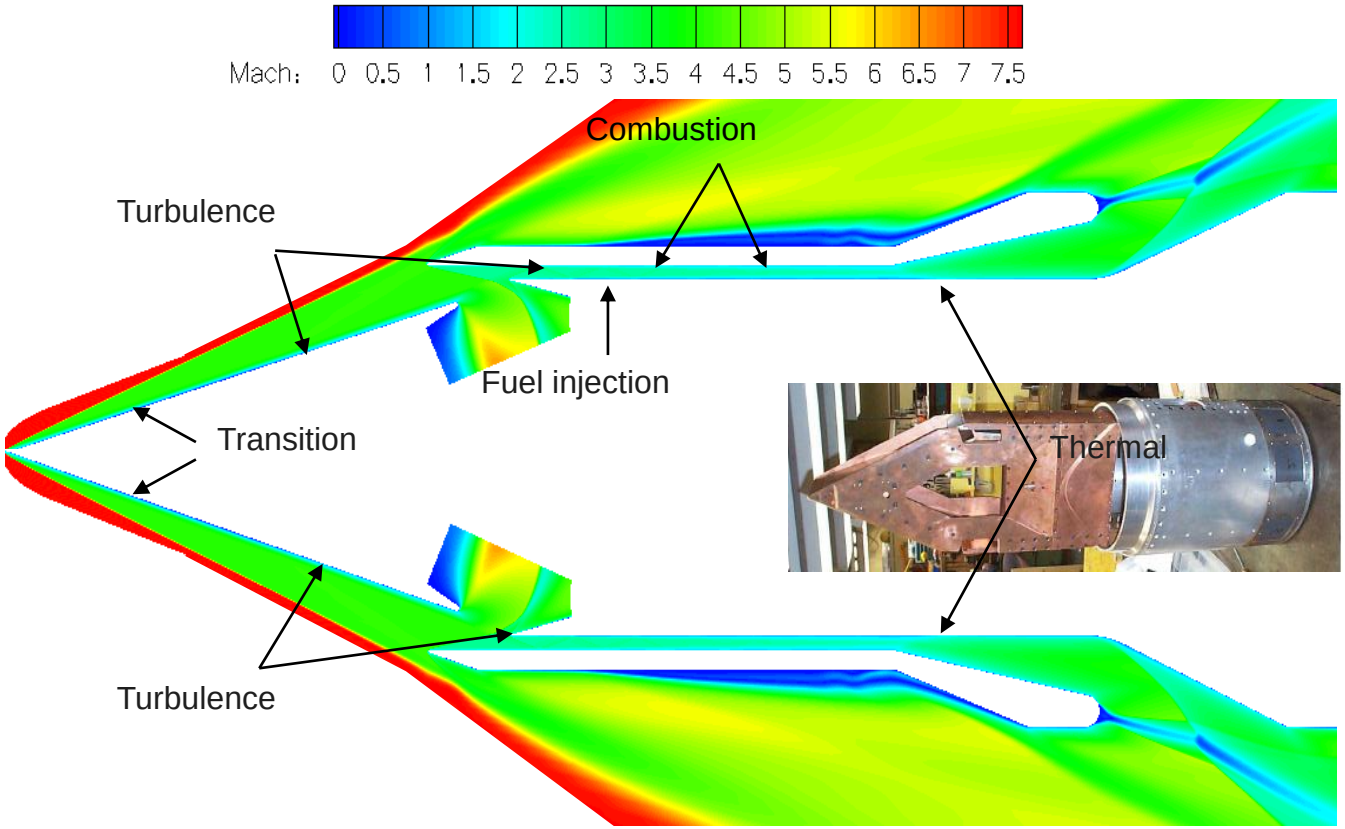
Liszt, a language for PDE solvers

Zachary DeVito, Niels Joubert,
Francisco Palacios, Stephen Oakley,
Montserrat Medina, Mike Barrientos,
Erich Elsen, Frank Ham, Alex Aiken,
Karthik Duraisamy, Eric Darve,
Juan Alonso, Pat Hanrahan

SCIENTISTS NEED PERFORMANCE

Example—PSAAP's simulation of fluid flow in a hypersonic scramjet



SCIENTISTS NEED PERFORMANCE

Example—PSAAP's simulation of fluid flow in a hypersonic scramjet

- State-of-the-art unstructured Renolds-average Navier Stokes Solver (RANS)
- Solving discretized PDEs on a 3D Mesh
- Large Meshes(100 million cells)
- Arbitrary polyhedra for complex geometry
- MPI implementation

SCIENTISTS WANT PORTABILITY

Tried porting to Cell in 2006

Wish to port to GPU in 2011

Worried about porting to ??? in 201X

PROGRAMMING MODELS IN FLUX

Cluster

- Message Passing—MPI

SMPs

- Threads and Locks—pthreads, OpenMP

Hybrid CMPs

- GPU cores—CUDA/OpenGL?
- CPU vector instructions—Intel's SPMD Compiler?
- Task queues for scheduling—AMD FSAIL?

How can scientists write applications when the programming models are changing?

OUR CONTRIBUTION—LISZT

Write code at a *higher level of abstraction*

- Data stored on a 3D mesh of discrete elements

Liszt code is *portable*

- Use different strategies to parallelize programs for clusters, SMPs, and GPUs

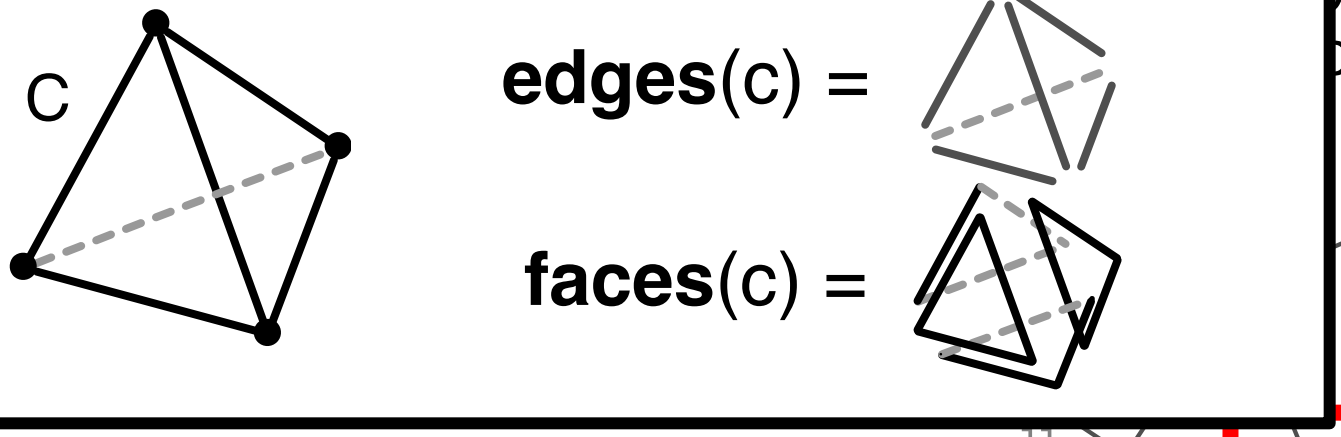
Liszt code is *efficient*

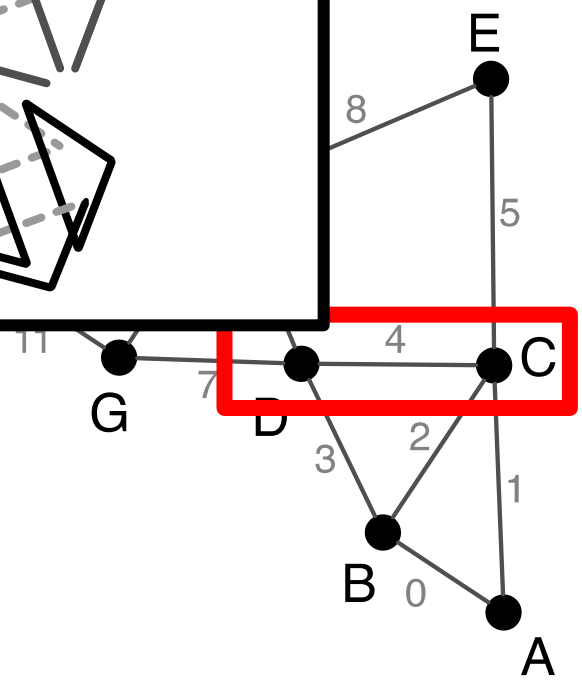
- Performance comparable to best hand-written code

EXAMPLE: HEAT CONDUCTION

```
val Position = FieldWithLabel[Vertex,Double3]("position")
val Temperature = FieldWithLabel[Vertex,Double]("init_temp")
val Flux = FieldWithConst[Vertex,Double](0.0)
val JacobiStep = FieldWithConst[Vertex,Double](0.0)
var i = 0
while (i < 1000) {
  val all = vertices(mesh)
  for (c <- all) {
    val edges = edges(c)
    val faces = faces(c)
    Flux(c) = 0.01*(edges.size - faces.size)
    JacobiStep(c) = 0.01*(edges.size - faces.size)
  }
  for (p <- vertices(mesh)) {
    Temperature(p) += 0.01*Flux(p)/JacobiStep(p)
  }
  for (p <- vertices(mesh)) {
    Flux(p) = 0.0; JacobiStep(p) = 0.0;
  }
  i += 1
}
```

Fields (data)
Mesh Elements
Sets
Parallel for
y Function
ons (assoc



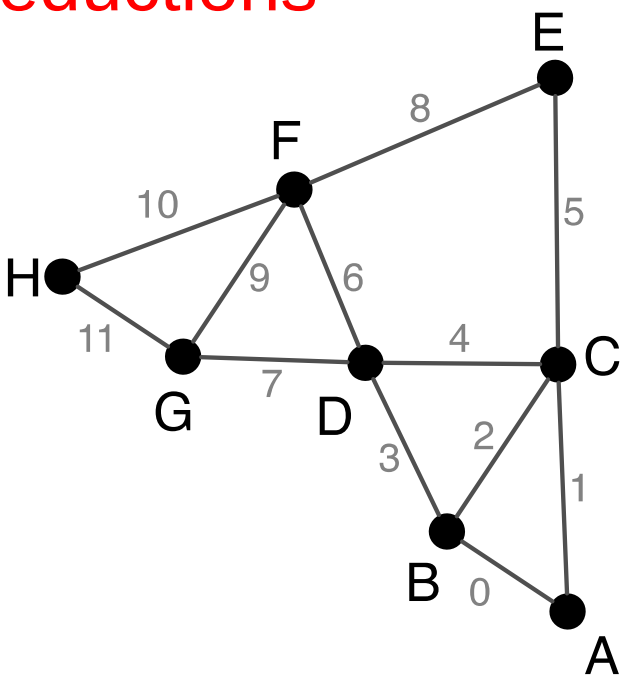


PROPERTIES OF PDE CODE: PARALLEL

```
val Position = FieldWithLabel[Vertex,Double3](“position”)
val Temperature = FieldWithLabel[Vertex,Double](“init_temp”)
val Flux = FieldWithConst[Vertex,Double](0.0)
val JacobiStep = FieldWithConst[Vertex,Double](0.0)
var i = 0
while (i < 1000) {
  val all_edges = edges(mesh)
  for (e <- all_edges) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)
    val step = 1.0/(length(dP))
    Flux(v1) += dT*step
    Flux(v2) -= dT*step
    JacobiStep(v1) += step
    JacobiStep(v2) += step
  }
  for (p <- vertices(mesh)) {
    Temperature(p) += 0.01*Flux(p)/JacobiStep(p)
  }
  for (p <- vertices(mesh)) {
    Flux(p) = 0.0; JacobiStep(p) = 0.0;
  }
  i += 1
}
```

