Lecture 20:

Domain-Specific Programming Systems

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2017

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

Tunes

Joss Stone

Less is More

(iii)

“Good DSL design is about identifying the right set of programming primitives, that together, can be composed to describe a useful set of tasks in a given domain.“

- Joss Stone

Course themes:

Designing computer systems that scale

(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 (last time)

Claim: most software uses modern hardware resources inefficiently

- Consider a piece of sequential C code
  - Let's call the performance of this code "baseline performance"

- Well-written sequential C code: ~ 5-10x faster

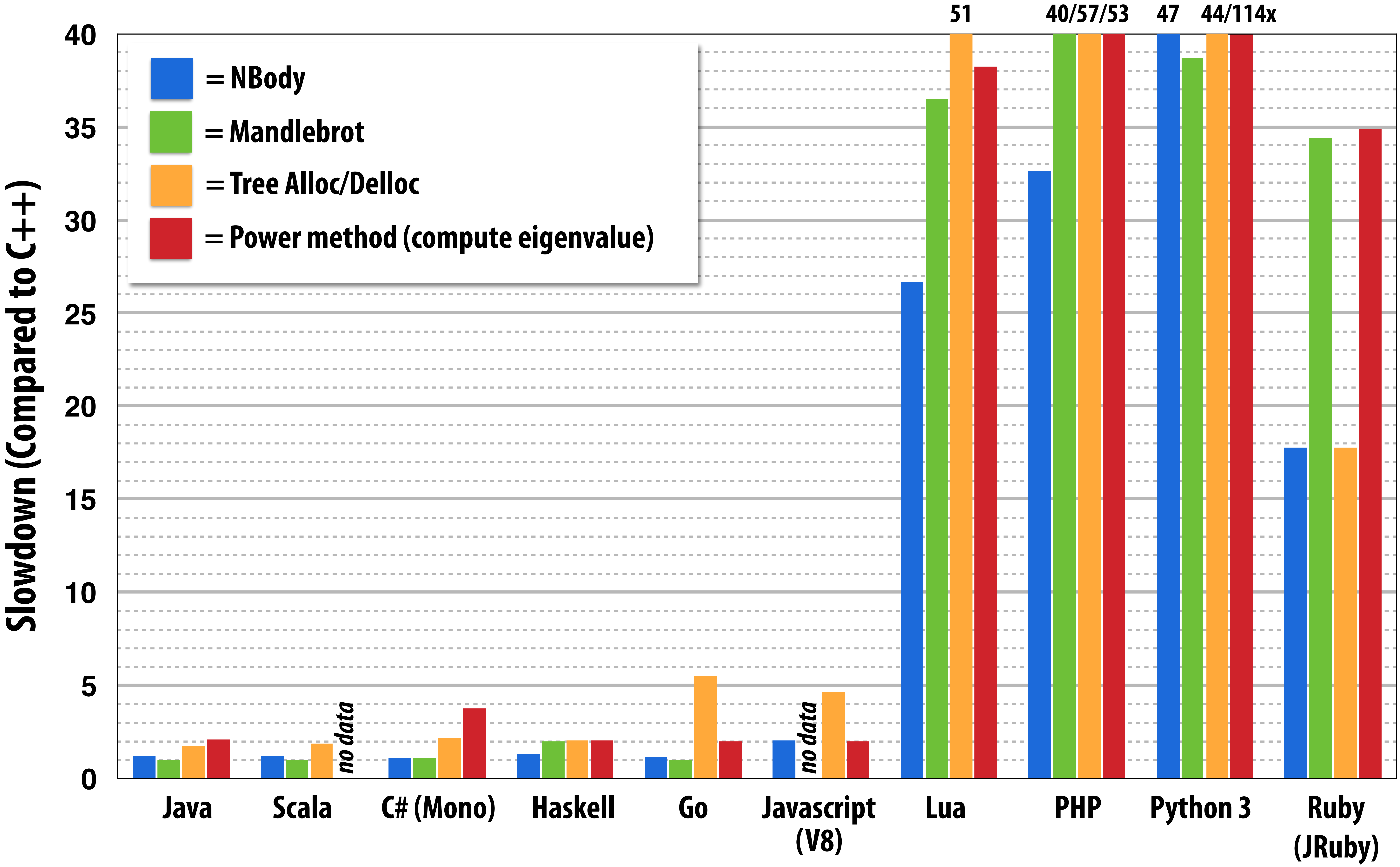
- Assembly language program: another small constant factor faster

- Java, Python, PHP, etc. ??

Credit: Pat Hanrahan

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

Recall: even good single-threaded C code is inefficient on a modern machine

Recall Assignment 1's Mandelbrot program

Consider execution on this laptop: quad-core, Intel Core i7, AVX...

Single core, with AVX vector instructions: 5.8x speedup over C code

Multi-core + hyper-threading + AVX instructions: ~30-40x speedup

Conclusion: basic C implementation compiled with -O3 leaves <u>a lot</u> of performance on the table

Making efficient use of modern parallel machines is challenging

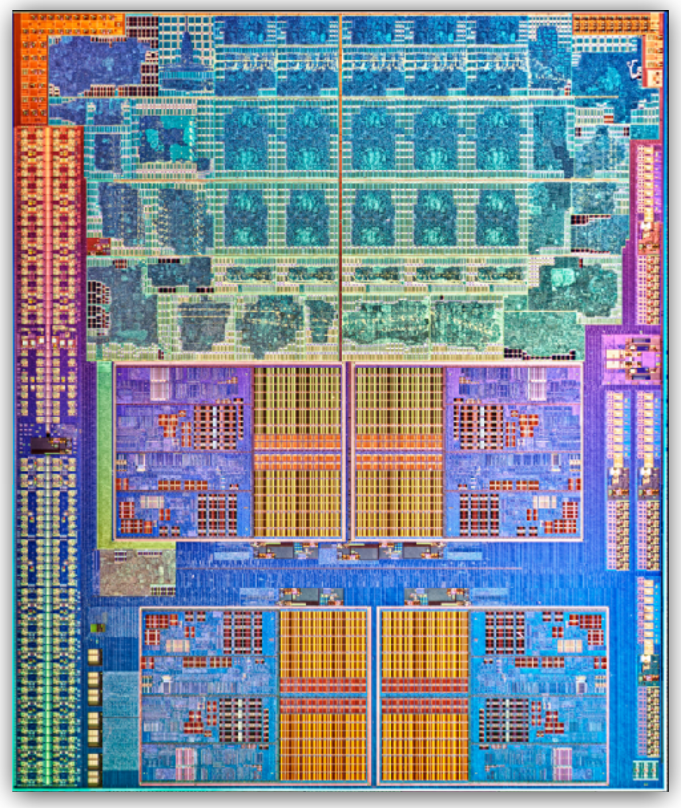
(proof by assignments 2, 3, and 4)

In our assignments you only programmed homogeneous parallel computers.
(and it was not particularly easy)

Assignment 2: GPU cores only
Assignment 3: multiple Xeon Phi CPUs
Assignment 4: multiple multi-core Xeon CPUs

Recall from last time: need for efficiency is motivating heterogeneous parallel platforms

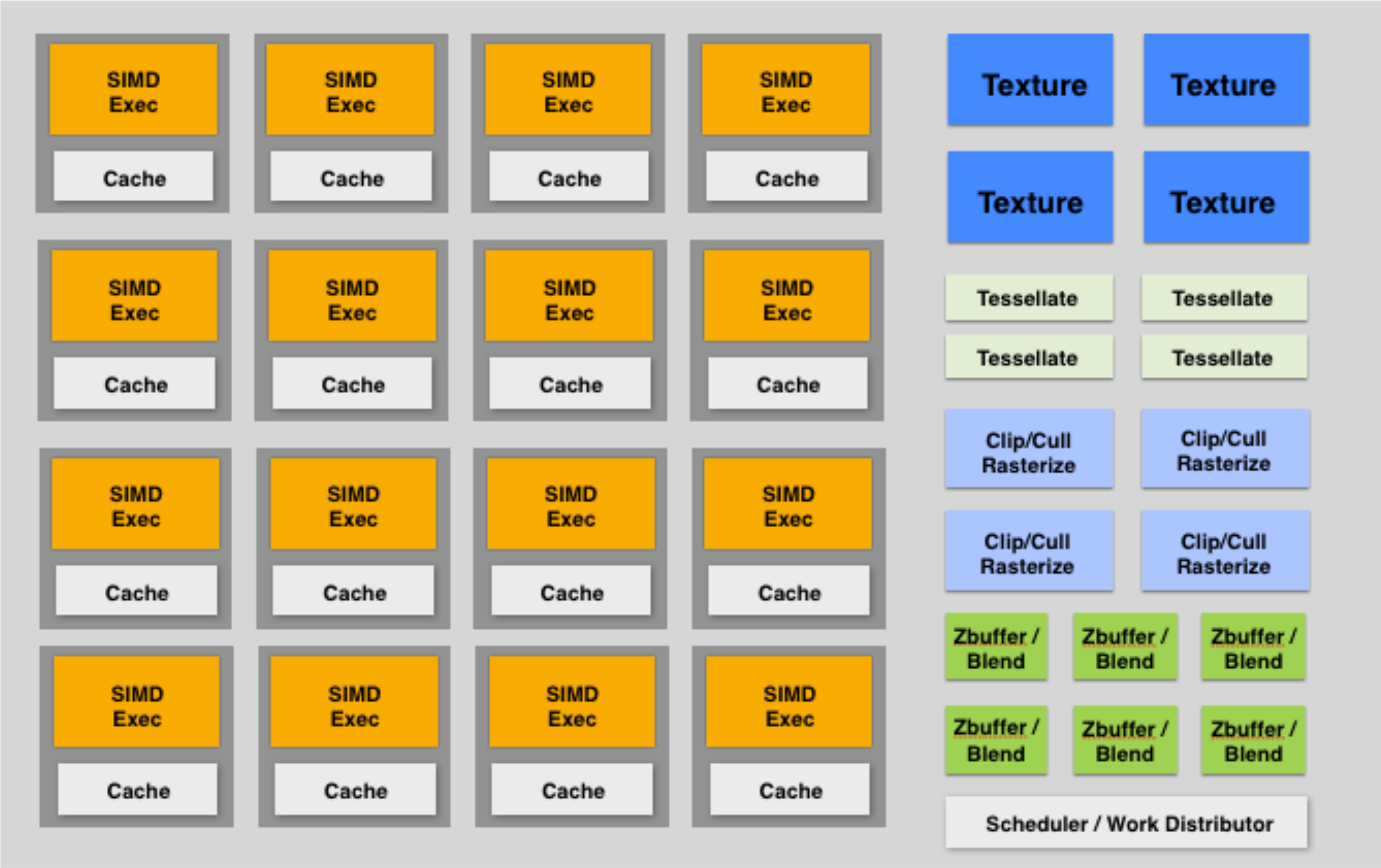
Integrated CPU + GPU



CPU+data-parallel accelerator

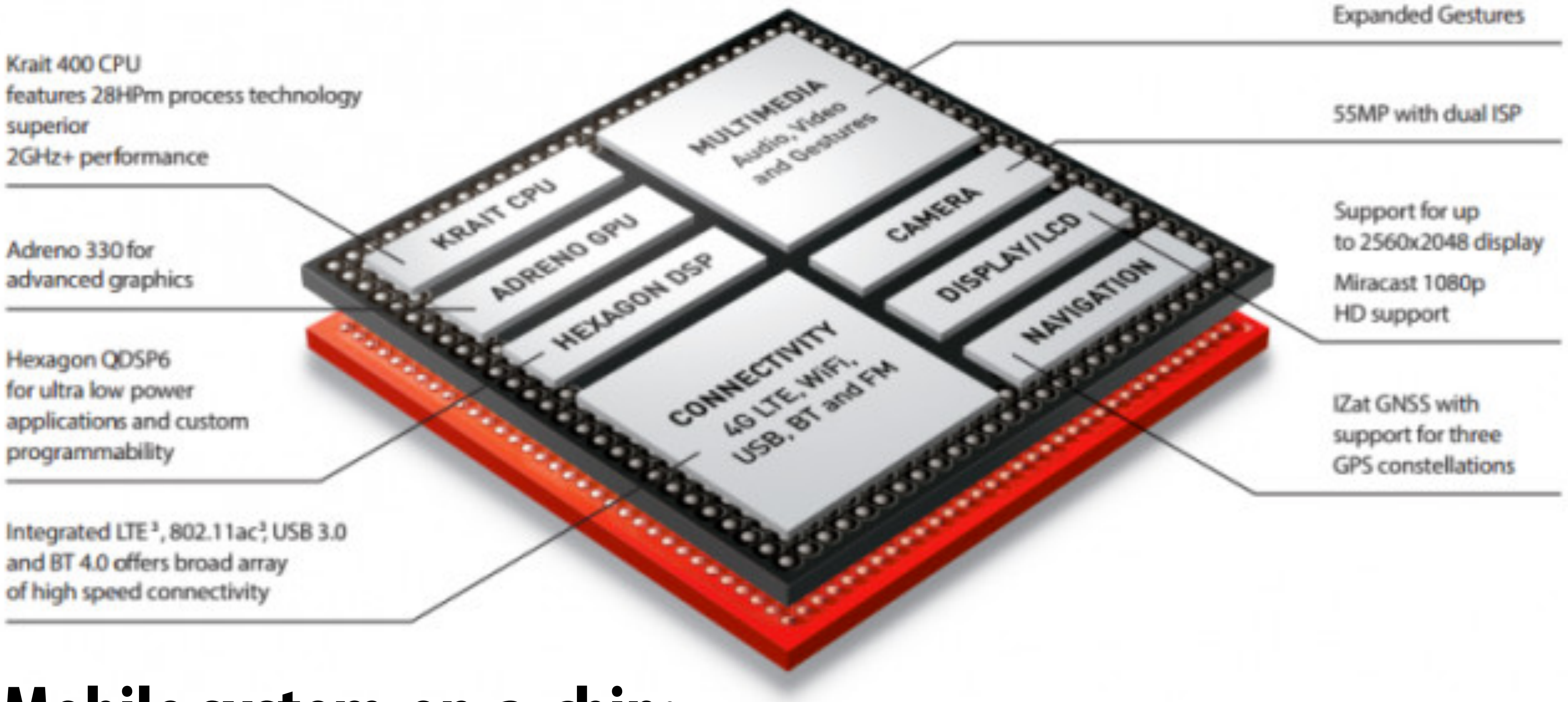
GPU:

throughput cores + fixed-function



Qualcomm Snapdragon SoC

800 PROCESSOR


KRAIT CPU
ADRENO GPU
HEXAGON DSP
MULTIMEDIA Audio, Video and Gestures
CAMERA
DISPLAY/LCD
NAVIGATION
CONNECTIVITY 4G LTE, WiFi, USB, BT and FM

Krait 400 CPU features 28HPm process technology superior 2GHz+ performance

Adreno 330 for advanced graphics

Hexagon QDSP6 for ultra low power applications and custom programmability

Integrated LTE¹, 802.11ac¹, USB 3.0 and BT 4.0 offers broad array of high speed connectivity

Ultra HD Capture and Playback

DTS-HD and Dolby Digital Plus audio

Expanded Gestures

55MP with dual ISP

Support for up to 2560x2048 display

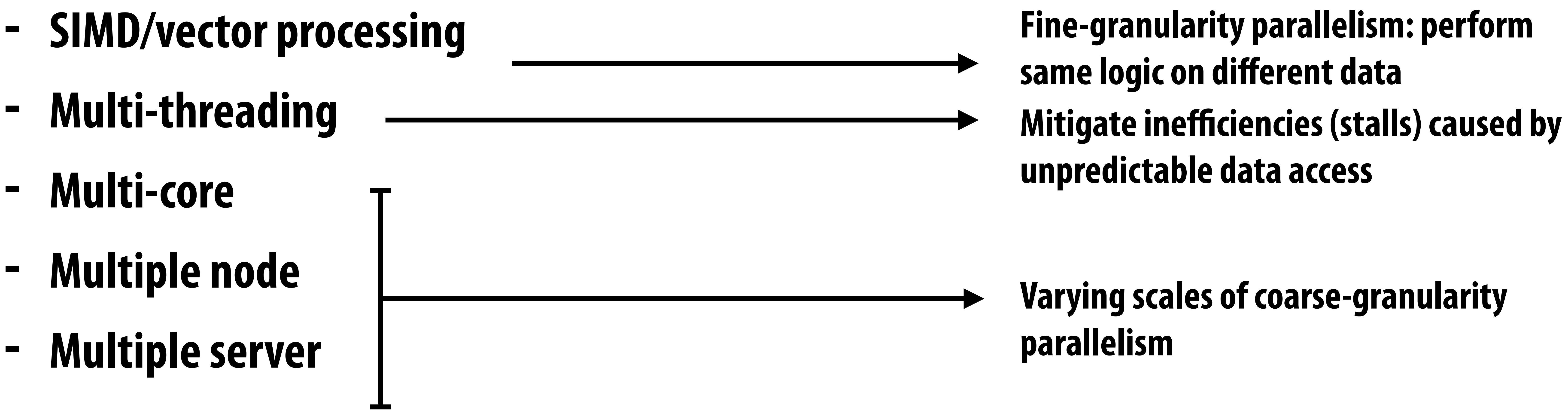
Miracast 1080p HD support

IZat GNSS with support for three GPS constellations

Mobile system-on-a-chip: CPU+GPU+media processing

Hardware trend: specialization of execution

- **Multiple forms of parallelism**



  - SIMD/vector processing
  - Multi-threading
  - Multi-core
  - Multiple node
  - Multiple server

SIMD/vector processing → Fine-granularity parallelism: perform same logic on different data

Multi-threading → Mitigate inefficiencies (stalls) caused by unpredictable data access

Multi-core, Multiple node, Multiple server → Varying scales of coarse-granularity parallelism

- **Heterogeneous execution capability**
  - Programmable, latency-centric (e.g., "CPU-like" cores)
  - Programmable, throughput-optimized (e.g., "GPU-like" cores)
  - Fixed-function, application-specific (e.g., image/video/audio processing)

Motivation for specialization: maximize compute capability given constraints on chip area, chip energy consumption.

Result: amazingly high compute capability in a wide range of devices!

