

Lecture 16:

Domain-Specific Programming Systems

Parallel Computing
Stanford CS149, Winter 2019

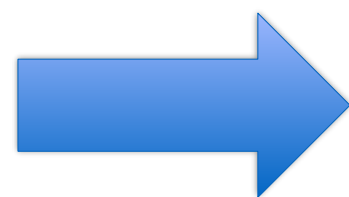
Power-constrained computing

- **Moore's law is failing and Dennard scaling is dead**
 - **Power limits how many transistors you can turn on**
- **Supercomputers/Datacenters are power constrained**
 - **Due to sheer scale of machine**
 - **Overall cost to operate (power for machine and for cooling)**
- **Mobile devices are power constrained**
 - **Limited battery life**
 - **Heat dissipation**

Computing system power

$$Power = Energy_{Op} \times \frac{Ops}{second}$$

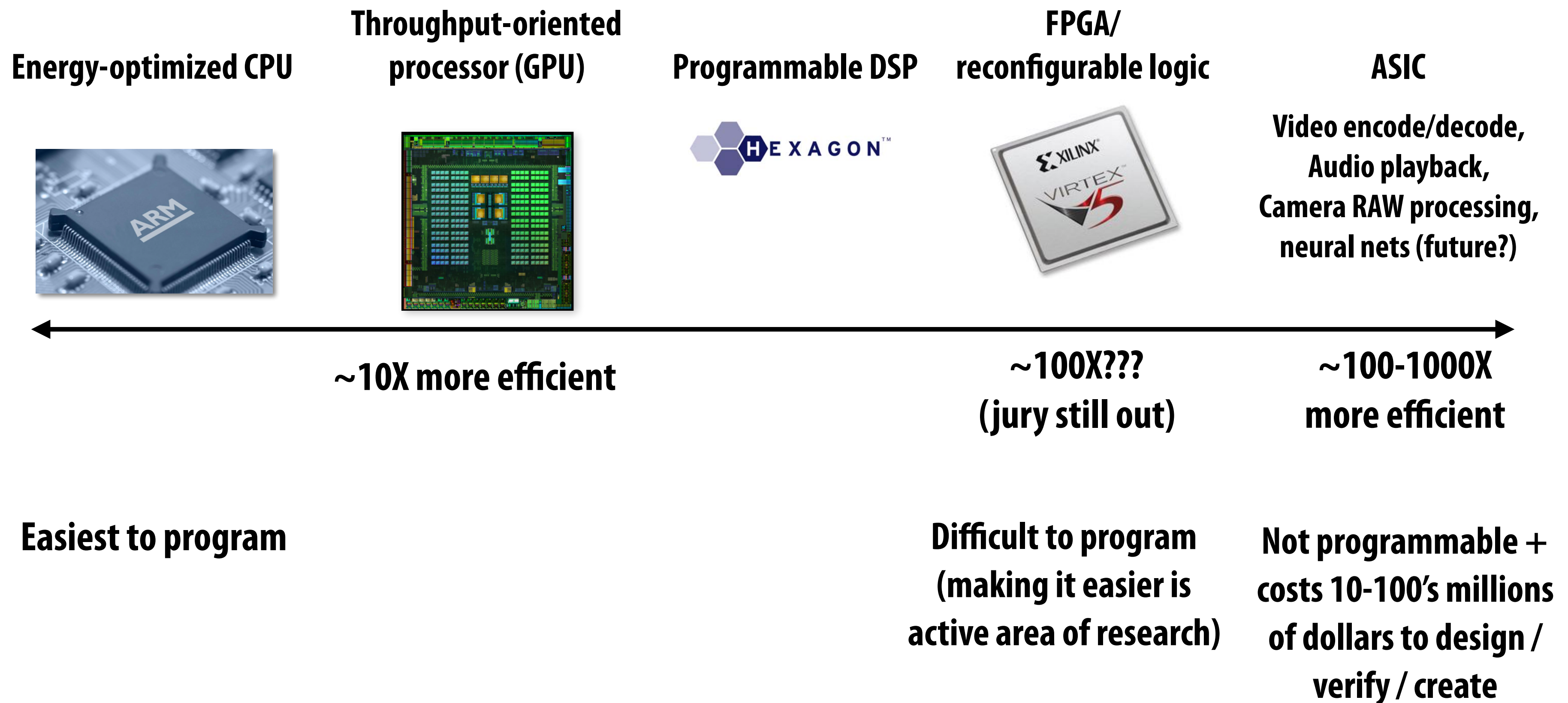
FIXED



Efficiency benefits of compute specialization

- **Rules of thumb: compared to high-quality C code on CPU...**
- **Throughput-maximized processor architectures: e.g., GPU cores**
 - **Approximately 10x improvement in perf / watt**
 - **Assuming code maps well to wide data-parallel execution and is compute bound**
- **Fixed-function ASIC (“application-specific integrated circuit”)**
 - **Can approach 100-1000x or greater improvement in perf/watt**
 - **Assuming code is compute bound and is not floating-point math**

Summary: choosing the right tool for the job

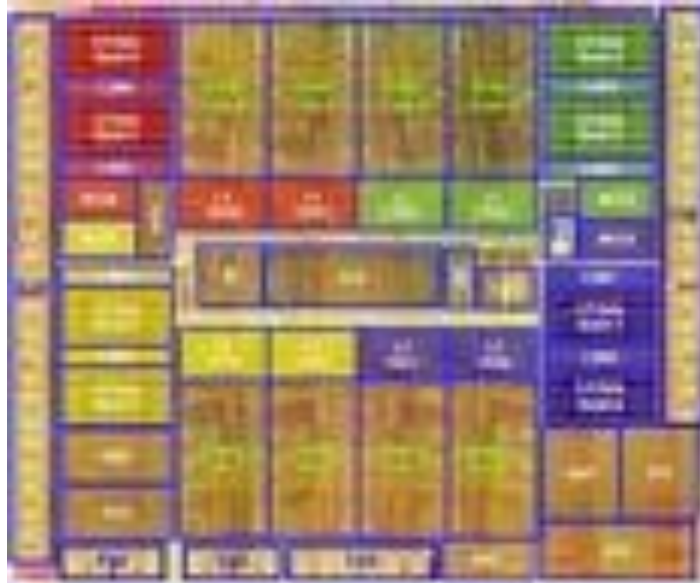


Heterogeneous processing for efficiency

- **Heterogeneous parallel processing: use a mixture of computing resources that fit mixture of needs of target applications**
 - Latency-optimized sequential cores, throughput-optimized parallel cores, domain-specialized fixed-function processors
 - Examples exist throughout modern computing: mobile processors, servers, supercomputers
- **Traditional rule of thumb in “good system design” is to design simple, general-purpose components**
 - This is not the case in emerging systems (optimized for perf/watt)
 - Today: want collection of components that meet perf requirement AND minimize energy use
- **Challenge of using these resources effectively is pushed up to the programmer**
 - Current CS research challenge: how to write efficient, portable programs for emerging heterogeneous architectures?

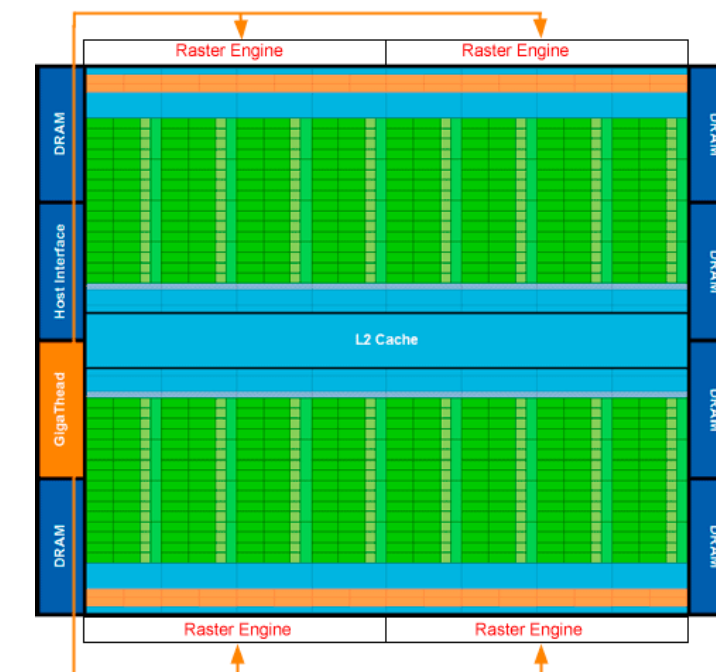
Heterogeneous Parallel Programming Today

Pthreads
OpenMP



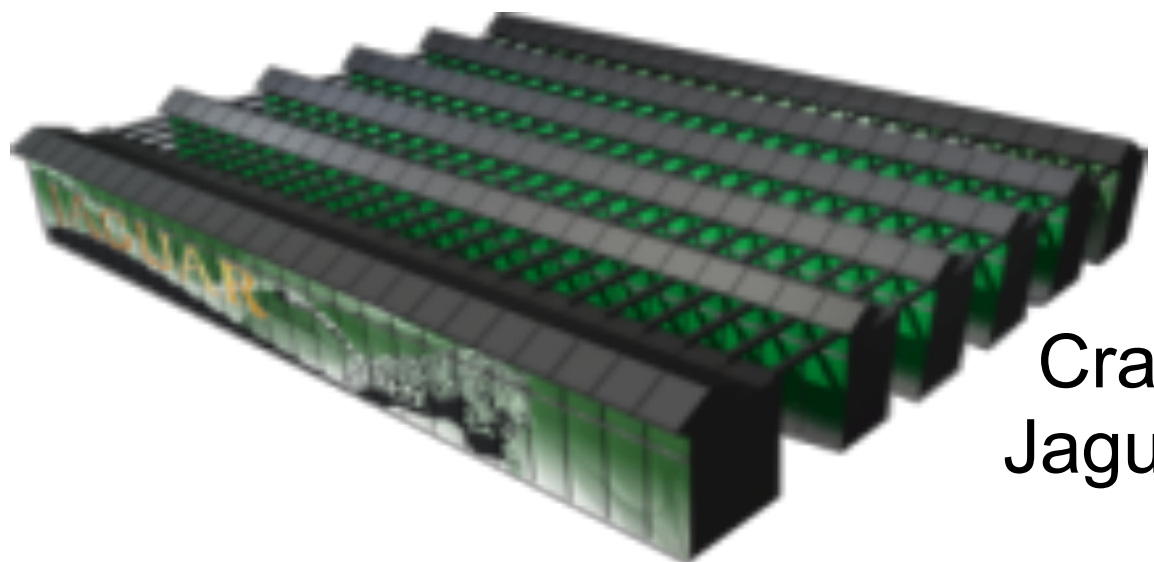
Sun
T2

CUDA
OpenCL



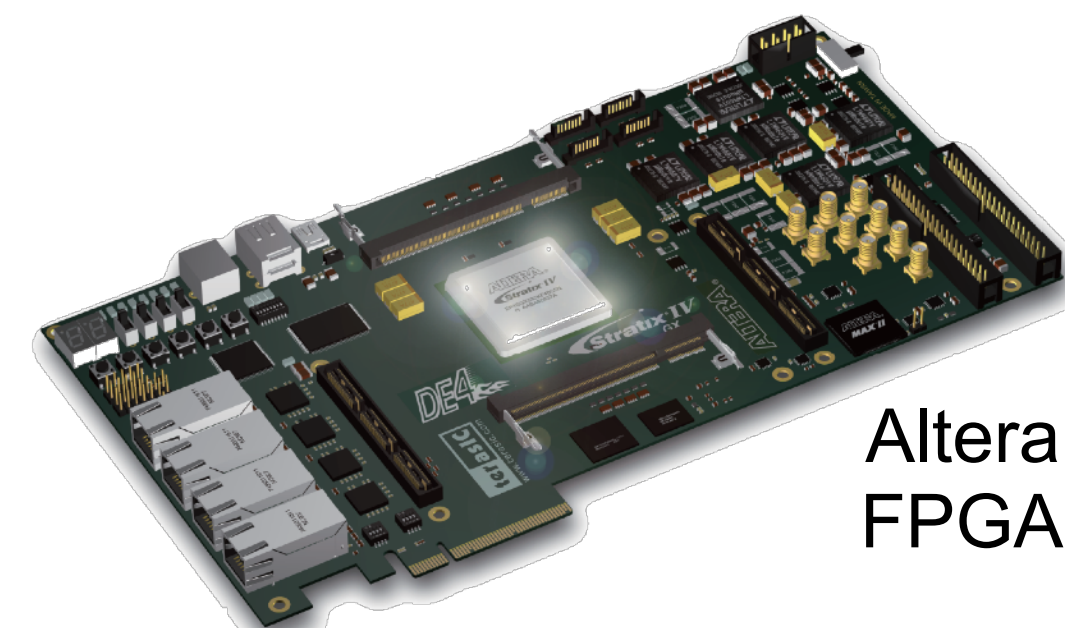
Nvidia
Fermi

MPI
PGAS
Spark



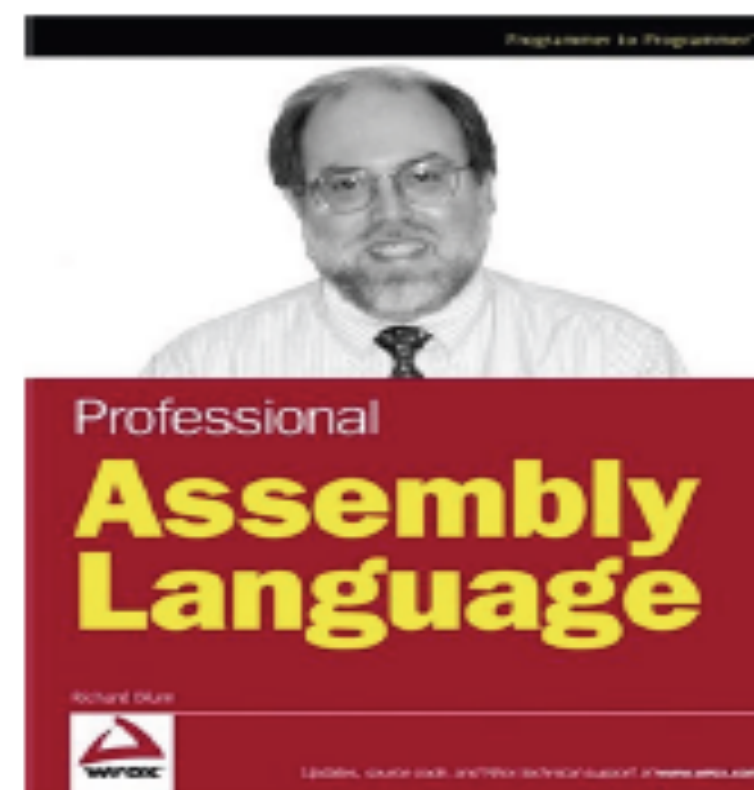
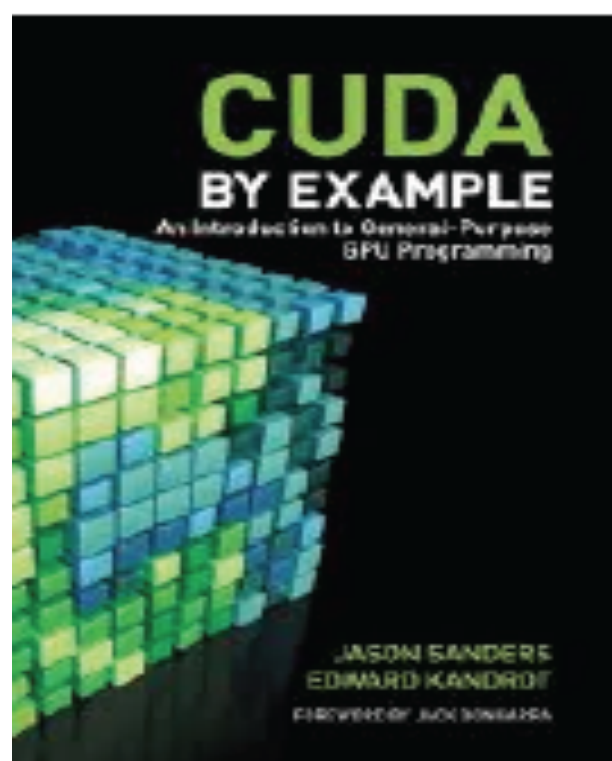
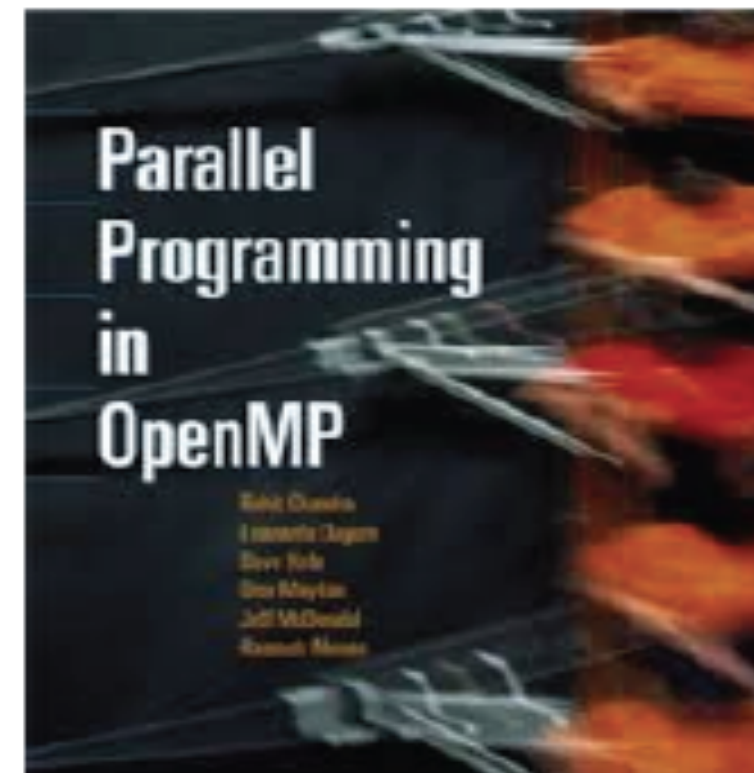
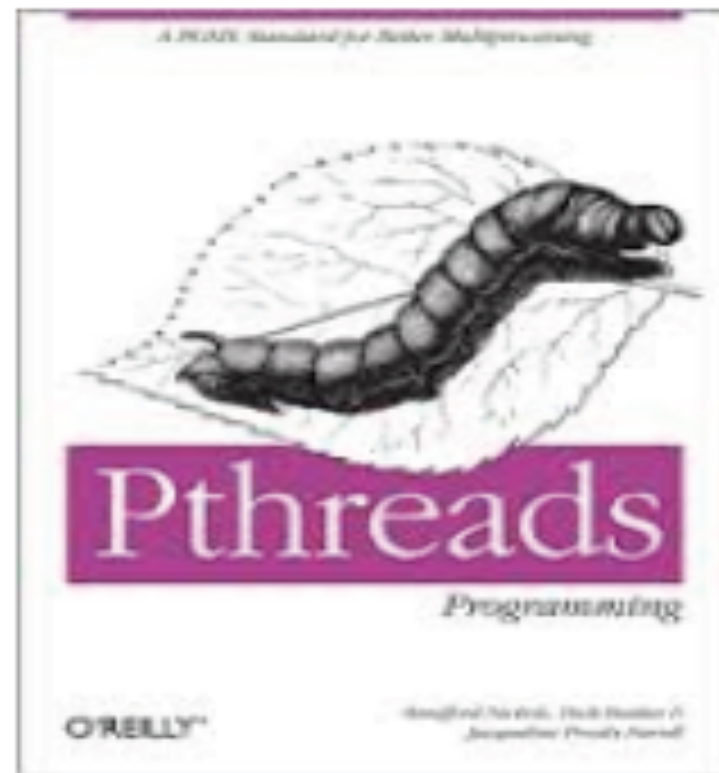
Cray
Jaguar