independent computation per edge

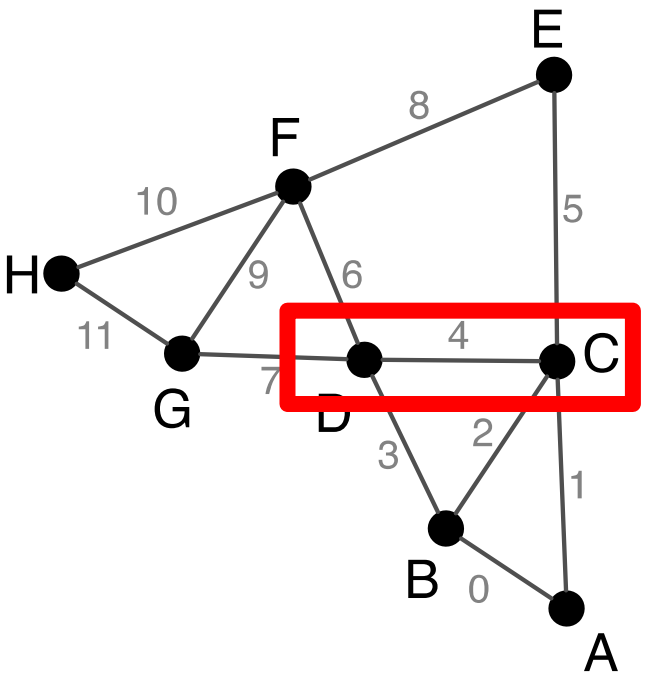
followed by reductions



PROPERTIES OF PDE CODE: STENCIL

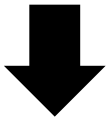
```
val Position = FieldWithLabel[Vertex,Double3](“position”)
val Temperature = FieldWithLabel[Vertex,Double](“init_temp”)
val Flux = FieldWithConst[Vertex,Double](0.0)
val JacobiStep = FieldWithConst[Vertex,Double](0.0)
var i = 0
while (i < 1000) {
  val all_edges = edges(mesh)
  for (e <- all_edges) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)
    val step = 1.0/(length(dP))
    Flux(v1) += dT*step
    Flux(v2) -= dT*step
    JacobiStep(v1) += step
    JacobiStep(v2) += step
  }
  for (p <- vertices(mesh)) {
    Temperature(p) += 0.01*Flux(p)/JacobiStep(p)
  }
  for (p <- vertices(mesh)) {
    Flux(p) = 0.0; JacobiStep(p) = 0.0;
  }
  i += 1
}
```

stencil is local and bounded



LISZT'S APPROACH

Abstract interface to data using mesh elements and fields



Reason about data-dependencies to infer the PDE's stencil

Stencil enables program analysis used to optimize code

- Detection of ghost cells/halos
- Writer conflicts in reductions

LISZT'S ABSTRACTION SIMPLIFIES DEPENDENCY ANALYSIS

"What data does this value depend on?"

Difficult in general

```
double a = B[f(i)]
```

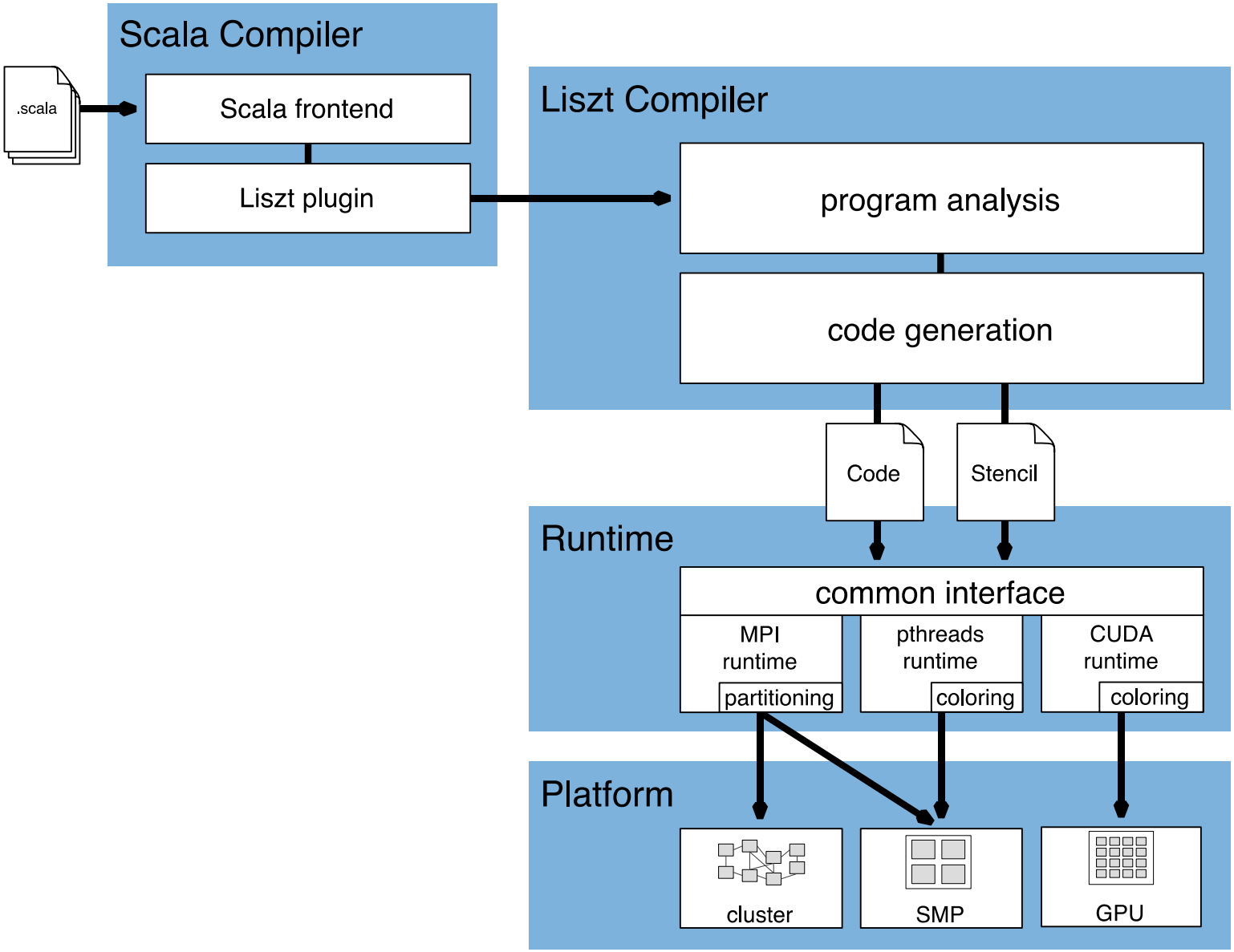
Must reason about `f(i)`, a general function

Simple in Liszt

```
val a = Temperature(head(e))
```

Must reason about `head(e)`, a built-in function on mesh topology

Implementation


Scala Compiler
.scala
Scala frontend
Liszt plugin
Liszt Compiler
program analysis
code generation
Code
Stencil
Runtime
common interface
MPI runtime
partitioning
pthreads runtime
coloring
CUDA runtime
coloring
Platform
cluster
SMP
GPU

LISZT & OPTIMESH

Delite and Liszt were developed *independently*

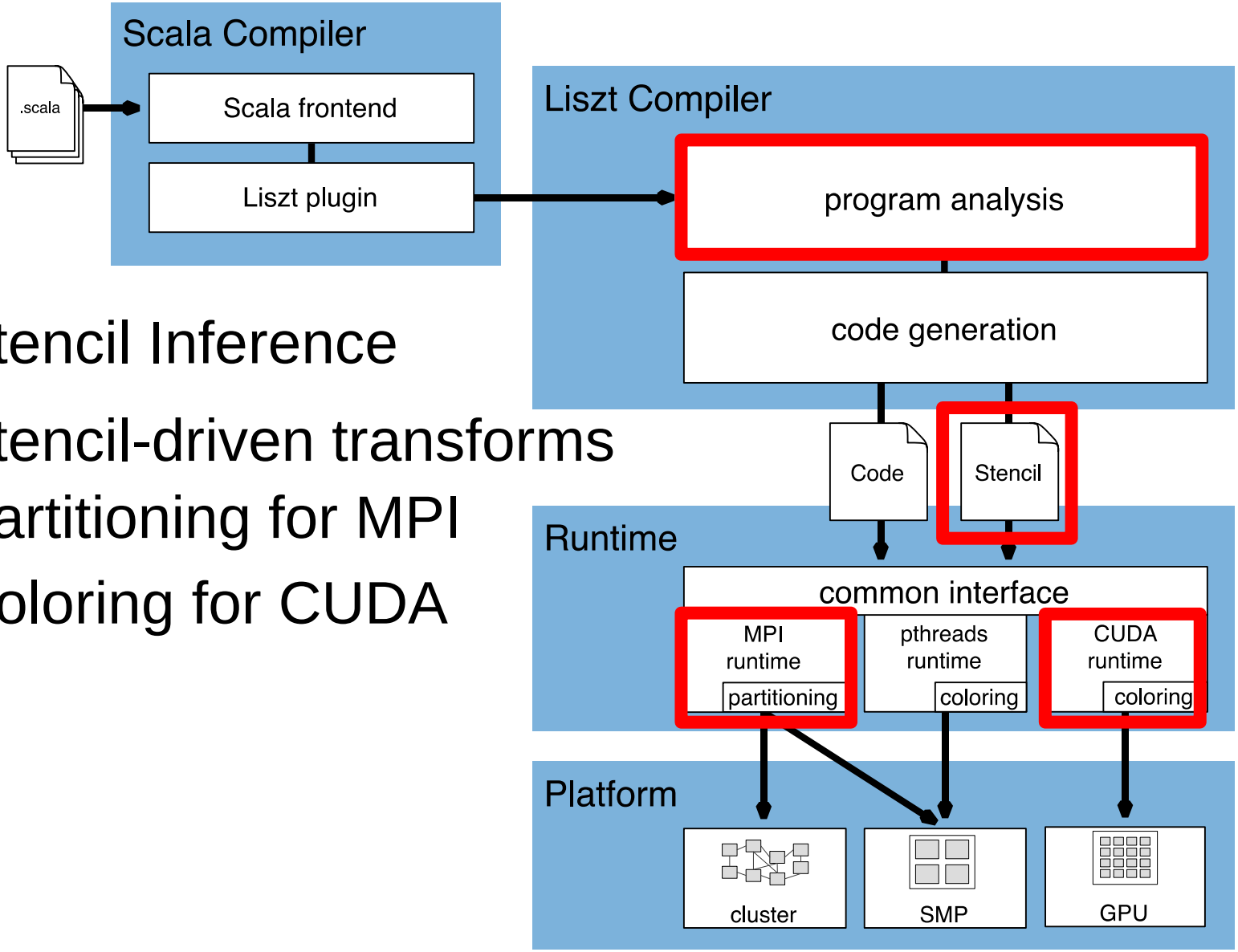
OptiMesh is a *port* of Liszt to the Delite framework

