#### Lecture 22:

# Domain-Specific Programming Systems

Parallel Computer Architecture and Programming CMU 15-418/15-618, Fall 2018

Slide acknowledgments:
Pat Hanrahan, Zach Devito (Stanford University)
Jonathan Ragan-Kelley (MIT)

#### Course themes:

#### Designing computer systems that <u>scale</u>

(running faster given more resources)

#### Designing computer systems that are efficient

(running faster under constraints on resources)

#### **Techniques discussed:**

**Exploiting parallelism in applications** 

**Exploiting locality in applications** 

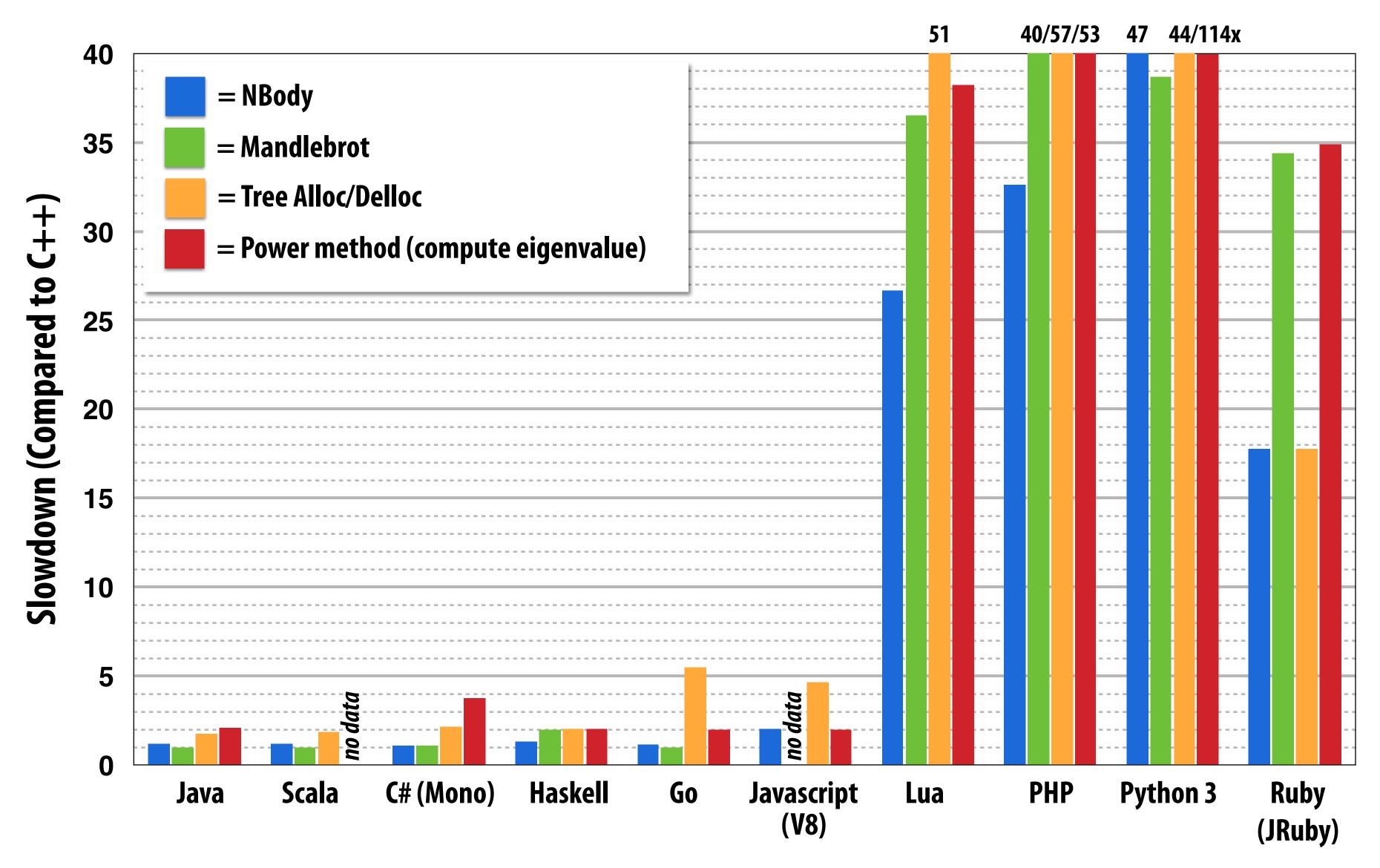
Leveraging hardware specialization (earlier lecture)

## Claim: most software uses modern hardware resources inefficiently

- Consider a piece of sequential C code
  - Let's consider the performance of this code "baseline performance"
- Well-written sequential C code: ~ 5-10x faster
- Assembly language program: another small constant factor faster
- Java, Python, PHP, etc. ??

## Code performance: relative to C (single core)

GCC -03 (no manual vector optimizations)



Data from: The Computer Language Benchmarks Game: http://shootout.alioth.debian.org

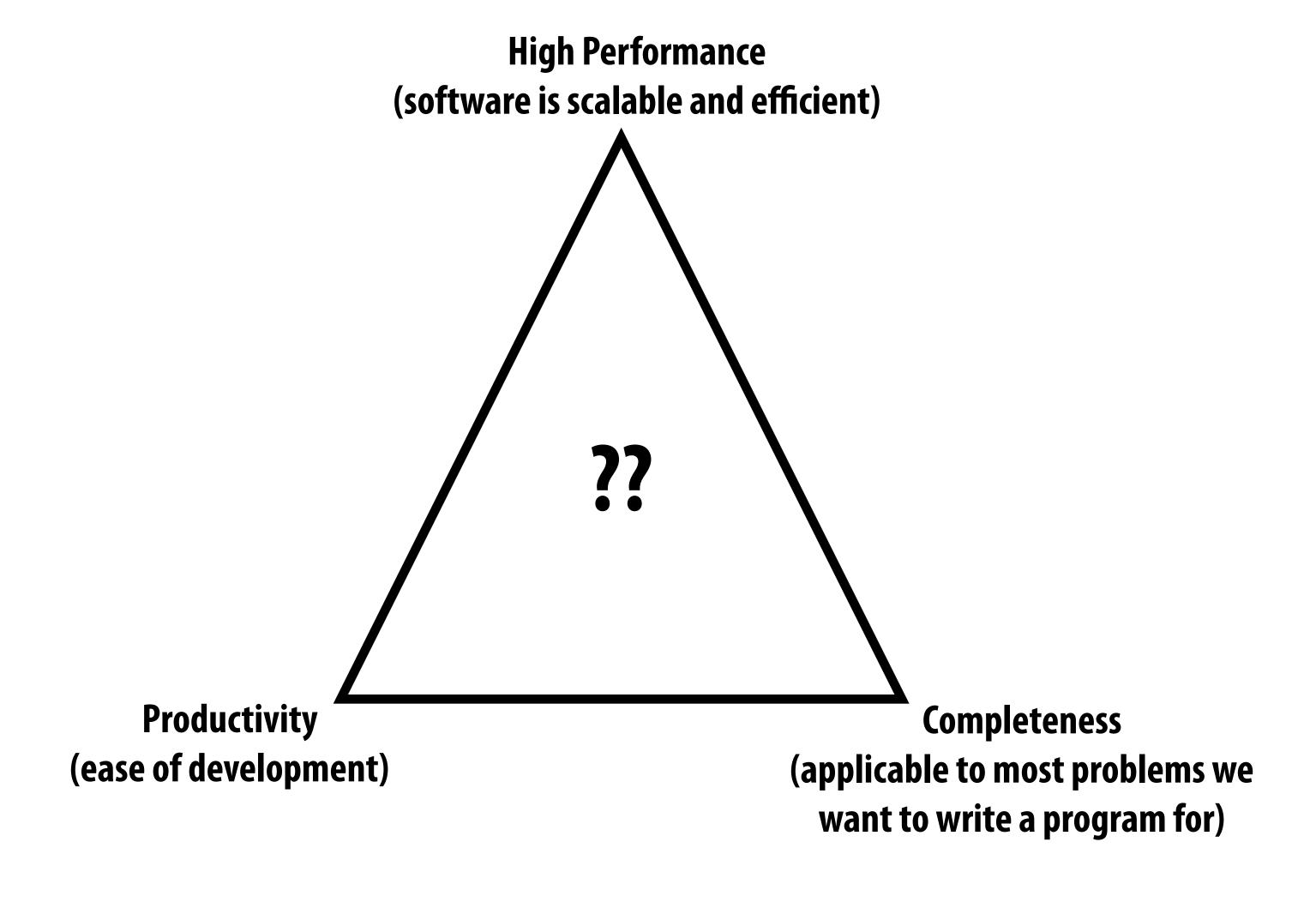
## Variety of programming models to abstract HW

- Machines with very different performance characteristics
- Worse: different technologies and performance characteristics within the same machine at different scales
  - Within a core: SIMD, multi-threading: fine grained sync and comm
    - Abstractions: SPMD programming (ISPC, Cuda, OpenCL, Metal, Renderscript)
  - Across cores: coherent shared memory via fast on-chip network
    - Abstractions: OpenMP pragma, Cilk, TBB
  - Hybrid CPU+GPU multi-core: incoherent (potentially) shared memory
    - Abstractions: OpenCL
  - Across racks: distributed memory, multi-stage network
    - Abstractions: message passing (MPI, Go, Spark, Legion, Charm++)

## This is a huge challenge

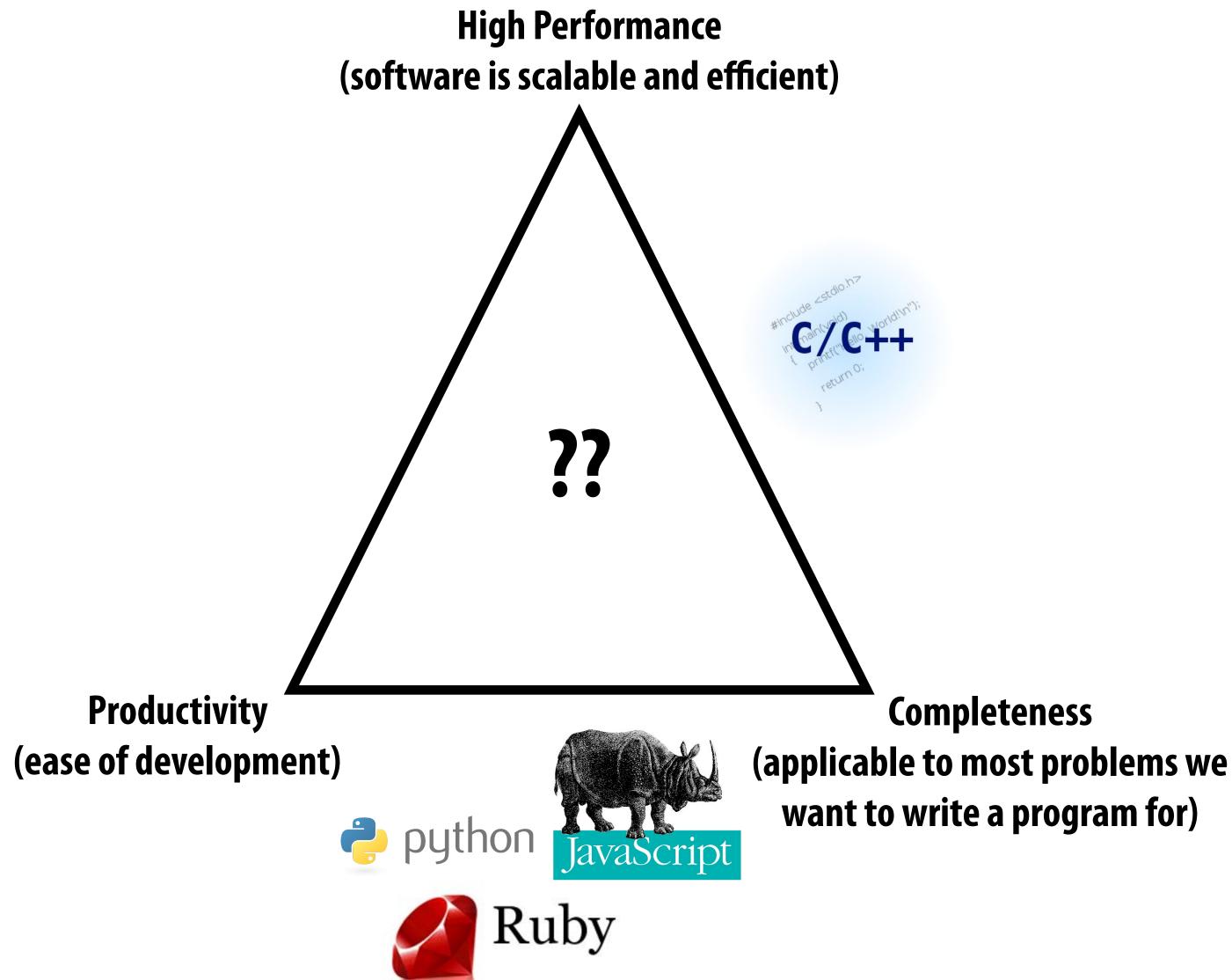
- Machines with very different performance characteristics
- Worse: different performance characteristics within the same machine at different scales
- To be efficient, software must be optimized for HW characteristics
  - Difficult even in the case of one level of one machine
  - Combinatorial complexity of optimizations when considering a complex machine, or different machines
  - Loss of software portability