Hardware diversity is a huge challenge

- **Different machines have very different performance characteristics**

- **Even worse: different technologies and performance characteristics within the same machine at different scales**
  - Within a core: SIMD, multi-threading: fine-granularity sync and communication
  - Across cores: coherent shared memory via fast on-chip network
  - Hybrid CPU+GPU multi-core: incoherent (potentially) shared memory
  - Across racks: distributed memory, multi-stage network

Variety of programming models to abstract HW

- **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)
  - Across cores: coherent shared memory via fast on-chip network
    - Abstractions: OpenMP pragma's, 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++)

Credit: Pat Hanrahan

Hardware diversity is a huge challenge

- **To be efficient, software must be optimized for the characteristics of target hardware**
  - **Difficult even in the case of one level of one machine**
  - **Combinatorial complexity of optimizations when considering a complex machine, or different machines**

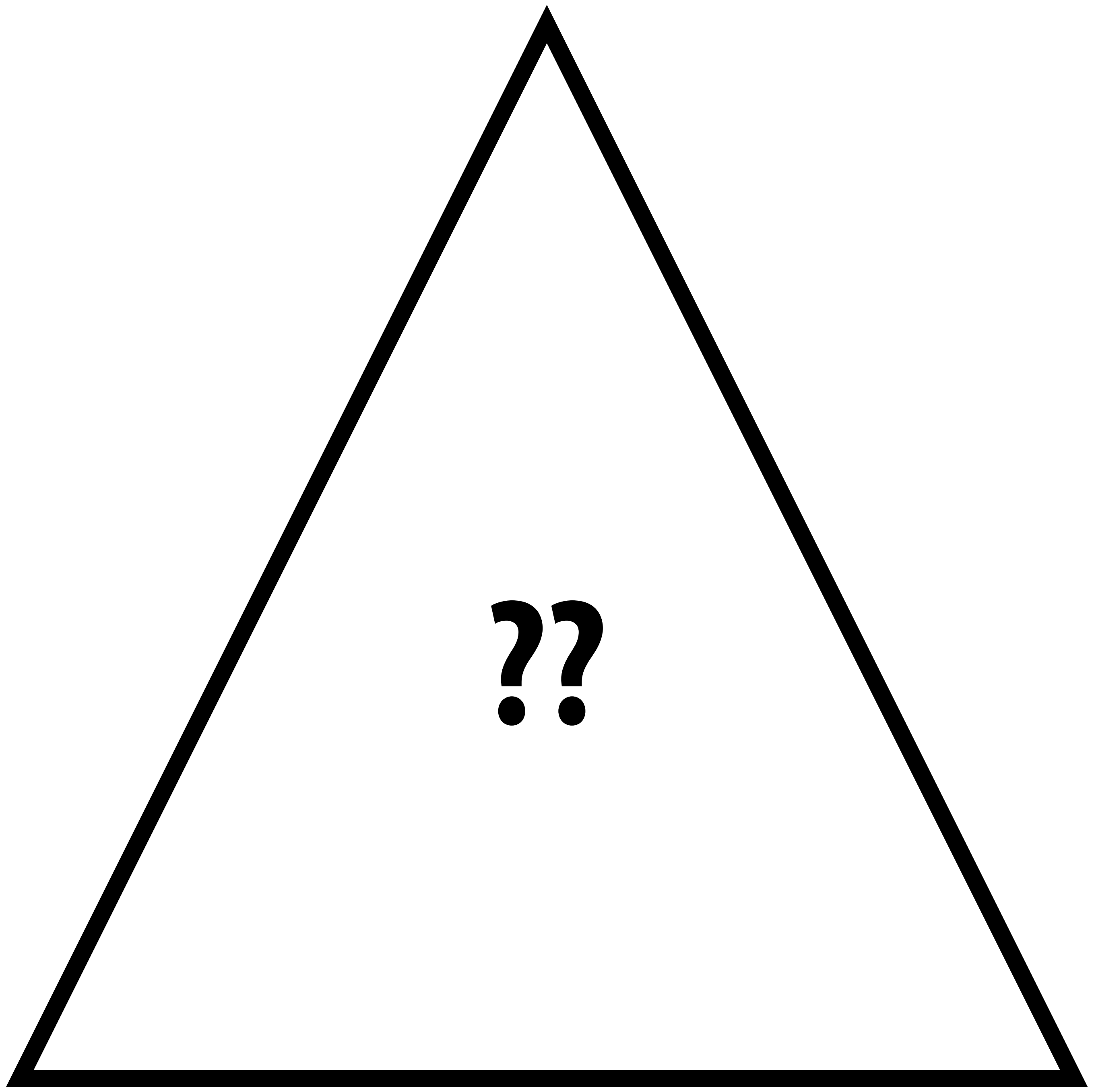
- **Result: loss of software portability**

Credit: Pat Hanrahan

Open computer science question:

How do we enable programmers to productively write software that efficiently uses current and future heterogeneous, parallel machines?

The [magical] ideal parallel programming language



High Performance
(software is scalable and efficient)

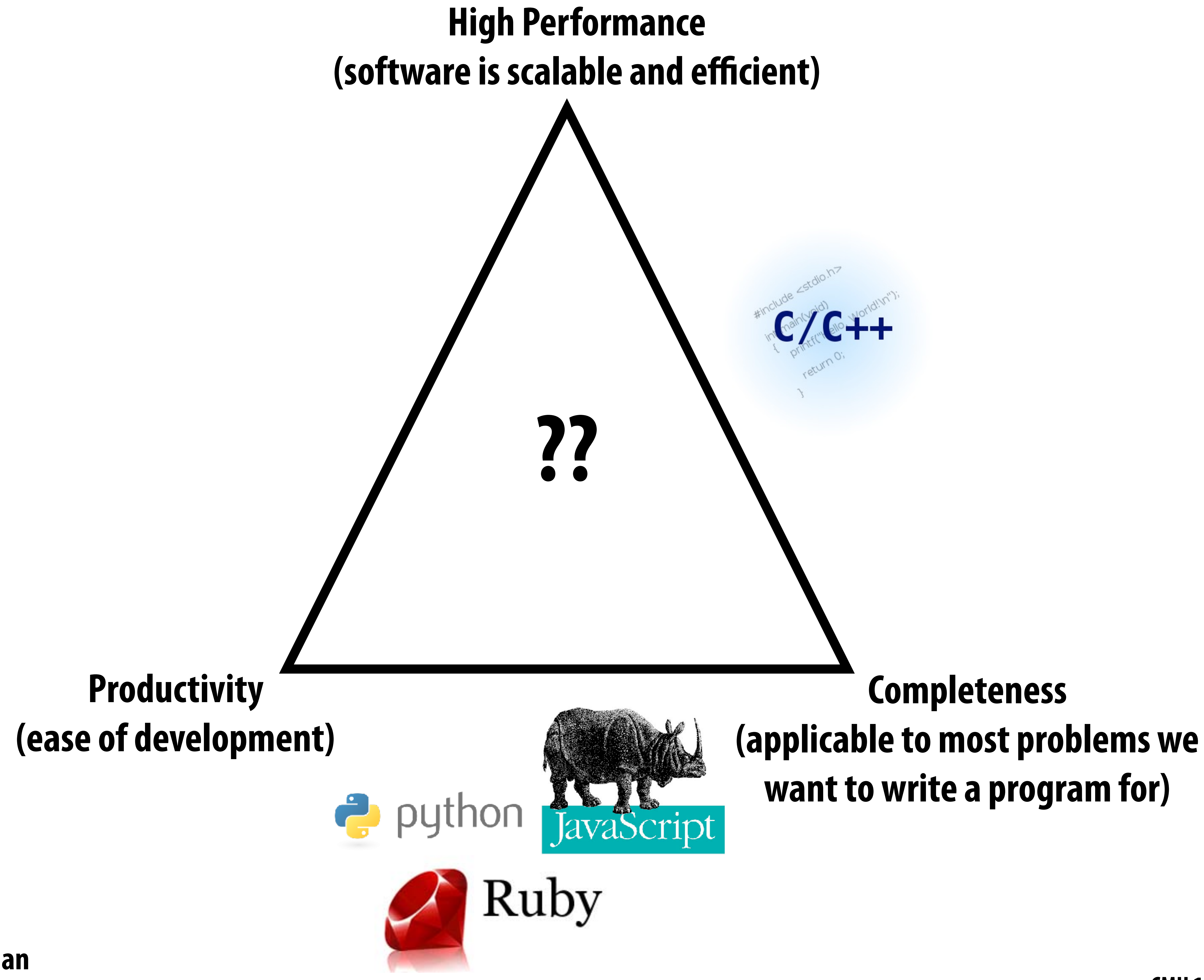
Productivity
(ease of development)

Completeness
(applicable to most problems we want to write a program for)

Credit: Pat Hanrahan

Successful programming languages

Here: definition of success = widely used



High Performance
(software is scalable and efficient)

??

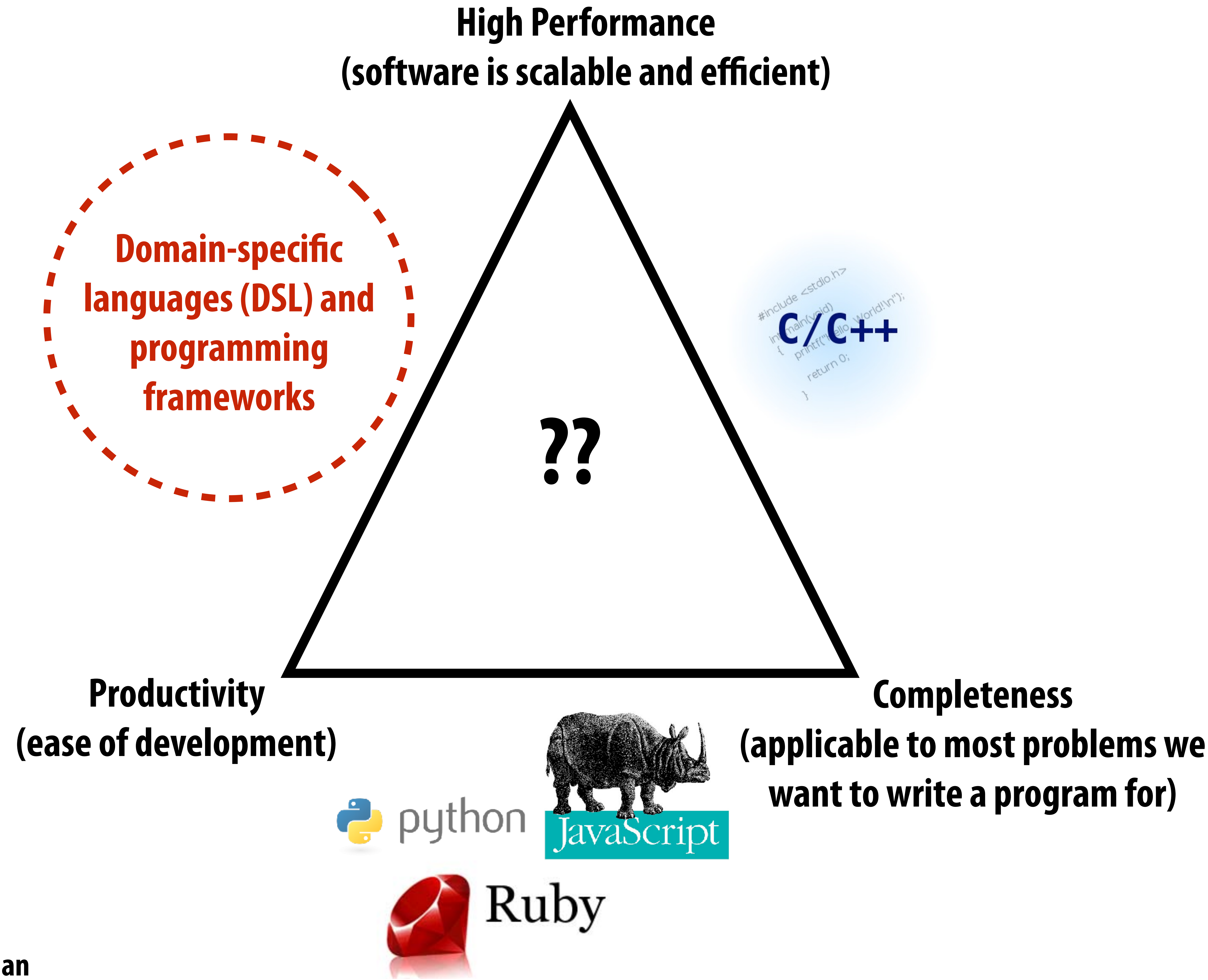
Productivity
(ease of development)

Completeness
(applicable to most problems we want to write a program for)

Credit: Pat Hanrahan

Growing interest in domain-specific programming systems

To realize high performance and productivity: willing to sacrifice completeness



High Performance
(software is scalable and efficient)

Domain-specific languages (DSL) and programming frameworks

??

Productivity
(ease of development)

Completeness
(applicable to most problems we want to write a program for)

Credit: Pat Hanrahan

Domain-specific programming systems

- **Main idea: raise level of abstraction for expressing programs**

- **Introduce high-level programming primitives specific to an application domain**
  - Productive: intuitive to use, portable across machines, primitives correspond to behaviors frequently used to solve problems in targeted domain
  - Performant: 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

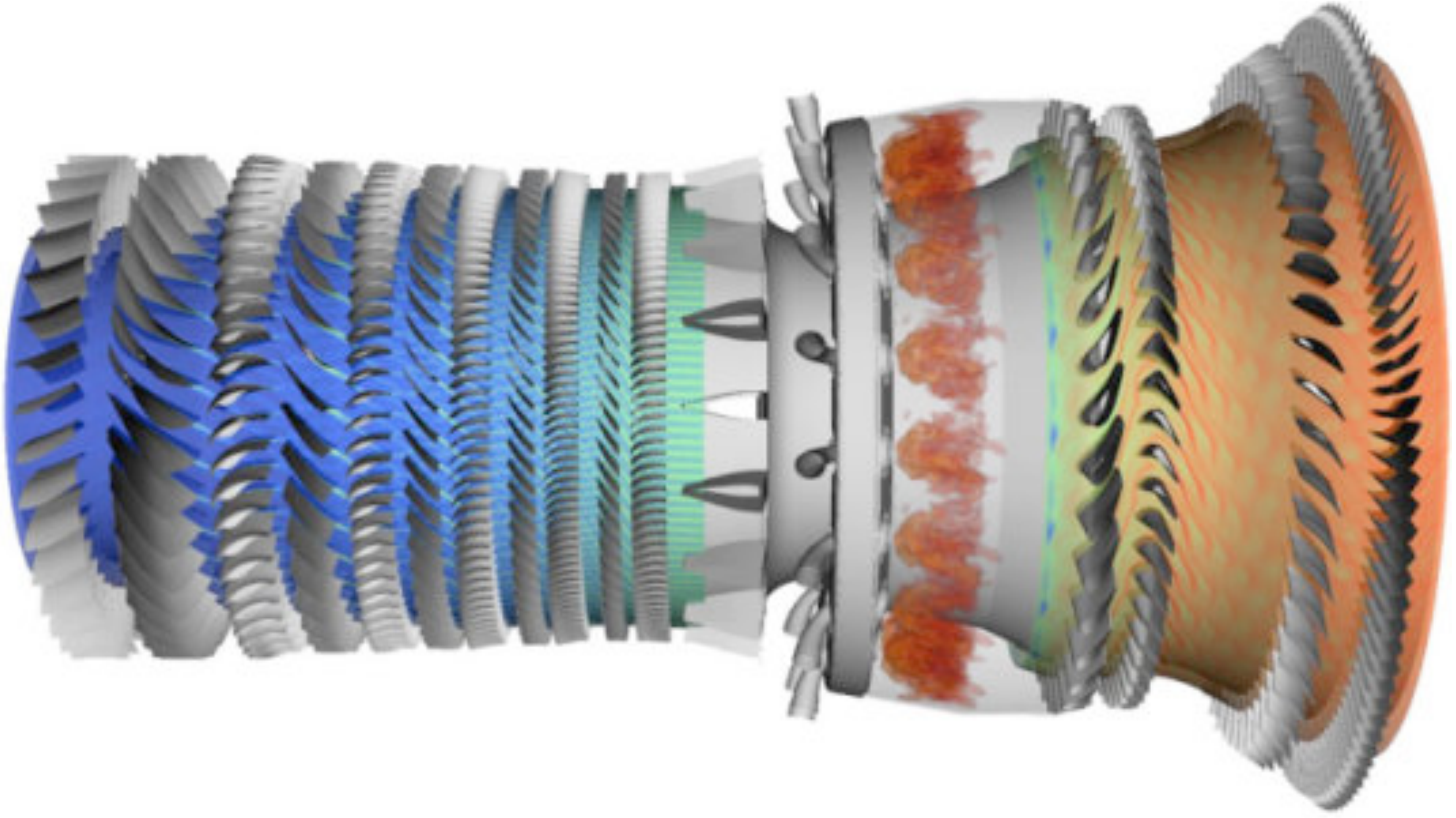
1. Liszt: for scientific computing on meshes
2. Halide: for image processing

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)

http://liszt.stanford.edu/

What a Liszt program does

A Liszt program is run on a mesh

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

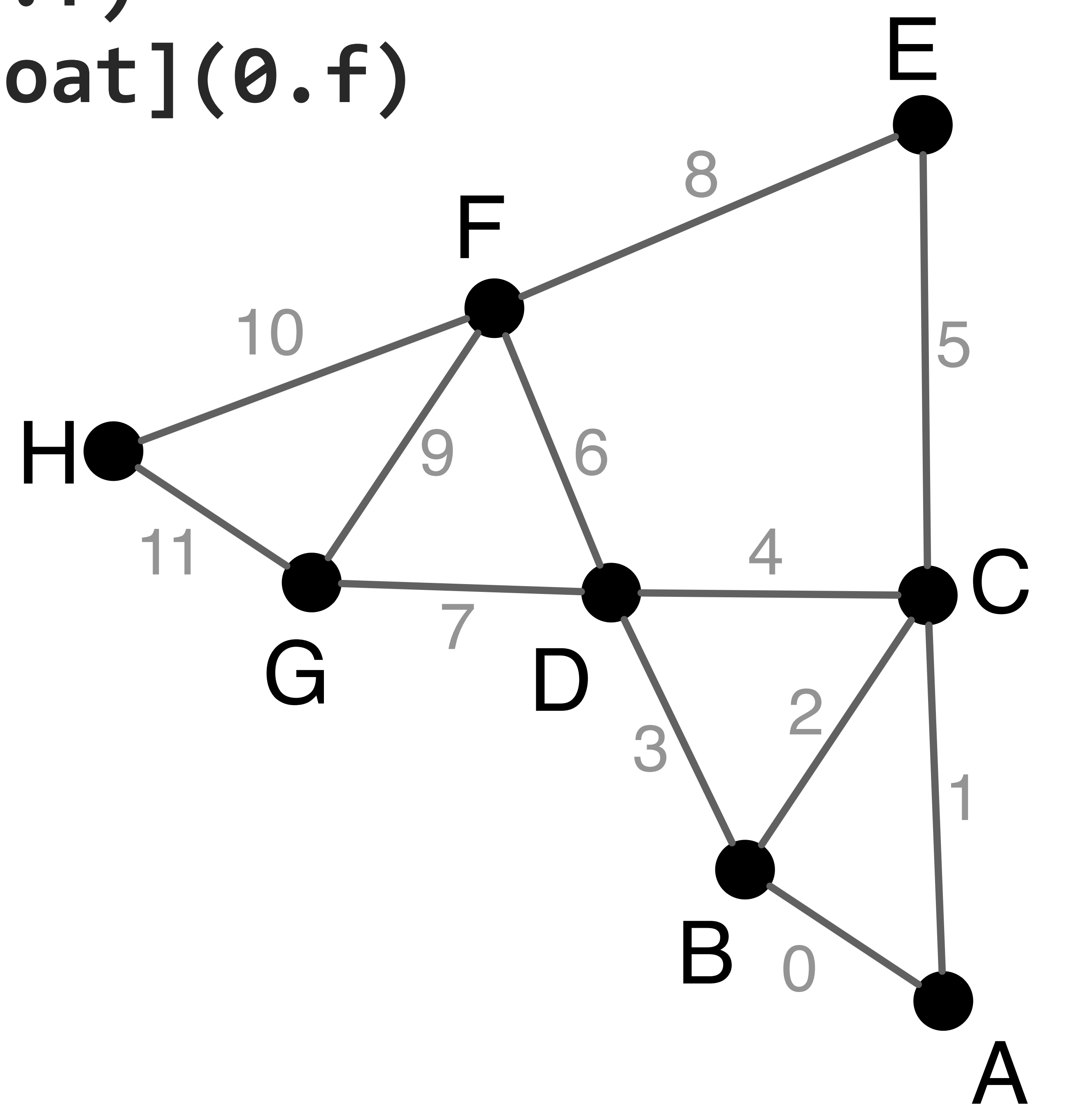
Position is a field defined at each mesh vertex.
The field's value is represented by a 3-vector.

```
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



Side note:

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

