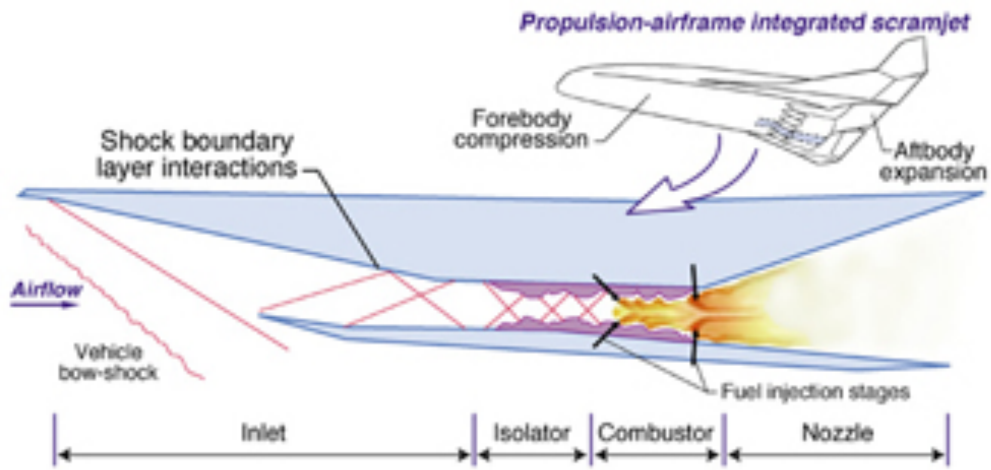
Designing DSLs

Zach DeVito

Intro



Propulsion-airframe integrated scramjet
Forebody compression
Aftbody expansion
Shock boundary layer interactions
Airflow
Vehicle bow-shock
Fuel injection stages
Inlet
Isolator
Combustor
Nozzle

Liszt

```
//Initialize data storage
val Position = FieldWithLabel[Vertex,Float3]("position")
val Temperature = FieldWithConst[Vertex,Float](0.f)
val Flux = FieldWithConst[Vertex,Float](0.f)
val JacobiStep = FieldWithConst[Vertex,Float](0.f)
//Set initial conditions
val Kq = 0.20f
for (v <- vertices(mesh)) {
   if (ID(v) == 1)
      Temperature(v) = 1000.0f
   else
      Temperature(v) = 0.0f
}
//Perform Jacobi iterative solve
var i = 0;
while (i < 1000) {
    for (e <- edges(mesh)) {
       val v1 = head(e)
       val v2 = tail(e)
       val dP = Position(v2) - Position(v1)
       val dT = Temperature(v2) - Temperature(v1)
       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
}
```

Problems

What set of applications should a DSL support?

How do you design a DSL for 3 P's?

- Portability
- Performance
- Productivity

Is there a domain-specific language for graphics?

OpenGL/DirectX

…a domain-specific language for graphics

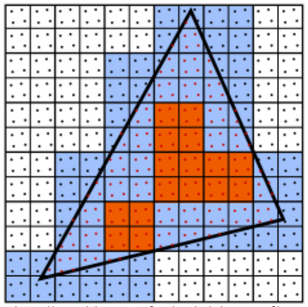
But what about geometry creation?

OpenGL/DirectX

…a domain-specific language for solving the light-transport equations

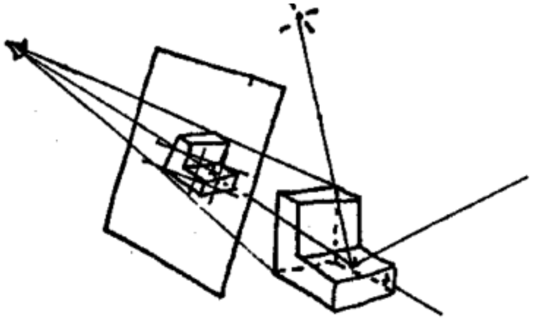
Light Transport?