## The [magical] ideal parallel programming language



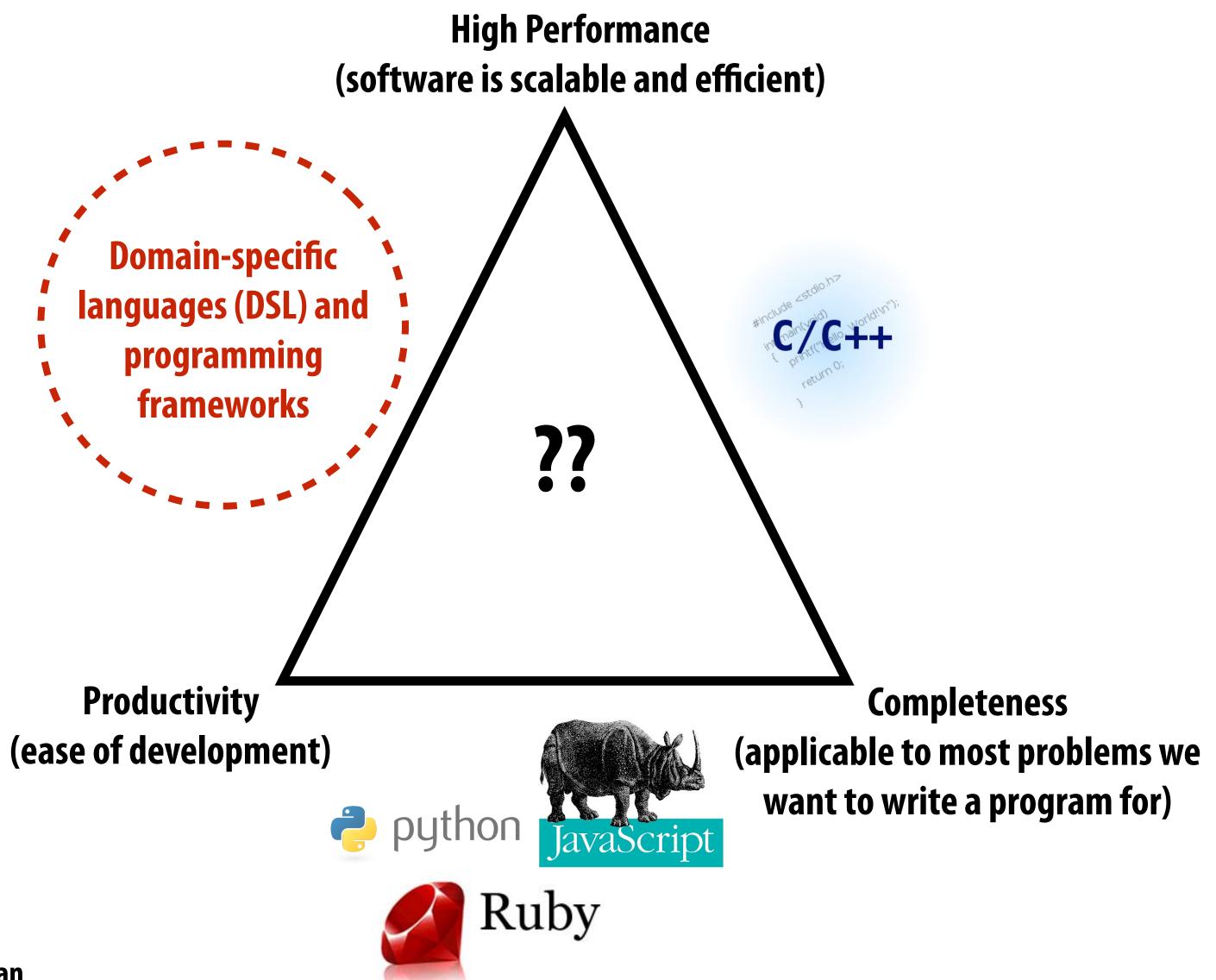
## Successful programming languages

Here: definition of success = widely used



## Growing interest in domain-specific programming systems

To realize high performance and productivity: willing to sacrifice completeness



## Domain-specific programming systems

- Main idea: raise level of abstraction for expressing programs
- Introduce high-level programming primitives specific to an application domain
  - <u>Productive</u>: intuitive to use, portable across machines, primitives correspond to behaviors frequently used to solve problems in targeted domain
  - <u>Performant</u>: system uses domain knowledge to provide efficient, optimized implementation(s)
    - Given a machine: system knows what algorithms to use, parallelization strategies to employ for this domain
    - Optimization goes beyond efficient mapping of software to hardware! The hardware platform itself can be optimized to the abstractions as well
- Cost: loss of generality/completeness

## Two domain-specific programming examples

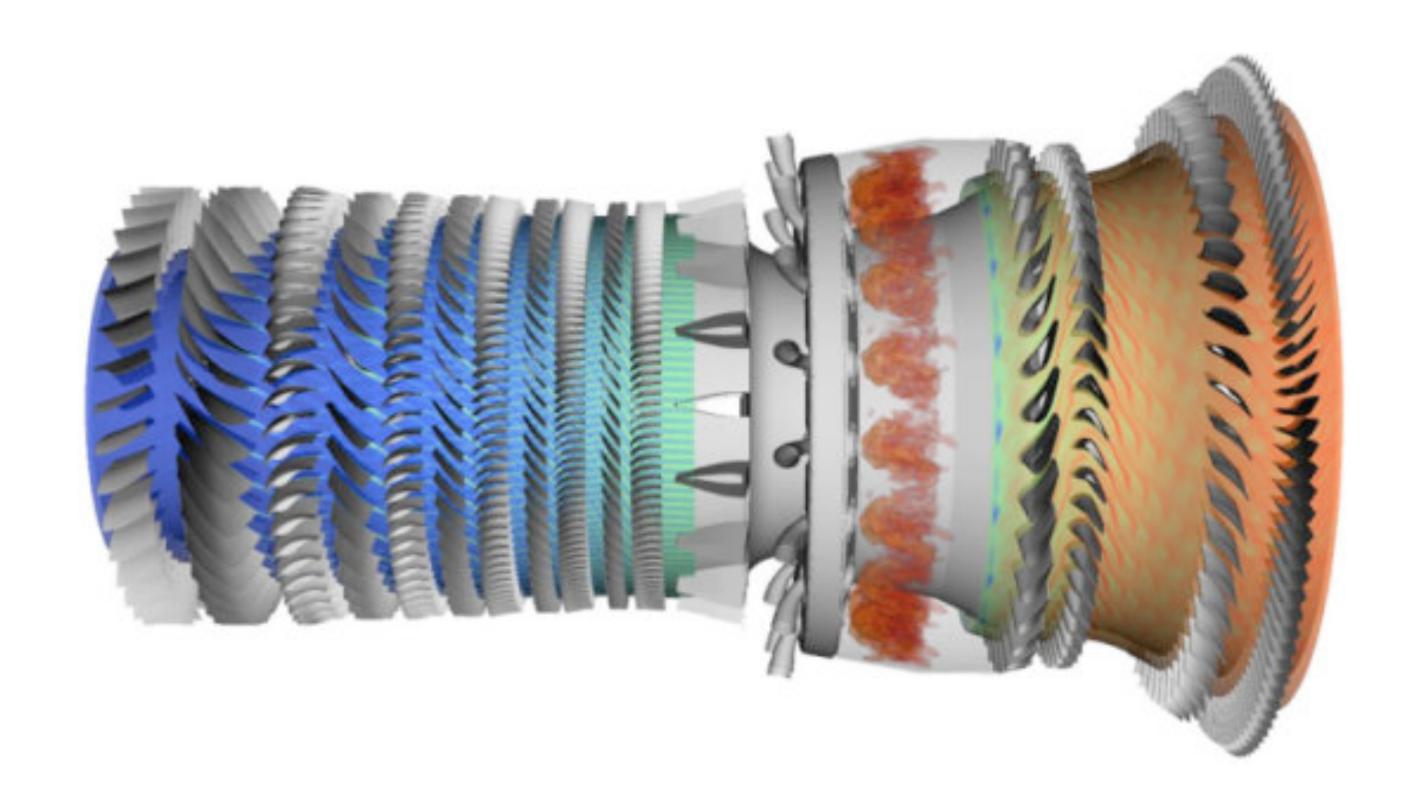
- 1. Liszt: for scientific computing on meshes
- 2. Halide: for image processing

What are other domain specific languages? (SQL is another good example)

## Example 1:

## Lizst: a language for solving PDE's on meshes

[DeVito et al. Supercomputing 11, SciDac '11]



Slide credit for this section of lecture: Pat Hanrahan and Zach Devito (Stanford)

## What a Liszt program does

A Liszt program is run on a mesh

A Liszt program defines, and compute the value of, fields defined on the mesh

```
Position is a field defined at each mesh vertex.

The field's value is represented by a 3-vector.

val Position = FieldWithConst[Vertex,Float3](0.f, 0.f, 0.f)

val Temperature = FieldWithConst[Vertex,Float](0.f)

val Flux = FieldWithConst[Vertex,Float](0.f)

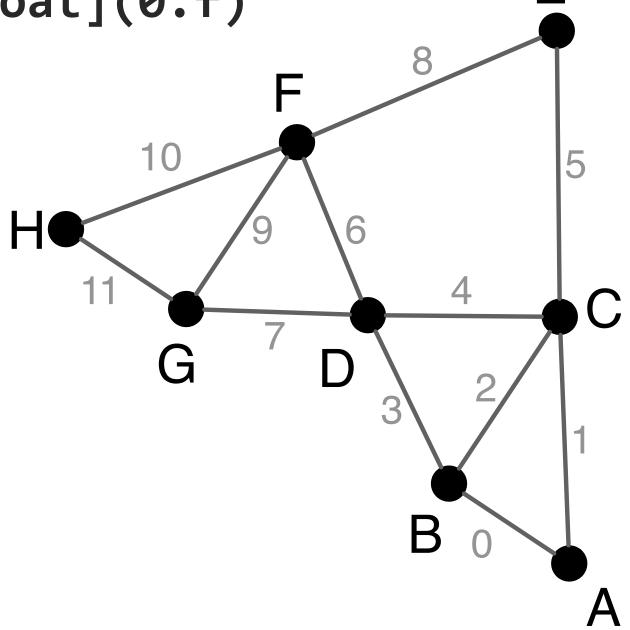
val JacobiStep = FieldWithConst[Vertex,Float](0.f)
```

