Lecture 13:

Domain-Specific Programming Systems and Automatic Performance Optimization

Parallel Computing

Stanford CS149, Fall 2025

Today

- Mechanisms and techniques that increase the productivity of performance optimization —> both by making expert human programmers more productive and via automation

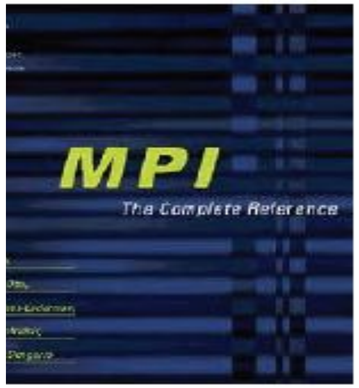
- Key idea 1: raise level of abstraction

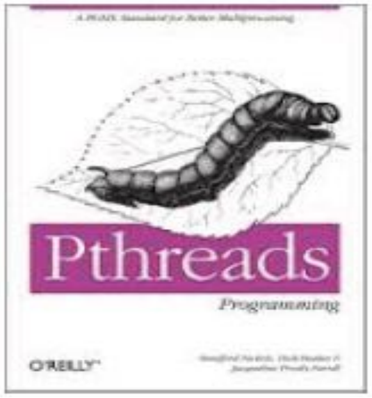
- Key idea 2: intelligent search

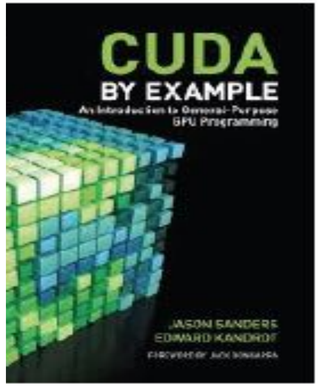
- [Emerging] idea 3: leverage problem solving ability and code generation capabilities of modern LLMs

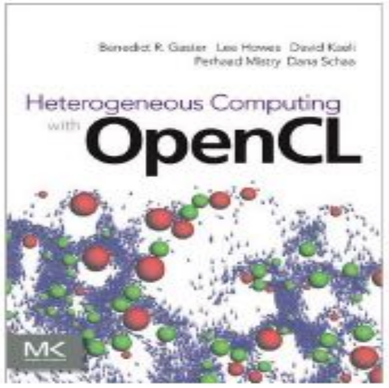
CS149 educated programmers = hard to find

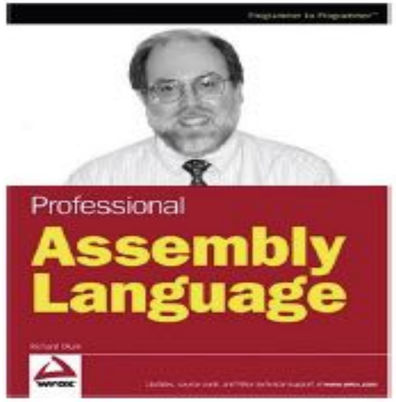
Performance optimization in languages like C++, ISPC, CUDA = low productivity
(Proof by assignments 1, 2, 3, 4, etc…)