Verilog
VHDL



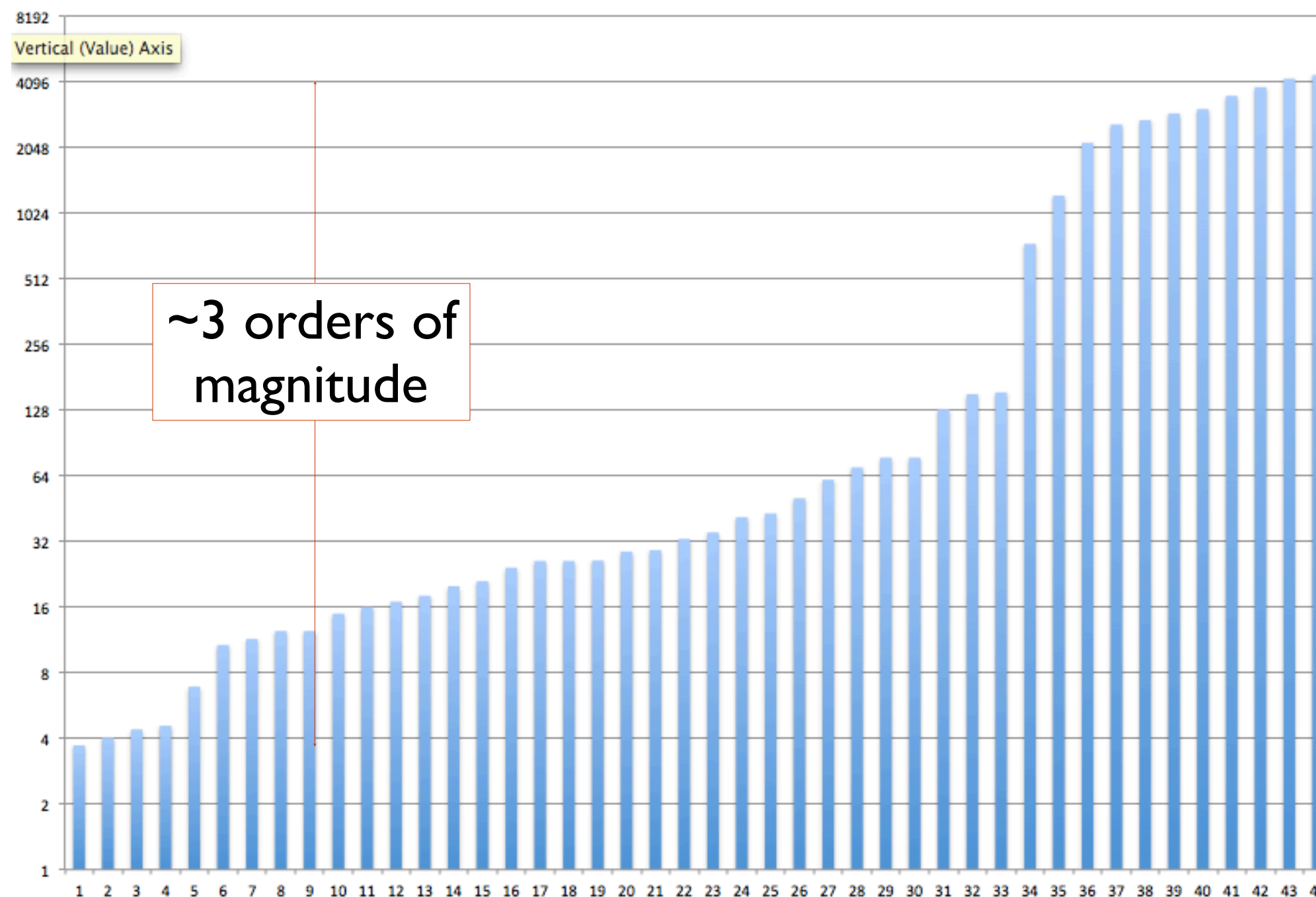
Altera
FPGA

EXPERT PROGRAMMERS \Rightarrow LOW PRODUCTIVITY



Expert Programming is Difficult

Image Filter in OpenMP



Optimizations:

- Precomputing twiddle
- Not computing what not part of the filter
- Transposing the matrix
- Using SSE

DSL Hypothesis

**It is possible to write one program
and
run it efficiently on all these machines**

Domain Specific Languages

■ Domain Specific Languages (DSLs)

- Programming language with restricted expressiveness for a particular domain
- High-level, usually declarative, and deterministic