- Light-weight module staging used to represent code
- Uses Delite's CSE, DSE, and code motion passes
- GPU and SMP implementations

Stencil Inference

Stencil-driven transforms

- Partitioning for MPI
- Coloring for CUDA

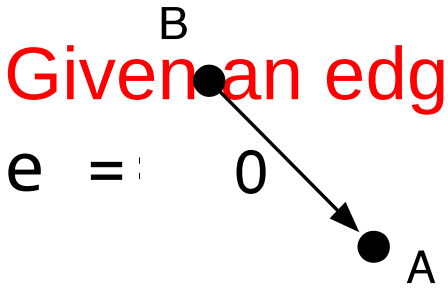


INFERRING THE STENCIL

Code

```
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
}
```

Environment



Given an edge what field values will this code read/w

INFERRING THE STENCIL

Code

```
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
}
```

Stencil

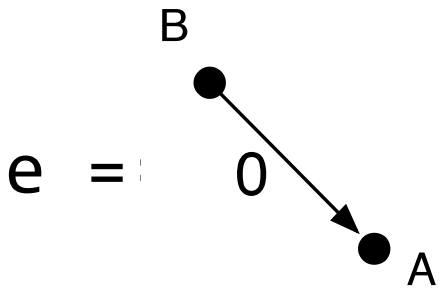
Reads:

```
Position(A)
Position(B)
Temperature(A)
Temperature(B)
```

Writes:

```
Flux(A)
Flux(B)
JacobiStep(A)
JacobiStep(B)
```

Environment




```
v1 = head(e) = A
v2 = tail(e) = B
```

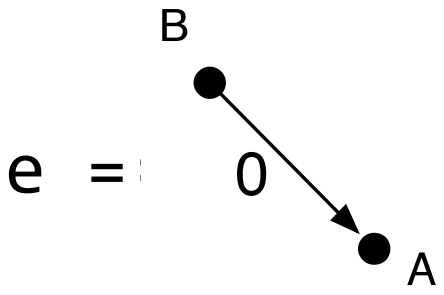
HANDLING BRANCHES

```
for (e <- edges(mesh)) {
  <...>
  if(foo()) {
    JacobiStep(head(e)) += 1
  }
}
```

Stencil Reads:

Conservatively assume that if-statement runs

Writes:
`JacobiStep(A)`




```
v1 = head(e) = A
v2 = tail(e) = B
```

HANDLING LOOPS

```
for (e <- edges(mesh)) {
  val v = head(e)
  while(foo()) {
     JacobiStep(v) += 1
     v = tail(e) /*Error*/
  }
}
```

Stencil Reads:

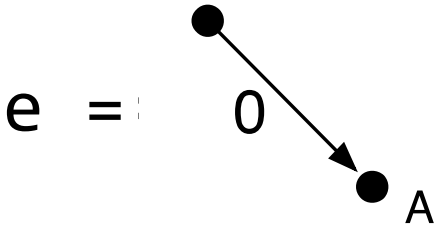
Writes:

`JacobiStep(A)`

Liszt enforces

- Mesh variables are constant
- No recursion

∴ Interpret loop once



`v = head(e) = A`

FORMALIZING STENCIL INFERENCE

By defining a program transformation $\mathcal{T}$

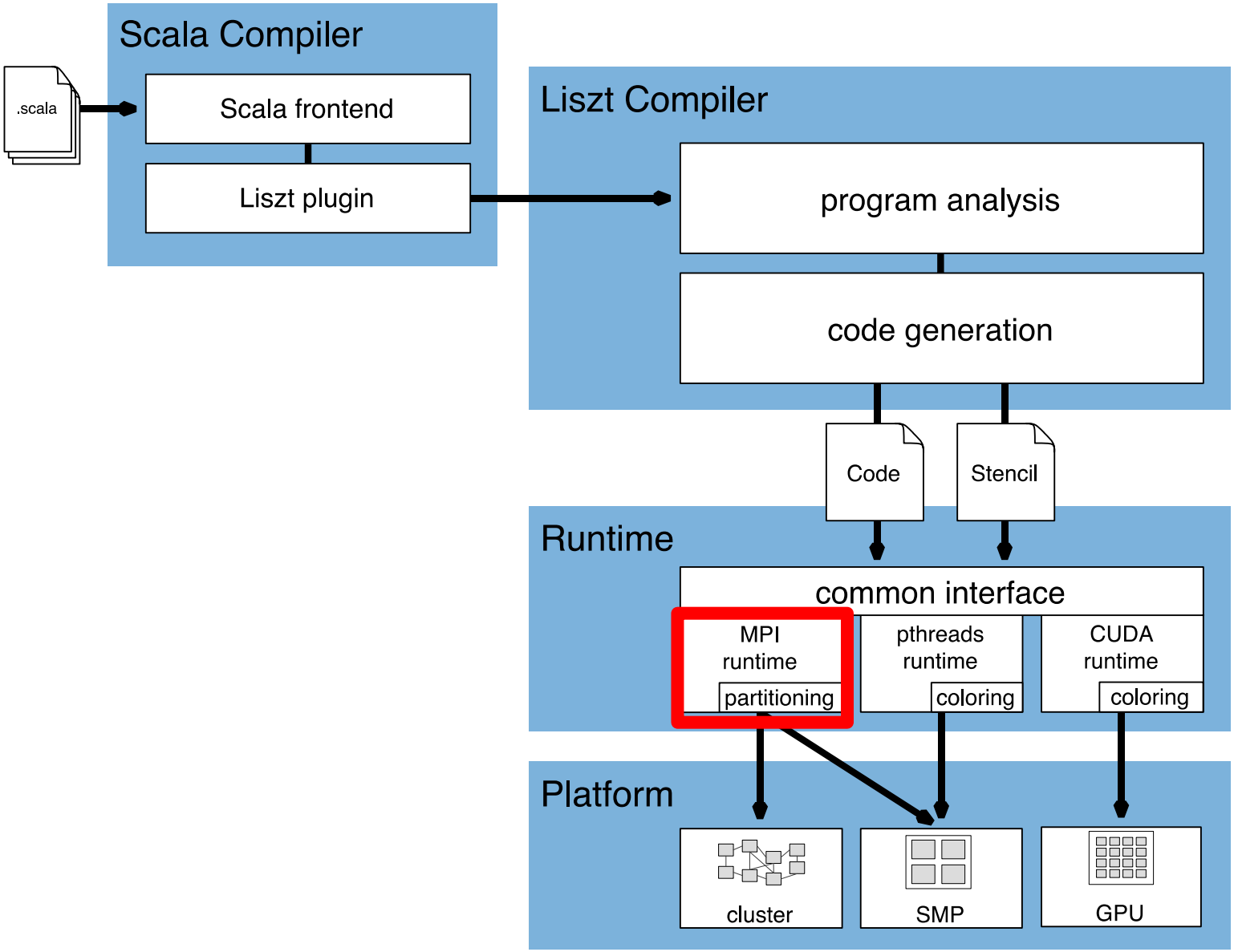
$$\mathcal{T}(\ \text{if}(e_p)\ e_t\ \text{else}\ e_e\ ) = \ \mathcal{T}(e_p);\ \mathcal{T}(e_t);\ \mathcal{T}(e_e);$$

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

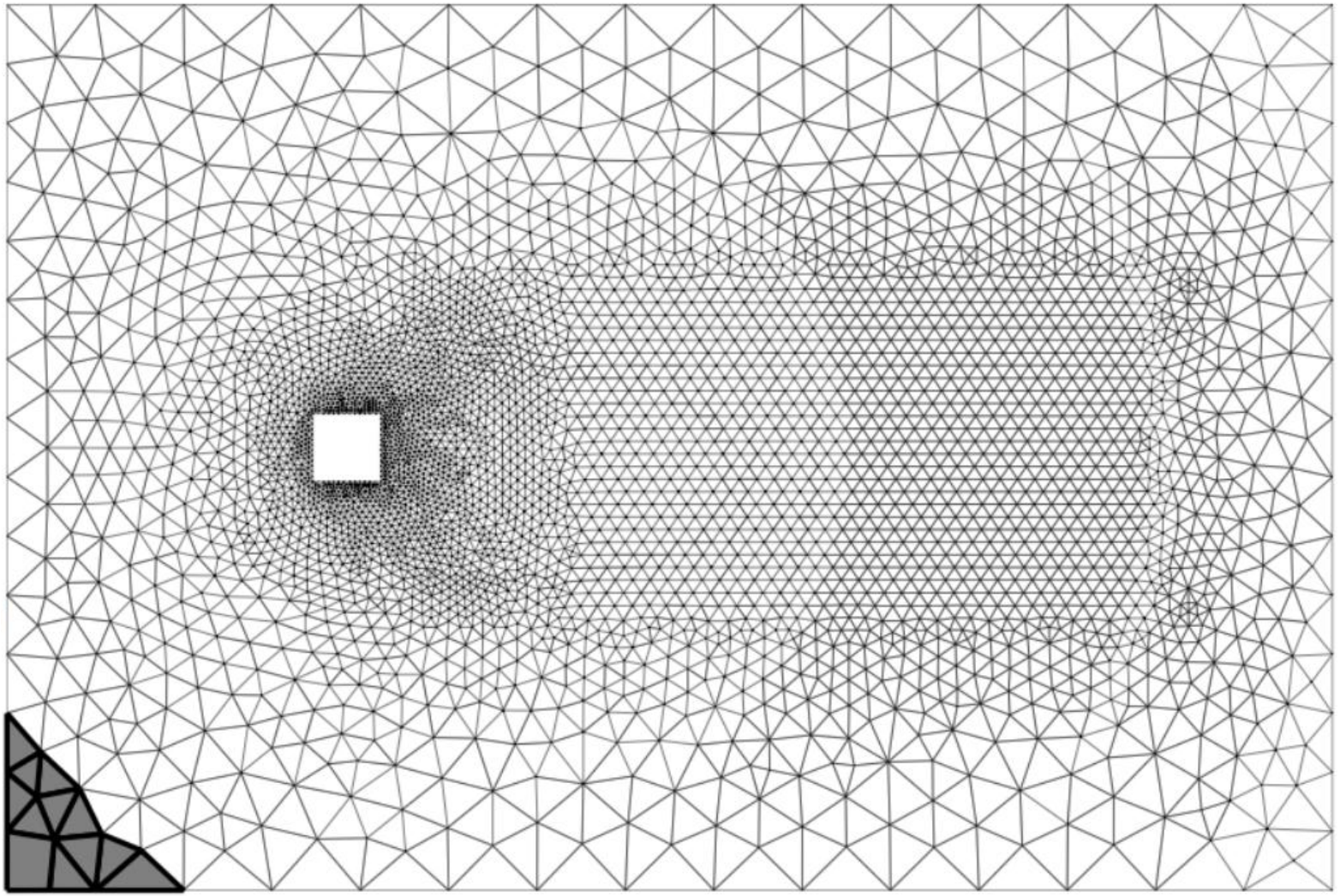
$$\mathcal{T}(\ f(a_0,...,a_n)\ ) = \ f'(\mathcal{T}(a_0),...,\mathcal{T}(a_n))$$