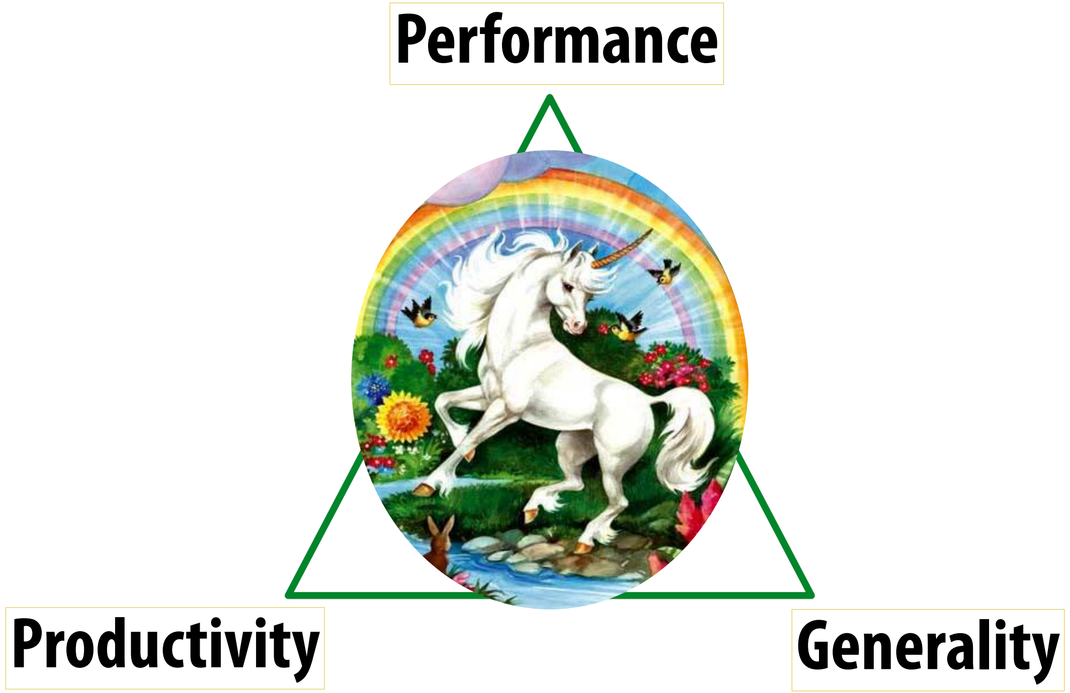






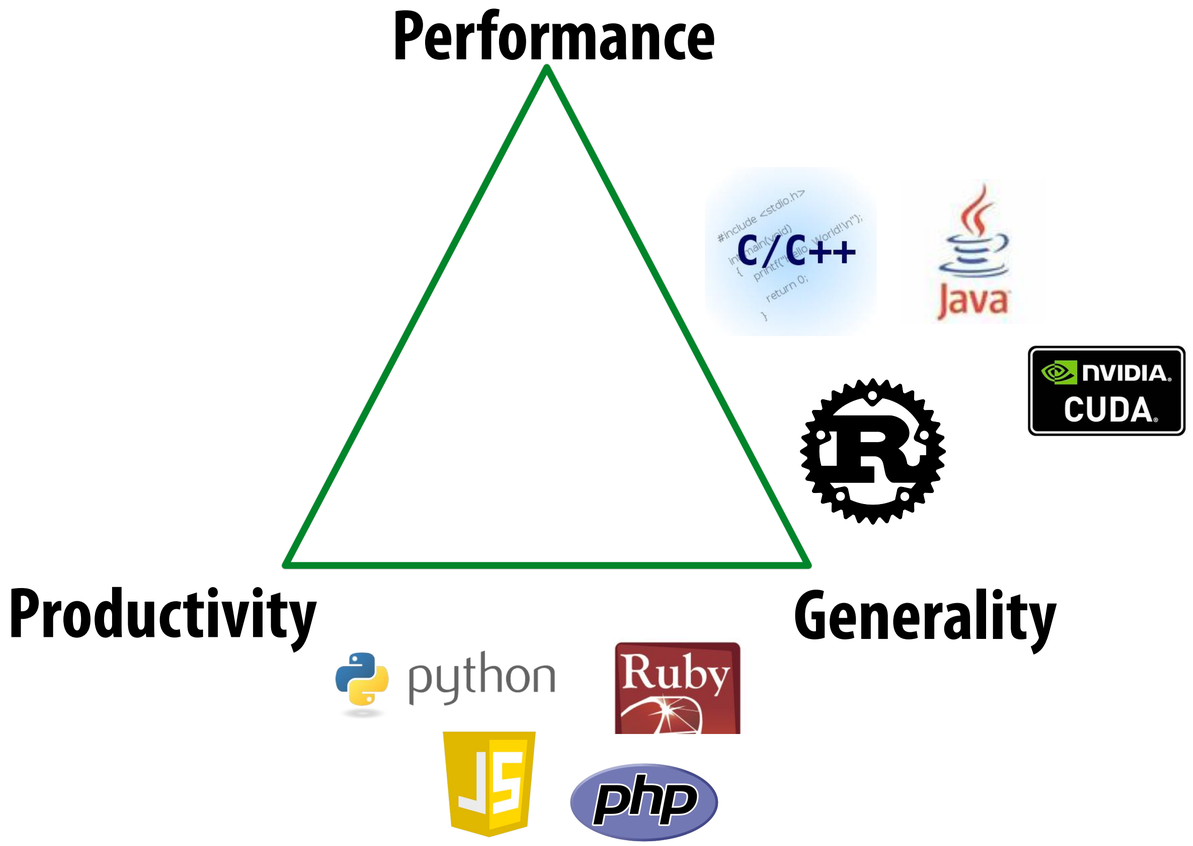


The ideal parallel programming language



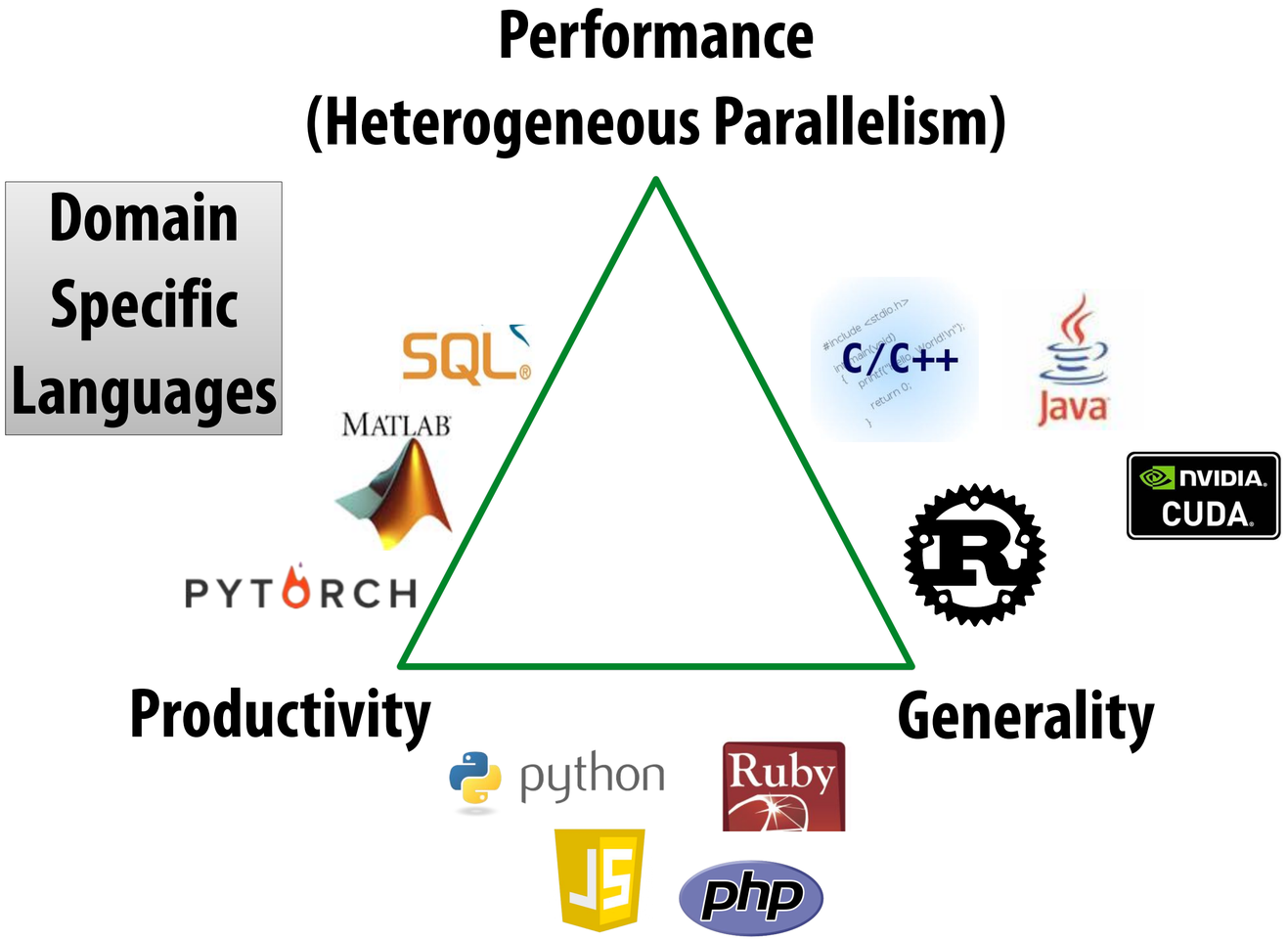
Credit: Pat Hanrahan for this slide design

Popular languages (not exhaustive ;-))