#### **Color key:**

Fields
Mesh entity

#### **Notes:**

Fields are a higher-kinded type (special function that maps a type to a new type)



## Liszt program: heat conduction on mesh

#### Program computes the value of fields defined on meshes

```
Set flux for all vertices to 0.f;
                                                           Color key:
var i = 0;
while ( i < 1000 ) {
                                                            Fields
  Flux(vertices(mesh)) = 0.f;
                                                            Mesh
  JacobiStep(vertices(mesh)) = 0.f;
                                                            Topology functions
  for (e <- edges(mesh)) { ✓······ Independently, for each
                                                            Iteration over set
    val v1 = head(e)
                                       edge in the mesh
   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
                                              Access value of field
        Given edge, loop body accesses/modifies field
                                              at mesh vertex v2
        values at adjacent mesh vertices
```

## Liszt's topological operators

towards<sup>5</sup>(e : Edge, t : Vertex) : Edge

Used to access mesh elements relative to some input vertex, edge, face, etc. Topological operators are the <u>only way</u> to access mesh data in a Liszt program Notice how many operators return sets (e.g., "all edges of this face")

```
BoundarySet¹[ME <: MeshElement](name : String) : Set[ME]</pre>
vertices(e : Mesh) : Set[Vertex]
cells(e : Mesh) : Set[Cell]
edges(e : Mesh) : Set[Edge]
faces(e : Mesh) : Set[Face]
                                                          cells(e : Cell) : Set[Cell]
vertices(e : Vertex) : Set[Vertex]
                                                          vertices(e : Cell) : Set[Vertex]
cells(e : Vertex) : Set[Cell]
                                                          faces(e : Cell) : Set[Face]
edges(e : Vertex) : Set[Edge]
                                                          edges(e : Cell) : Set[Edge]
faces(e : Vertex) : Set[Face]
                                                          cells(e : Face) : Set[Cell]
vertices(e : Edge) : Set[Vertex]
                                                          edgesCCW<sup>2</sup>(e : Face) : Set[Edge]
facesCCW<sup>2</sup>(e : Edge) : Set[Face]
                                                          vertices(e : Face) : Set[Vertex]
cells(e : Edge) : Set[Cell]
                                                          inside<sup>3</sup>(e : Face) : Cell
head(e : Edge) : Vertex
                                                          outside<sup>3</sup>(e : Face) : Cell
tail(e : Edge) : Vertex
                                                          flip<sup>4</sup>(e : Face) : Face
flip<sup>4</sup>(e : Edge) : Edge
                                                          towards<sup>5</sup>(e : Face,t : Cell) : Face
```

## Liszt programming

- A Liszt program describes operations on fields of an abstract mesh representation
- Application specifies type of mesh (regular, irregular) and its topology
- Mesh representation is chosen by Liszt (not by the programmer)

Well, that's interesting. I write a program, and the compiler decides what data structure it should use based on what operations my code performs.

## Compiling to parallel computers

Recall challenges you have faced in your assignments

- 1. Identify parallelism
- 2. Identify data locality
- 3. Reason about required synchronization

Now consider how to automate this process in the Liszt compiler.

## Key: determining program dependencies

#### 1. Identify parallelism

- Absence of dependencies implies code can be executed in parallel

#### 2. Identify data locality

- Partition data based on dependencies (localize dependent computations for faster synchronization)

#### 3. Reason about required synchronization

- Synchronization is needed to respect dependencies (must wait until the values a computation depends on are known)

In general programs, compilers are unable to infer dependencies at global scale: a[f(i)] += b[i] (must execute f(i) to know if dependency exists across loop iterations i)

## Liszt is constrained to allow dependency analysis

Lizst infers "stencils": "stencil" = mesh elements accessed in an iteration of loop = dependencies for the iteration

Statically analyze code to find stencil of each top-level for loop

- Extract nested mesh element reads
- Extract field operations

```
Edge 6's read stencil is D and F
for (e <- edges(mesh)) {</pre>
  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
                                                         e in
  Flux(v2) -= dT*step
                                                      edges(mesh)
  JacobiStep(v1) += step
  JacobiStep(v2) += step
                                                                 tail(e)
                                                head(e)
                                             Read Position, Temperature
                                                              Read Position, Temperature
                                             Write Flux, JacobiStep
                                                              Write Flux, JacobiStep
```

## Restrict language for dependency analysis

#### Language restrictions:

Mesh elements are only accessed through built-in topological functions:

```
cells(mesh), ...
```

Single static assignment:

– Data in fields can only be accessed using mesh elements:

No recursive functions

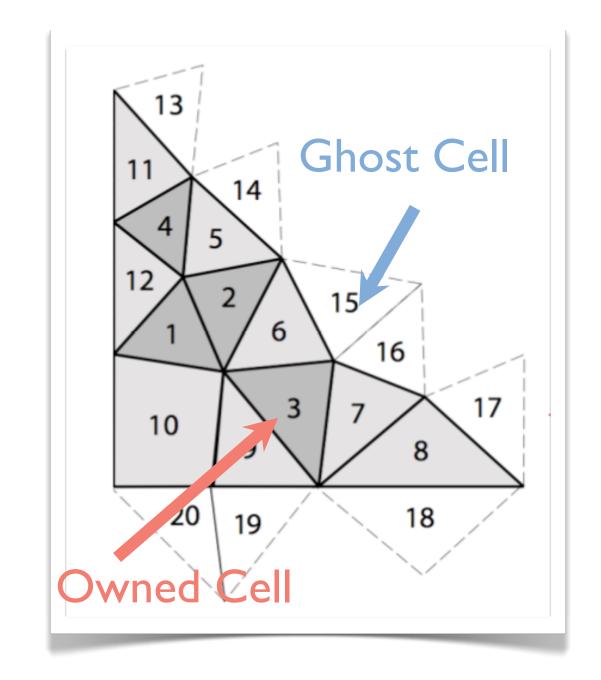
Restrictions allow compiler to automatically infer stencil for a loop iteration.

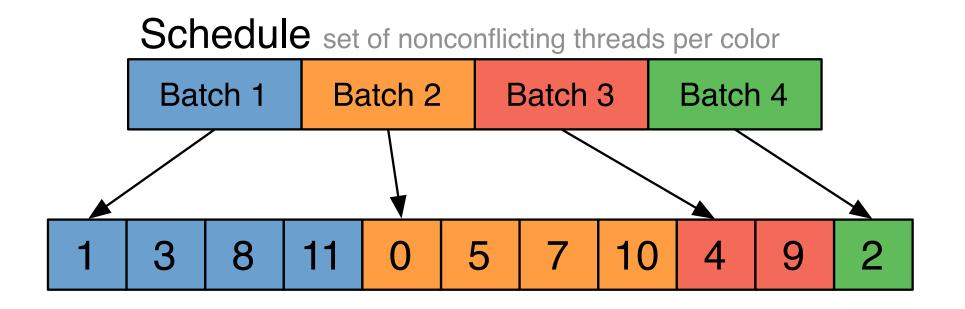
## Portable parallelism: use dependencies to implement different parallel execution strategies

I'll discuss two strategies...