More details at liszt.stanford.edu

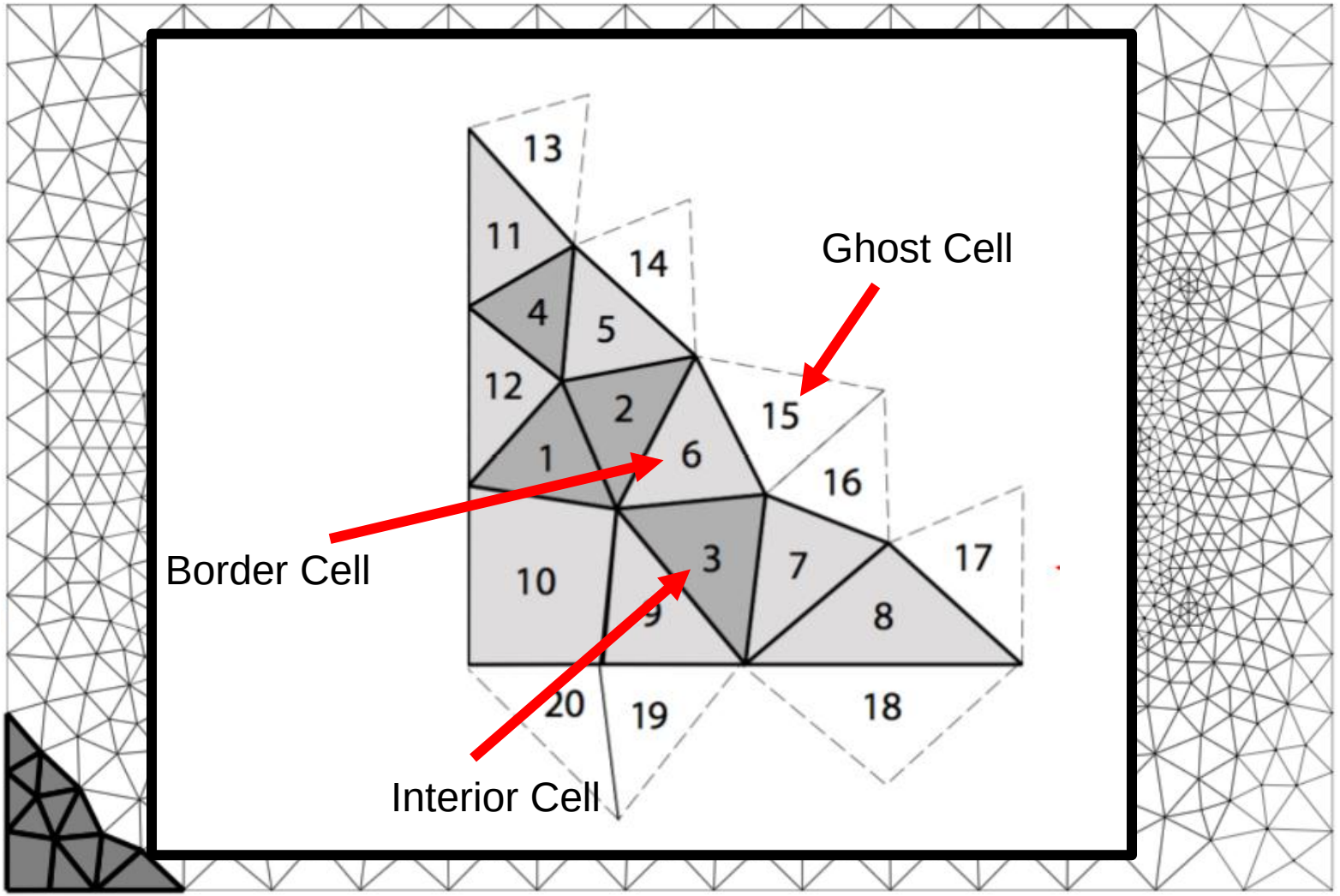
Using the stencil


Scala Compiler
.scala
Scala frontend
Liszt plugin
Liszt Compiler
program analysis
code generation
Code
Stencil
Runtime
common interface
MPI runtime
partitioning
pthreads runtime
coloring
CUDA runtime
coloring
Platform
cluster
SMP
GPU

MPI: PARTITIONING WITH GHOSTS



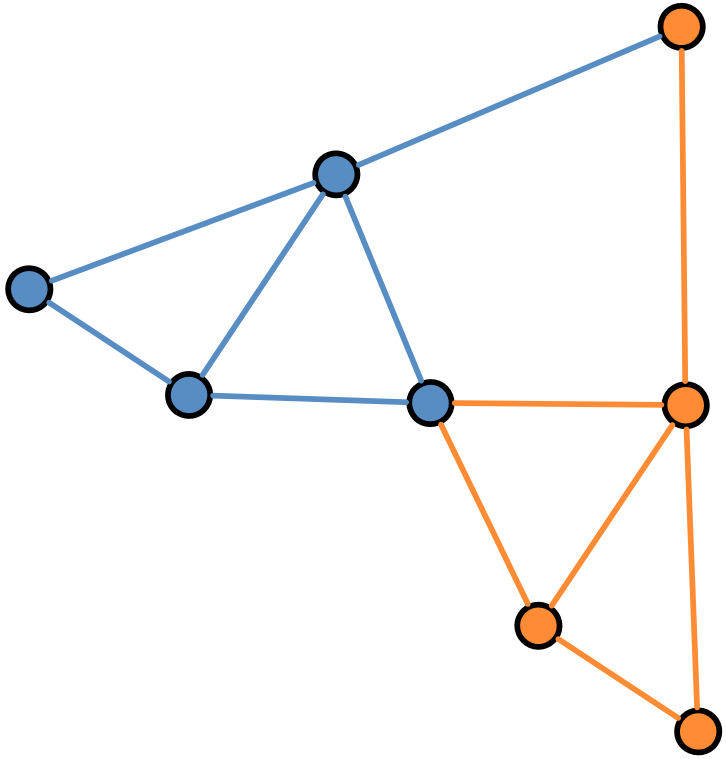
MPI: PARTITIONING WITH GHOSTS



MPI: PARTITIONING WITH GHOSTS

1. Decompose into threads of execution:

   Partition Mesh (ParMETIS, G. Karypis)

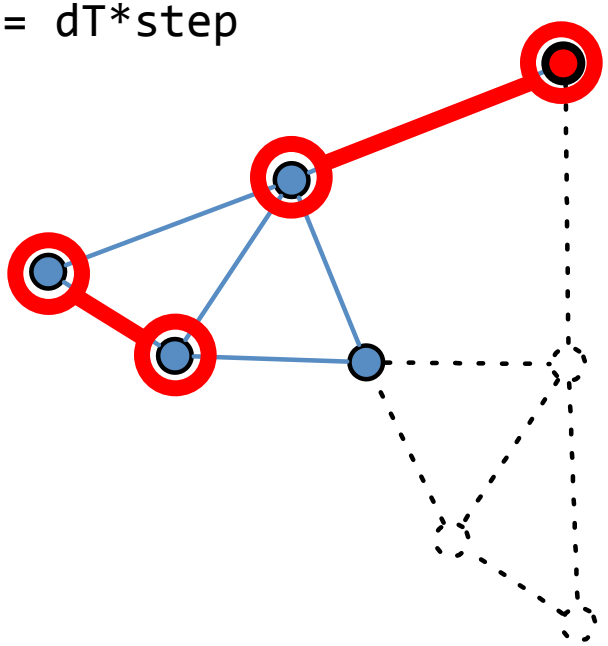


MPI: PARTITIONING WITH GHOSTS

2. Resolve dependencies using the stencil

Given a partition, determine its ghosts elements by running the stencil over the local topology

```
Flux(head(e)) += dT*step
Flux(tail(e)) -= dT*step
```



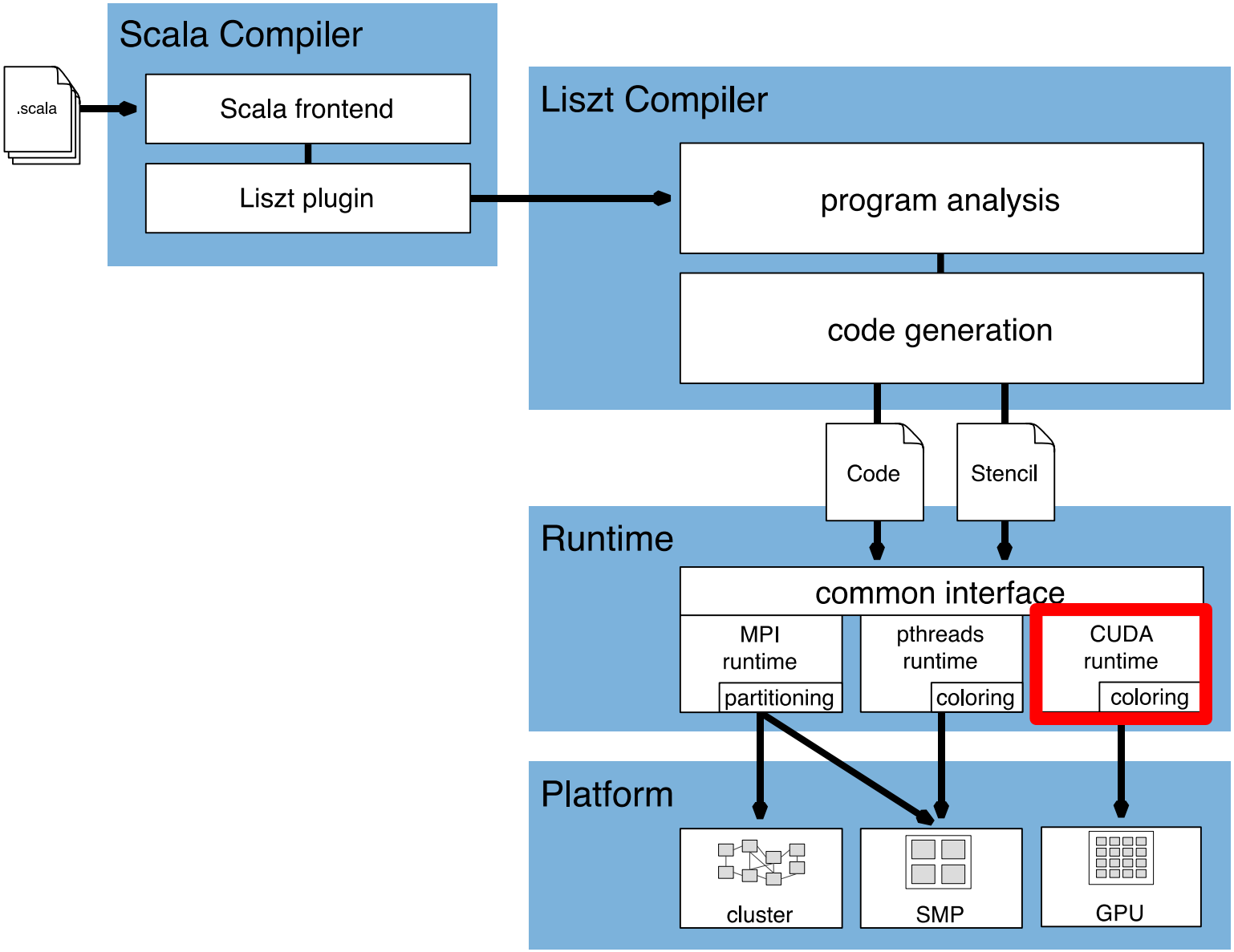
non-local !

