program analysis to

■ Perform domain decomposition
■ Transform code to machine-specific implementations
■ Optimize data layout for caches and vector machines
■ Choose between conflicting approaches to parallelize code depending on the specific target architecture
Designed for Productivity

Built-in Topology
- Mesh, Topological Elements
- Topological Sets, Topological Relationships
- Sets (unordered) and lists (ordered)
- Short vectors

Fields attached to topological elements
- Fields: val vert_position = position(v)

Set iteration
- forall statements: for( f <- faces(cell) ) { ... }

Solvers
- Sparse matrix solver interface
Fluid Flow (Joe) in Liszt

```scala
def main() {
  val pos = new Field[Vertex, Vector3]
  val A = new SparseMatrix[Vertex, Vertex]

  for (c <- cells(mesh)) {
    val center = avg(pos(vertices(c)))
    for (f <- faces(c)) {
      val face_dx = avg(pos(vertices(f)) - center)
      for (e <- edgesCCW(f)) {
        val v0 = tail(e)
        val v1 = head(e)
        val v0_dx = pos(v0) - center
        val v1_dx = pos(v1) - center
        val face_normal = v0_dx cross v1_dx
        // calculate flux for face ...
        A(v0, v1) += ...
        A(v1, v0) -= ...
      }
    }
  }
}
```
Simple Liszt Code Example

```scala
for (edge <- edges(mesh)) {
  val flux = flux_calc(edge)
  val v0 = head(edge)
  val v1 = tail(edge)
  Flux(v0) += flux
  Flux(v1) -= flux
}
```

Note: Cannot naively parallelize this code due to write conflicts
Designed for Parallel Performance

for loops equivalent to maps

- Independent iterations over topological sets:
  ```
  for( f <- faces(cell) ) { … }
  ```
- Easy to parallelize, leads programmer to write parallelizable code

Mesh neighborhood accessed through built-in functions

- Pattern of access defines stencil, Stencil can be statically analyzed.
- Mesh variables cannot be reassigned.

Fields must be consistently accessed during loops

- Field accesses are analyzable (no aliases)
- Either read, write or reduce
Parallel Field Usage

Fields change state throughout the program – read, write, reduce. A field **cannot** change state during a for comprehension.

- **Parallel for loops:**
  - Liszt prevents creating an undefined value in a field by limiting the type of access per loop.
  - Allow platform specific actions to produce correct final values
    - Deferred computation & communication
    - Overlap communication with computation
Parallel Fields Example

A very simple mass-spring forward Euler simulation

```cpp
for (int vi = 0; vi < vertices.size(); vi++) {
    forces[vi] = calculate_forces_on(vi);
}
for (int vi = 0; vi < vertices.size(); vi++) {
    double delV = delta_t * forces[vi] / mass[vi];
    double delX = delta_t * velocity[vi];
    velocity[vi] += delV;
    position[vi] += delX;
}
```

```liszt
for (v <- vertices(mesh)) {
    forces(v) = calculate_forces_on(v)
}
for (v <- vertices(mesh)) {
    val delX = delta_t * velocity(v);
    position(v) += delX;
}
for (v <- vertices(mesh)) {
    val delV = delta_t * forces(v) / mass(v);
    velocity(v) += delV;
}
```
Static Mesh Variables

We only allow single static assignment of mesh variables.

- Code cannot access mesh topology arbitrarily far long the mesh without explicitly accessing it.
- We can analyze code to exactly know what mesh elements it uses.
- No recursion on mesh variables

Compiler Error:

```plaintext
for (v <- vertices(mesh)) {
    while (t < 1.0) {
        verts = vertices(v);
        t += 0.1
    }
}
```

Only allow Single Static Assignment of mesh topology variables
Architecture

- DSL Code
- Scala Compiler: Build AST
- Liszt JIT
  - DSL Transformations
  - Platform Transformations: MPI Codegen, SMP Codegen, GPU Codegen
- Platform-Specific Compilation: MPI-CC, Viz-CC, SMP-CC, GPU-CC, NVCC
- Runtime: Mesh Analysis, Run
- RTP: mpirun, cocoa, threads, ptx
Transformations on the AST

- **Analyze shape of memory access stencil**
  - Topological sets accessed through built-in functions
  - Mesh elements accessed by for comprehension
  - Field entries read & written through mesh elements

- **Analyze dependencies through fields**
  - Fields depend on other fields and scalars

- **Detect variables used as scalar reductions**
  - Defining a variable in a serial block, writing to a variable inside a for-comprehension (parallel block).
  - More behind the scenes
Transformations on the AST