Rasterization

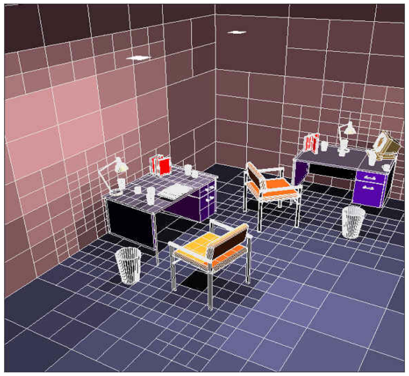


http://graphics.stanford.edu/~kayvonf/papers/kayvonf_dissertation.pdf

Monte-carlo Ray-tracing



Radiosity



http://www.cs.duke.edu/courses/fall02/cps124/
notes/08_rendering/

OpenGL/DirectX

…a domain-specific language for approximating the light-transport equations via rasterization

OpenGL/DirectX

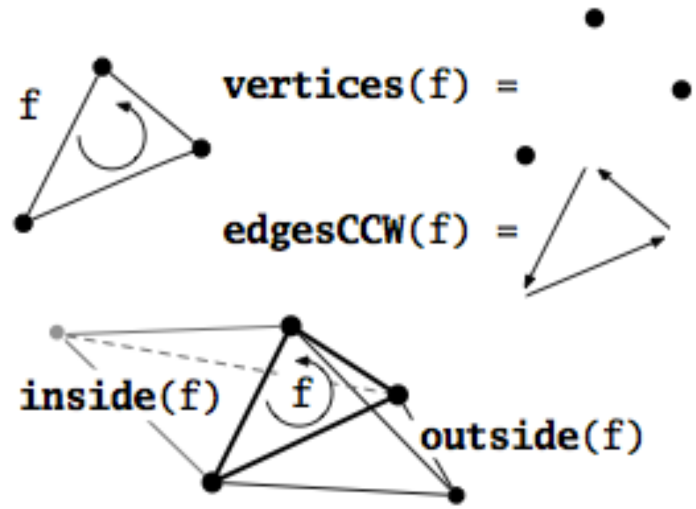
…a domain-specific language for
approximating the light-transport equations via rasterization … but not full REYES-style tessellation

Is there a domain-specific language for graphics?