MPI: PARTITIONING WITH GHOSTS

3. Execute using a parallel programming model

SPMD with communication between for-statements

```
for (e <- edges(mesh)) {
    <...>
    Flux(head(e)) += dT*step
    Flux(tail(e)) -= dT*step
    <...>
}
transferFlux()
for (p <- vertices(mesh)) {
     val a = Flux(p)
}
```


Scala Compiler
.scala
Scala frontend
Liszt plugin
Liszt Compiler
program analysis
code generation
Code
Stencil
Runtime
common interface
MPI runtime
partitioning
pthreads runtime
coloring
CUDA runtime
coloring
Platform
cluster
SMP
GPU

CUDA: PARTITIONING WITH GHOSTS?

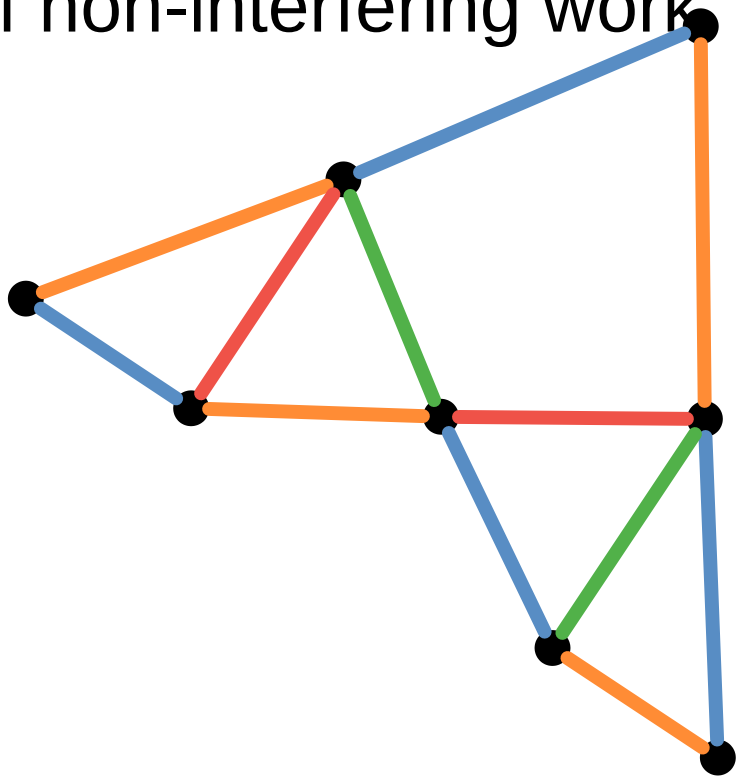
But!

- 20,000+ parallel threads of execution
- Small memory (~2 GB)
- Surface area vs. volume: most elements need ghosts

CUDA: ~~PARTITIONING WITH GHOSTS?~~ COLORING TO AVOID CONFLICTS

Use shared memory to handle *reads*

Resolve *writes* (i.e. reductions) by launching batches of non-interfering work

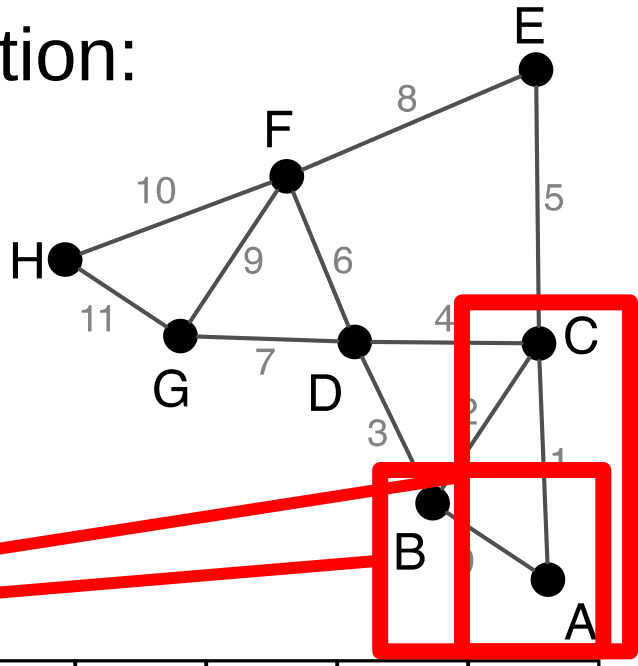


CUDA: COLORING TO AVOID CONFLICTS

1. Decompose into threads of execution:

   Assign each edge to a thread

```
Flux(head(e)) += dT*step
Flux(tail(e)) -= dT*step
```



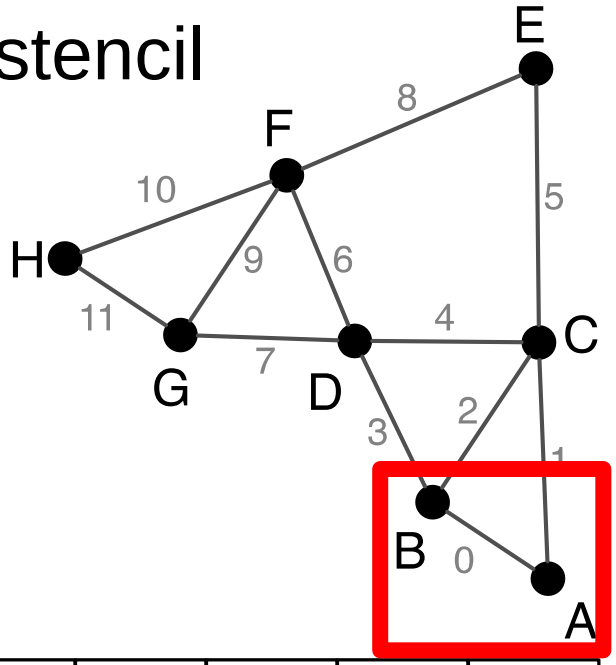
CUDA Threads

0	1	2	3	4	5	6	7	8	9	10	11

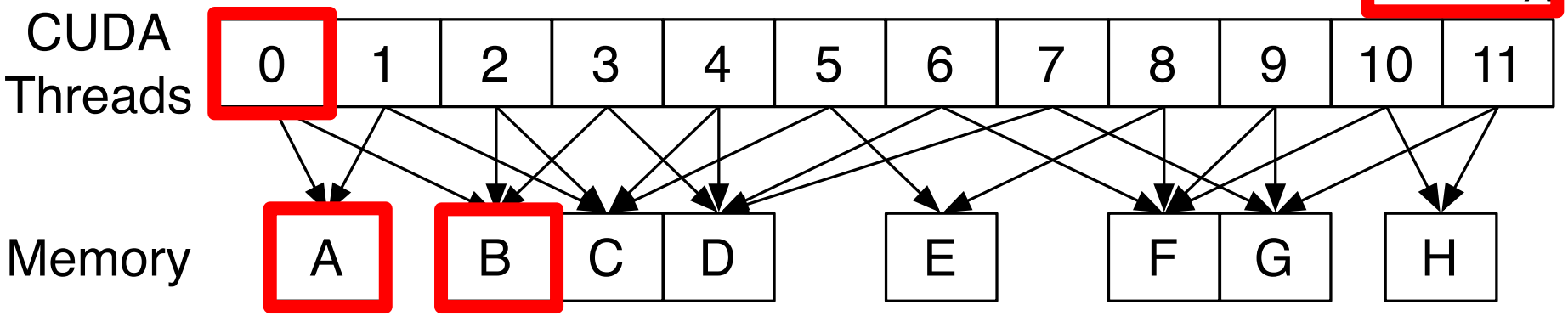
CUDA: COLORING TO AVOID CONFLICTS

2. Resolve dependencies using the stencil

Use stencil to create a graph connecting each thread to the memory it writes



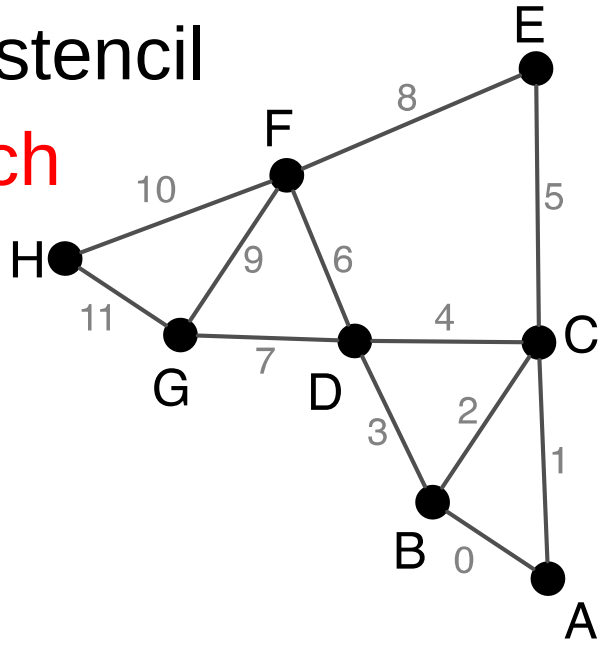

```
Flux(head(e)) += dT*step
Flux(tail(e)) -= dT*step
```



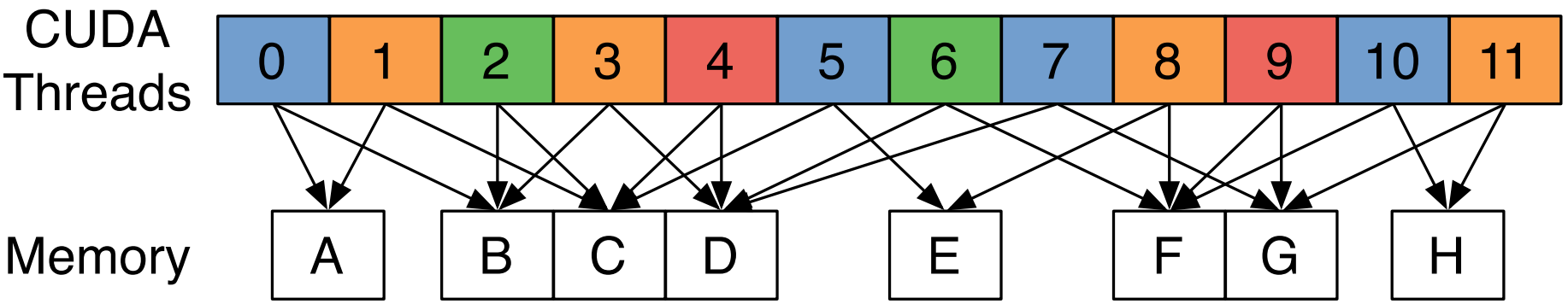
CUDA: COLORING TO AVOID CONFLICTS

2. Resolve dependencies using the stencil

Color the graph s.t. 2 threads which write to the same memory have different colors




```
Flux(head(e)) += dT*step
Flux(tail(e)) -= dT*step
```