```
var i = 0;
while ( i < 1000 ) {
  Flux(vertices(mesh)) = 0.f;
  JacobiStep(vertices(mesh)) = 0.f;
  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
  }
  i += 1
}
```

Set flux for all vertices to 0.f;

Independently, for each edge in the mesh

Given edge, loop body accesses/modifies field values at adjacent mesh vertices

Access value of field at mesh vertex v2

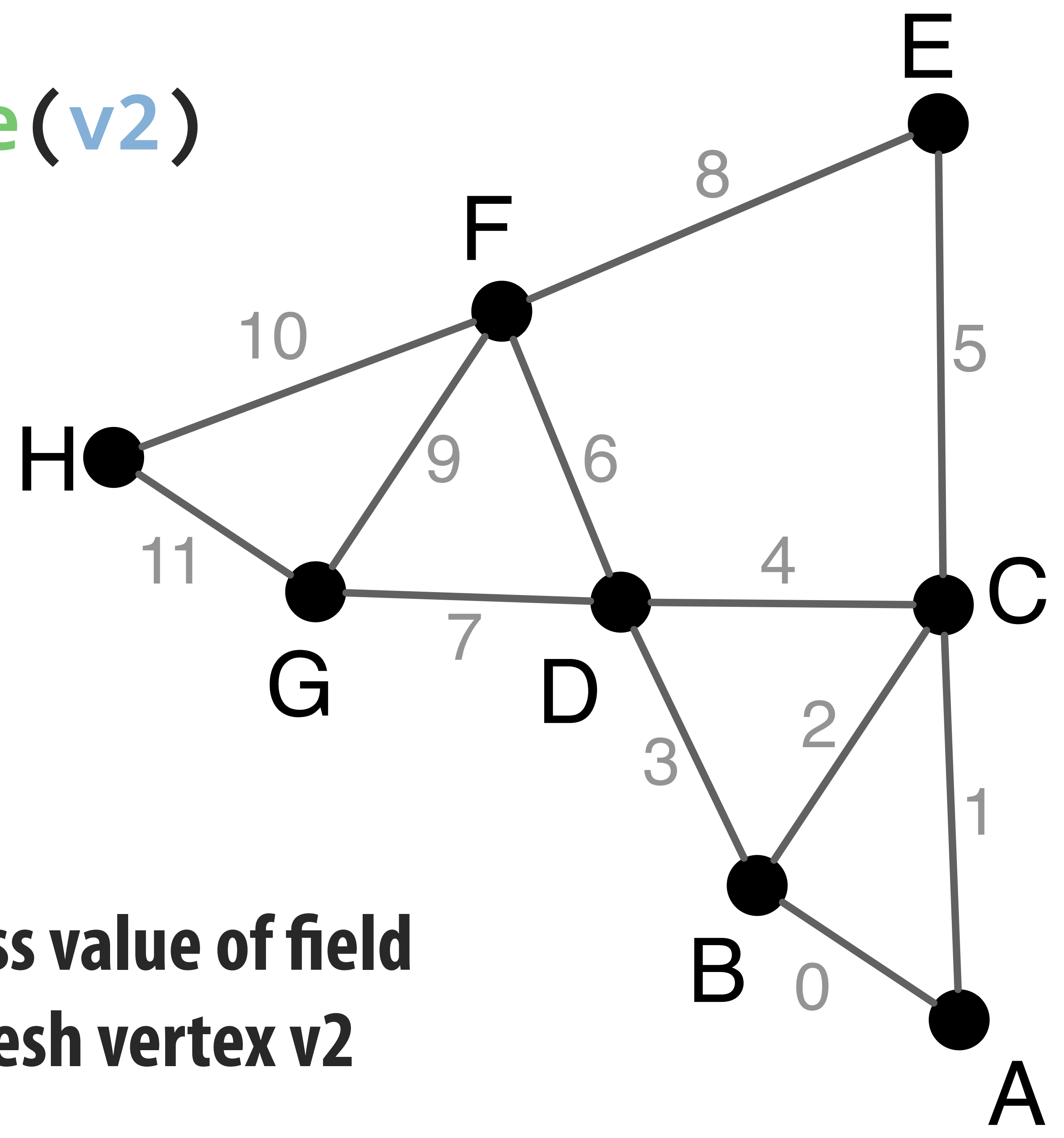
Color key:

Fields

Mesh

Topology functions

Iteration over set



Liszt's topological operators

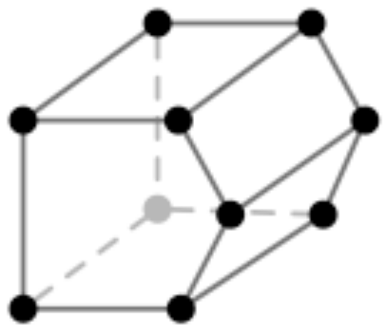
Used to access mesh elements relative to some input vertex, edge, face, etc.

Topological operators are the only way to access mesh data in a Liszt program

Notice how many operators return sets (e.g., "all edges of this face")

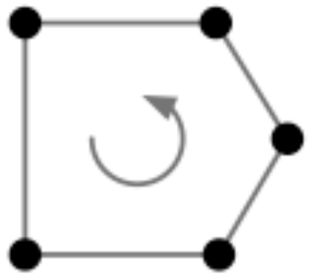
```
BoundarySet[1][ME <: MeshElement](name : String) : Set[ME]
vertices(e : Mesh) : Set[Vertex]
cells(e : Mesh) : Set[Cell]
edges(e : Mesh) : Set[Edge]
faces(e : Mesh) : Set[Face]
```

```
vertices(e : Vertex) : Set[Vertex]
cells(e : Vertex) : Set[Cell]
edges(e : Vertex) : Set[Edge]
faces(e : Vertex) : Set[Face]
```



```
cells(e : Cell) : Set[Cell]
vertices(e : Cell) : Set[Vertex]
faces(e : Cell) : Set[Face]
edges(e : Cell) : Set[Edge]
```

```
vertices(e : Edge) : Set[Vertex]
facesCCW[2](e : Edge) : Set[Face]
cells(e : Edge) : Set[Cell]
head(e : Edge) : Vertex
tail(e : Edge) : Vertex
flip[4](e : Edge) : Edge
towards[5](e : Edge, t : Vertex) : Edge
```



```
cells(e : Face) : Set[Cell]
edgesCCW[2](e : Face) : Set[Edge]
vertices(e : Face) : Set[Vertex]
inside[3](e : Face) : Cell
outside[3](e : Face) : Cell
flip[4](e : Face) : Face
towards[5](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)
  - Based on mesh type, program behavior, and target machine

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 what synchronization is required**

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

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:

Consider: `a[f(i)] += b[i];`
(must execute `f(i)` **to know if dependency exists across loop iterations** `i`**)**

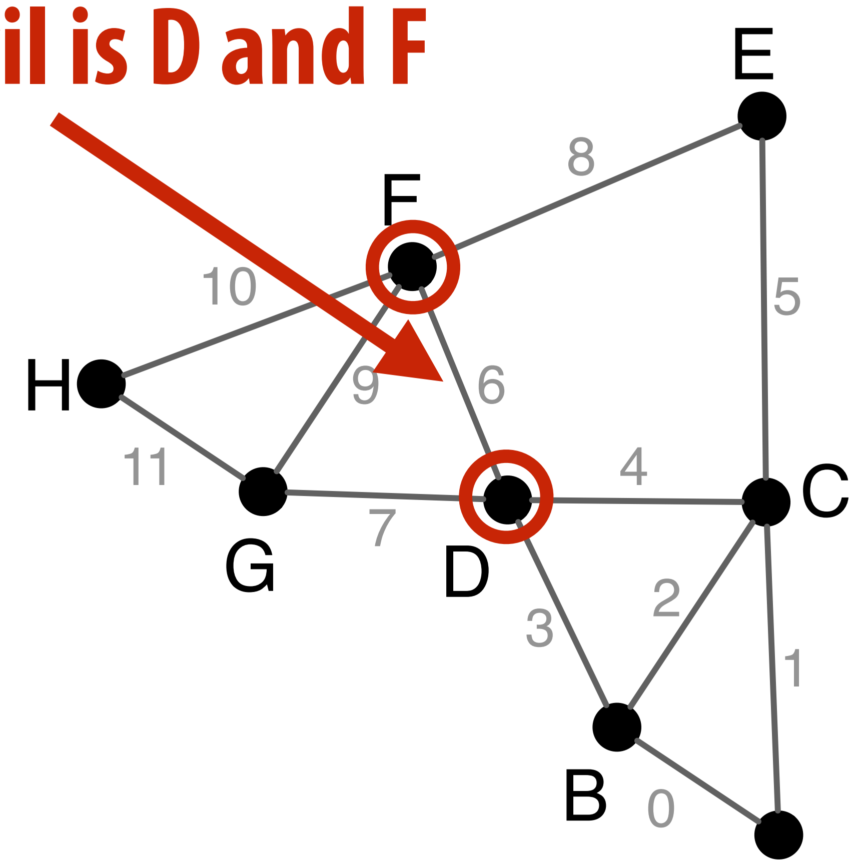
Liszt is constrained to allow dependency analysis

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

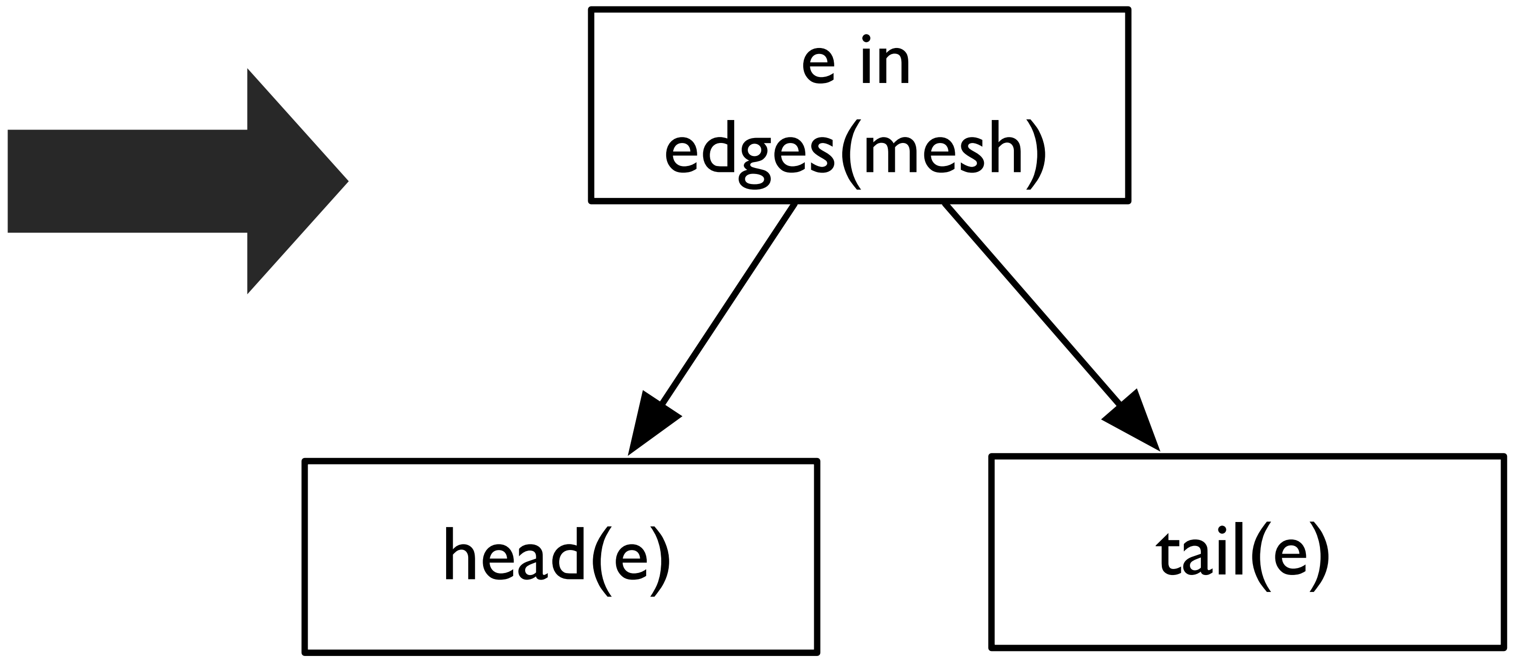
Statically analyze code to find stencil of each top-level for loop

- Extract nested mesh element reads
- Extract field operations

Edge 6's read stencil is D and F



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



Read Position,Temperature
Write Flux, JacobiStep

Read Position, Temperature
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: (immutable values)

```
val v1 = head(e)
```

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

```
Pressure(v)
```

- No recursive functions

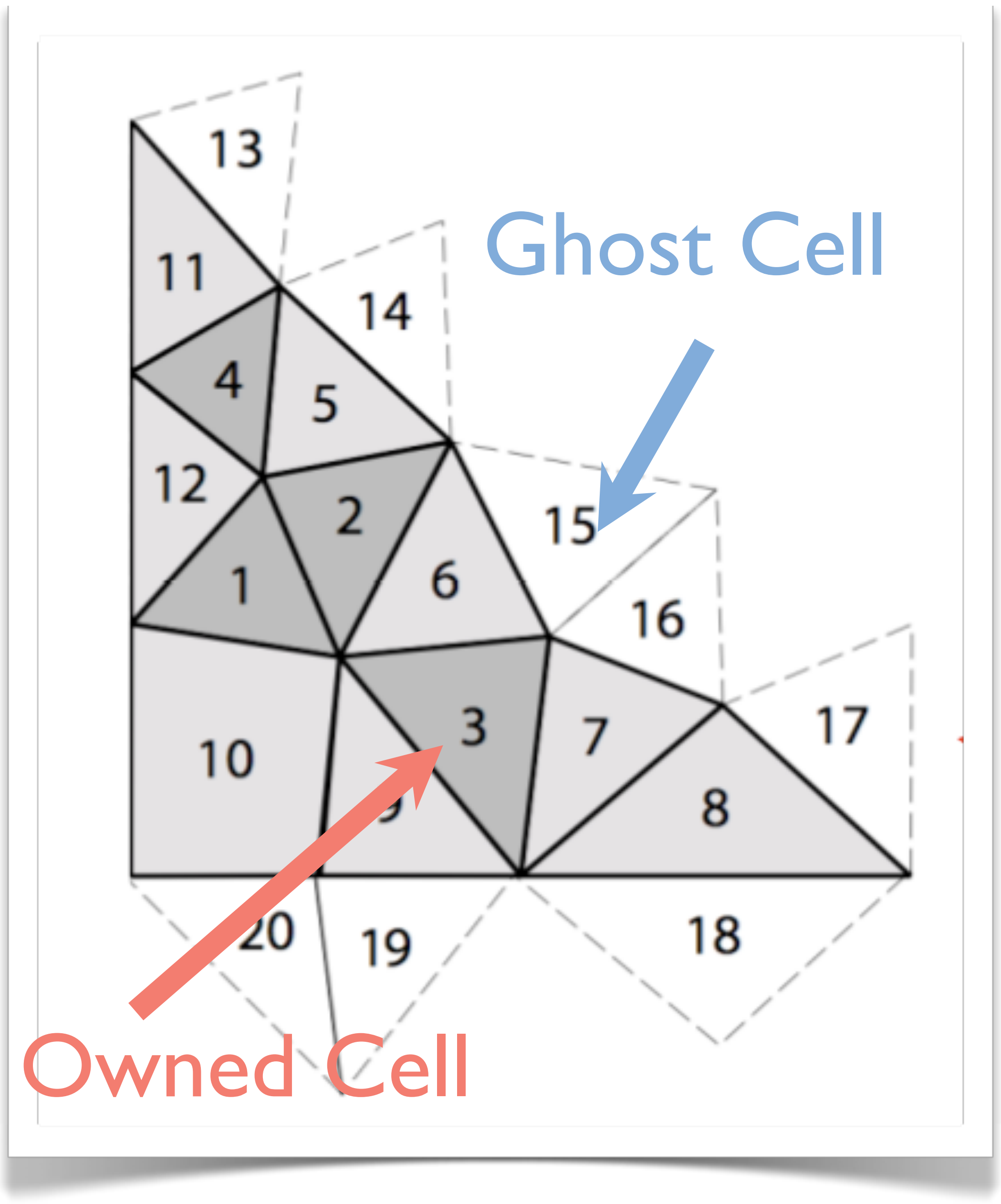
Restrictions allow compiler to automatically infer stencil for a loop iteration

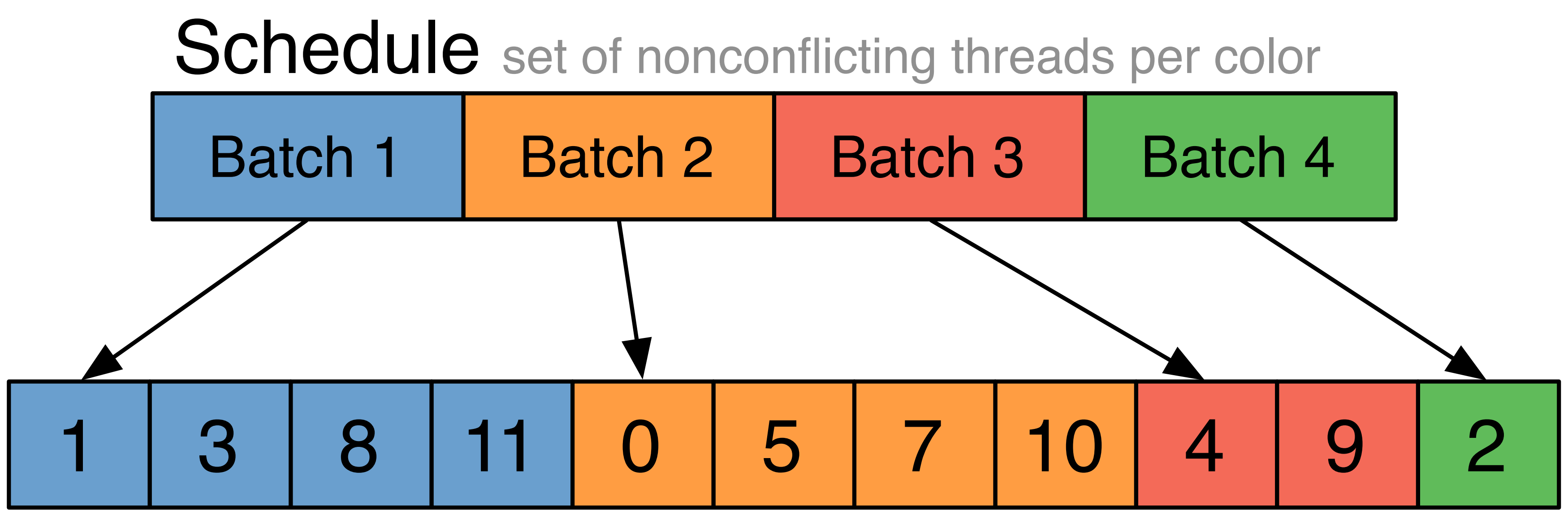
Portable parallelism: compiler uses knowledge of dependencies to implement different parallel execution strategies