…yes, but only for a subset of applications that have a very similar style

Designing DSL: Expertise

Domain Expertise



Finite Element Method

Dirichlet Boundary

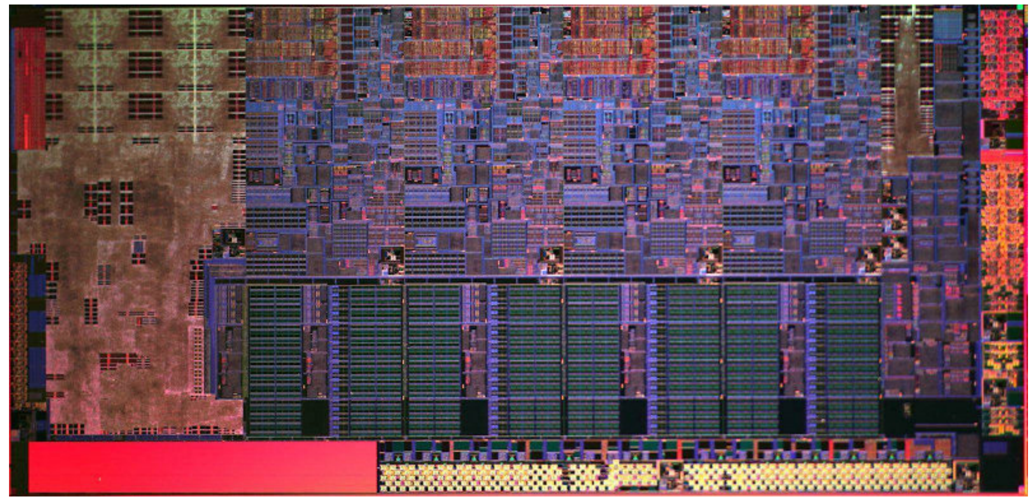
Basis Functions

Trilinear Basis

Discontinuous Galerkin

Mesh Element

Performance Expertise



Thread

False Sharing

Mutex

Coherency
Protocol

SSE

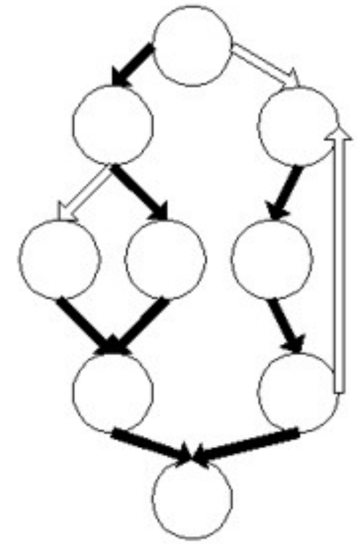
Synchronization

Bandwidth

TLB
Shootdown

Locality

Language Expertise



Control Flow
Graph

Abstract
Syntax Tree

Program
Transformation

Alias Analysis

Code
Generation

Loop-invariant
Code Motion

Domain Expert

- Domain
- Performance

Computer Scientist

- Language
- Performance

It’s rare to find all 3

…which leads to pitfalls in design

Pitfall: Domain

miss important aspects of the domain

Flux Calc

```
def calcEulerFlux_HLLC( rhoL : Float, uL : Vec[_3,Float], ...
    ...
    val unL = dot(uL,nVec) ;
    val uLuL = dot(uL,uL) ;
    val cL = sqrt( gammaL * pL / rhoL ) ;
    val hL = gammaL / ( gammaL - 1.f ) * pL / rhoL + 0.5f * ...
    val eL = hL * rhoL - pL ;
    ...
    val Rrho = sqrt( rhoR / rhoL ) ;
    val tmp = 1.f / ( 1.f + Rrho ) ;
    val velRoe = tmp * ( uL + uR *Rrho ) ;
    val uRoe = dot( velRoe, nVec ) ;
    val gamPdivRho = tmp * ( (gammaL * pL / rhoL + 0.5f * ...
    val cRoe = sqrt( gamPdivRho - ((gammaL + gammaR) * 0.5f - ...  *
    val sL = (uRoe - cRoe).min(unL - cL) ;
    val sR = (uRoe + cRoe).max(unR + cR) ;
    val sM = (pL - pR - rhoL * unL * (sL - unL) + rhoR * unR ...  (rh
    val pStar = rhoR * (unR - sR) * (unR - sM) + pR ;
    ...
```

But... 17 doubles / edge

```
val icv0 = outside(f)
val icv1 = inside(f)
val rho0 = rho(icv0)
val u0 = UgpWithCvCompFlow.vel(icv0)
val p0 = UgpWithCvCompFlow.press(icv0)
val h0 = UgpWithCvCompFlow.enthalpy(icv0)
val rho1 = rho(icv1)
val u1 = UgpWithCvCompFlow.vel(icv1)
val p1 = UgpWithCvCompFlow.press(icv1)
val h1 = UgpWithCvCompFlow.enthalpy(icv1)
var nVec = MeshGeometryCalc.fa_normal(f)
val area = sqrt(dot(nVec,nVec))
nVec /= area
val kine0 = UgpWithCvCompFlow.kine(icv0)
val kine1 = UgpWithCvCompFlow.kine(icv1)
val Frho5 = UgpWithCvCompFlow.calcEulerFlux_HLLC(rho0, u0, ...
```