CUDA: COLORING TO AVOID CONFLICTS

3. Execute using a parallel programming model

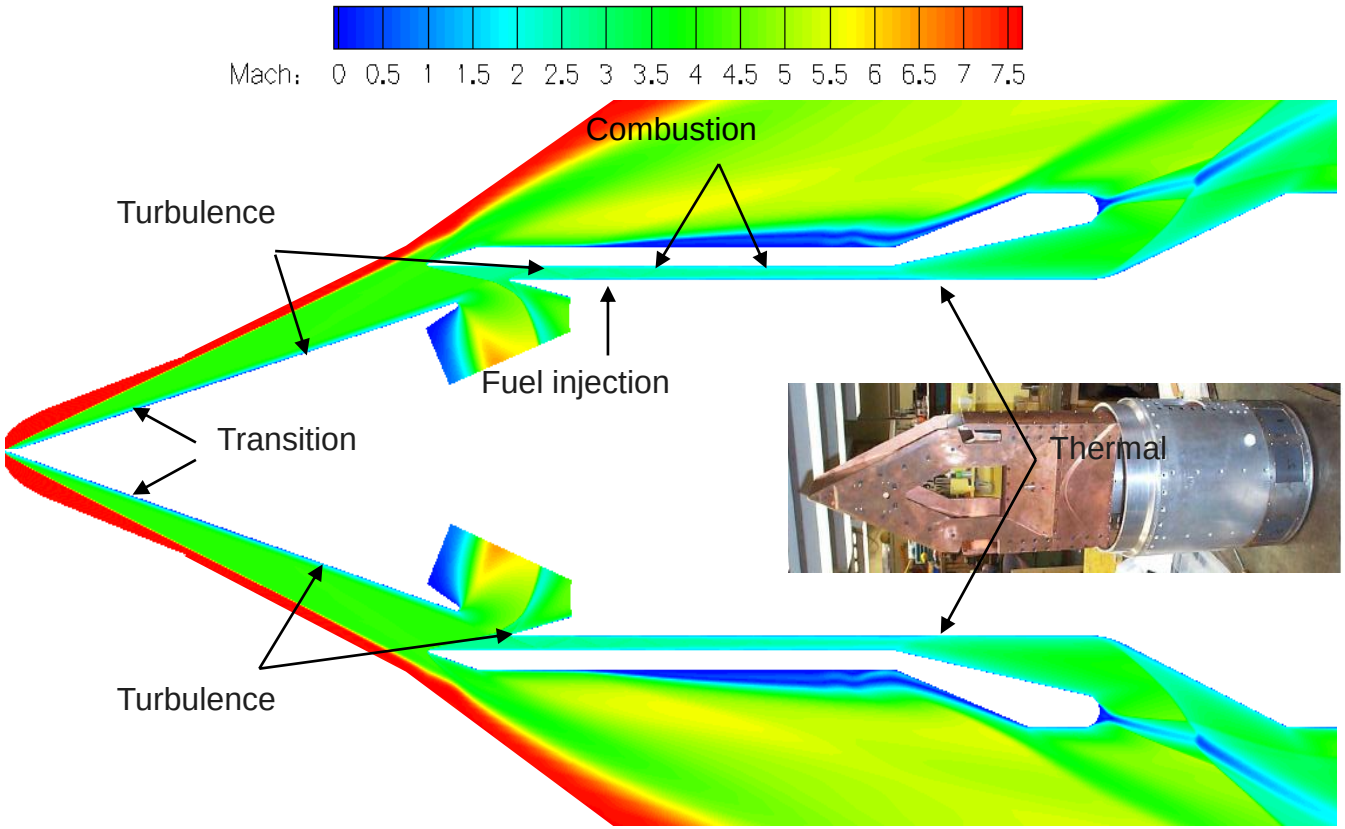
   Launch batches of independent threads

```
fluxCalc(color0)
sync()
fluxCalc(color1)
sync()
fluxCalc(color2)
sync()
fluxCalc(color3)
sync()
```

```
def fluxCalc(color : Set[Edge]) {
  for (e <- color) {
    <...>
    Flux(head(e)) += dT*step
    Flux(tail(e)) -= dT*step
    <...>
  }
}
```

Results

PSAAP'S CODE PORTED TO LISZT



PSAAP'S CODE PORTED TO LISZT

Full system scramjet simulation [Pecnik et al.]

- ~2,000 lines of Liszt code
- Ported from ~40,000 lines of MPI/C++ code

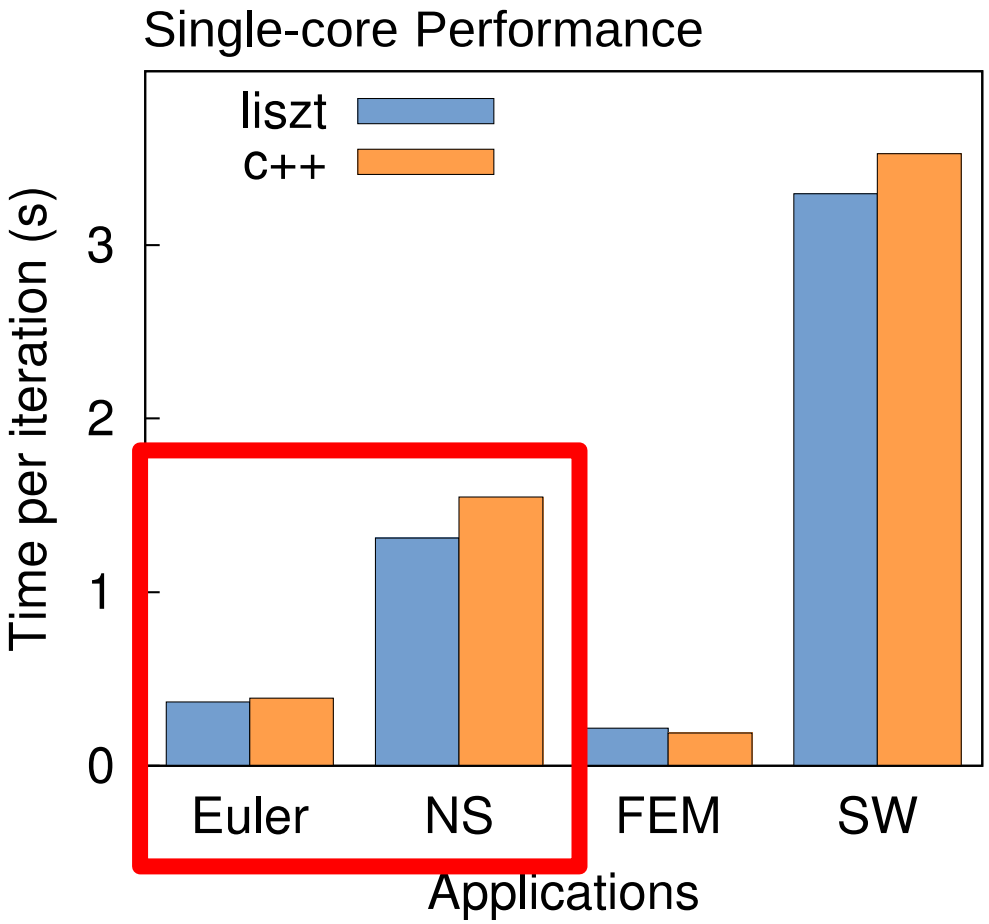
Two Versions

- Euler—inviscid only
- Navier-Stokes (NS)—viscous + inviscid

To test different working sets (ported from scalar C++)