I'll discuss two strategies…

Strategy 1: mesh partitioning

Strategy 2: mesh coloring





Schedule set of nonconflicting threads per color

Batch 1 | Batch 2 | Batch 3 | Batch 4

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

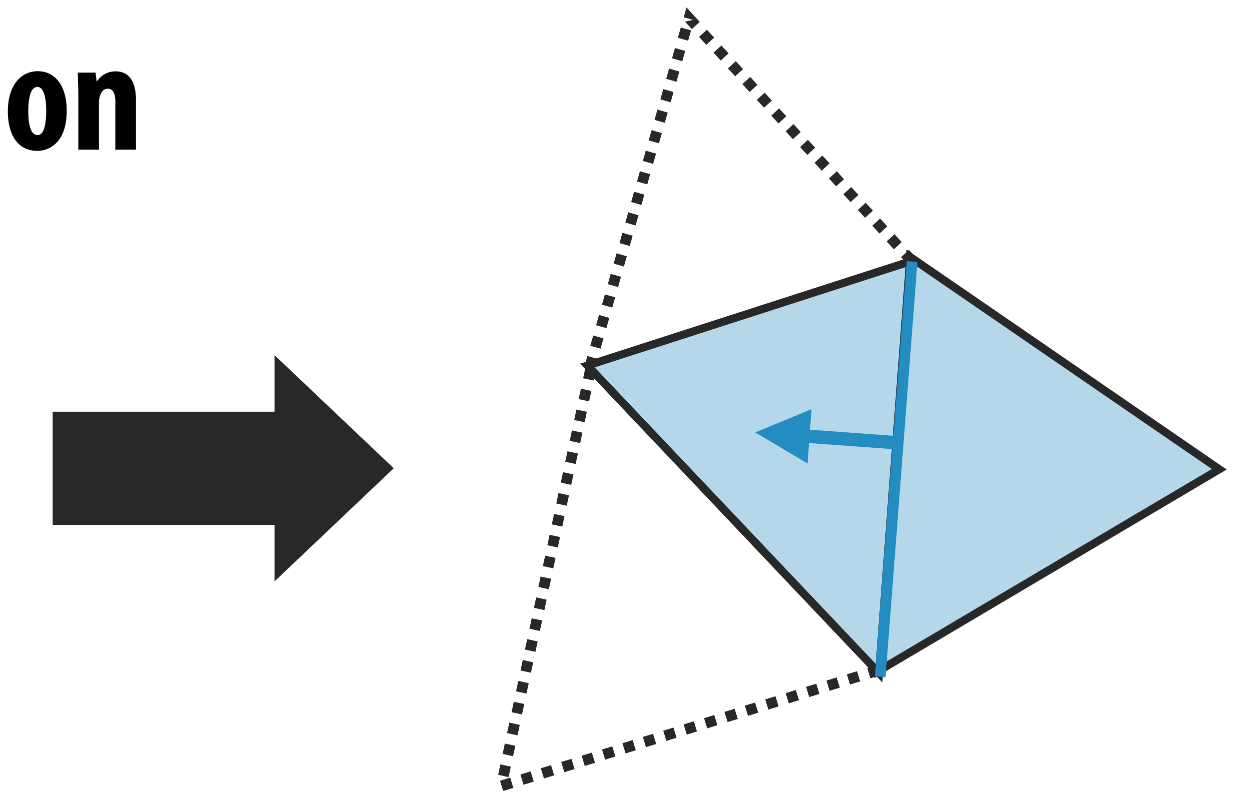
Imagine compiling a Lizst program to the latedays cluster (multiple nodes, distributed address space)

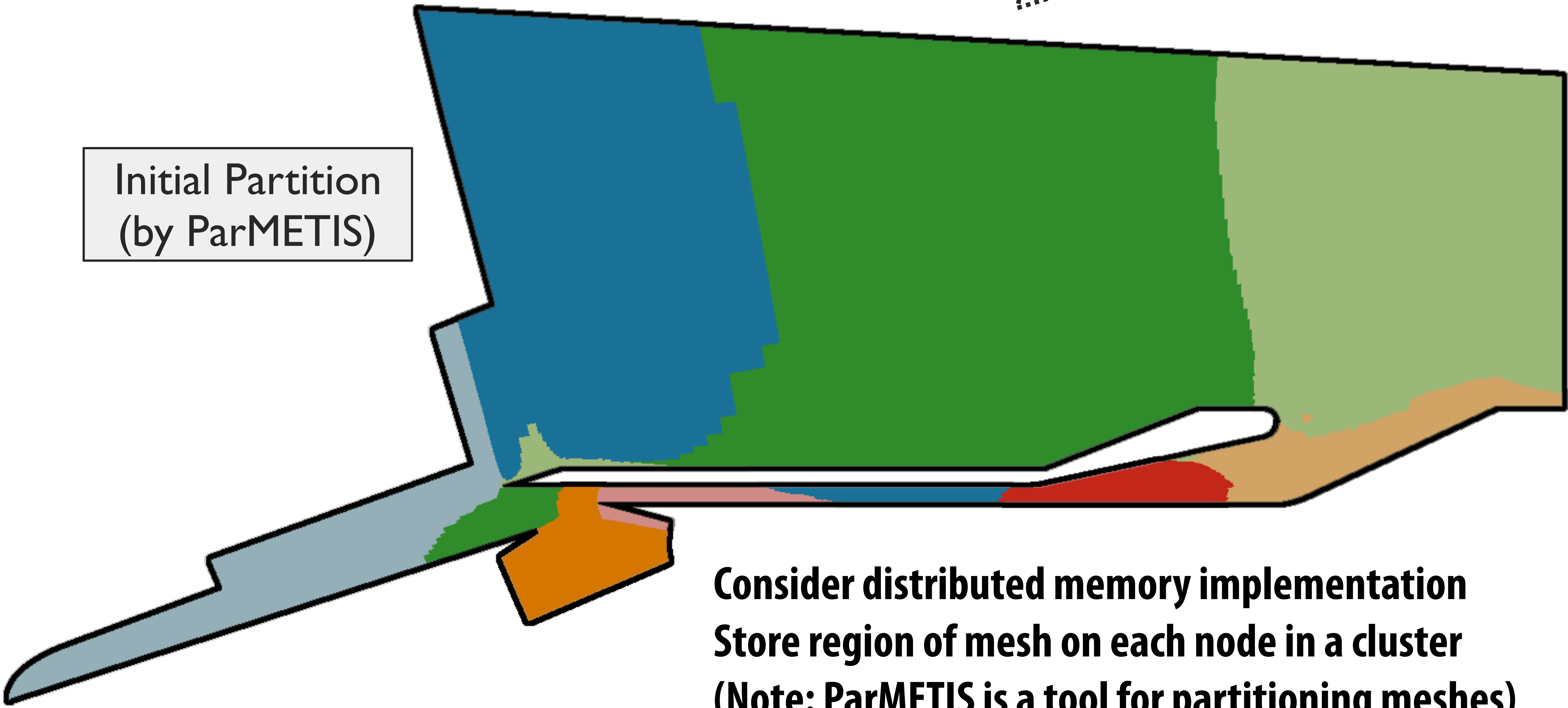
How might Liszt distribute a graph across these nodes?

Distributed memory implementation of Liszt

Mesh + Stencil → Graph → Partition

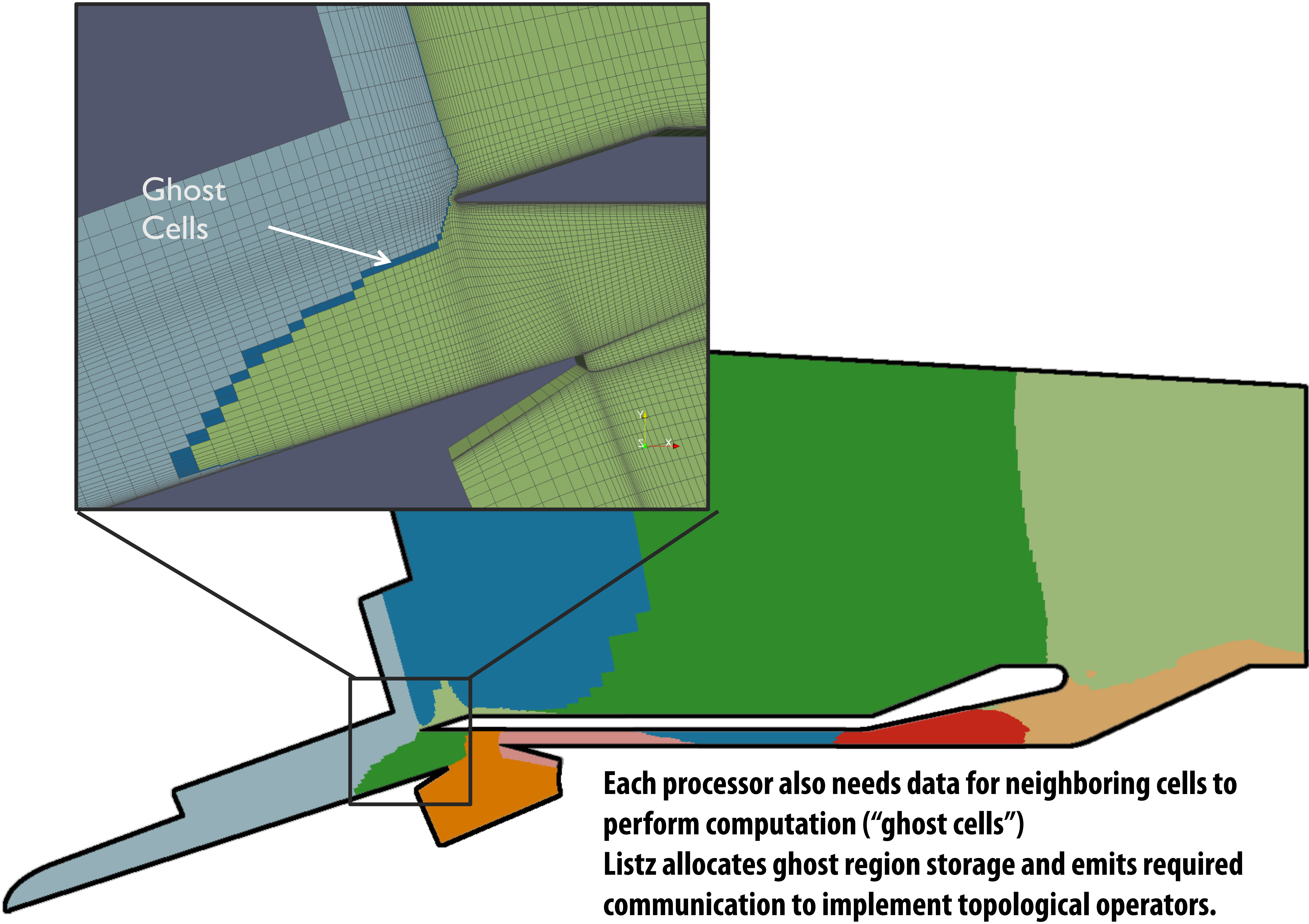
```
for(f <- faces(mesh)) {
  rhoOutside(f) =
    calc_flux(f, rho(outside(f))) +
    calc_flux(f, rho(inside(f)))
}
```





Initial Partition
(by ParMETIS)

Consider distributed memory implementation
Store region of mesh on each node in a cluster
(Note: ParMETIS is a tool for partitioning meshes)



Each processor also needs data for neighboring cells to perform computation ("ghost cells")

Listz allocates ghost region storage and emits required communication to implement topological operators.

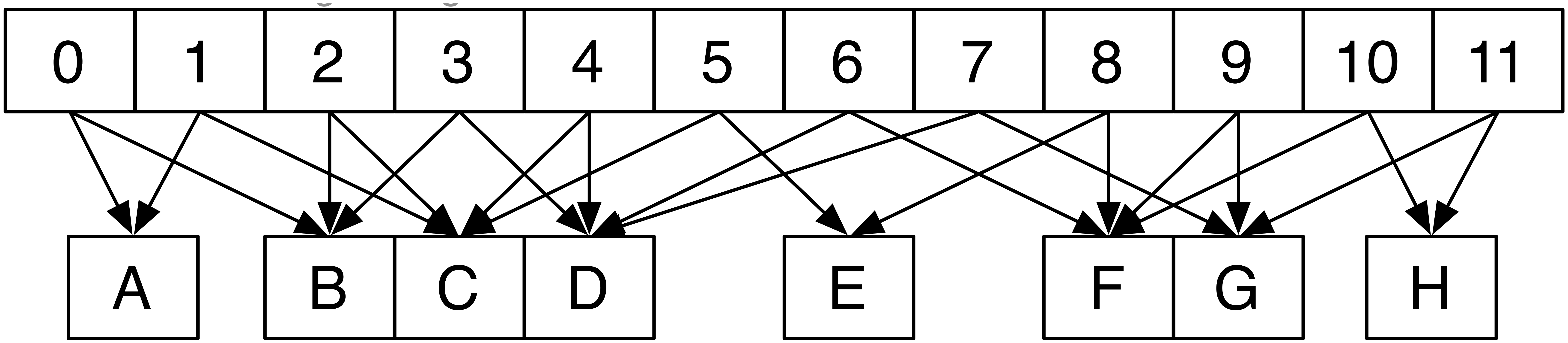
Imagine compiling a Lizst program to a GPU (single address space, many tiny threads)

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

Edges (each edge assigned to 1 CUDA thread)



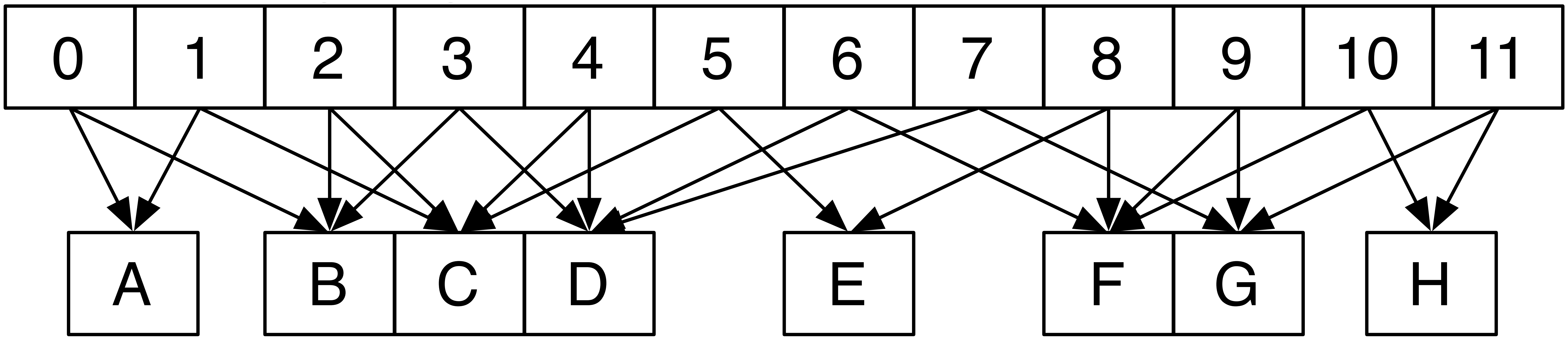
Flux field values (stored per vertex)

```
for (e <- edges(mesh)) {
  ...
  Flux(v1) += dT*step
  Flux(v2) -= dT*step
  ...
}
```

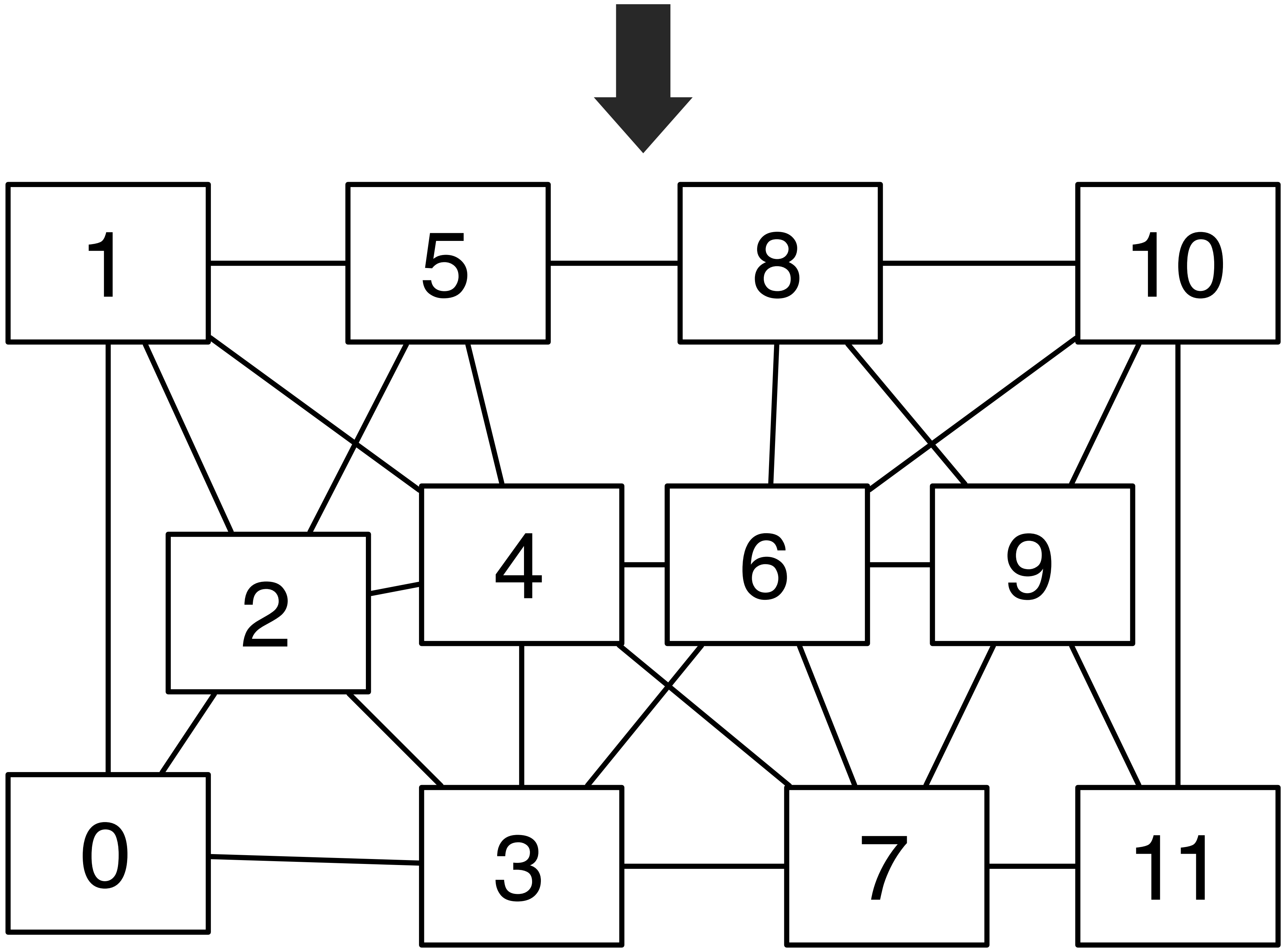
Different edges share a vertex: requires atomic update of per-vertex field data

GPU implementation: conflict graph

Edges (each edge assigned to 1 CUDA thread)



Flux field values (per vertex)

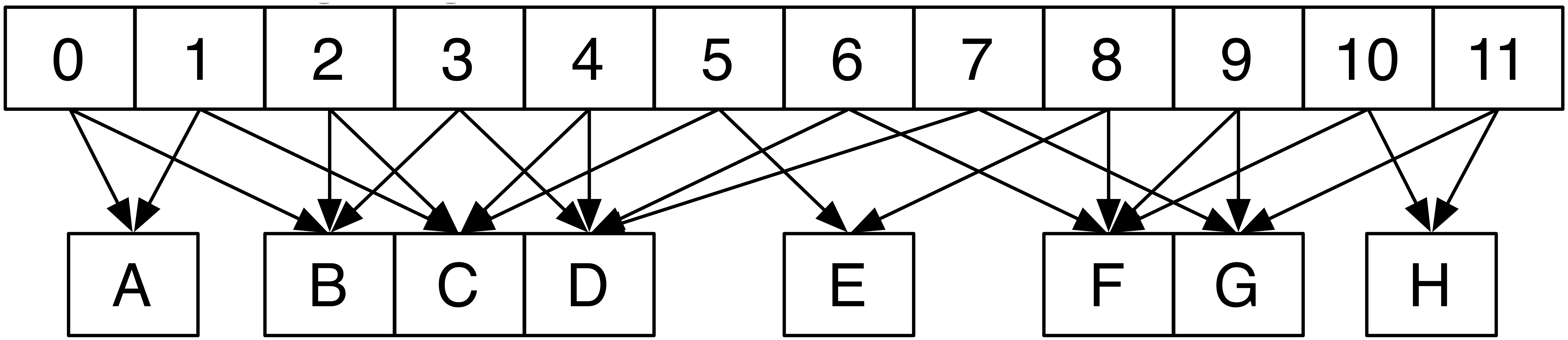


Identify mesh edges with colliding writes (lines in graph indicate presence of collision)