Pitfall: Languages

language and library design looks like the target architecture

Single Core

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

MPI

```
for (e <- edges(mesh)) {
   val v1 = head(e)
   val v2 = tail(e)
   val dP = Position(v2) - Position(v1)
   val dT = Temperature(v2) - Temperature(v1)
   val step = 1.0f/(length(dP))
   Flux(v1) += dT*step
   Flux(v2) -= dT*step
   JacobiStep(v1) += step
   JacobiStep(v2) += step
}
update(Flux)
update(JacobiStep)
for (p <- vertices(mesh)) {
   Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
}
```

CUDA

```
def fluxCalc(e : Edge) {
val v1 = head(e)
val v2 = tail(e)
val dP = Position(v2) - Position(v1)
val dT = Temperature(v2) - Temperature(v1)
val step = 1.0f/(length(dP))
Flux(v1) += dT*step
Flux(v2) -= dT*step
JacobiStep(v1) += step
JacobiStep(v2) += step
}
def updateTemp(p : Vertex) {
Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
}

launch(fluxCalc,edges(mesh))
launch(updateTemp,vertices(mesh))
```

Pitfall: Performance

create operations that are hard to implement efficiently

Sparse Matrices

```
for(c <- cells(mesh)) {
  for(v <- vertices(c)) {
for(v2 <- vertices(c)) {
   val d : Float3 = calcMatValue(v,v2)
   M(3*ID(v),3*ID(v2)) += d(0)
   M(3*ID(v) + 1,3*ID(v2) + 1) += d(1)
   M(3*ID(v) + 2,3*ID(v2) + 2) += d(2)
}
  }
}
```

Locality?

Delite

Language

- Lightweight modular staging
- Libraries for IR manipulation

Performance

- Program to abstracted parallel paradigms

Still need to design the language to make performance possible

Domain Generalization

An approach for designing DSLs

Contrast: Hardware Abstraction

Most parallel programming environments abstract some aspects of hardware

- Infiniband → MPI → Actors
- SIMD → CUDA/ArBB
- Multi-core → Threads → OpenMP/Cilk

For DSLs, we want to abstract the aspects of the domain instead of a specific architecture

Start with Examples

Collect a series of representative examples from your domain

- Look for breadth across domain
- Look for common patterns
- Parallel implementations

Your DSL will not fit the entire domain, instead it should be tailored to express your examples well

Why Examples?

- Makes the domain more concrete to non-domain experts
- Determines what features the language should support
- Parallel implementations show how to generate efficient code
- Overlap between examples shows what should be handled by the language

Language Features

Efficient applications can

- Find parallelism
- Expose locality
- Reason about synchronization

A DSL's Language features should enable the compiler to perform this automatically