Big-Data Analytics Programming Challenge

Data Analytics
Application

Data Prep

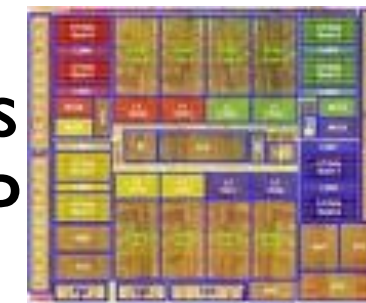
Data
Transform

Network
Analysis

Predictive
Analytics

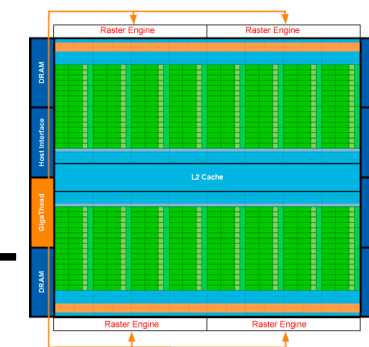
Ideal Parallel
Programming
Language

Pthreads
OpenMP



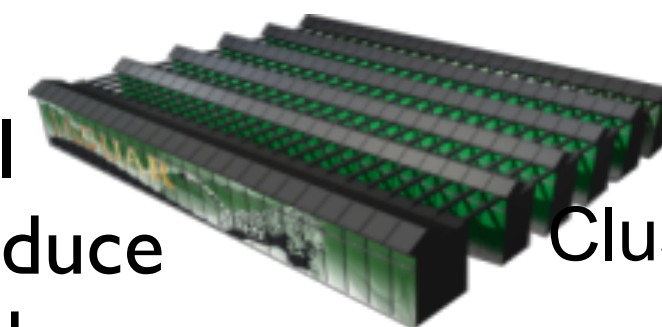
Multicore

CUDA
OpenCL



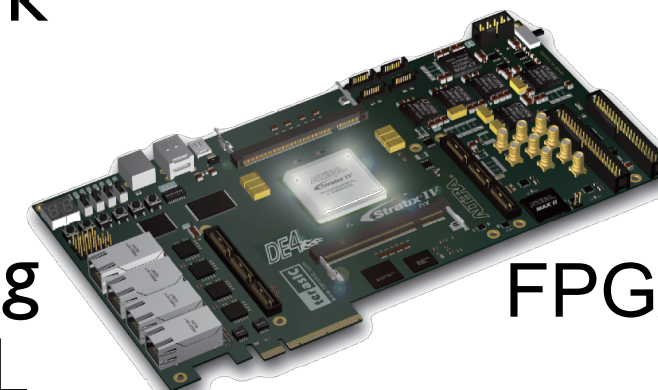
GPU

MPI
Map Reduce
Spark



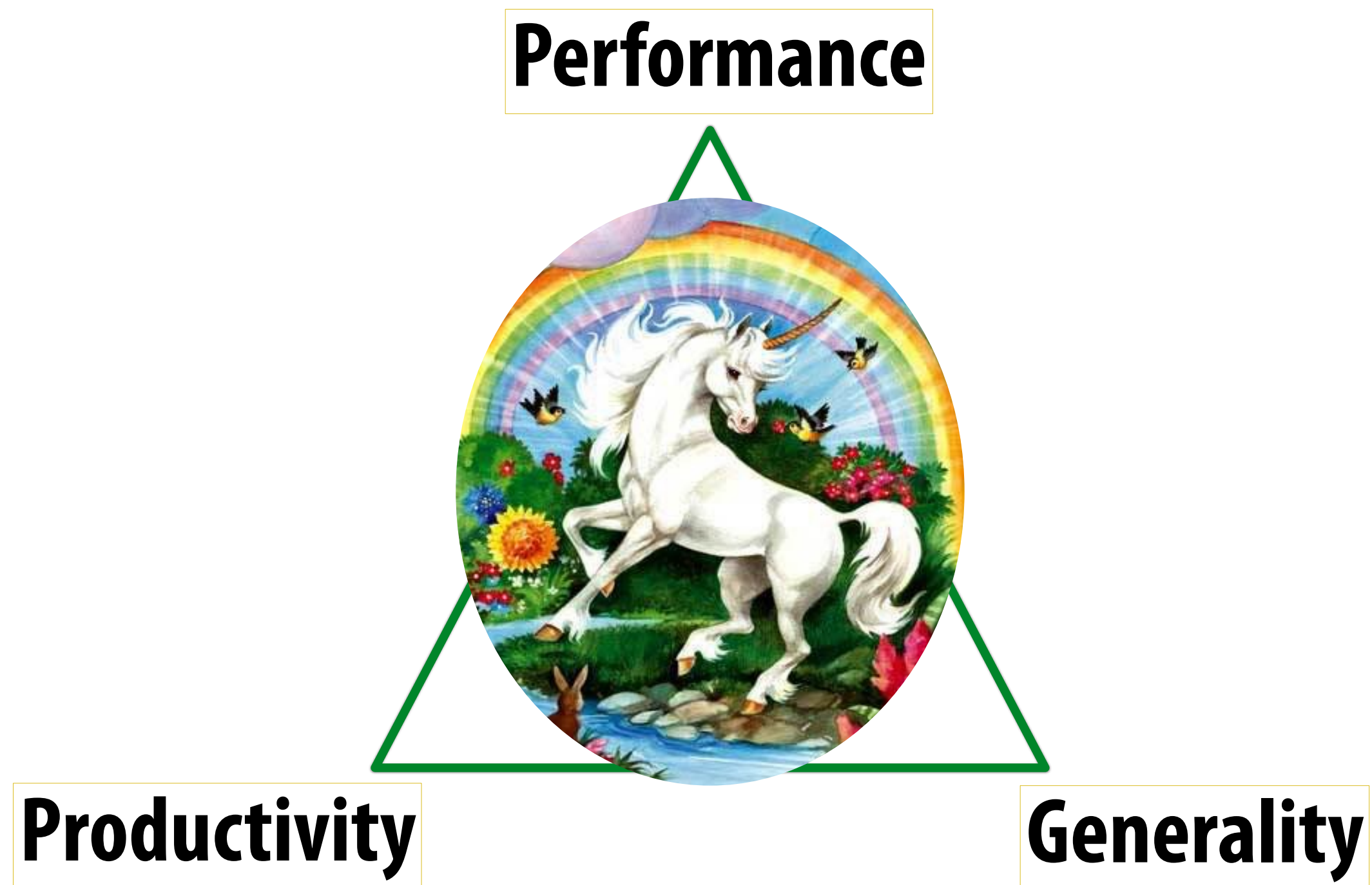
Cluster

Verilog
VHDL

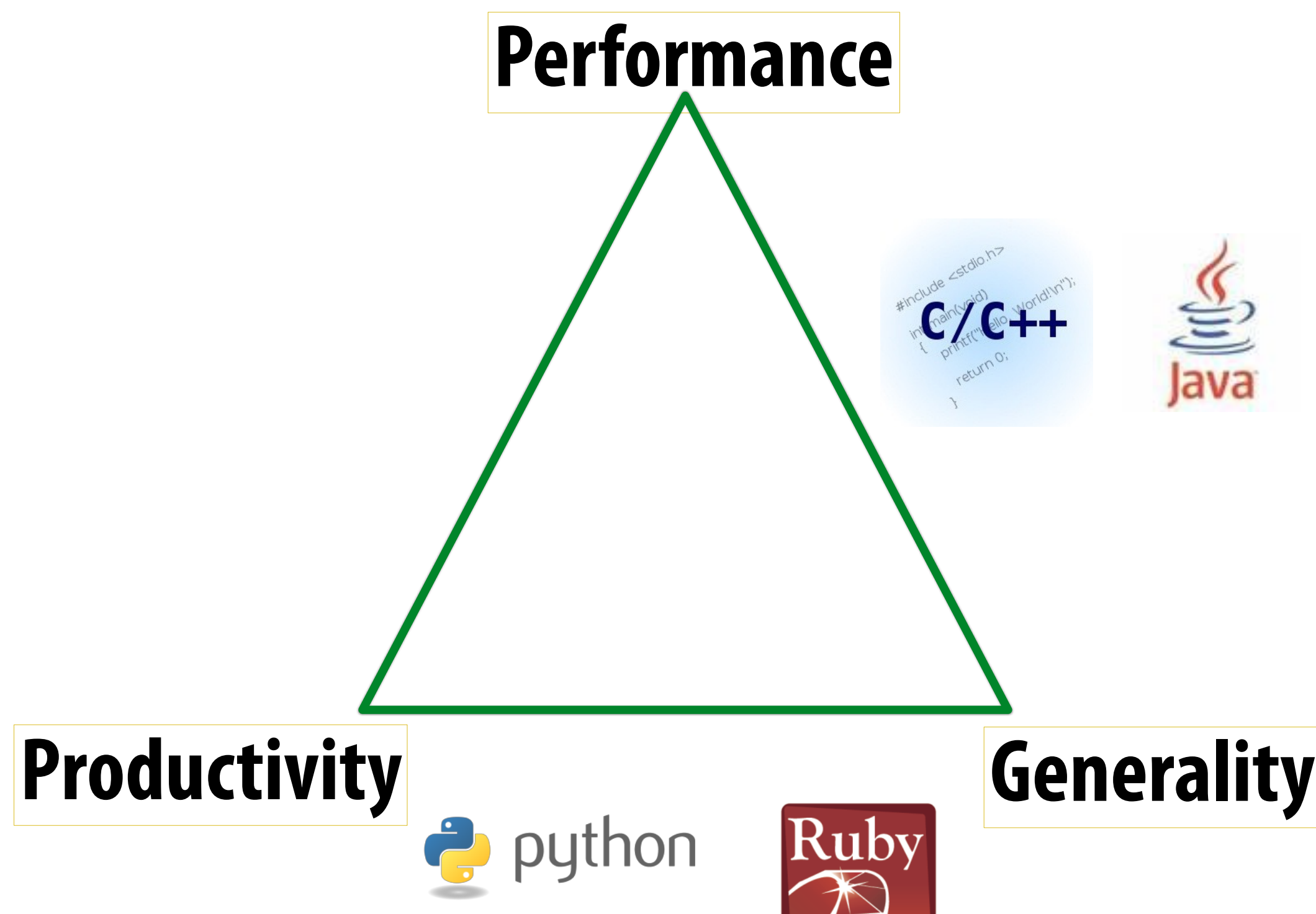


FPGA

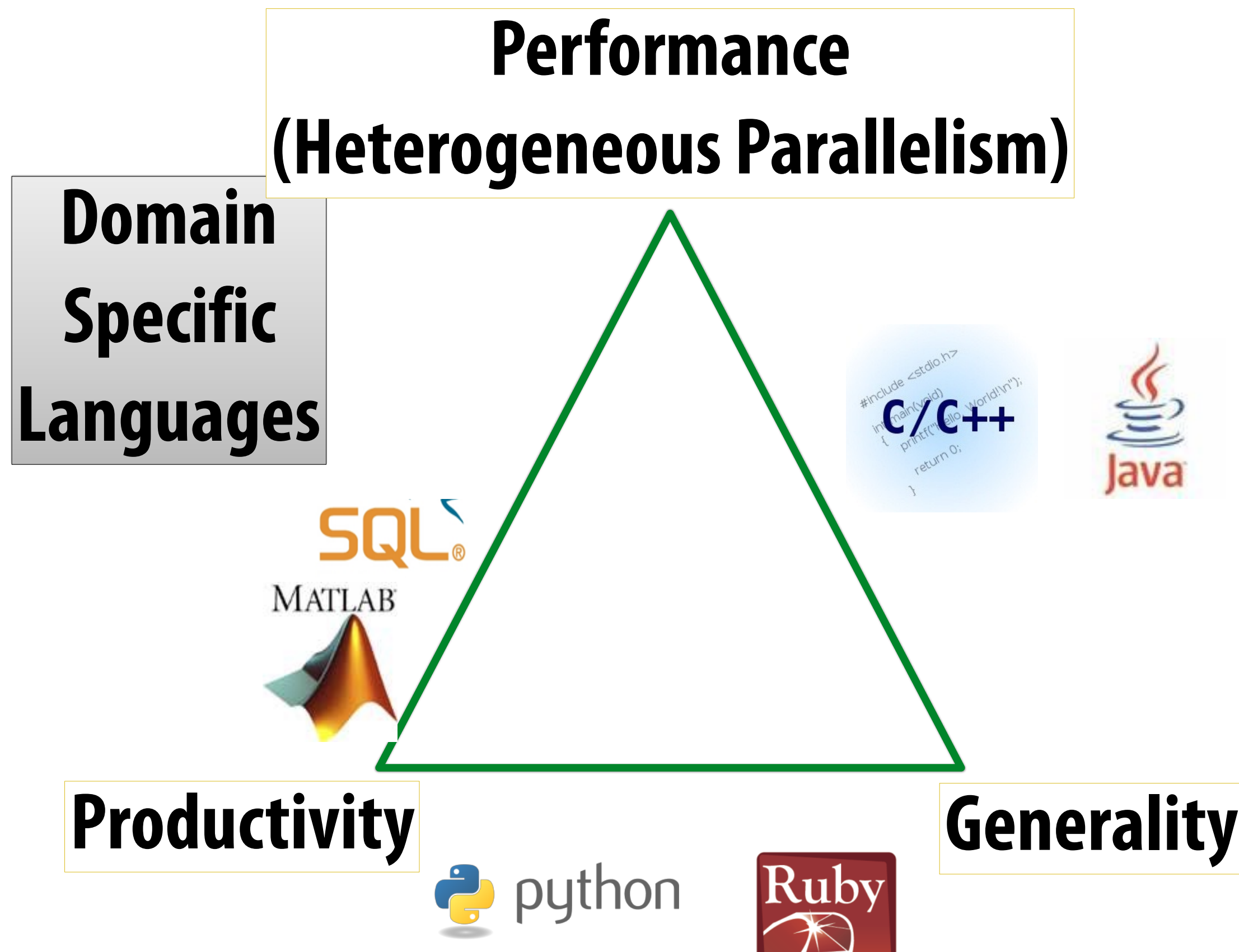
The Ideal Parallel Programming Language



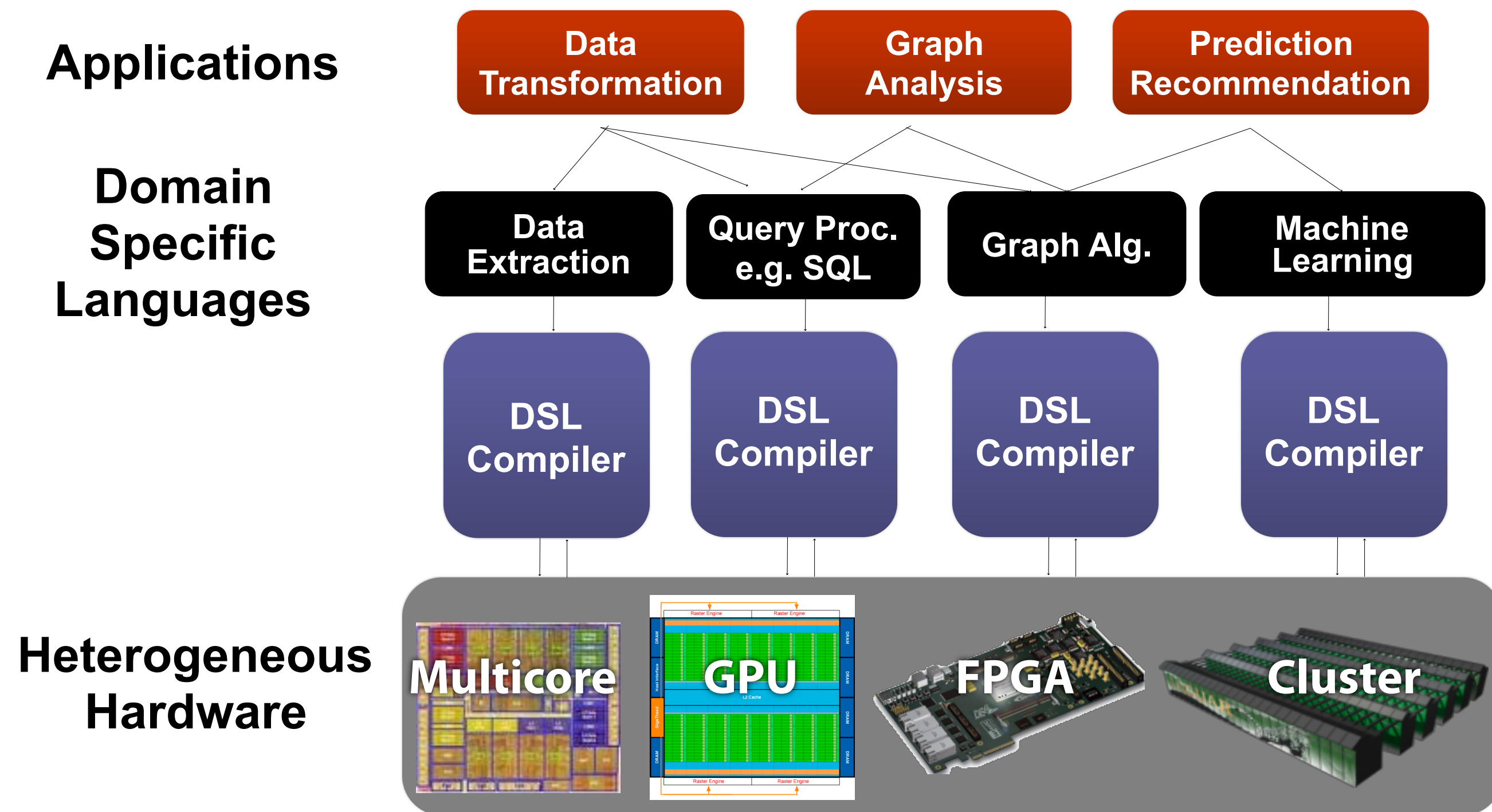
Successful Languages (not exhaustive ;-))



Way Forward \Rightarrow Domain Specific Languages



High Performance DSLs for Data Analytics



OptiML: Overview

- Provides a familiar (MATLAB-like) language and API for writing ML applications
 - Ex. **val** `c = a * b` (`a, b` are `Matrix[Double]`)
- Implicitly parallel data structures
 - Base types
 - `Vector[T]`, `Matrix[T]`, `Graph[V,E]`, `Stream[T]`
 - Subtypes
 - `TrainingSet`, `IndexVector`, `Image`, ...
- Implicitly parallel control structures
 - `sum{...}`, `(0::end){...}`, `gradient{...}`, `untilconverged{...}`
 - Allow anonymous functions with restricted semantics to be passed as arguments of the control structures

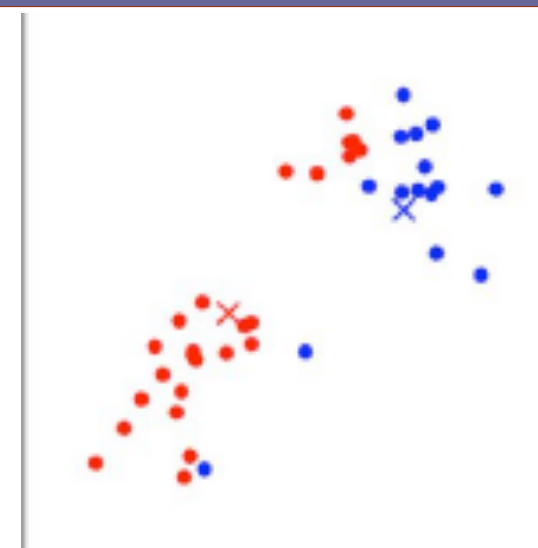
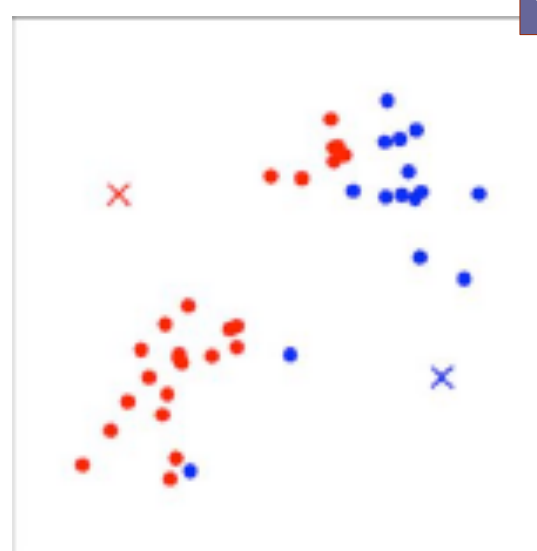
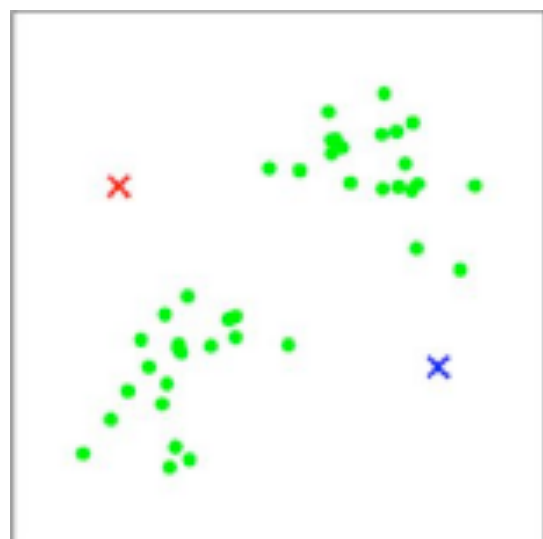
K-means Clustering in OptiML

```
untilconverged(kMeans, tol){kMeans =>
  val clusters = samples.groupRowsBy { sample =>
    kMeans.mapRows(mean => dist(sample, mean)).minIndex
  }
  val newKmeans = clusters.map(e => e.sumLength)
  newKmeans
}
```

assign each
sample to the
closest mean

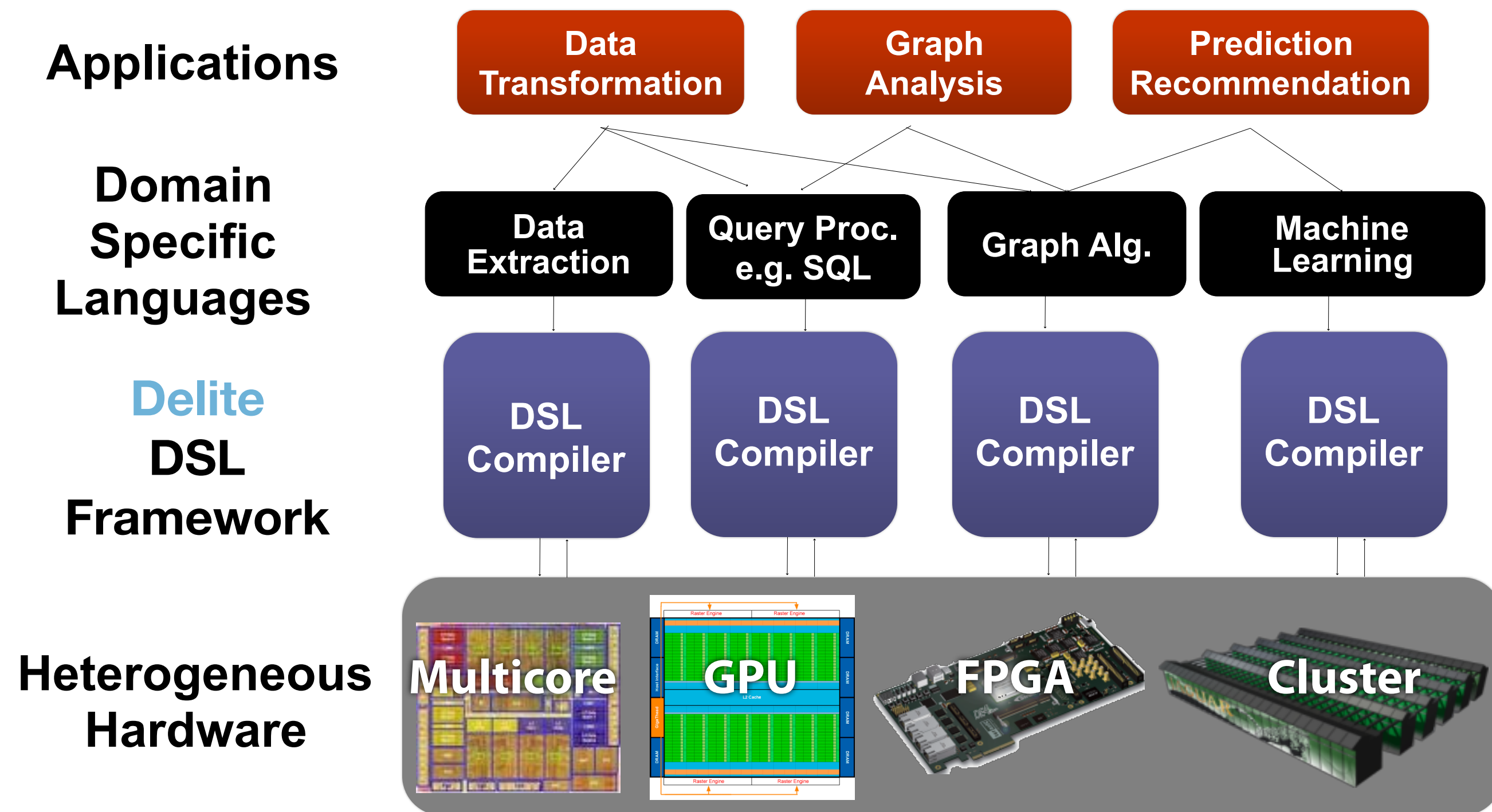
calculate
distances to
current means

move each cluster centroid to the
mean of the points assigned to it



- No explicit map-reduce, no key-value pairs
- No distributed data structures (e.g. RDDs)
- No annotations for hardware design
- Efficient multicore and GPU execution
- Efficient cluster implementation
- Efficient FPGA hardware

Common DSL Infrastructure: Delite

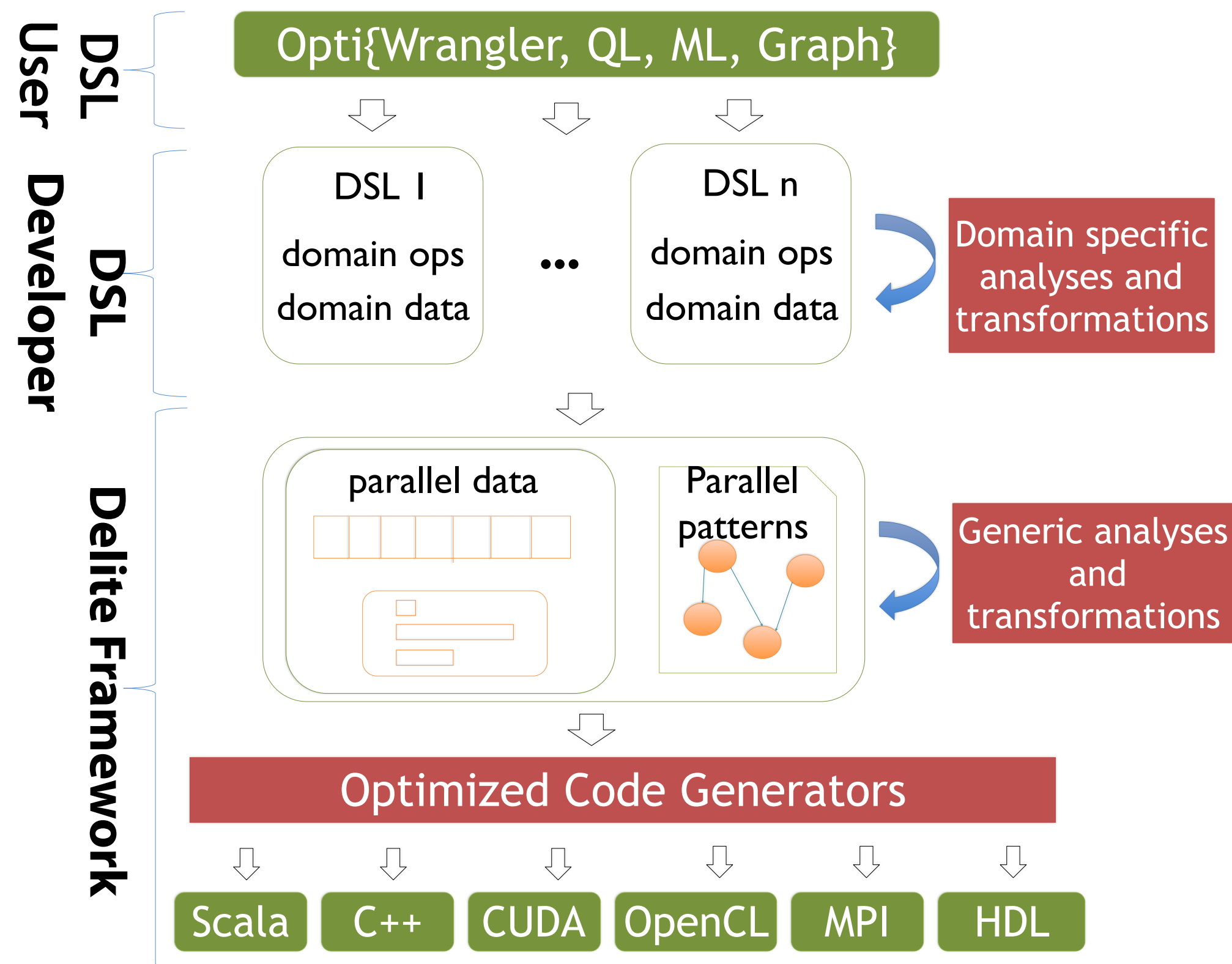


Delite: A Framework for High Performance DSLs

- **Overall Approach: Generative Programming for “Abstraction without regret”**
 - **Embed compilers in Scala libraries: Scala does syntax and type checking**
 - **Use metaprogramming with LMS (type-directed staging) to build an intermediate representation (IR) of the user program**
 - **Optimize IR and map to multiple targets**
- **Goal: Make embedded DSL compilers easier to develop than stand alone DSLs**
 - **As easy as developing a library**

Delite Overview

K. J. Brown et. al., "A heterogeneous parallel framework for domain-specific languages," PACT, 2011.



Key elements

- **DSLs embedded in Scala**
- **IR created using type-directed staging**
- **Domain specific optimization**
- **General parallelism and locality optimizations**
- **Optimized mapping to HW targets**