Credit: Pat Hanrahan for this slide design

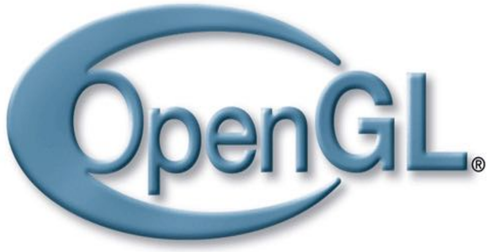
Way forward ⇒ domain-specific languages



Credit: Pat Hanrahan for this slide design

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: quickly write a high-performance program for a target machine
  - 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:

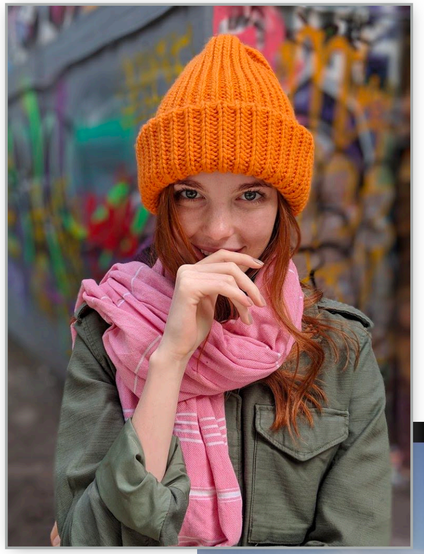
Halide: a domain-specific language for image processing

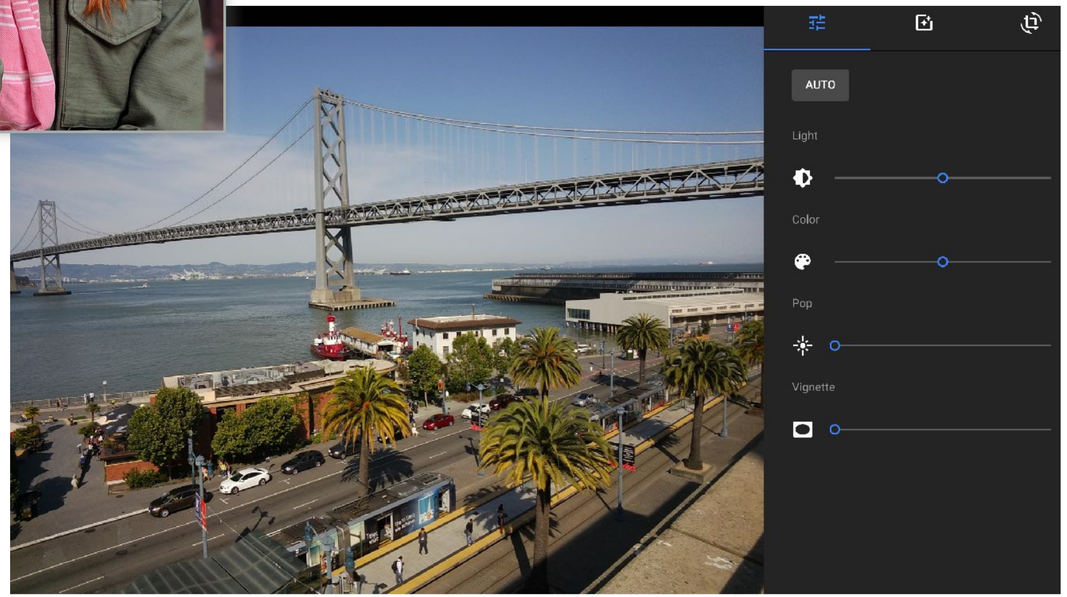
Jonathan Ragan-Kelley, Andrew Adams et al.

[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

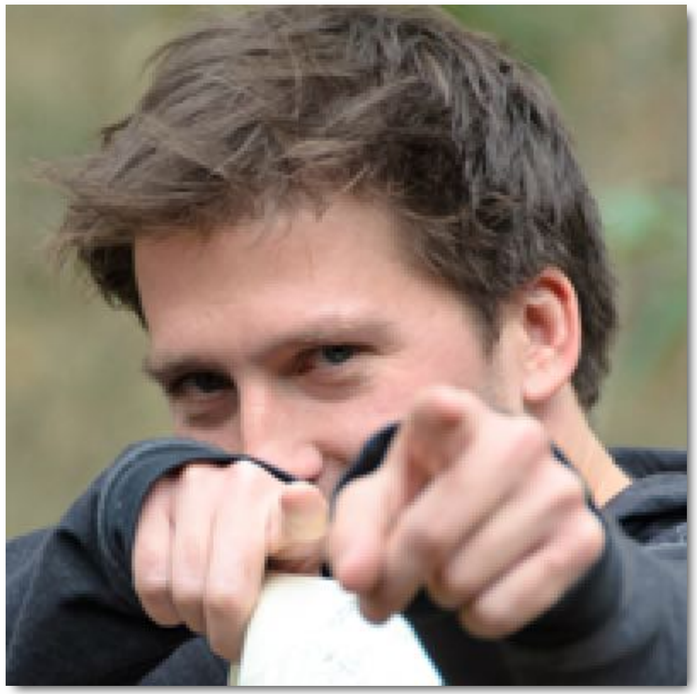
Bad: specific to SSE (not AVX2), CPU-code only, hard to tell what is going on at all!

```
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

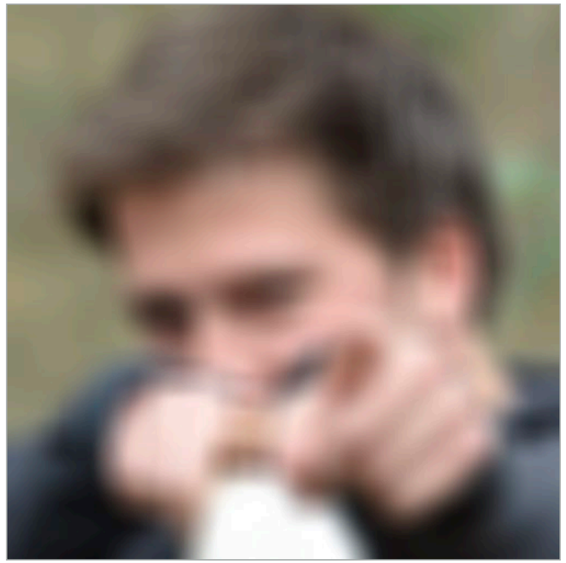
For NxN filter: N^2 x WIDTH x HEIGHT

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