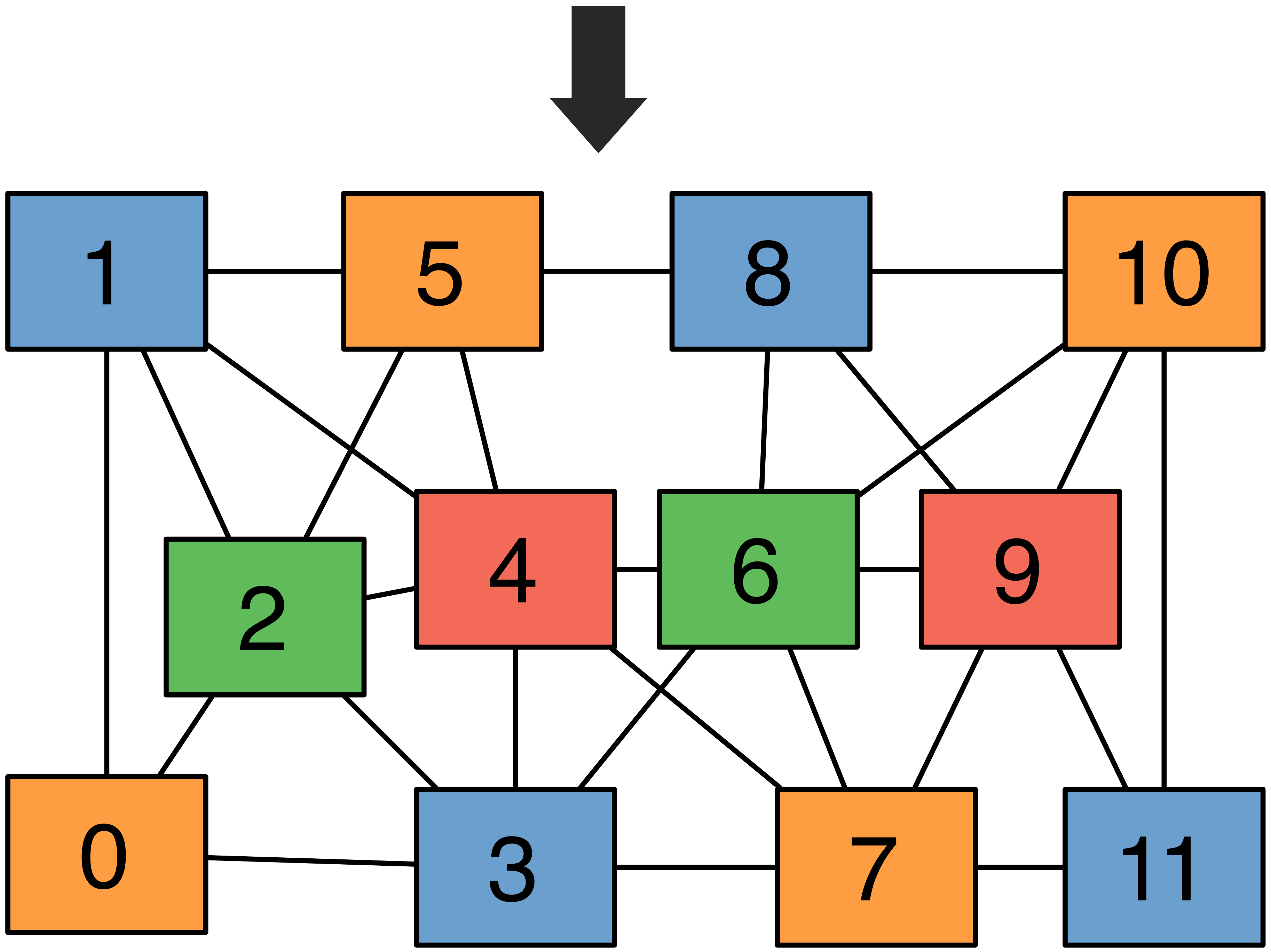
Can simply run program once to get this information.
(results remain valid for subsequent executions provided mesh does not change)

GPU implementation: conflict graph

Threads (each edge assigned to 1 CUDA thread)



Flux field values (per vertex)

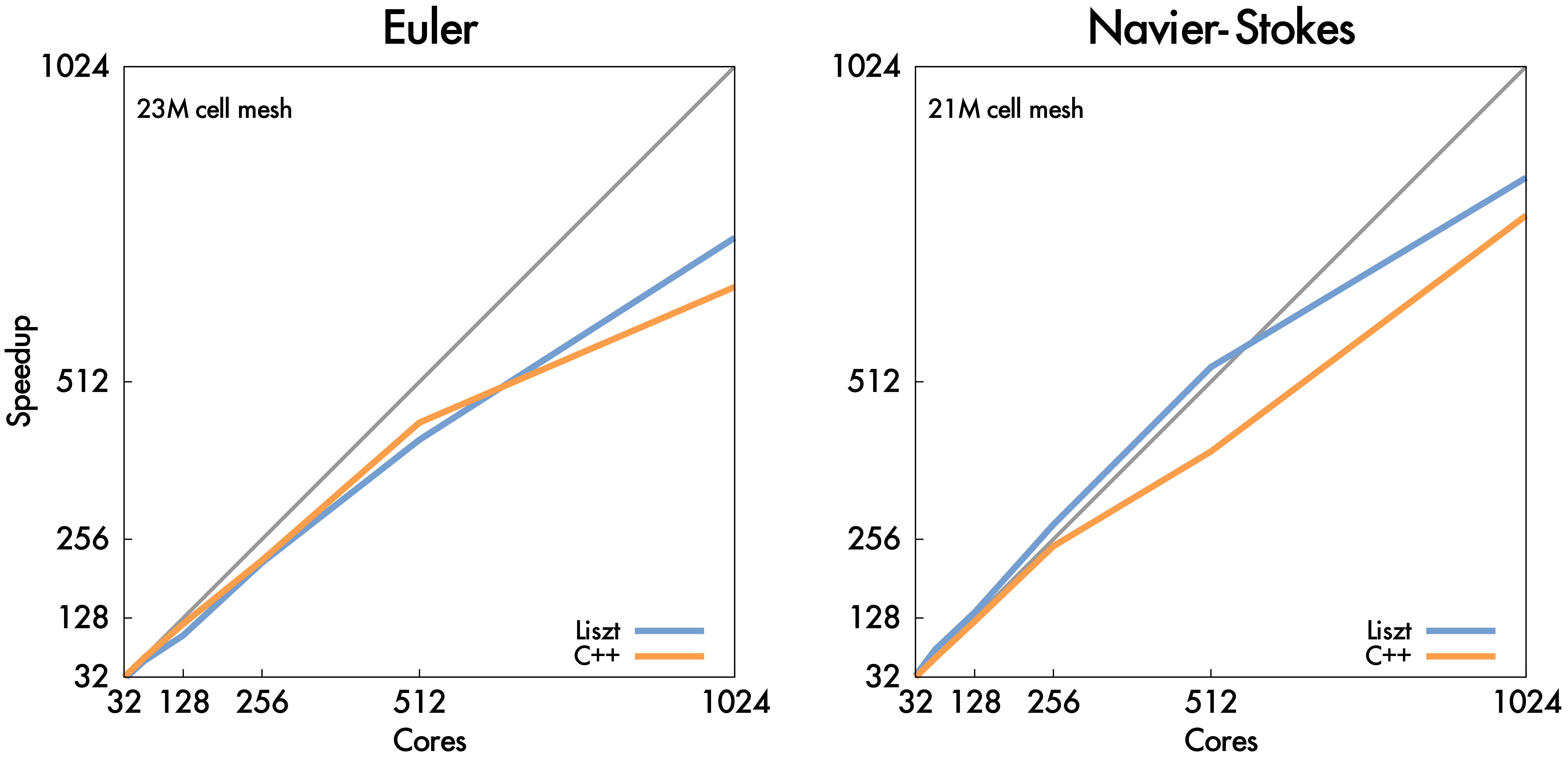


“Color” nodes in graph such that no connected nodes have the same color

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)

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 and GPUs (and combinations thereof!)

- **High performance**
  - Language is constrained to allow compiler to track dependencies
  - Used for locality-aware partitioning (distributed memory implementation)
  - Used for graph coloring to avoid sync (GPU implementation)
  - Compiler chooses different parallelization strategies for different platforms
  - System can customize mesh representation based on application 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)

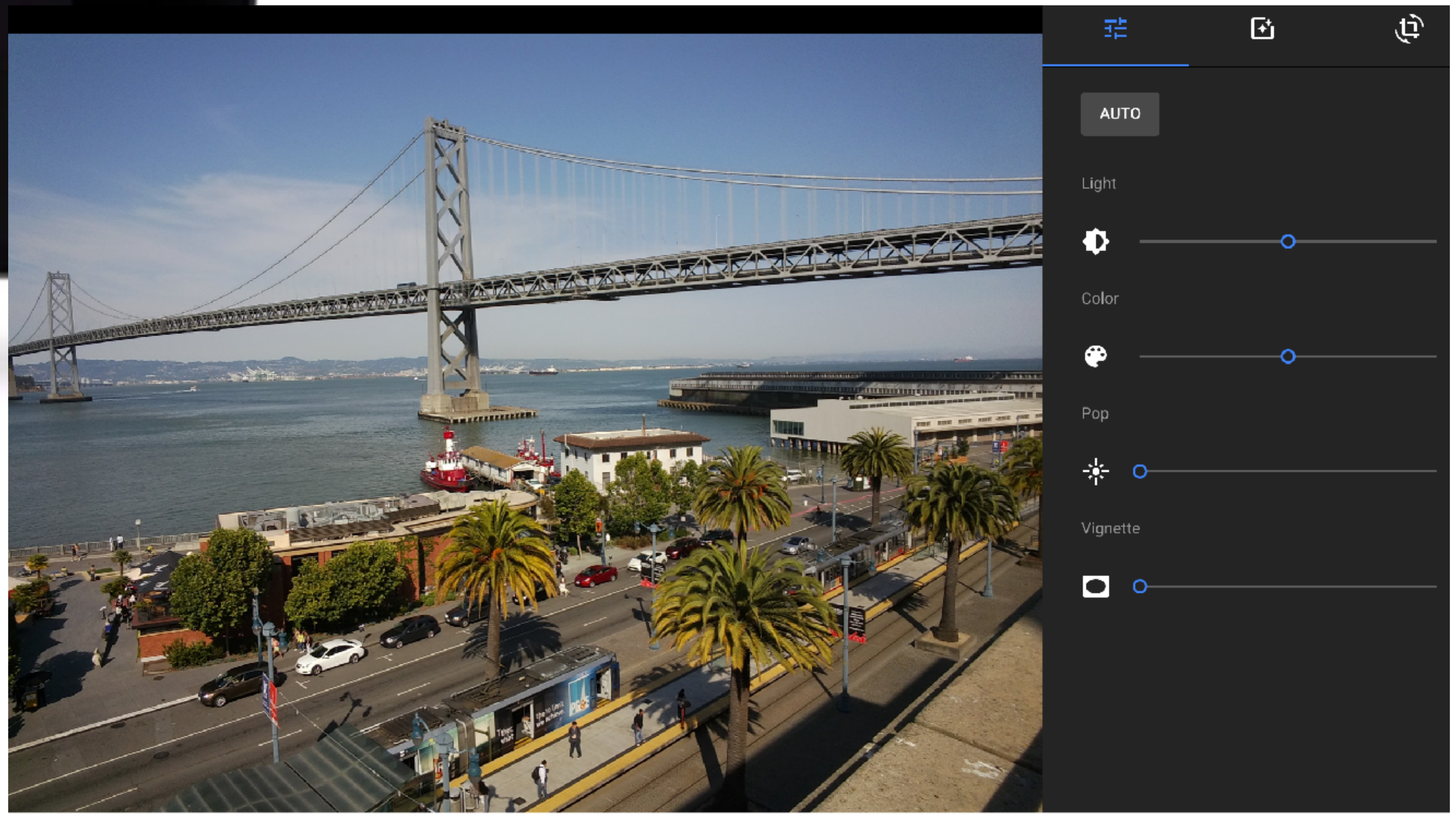
Example 2:

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 Google Pixel Photos app
- Halide code used to process images uploaded to Google Photos

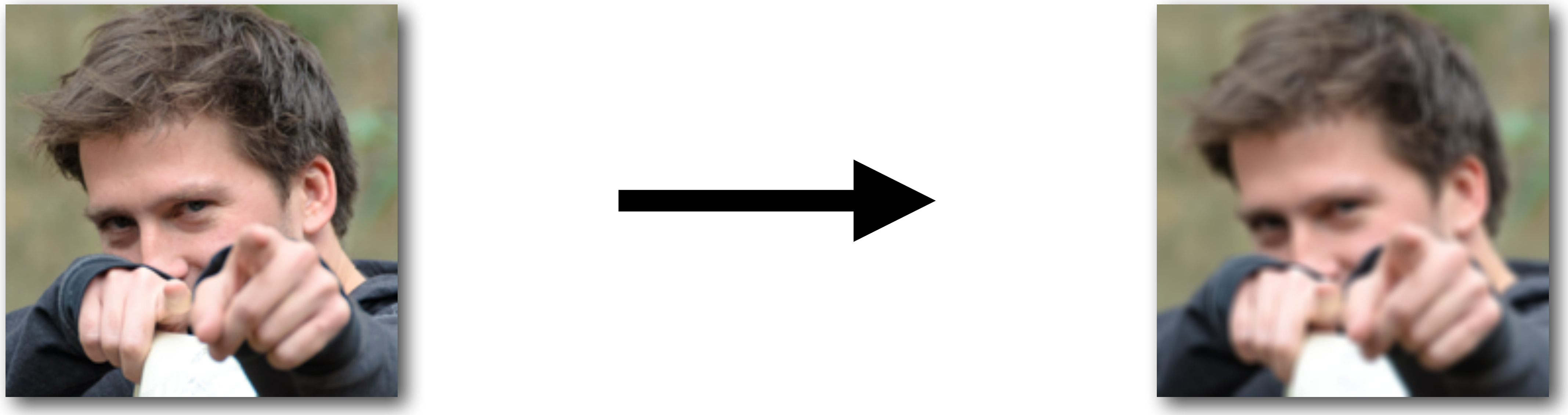


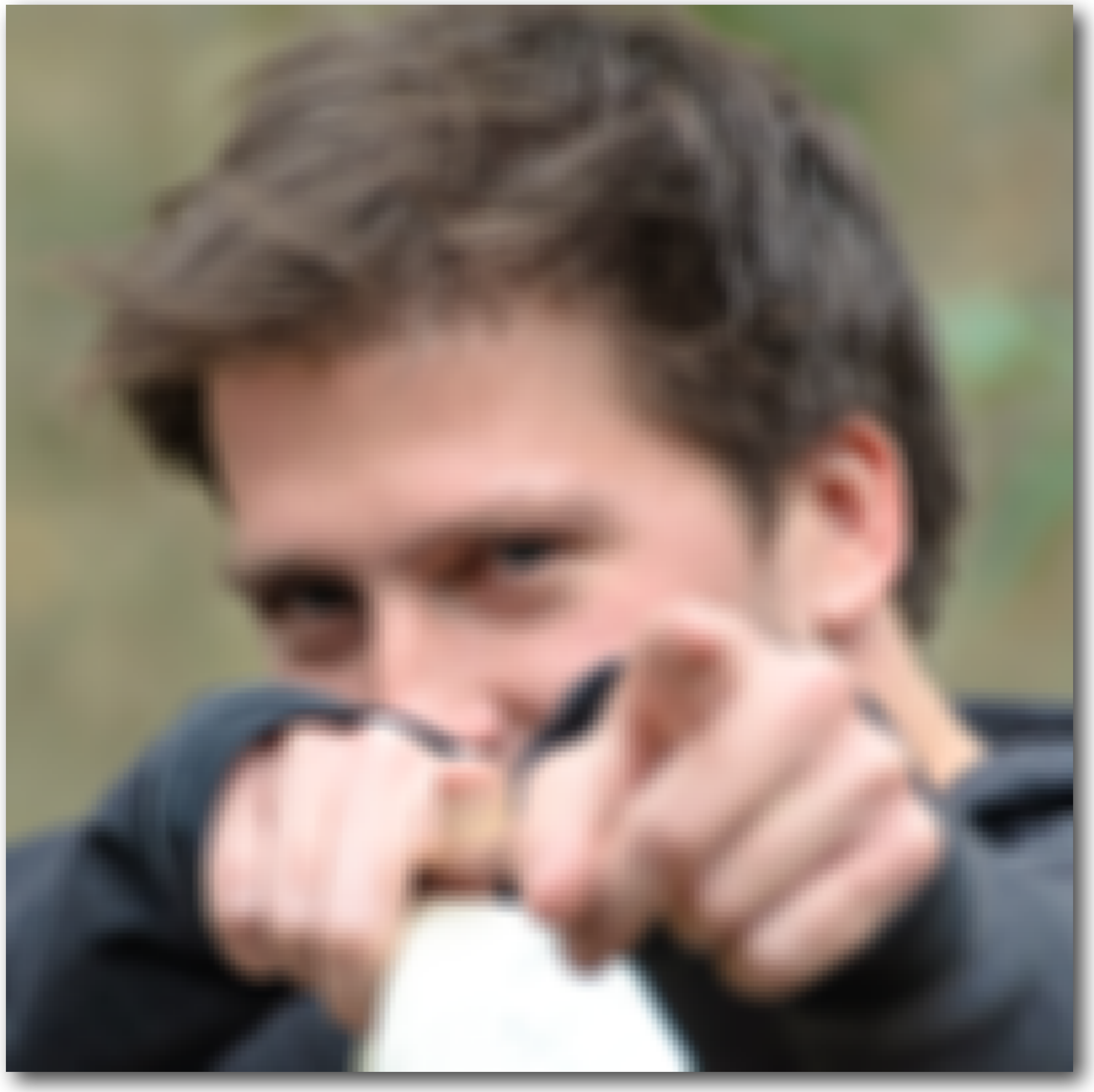
A quick tutorial on high-performance image processing

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.f/9, 1.f/9, 1.f/9,
                   1.f/9, 1.f/9, 1.f/9,
                   1.f/9, 1.f/9, 1.f/9};

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    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;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/9, 1.f/9, 1.f/9,
                   1.f/9, 1.f/9, 1.f/9,
                   1.f/9, 1.f/9, 1.f/9};