Language Features: Liszt

Parallelism

- Parallel for-comprehensions

Locality

- Automatically extract a local stencil of a 3D mesh

Synchronization

- Limit data-dependencies by restricting access to fields through program phases

Design Tradeoffs

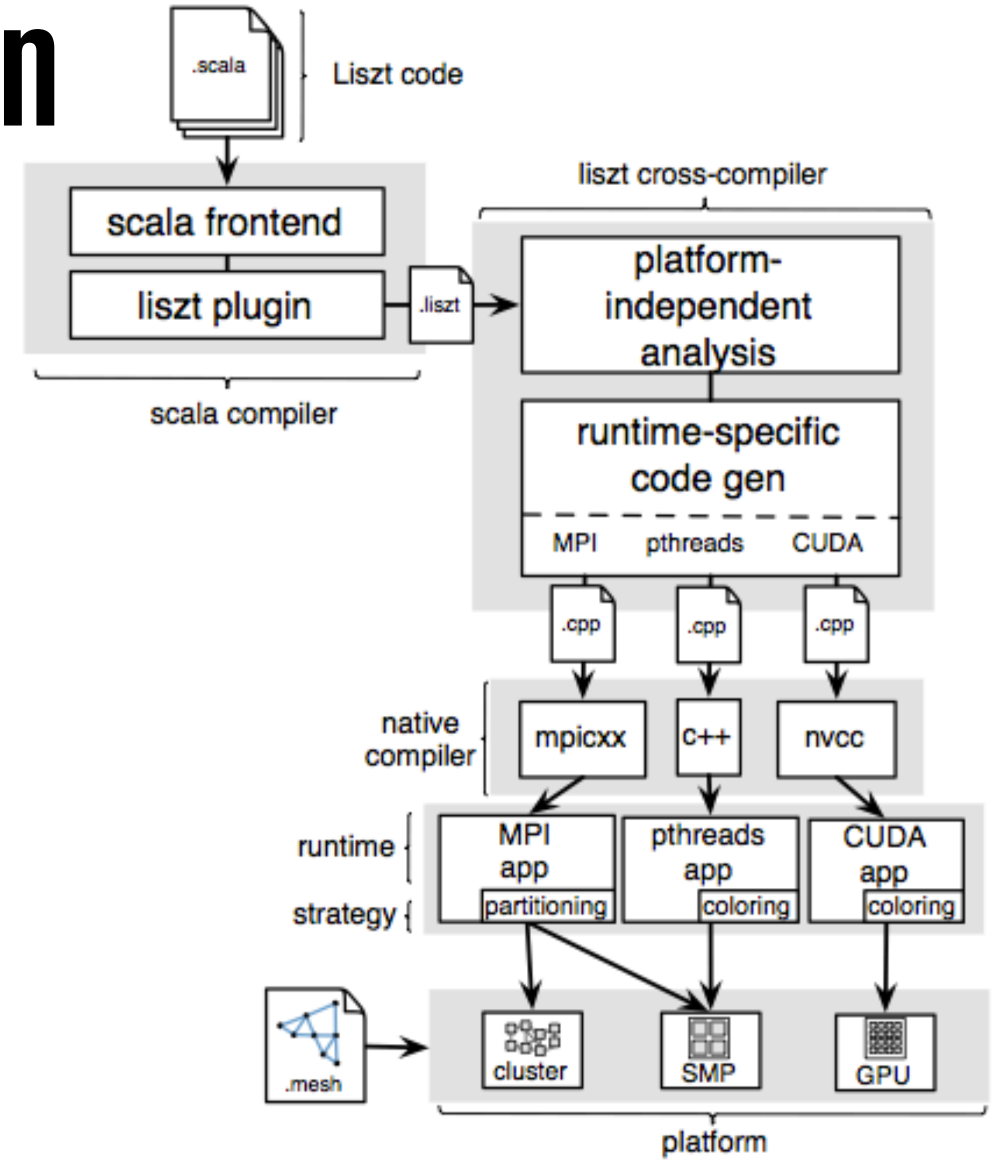
What should be a build-in construct? a library written in the DSL?

- Built-in constructs allow the compiler to reason about their semantics, possibly provide a better implementation
- Built-in constructs often need per-platform implementation

Liszt: Built-in Mesh Operators

```
def vertices(e : Mesh) : Set[Vertex]
def vertices(e : Vertex) : Set[Vertex]
def vertices(e : Edge) : Set[Vertex]
def vertices(e : Face) : Set[Vertex]
def vertices(e : Cell) : Set[Vertex]

def verticesCCW(e : Face) : Set[Vertex]
def verticesCW(e : Face) : Set[Vertex]

def cells(e : Mesh) : Set[Cell]
def cells(e : Vertex) : Set[Cell]
def cells(e : Edge) : Set[Cell]
def cells(e : Face) : Set[Cell]
def cells(e : Cell) : Set[Cell]

def cellsCCW(e : Edge) : Set[Cell]
def cellsCW(e : Edge) : Set[Cell]

def edges(e : Mesh) : Set[Edge]
def edges(e : Vertex) : Set[Edge]
def edges(e : Face) : Set[Edge]
def edges(e : Cell) : Set[Edge]

def edgesCCW(e : Face) : Set[Edge]
def edgesCW(e : Face) : Set[Edge]

def faces(e : Mesh) : Set[Face]
def faces(e : Vertex) : Set[Face]
def faces(e : Edge) : Set[Face]
def faces(e : Cell) : Set[Face]

def facesCCW(e : Edge) : Set[Face]
def facesCW(e : Edge) : Set[Face]

def head(e : Edge) : Vertex
def tail(e : Edge) : Vertex

def inside(e : Face) : Cell
def outside(e : Face) : Cell

def flip(e : Edge) : Edge
def flip(e : Face) : Face
```