```
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;
  }
}
```

1D horizontal blur

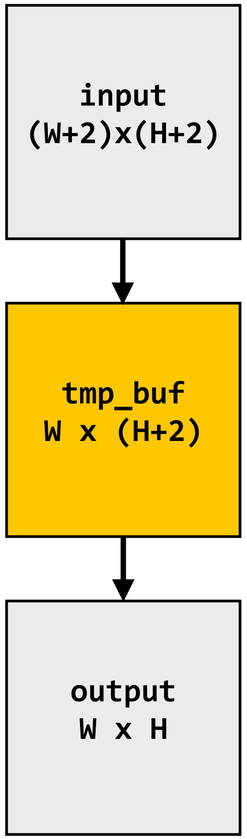
1D vertical blur

Total work per image = 6 x WIDTH x HEIGHT

For NxN filter: 2N x WIDTH x HEIGHT

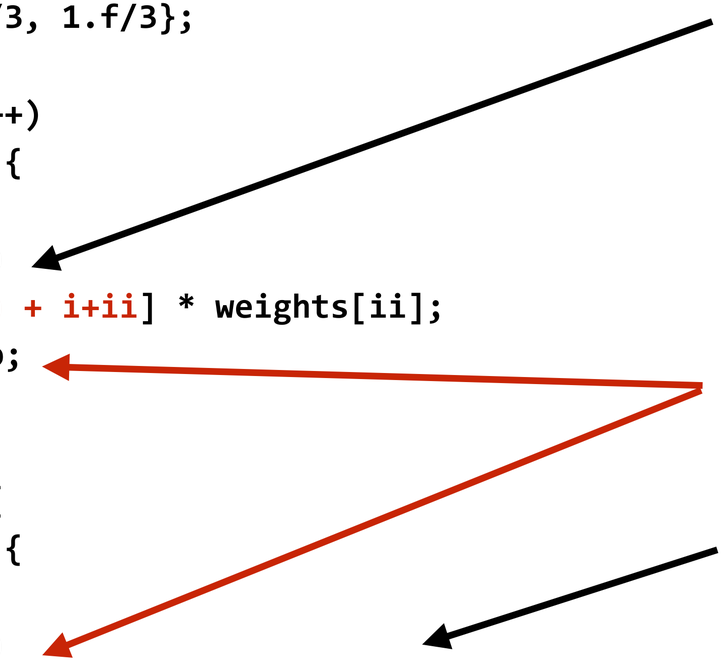
WIDTH x HEIGHT extra storage

2x lower arithmetic intensity than 2D blur. Why?



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)

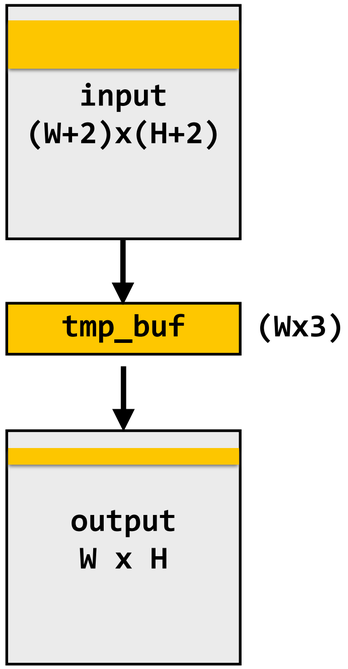
Two-pass image blur, "chunked" (version 1)

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * 3];
float output[WIDTH * HEIGHT];

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)

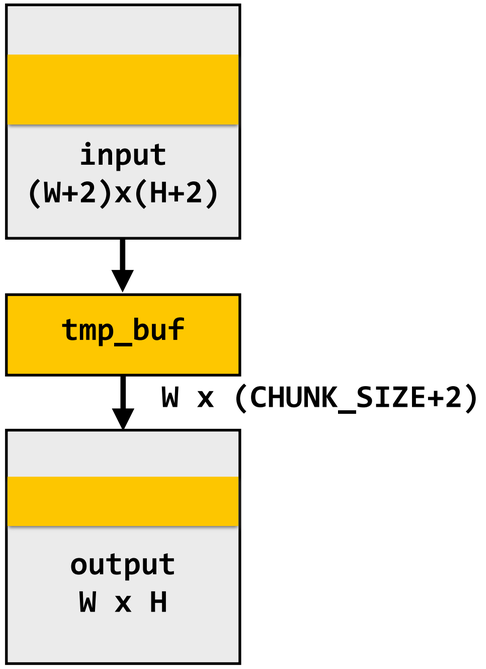
Two-pass image blur, "chunked" (version 2)

```
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)

