Lecture 14:

Domain-Specific Programming Systems

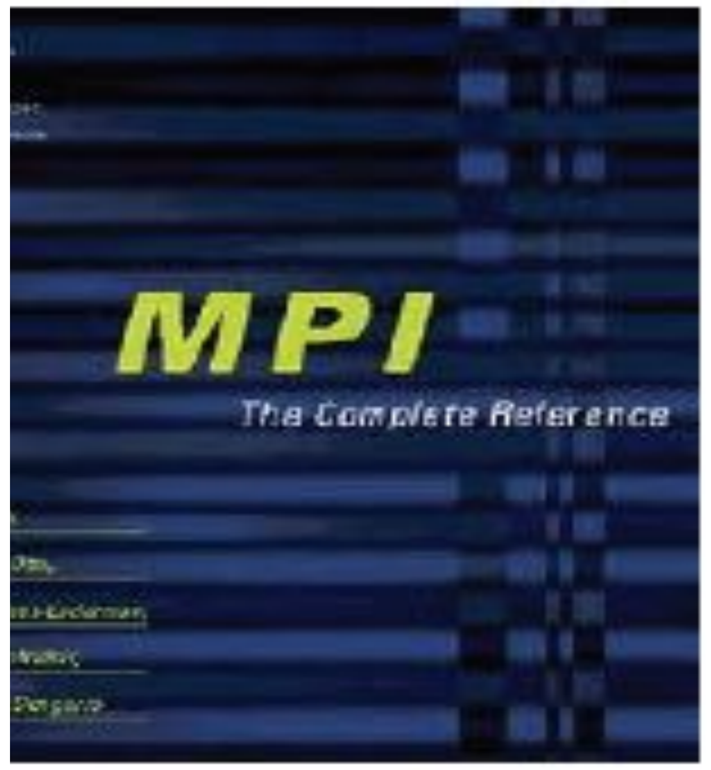
Parallel Computing
Stanford CS149, Fall 2023

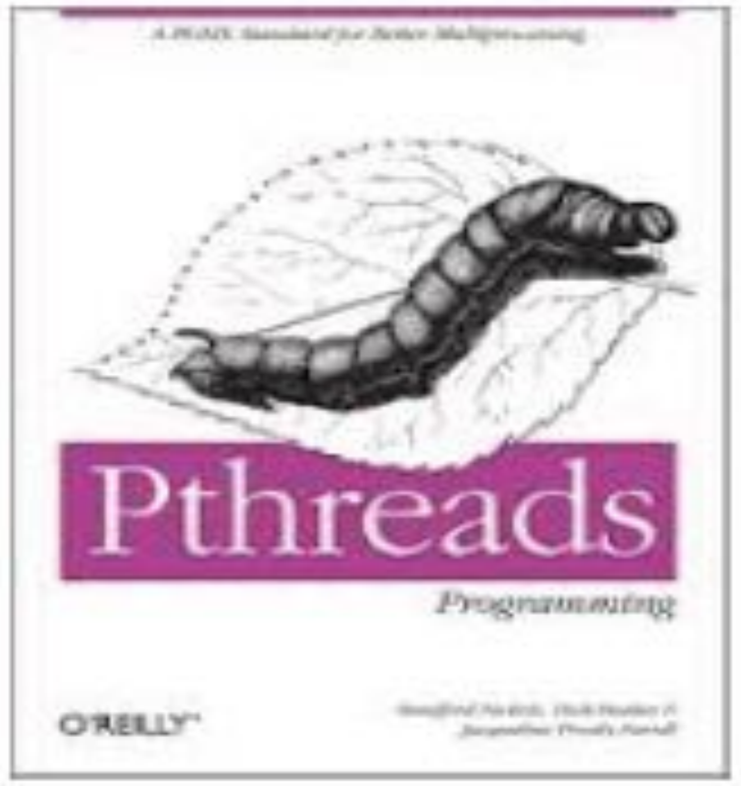
Today

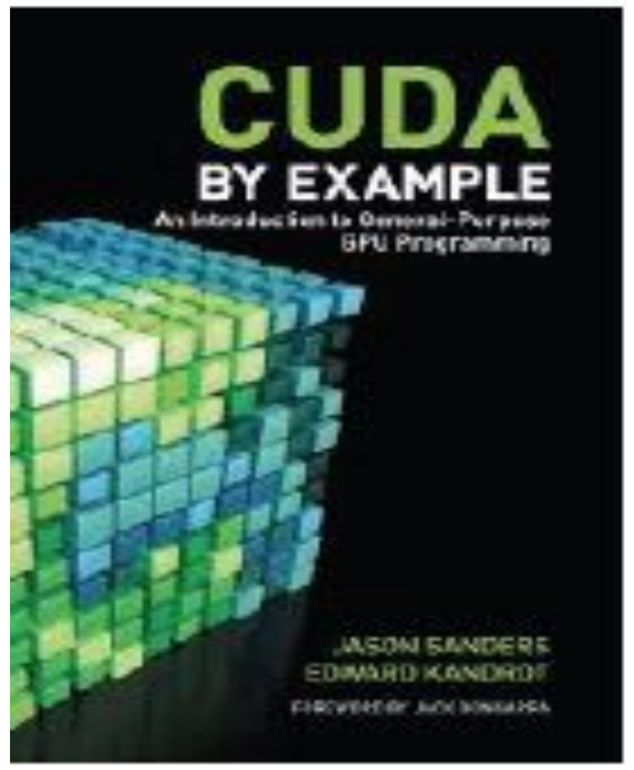
- Deeper dive into the idea of choosing the right abstractions for the job
- What is a domain specific programming language (DSL)?
- Two concrete examples:
    - Image processing in Halide
    - Computational fluid dynamics in Lizst

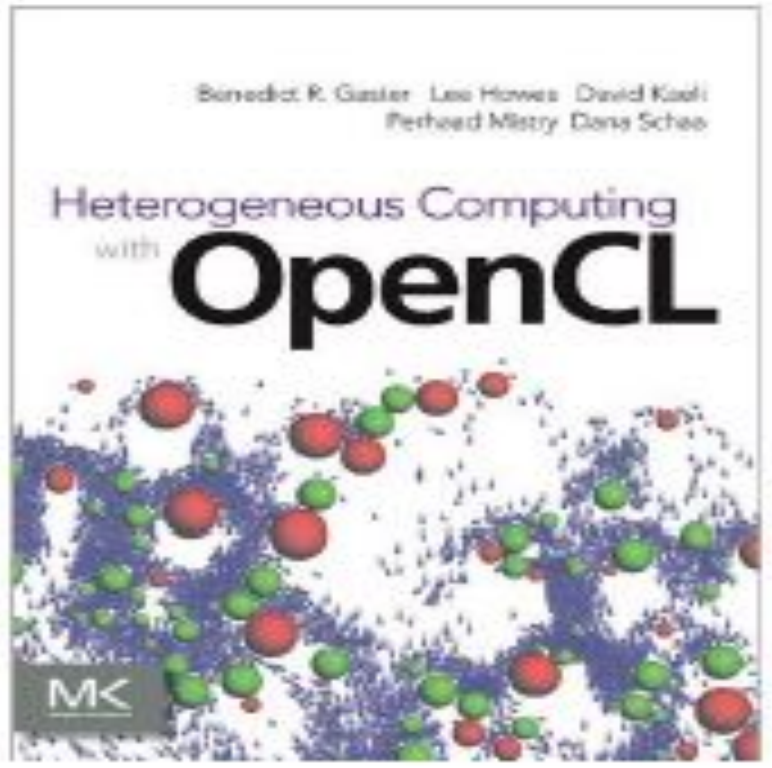
- Key concept: what are the advantages of performance-oriented application development using DSLs

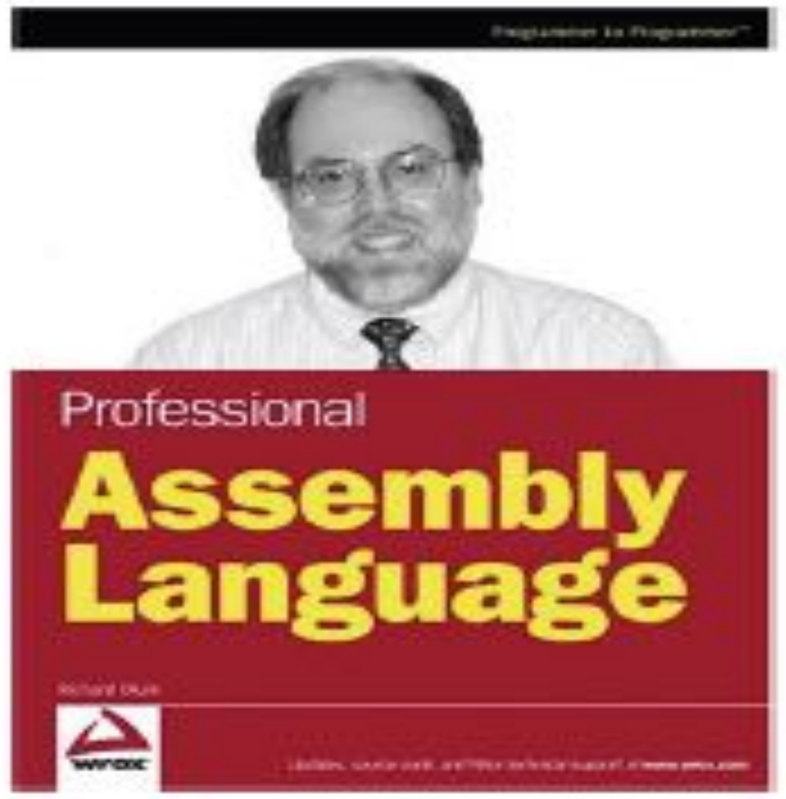
CS149 educated programmers = hard to find
Performance optimization in languages like C++ (threads), ISPC, CUDA = low productivity
(Proof by assignments 1, 2, 3, etc…)