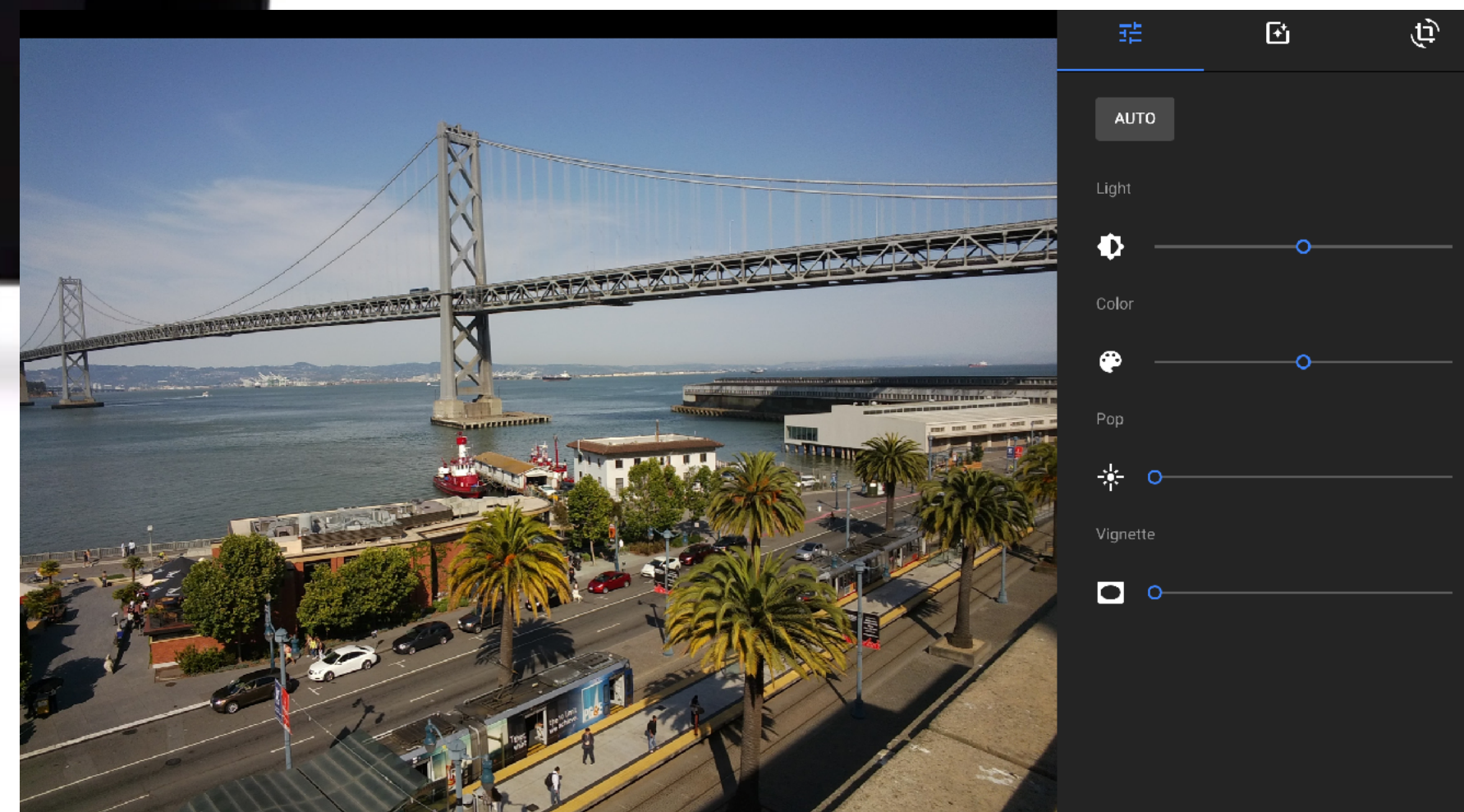
DSL Example:

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 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

```
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;
    }

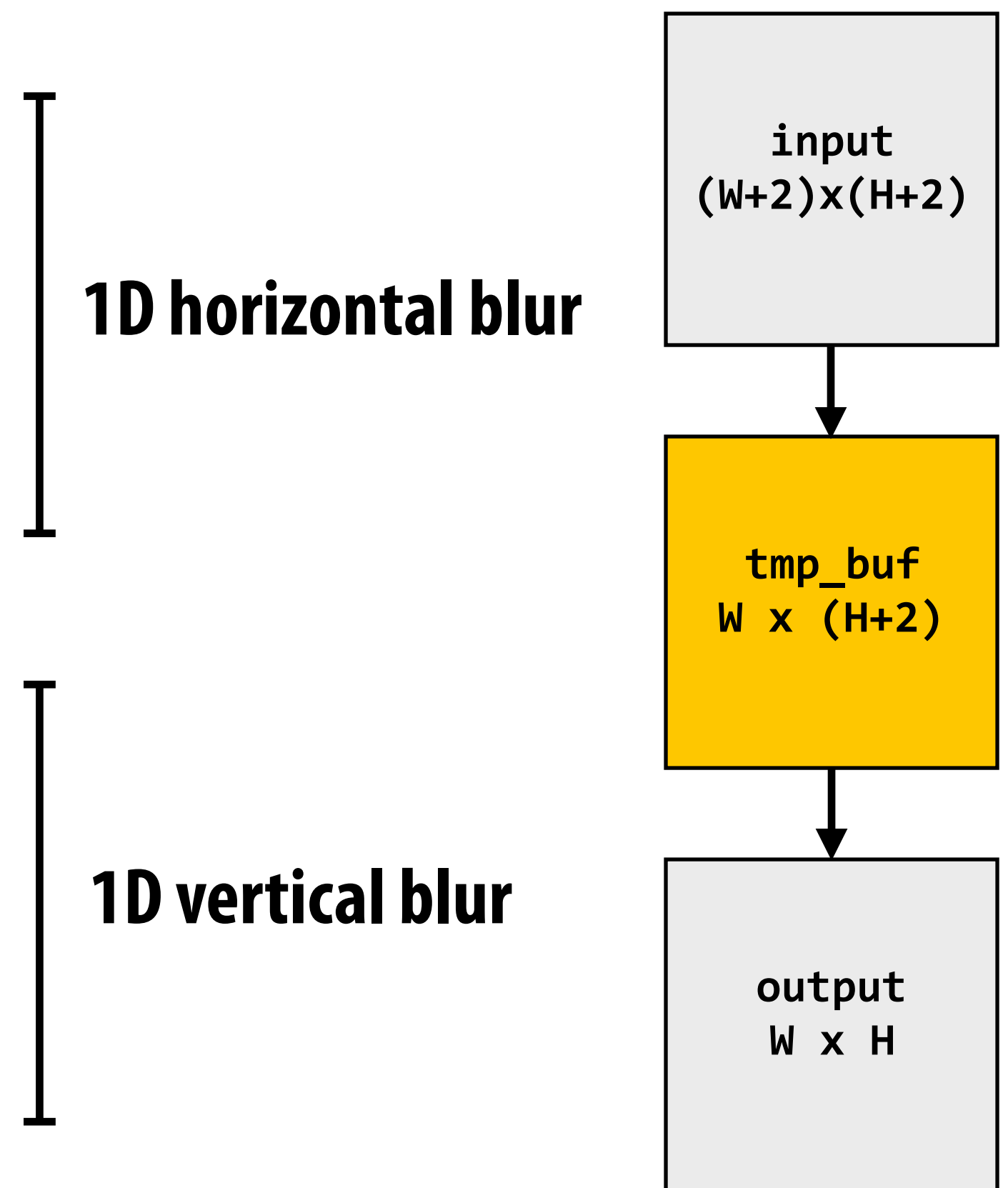
for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
        output[j*WIDTH + i] = tmp;
    }
}
```

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



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 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;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * 3];
float output[WIDTH * HEIGHT];
```

Only 3 rows of intermediate buffer need to be allocated

```
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;
```

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

```
        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;
```

```
        }
```

```
    }
```

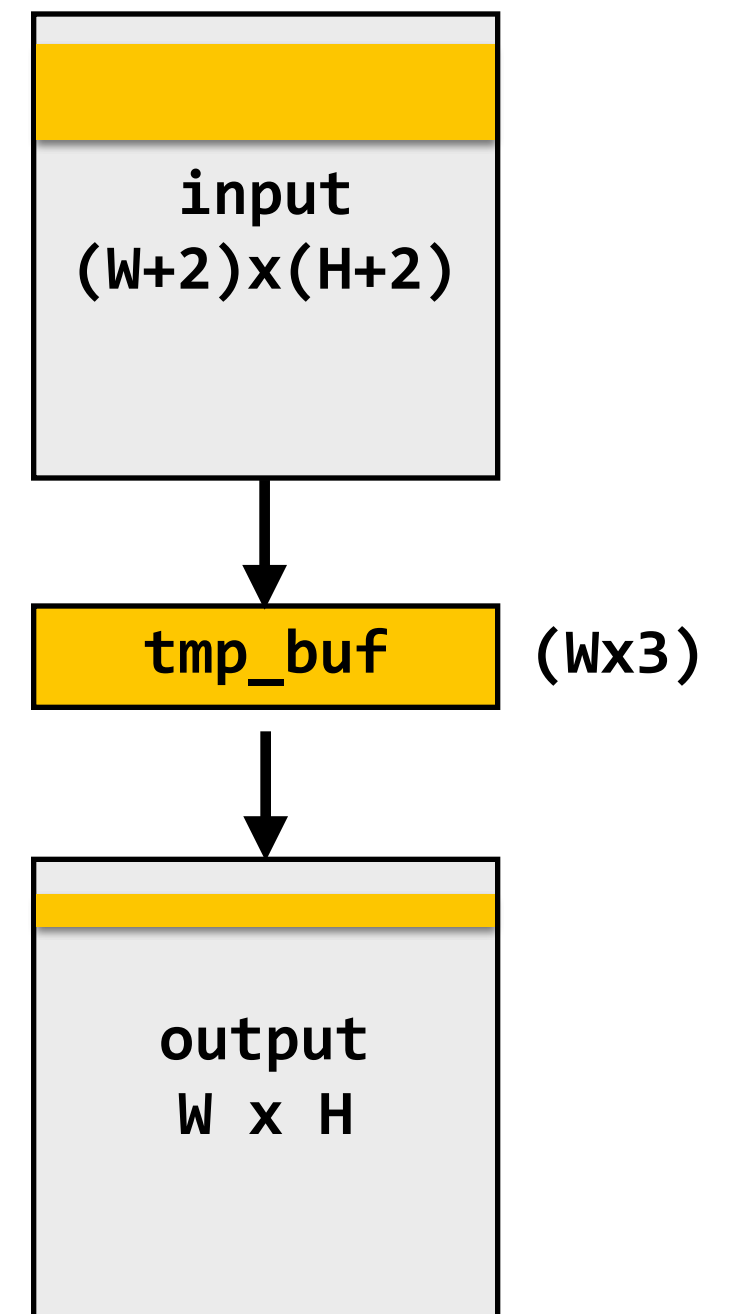
Combine them together to get one row of output

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;
```

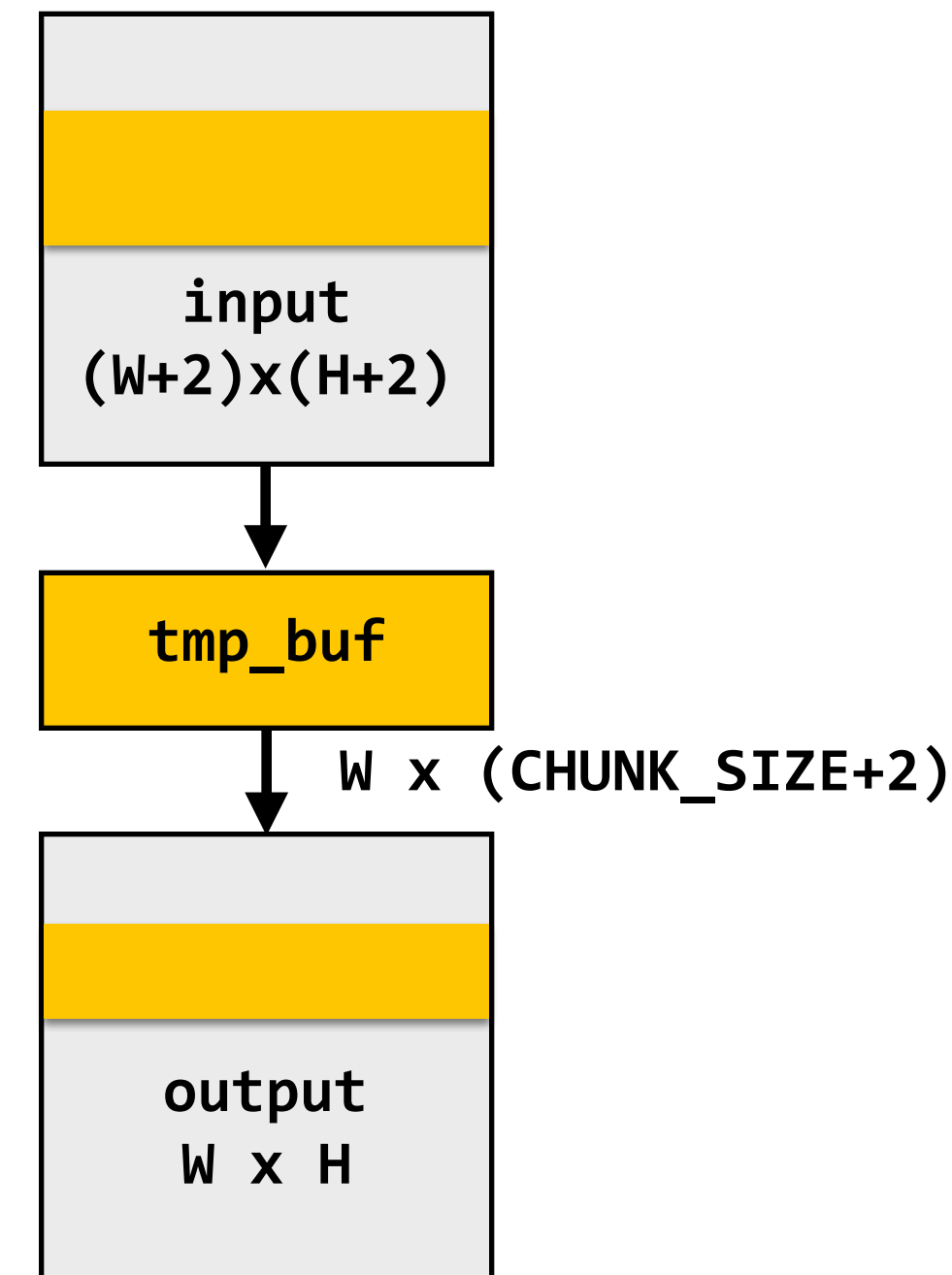
```
            }
```

```
    }
```

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 chunk of output:
(assume `CHUNK_SIZE = 16`)

- Step 1: $18 \times 3 \times \text{WIDTH}$ work

- Step 2: $16 \times 3 \times \text{WIDTH}$ work

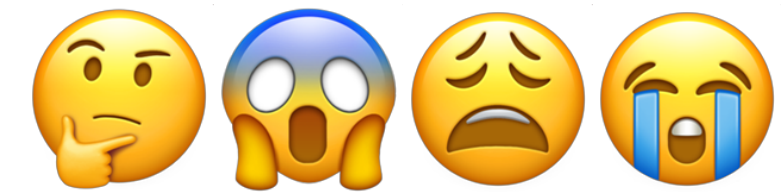
Total work per image: $(34/16) \times 3 \times \text{WIDTH} \times \text{HEIGHT}$
 $= 6.4 \times \text{WIDTH} \times \text{HEIGHT}$

Trends to ideal value of $6 \times \text{WIDTH} \times \text{HEIGHT}$ 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

```
Var x, y;
Func blurx, blurry, 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));
blurry(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(blurry(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 (0:800,0:600)
Halide::Buffer<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`

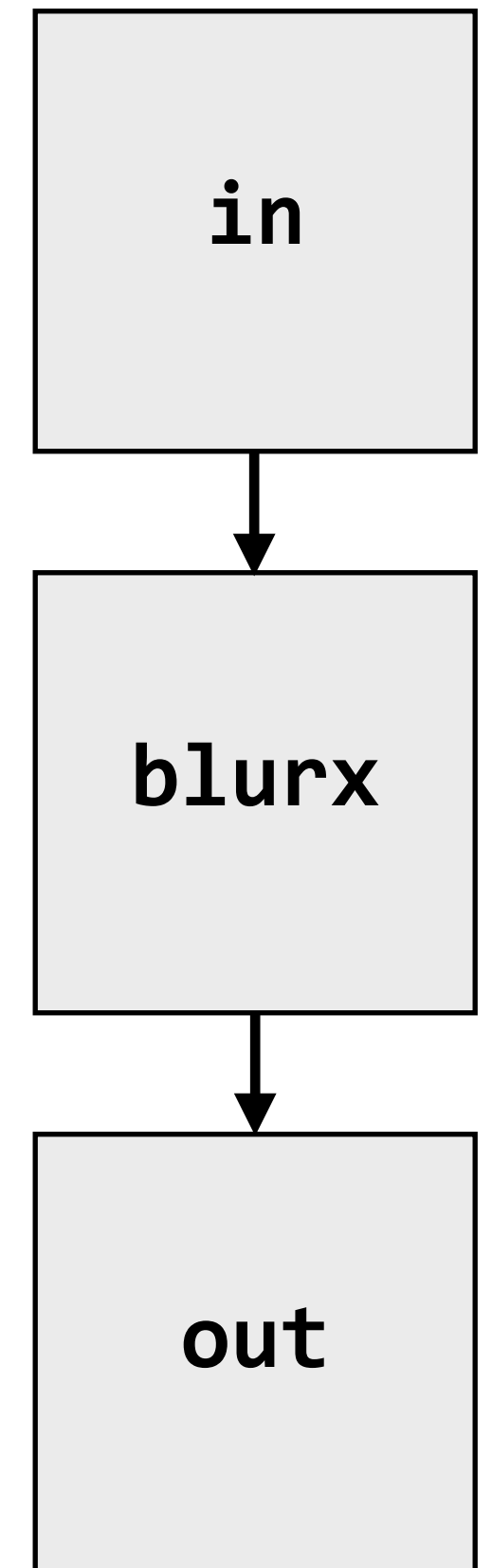
Halide function: an infinite (but discrete) set of values defined on N-D domain

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.

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 operations are 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 800x600
```

```
Halide::Buffer<uint8_t> result = out.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;  
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);
```

Produce elements `blurx` on demand for each tile of output.