Strategy 1: mesh partitioning

Strategy 2: mesh coloring

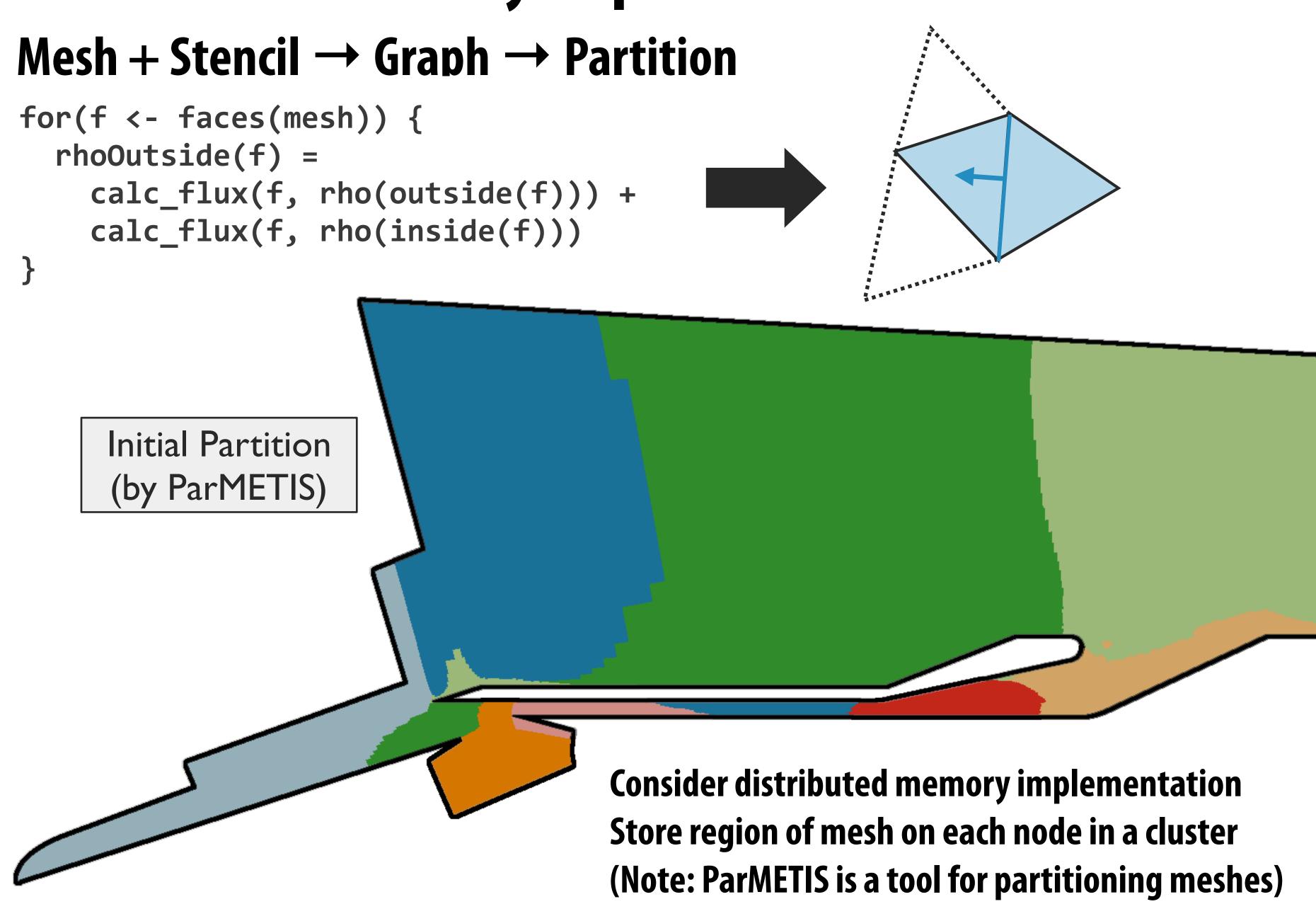


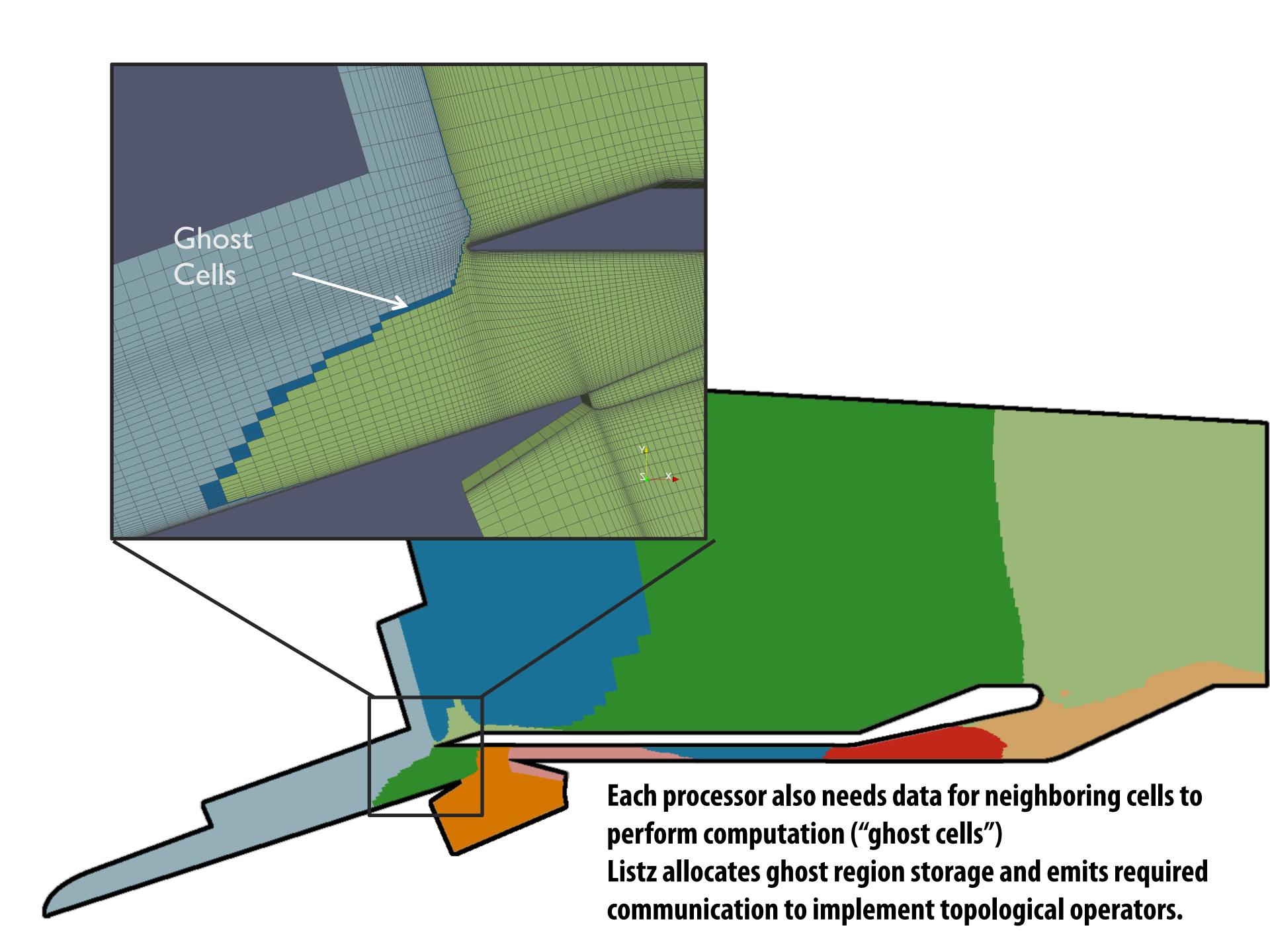


## Imagine compiling a Lizst program to the late days cluster (multiple nodes, distributed address space)

How might Liszt distribute a graph across these nodes?

## Distributed memory implementation of Liszt



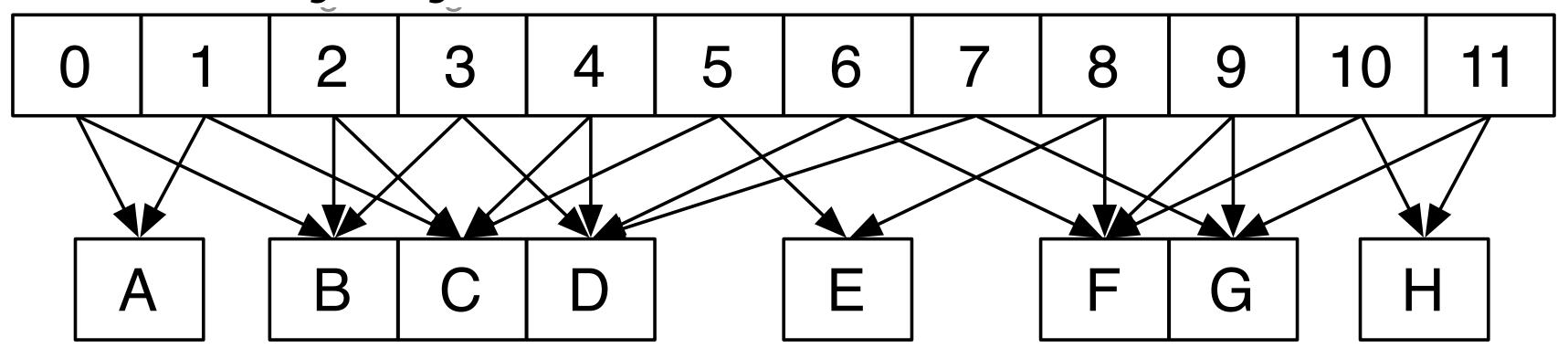


## Imagine compiling a Lizst program to a GPU (single address space, many tiny threads)

## GPU implementation: parallel reductions

In previous example, one region of mesh assigned per processor (or node in MPI cluster) On GPU, natural parallelization is one edge per CUDA thread

Threads (each edge assigned to 1 CUDA thread)

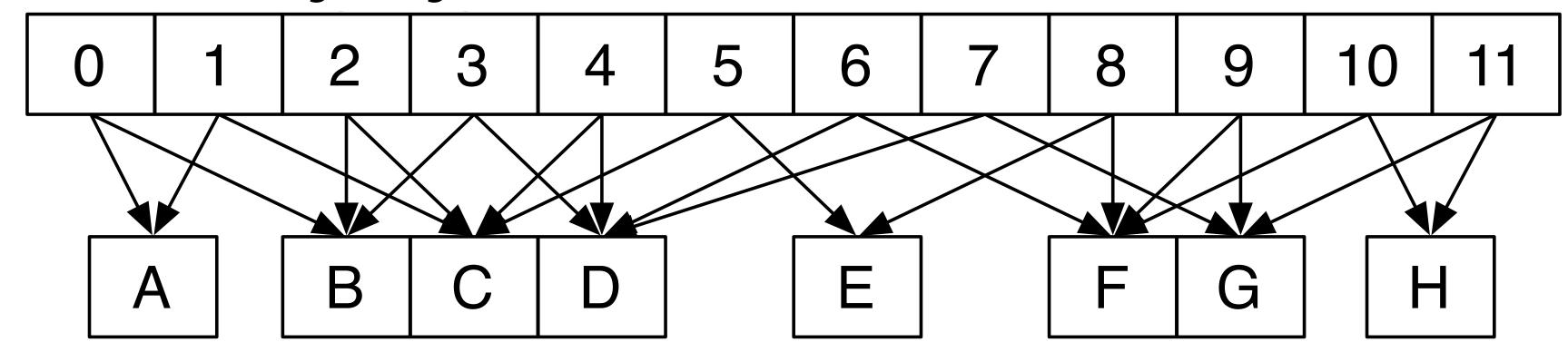


Flux field values (per vertex)

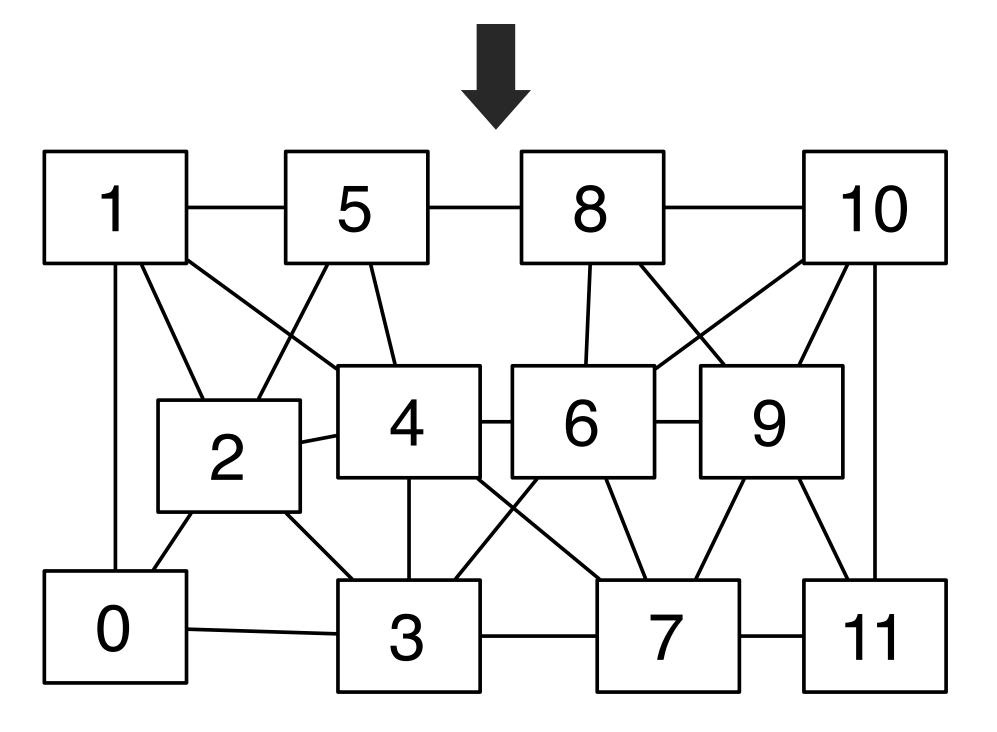
```
for (e <- edges(mesh)) {
    ...
Flux(v1) += dT*step
Flux(v2) -= dT*step
    atomic update of per-vertex field data
    ...
}</pre>
Different edges share a vertex: requires
    atomic update of per-vertex field data
```

## GPU implementation: conflict graph

Threads (each edge assigned to 1 CUDA thread)



Flux field values (per vertex)



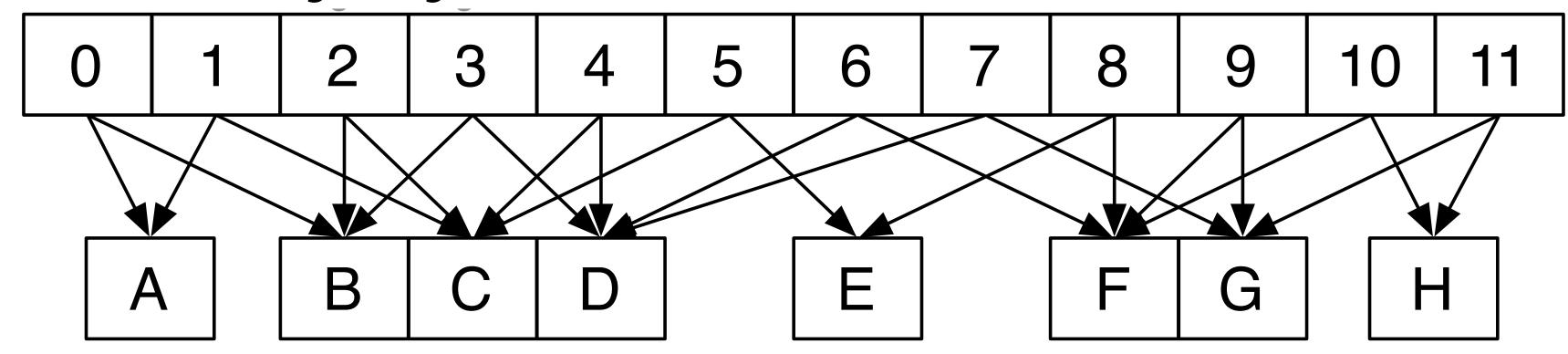
Identify mesh edges with colliding writes (lines in graph indicate presence of collision)

Can simply run program once to get this information.