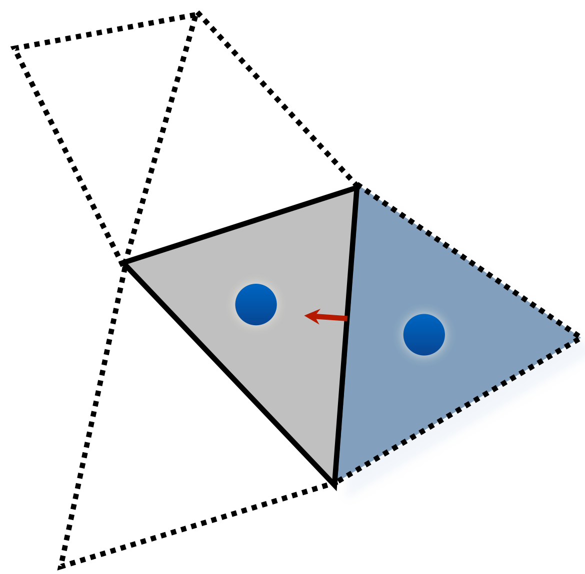
Implementation



Platform-independent

Using the domain-knowledge from the language design, find parallelism, locality, synchronization in a specific program

Liszt Stencil



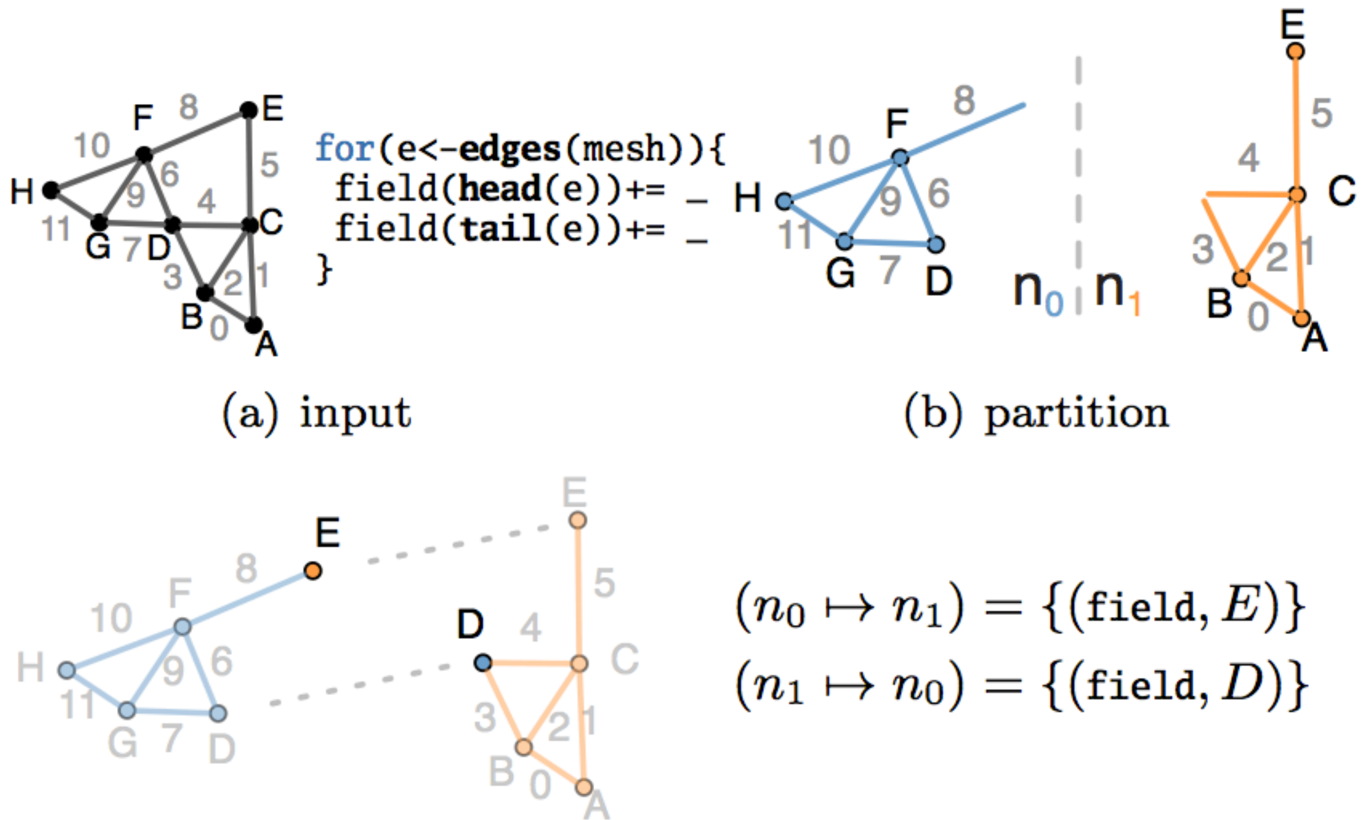
Liszt Phases

```
enterPhase(READ,Position); enterPhase(READ,Temperature);
enterPhase(+=,Flux); enterPhase(+=,JacobiStep);
for (e <- edges(mesh)) {
   val v1 = head(e)
   val v2 = tail(e)
   val dP = Position(v2) - Position(v1)
   val dT = Temperature(v2) - Temperature(v1)
   val step = 1.0f/(length(dP))
   Flux(v1) += dT*step
   Flux(v2) -= dT*step
   JacobiStep(v1) += step
   JacobiStep(v2) += step
}
enterPhase(+=,Temperature); enterPhase(READ,Flux);
enterPhase(READ,JacobiStep);
for (p <- vertices(mesh)) {
   Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
}
```

Platform-specific

- Apply the results of analysis to specific parallel execution strategies
- Use your parallel example codes as prototypes for the strategy