Vectorize the `x` (innermost) loop

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

```
// 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 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:  
      // vectorize body of this loop with SIMD instructions  
      for xi=0 to 256 by 8:  
        idx_x = x*256+xi;  
        idx_y = y*32+yi  
        out(idx_x, 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_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
```

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

```
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) = ...
```

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
```

tile of blurx is
computed here

```
    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 consumed here

An interesting Halide schedule

```
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.store_at(out, x) | Compute necessary elements of blurx within out's xi loop nest, but fill in tile-sized buffer allocated at x loop nest. |
| blurx.compute_at(out, xi); | |

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

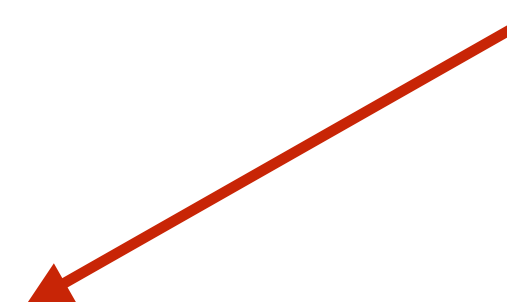
```
    allocate 258x34 buffer for tile tmp_blurx
```

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

```
          // 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) = ...
```

Can compiler be smarter?



“Sliding optimization” (reduces redundant computation)

```
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.store_at(out, x) Compute necessary elements of blurx within out's xi loop
blurx.compute_at(out, xi); nest, but fill in tile-sized buffer allocated at x loop nest.

```
for y=0 to num_tiles_y:  
  for x=0 to num_tiles_x:  
    allocate 258x34 buffer for tile tmp_blurx
```

```
    for yi=0 to 32:  
      for xi=0 to 256:  
        idx_x = x*256+xi;  
        idx_y = y*32+yi;  
  
        if (yi=0) {  
          // compute 3 elements of blurx needed for out(idx_x, idx_y) here  
          for (blur_x=0 to 3)  
            tmp_blurx(blur_x) = ...  
        } else  
          // only compute one additional element of tmp_blurx
```

```
    out(idx_x, idx_y) = ...
```

Steady state: only one new
element of tmp_blurx needs to
be computed per output



“Folding optimization” (reduces intermediate storage)

```
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.store_at(out, x) Compute necessary elements of blurx within out's xi loop
blurx.compute_at(out, xi); nest, but fill in tile-sized buffer allocated at x loop nest.

```
for y=0 to num_tiles_y:  
  for x=0 to num_tiles_x:  
    allocate 3x256 buffer for tmp_blurx  
  
    for yi=0 to 32:  
      for xi=0 to 256:  
        idx_x = x*256+xi;  
        idx_y = y*32+yi;  
  
        if (yi=0) {  
          // compute 3 elements of blurx needed for out(idx_x, idx_y) here  
          for (blur_x=0 to 3)  
            tmp_blurx(blur_x) = ...  
        } else  
          // only compute one additional element of tmp_blurx  
  
        out(idx_x, idx_y) = ...
```

Circular buffer of 3 rows →

Steady state: only one new element of tmp_blurx needs to be computed per output

← **Accesses to tmp_blurx modified to access appropriate row of circular buffer: e.g., ((idx_y+1)%3)**

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 to schedule application efficiently on machine (but it's slow and tedious), so give 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 (pthreads, AVX intrinsics, etc.)
 - There are two major examples of deviation from this philosophy. What are they?

Constraints on language

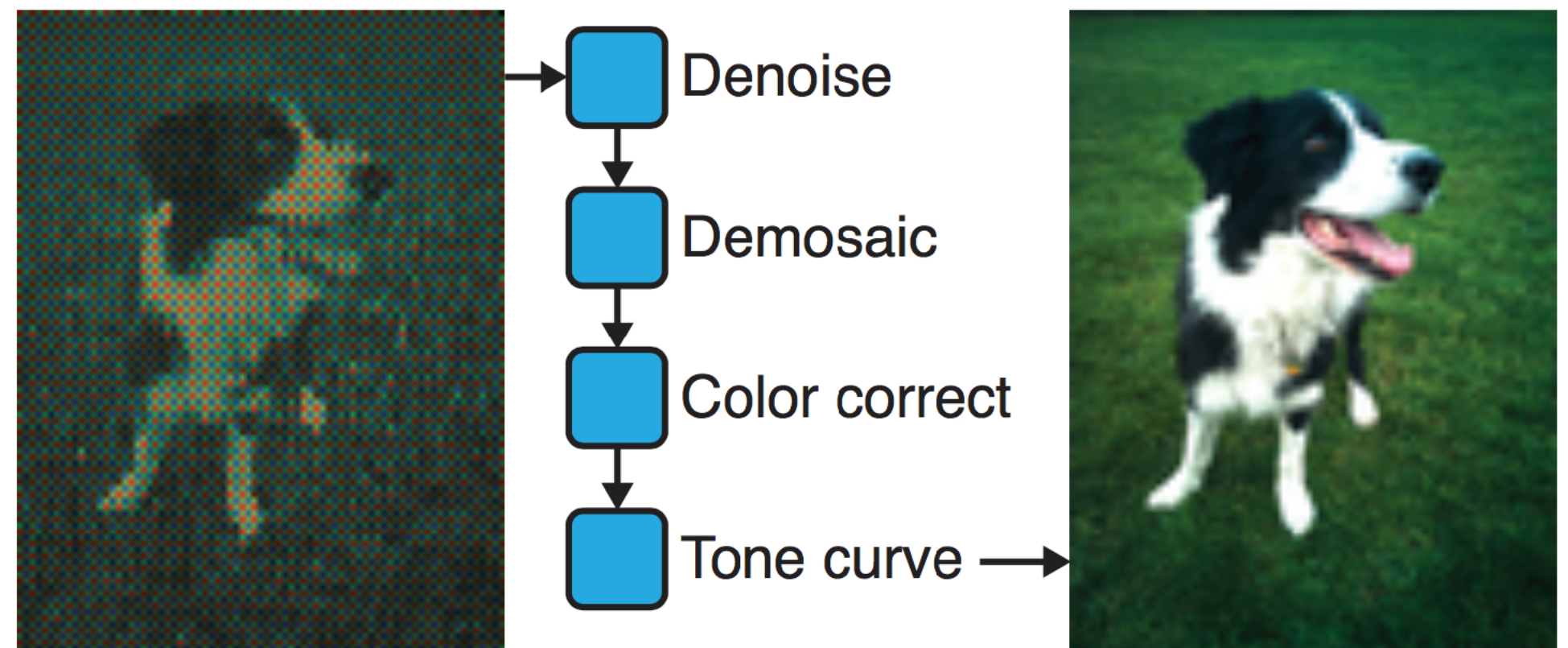
(to enable compiler to provide desired services)

- **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**

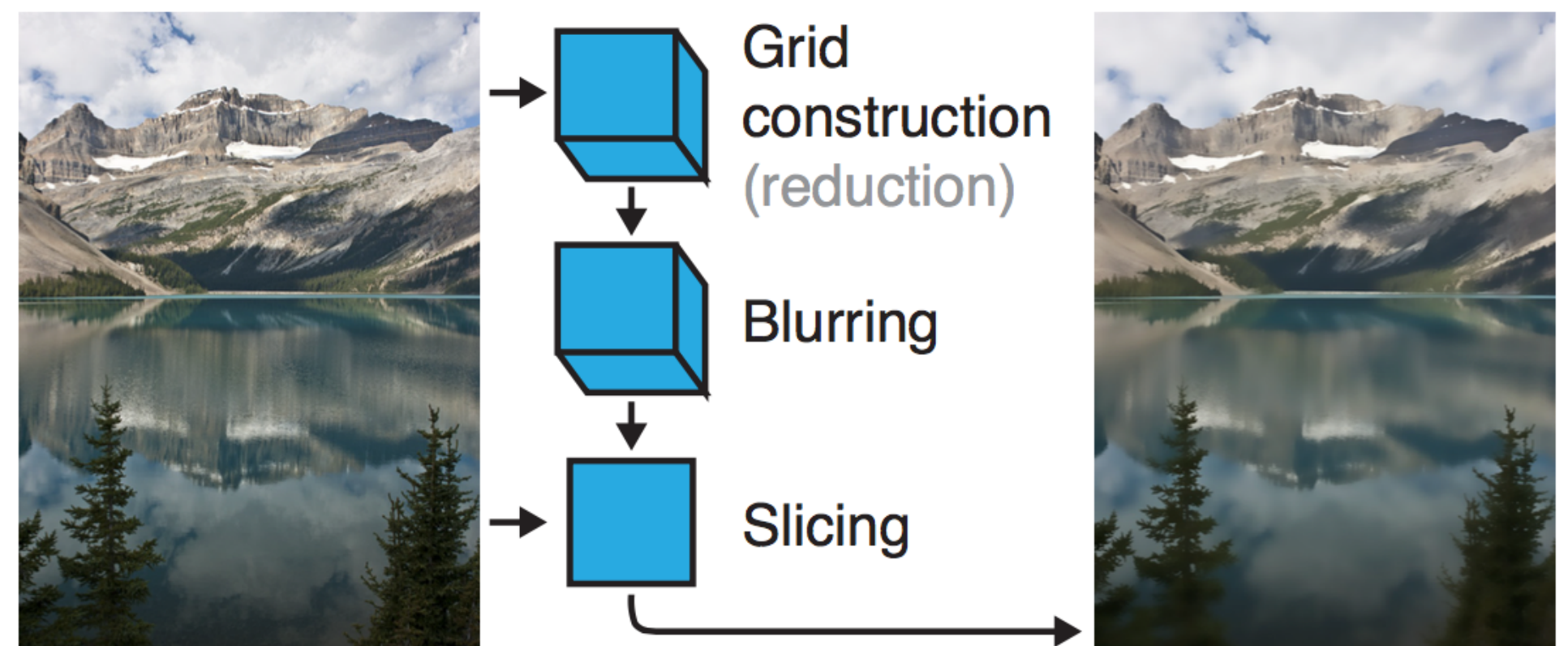
Initial academic Halide results

[Ragan-Kelley 2012]

- **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**

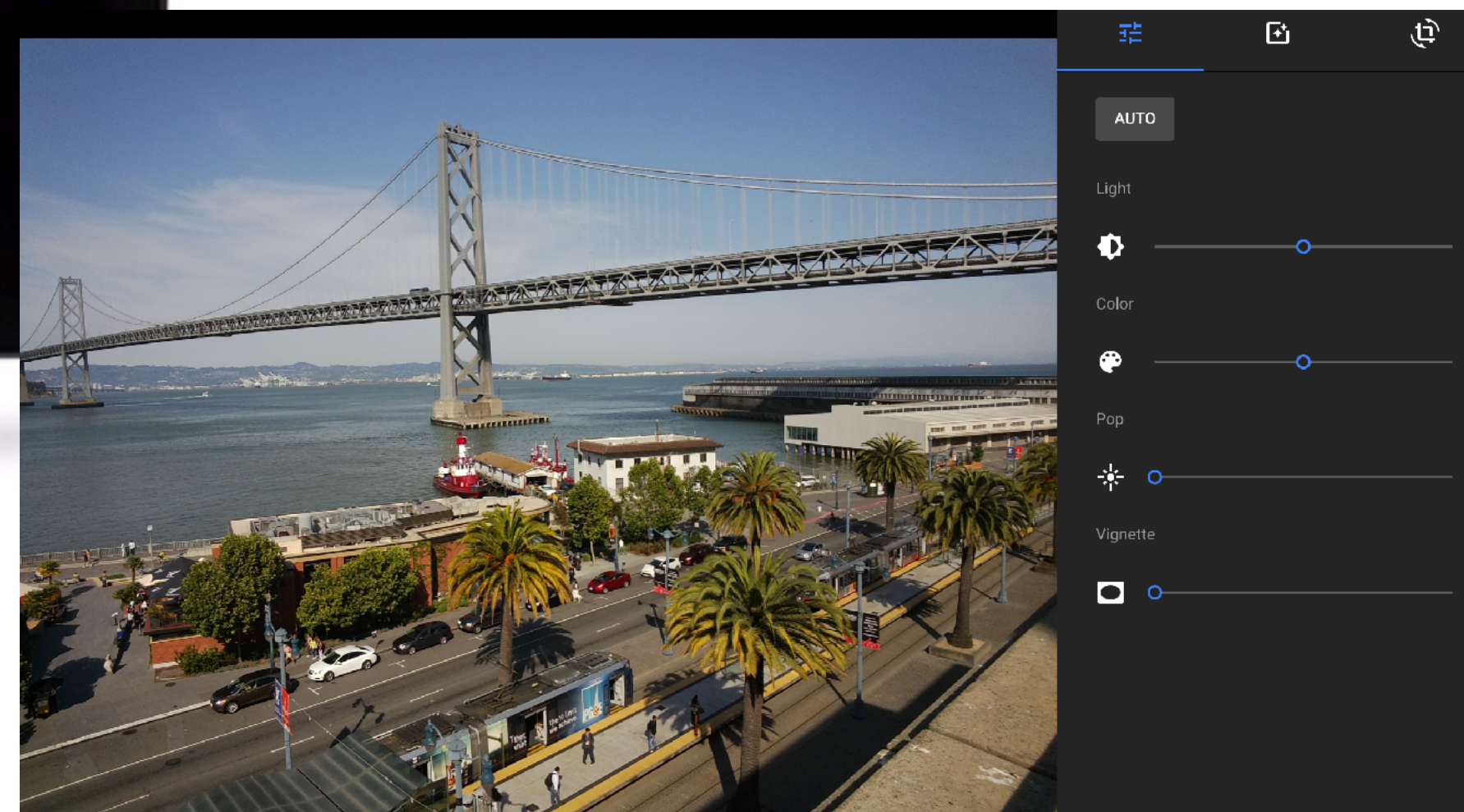


- **Bilateral filter**
(Common image filtering operation used in many applications)
 - **Original 122 lines of C++**
 - **Halide: 34 lines algorithm + 6 lines schedule**
 - **CPU implementation: 5.9x faster**
 - **GPU implementation: 2x faster than hand-written CUDA**



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.



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 [optional reading: Mullapudi 2016]**

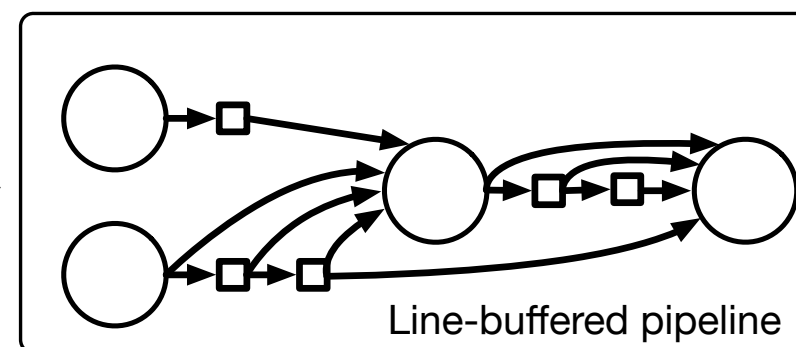
Darkroom/Rigel

[Hegarty 2014, Hegarty 2016]

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

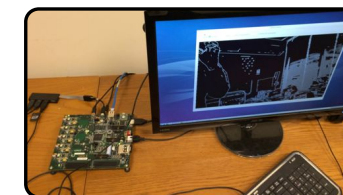
```
bx = im(x,y)
  (I(x-1,y) +
   I(x,y) +
   I(x+1,y))/3
end
by = im(x,y)
  (bx(x,y-1) +
   bx(x,y) +
   bx(x,y+1))/3
end
sharpened = im(x,y)
  I(x,y) + 0.1*
  (I(x,y) - by(x,y))
end
Stencil Language
```

Darkroom

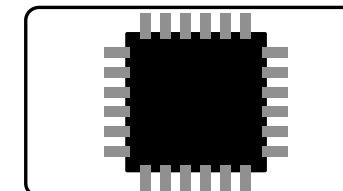


Darkroom

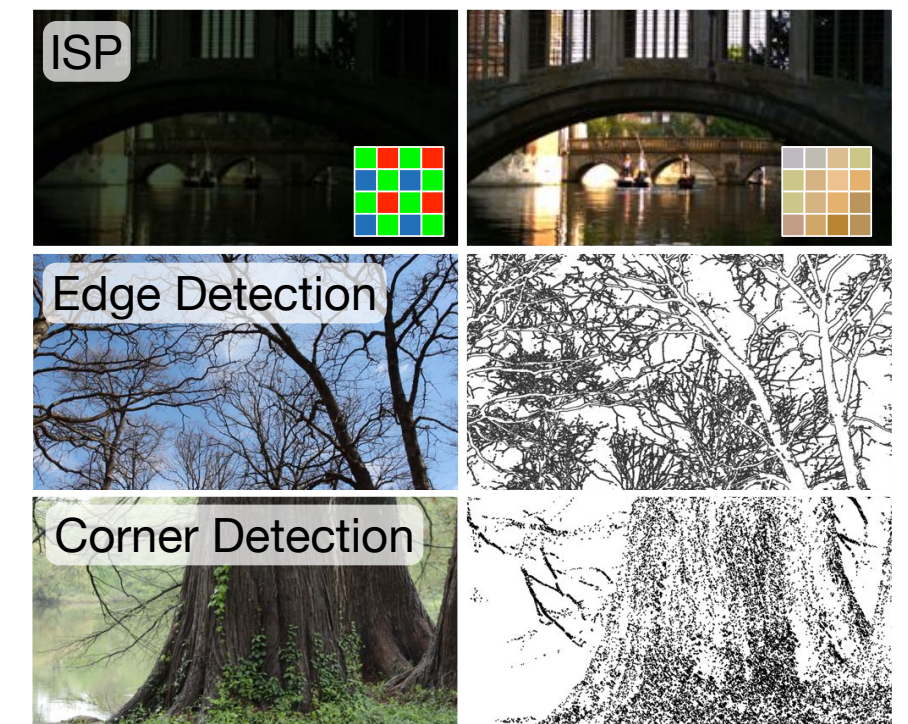
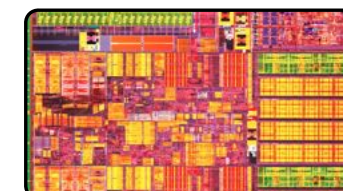
FPGA



ASIC



CPU



Seeking 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”)



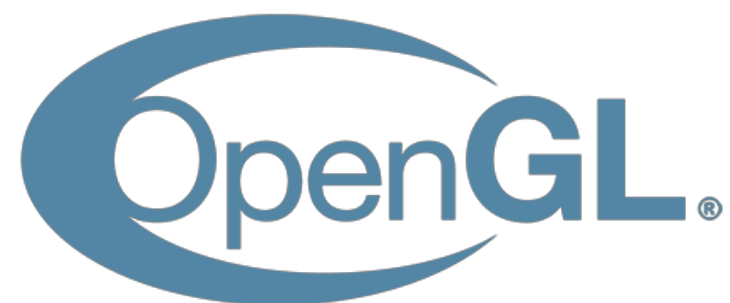
DSL for graph-based machine learning computations
Also see Ligra
(DSLs for describing operations on graphs)



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]

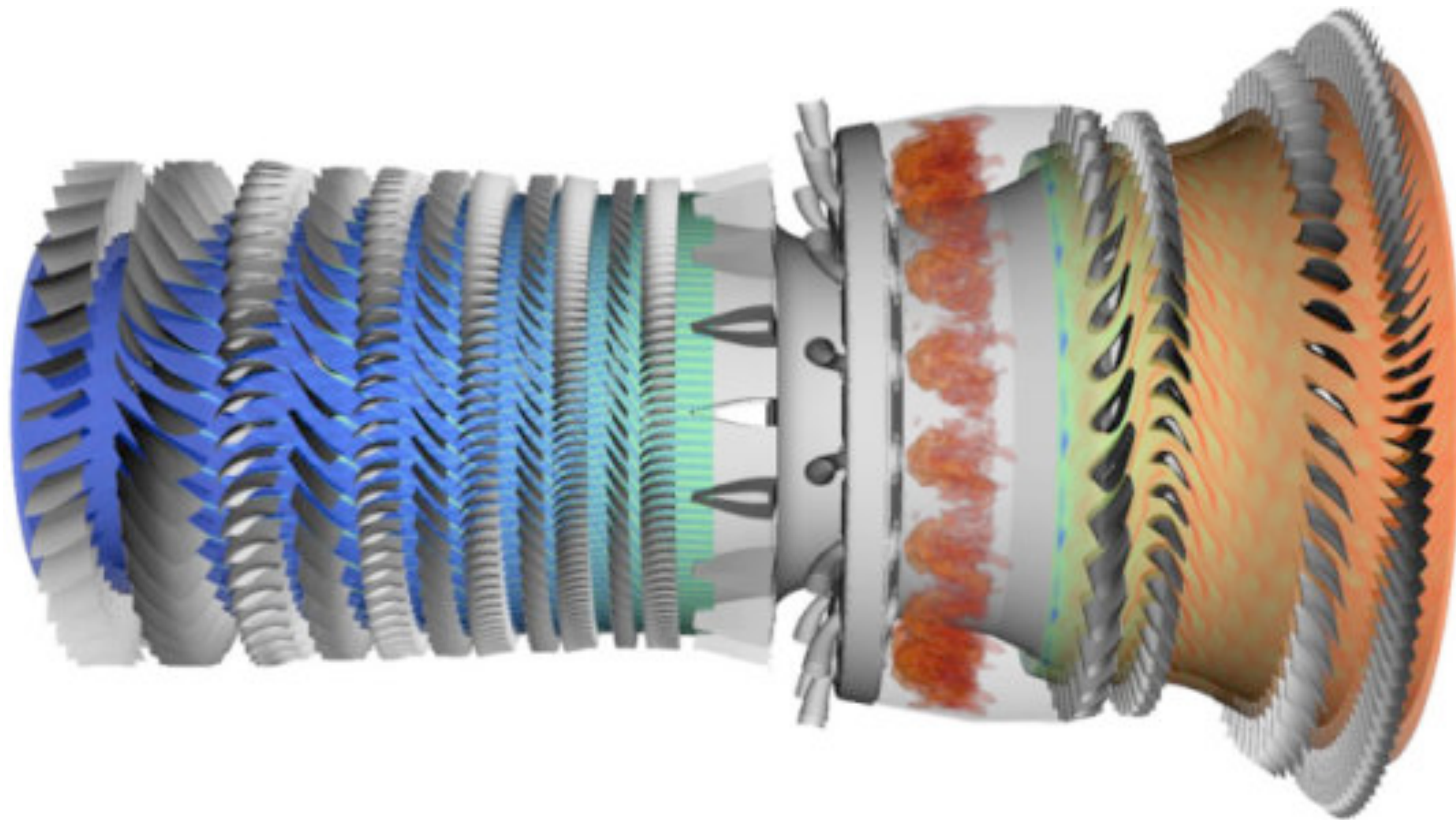
Summary

- **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: (only if time in class)

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://lizst.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)
```