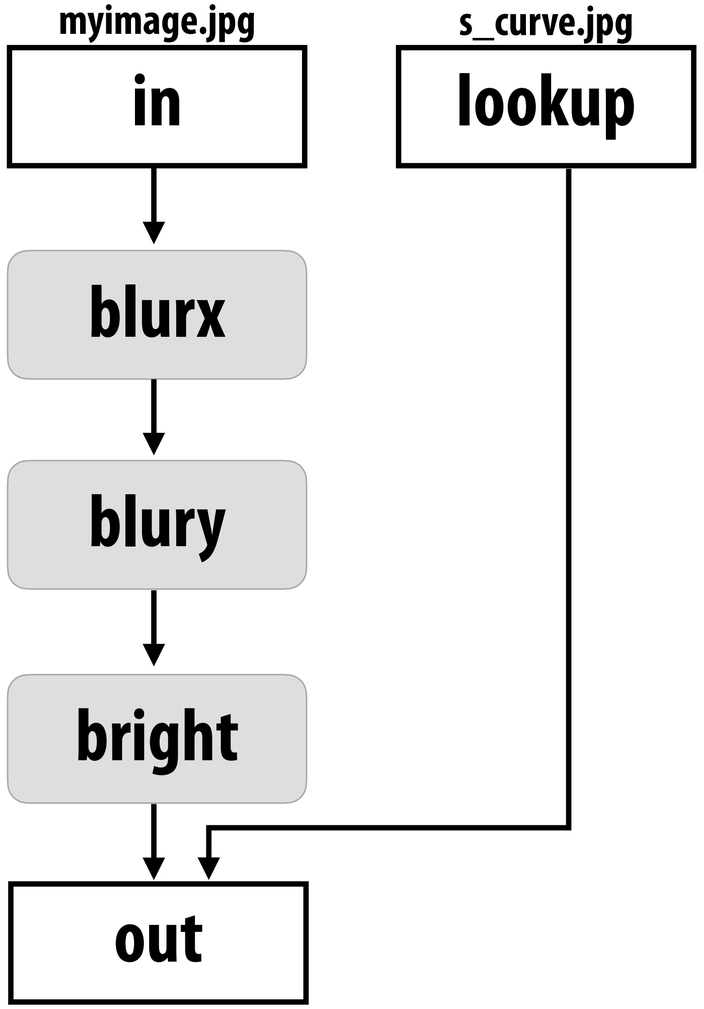
```
Var x, y;
Func blurx, blury, bright, out;
Halide::Buffer<uint8_t> in = load_image("myimage.jpg");
Halide::Buffer<uint8_t> lookup = load_image("s_curve.jpg");  // 255-pixel 1D image

// 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

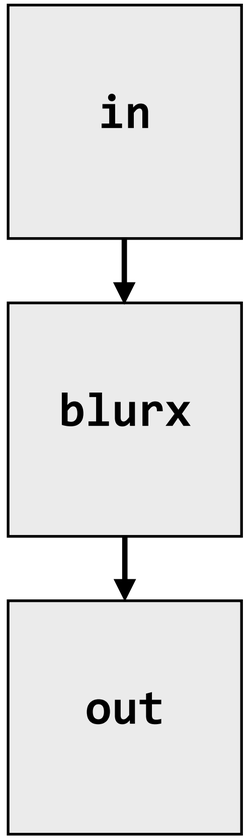
Halide expression: a side-effect free expression that describes how to compute a function's value at a point in its domain in terms of the values of other functions.

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

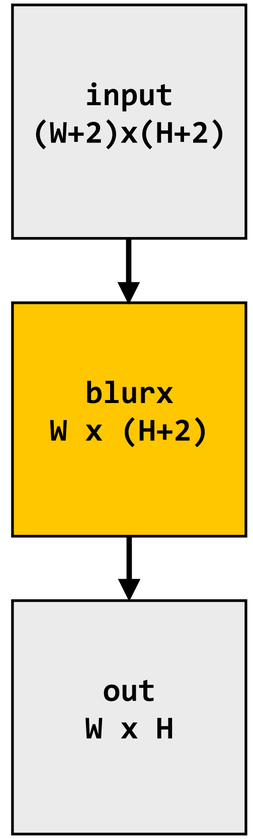
Real-world production applications may features hundreds to thousands of functions!
Google HDR+ pipeline: over 2000 Halide functions.

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);

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

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)

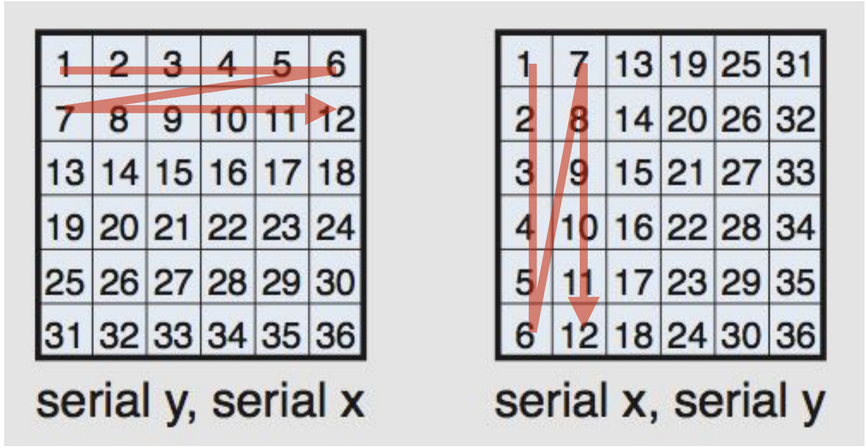
Use threads to parallelize the y loop

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

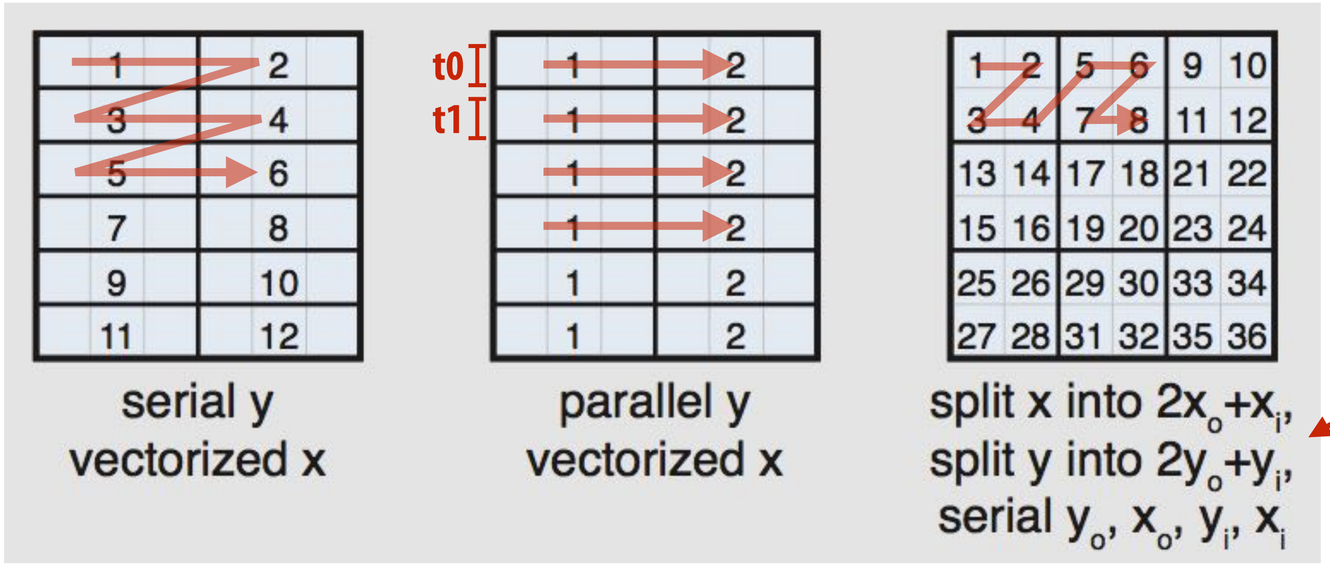
"Schedule"

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)

Ordering Halide loop nests

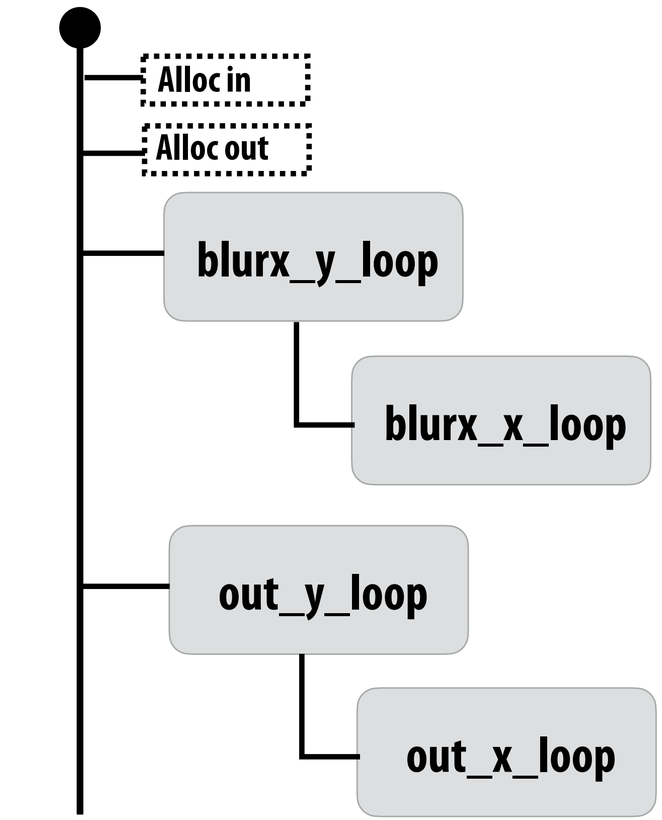
Halide algorithm:

```
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;
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

Halide schedule:

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

Loop nest diagram of implementation:



C-code equivalent:

```
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
```

Loops for computing values of blurx

Loops for computing values of out

Ordering Halide loop nests

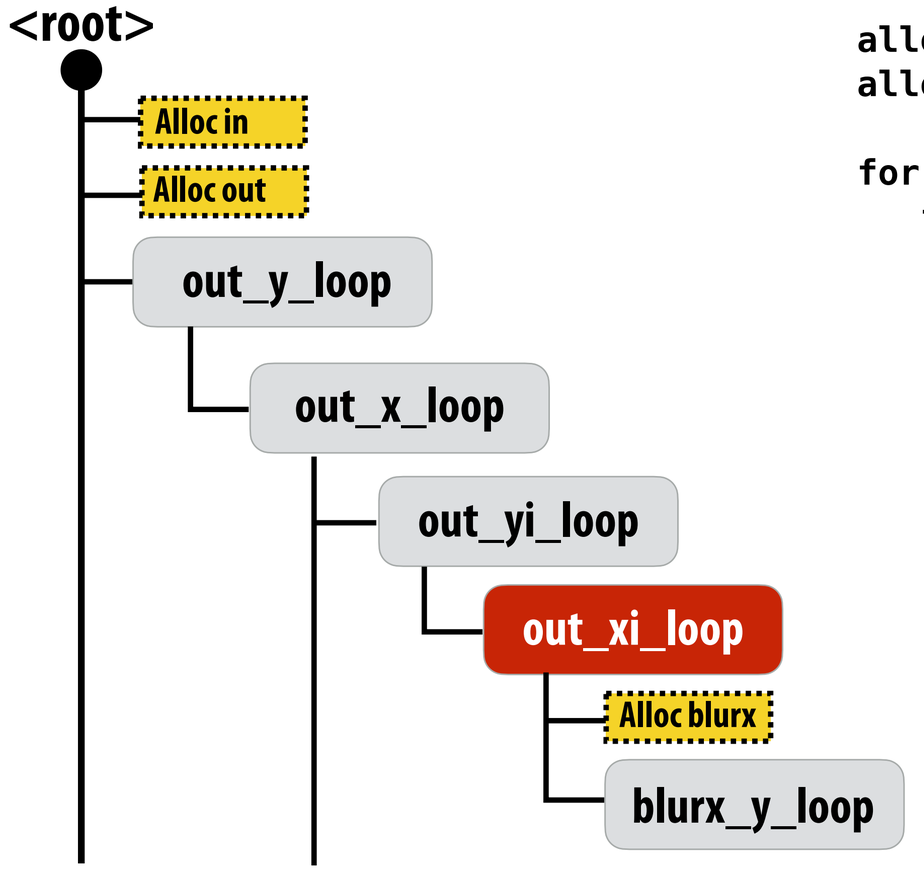
Halide algorithm:

```
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;
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

Halide schedule:

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

Loop nest diagram of implementation:



Another possible implementation:

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

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 blurx(1,3)

            // compute 3 elements of blurx needed for out(idx_x, idx_y) here
            for blurx_y=0 to 3:
                blurx(0, blurx_y) = … // compute blurx from in

            out(idx_x, idx_y) = … // compute out from blurx
```

Outer loops over tiles of out

Inner loops for computing values of out

Only allocate 3 elements of blurx

Ordering Halide loop nests

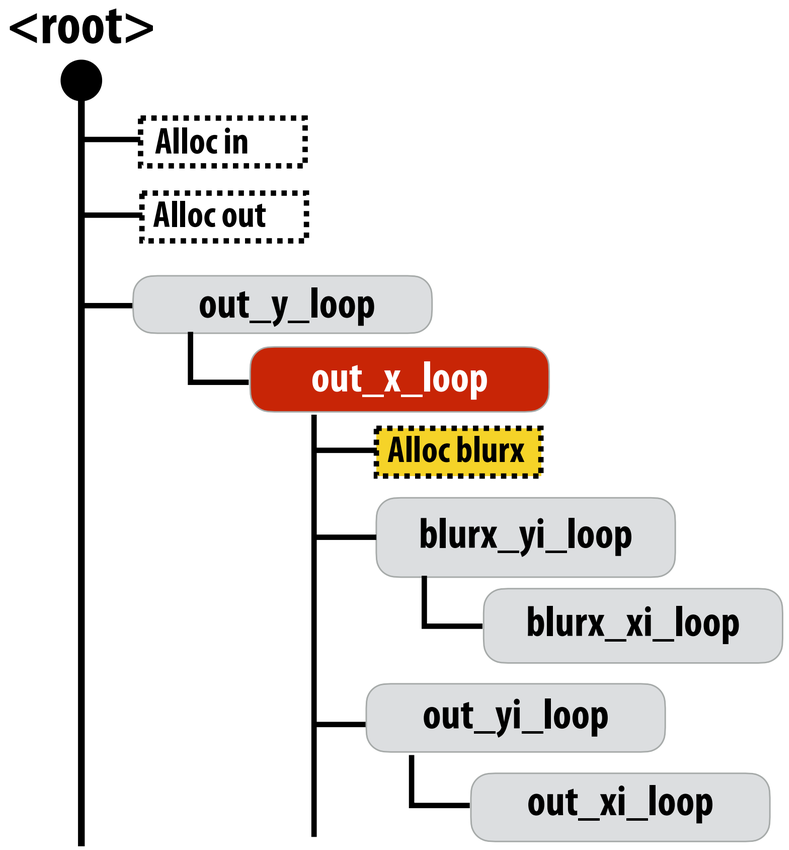
Halide algorithm:

```
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;
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

Halide schedule:

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

Loop nest diagram of implementation:



C-code equivalent:

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

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

      allocate blurx(256, 34)
      for yi=0 to 32+2:
         for xi=0 to 256:
            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) = // compute out from blurx
```