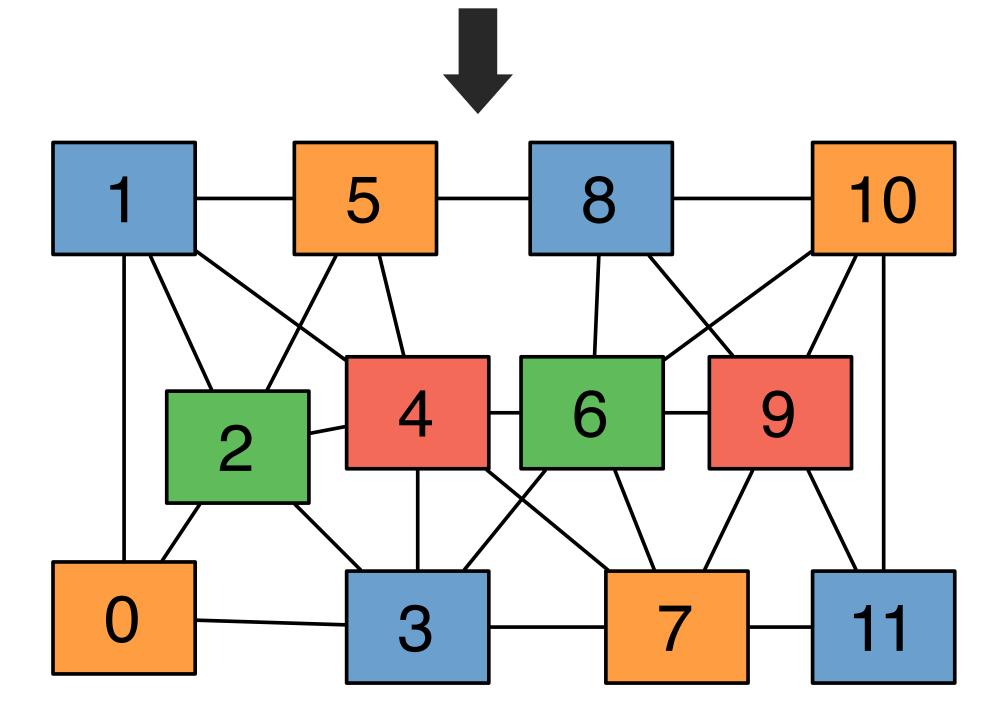
(results valid for subsequent executions provided mesh does not change)

## GPU implementation: conflict graph

Threads (each edge assigned to 1 CUDA thread)



Flux field values (per vertex)

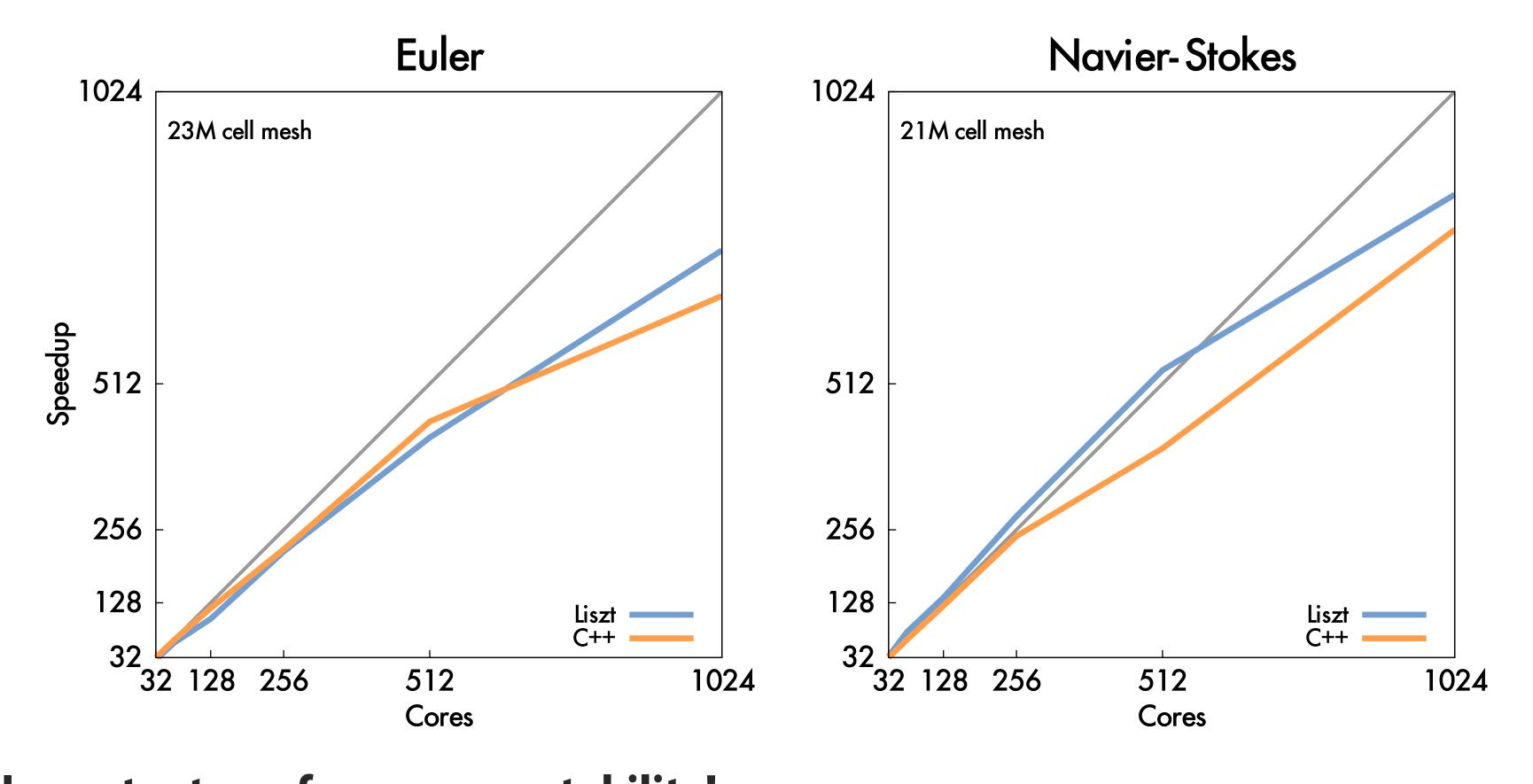


"Color" nodes in graph such that no connected nodes have the same color

Can execute on GPU in parallel, without atomic operations, by running all nodes with the same color in a single CUDA launch.

## Cluster performance of Lizst program

256 nodes, 8 cores per node (message-passing implemented using MPI)



Important: performance portability!
Same Liszt program also runs with high efficiency on GPU (results not shown here).
But uses a <u>different algorithm</u> when compiled to GPU! (graph coloring)

## Liszt summary

#### Productivity:

- Abstract representation of mesh: vertices, edges, faces, fields (concepts that a scientist thinks about already!)
- Intuitive topological operators

#### Portability

Same code runs on large cluster of CPUs (MPI) and GPUs (and combinations thereof!)

#### High-performance

- Language is constrained to allow compiler to track dependencies
- Used for locality-aware partitioning in distributed memory implementation
- Used for graph coloring in GPU implementation
- Compiler knows how to chooses different parallelization strategies for different platforms
- Underlying mesh representation can be customized by system based on usage and platform (e.g, don't store edge pointers if code doesn't need it, choose struct of arrays vs. array of structs for per-vertex fields)

## Example 2:

### Halide: a domain-specific language for image processing

Jonathan Ragan-Kelley, Andrew Adams et al. [SIGGRAPH 2012, PLDI 13]

## Halide used in practice

- Halide used to implement Android HDR+ app
- Halide code used to process all images uploaded to Google Photos



## A quick tutorial on high-performance image processing

### What does this C code do?

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];
float weights[] = \{1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.
                                                                                                                   1.0/9, 1.0/9, 1.0/9,
                                                                                                                   1.0/9, 1.0/9, 1.0/9};
for (int j=0; j<HEIGHT; j++) {</pre>
            for (int i=0; i<WIDTH; i++) {</pre>
                        float tmp = 0.f;
                        for (int jj=0; jj<3; jj++)
                                    for (int ii=0; ii<3; ii++)
                                                tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
                        output[j*WIDTH + i] = tmp;
```

## 3x3 box blur









(Zoom view)

## 3x3 image blur

```
int WIDTH = 1024;
                                                                                                                                                                                                                                     Total work per image = 9 x WIDTH x HEIGHT
int HEIGHT = 1024;
                                                                                                                                                                                                                                       For NxN filter: N<sup>2</sup> x WIDTH x HEIGHT
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];
float weights[] = \{1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.
                                                                                                    1.0/9, 1.0/9, 1.0/9,
                                                                                                     1.0/9, 1.0/9, 1.0/9};
for (int j=0; j<HEIGHT; j++) {</pre>
           for (int i=0; i<WIDTH; i++) {</pre>
                    float tmp = 0.f;
                    for (int jj=0; jj<3; jj++)
                                for (int ii=0; ii<3; ii++)
                                          tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
                    output[j*WIDTH + i] = tmp;
```

## Two-pass 3x3 blur

```
Total work per image = 6 x WIDTH x HEIGHT
int WIDTH = 1024;
int HEIGHT = 1024;
                                               For NxN filter: 2N x WIDTH x HEIGHT
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
                                               WIDTH x HEIGHT extra storage
float output[WIDTH * HEIGHT];
                                               3X lower arithmetic intensity than 3D blur
float weights[] = \{1.0/3, 1.0/3, 1.0/3\};
                                                                                        input
for (int j=0; j<(HEIGHT+2); j++)</pre>
                                                                                     (W+2)x(H+2)
  for (int i=0; i<WIDTH; i++) {</pre>
    float tmp = 0.f;
                                                             1D horizontal blur
    for (int ii=0; ii<3; ii++)
      tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
    tmp_buf[j*WIDTH + i] = tmp;
                                                                                      tmp_buf
                                                                                      W \times (H+2)
for (int j=0; j<HEIGHT; j++) {</pre>
  for (int i=0; i<WIDTH; i++) {</pre>
    float tmp = 0.f;
                                                             1D vertical blur
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
                                                                                       output
                                                                                       W x H
    output[j*WIDTH + i] = tmp;
```

#### Two-pass image blur: locality

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];
float weights[] = \{1.0/3, 1.0/3, 1.0/3\};
for (int j=0; j<(HEIGHT+2); j++)</pre>
  for (int i=0; i<WIDTH; i++) {</pre>
    float tmp = 0.f;
    for (int ii=0; ii<3; ii++)
      tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
    tmp_buf[j*WIDTH + i] = tmp;
for (int j=0; j<HEIGHT; j++) {</pre>
  for (int i=0; i<WIDTH; i++)</pre>
    float tmp = 0.f;
    for (int jj=0; jj<3; // j++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
```

Intrinsic bandwidth requirements of algorithm:
Application must read each element of input image and must write each element of output image.