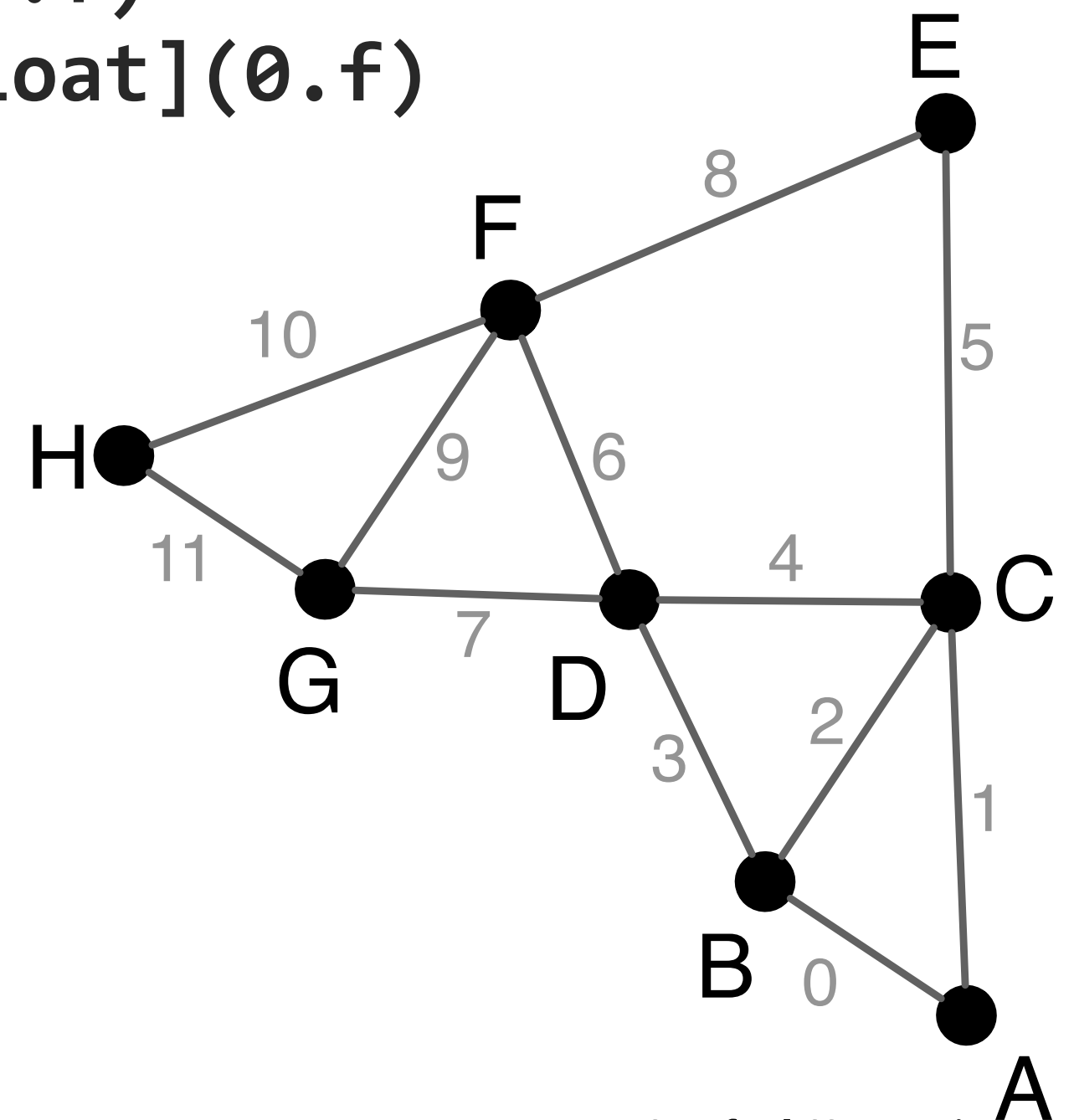
Color key:

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;

Color key:

Fields

Mesh

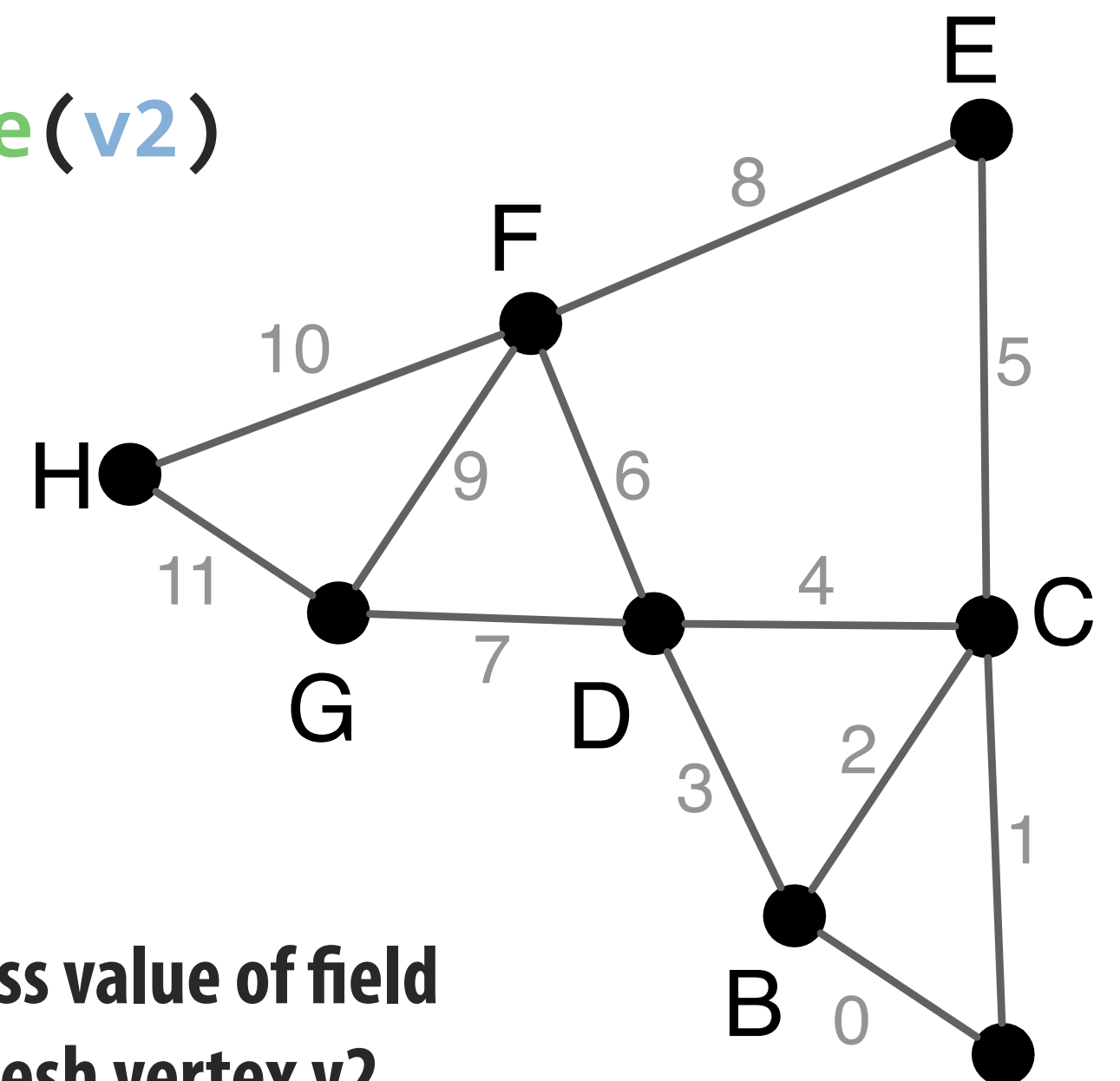
Topology functions

Iteration over set

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



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

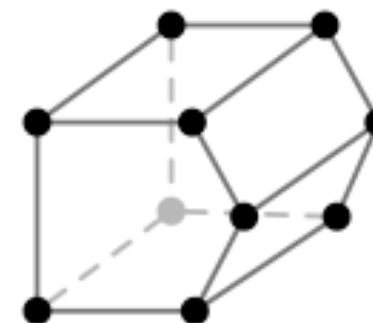
Notice how many operators return sets (e.g., “all edges of this face”)



```
BoundarySet1[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]
```



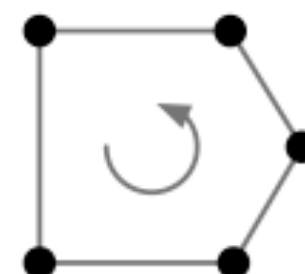
```
vertices(e : Vertex) : Set[Vertex]  
cells(e : Vertex) : Set[Cell]  
edges(e : Vertex) : Set[Edge]  
faces(e : Vertex) : Set[Face]
```



```
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]  
facesCCW2(e : Edge) : Set[Face]  
cells(e : Edge) : Set[Cell]  
head(e : Edge) : Vertex  
tail(e : Edge) : Vertex  
flip4(e : Edge) : Edge  
towards5(e : Edge, t : Vertex) : Edge
```



```
cells(e : Face) : Set[Cell]  
edgesCCW2(e : Face) : Set[Edge]  
vertices(e : Face) : Set[Vertex]  
inside3(e : Face) : Cell  
outside3(e : Face) : Cell  
flip4(e : Face) : Face  
towards5(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

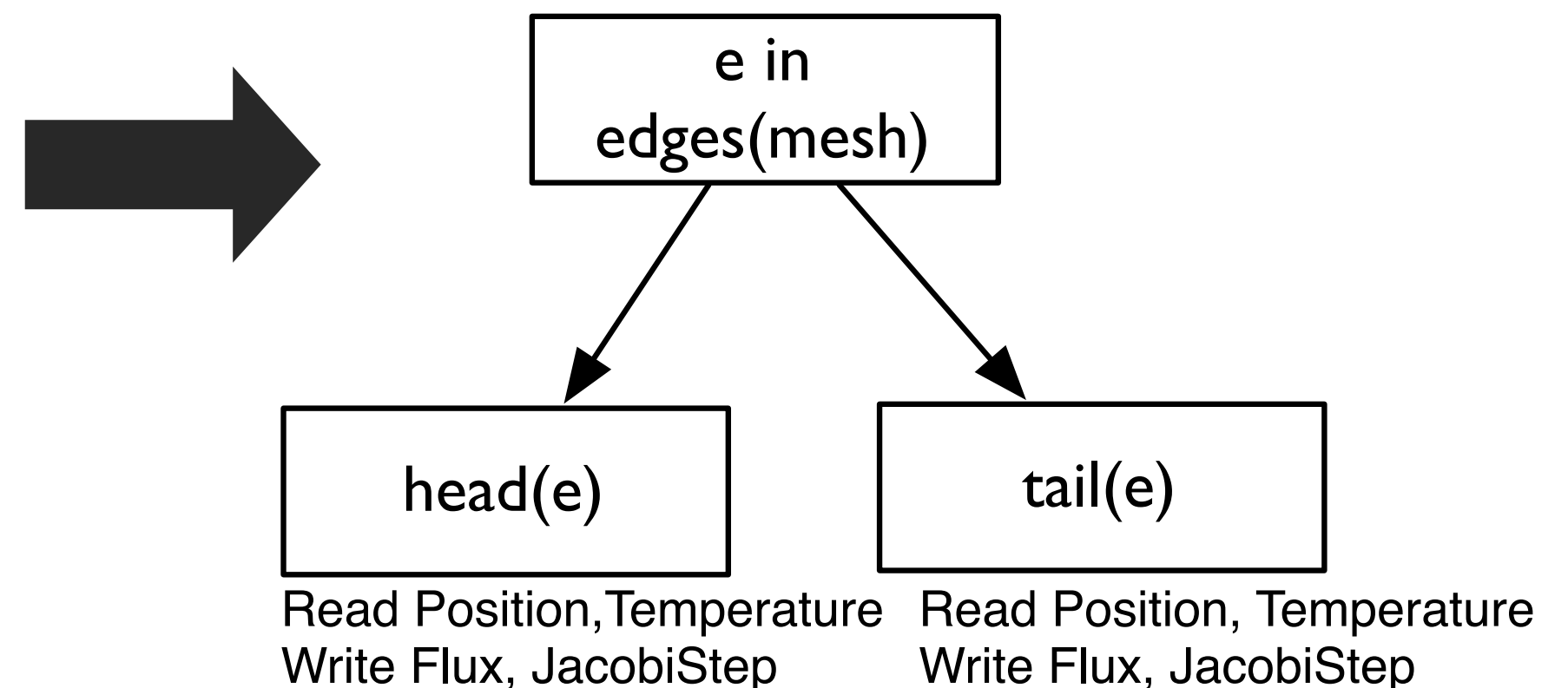
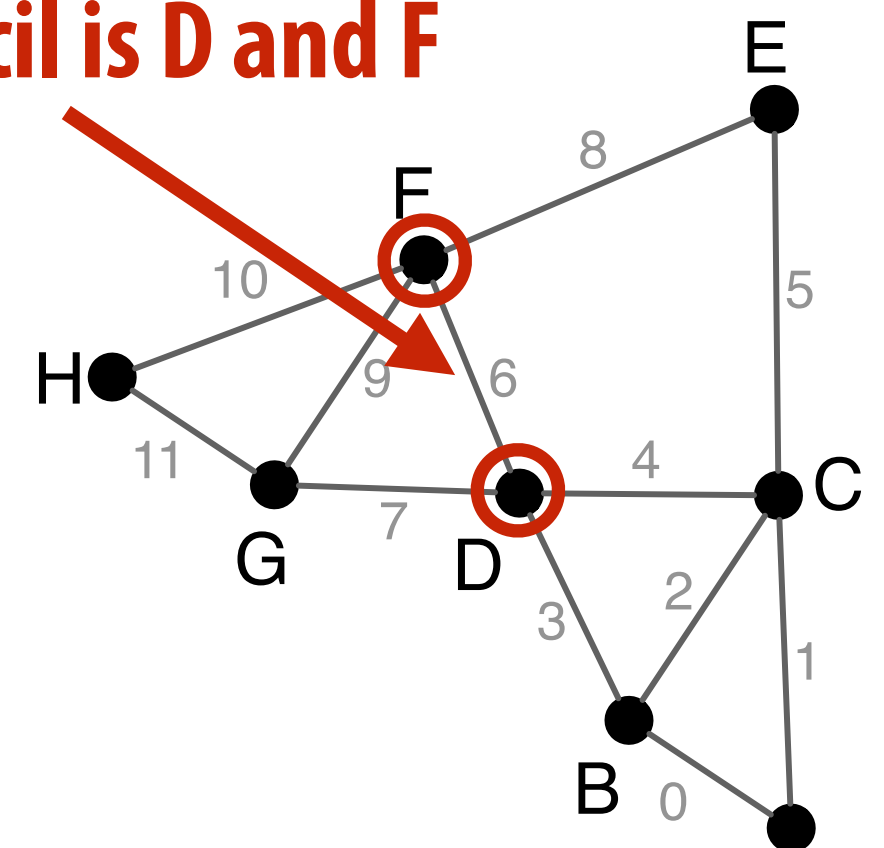
Lizst infers “stencils”: “stencil” = mesh elements accessed in an iteration of loop
= dependencies for the iteration

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

- Extract nested mesh element reads
- Extract field operations