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    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;
  }
}
```

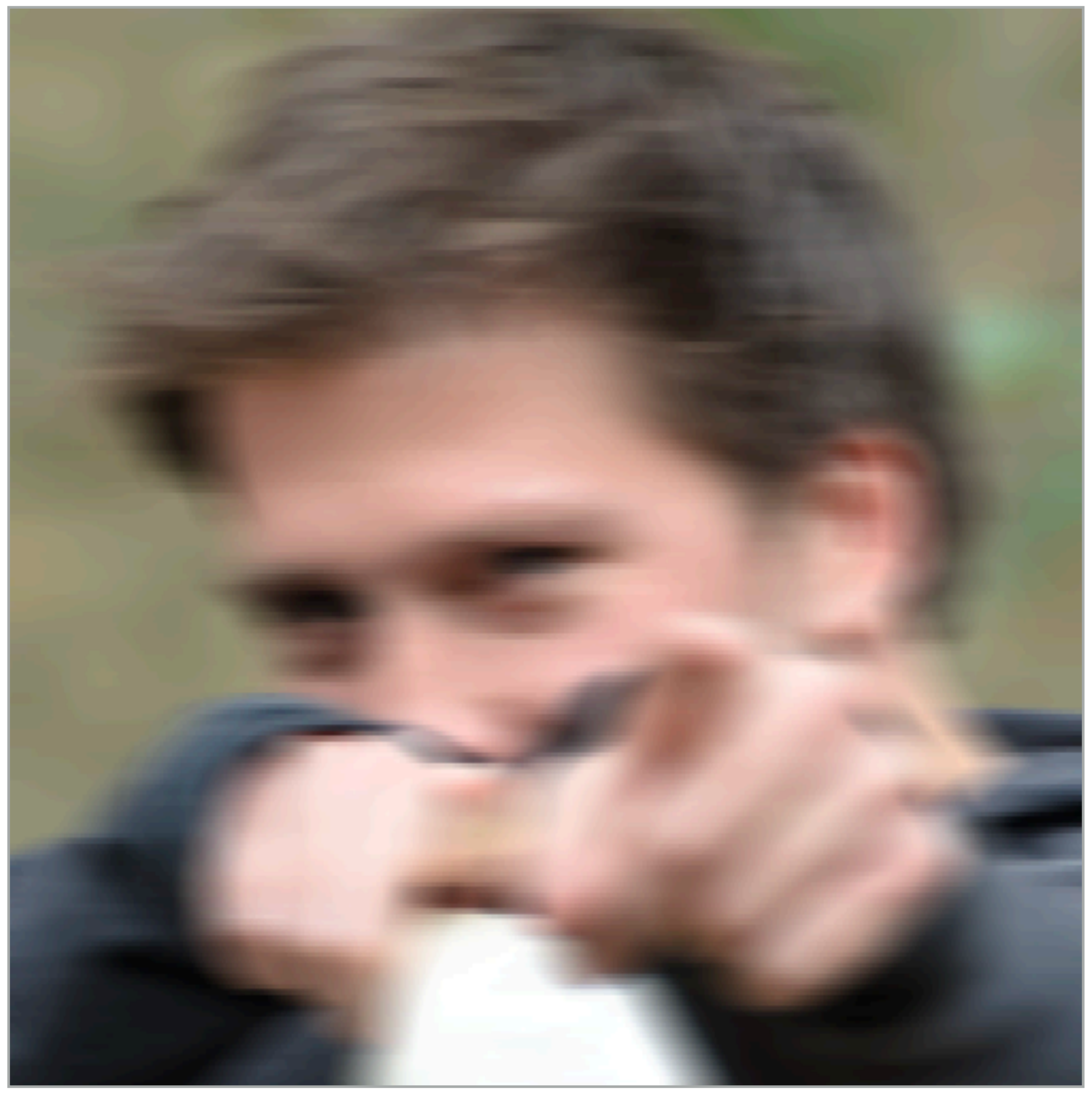
Total work per image = 9 x WIDTH x HEIGHT

For NxN filter: N^2 x WIDTH x HEIGHT

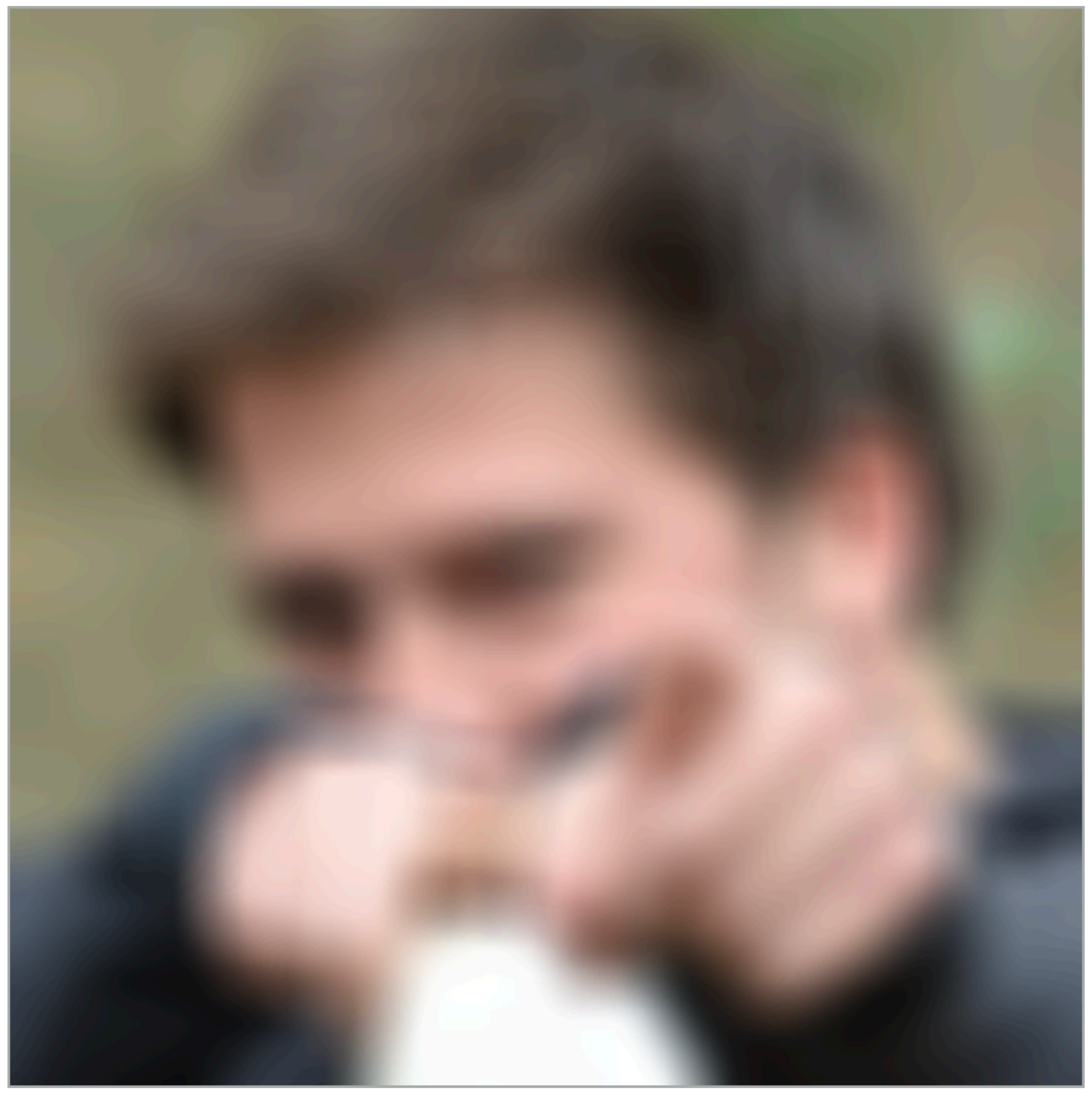
Two-pass blur

A 2D separable filter (such as a box filter) can be evaluated via two 1D filtering operations

Input



Horizontal Blur



Vertical Blur

Note: I've exaggerated the blur for illustration (the end result is 30x30 blur, not 3x3)

Two-pass 3x3 blur

Total work per image = 6 x WIDTH x HEIGHT

For NxN filter: 2N x WIDTH x HEIGHT

WIDTH x HEIGHT extra storage

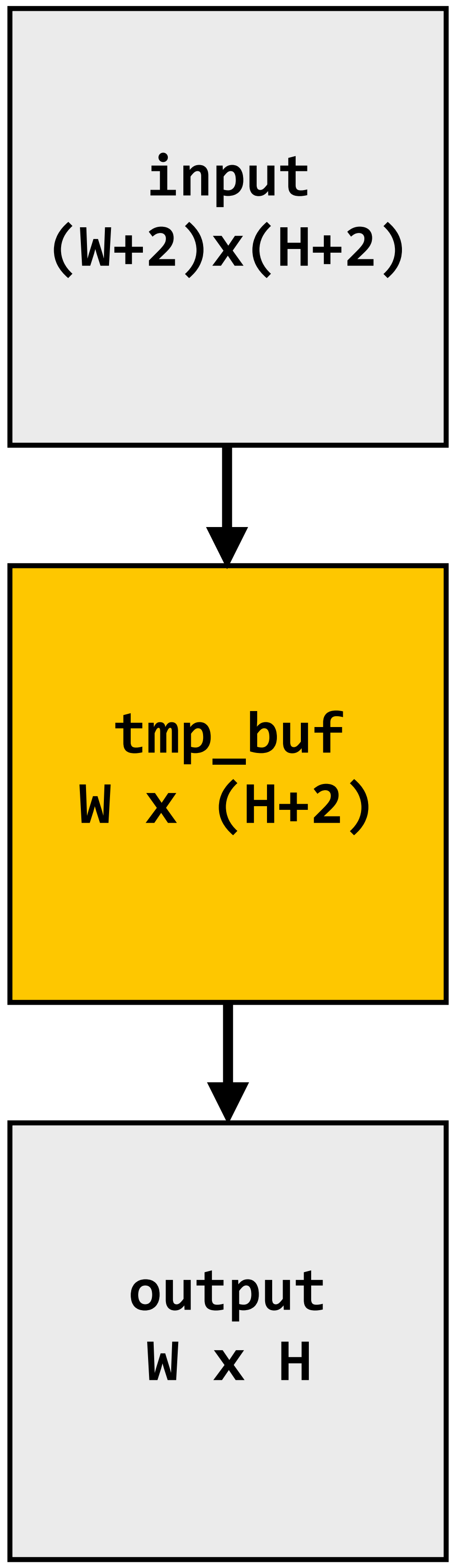
2X lower arithmetic intensity than 2D blur

```
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.f/3, 1.f/3, 1.f/3};

for (int j=0; j<(HEIGHT+2); j++)
  for (int i=0; i<WIDTH; i++) {
    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;
  }
```

1D horizontal blur

```
for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

1D vertical blur



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.f/3, 1.f/3, 1.f/3};

for (int j=0; j<(HEIGHT+2); j++)
  for (int i=0; i<WIDTH; i++) {
    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++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

Intrinsic bandwidth requirements of blur 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)

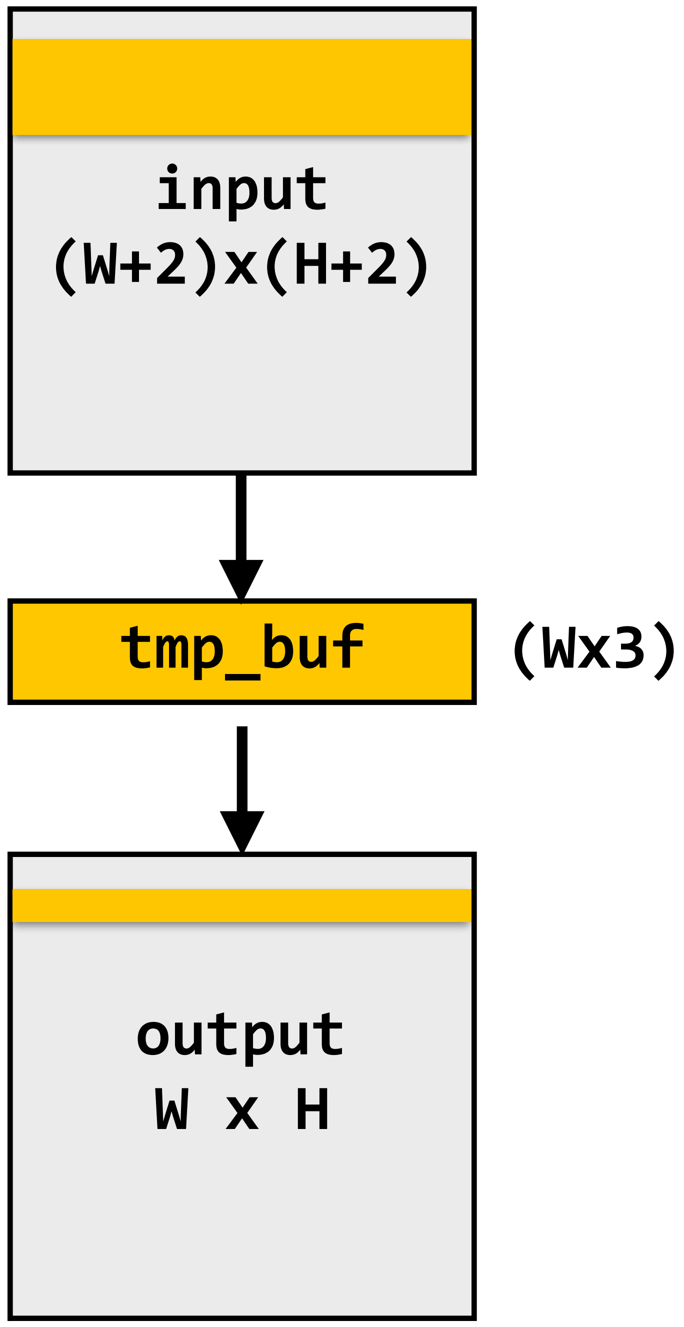
Data from `tmp_buf` reused three times (but three rows of image data are accessed in between)

- Never load required data more than once… if cache has capacity for <u>three rows of image</u>
- 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;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * 3];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<HEIGHT; j++) {

  for (int j2=0; j2<3; j2++)
    for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
      tmp_buf[j2*WIDTH + i] = tmp;

  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[jj*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

Only 3 rows of intermediate buffer need to be allocated



input (W+2)x(H+2)

tmp_buf (Wx3)

output W x H

Produce 3 rows of tmp_buf (only what’s needed for one row of output)

Combine them together to get one row of output

Total work per row of output:
- step 1: 3 x 3 x WIDTH work
- step 2: 3 x WIDTH work

Total work per image = 12 x WIDTH x HEIGHT ????

Loads from tmp_buffer are cached (assuming tmp_buffer fits in cache)

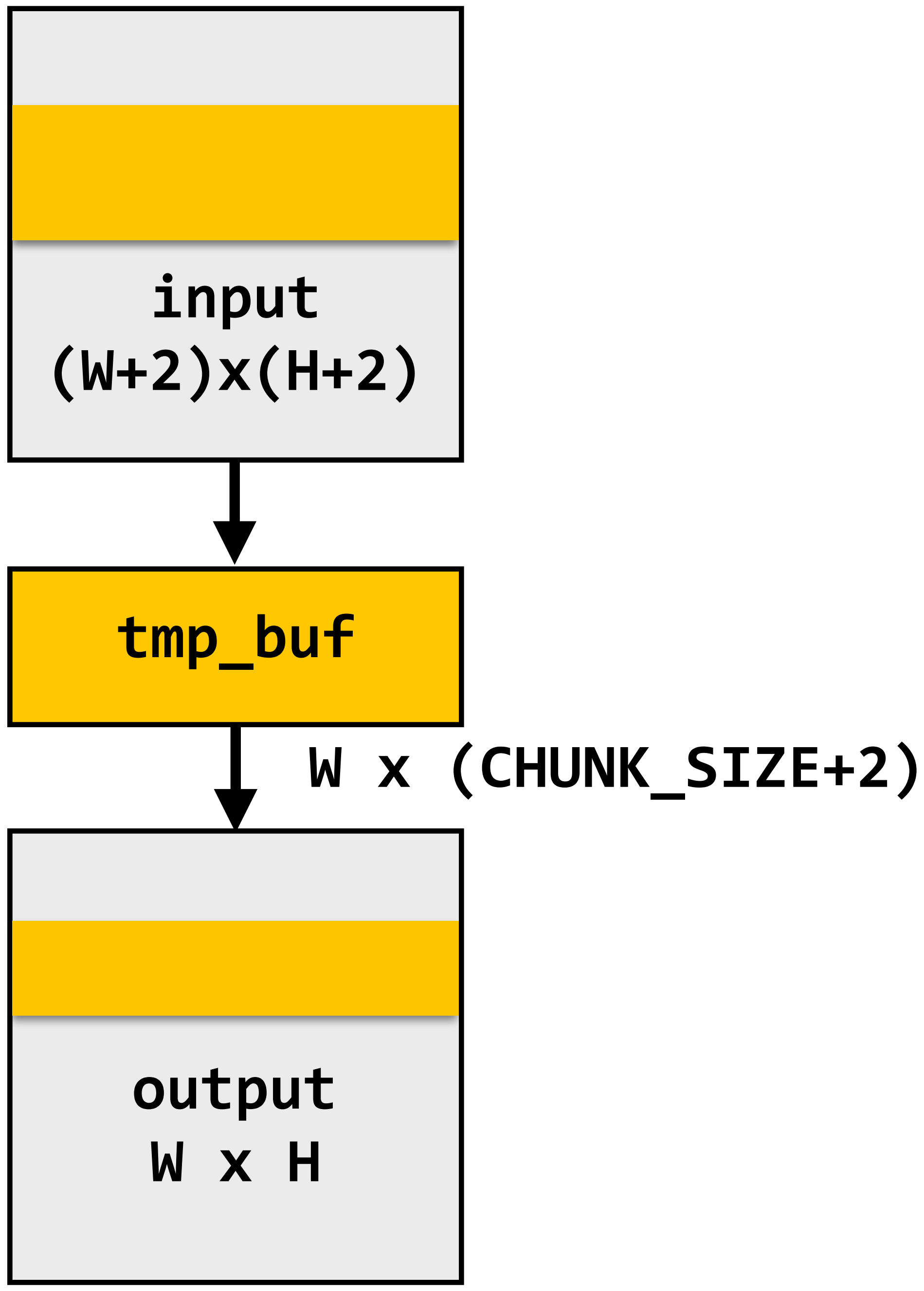
Two-pass image blur, “chunked” (version 2)