Data from input reused three times. (immediately reused in next two i-loop iterations after first load, never loaded again.)

- Perfect cache behavior: never load required data more than once
- Perfect use of cache lines (don't load unnecessary data into cache)

Two pass: loads/stores to tmp\_buf are overhead (this memory traffic is an artifact of the two-pass implementation: it is not intrinsic to computation being performed)

Data from tmp\_buf reused three times (but three rows of image data are accessed in between)

- Never load required data more than once... if cache has capacity for three rows of image
- Perfect use of cache lines (don't load unnecessary data into cache)

#### Two-pass image blur, "chunked" (version 1)

```
int WIDTH = 1024;
int HEIGHT = 1024;
                                                                                         input
float input[(WIDTH+2) * (HEIGHT+2)];
                                                                                      (W+2)x(H+2)
                                                          Only 3 rows of intermediate
float tmp_buf[WIDTH * 3]; 
                                                          buffer need to be allocated
float output[WIDTH * HEIGHT];
float weights[] = \{1.0/3, 1.0/3, 1.0/3\};
                                                                                        tmp_buf
                                                                                                    (Wx3)
for (int j=0; j<HEIGHT; j++) {</pre>
                                                          Produce 3 rows of tmp_buf
  for (int j2=0; j2<3; j2++)
                                                          (only what's needed for one
                                                                                         output
    for (int i=0; i<WIDTH; i++) {</pre>
                                                                                         W \times H
                                                          row of output)
      float tmp = 0.f;
      for (int ii=0; ii<3; ii++)
                                            i+ii] * weights[ii];
        tmp += input[(j+j2)*(WIDTH+2) /
      tmp_buf[j2*WIDTH + i] = tmp;
                                                          Combine them together to get one row of output
  for (int i=0; i<WIDTH; i++) {</pre>
                                                          Total work per row of output:
    float tmp = 0.f;
                                                            - step 1: 3 x 3 x WIDTH work
    for (int jj=0; jj<3; jj++)
                                                            - step 2: 3 x WIDTH work
      tmp += tmp_buf[jj*WIDTH + i] * weights[jj];
                                                          Total work per image = 12 x WIDTH x HEIGHT ?????
    output[j*WIDTH + i] = tmp;
                                                          Loads from tmp_buffer are cached (assuming
                                                          tmp_buffer fits in cache)
```

CMU 15-418/618, Fall 2018

# Two-pass image blur, "chunked" (version 2)

```
int WIDTH = 1024;
int HEIGHT = 1024;
                                                         Sized to fit in cache
float input[(WIDTH+2) * (HEIGHT+2)];
                                                         (capture all producer-
                                                                                        input
float tmp_buf[WIDTH * (CHUNK_SIZE+2)];
                                                                                     (W+2)x(H+2)
                                                         consumer locality)
float output[WIDTH * HEIGHT];
                                                         Produce enough rows of
float weights[] = \{1.0/3, 1.0/3, 1.0/3\};
                                                                                       tmp_buf
                                                         tmp_buf to produce a
                                                                                             W x (CHUNK SIZE+2)
                                                         CHUNK_SIZE number of
for (int j=0; j<HEIGHT; j+CHUNK_SIZE) {</pre>
                                                         rows of output
  for (int j2=0; j2<CHUNK_SIZE+2; j2++)</pre>
    for (int i=0; i<WIDTH; i++) {</pre>
                                                                                       output
       float tmp = 0.f;
                                                                                        W \times H
       for (int ii=0; ii<3; ii++)
         tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
       tmp_buf[j2*WIDTH + i] = tmp;
                                                            Produce CHUNK_SIZE rows of output
  for (int j2=0; j2<CHUNK_SIZE; j2++)</pre>
    for (int i=0; i<WIDTH; i++) {</pre>
                                                                    Total work per chuck of output:
       float tmp = 0.f;
                                                                    (assume CHUNK SIZE = 16)
       for (int jj=0; jj<3; jj++)
                                                                      - Step 1: 18 x 3 x WIDTH work
         tmp += tmp_buf[(j2+jj)*WIDTH + i] * weights[jj];
                                                                      - Step 2: 16 x 3 x WIDTH work
                                                                    Total work per image: (34/16) x 3 x WIDTH x HEIGHT
       output[(j+j2)*WIDTH + i] = tmp;
                  Trends to ideal 6 x WIDTH x HEIGHT as CHUNK_SIZE is increased! ...... = 6.4 x WIDTH x HEIGHT
                                                                                               CMU 15-418/618, Fall 2018
```

## Conflicting goals (once again...)

- Want to be work efficient (perform fewer operations)
- Want to take advantage of locality when present
  - Otherwise work-efficient code will be bandwidth bound
  - Ideally: bandwidth cost of implementation is very close to intrinsic cost of algorithm: data is loaded from memory once and reused as much as needed prior to being discarded from processor's cache
- Want to execute in parallel (multi-core, SIMD within core)

# Optimized C++ code: 3x3 image blur

Good: 10x faster: on a quad-core CPU than my original two-pass code Bad: specific to SSE (not AVX2), CPU-code only, hard to tell what is going on at all!

```
void fast_blur(const Image &in, Image &blurred) {
                                                                           Multi-core execution
 _{m128i} one_third = _{mm}_{set1}_{epi16(21846);}
                                                                           (partition image vertically)
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {</pre>
  _{m128i} a, b, c, sum, avg;
  _m128i tmp[(256/8)*(32+2)];
for (int xTile = 0; xTile < in width(); xTile += 256) {
                                                                           Modified iteration order:
   _m128i *tmpPtr = tmp;
                                                                           256x32 block-major iteration
   for (int y = -1; y < 32+1; y++)
                                                                           (to maximize cache hit rate)
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _{mm}loadu_si128((_{m}128i*)(inPtr-1));
     b = _mm_loadu_si128((_m128i*)(inPtr+1));
     c = _{mm}load_si128((_{m}128i*)(inPtr));
                                                                              use of SIMD vector intrinsics
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
                                                                              two passes fused into one:
    _m128i *outPtr = (_m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
                                                                              tmp data read from cache
     a = _{mm}load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
      mm store si128(outPtr++, avg);
}}}}
```

# Halide blur (algorithm description)

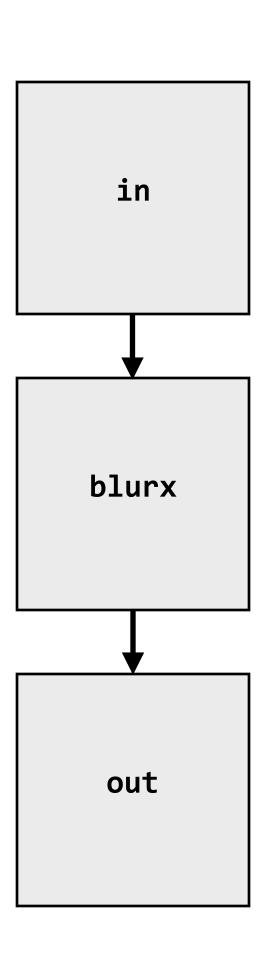
```
// Halide 3x3 blur program definition
                                                 Images are pure functions
Func halide_blur(Func in) {
                                                 Functions map integer coordinates (in up to a 4D domain)
                                                 to values (e.g., colors of corresponding pixels)
                                                 (in, blurx and out are functions)
  Func blurx, out;
  Var x, y;
                                                 Algorithms are a series of functions (think: pipeline stages)
  blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f; \leftarrow
  out(x,y) = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
  return out;
                                                                      Value of blurx at coordinate (x,y)
                                                                      is given by expression accessing
                                                                      three values of in
// top-level calling code
Image<uint8_t> input = load_image("myimage.png");
                                                                      // define input image
Func my_program = halide_blur(input);
                                                                      // define pipeline
Image<uint8_t> output = my_program.realize(input.width(), input.height(),
                                                input.channels()); // execute pipeline
output.save("myblurredimage.png");
```

NOTE: execution order and storage are unspecified by the abstraction. The implementation can evaluate, reevaluate, cache individual points as desired!

## Think of a Halide program as a pipeline

```
// Halide 3x3 blur program definition
Func halide_blur(Func in) {
   Func blurx, out;
   Var x, y;

   blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
   out(x,y) = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
   return out;
}
```



#### Halide schedule describes how to execute a pipeline

```
// Halide program definition
Func halide_blur(Func in) {
  Func blurx, out;
  Var x, y, xi, yi
  // the "algorithm description" (what to do)
  blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
  out(x,y) = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
  // "the schedule" (how to do it)
  out.tile(x, y, xi, yi, 256, 32).vectorize(xi,8).parallel(y);
                                                        When evaluating out, use 2D tiling order
                                                        (loops named by x, y, xi, yi).
  blurx.chunk(x).vectorize(x, 8);
                                                        Use tile size 256 x 32.
  return out;
                                                        Vectorize the xi loop (8-wide)
                                                        Use threads to parallelize the y loop
                                                        Produce only chunks of blurx at a time.
                                                        Vectorize the x (innermost) loop
```

#### Halide schedule describes how to execute a pipeline

```
// Halide program definition
Func halide_blur(Func in) {
   Func blurx, out;
   Var x, y, xi, yi

   // the "algorithm description" (what to do)
   blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
   out(x,y) = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

   // "the schedule" (how to do it)
   out.tile(x, y, xi, yi, 256, 32).vectorize(xi,8).parallel(y);
   blurx.chunk(x).vectorize(x, 8);
   return out;
}
```

Given a schedule, Halide carries out mechanical process of implementing the specified schedule

```
void halide_blur(uint8_t* in, uint8_t* out) {
   #pragma omp parallel for
   for (int y=0; y<HEIGHT; y+=32) {
                                        // tile loop
      for (int x=0; y<WIDTH; x+=256) { // tile loop
         // buffer
         uint8_t* blurx[34 * 256];
         // produce intermediate buffer
         for (int yi=0; yi<34; yi++) {
            // SIMD vectorize this loop (not shown)
            for (int xi=0; xi<256; xi++) {
               blurx[yi*256+xi] =
                   (in[(y+yi-1)*WIDTH+x+xi-1] +
                   in[(y+yi-1)*WIDTH+x+xi] +
                   in[(y+yi-1)*WIDTH+x+xi+1]) / 3.0;
         // consumer intermediate buffer
        for (int yi=0; yi<32; yi++) {
            // SIMD vectorize this loop (not shown)
            for (int xi=0; xi<256; xi++) {
               out[(y+yi)*256+(x+xi)] =
                   (blurx[yi*256+xi] +
                   blurx[(yi+1)*256+xi] +
                   blurx[(yi+2)*256+xi]) / 3.0;
     } // loop over tiles
   } // loop over tiles
```