Platform-specific: MPI



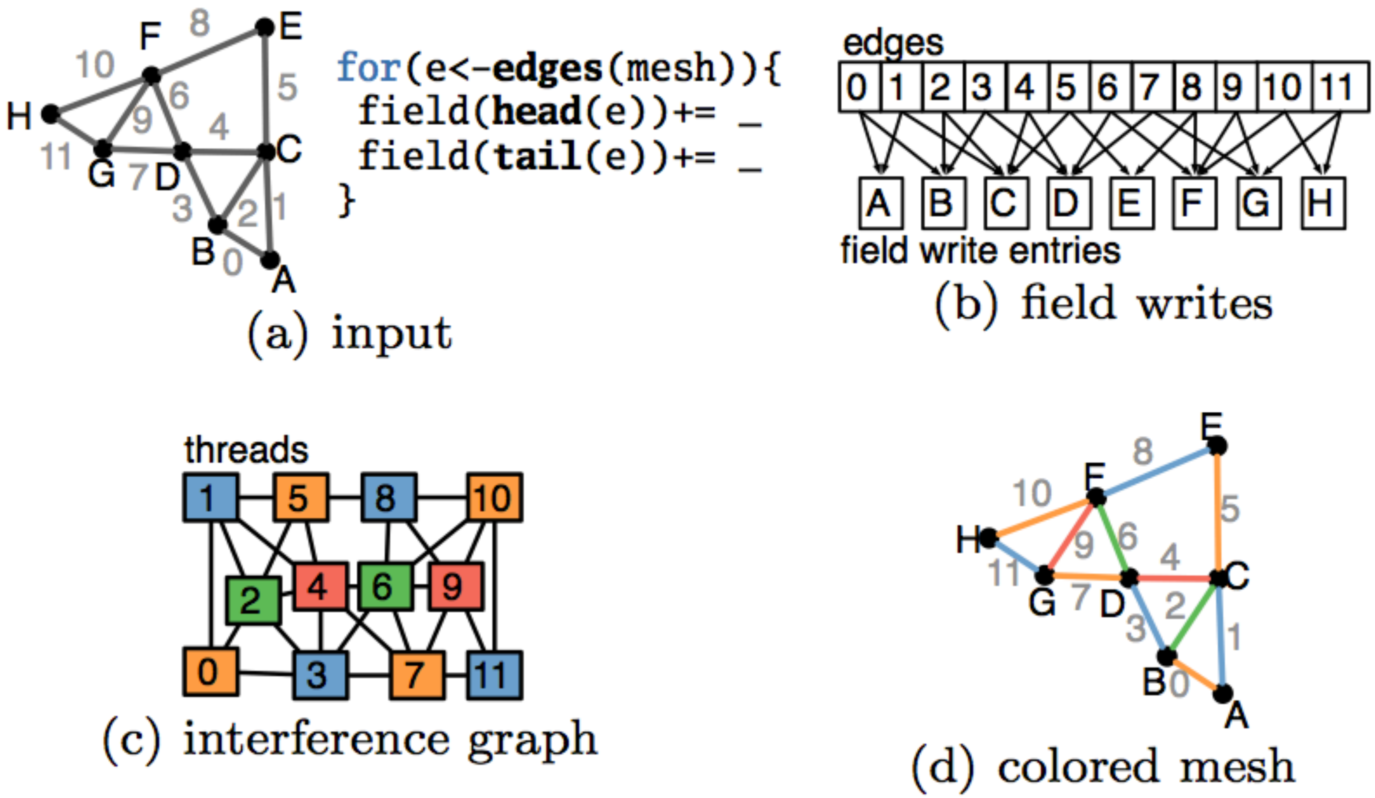
(a) input (b) partition

(c) ghosts $\mathcal{G}_0 = \{E\}$ $\mathcal{G}_1 = \{D\}$

$$(n_0 \mapsto n_1) = \{(\mathtt{field}, E)\}$$
$$(n_1 \mapsto n_0) = \{(\mathtt{field}, D)\}$$

(d) message pattern

Platform-specific: CUDA



(a) input

(b) field writes

(c) interference graph

(d) colored mesh

Evaluating a Design

Our goals were

- Portability
- Performance
- Productivity

Does your language meet these goals?

Portability

Can you automatically retarget the code to different platforms (e.g. an SMP and a GPU)?

To add a new architecture (e.g. a Cell processor), would you need to change the language?

Performance

What is the scalar overhead of your DSL?

- Goal is to have 0 overhead

How does the language scale as you increase the computational resources?

- How does this compare to hand-written code?

Starting with examples helps, since you have reference code to compare against

Productivity

This is usually the hardest to assess.

User studies

- Can users write code in your language faster than writing parallel code for a specific architecture?

How many lines of code is an application in your language vs a general-purpose language?

Summary

Designing DSLs is difficult

- Large range of expertise needed
- Lack of mature frameworks for building DSLs
- Simply choosing what programs to handle takes effort

One approach is to generalize from examples

- Language features to expose parallelism, locality synchronization
- Compiler to automatically extract this knowledge and retarget to different architectures