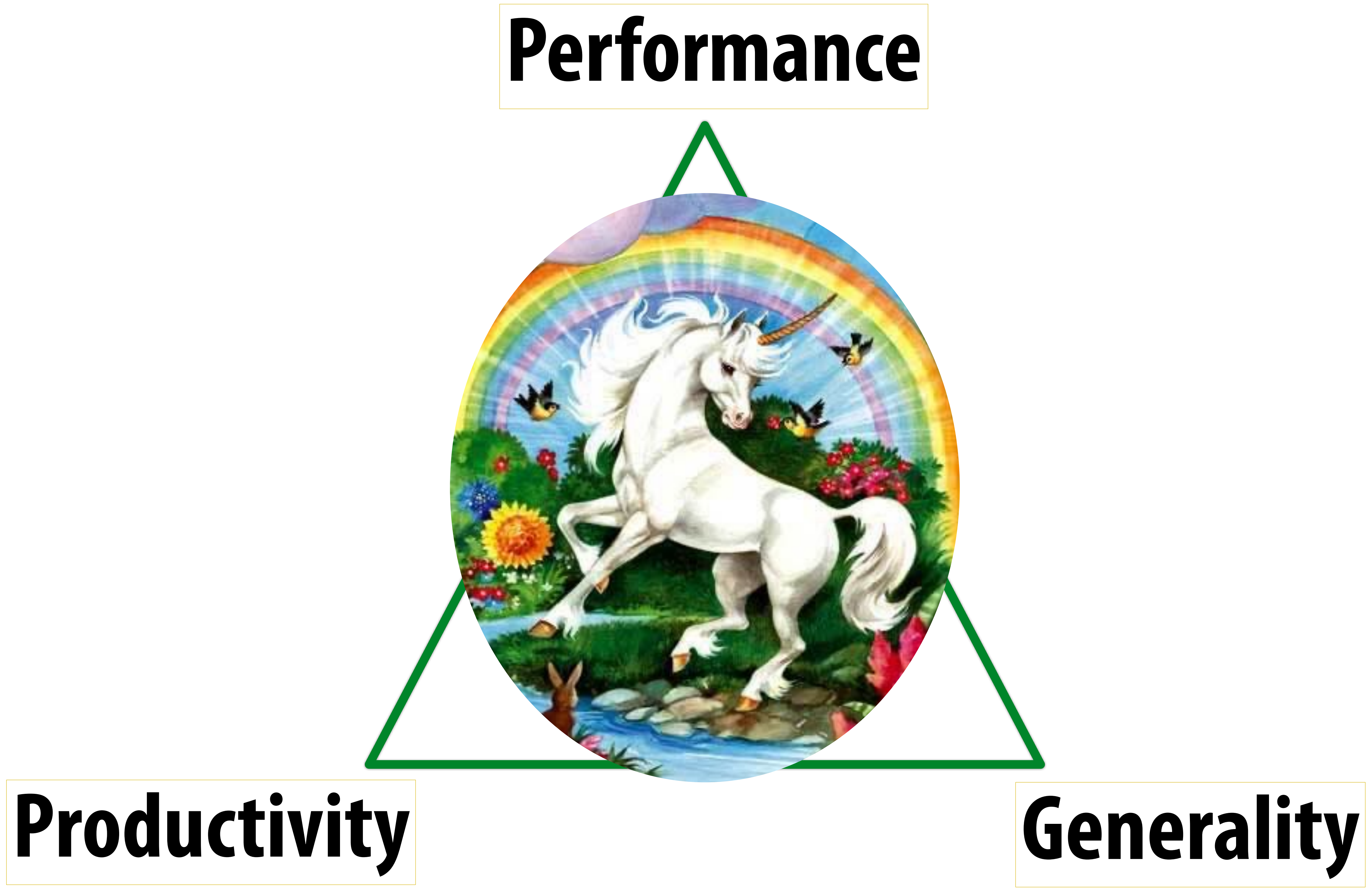






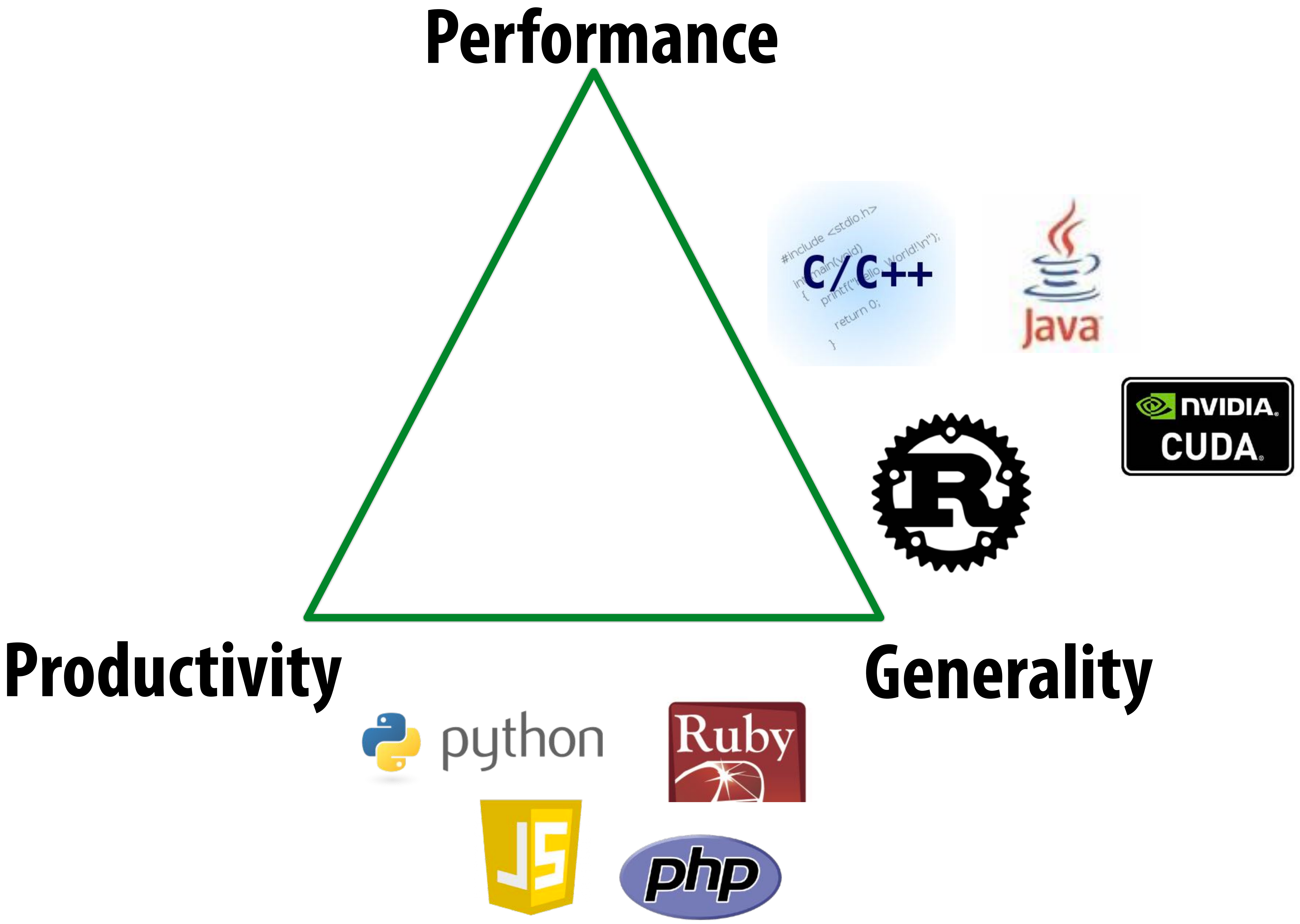


The ideal parallel programming language



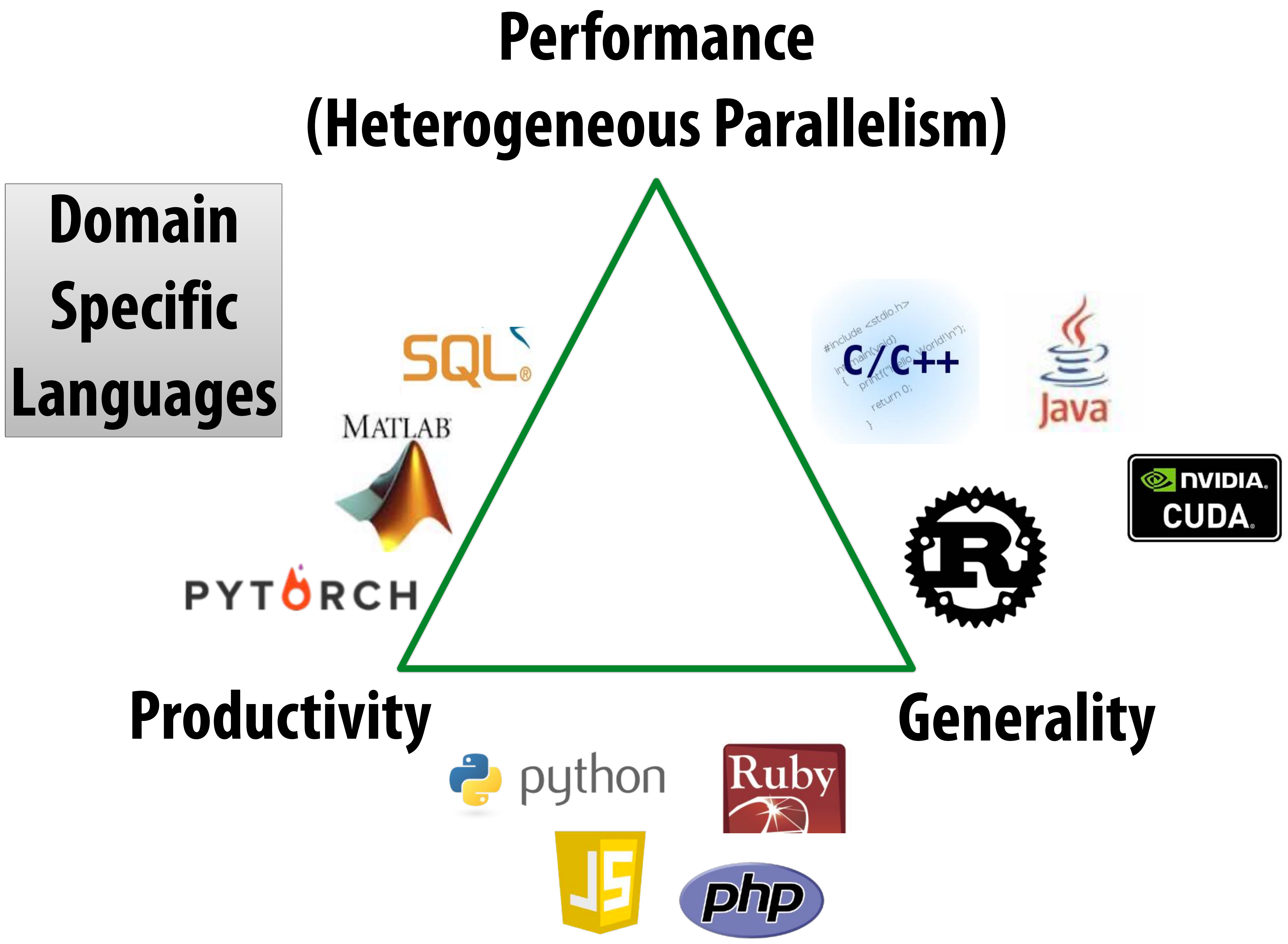
Credit: Pat Hanrahan for this slide design

Popular languages (not exhaustive ;-))



Credit: Pat Hanrahan for this slide design

Way forward ⇒ domain-specific languages



Credit: Pat Hanrahan for this slide design

DSL hypothesis

It is possible to write one program…

and

run it efficiently on a range of heterogeneous parallel systems

Domain Specific Languages

- Domain Specific Languages (DSLs)
  - Programming language with restricted expressiveness for a particular domain
  - High-level, usually declarative, and deterministic

Domain-specific programming systems

- Main idea: raise level of abstraction for expressing programs
  - Goal: write one program, and run it efficiently on different machines
- 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

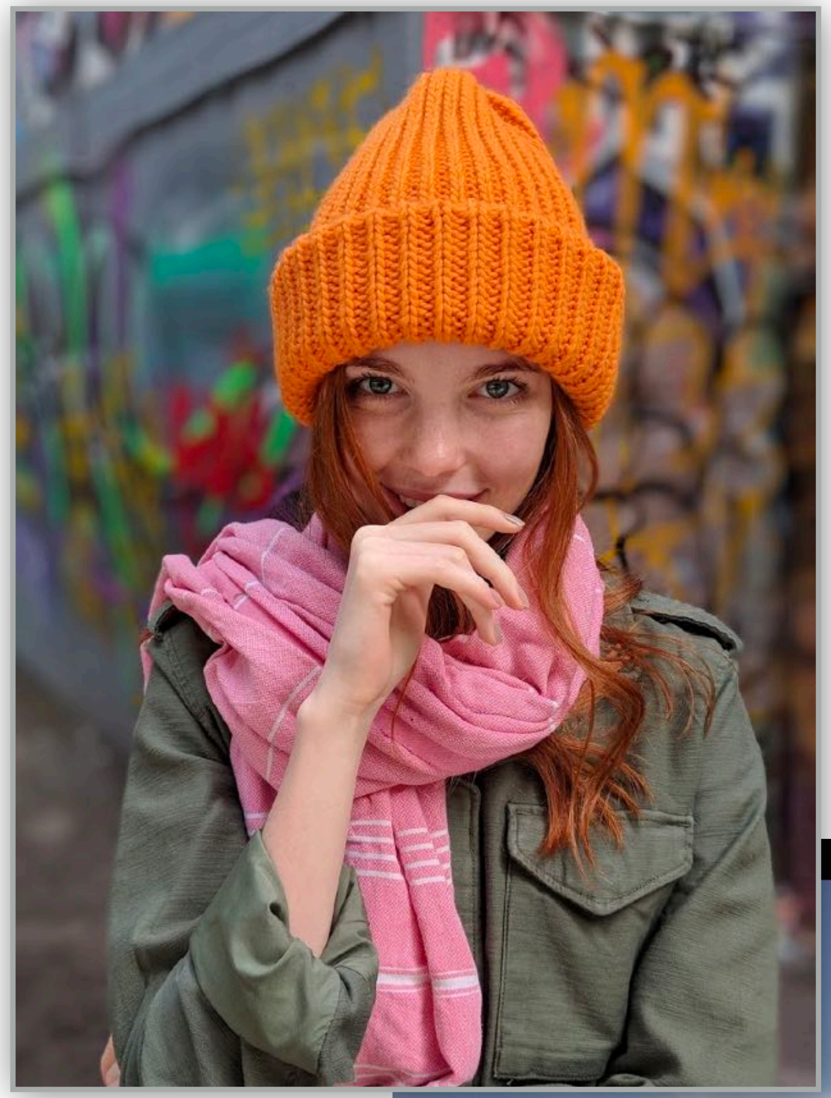
A DSL example:

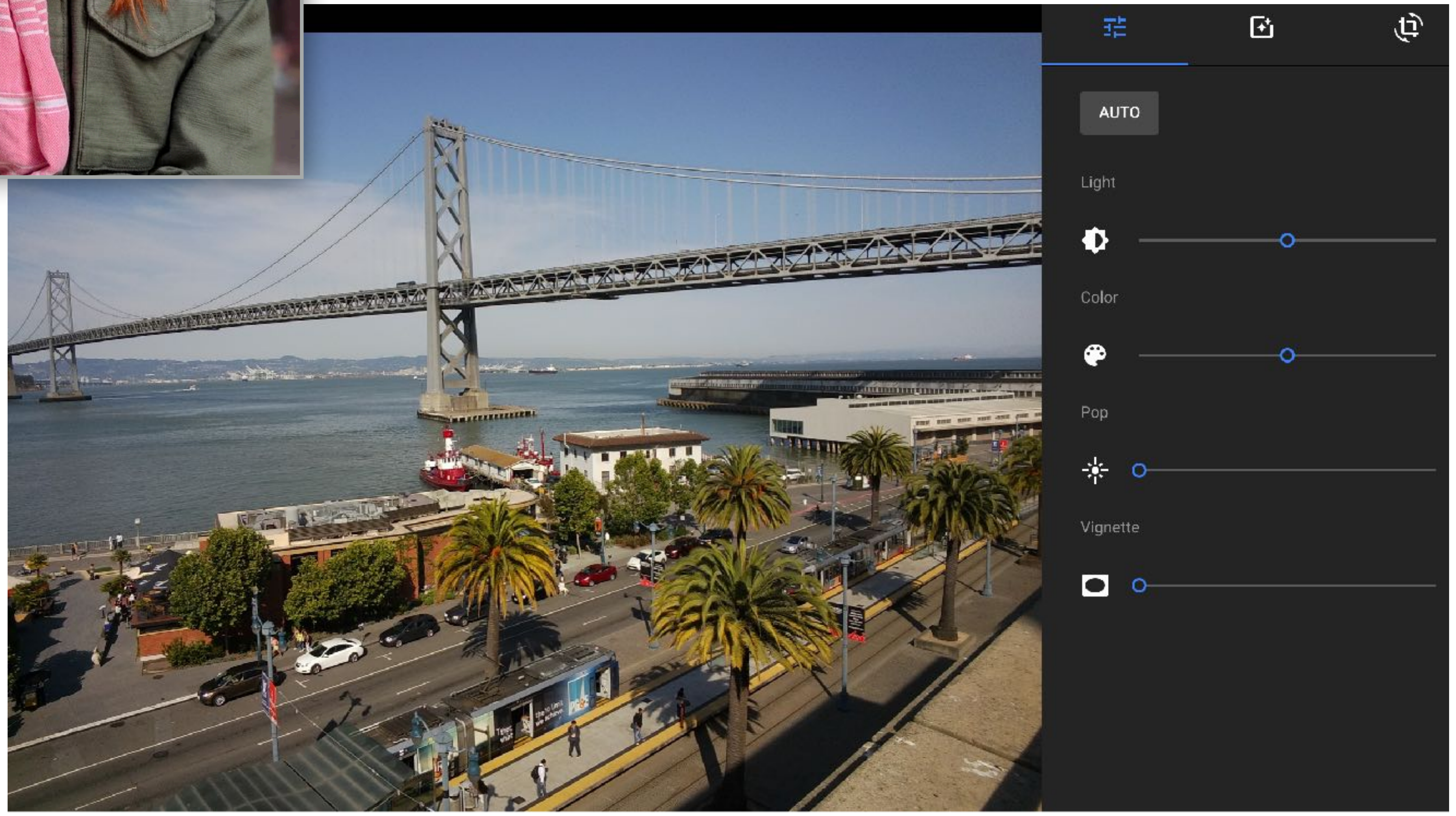
Halide: a domain-specific language for image processing

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

Halide used in practice

- Halide used to implement camera processing pipelines on Google phones
  - HDR+, aspects of portrait mode, etc…
- Industry usage at Instagram, Adobe, etc.





A quick tutorial on high-performance image processing

What does this code do? 🤔😱😩😭

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);
}}}}}
```

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;
  }
}
```

The code on the previous slide performed a 3x3 box blur

(Zoomed 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

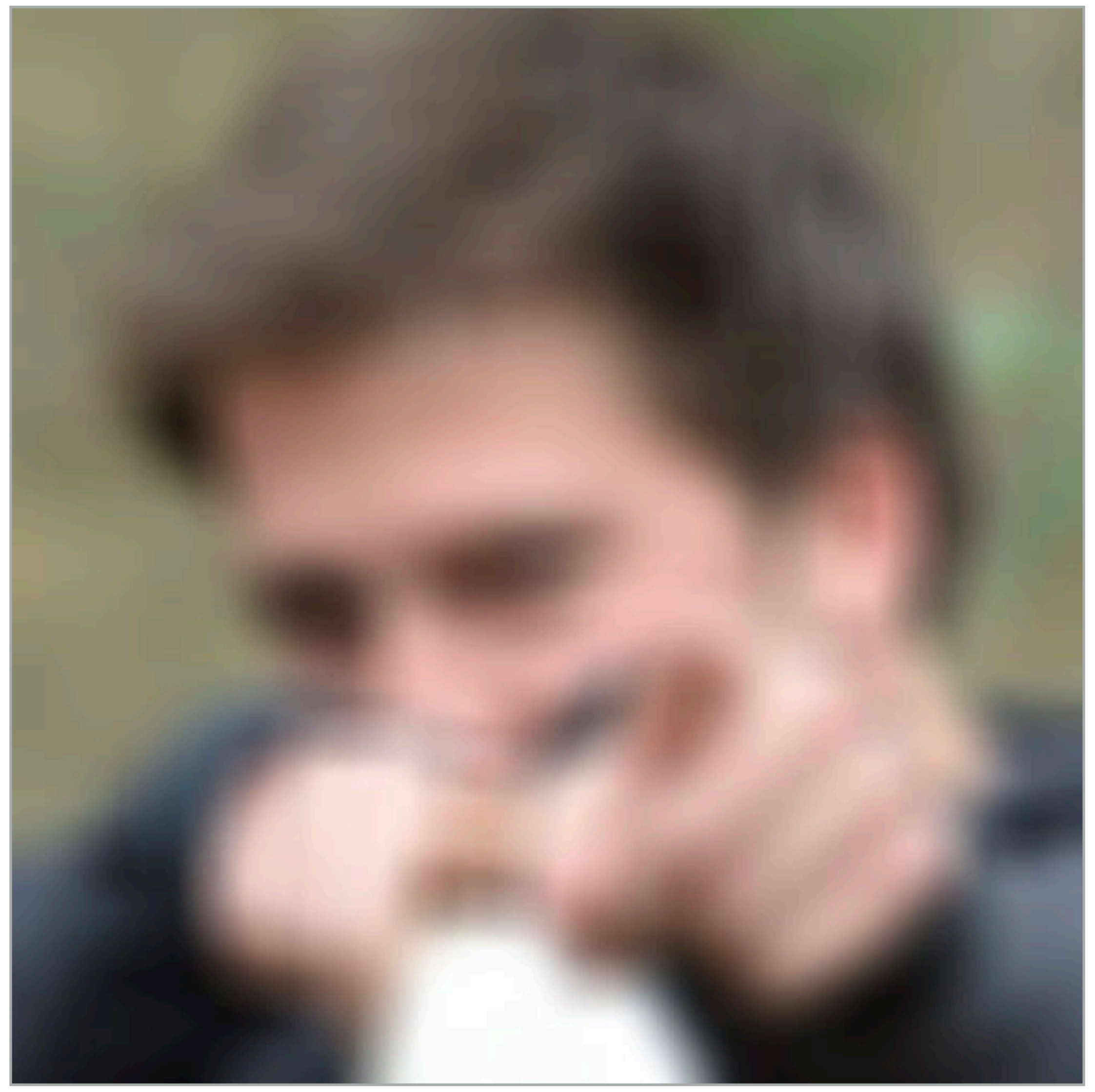
For NxN filter: N^2 x WIDTH x HEIGHT

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 actually a 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. Why?