Outer loops over tiles of out

Only allocate a tile of blurx

Loops for computing values of blurx

Inner loops for computing values of out (loops over elements)

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;
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);

// “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:

```
allocate in(1024+2, 1024+2)
allocate out(1024, 1024)

for y=0 to num_tiles_y:   // iters of this loop are parallelized using threads
   for x=0 to num_tiles_x:
      allocate blur_x(258,34)  // buffer for tile blurx
      for yi=0 to 32+2:
         for xi=0 to 256+2 BY 8:
            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 second 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

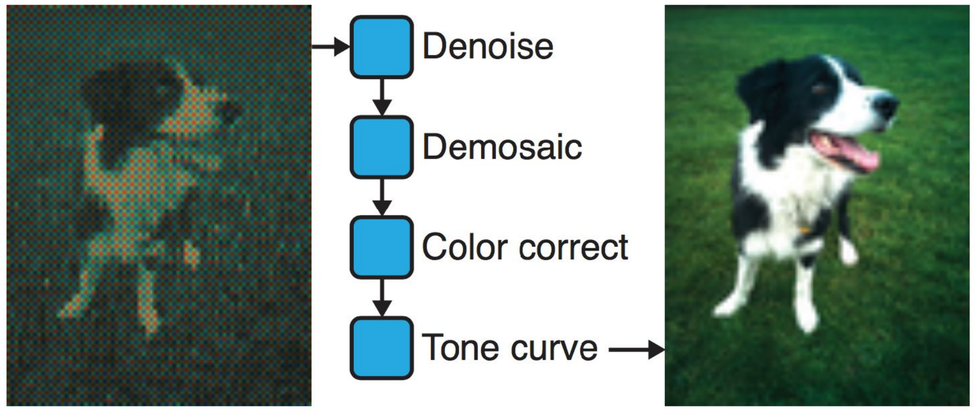
(to enable compiler to provide desired services)

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

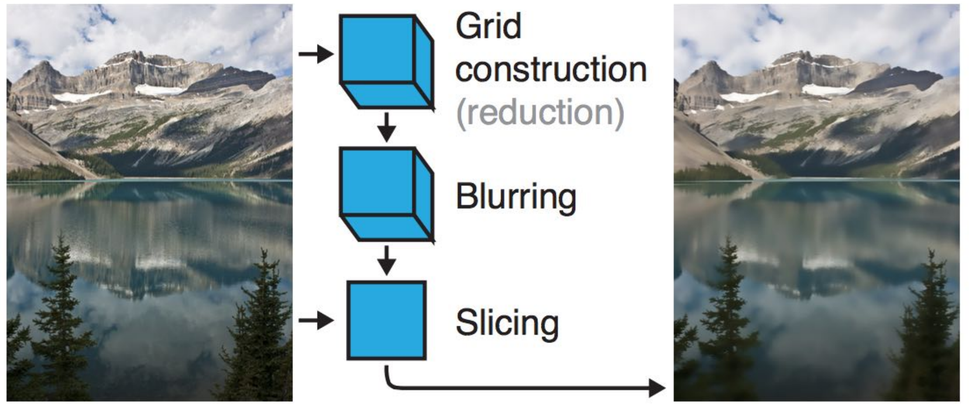
Initial academic Halide results

[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



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

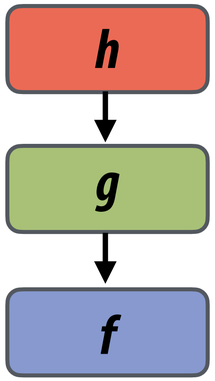
- **Solution: extend compiler to analyze Halide program to automatically generate efficient schedules for the programmer [Adams 2019]**

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

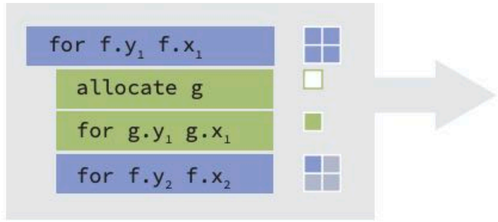
Modeling scheduling as a sequence of choices

- For each node *N* in the program DAG, starting from the end of the DAG…
  - Choose where to place current node *N* in the existing loop nest (determine N.compute_at())