```
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



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

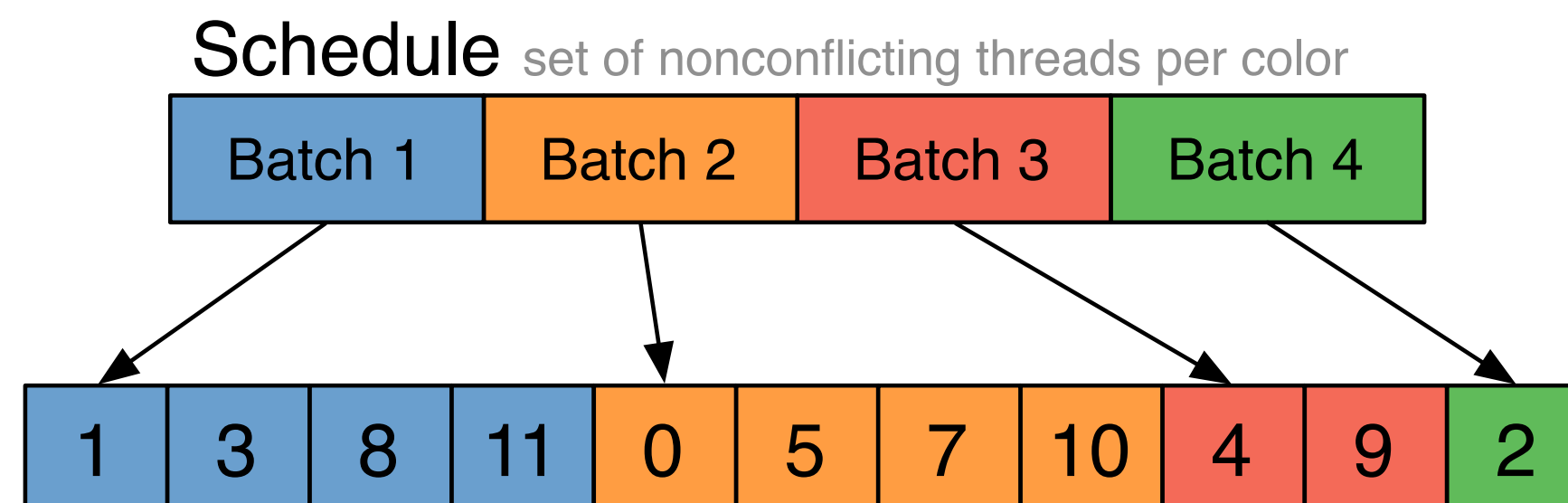
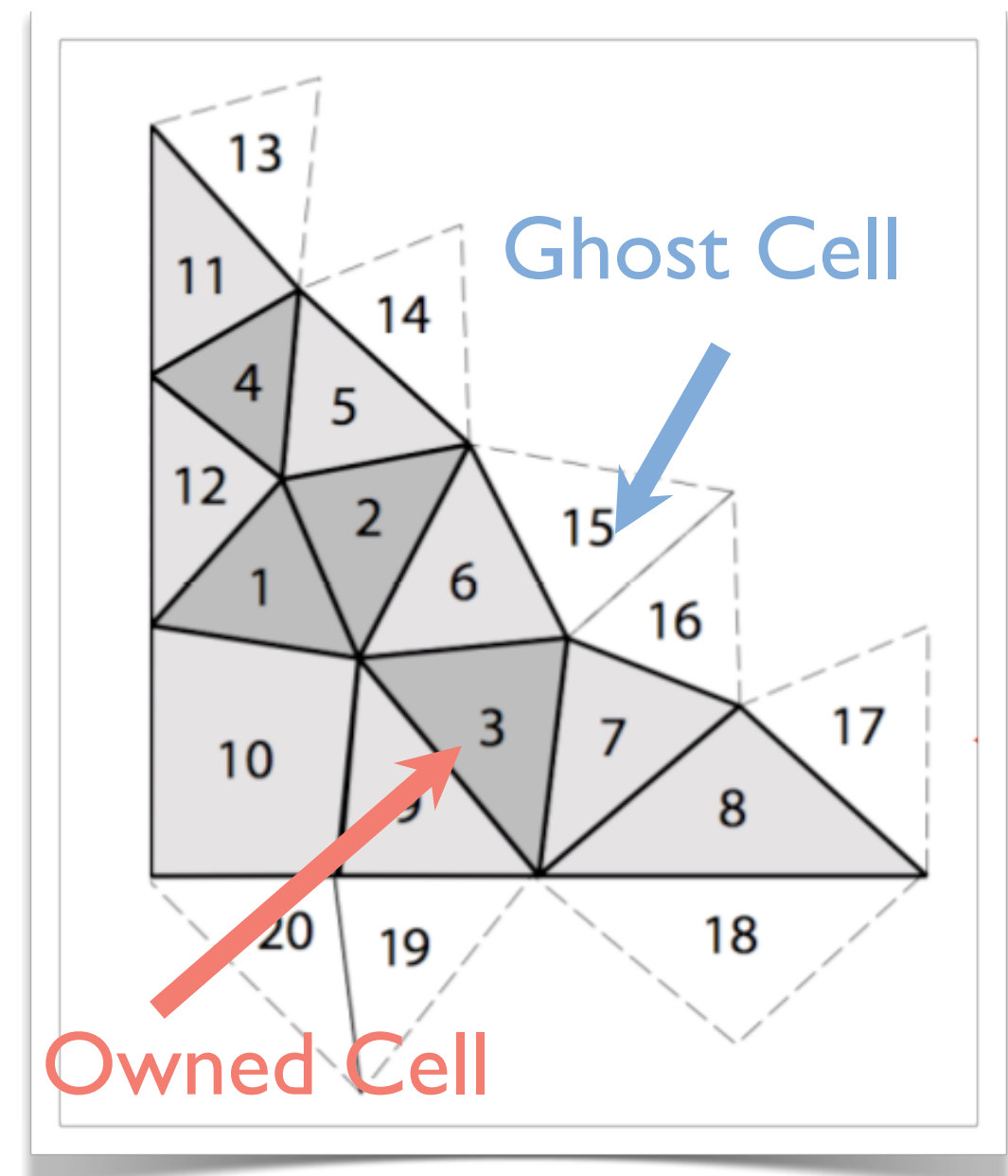
Restrictions allow compiler to automatically infer stencil for a loop iteration

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



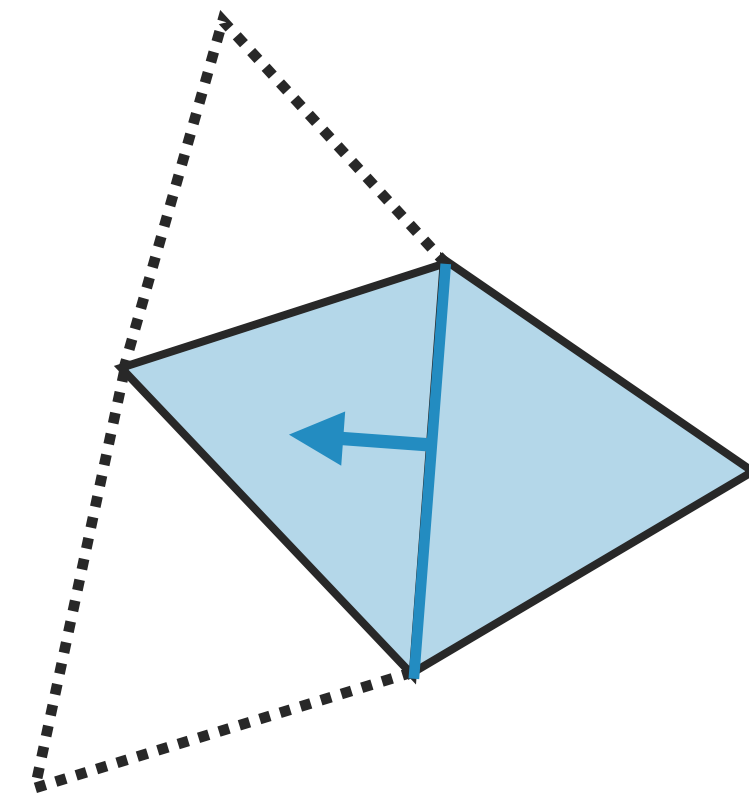
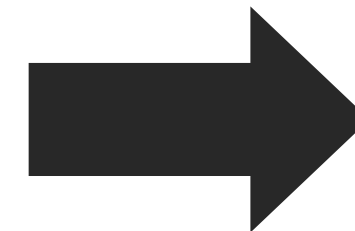
**Imagine compiling a Liszt program to a cluster
(multiple nodes, distributed address space)**

How might Liszt distribute a graph across these nodes?

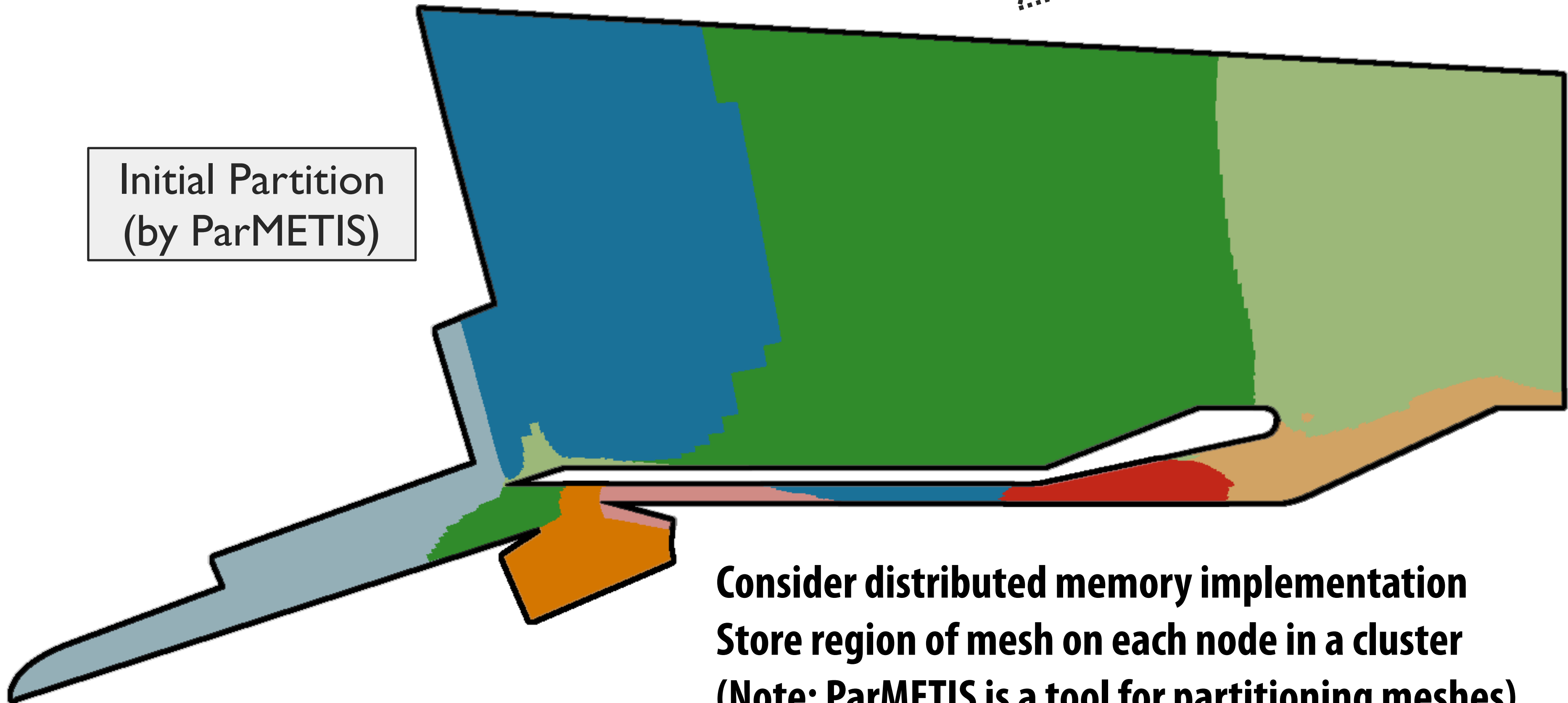
Distributed memory implementation of Liszt

Mesh + Stencil \rightarrow Graph \rightarrow Partition

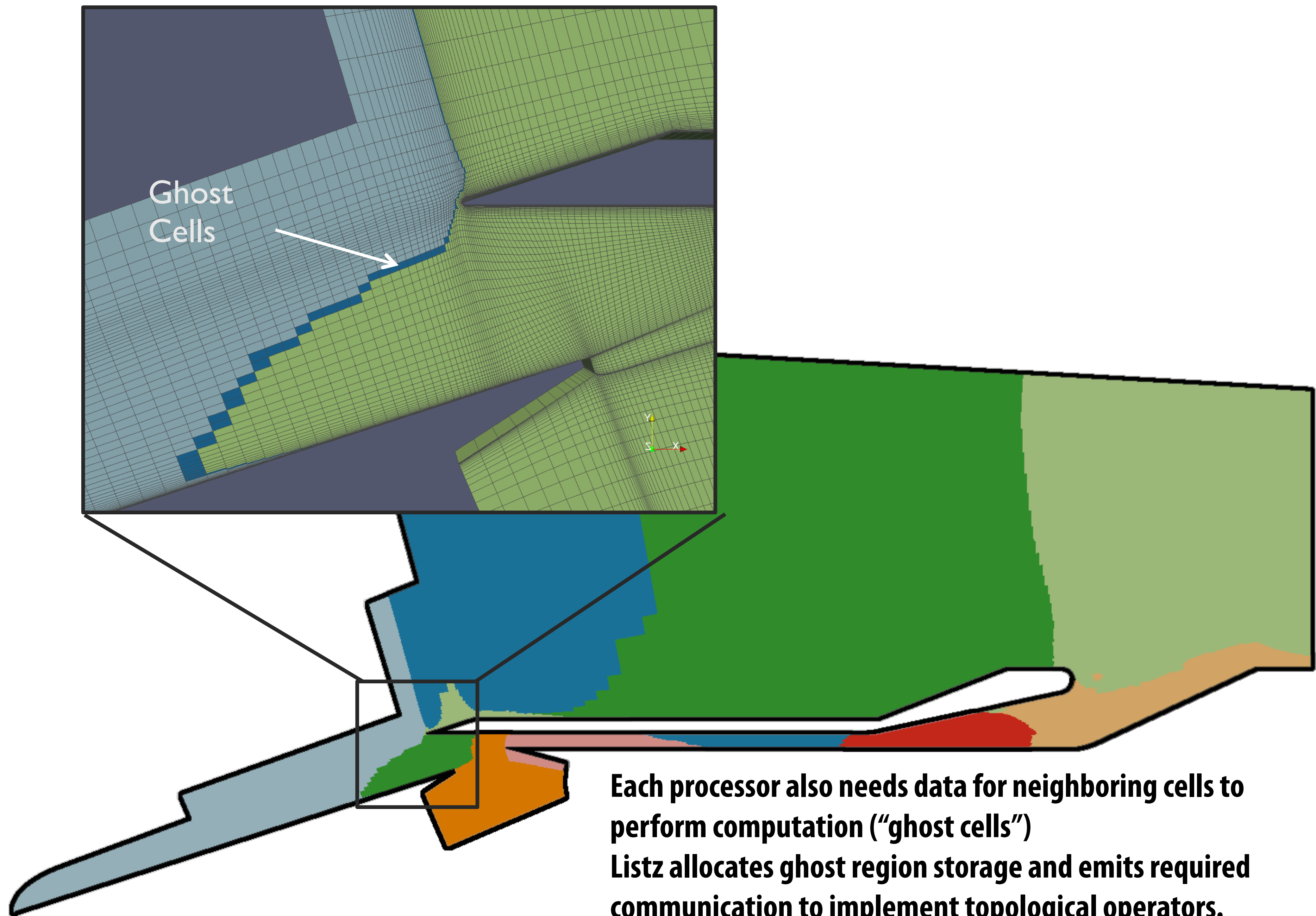
```
for(f <- faces(mesh)) {  
  rhoOutside(f) =  
    calc_flux(f, rho(outside(f))) +  
    calc_flux(f, rho(inside(f)))  
}
```



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)



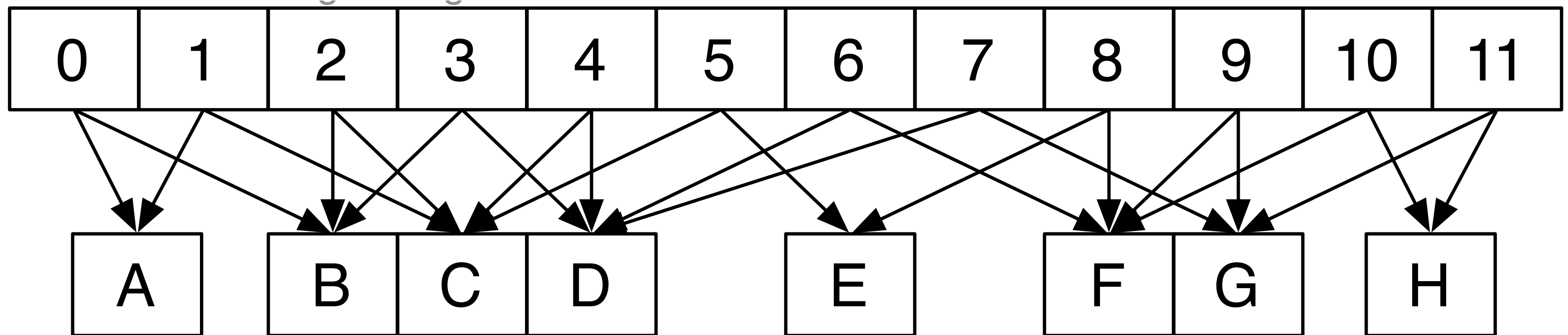
Each processor also needs data for neighboring cells to perform computation ("ghost cells")
Listz allocates ghost region storage and emits required communication to implement topological operators.