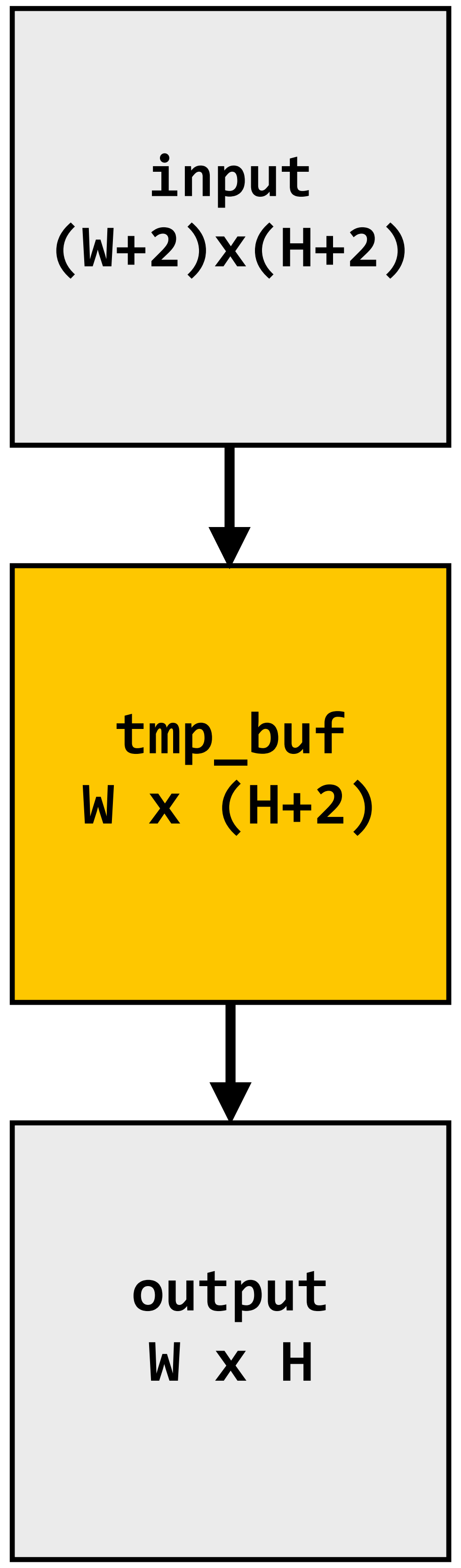
```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

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)

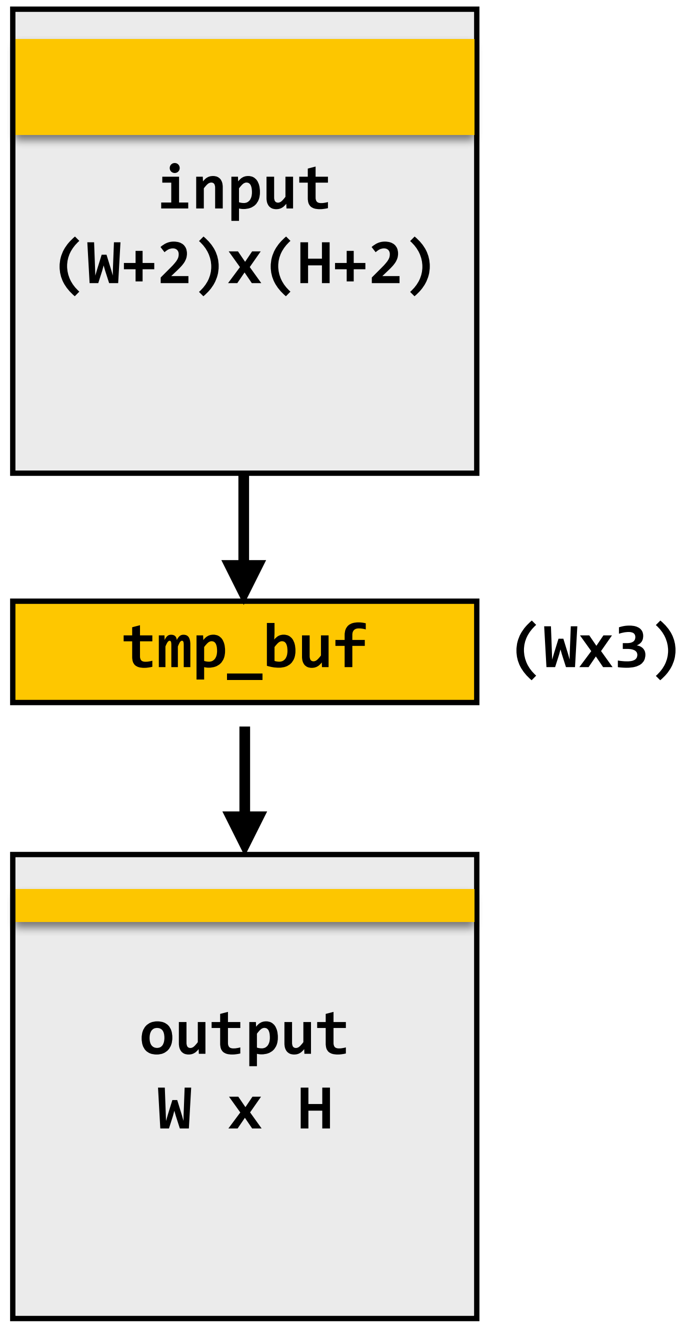
Two pass: loads/stores to `tmp_buf` are overhead (this memory traffic is an artifact of the two-pass implementation: it is not intrinsic to computation being performed)

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



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**

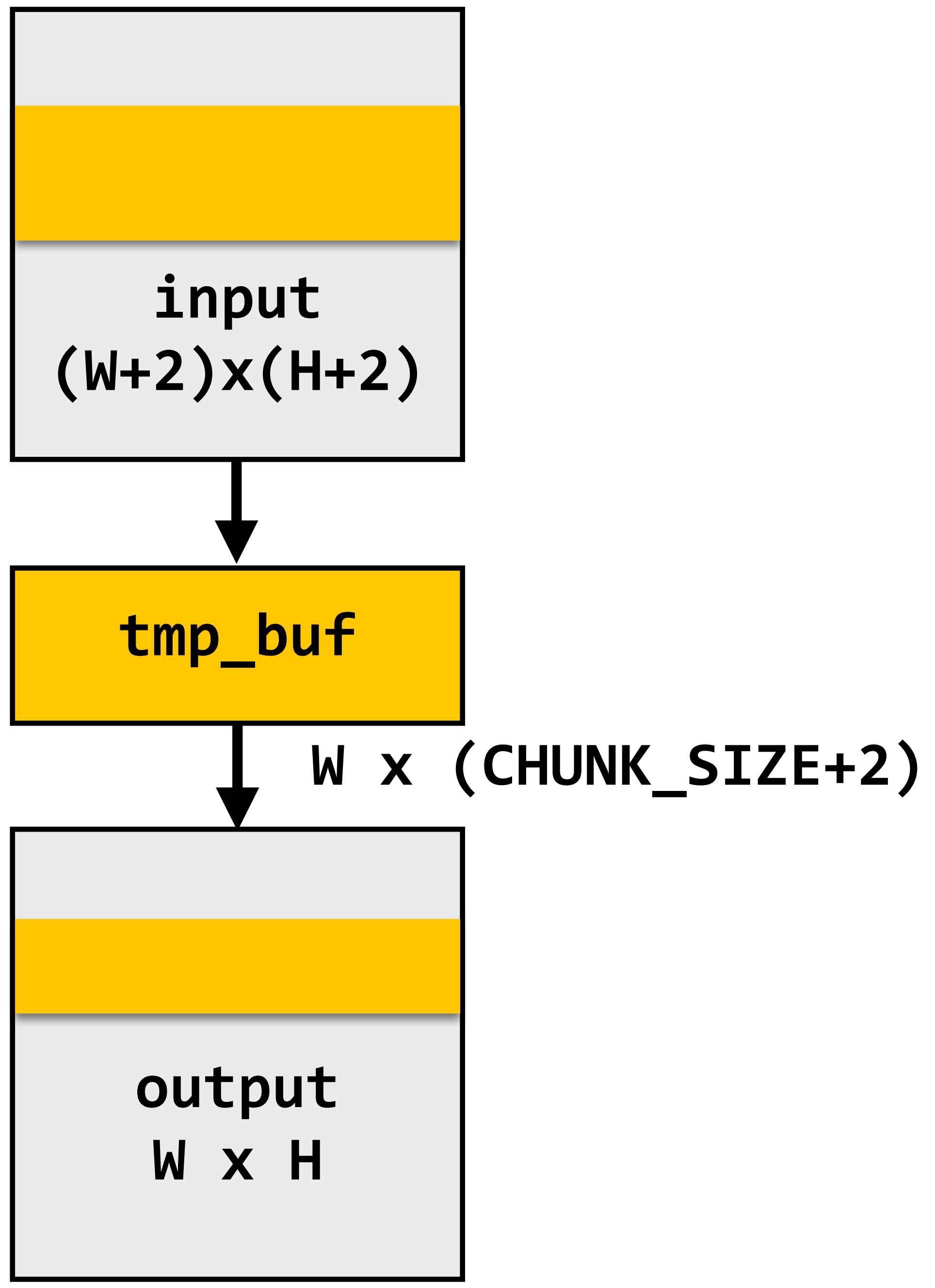
Total work per image = 12 x WIDTH x HEIGHT ????

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 / Adams 2012]

Simple domain-specific language embedded in C++ for describing sequences of image processing operations

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

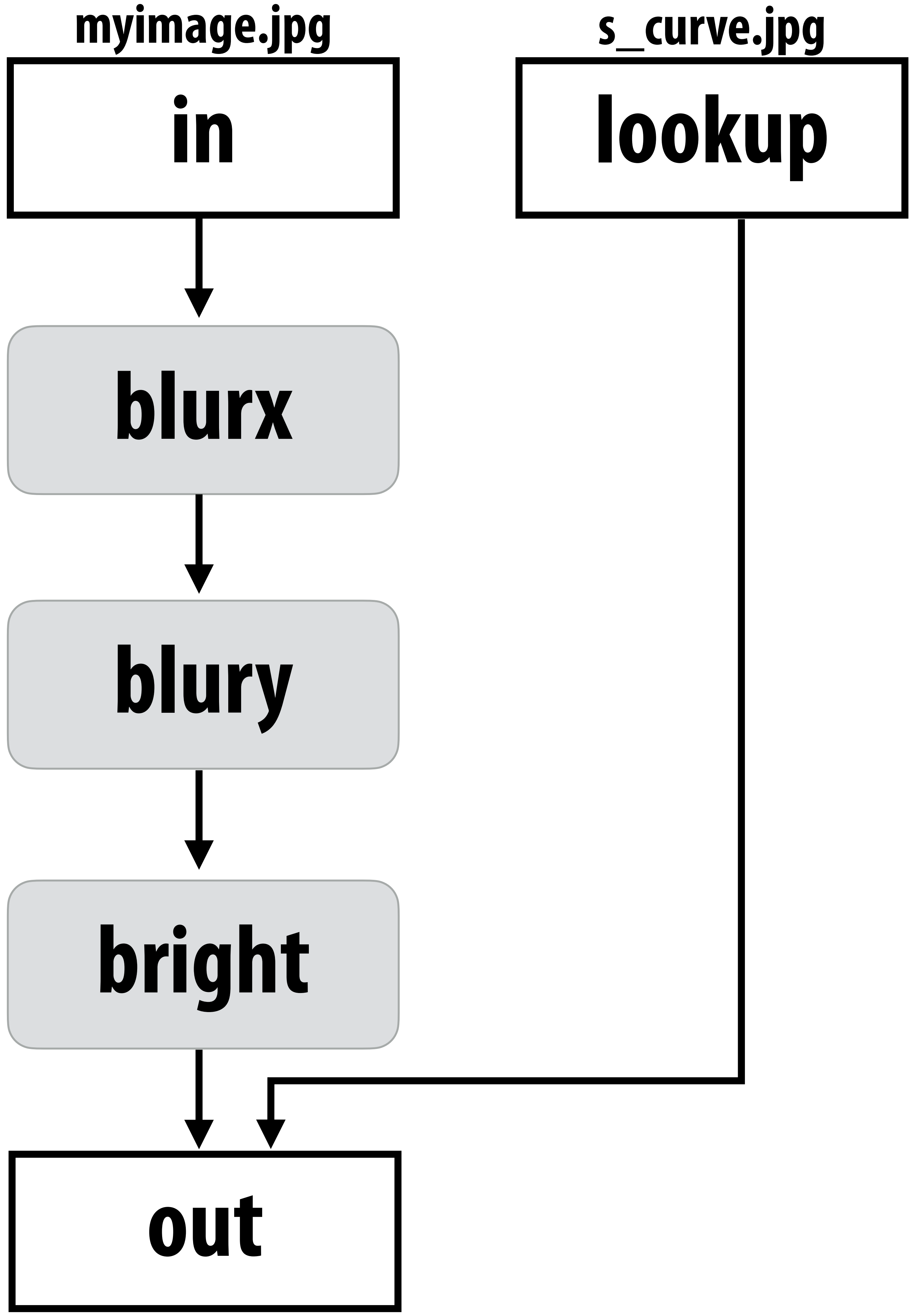
```
Var x, y;
Func blurx, blury, bright, out;
Halide::Buffer<uint8_t> in = load_image("myimage.jpg");
Halide::Buffer<uint8_t> lookup = load_image("s_curve.jpg");  // 255-pixel 1D image

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

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

// access lookup table to contrast enhance
out(x,y) = lookup(bright(x,y));

// execute pipeline to materialize values of out in range (0:1024,0:1024)
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

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 defined on N-D domain

Halide expression: a side-effect free expression that describes how to compute a function's value at a point in its domain in terms of the values of other functions.

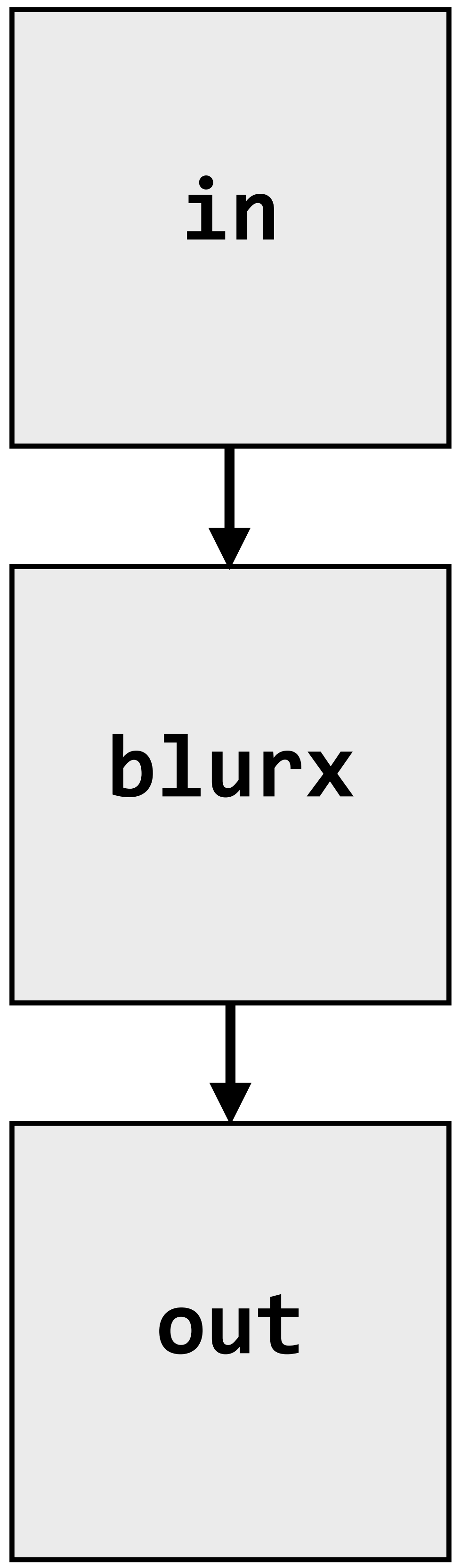
Image processing application as a DAG



Key aspects of representation

- Intuitive expression:
  - Adopts local "point wise" view of expressing algorithms
  - Halide language is declarative. It does not define order of iteration, or what values in domain are stored!
    - It only defines what is needed to compute these values.
    - Iteration over domain points is implicit (no explicit loops)

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

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

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



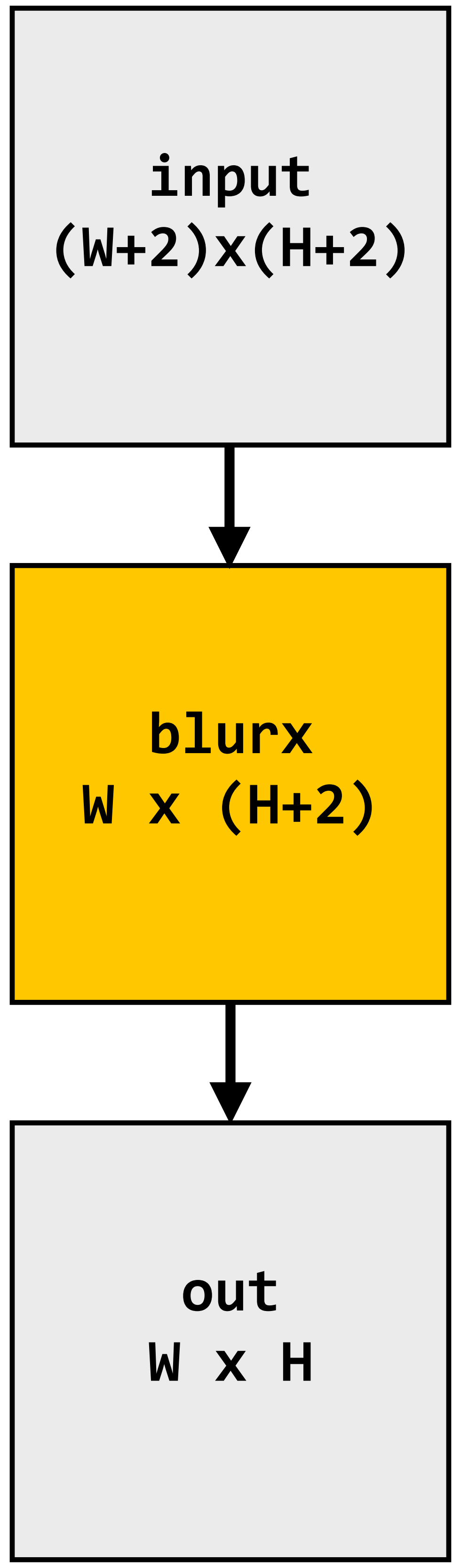
Real-world image processing pipelines feature complex sequences of functions

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.