- Simple FEM code
- Shallow Water (SW) simulator

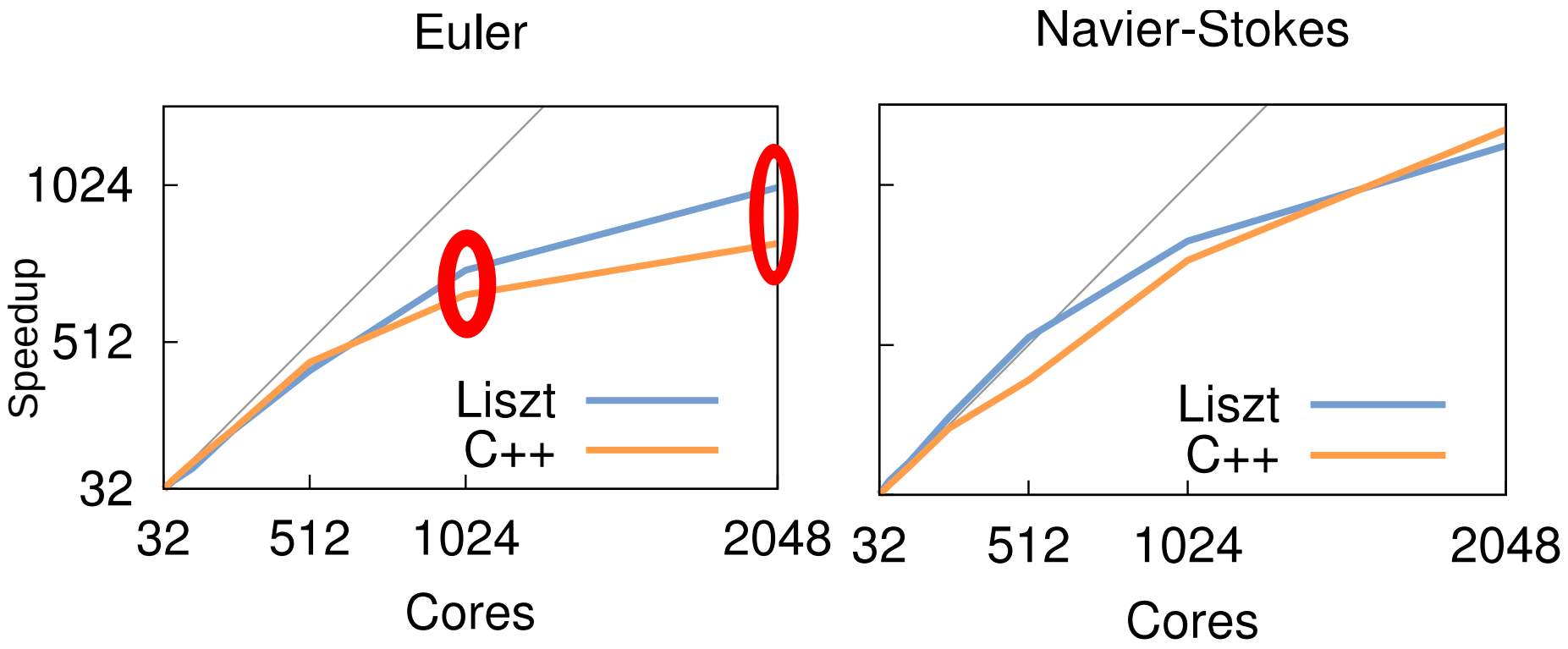
LISZT PERFORMS AS WELL AS C++



8-core Intel Nehalem E5520 2.26Ghz, 8GB RAM

All performance results use double precision

LISZT SCALES ON LARGE CLUSTERS



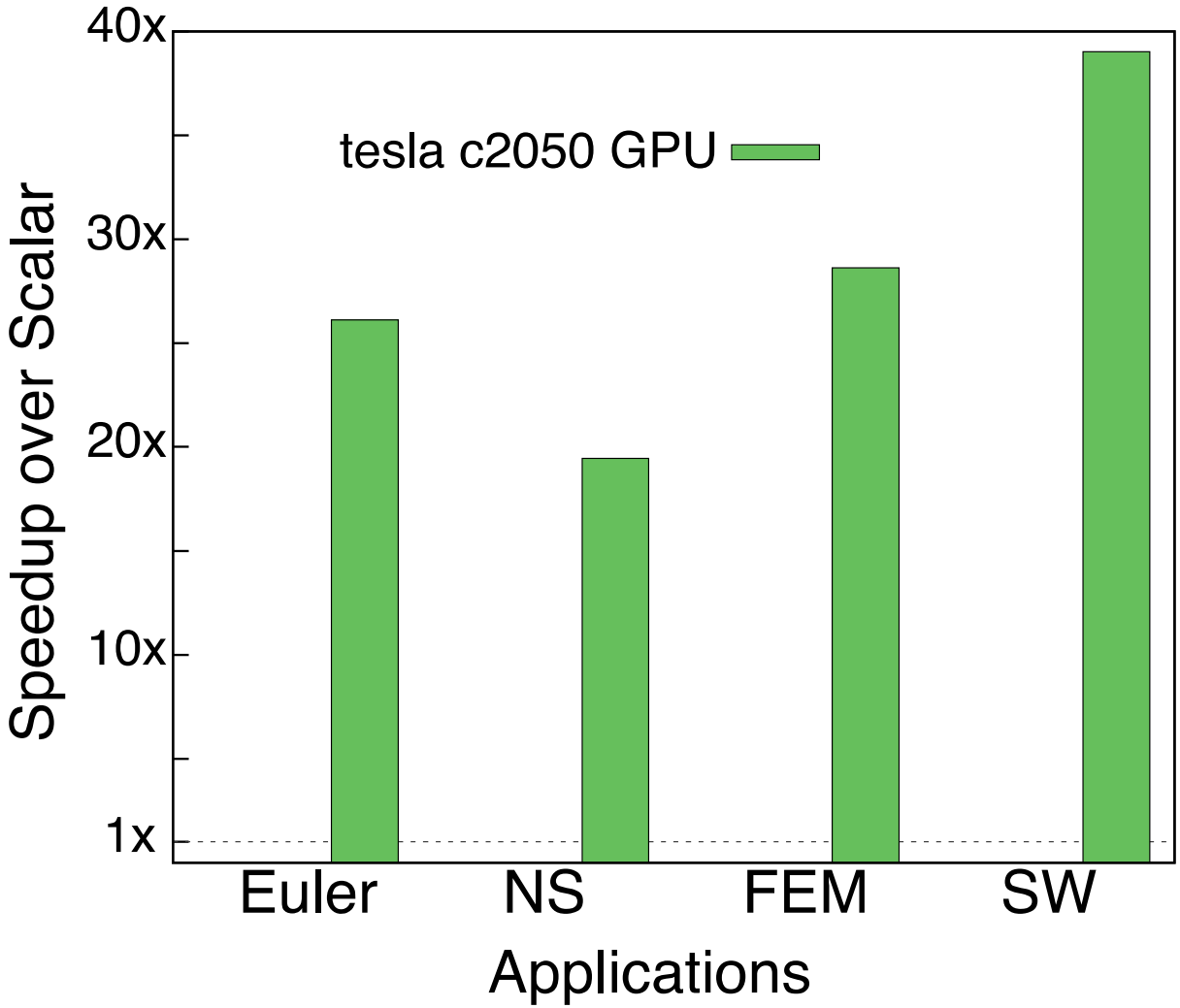
~20 million cells/mesh

Knee at 20k cells/core at 1024 cores (SA vs. Volum

Liszt optimized away a message in Euler code

256 boards, 2 Nehalem X5650 processors/board , 4 cores/processor, OpenMPI

LISZT RUNS ON GPUs

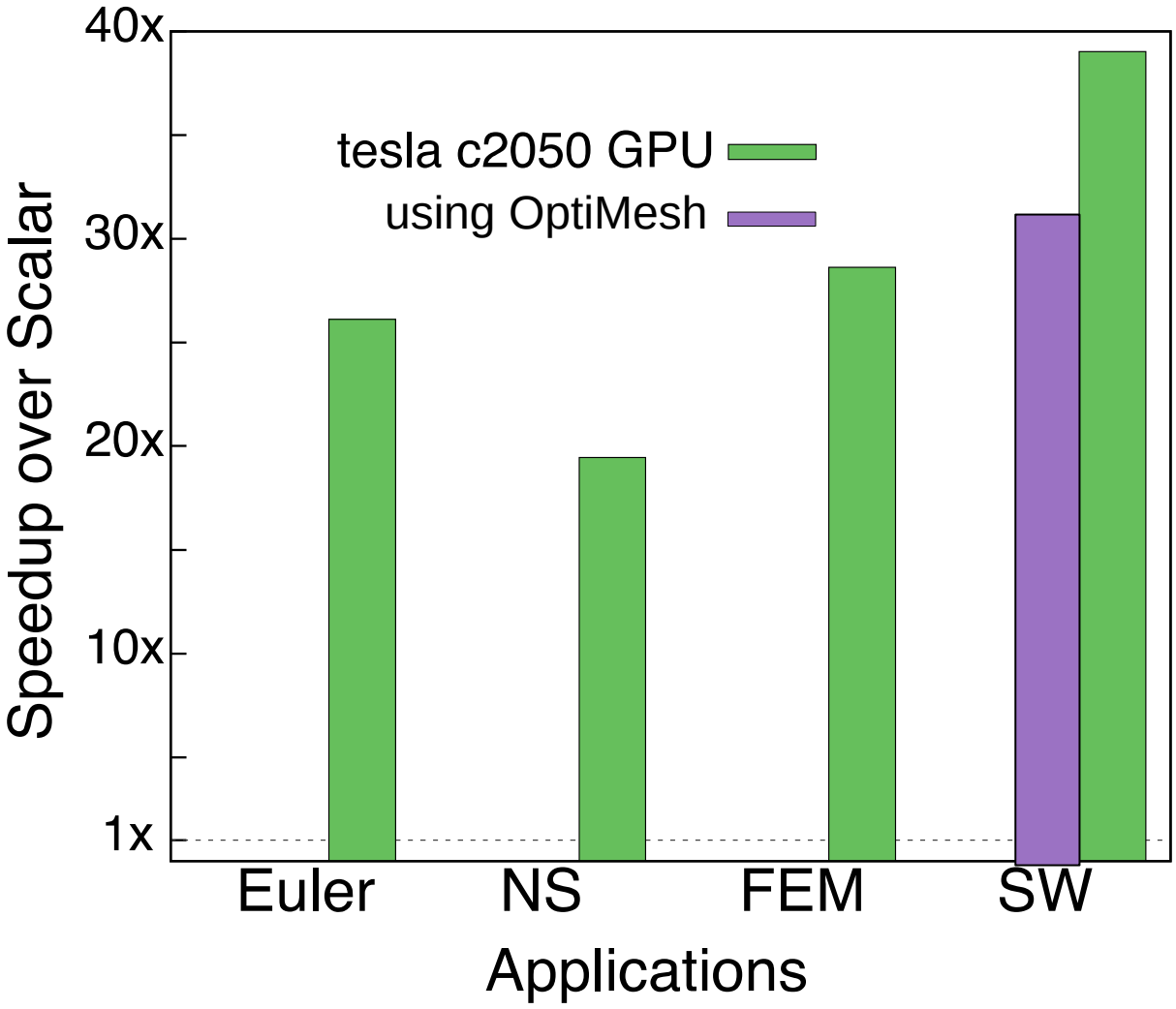


NVidia Tesla C2050 GPU

NS speedups from 7x to 28x in literature

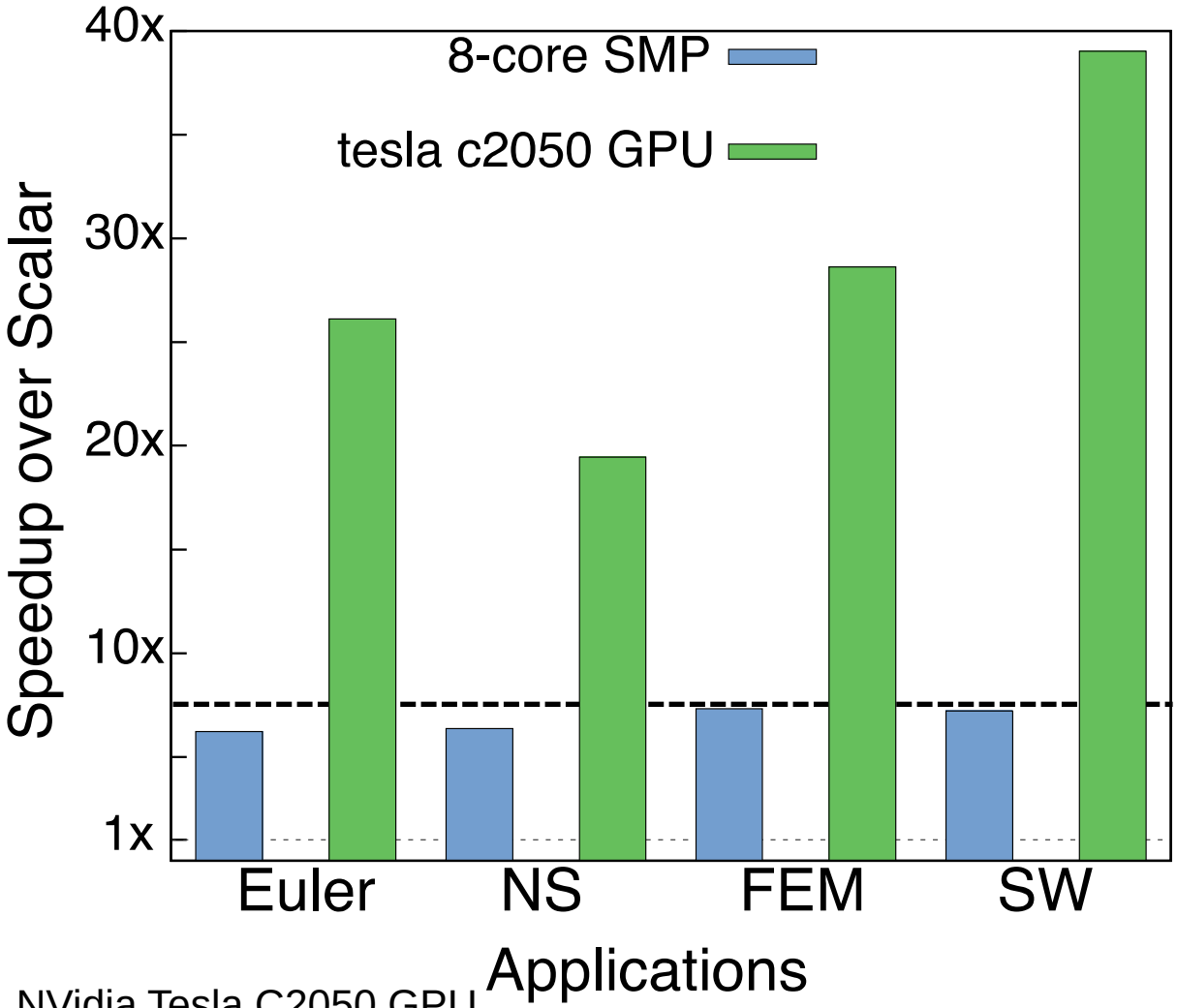
[Corrigan et al., Kampolis et al., Giles et al.]