- **Analyze shape of memory access stencil**
  - MPI: Partition the mesh & create ghost cells automatically
  - GPU: Assign iterations to threads

- **Analyze dependencies through fields**
  - MPI: Synchronize and communicate ghost cells
  - GPU: Create static thread schedule

- **Detect variables used as scalar reductions**
  - MPI: Inter-node variable synchronization
  - GPU: Create field with on-demand GPU reductions
Domain Specific Transform: Stencil Detection

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

- Extract nested mesh element reads
- Extract field operations
- Cannot do this with a traditional library

```scala
for(edge <- edges(mesh)) {
  val flux = flux_calc(edge)
  val v0 = head(edge)
  val v1 = tail(edge)
  Flux(v0) += flux
  Flux(v1) -= flux
}
```

```scala
FOR_SET(edge, edges(mesh)) {
  const Vertex v0 = HEAD(edge)
  const Vertex v1 = TAIL(edge)
  FIELDOP(Flux, v0, PLUSEQ)
  FIELDOP(Flux, v1, PLUSEQ)
}
```
Domain Specific Transform:

Stencil Detection

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

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

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

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

```plaintext
for(f <- faces(mesh)) {
    rhoOutside(f) :=
        calc_flux(f, rho(outside(f)))
    + calc_flux(f, rho(inside(f)))
}
```
Cross-partition writes are stored locally in "ghosts," aggregated locally on a node, and combined with other nodes during communication step.

Implementation directly depends on algorithm’s access patterns.

```scala
Flux.prepare_for_write()
for(edge <- edges(mesh)) {
    val flux = flux_calc(edge)
    val v0 = head(edge)
    val v1 = tail(edge)
    Flux(v0) += flux
    Flux(v1) -= flux
}
Flux.finish_write()
```
**MPI: Automatic Ghost Cell Creation**

Using the detected memory access stencil:

- Partition Mesh (ParMETIS)
- Find mesh elements used by stencils and duplicate elements locally
- Set up local field storage and ghost field entries according to stencil’s reachable field entries.
- Annotate for comprehensions with field preparation and finalization statements – handles state of fields (as per language restrictions) and MPI communication of ghost reads and writes
Applying Program Analysis: Results

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

Initial Partition (ParMETIS)
```
Applying Program Analysis: Results
Applying Program Analysis: Results
GPU: Atomic Updates Expensive

- Field updates need to be atomic
- Requires expensive locks
- MPI approach doesn’t work – volume vs surface area

```scala
for(edge <- edges(mesh)) {
    val flux = flux_calc(edge)
    val v0 = head(edge)
    val v1 = tail(edge)
    Flux(v0) += flux
    Flux(v1) -= flux
}
```
GPU: Scheduling Eliminates Conflicts

- Combine stencil with assigning threads to for-loop iterations
- Statically create field write pattern at compile time
- Build graph of conflicts between field writes of threads

```scala
for(edge <- edges(mesh)) {
  val flux = flux_calc(edge)
  val v0 = head(edge)
  val v1 = tail(edge)
  Flux(v0) += flux
  Flux(v1) -= flux
}
```
GPU: Scheduling using Coloring

- Automatically color iterations depending on mesh access
- Generates static schedule of GPU threads – run one color at a time
Evaluation

- Ported early version of Joe to Liszt
- Spatial scheme: Forward Euler
- Mesh size: 750k cells

**MPI results:**
- Certainty Cluster – 4-socket 6-core 2.66Ghz Xeon CPU per node (24 cores), 36GB RAM per node.

**GPU Results:**
- Tesla C2050 (Fermi-based) GPU – 16 SMPs, 3GB GDDR5 RAM
MPI Performance

12 nodes, 8 cores per node

MPI Speedup 750k Mesh

MPI Wall-Clock Runtime
GPU Performance

Scaling mesh size from 50k (unit-sized) cells to 750k (16x) on a Tesla C2050.

Comparison is against single threaded runtime on host CPU (Core 2 Quad 2.66Ghz)

![Graph showing GPU Speedup over Single-Core](image)

Single-Precision: **31.5x**, Double-precision: **28x**
Liszt Hackathon

Invited 12 people to spend half a day writing a Liszt program.

- Made available on http://hackliszt.com/

From never-seen-before until working applications in 6 hours:

- Mesh Smoothing algorithm
- 1st order dynamic FEM deformable body simulation
- 1st order Finite Difference Fluid Simulation
- ...

External hackathon coming up.
Making a Library Act Like a Language
Criticisms of Domain-Specific Languages

Separate system with unique syntax
  ■ Do I have to learn another language?

Usually a poor language designed by amateurs
  ■ Why isn’t it more general?

Difficult to integrate with other libraries/languages
  ■ How do DSLs and non-DSLs interoperate?

No development environment
  ■ Where is the debugger? Performance analysis tools?

Expensive to build complete compiler
Frameworks and Libraries

trait TestMatrix {

    def example(a: Matrix, b: Matrix, c: Matrix, d: Matrix) = {
        val x = a*b + a*c
        val y = a*c + a*d
        return x+y
    }

}

Objects / types define the domain
Object methods define the semantics of the domain
Looks like a language - “Embedded DSL”
Making a Library act like a Language

Act like a language

   Clean syntax

Act like a compiler

   Check that the objects are being used correctly
   Rewrite sequence of calls into the more efficient sequence
   Generate optimized low-level code for different platforms
Clean Syntax

Operator overloading