One (serial) implementation of Halide

```
Func blurx, out;
Var  x, y, xi, yi;
Halide::Buffer<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;

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



Equivalent "C-style" loop nest:

```
allocate in(1024+2, 1024+2);    // (width,height)… initialize from image
allocate blurx(1024,1024+2);    // (width,height)
allocate out(1024,1024);        // (width,height)

for y=0 to 1024:
   for x=0 to 1024+2:
      blurx(x,y) = … compute from in

for y=0 to 1024:
   for x=0 to 1024:
      out(x,y) = … compute from blurx
```

Key aspect in the design of any system:

Choosing the “right” representations for the job

- **Good representations are productive to use:**
  - Embody the natural way of thinking about a problem
- **Good representations enable the system to provide the application useful services:**
  - Validating/providing certain guarantees (correctness, resource bounds, conversation of quantities, type checking)
  - Performance (parallelization, vectorization, use of specialized hardware)

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;
Halide::Buffer<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.

Vectorize the xi loop (8-wide)

Use threads to parallelize the y loop

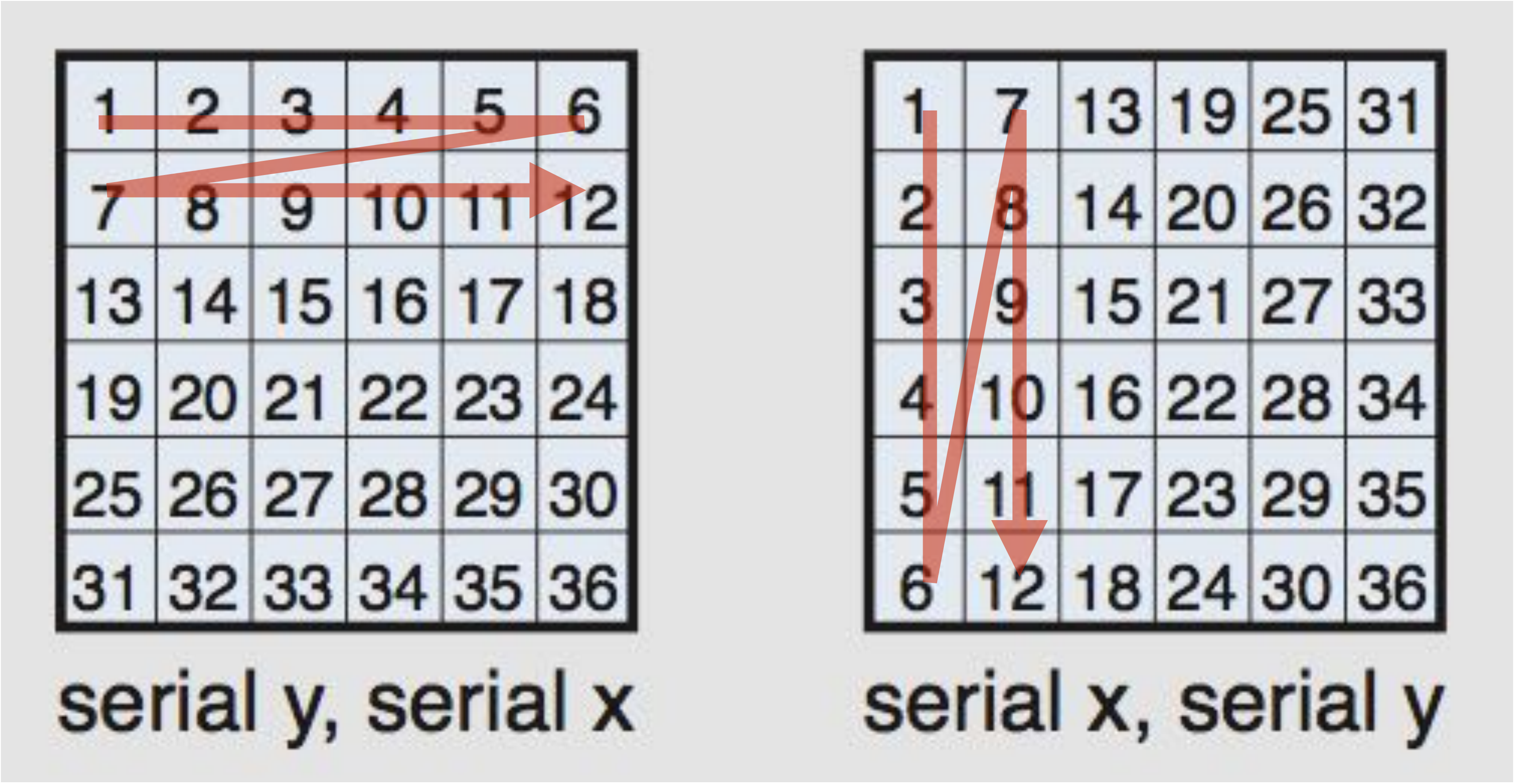
Produce elements `blurx` on demand for each tile of output. Vectorize the x (innermost) loop

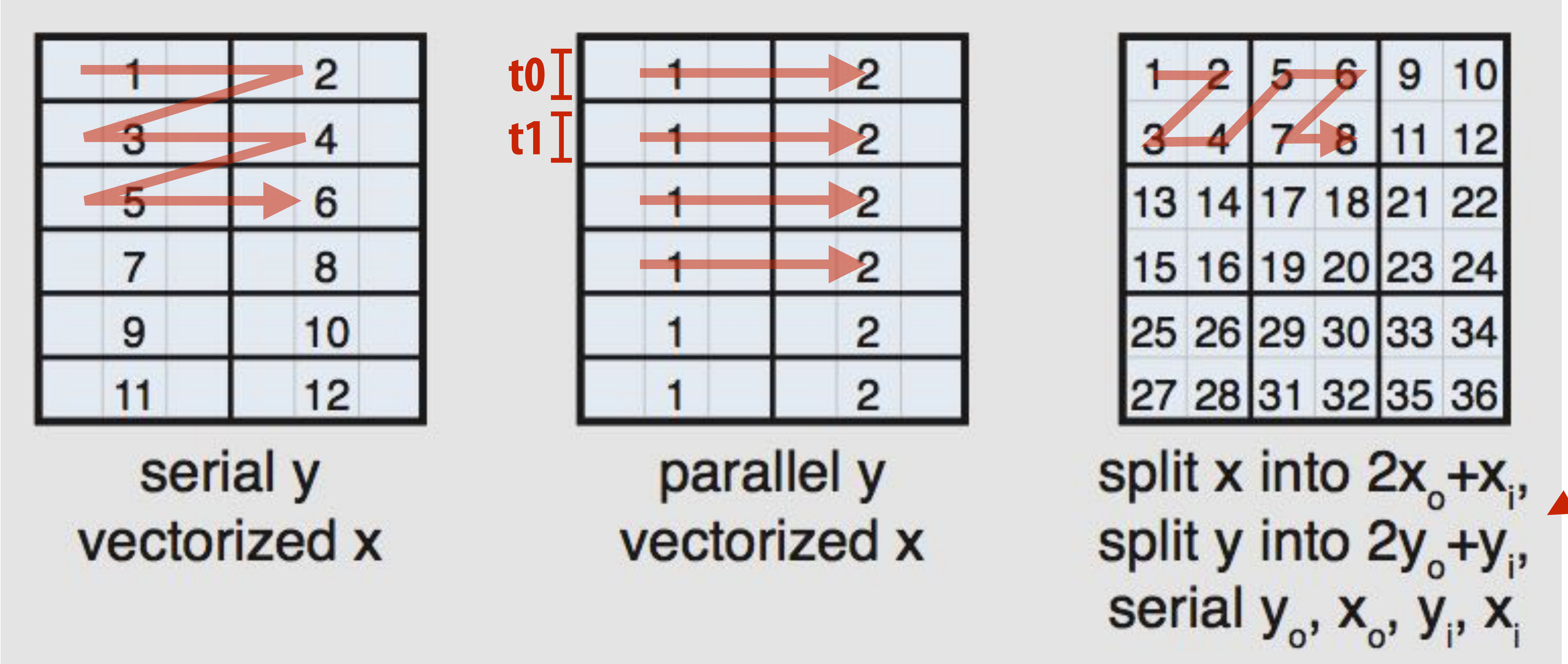
“Schedule”

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

Scheduling primitives allow the programmer to specify a high-level “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 N-D domains





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

2D blocked iteration order

(In diagram, numbers indicate sequential order of processing within a thread)

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 HEIGHT
  for x=0 to WIDTH
      blurx(x,y) = ...

for y=0 to num_tiles_y:            // parallelize this loop with threads
   for x=0 to num_tiles_x:
      for yi=0 to 32:
         for xi=0 to 256 by 8:    // vectorize this loop with SIMD instr
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_x, idx_y) = ... (simd arithmetic 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_root();
```

Do not compute blurx within out's loop nest.
Compute all of blurx, then all of out

```
allocate buffer for all of blurx(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_x, 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(out, xi);
```

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 tmp_blurx

            // compute 3 elements of blurx needed for out(idx_x, idx_y) here
            for (blur_x=0 to 3)
                tmp_blurx(blur_x) = …

            out(idx_x, idx_y) = …
```

Note: Halide compiler performs analysis that the output of each iteration of the xi loop required 3 elements of blurx

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(out, 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:
            tmp_blurx(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_x, idx_y) = …
```

tile of blurx is computed here

tile of blurx is consumed here

Summary of scheduling the 3x3 box blur

```
// 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(out, x).vectorize(x, 8);
```

Equivalent parallel loop nest:

```
for y=0 to num_tiles_y:   // iters of this loop are parallelized using threads
   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 BY 8:
            tmp_blurx(xi,yi) = … // compute blurx from in using 8-wide
                                 // SIMD instructions here
                                 // compiler generates boundary conditions
                                 // since 256+2 isn't evenly divided by 8
      for yi=0 to 32:
         for xi=0 to 256 BY 8:
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_x, idx_y) = … // compute out from blurx using 8-wide
                                  // SIMD instructions here
```

What is the philosophy of Halide

- Programmer is responsible for describing an image processing algorithm
- Programmer has knowledge of how to schedule the application efficiently on machine (but it's slow and tedious), so Halide gives programmer a language to express high-level scheduling decisions
  - Loop structure of code
  - Unrolling / vectorization / multi-core parallelization
- The system (Halide compiler) is not smart, it provides the service of mechanically carrying out the details of the schedule in terms of mechanisms available on the target machine (phthreads, AVX intrinsics, etc.)

Constraints on language

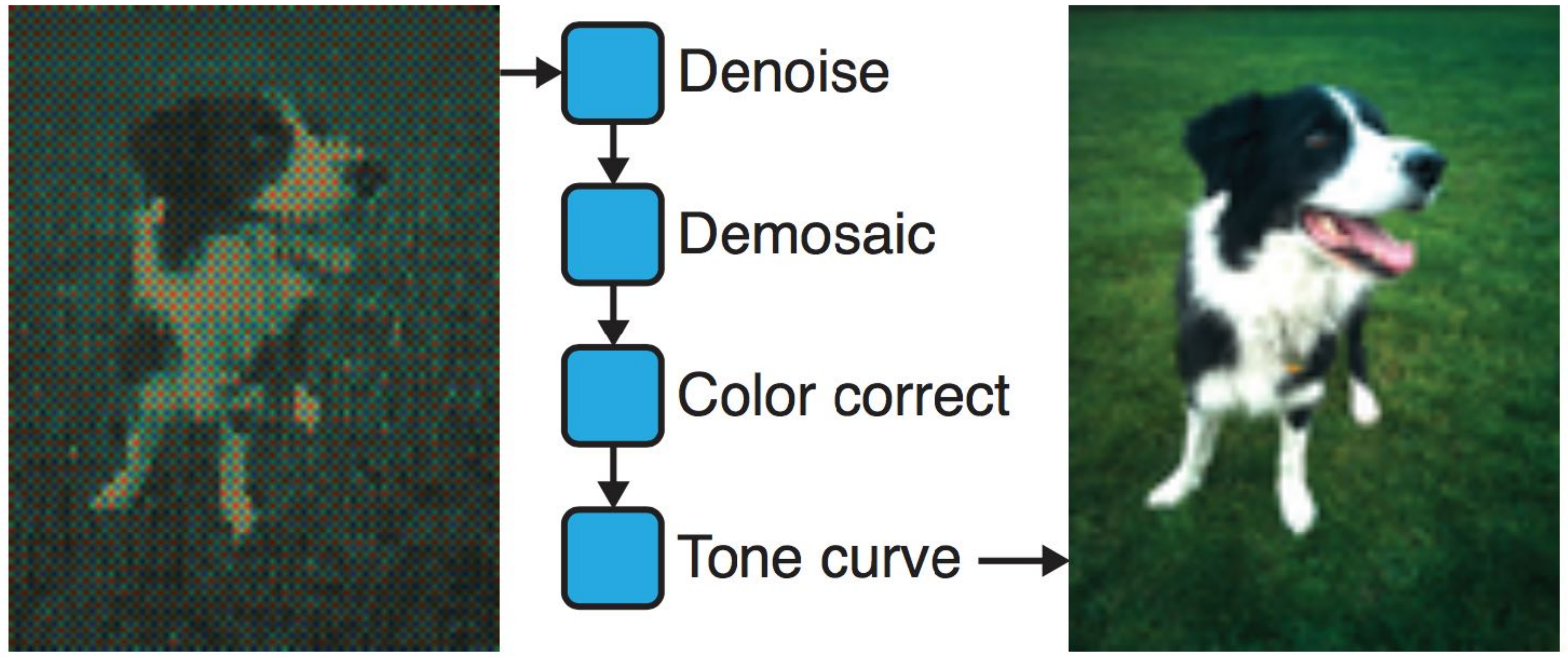
(to enable compiler to provide desired services)

- Application domain scope: computation on regular N-D domains
- Only feed-forward pipelines (includes special support for reductions and fixed recursion depth)
- All dependencies inferable by compiler

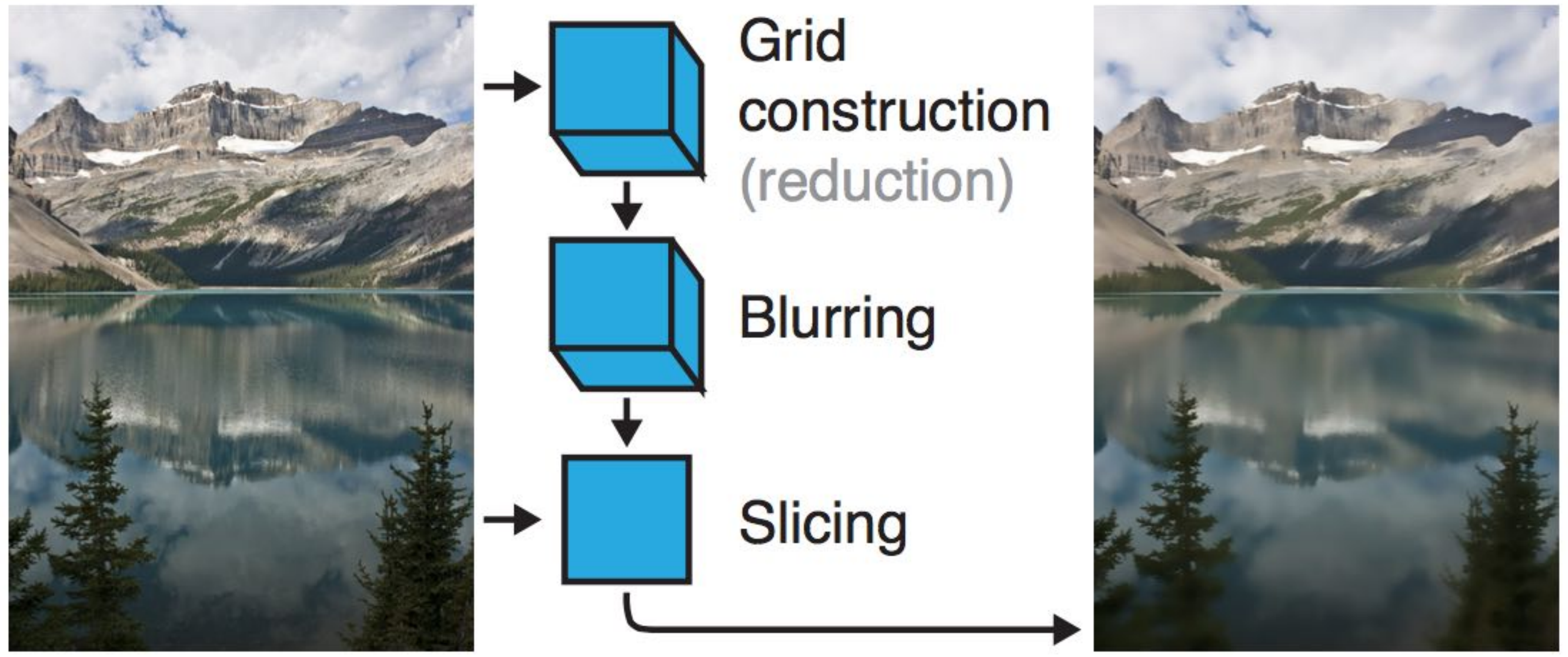
Initial academic Halide results

[Ragan-Kelley 2012]

- Application 1: 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



- Application 2: 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) 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 Halide schedules