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (CHUNK_SIZE+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<HEIGHT; j+CHUNK_SIZE) {

  for (int j2=0; j2<CHUNK_SIZE+2; j2++)
    for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
      tmp_buf[j2*WIDTH + i] = tmp;

  for (int j2=0; j2<CHUNK_SIZE; j2++)
    for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int jj=0; jj<3; jj++)
        tmp += tmp_buf[(j2+jj)*WIDTH + i] * weights[jj];
      output[(j+j2)*WIDTH + i] = tmp;
    }
}
```

Sized so entire buffer fits in cache (capture all producer-consumer locality)

Produce enough rows of tmp_buf to produce a CHUNK_SIZE number of rows of output

Produce CHUNK_SIZE rows of output



Total work per chuck of output:
(assume CHUNK_SIZE = 16)

- Step 1: 18 x 3 x WIDTH work
- Step 2: 16 x 3 x WIDTH work

Total work per image: (34/16) x 3 x WIDTH x HEIGHT
= 6.4 x WIDTH x HEIGHT

Trends to ideal value of 6 x WIDTH x HEIGHT as CHUNK_SIZE is increased!

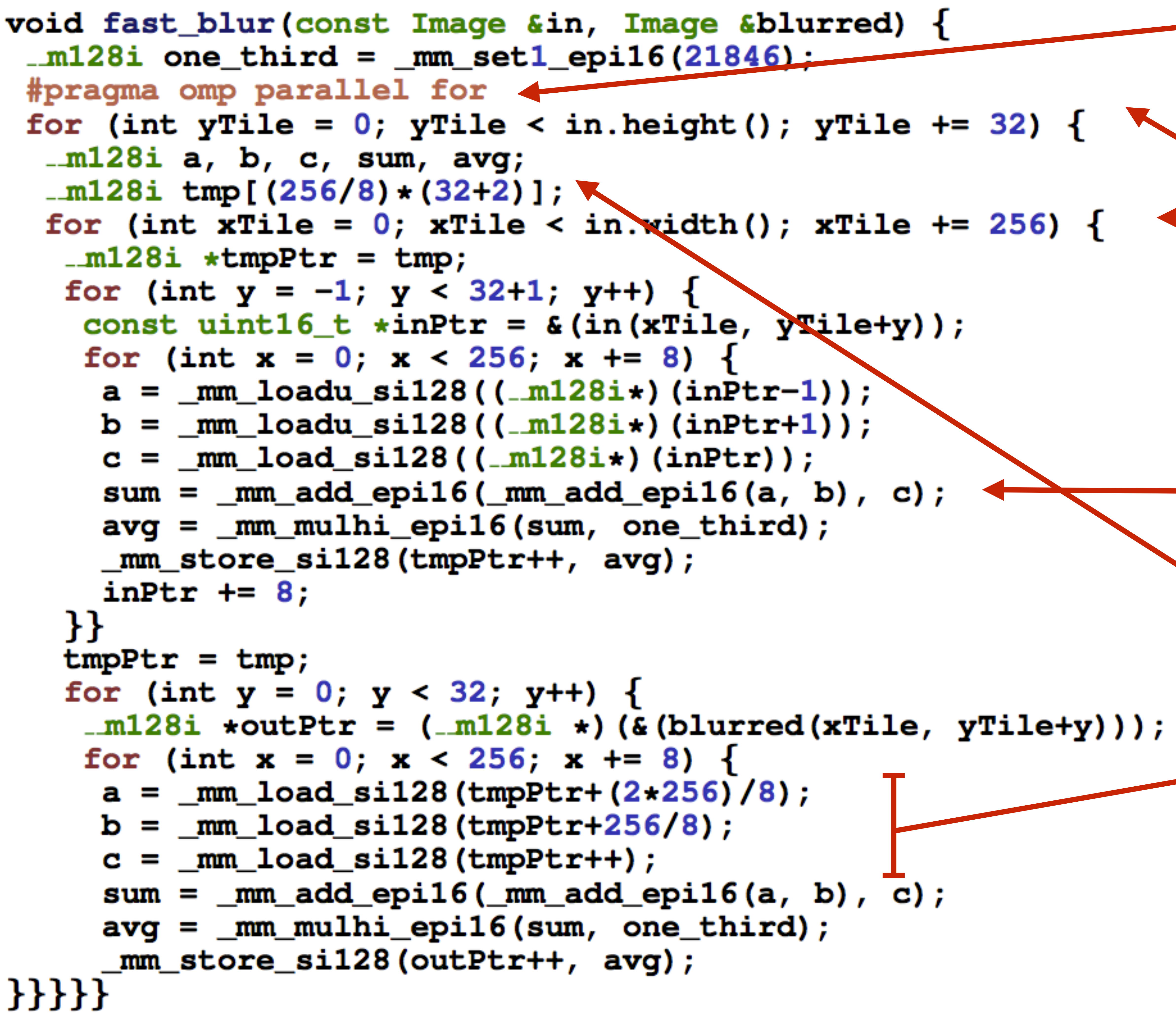
Still not done

- We have not parallelized loops for multi-core execution
- We have not used SIMD instructions to execute loops bodies
- Other basic optimizations: loop unrolling, etc…

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) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     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++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     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);
}}}}}
```

Multi-core execution (partition image vertically)

Modified iteration order: 256x32 tiled iteration (to maximize cache hit rate)

use of SIMD vector intrinsics

two passes fused into one: `tmp` data read from cache

Halide language

[Ragan-Kelley 2012]

Simple language embedded in C++ for describing sequences of image processing operations ("image processing pipelines")

```
Var x, y;
Func blurx, blury, out;
Image<uint8_t> in = load_image("myimage.jpg");

// perform 3x3 box blur in two-passes
blurx(x,y) = (in(x-1,y)    + in(x,y)      + in(x,y)) / 3.f;
blury(x,y) = (blurx(x,y-1) + blurx(x,y+1) + blurx(x,y+1)) / 3.f;

// brighten blurred result by 25%, then clamp
out(x,y) = min(blury(x,y) * 1.25f, 255);

// execute pipeline on domain of size 800x600
Image<uint8_t> result = out.realize(800, 600);
```

Functions map integer coordinates to values (e.g., colors of corresponding pixels)

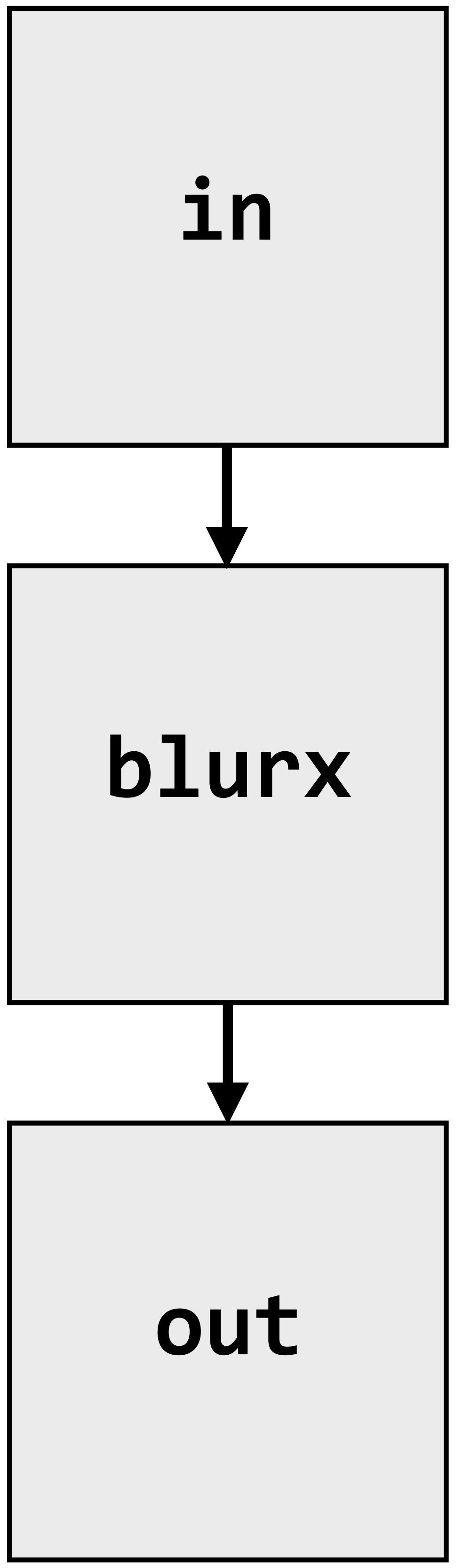
Value of `blurx` at coordinate (x,y) is given by expression accessing three values of `in`

- Halide function: an infinite (but discrete) set of values
- Halide expression: a side-effect free expression describes how to compute a function's value at a point in it's domain in terms of the values of other functions.

Key aspects of Halide's design

- Local "pointwise" view of expressing algorithms
- Language is constrained so that iteration over domain points is implicit (no explicit loops in Halide)
  - Halide is declarative. It does not define order of iteration, or what values in domain or stored!
  - It only defines what operations are needed to compute these values.

```
Var x, y;
Func blurx, out;
Image<uint8_t> in = load_image(“myimage.jpg”);

// perform 3x3 box blur in two-passes
blurx(x,y) = (in(x-1,y)    + in(x,y)      + in(x,y)) / 3.f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y+1) + blurx(x,y+1)) / 3.f;

// execute pipeline on domain of size 800x600
Image<uint8_t> result = our.realize(800, 600);
```



Real-world image processing pipelines feature complex sequences of functions

Benchmark	Number of Halide functions
Two-pass blur	2
Unsharp mask	9
Harris Corner detection	13
Camera RAW processing	30
Non-local means denoising	13
Max-brightness filter	9
Multi-scale interpolation	52
Local-laplacian filter	103
Synthetic depth-of-field	74
Bilateral filter	8
Histogram equalization	7
VGG-16 deep network eval	64

Real-world production applications may features hundreds to thousands of functions!

Google HDR+ pipeline: over 2000 Halide functions.

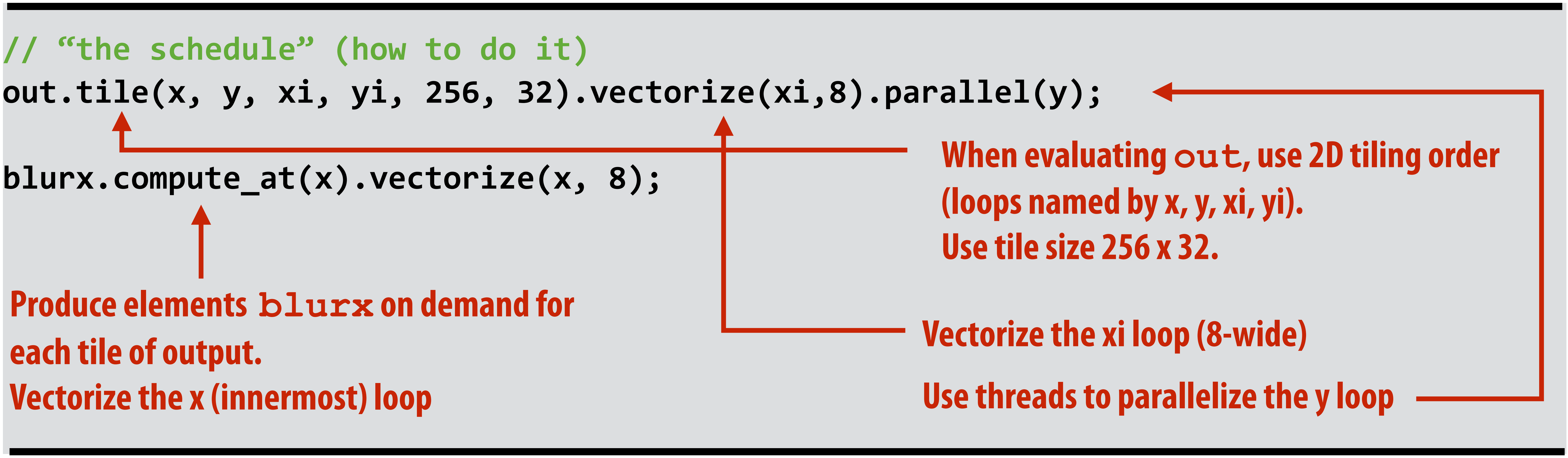
Key aspect in the design of any system:

Choosing the “right” representations for the job

Now the job is not expressing an image processing computation, but generating an efficient implementation of a specific Halide program.

A second set of representations for “scheduling”

```
Func blurx, out;
Var  x, y, xi, yi;
Image<uint8_t> in = load_image(“myimage.jpg”);

// the “algorithm description”  (declaration of 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.compute_at(x).vectorize(x, 8);
```

When evaluating `out`, use 2D tiling order (loops named by x, y, xi, yi). Use tile size 256 x 32.

Produce elements `blurx` on demand for each tile of output. Vectorize the x (innermost) loop

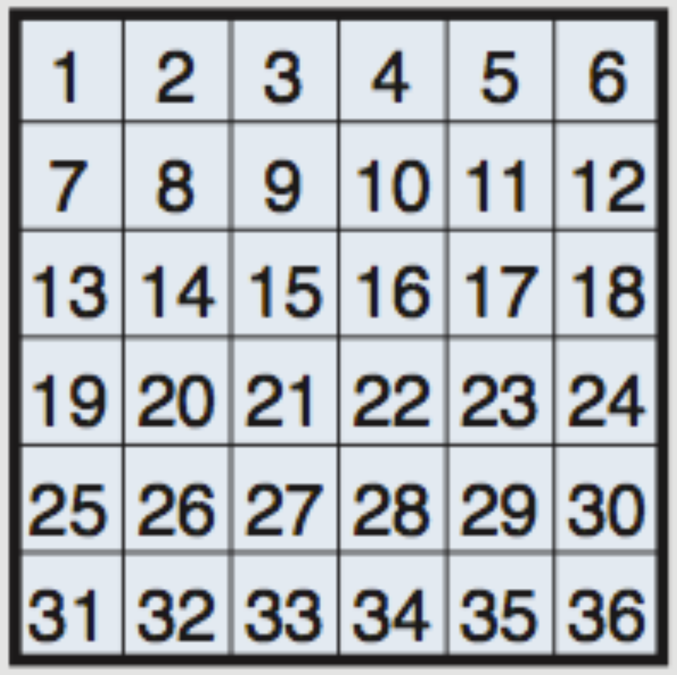
Vectorize the xi loop (8-wide)

Use threads to parallelize the y loop

```
// execute pipeline on domain of size 1024x1024
Image<uint8_t> result = out.realize(1024, 1024);
```

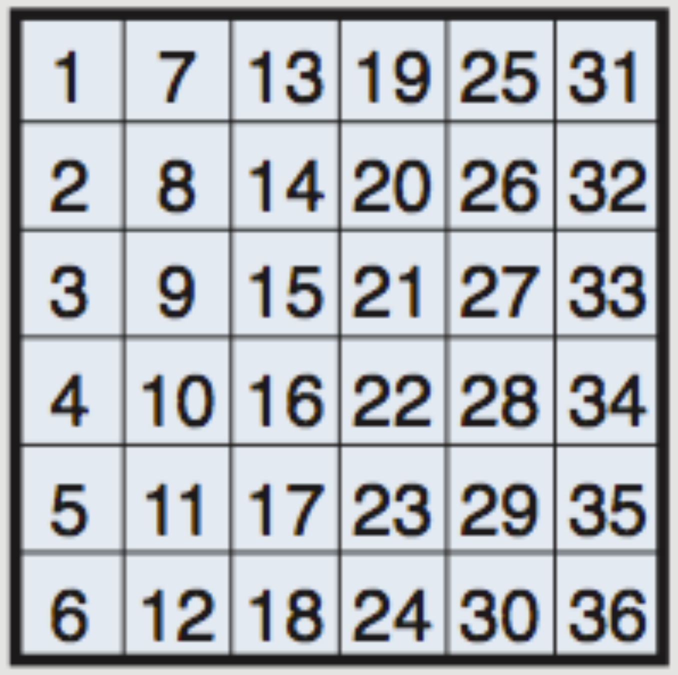
Scheduling primitives allow the programmer to specify a global “sketch” of how to schedule the algorithm onto a parallel machine, but leave the details of emitting the low-level platform-specific code to the Halide compiler

Primitives for iterating over domains



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

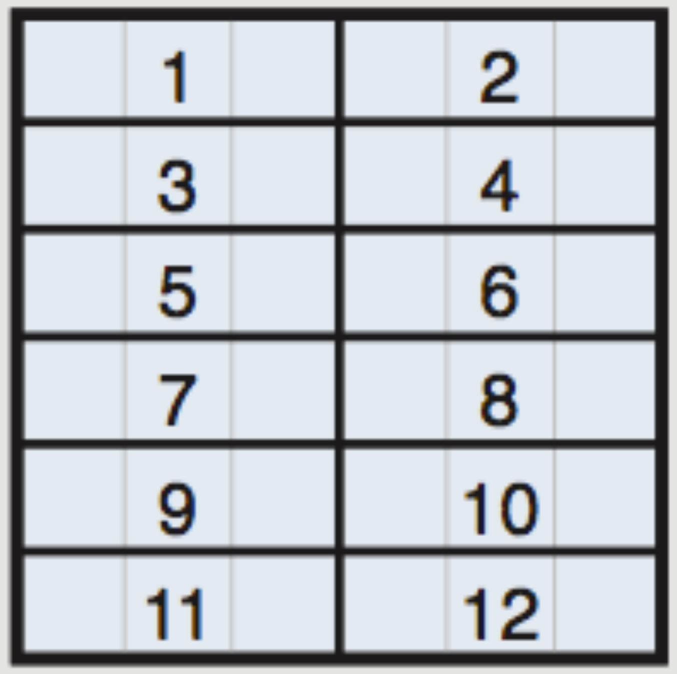
serial y, serial x



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

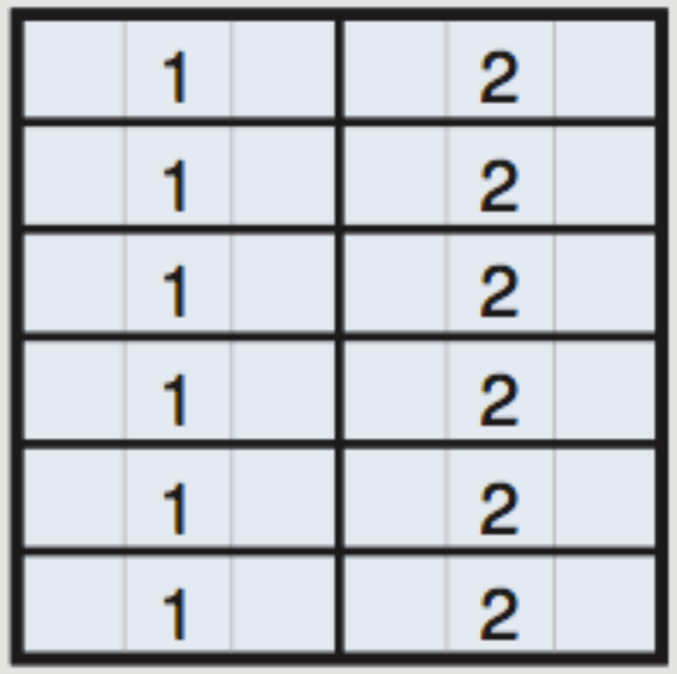
serial x, serial y

Specify both order and how to parallelize (multi-thread, vectorize via SIMD instr)



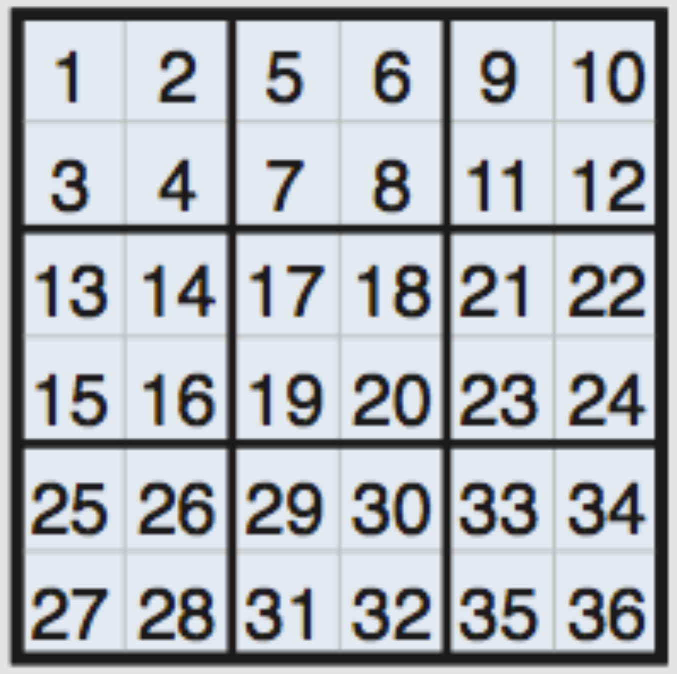
1	2
3	4
5	6
7	8
9	10
11	12

serial y
vectorized x



1	2
1	2
1	2
1	2
1	2
1	2

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

Specifying loop iteration order and parallelism

```
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;
```

Given this schedule for the function "out"...

```
out.tile(x, y, xi, yi, 256, 32).vectorize(xi,8).parallel(y);
```

Halide compiler will generate this parallel, vectorized loop nest for computing elements of `out`...

```
for y=0 to num_tiles_y:  // parallelize this loop over multiple threads
   for x=0 to num_tiles_x:
      for yi=0 to 32:
         for xi=0 to 256: // vectorize this loop with SIMD instructions
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_y, idx_y) = …
```

Primitives for how to interleave producer/ consumer processing

```
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;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_root();
```

Do not compute blurx within out's loop nest.