LISZT RUNS ON GPUs—IN DELITE



NVidia Tesla C2050 GPU

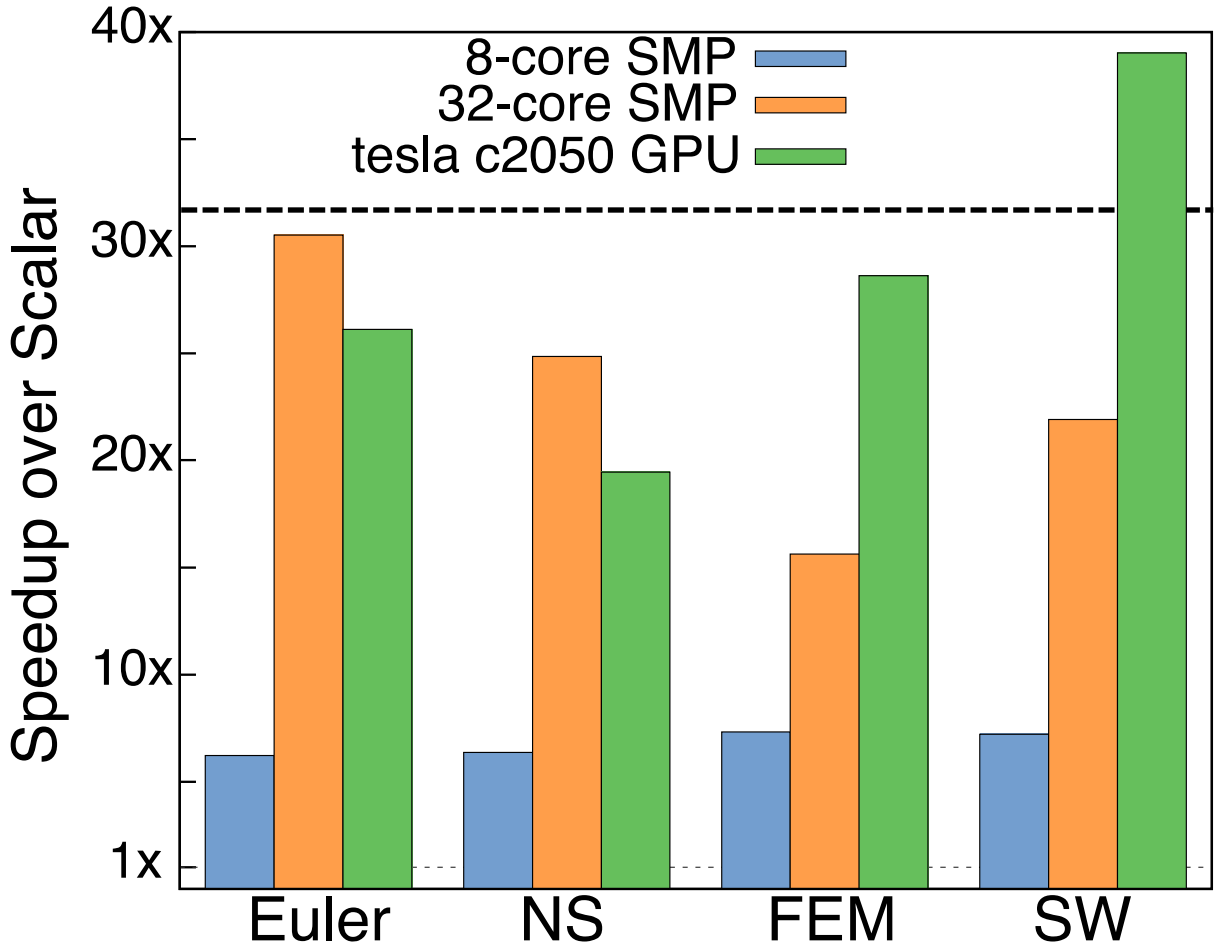
LISZT ALSO RUNS ON SMPs



NVidia Tesla C2050 GPU

8-core Intel Nehalem E5520 2.26Ghz, 8GB RAM

LISZT ALSO RUNS ON SMPs

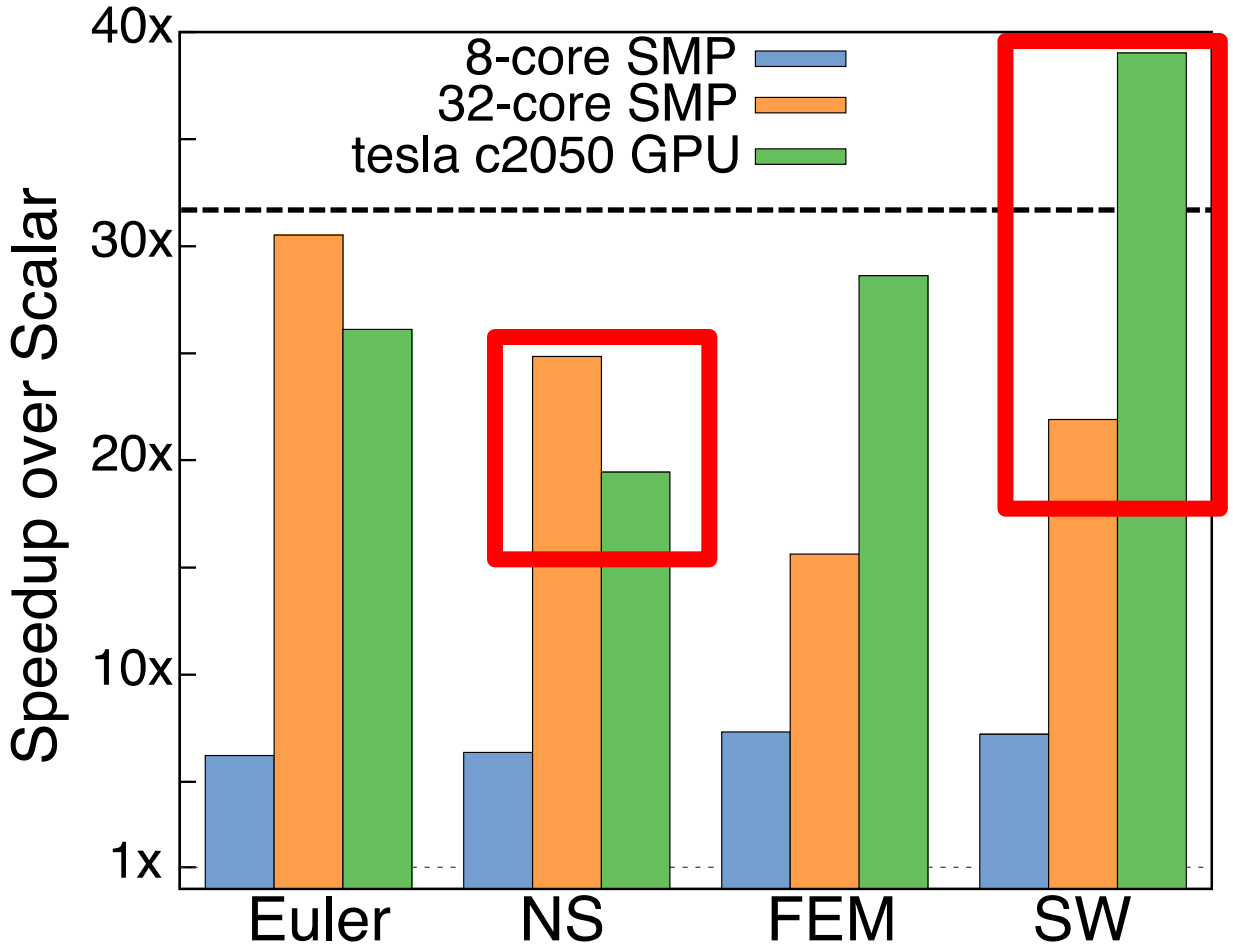


NVidia Tesla C2050 GPU

8-core Intel Nehalem E5520 2.26Ghz, 8GB RAM

32-core Intel Nehalem-EX X7560 2.26GHz, 128GB RAM

LISZT ALSO RUNS ON SMPs



NVidia Tesla C2050 GPU

8-core Intel Nehalem E5520 2.26Ghz, 8GB RAM

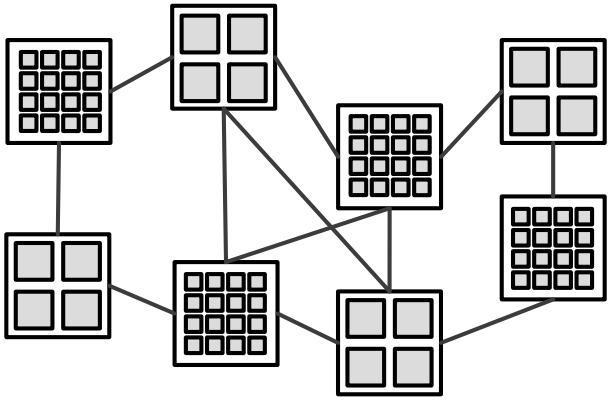
32-core Intel Nehalem-EX X7560 2.26GHz, 128GB RAM

LIMITATIONS & FUTURE WORK

Not all PDE solvers can be expressed

- Want adaptive (e.g. AMR) and regular meshes
- Want sparse matrix libraries/solvers (in progress)

Combination of runtimes
(e.g. MPI + CUDA, in progress)



SUMMARY

Write at a *high level*—fields defined on a mesh

Portable to clusters, SMPs, and GPUs without modification

Performance equivalent to handwritten code

liszt .stanford.edu
zdevito @stanford.edu