## Halide: two domain-specific co-languages

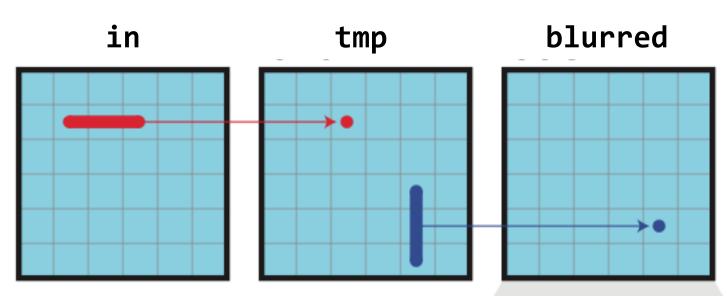
- Functional language for describing image processing operations
- Domain-specific language for describing schedules
- <u>Design principle</u>: separate "algorithm specification" from its schedule
  - Programmer's responsibility: provide a high-performance schedule
  - Compiler's responsibility: carry out mechanical process of generating threads, SIMD instructions, managing buffers, etc.
  - Result: enable programmer to rapidly explore space of schedules
    - (e.g., "tile these loops", "vectorize this loop", "parallelize this loop across cores")

#### Domain scope:

- All computation on regular N-D coordinate spaces
- Only feed-forward pipelines (includes special support for reductions and fixed recursion depth)
- All dependencies inferable by compiler

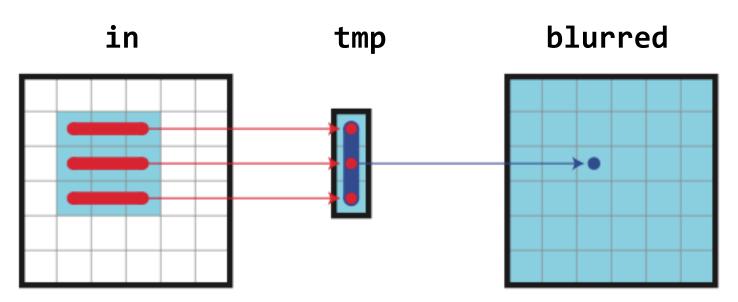
#### Producer/consumer scheduling primitives

#### Four basic scheduling primitives shown below



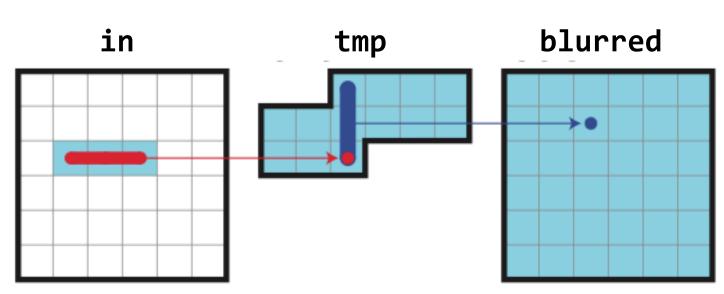
breadth first: each function is entirely evaluated before the next one.

"Root"



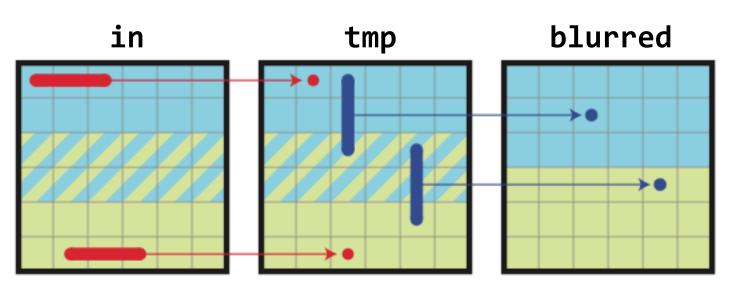
total fusion: values are computed on the fly each time that they are needed.

"Inline"



sliding window: values are computed when needed then stored until not useful anymore.

"Sliding Window"



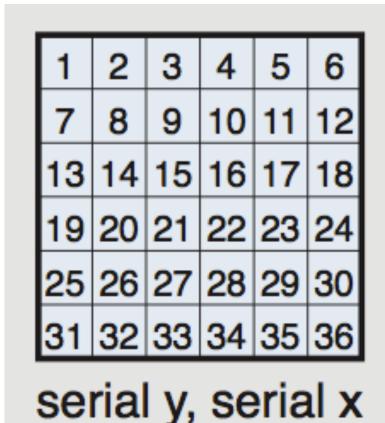
tiles: overlapping regions are processed in parallel, functions are evaluated one after another.

"Chunked"

# Producer/consumer scheduling primitives

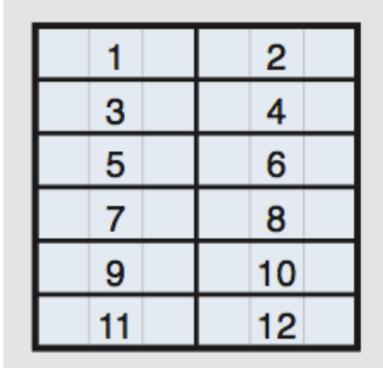
```
// Halide program definition
                                                                            void halide_blur(uint8_t* in, uint8_t* out) {
                                 "Root":
Func halide_blur(Func in) {
                                                                                uint8_t blurx[WIDTH * HEIGHT];
                                 compute all points of the producer,
  Func blurx, out;
                                                                                for (int y=0; y<HEIGHT; y++) {</pre>
                                 then run consumer (minimal locality)
  Var x, y, xi, yi
                                                                                  for (int x=0; y<WIDTH; x++) {</pre>
                                                                                      blurx[] = ...
  // the "algorithm description" (what to do)
  blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
                                                                                for (int y=0; y<HEIGHT; y++) {</pre>
  out(x,y) = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
                                                                                  for (int x=0; y<WIDTH; x++) {</pre>
                                                                                      out[] = ...
  // "the schedule" (how to do it)
                                                                            }
  blurx.compute_at(ROOT);
  return out;
                                 "Inline":
// Halide program definition
                                                                          void halide_blur(uint8_t* in, uint8_t* out) {
                                 revaluate producer at every use site
Func halide_blur(Func in) {
                                                                              for (int y=0; y<HEIGHT; y++) {</pre>
                                 in consumer (maximal locality)
                                                                                for (int x=0; y<WIDTH; x++) {</pre>
 Func blurx, out;
                                                                                    out[] = (((in[(y-1)*WIDTH+x-1] +
 Var x, y, xi, yi
                                                                                               in[(y-1)*WIDTH+x] +
                                                                                               in[(y-1)*WIDTH+x+1]) / 3) +
 // the "algorithm description" (what to do)
                                                                                             ((in[y*WIDTH+x-1] +
 blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
                                                                                               in[y*WIDTH+x] +
 out(x,y) = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
                                                                                               in[y*WIDTH+x+1]) / 3) +
                                                                                             ((in[(y+1)*WIDTH+x-1] +
  // "the schedule" (how to do it)
                                                                                               in[(y+1)*WIDTH+x] +
 blurx.inline();
                                                                                               in[(y+1)*WIDTH+x+1]) / 3));
 return out;
```

#### Domain iteration primitives

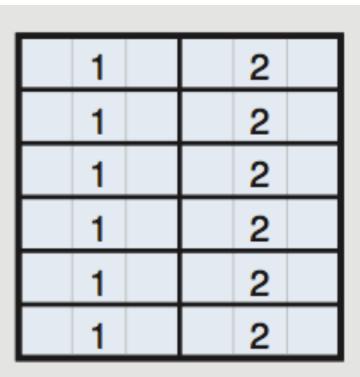


1	7	13	19	25	31		
2	8	14	20	26	32		
3	9	15	21	27	33		
4	10	16	22	28	34		
5	11	17	23	29	35		
6	12	18	24	30	36		
serial x, serial y							

Specify both order and how to parallelize (multi-thread, SIMD vector)



serial y vectorized x



parallel y vectorized x

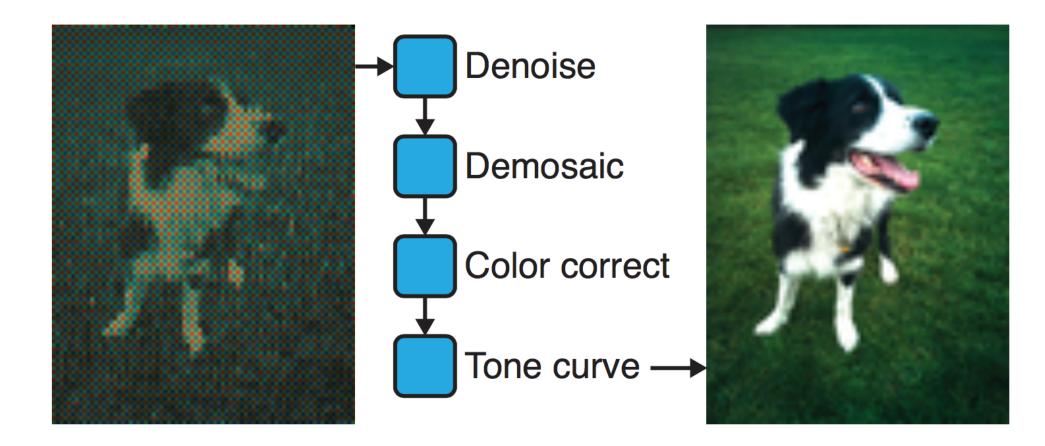
1	2	5	6	9	10
3	4	7	8	11	12
13	14	17	18	21	22
15	16	19	20	23	24
25	26	29	30	33	34
27	28	31	32	35	36

split x into  $2x_o + x_i$ , split y into  $2y_o + y_i$ , serial  $y_o$ ,  $x_o$ ,  $y_i$ ,  $x_i$ 

2D blocked iteration order

#### **Example Halide results**

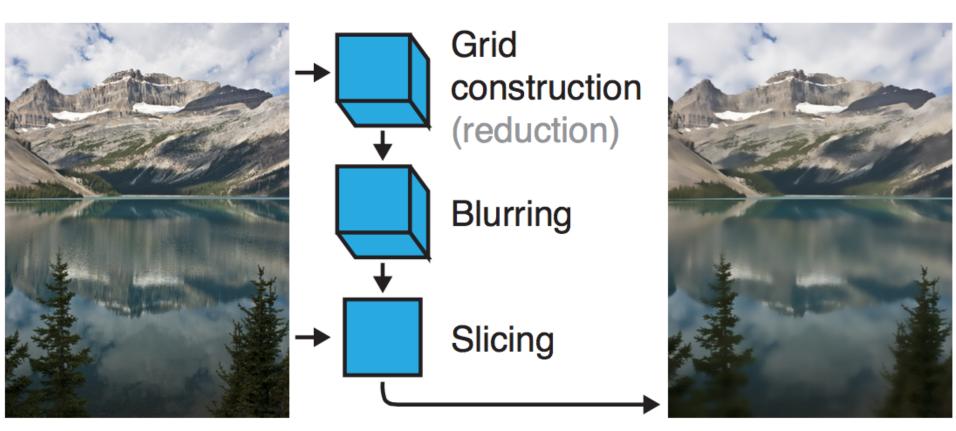
- Camera RAW processing pipeline (Convert RAW sensor data to RGB image)
  - Original: 463 lines of hand-tuned ARM NEON assembly
  - Halide: 2.75x less code, 5% faster



Bilateral filter

(Common image filtering operation used in many applications)

- Original 122 lines of C++
- Halide: 34 lines algorithm + 6 lines schedule
  - CPU implementation: 5.9x faster
  - GPU implementation: 2x faster than hand-written CUDA