Compute all of blurx, then all of out

```
allocate buffer for all of blur(x,y)
for y=0 to HEIGHT:
  for x=0 to WIDTH:
     blurx(x,y) = …
```

all of blurx is computed here

```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:
      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_y, idx_y) = …
```

values of blurx consumed here

Primitives for how to interleave producer/consumer processing

```
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;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_at(x_i);
```

Compute necessary elements of blurx within out’s xi loop nest

```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:
      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi

            allocate 3-element buffer for blurx
            // compute 3 elements of blurx needed for out(idx_x, idx_y) here

            out(idx_y, idx_y) = …
```

Primitives for how to interleave producer/ consumer processing

```
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;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_at(x);
```

Compute necessary elements of blurx within out's x loop nest (all necessary elements for one tile of out)

```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:

      allocate 258x34 buffer for tile blurx
      for yi=0 to 32+2:
         for xi=0 to 256+2:
            blur(xi,yi) = // compute blurx from in

      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_y, idx_y) = …
```

tile of blurx is computed here

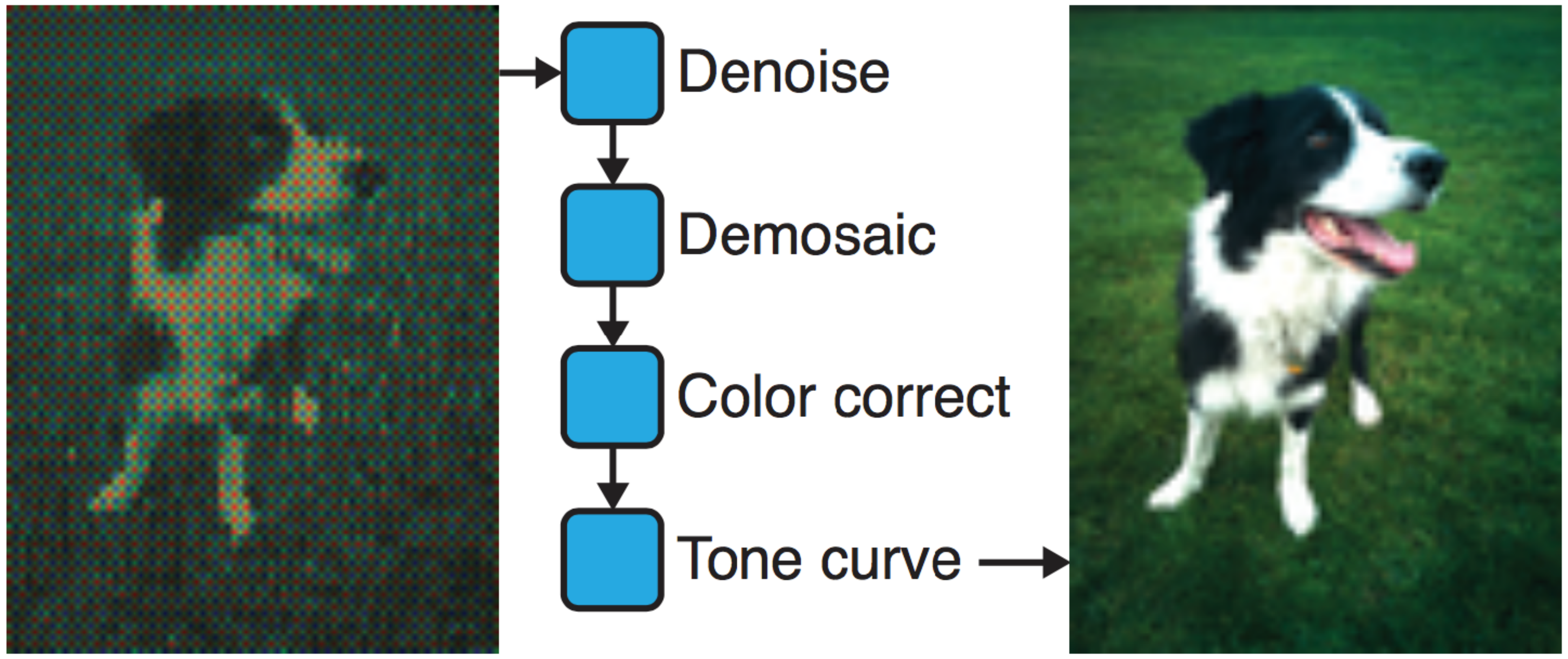
tile of blurx is consumed here

Halide: two domain-specific co-languages

- Functional primitives for describing image processing operations
- Additional primitives for describing schedules
- Design principle: separate "algorithm specification" from 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 exploration of space of schedules ("tile these loops", vectorize this loop", "parallelize this loop across cores")
- Application 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

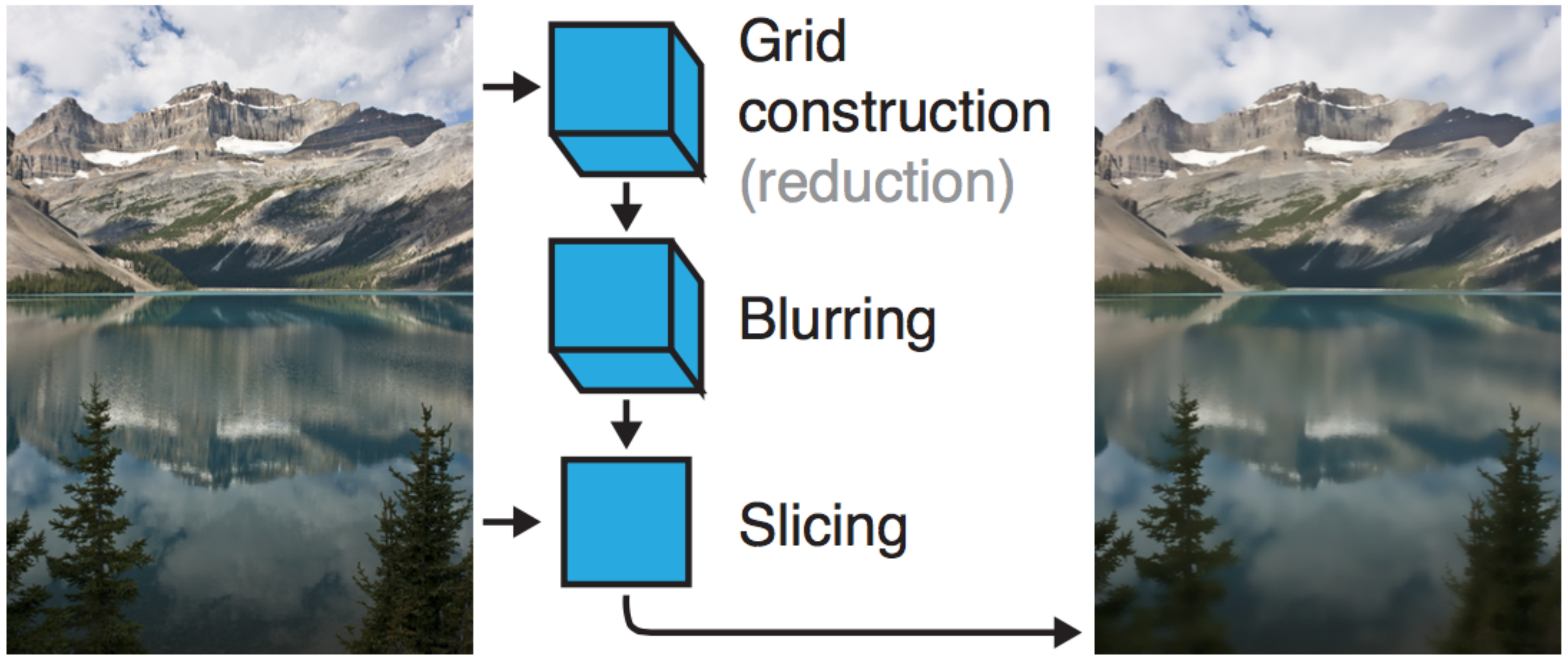
Early Halide results

[Ragan-Kelley 2012]



- 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?

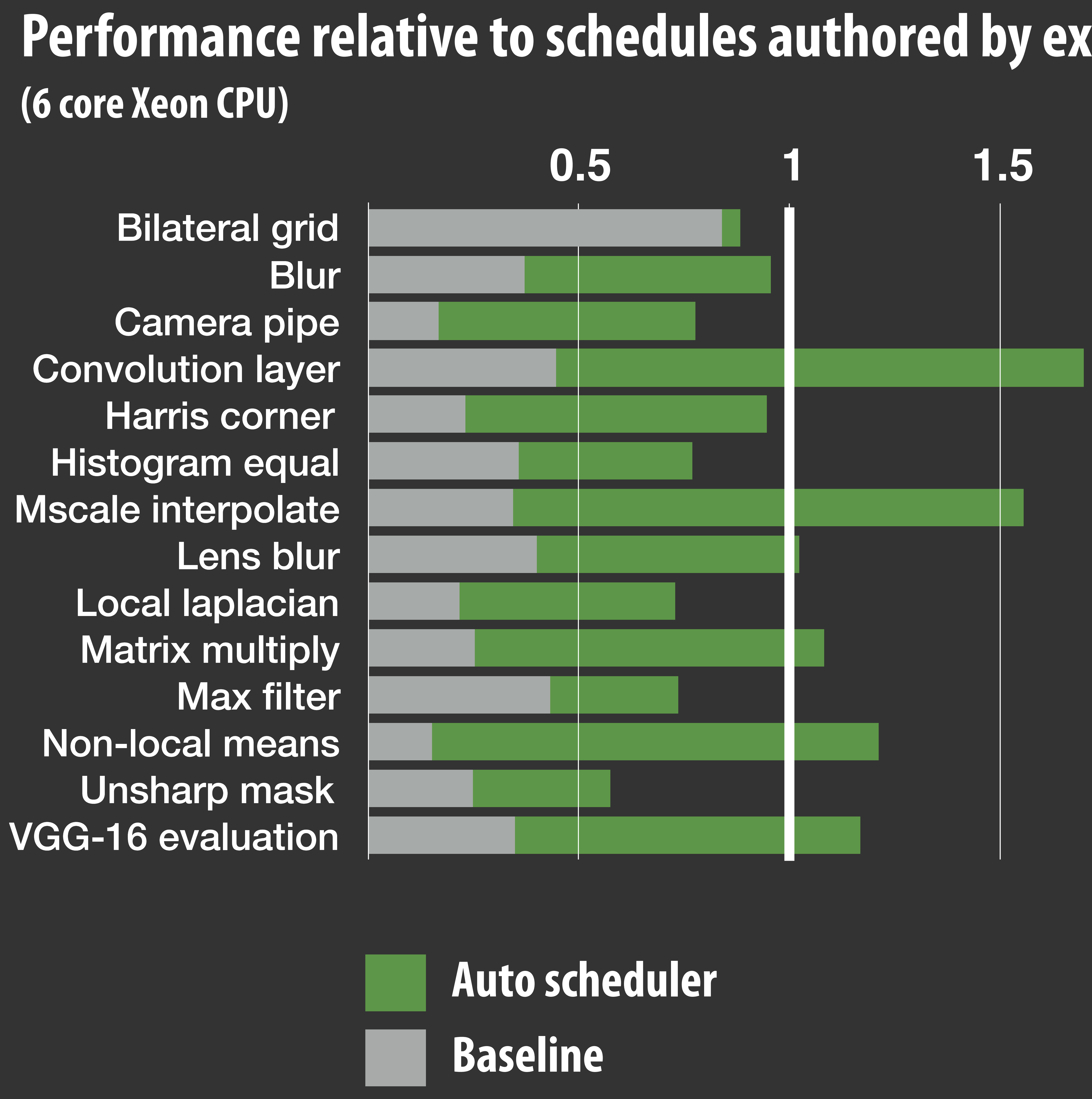
- Halide is a DSL for helping expert developers optimize image processing code more rapidly
  - Halide does not decide how to optimize a program for a novice programmer
  - Halide provides primitives for a programmer (that has strong knowledge of code optimization, such as a 15-418 student) to rapidly express what optimizations the system should apply
  - Halide compiler carries out the nitty-gritty of mapping that strategy to a machine

Automatically generating schedules

- Problem: it turned out that very few programmers have the ability to write good Halide schedules
  - 80+ programmers at Google write Halide
  - Very small number trusted to write schedules

- Recent work: analyzing the Halide program to automatically generate efficient schedules for the user
  - Talk to Ravi! [Mullapudi 2016]

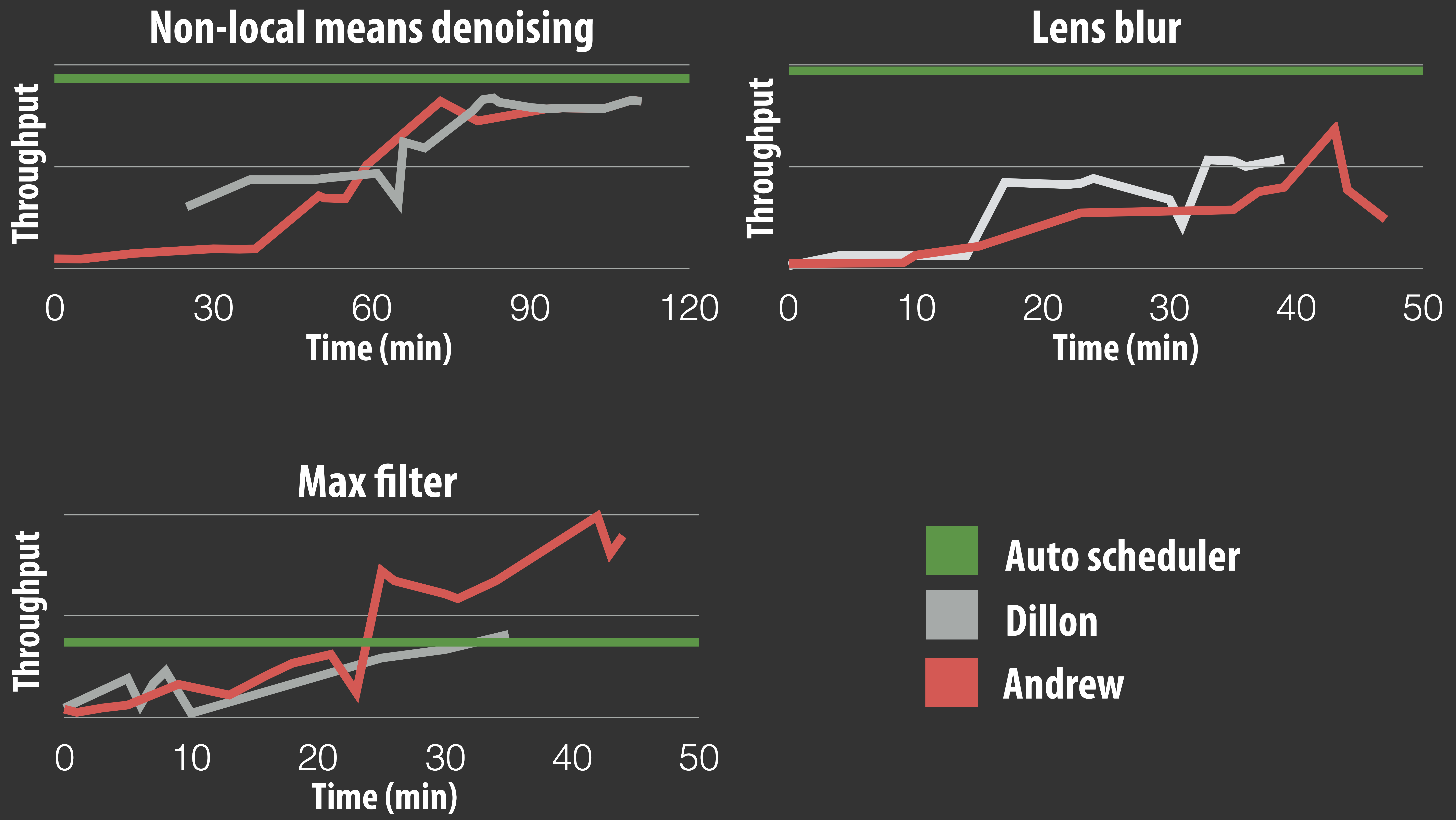
Autoscheduler performs comparably to experts



On 8 of the 14 benchmarks performance within 10% of experts or better

Baseline schedules exploit multi-core/vector parallelism and pointwise inlining but no global locality optimizations

Halide auto scheduler vs. experts



Non-local means denoising

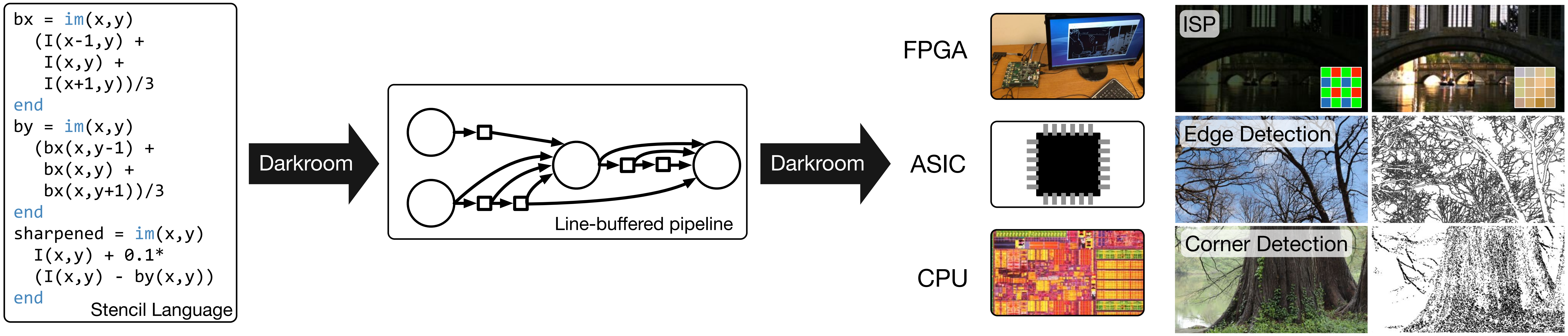
Lens blur

Max filter

Darkroom/Rigel

[Hegarty 2014, Hegarty 2016]

Goal: directly synthesize FGPA implementation of image processing pipelines from a high-level description (a constrained “Halide-like” language)



```
bx = im(x,y)
  (I(x-1,y) +
   I(x,y) +
   I(x+1,y))/3
end
by = im(x,y)
  (bx(x,y-1) +
   bx(x,y) +
   bx(x,y+1))/3
end
sharpened = im(x,y)
  I(x,y) + 0.1*
  (I(x,y) - by(x,y))
end
```

Seeking very-high efficiency image processing

Many other recent domain-specific programming systems

hadoop

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

GraphLab
Carnegie Mellon

DSL for graph-based machine learning computations
Also see Ligra
(DSLs for describing operations on graphs)

RAILS

Model-view-controller paradigm for web-applications

TensorFlow

DSL for defining deep neural networks and training/inference computations on those networks

OpenGL®

Language for real-time 3D graphics

julia

Numerical computing

Ongoing efforts in many domains...

Languages for physical simulation: Simit [MIT], Ebb [Stanford]
Opt: a language for non-linear least squares optimization [Stanford]

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
  - Leverage explicit dependencies, domain restrictions, domain knowledge for system to synthesize efficient implementations