**Imagine compiling a Lisp 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)



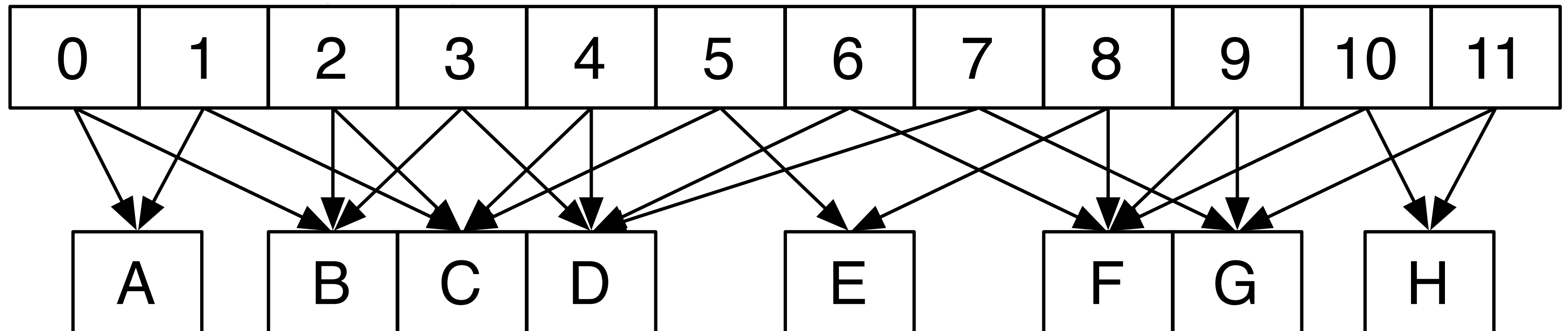
Flux field values (stored per vertex)

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

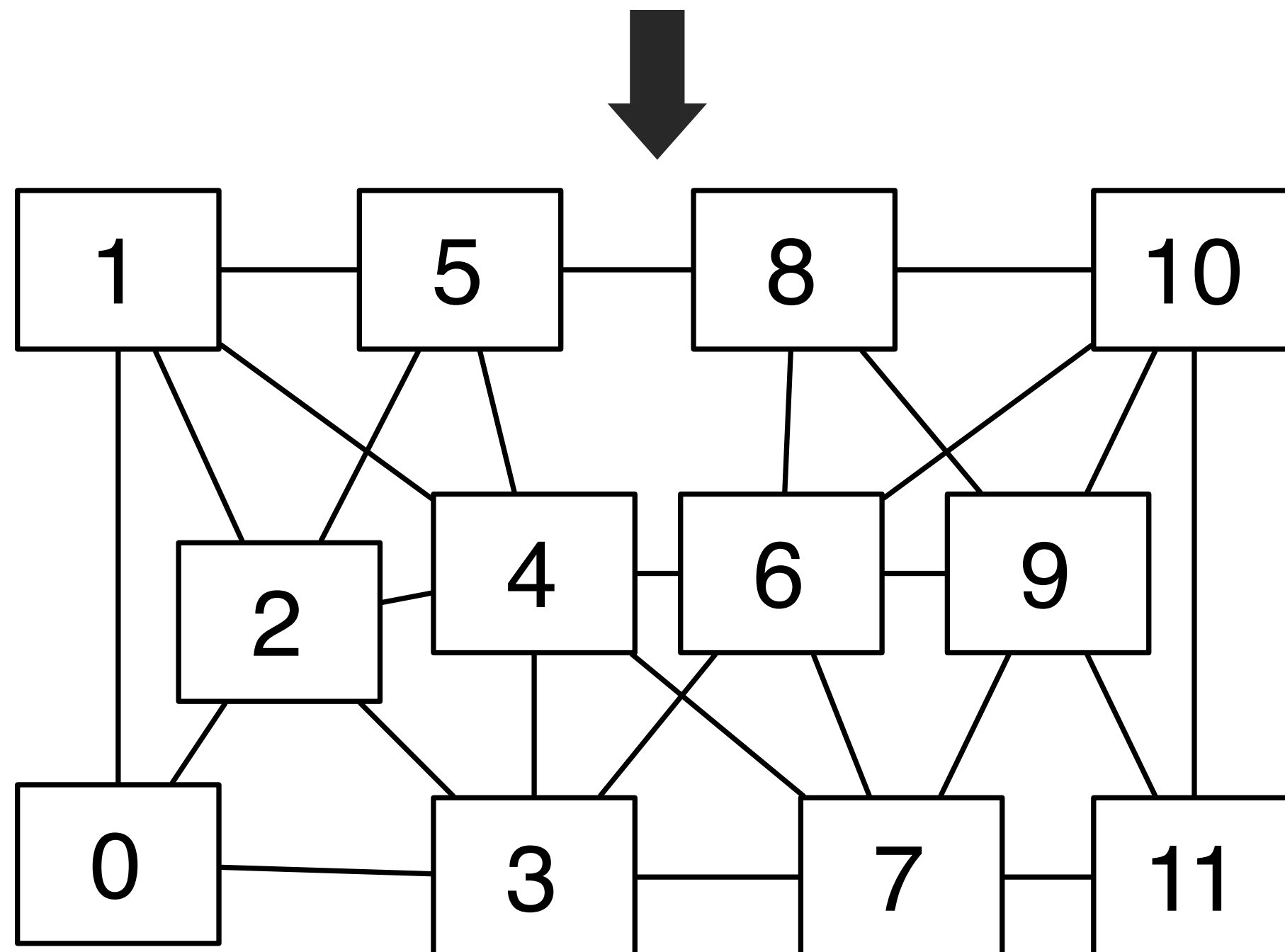
← Different edges share a vertex: requires atomic update of per-vertex field data

GPU implementation: conflict graph

Edges (each edge assigned to 1 CUDA thread)



Flux field values (per vertex)

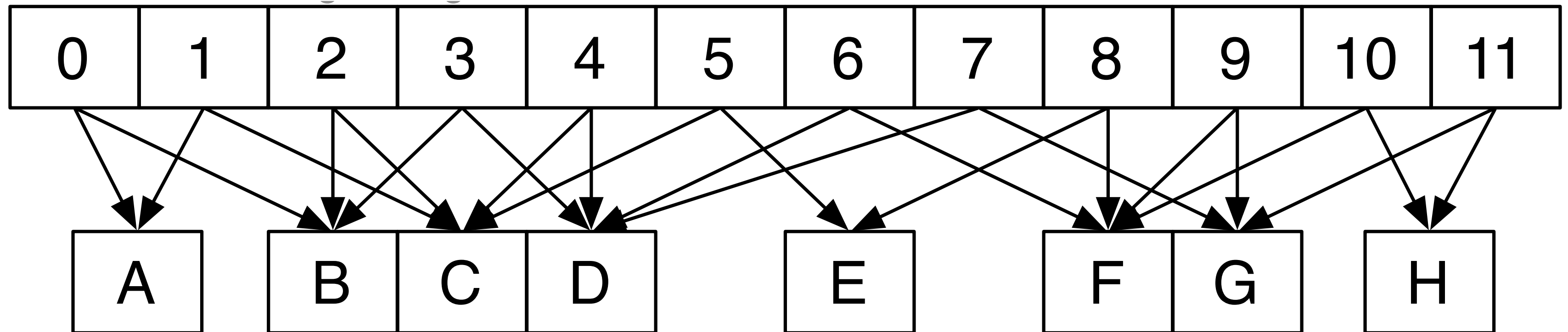


Identify mesh edges with colliding writes
(lines in graph indicate presence of collision)

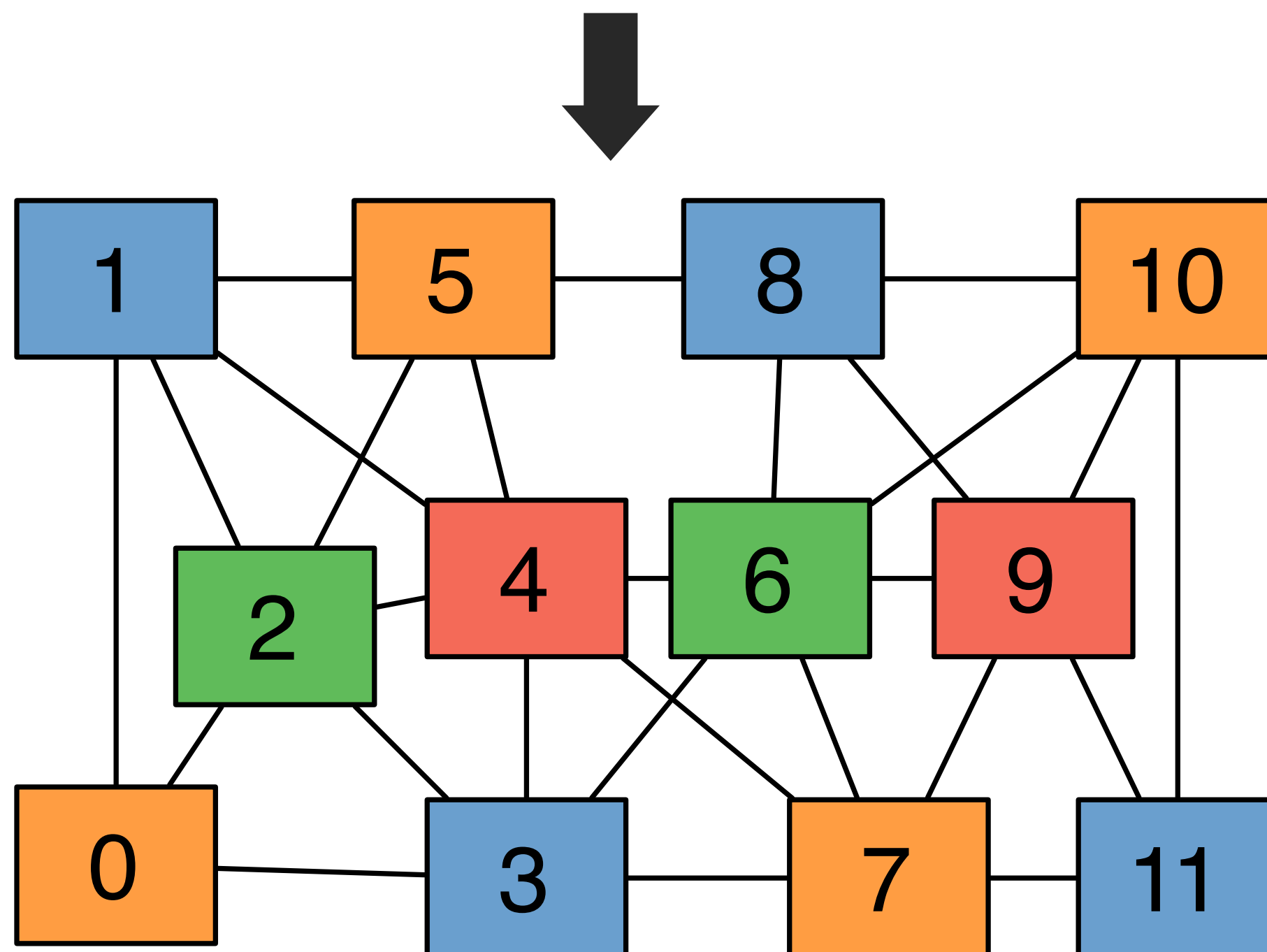
Can simply run program once to get this
information.
(results remain valid for subsequent
executions provided mesh does not change)

GPU implementation: conflict graph

Threads (each edge assigned to 1 CUDA thread)



Flux field values (per vertex)

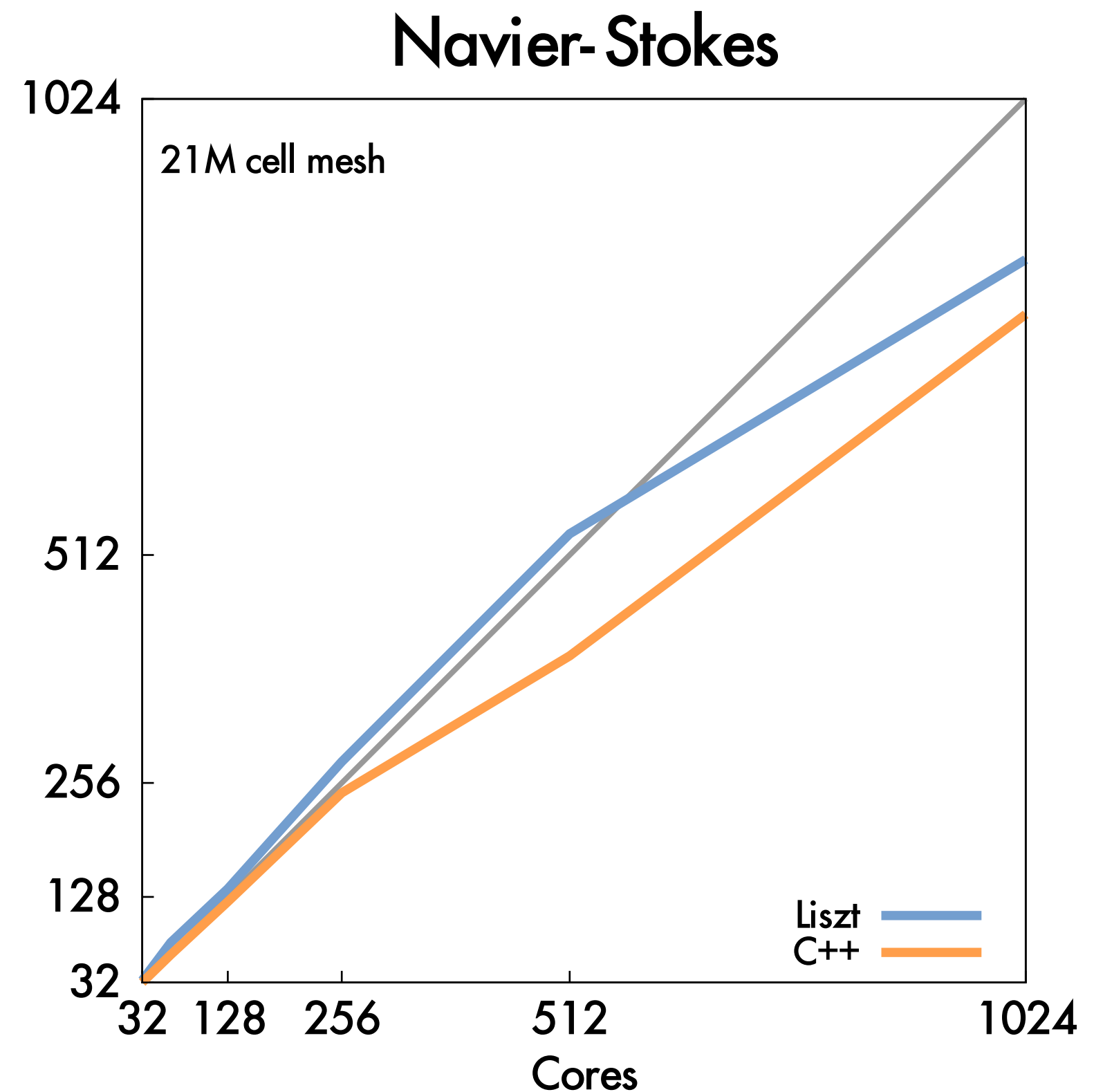
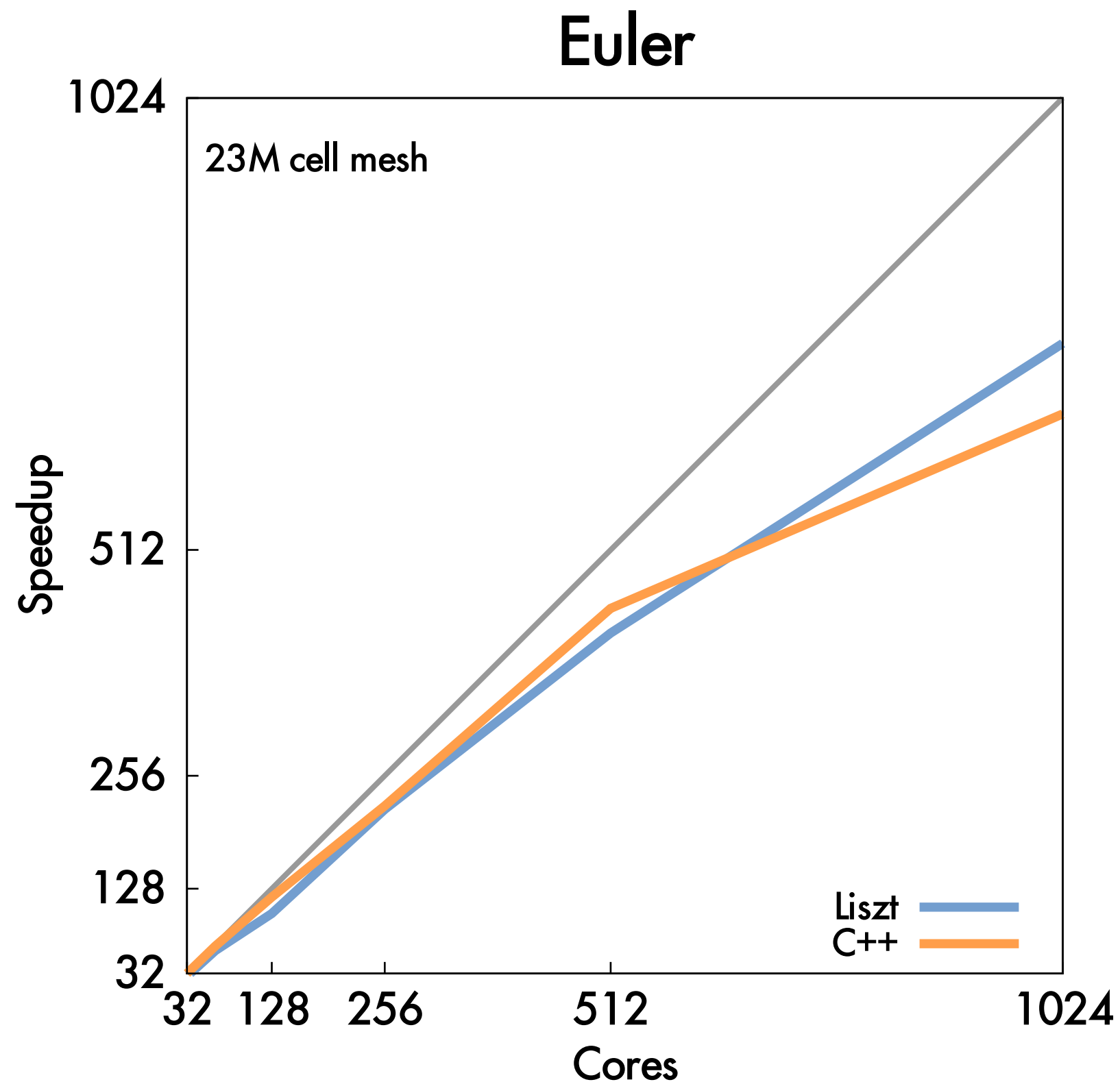


“Color” nodes in graph such that no connected nodes have the same color

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 Lizst program also runs with high efficiency on GPU (results not shown)

But uses a different algorithm 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)**