#### Stepping back: what is Halide?

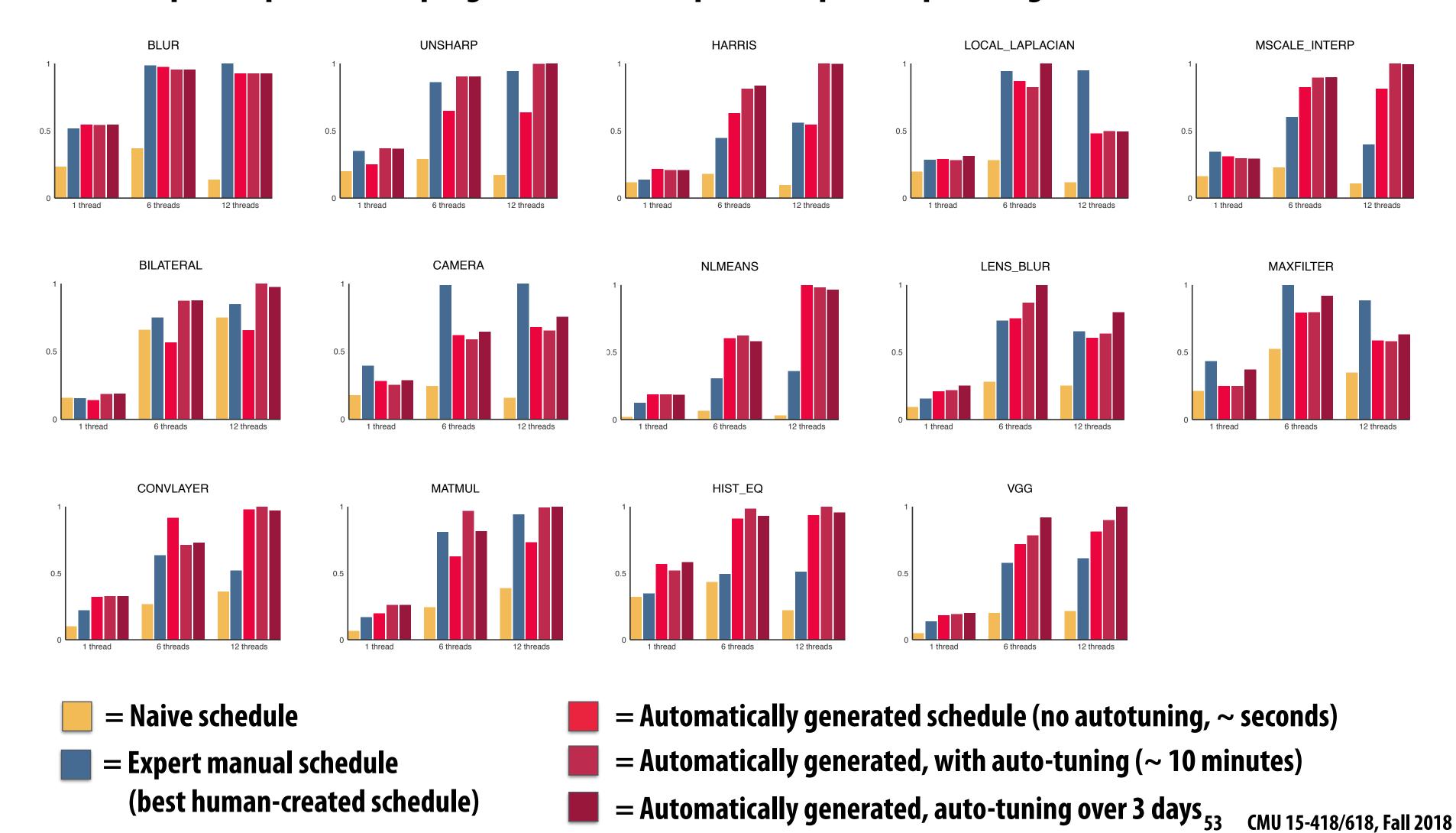
- Halide is a DSL for helping good developers optimize image processing code more rapidly
  - Halide doesn't decide how to optimize a program for a novice programmer
  - Halide provides primitives for a programmer (that has strong knowledge of code optimization, such as a 418 student) to rapidly express what optimizations the system should apply
  - Halide carries out the nitty-gritty of mapping that strategy to a machine

#### Automatically generating Halide schedules

[Mullapudi 2016]

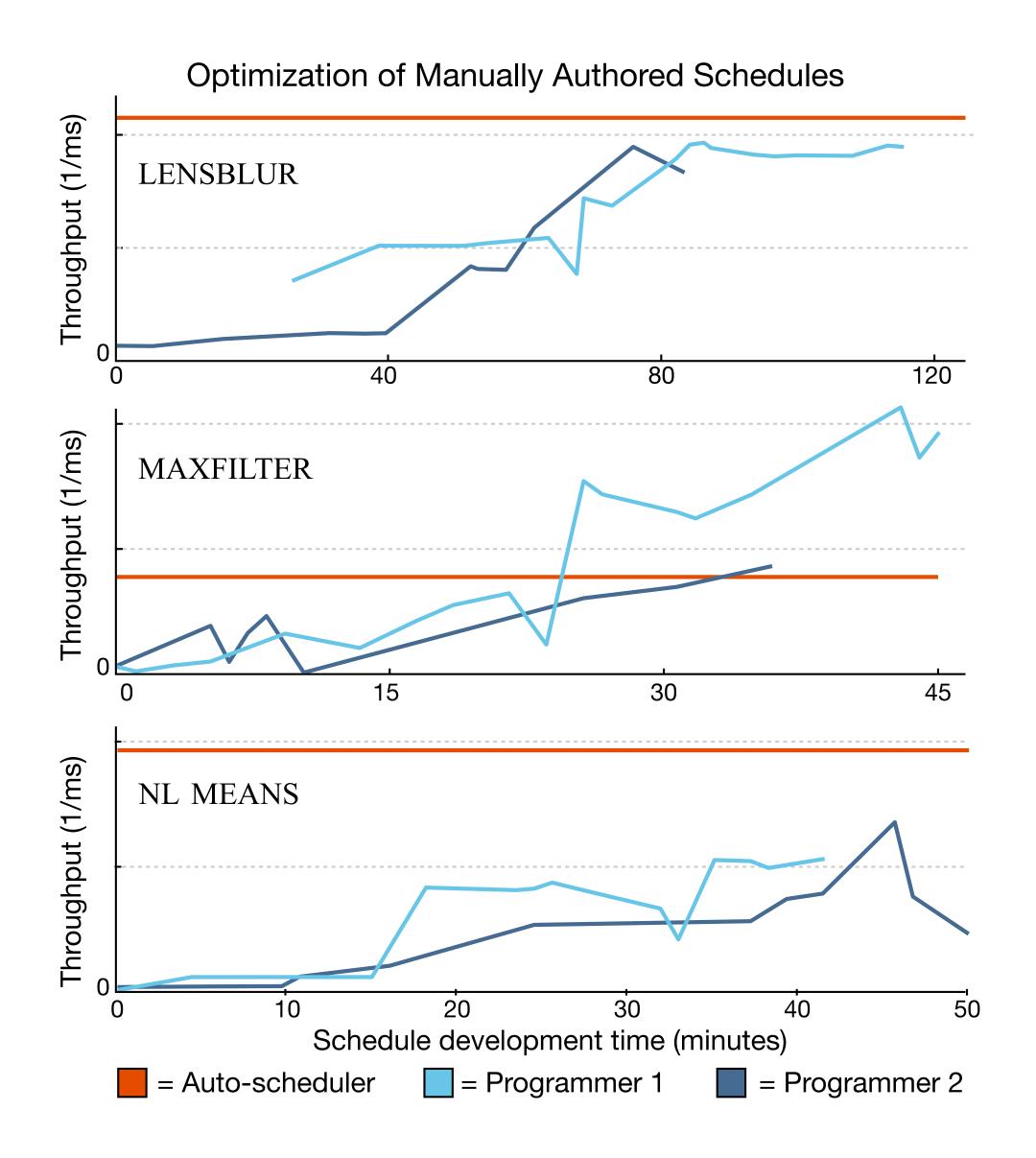
#### Extend Halide compiler to automatically generate schedule for programmer

- Compiler input: Halide program + size of expected input/output images



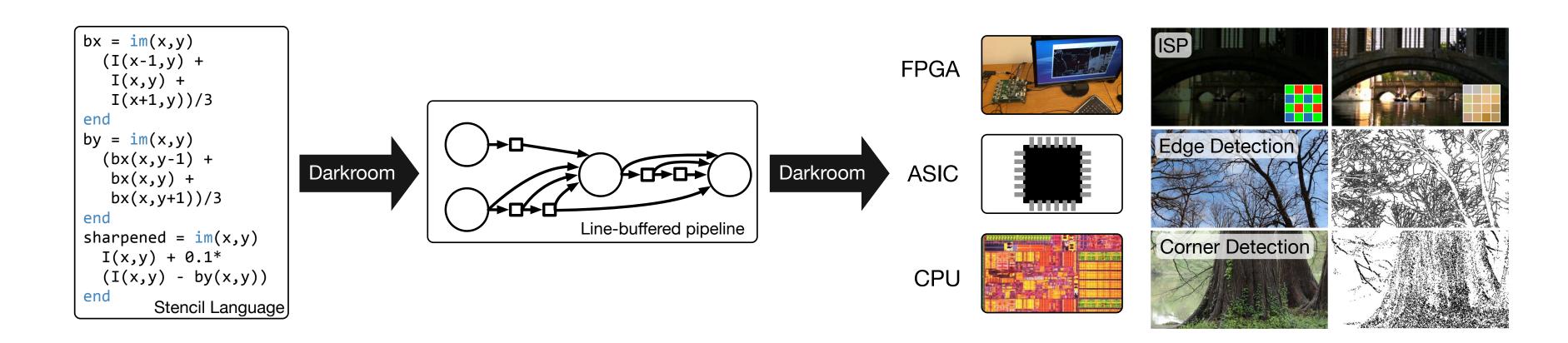
## "Racing" top Halide programmers

Halide auto-scheduler produced schedules that were better than those of expert Google Halide programmers in two of three cases (it got beat in one!)



# Darkroom/Rigel

Directly synthesize FGPA implementation of image processing pipeline from a high-level description (a constrained "Halide-like" language)



Goal: ultra high efficiency image processing

#### Many other recent domain-specific programming systems



Less domain specific than examples given today, but still designed specifically for: data-parallel computations on big data for distributed systems ("Map-Reduce")



DSL for graph-based machine learning computations

Also see Green-Marl, Ligra (DSLs for describing operations on graphs)



Model-view-controller paradigm for web-applications

Ongoing efforts in many domains...

Simit: a language for physical simulation [MIT]

#### Domain-specific programming system development

#### Can develop DSL as a stand-alone language

- Graphics shading languages
- MATLAB, SQL

#### "Embed" DSL in an existing generic language

- e.g., C++ library (GraphLab, OpenGL host-side API, Map-Reduce)
- Lizst syntax above was all valid Scala code

#### Active research idea:

- Design generic languages that have facilities that assist rapid embedding of new domain-specific languages
- "What is a good language for rapidly making new DSLs?"

#### Summary

- Modern machines: parallel and heterogeneous
  - Only way to increase compute capability in energy-constrained world
- Most software uses small fraction of peak capability of machine
  - Very challenging to tune programs to these machines
  - Tuning efforts are not portable across machines
- Domain-specific programming environments trade-off generality to achieve productivity, performance, and portability
  - Case studies today: Liszt, Halide
  - Common trait: languages provide abstractions that make dependencies known
    - Understanding dependencies is necessary but not sufficient: need domain restrictions and domain knowledge for system to synthesize efficient implementations