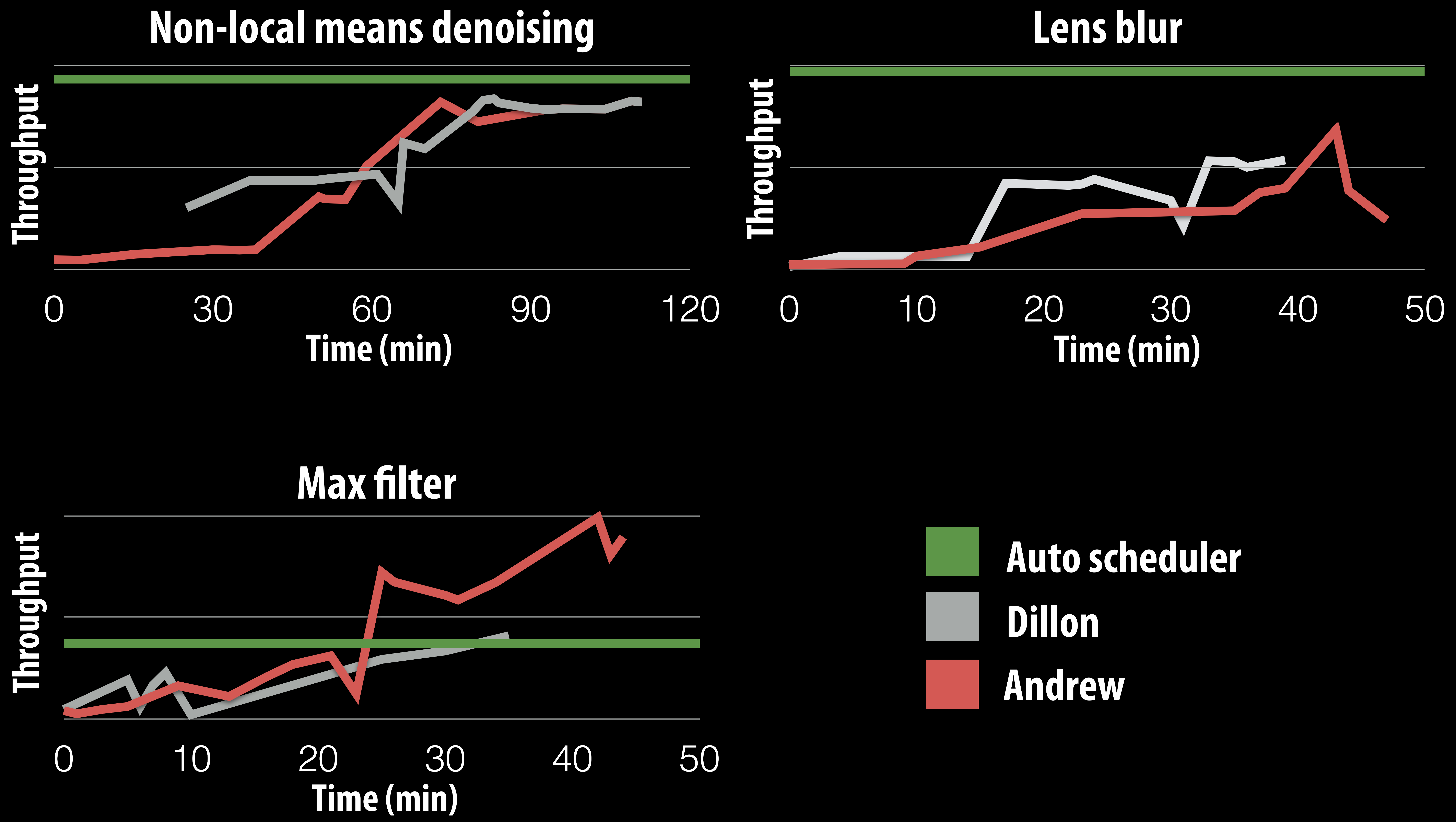
- Problem: it turned out that very few programmers have the ability to write good Halide schedules
  - 80+ programmers at Google write Halide
  - Very small number trusted to write schedules

- Recent work: compiler analyzes the Halide program to automatically generate efficient schedules for the programmer [Adams 2019]
  - As of [Adams 2019], you'd have to work pretty hard to manually author a schedule that is better than the schedule generated by the Halide autoscheduler for image processing applications

See *"Learning to Optimize Halide with Tree Search and Random Programs",* Adams et al. SIGGRAPH 2019

Autoscheduler saves time for experts

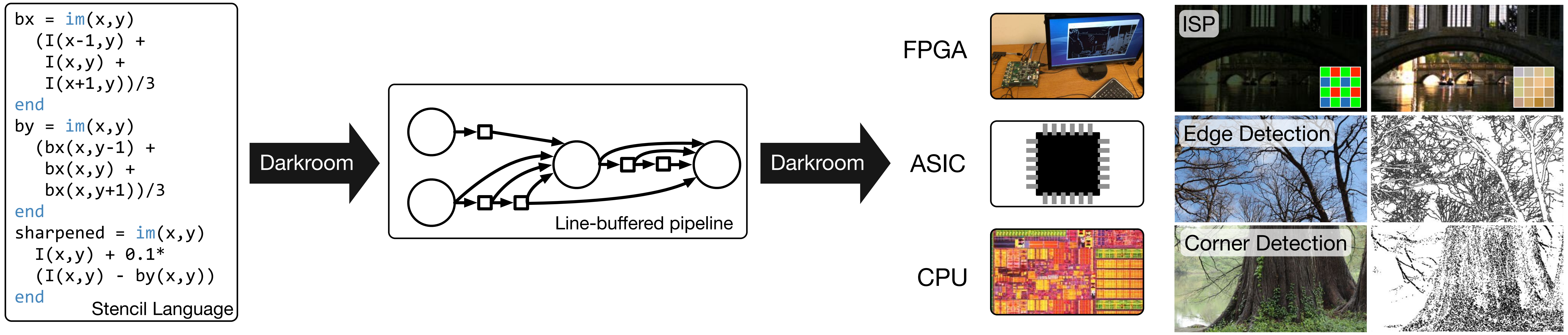
Early results from [Mullapudi 2016]



Darkroom/Rigel/Aetherling

[Hegarty 2014, Hegarty 2016, Durst 2020]

Goal: directly synthesize ASIC or FGPA implementation of image processing pipelines from a high-level algorithm description
(a constrained "Halide-like" language)



Goal: very-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 Ligra
(DSLs for describing operations on graphs)

Model-view-controller paradigm for
web-applications

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

Language for real-time 3D graphics

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]

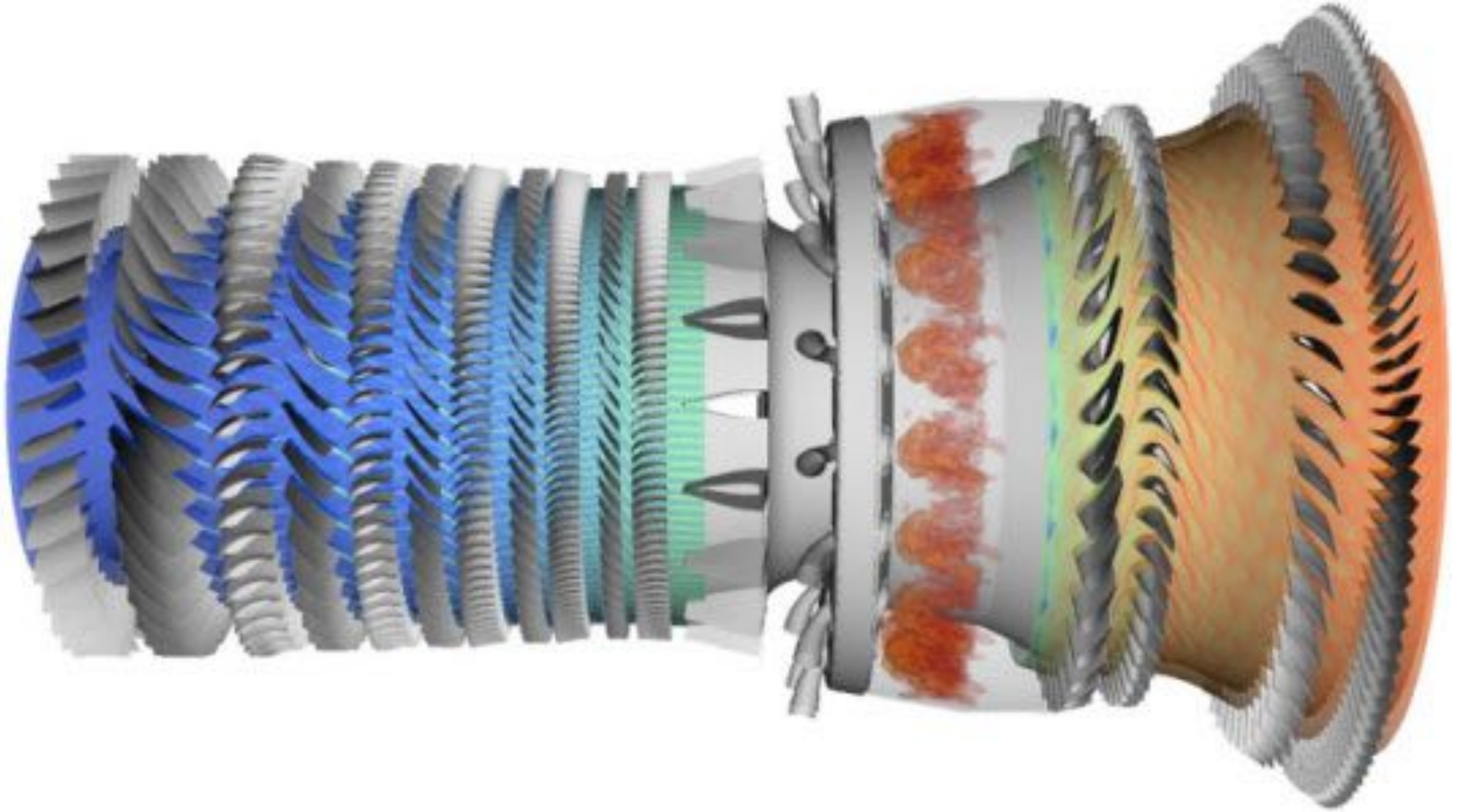
Summary

- **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 study today: Halide
  - Leverage explicit dependencies, domain restrictions, domain knowledge for system to synthesize efficient implementations

Another DSL example:

Lizst: a language for solving PDE's on meshes

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

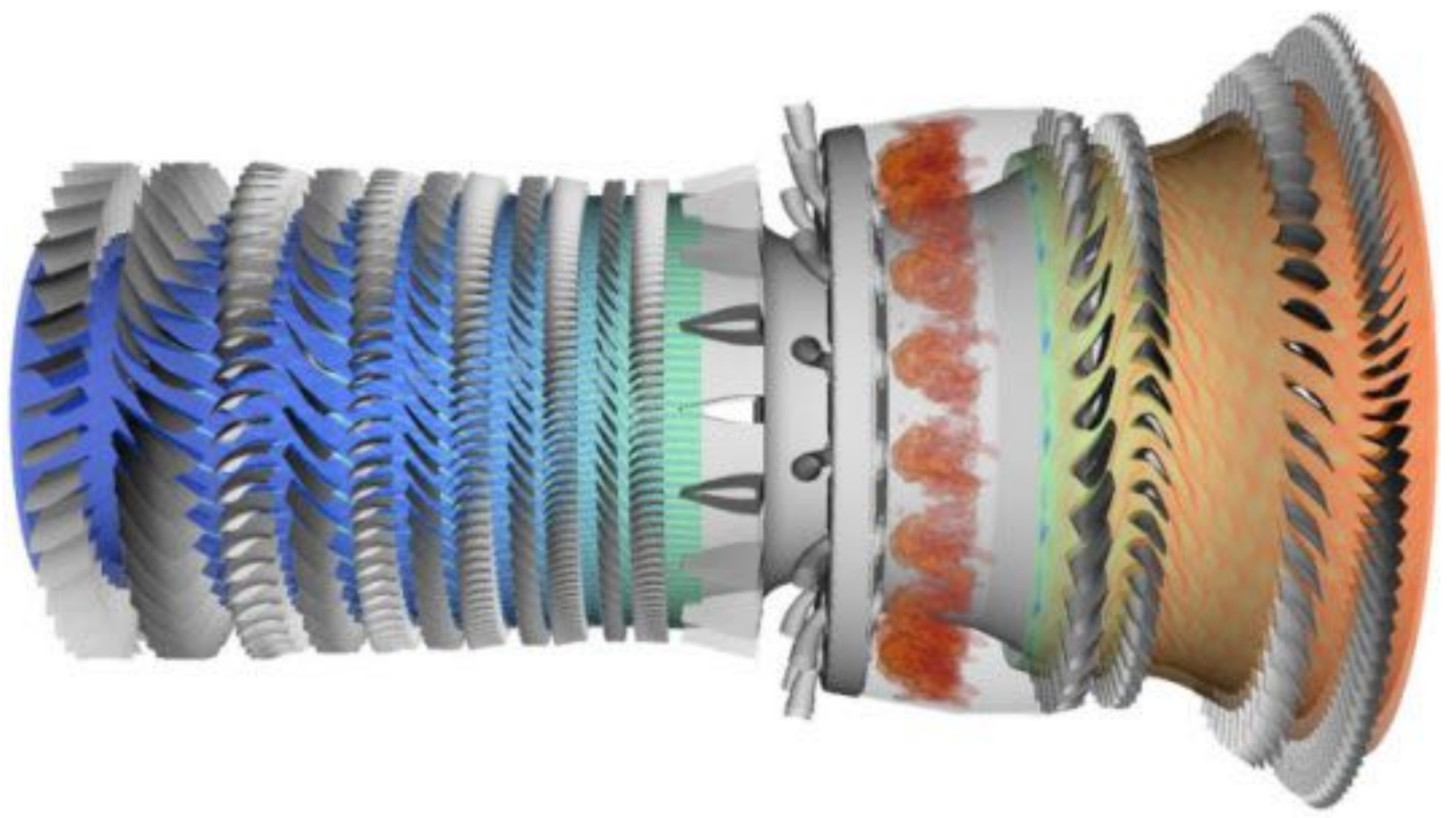


Slide credit for this section of lecture:
Pat Hanrahan and Zach Devito (Stanford)

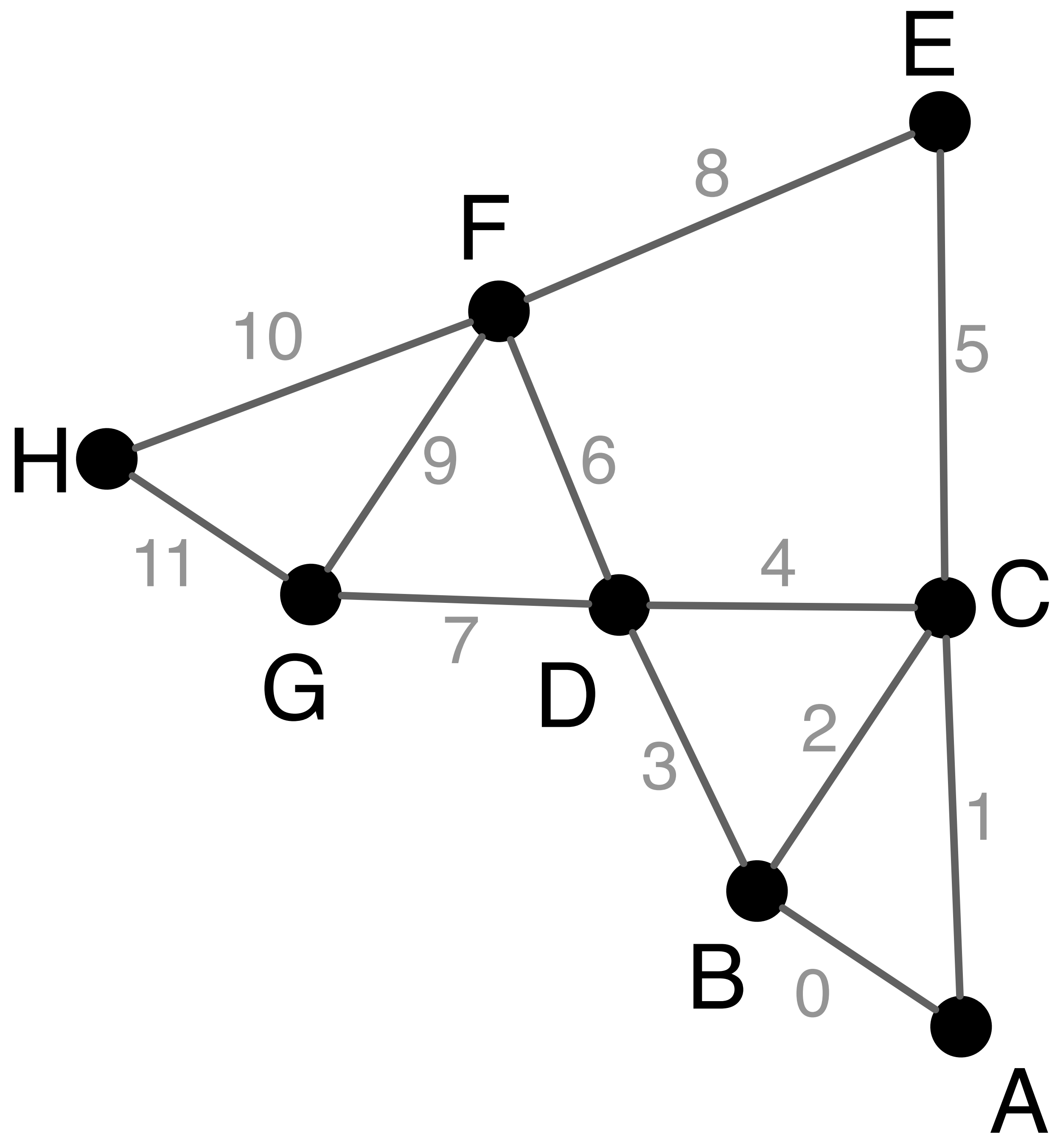
http://liszt.stanford.edu/

What a Liszt program does

A Liszt program is run on a mesh:



A Liszt program computes the value of fields defined on mesh faces, edges, or vertices



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;

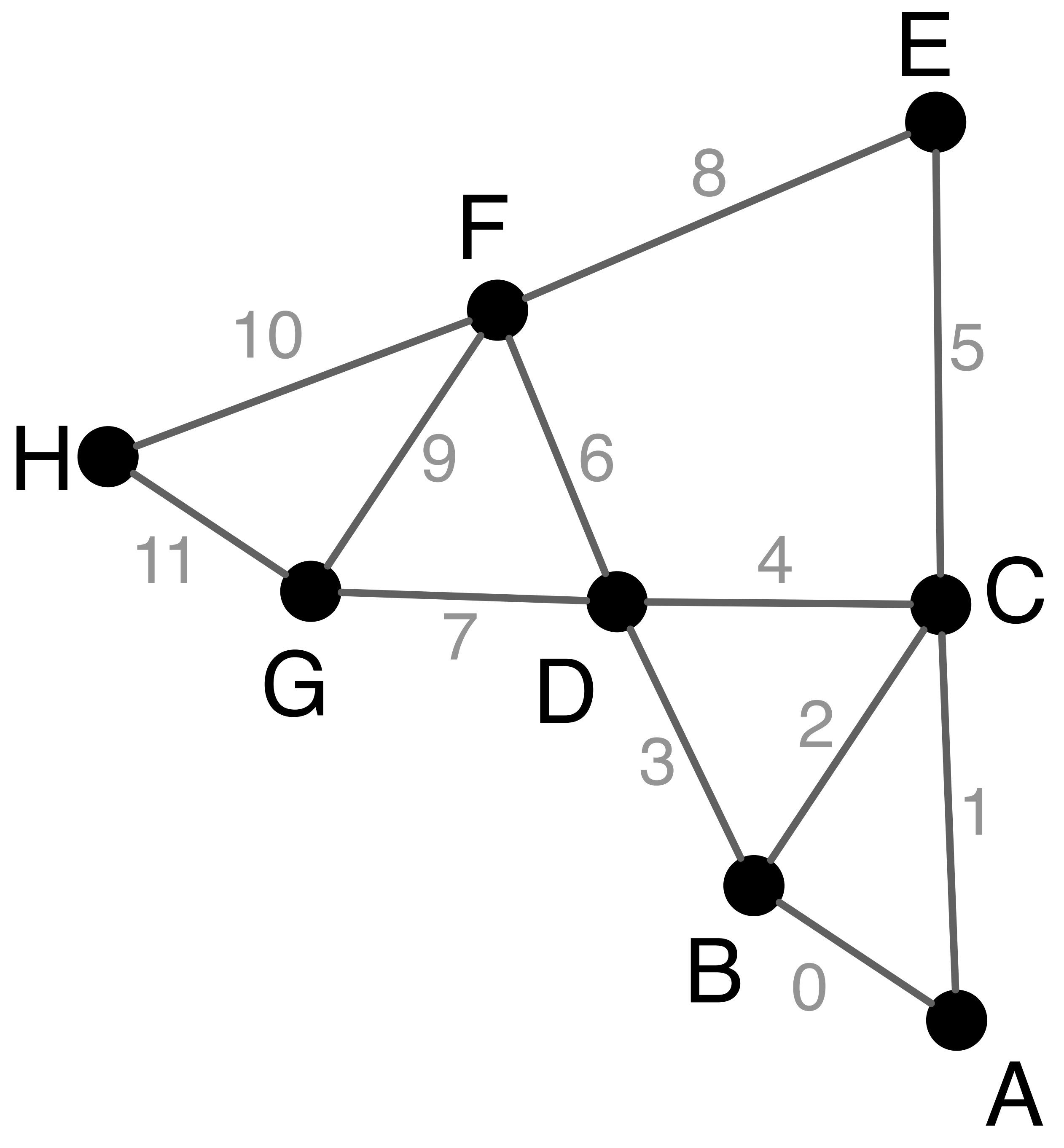
Independently, for each edge in the mesh

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 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

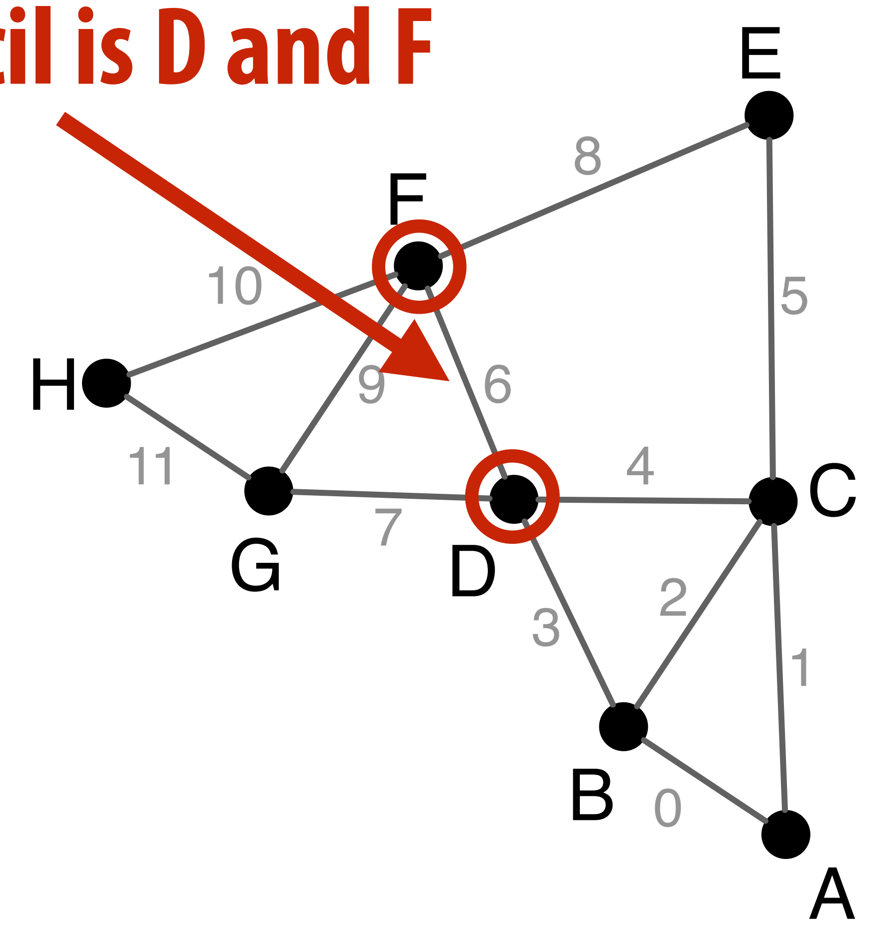
Lizst infers "stencils": "stencil" = mesh elements accessed in an iteration of loop
= dependencies for the iteration

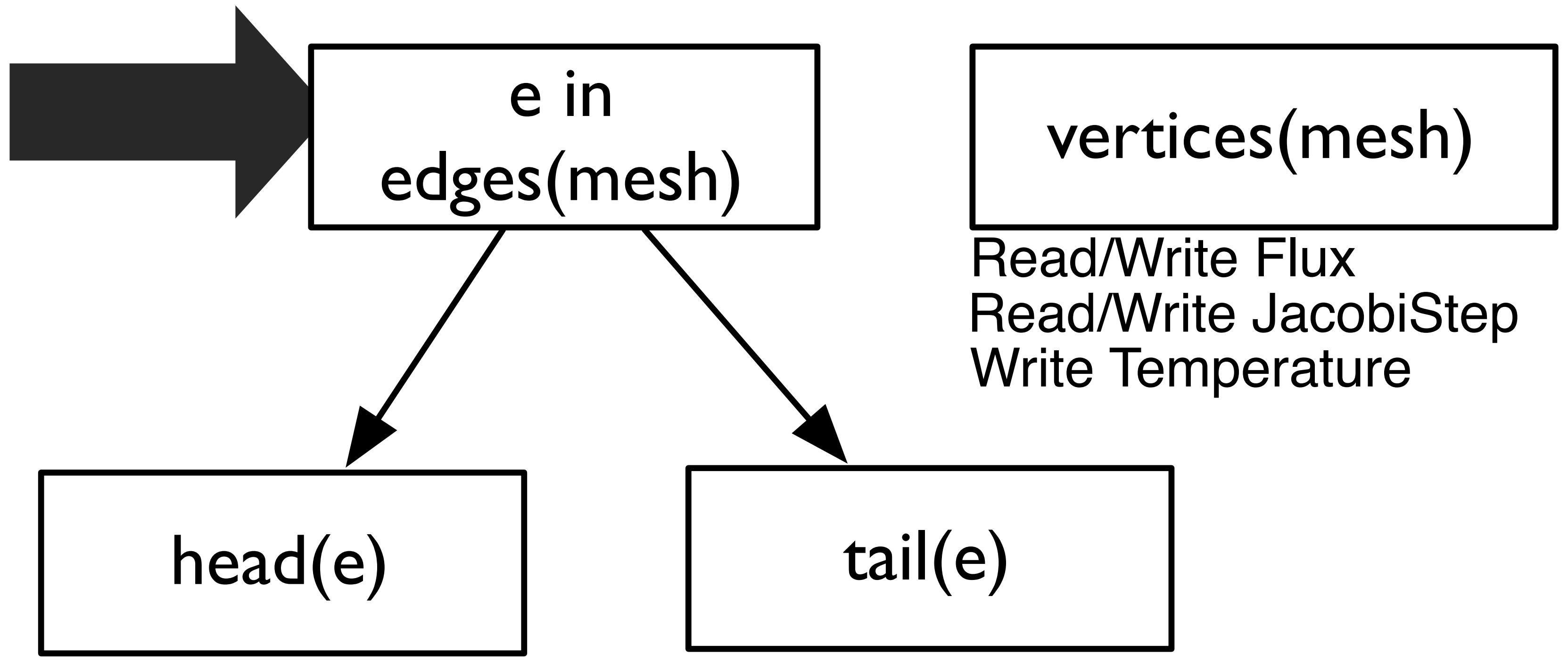
Statically analyze code to find stencil of each top-level for loop

- Extract nested mesh element reads
- Extract operations on data at mesh elements

```
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
}
...
```

Edge 6's read stencil is D and F



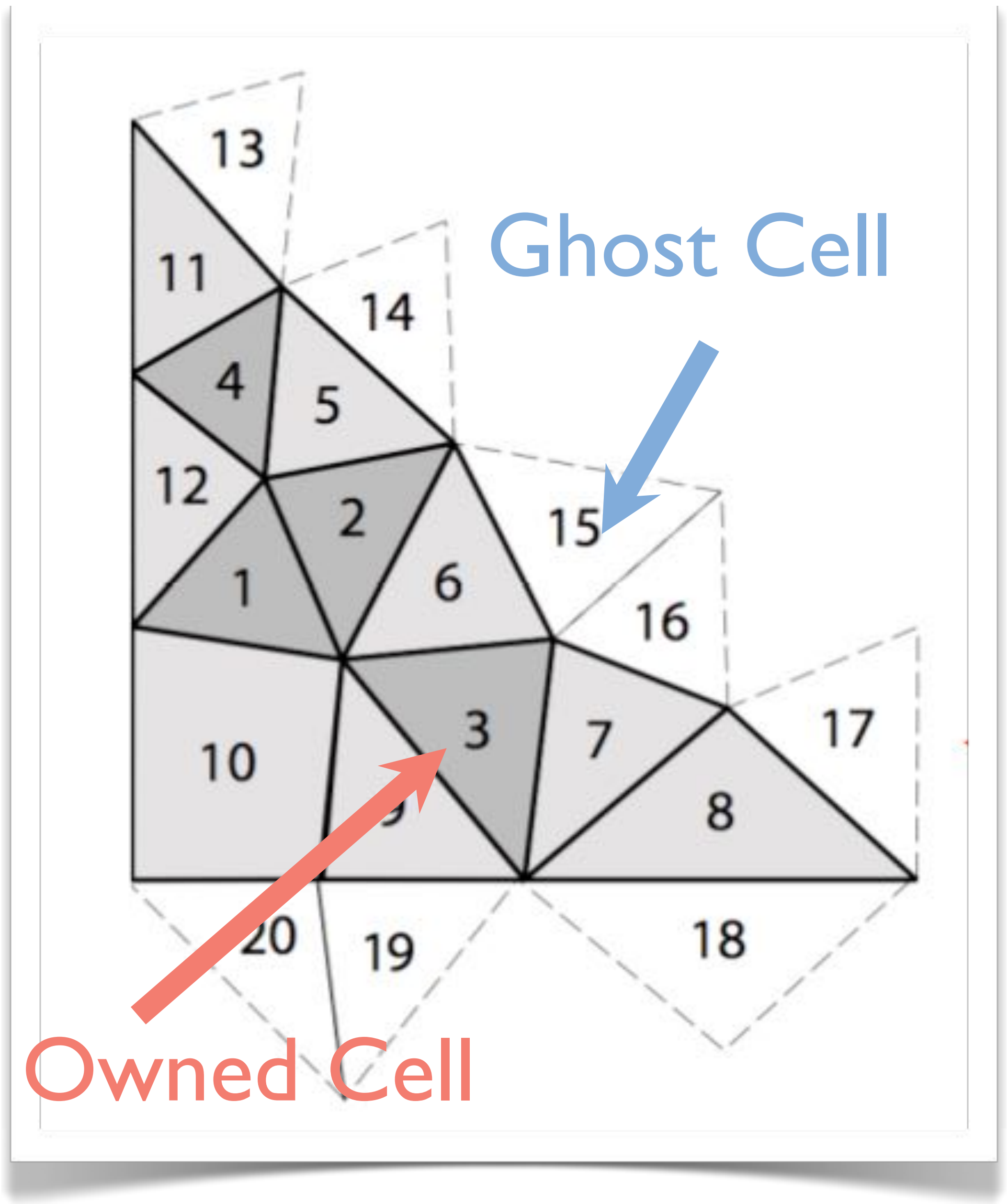


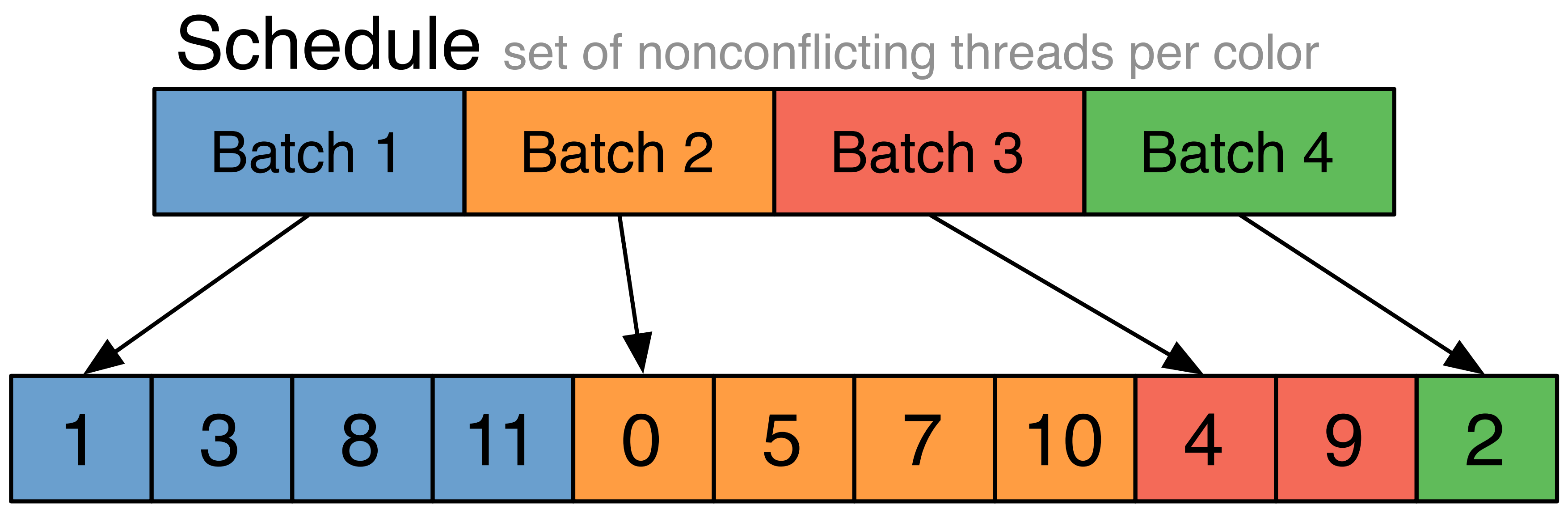
Portable parallelism: compiler uses knowledge of dependencies to implement different parallel execution strategies