```scala
var a, b, c : Matrix
val x = a*b + a*c
```

Allow all characters to be operators

```scala
def #(b: Matrix) : Int
```

Free-form syntax : a b c d e = a.b(c).d(e)

```scala
def outer(b : Vector) : Matrix
var a, b : Vector
val x = a outer b
```
Analyzing and Optimizing Code

val x = a*b + a*c
val y = a*c + a*d
val y = a * (b + c + d + e)
“Quoting” or “Lifting” Code

\[
a = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}
b = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}
a*b
\]

\[
a = \text{Literal}\left(\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}\right)
b = \text{Literal}\left(\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}\right)
a * b = \text{Mult}(, )
\]

Matrix  Expression Tree of Matrix
Polymorphic Embedding

```scala
trait MatrixProgram with MatrixLanguage {
  def example(a: AbstractMatrix, b: AbstractMatrix,
              c: AbstractMatrix, d: AbstractMatrix) = {
    val x = a*b + a*c
    val y = a*c + a*d
    return x+y
  }
}
```

The trait `MatrixLanguage` defines the DSL (types and operators)
The type `AbstractMatrix` is an abstract type (no implementation) for matrices
Different concrete (specific) implementations

- `AbstractMatrix => Matrix` // Actual matrix class
- `AbstractMatrix => Exp[Matrix]` // Expression tree for matrix operations
Lifting Any Program

Quoting the code to generate abstract syntax tree

Special quote (`) operator

Scala plug-in

Overload all special syntactical forms

if, then, else

for, while, repeat, ..

val, var, def

lambda

==?

class?
Multi-Stage Polymorphic Embedding

Stages triggered by mixing in traits (with X)

Optimizations performed by mixing in optimizer traits
  - MatrixProgram with MatrixLanguage with AlgebraicOpt

Machines have their own implementation of types
  - with GPU
  - with SMP
  - with MPI

Compiler modules are also traits
  - Generate Scala code and then compile and dynamically link
    - with Scala
  - Generate C++/CUDA code and run
    - with CUDA
Scala: Scalable Language

Designed to embed DSLs
  - Concise syntax, implicit type conversions, …

Functional programming
  - Higher-order functions, lambdas, closures and continuations
  - Encourages the use of immutable data structures
  - Discourages programs with side-effects

Object-oriented programming
  - Allows mutable data structures
  - Strong type system, parameterized types, traits

Support for concurrency
Progress since last Fall AST Meeting

Joe

- Forward/Backward Euler
- Turbulence (in progress)

Automatic Optimizations

- Memory access stencil analysis
- Coloring, static thread scheduling

Metaprogramming

- Language virtualization & multistage polymorphic embedding

Runtimes

- Scalable MPI
- GPU back-end
Goals for Next Year

Applications
- Turbulence & Viscosity
- Combustion
- Complete Joe simulations

Language
- Improve implicit method support

Performance of solvers
- Improve sparse matrices on MPI
- Sparse matrices on GPUs

Runtimes
- MPI-GPU heterogeneous runtime
- Efficient SMP version for CPUs with extended vector instruction sets
Expanding the Effort

Participated in two Exascale Co-Design Center Proposals (Materials and Combustion)

DSL for exascale: Proposal with LANL, EPFL and Stanford to XSTACK
A. Mcpherson, P. McCormick, S. Pakin, LANL; M. Odersky, EPFL
Proposal with NIH Simbios Center: Core 1 - DSLs for biological problems
R. Altman, S. Delp, V. Pande, …

Active collaboration with Pervasive Parallelism Lab
Funded by Sun, AMD, NVIDIA, IBM, Intel, NEC, HP
LANL representative attended last two PPL meetings

Visited DOE labs: LANL, LLNL, ORNL (scheduling)

Co-organized LACSS Workshop on Non-Traditional PL

First Liszt Code-a-thon/Hack-a-thon in May