I'll discuss two strategies…

Strategy 1: mesh partitioning

Strategy 2: mesh coloring





Imagine compiling a Lizst program to a cluster

(multiple nodes, distributed address space)

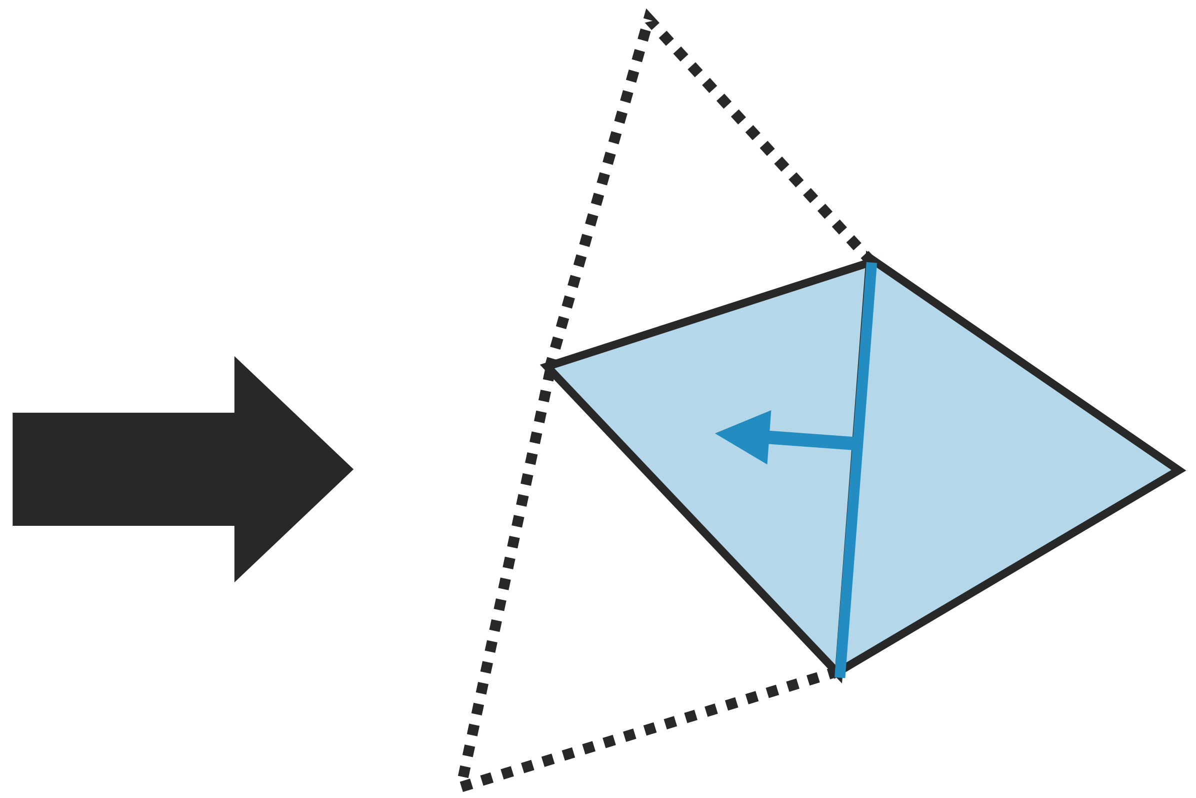
How might Liszt distribute a graph across these nodes?

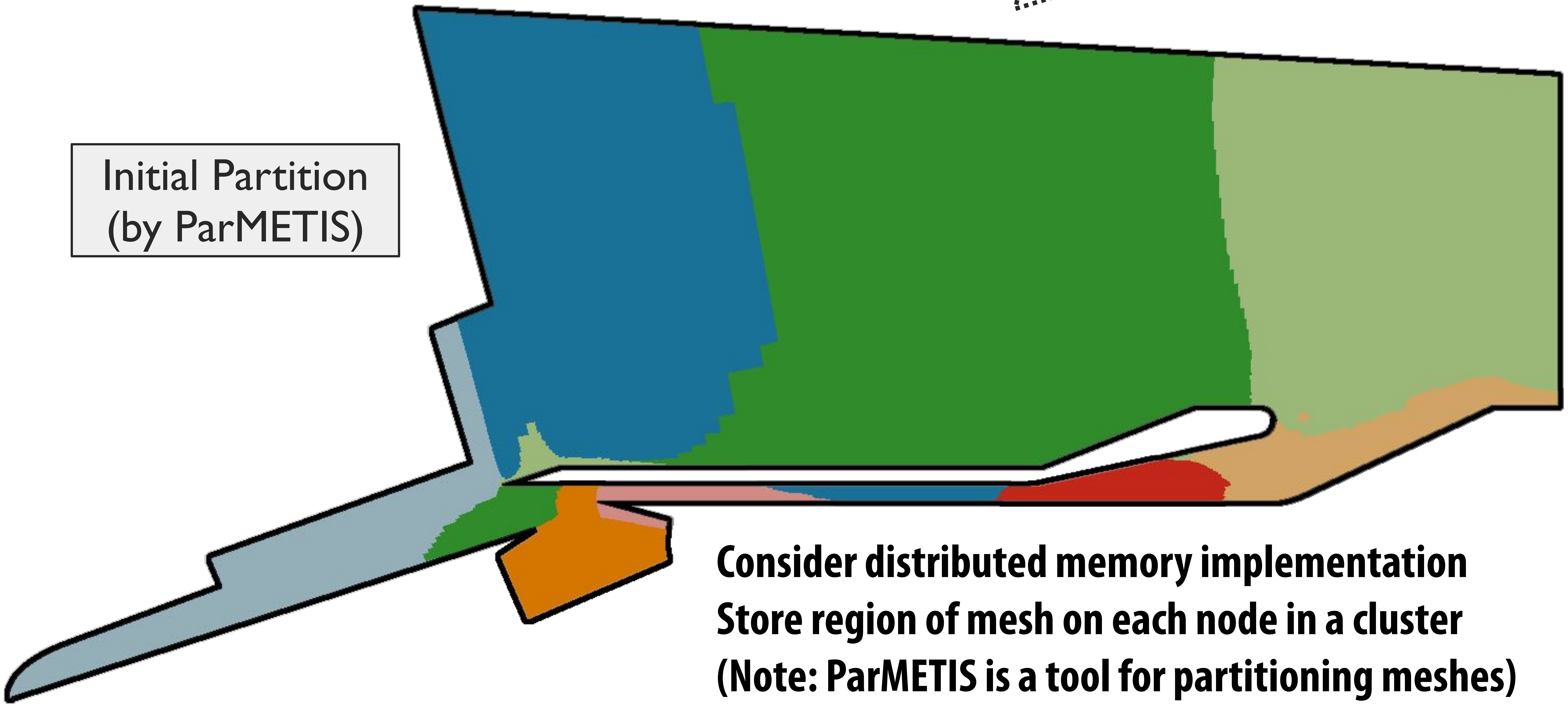
- **Must access mesh elements relative to some input vertex, edge, face, etc.)**
- **Notice how many operators return sets (e.g., “all edges of this face”)**

Distributed memory implementation of Liszt

Mesh + stencil → graph → partition

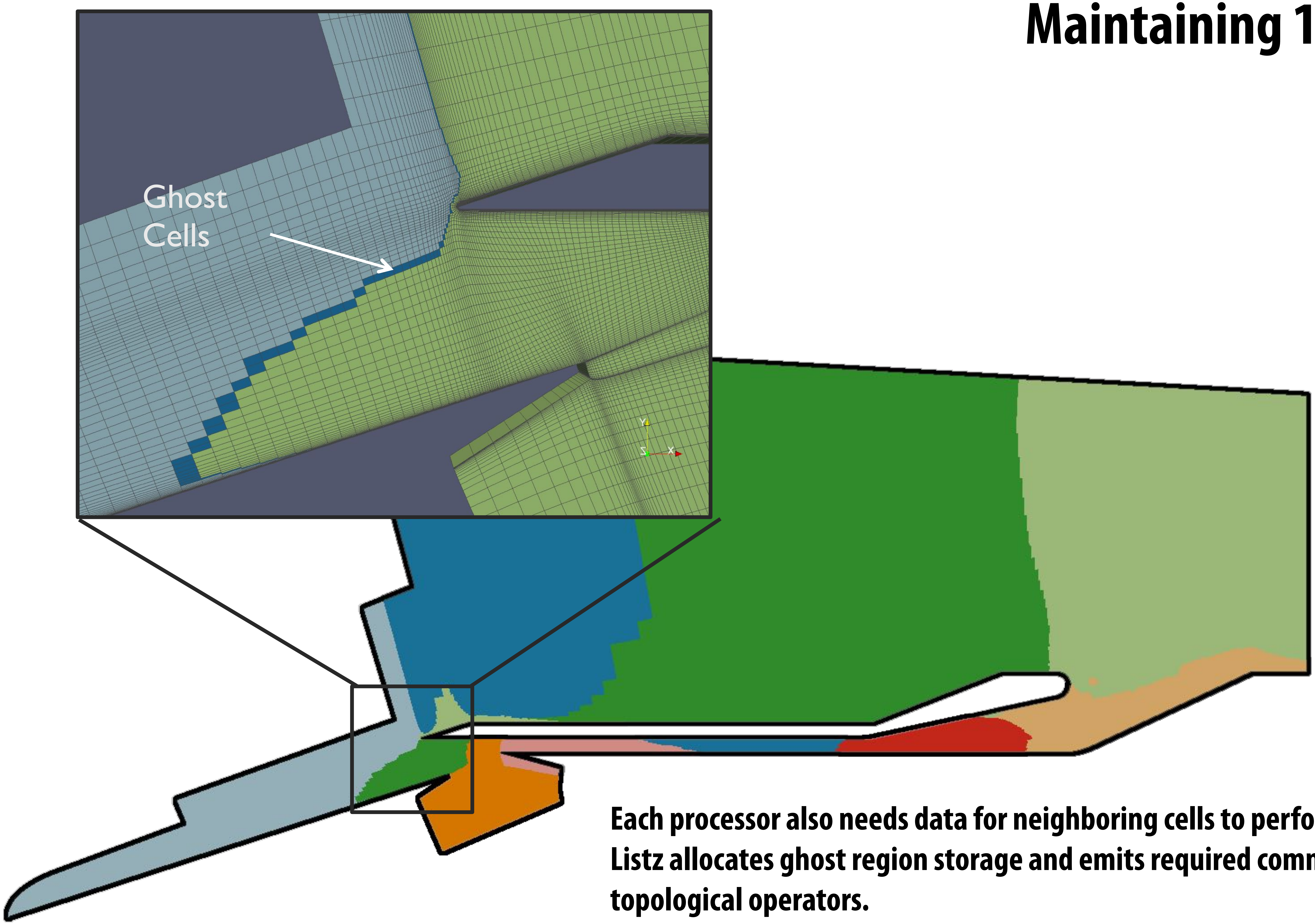
```
for(f <- faces(mesh)) {
  rhoOutside(f) =
    calc_flux(f, rho(outside(f))) +
    calc_flux(f, rho(inside(f)))
}
```





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

Maintaining 1-Level Ghost Cells



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

- Used to access mesh elements relative to some input vertex, edge, face, etc.)
- Notice how many operators return sets (e.g., "all edges of this face")

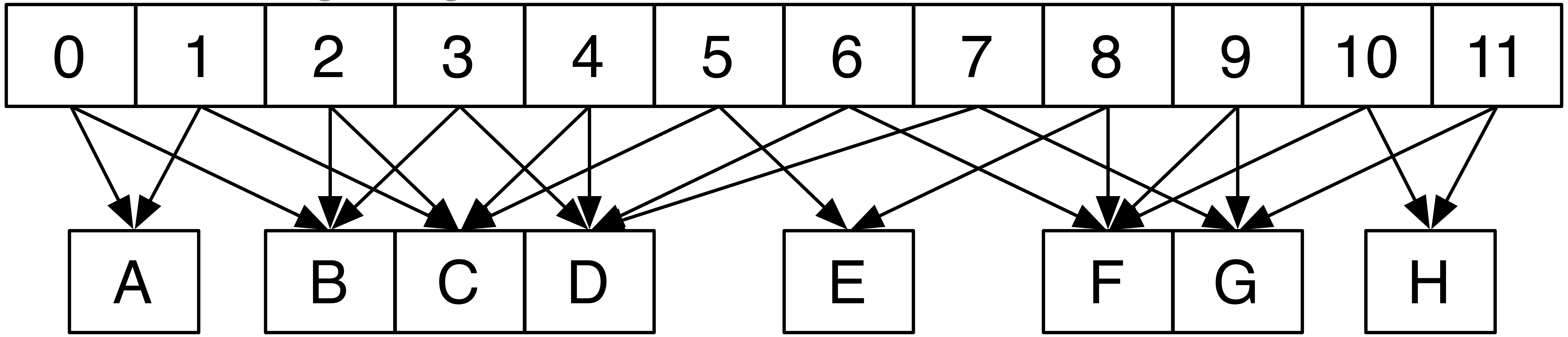
(single address space, many tiny threads)

GPU implementation: parallel reductions

In previous example, one region of mesh assigned per processor (or node in cluster)

On GPU, natural parallelization is one edge per CUDA thread

Edges (each edge assigned to 1 CUDA thread)



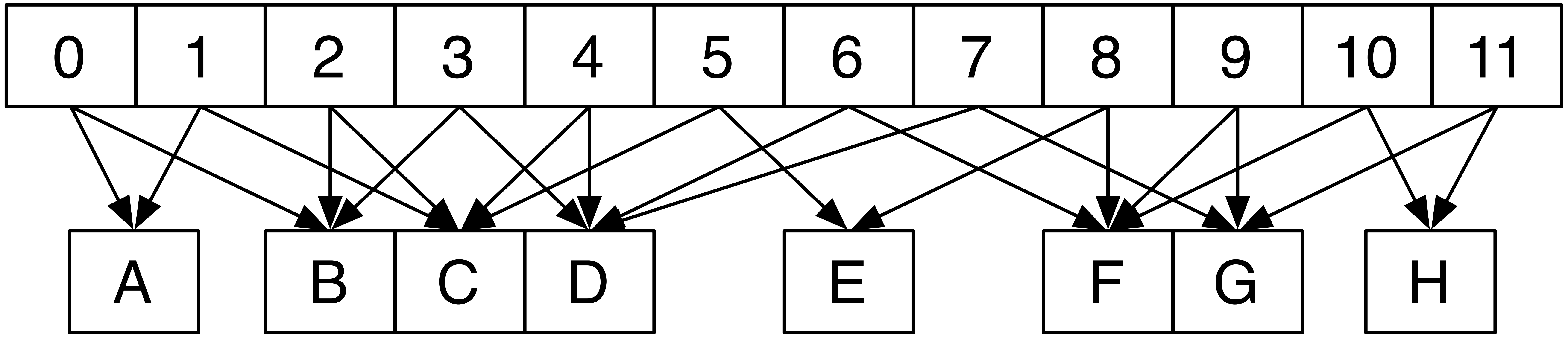
Flux field values (stored per vertex)

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

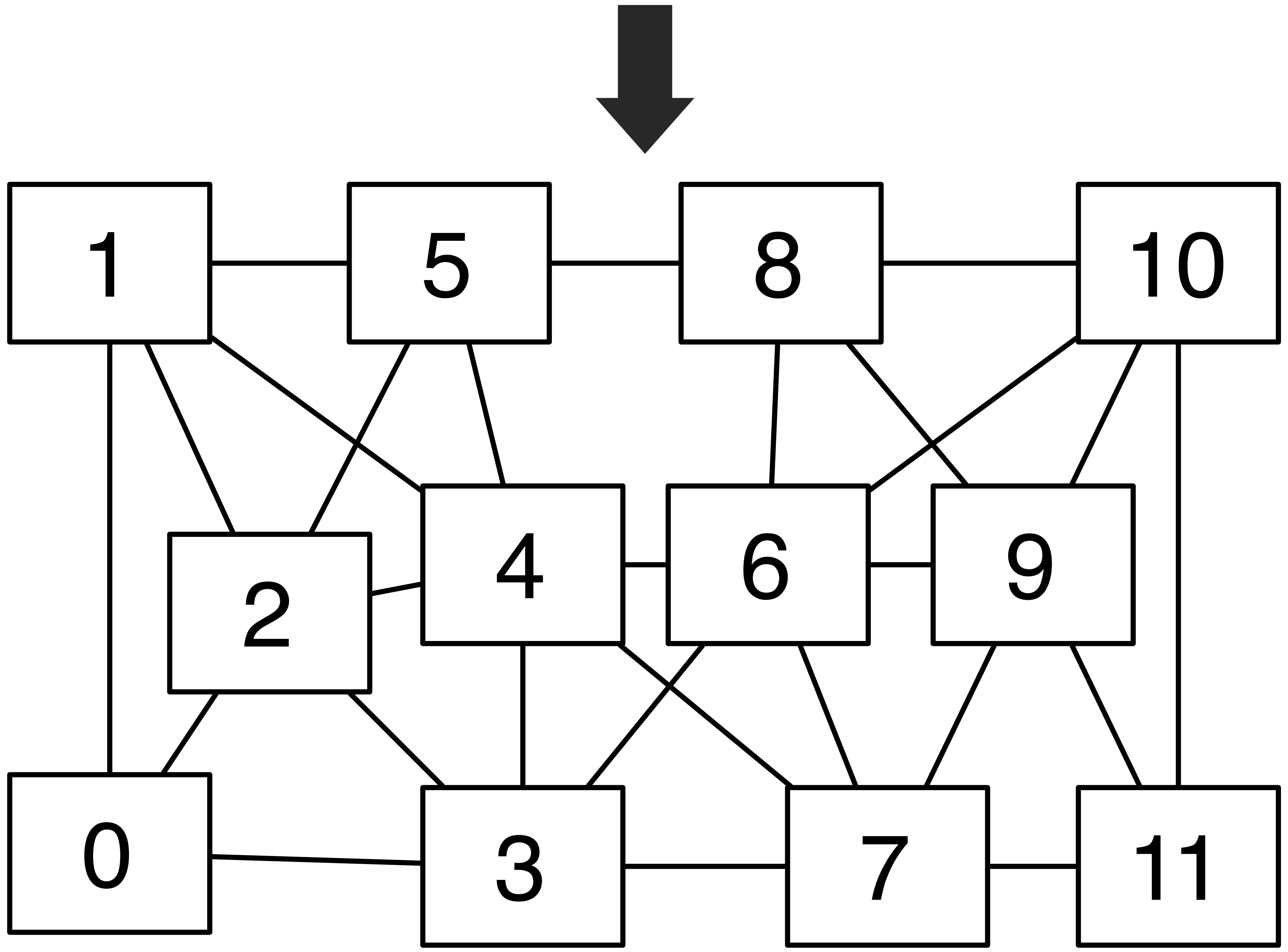
Different edges share a vertex: requires atomic update of per-vertex field data

GPU implementation: conflict graph

Edges (each edge assigned to 1 CUDA thread)



Flux field values (per vertex)

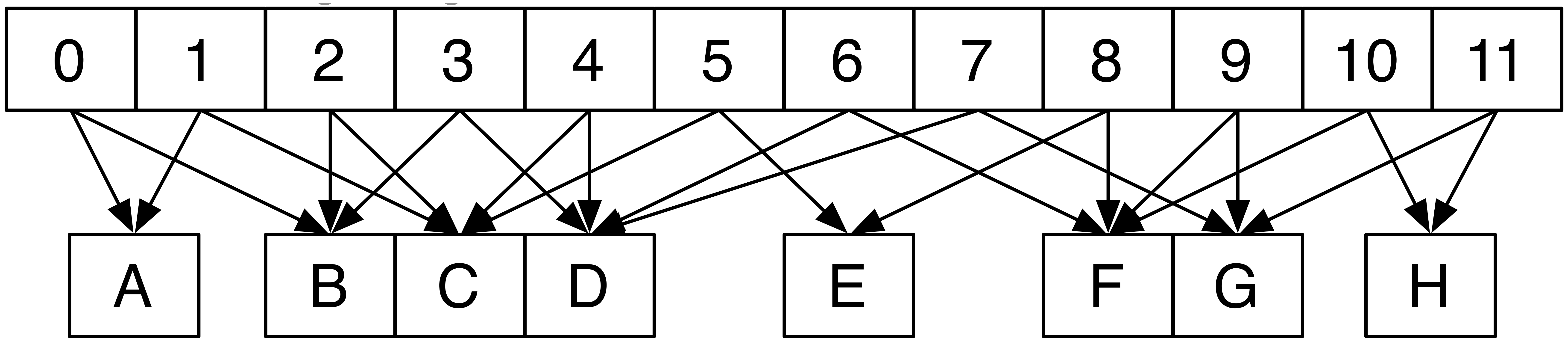


Identify mesh edges with colliding writes
(lines in graph indicate presence of collision)

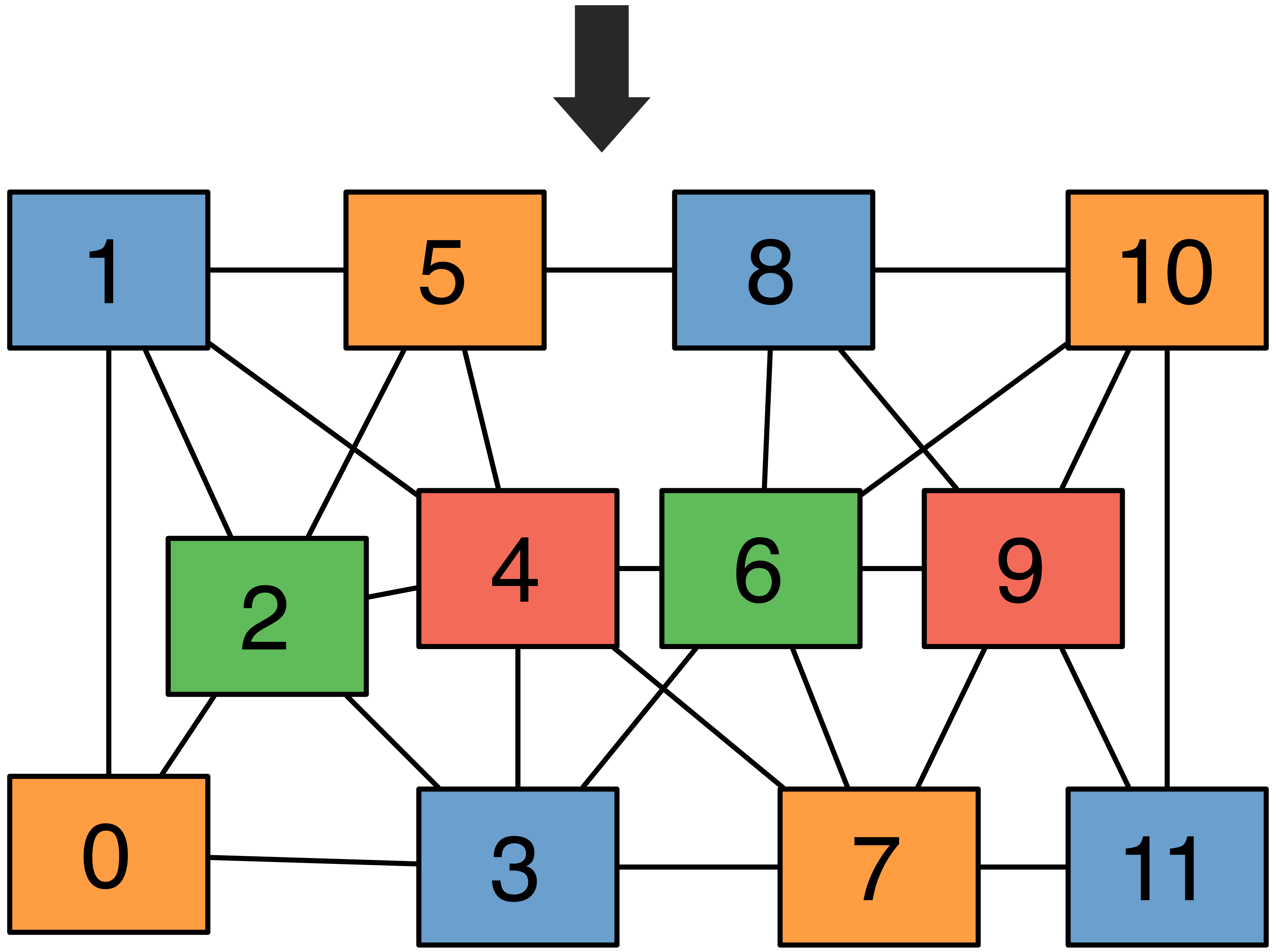
Can simply run program once to get this information.
(results remain valid for subsequent executions provided mesh does not change)

GPU implementation: conflict graph

Threads (each edge assigned to 1 CUDA thread)



Flux field values (per vertex)

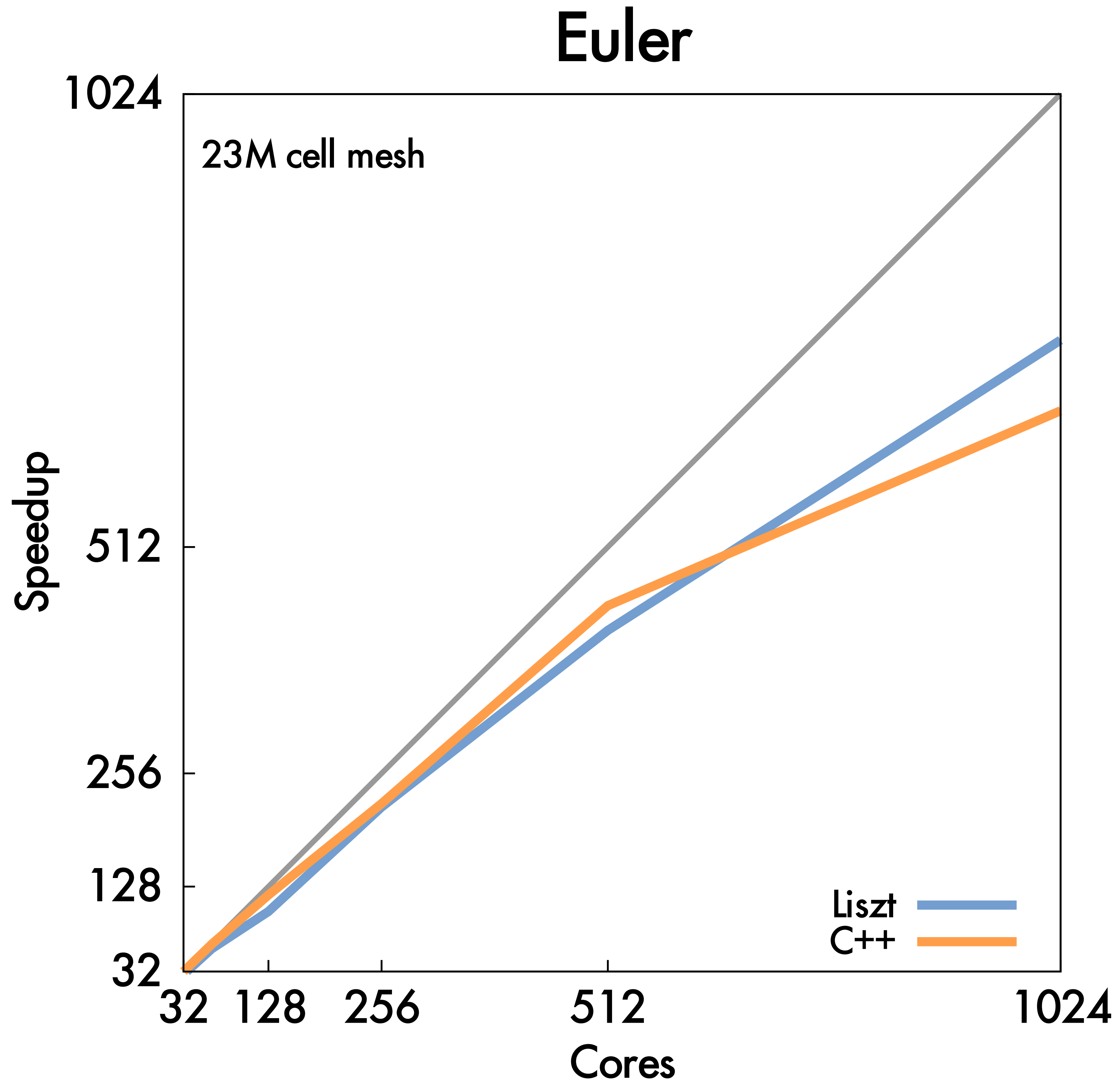


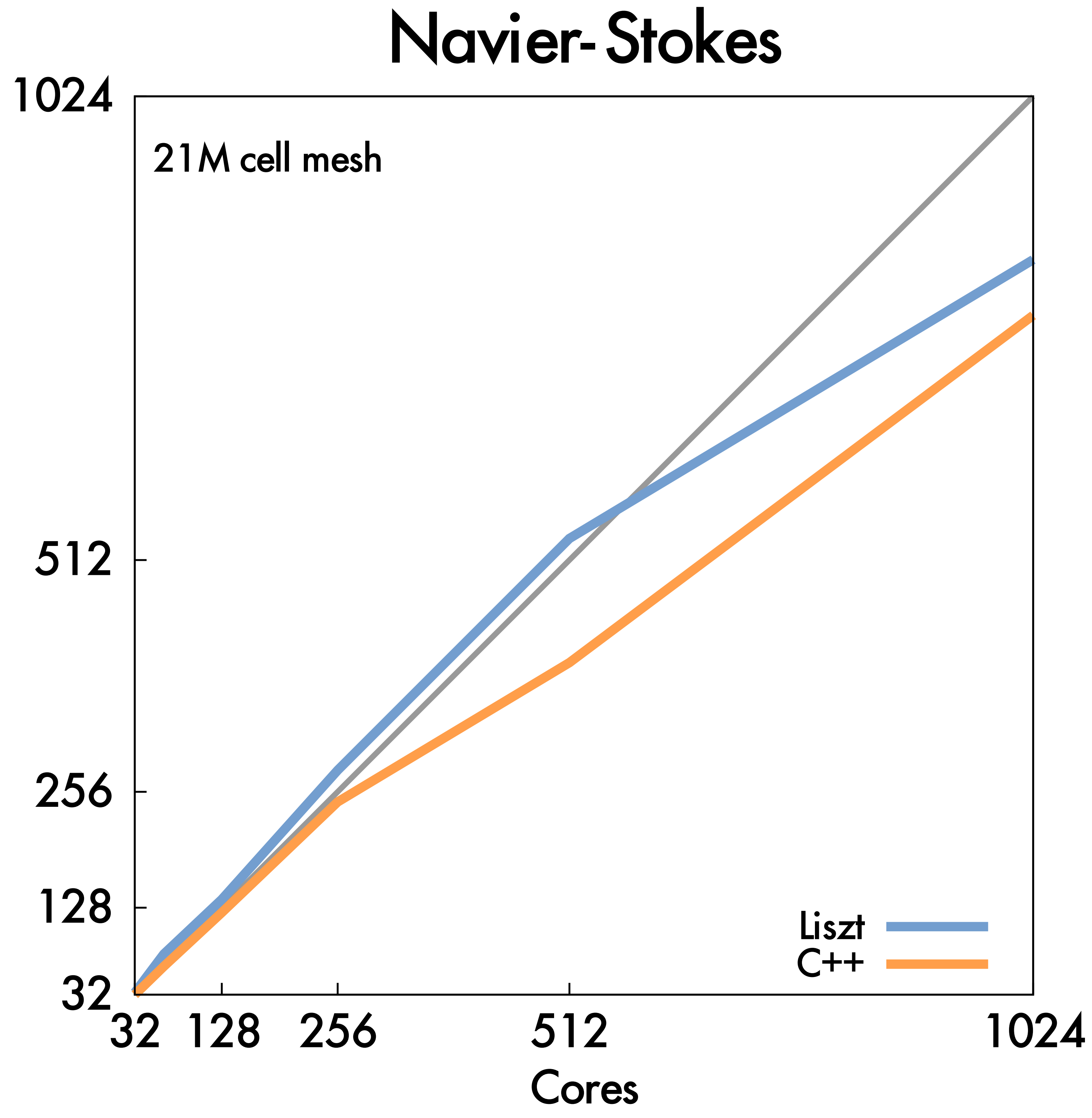
"Color" nodes in graph such that no connected nodes have the same color

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

Performance of Lizst program on a cluster

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





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)

Elements of good domain-specific programming system design

#1: good systems identify the most important cases, and provide most benefit in these situations

- **Structure of code mimics the natural structure of problems in the domain**
  - **Halide: pixel-wise view of filters: pixel(x,y) computed as expression of these input pixel values**
  - **Graph processing algorithms: per-vertex operations**

- **Efficient expression: common operations are easy and intuitive to express**

- **Efficient implementation: the most important optimizations in the domain are performed by the system for the programmer**
  - **My experience: a parallel programming system with "convenient" abstractions that precludes best-known implementation strategies will almost always fail**

#2: good systems are simple systems

- They have a small number of key primitives and operations
  - Halide: a few scheduling primitives for describing loop nests
  - Hadoop: map + reduce

- Allows compiler/runtime to focus on optimizing these primitives
  - Provide parallel implementations, utilize appropriate hardware

- Common question that good architects ask: "do we really need that?"
  (can this concept be reduced to a primitive we already have?)
  - For every domain-specific primitive in the system: there better be a strong performance or expressivity justification for its existence

#3: good primitives compose

- **Composition of primitives allows for wide application scope, even if scope is limited to a domain**
  - e.g., frameworks discussed today support a wide variety of graph algorithms
  - Halide's loop ordering + loop interleaving schedule primitives allow for expression of wide range of schedules
- **Composition often allows optimization to generalizable**
  - If system can optimize A and optimize B, then it can optimize programs that combine A and B
- **Common sign that a feature <u>should not</u> be added (or added in a different way):**
  - The new feature does not compose with all existing features in the system
- **Sign of a good design:**
  - System ultimately is used for applications original designers never anticipated