  - Choose a tile sizes for *N* (assume outer dimension is parallel over threads, inner dimension is vectorized)
- Repeat until entire DAG is scheduled

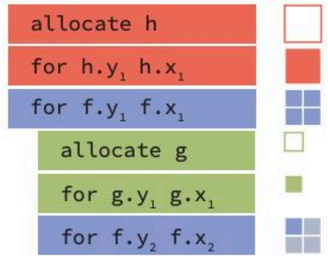
Example Halide DAG



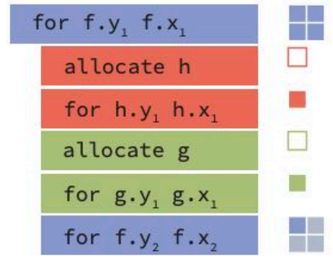
Current state of schedule (after scheduling node f and g)



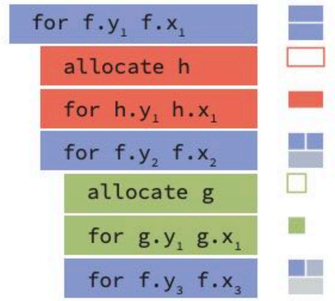
CANDIDATE SUCCESSOR STATES



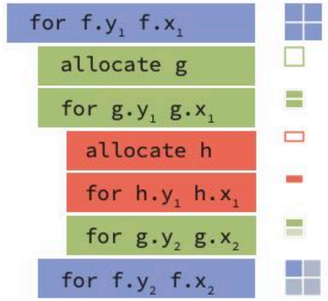
(a) compute at root



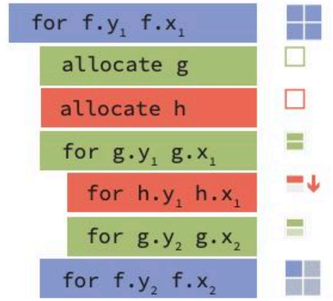
(b) compute h at an existing tiling



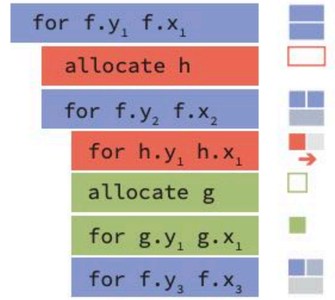
(c) compute h at a new outer tiling of f



(d) compute h at a new sub-tiling of g



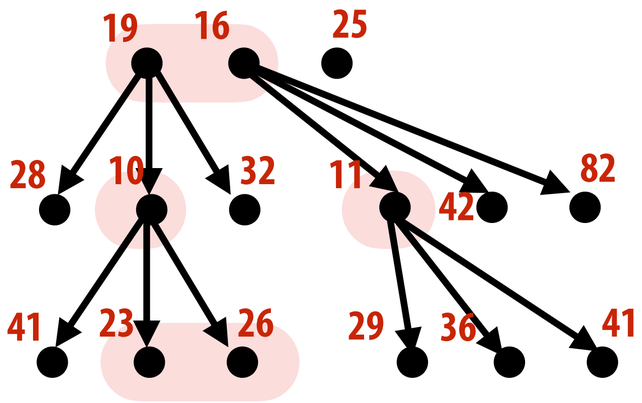
(e) store h at tiles of f, compute h at a new sub-tiling of g



(f) store h at a new outer tiling of f, compute h at sub-tiles of f

Use search to find best performing schedule

- Search over large space of schedules (e.g., greedy search, beam search)



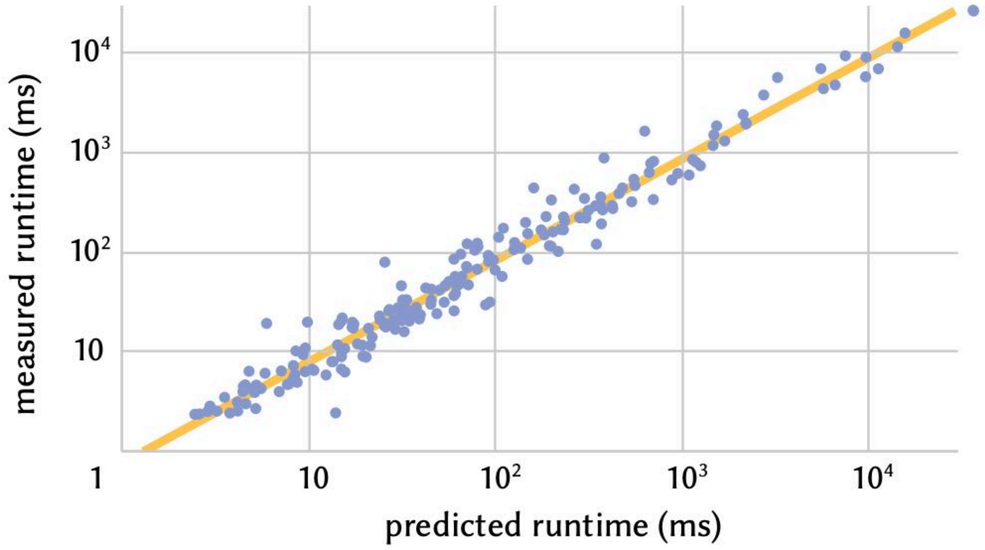
● = a partially scheduled DAG

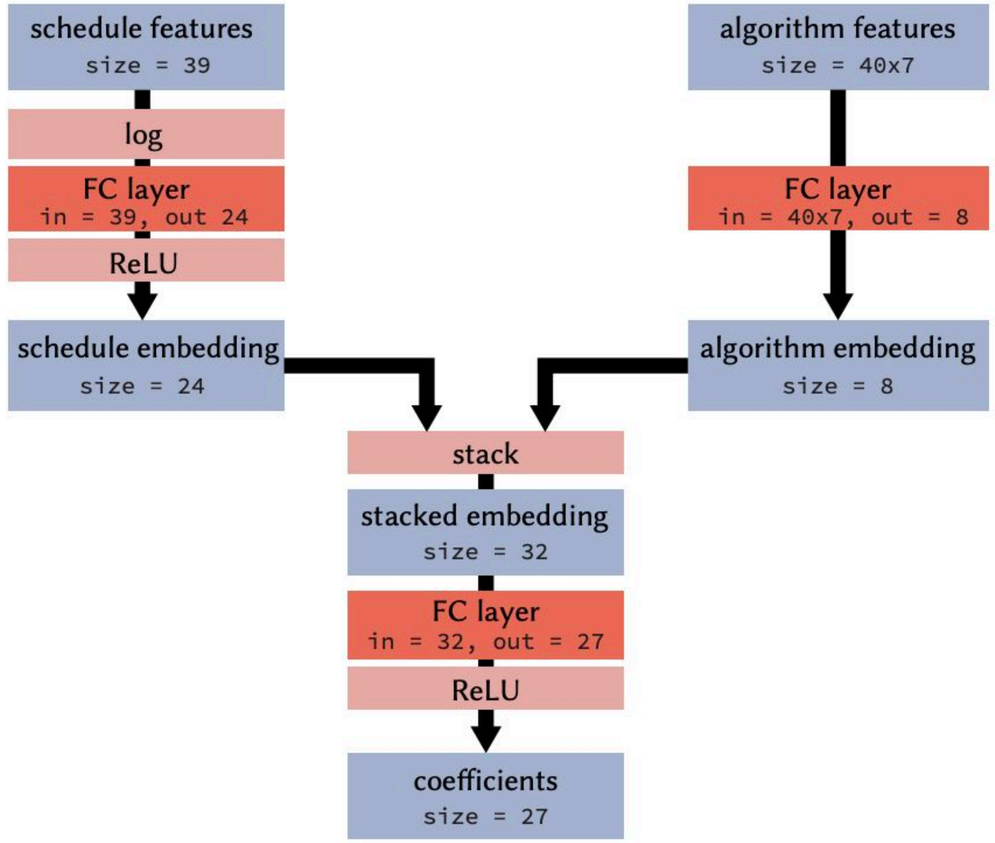
Number = estimated cost of schedule (as given so far)

- Challenge: might need to search over hundreds of thousands of possible schedules… how do we get the cost of a schedule?

Cost estimation using AI

- Given program + schedule… estimate cost *
- Simple MLP that runs in 10's microseconds per schedule (e.g., 1.4M schedules tested in 166 seconds)
  - Trained on a large database of randomly generated Halide programs
  - Training programs compiled and executed to get actual cost

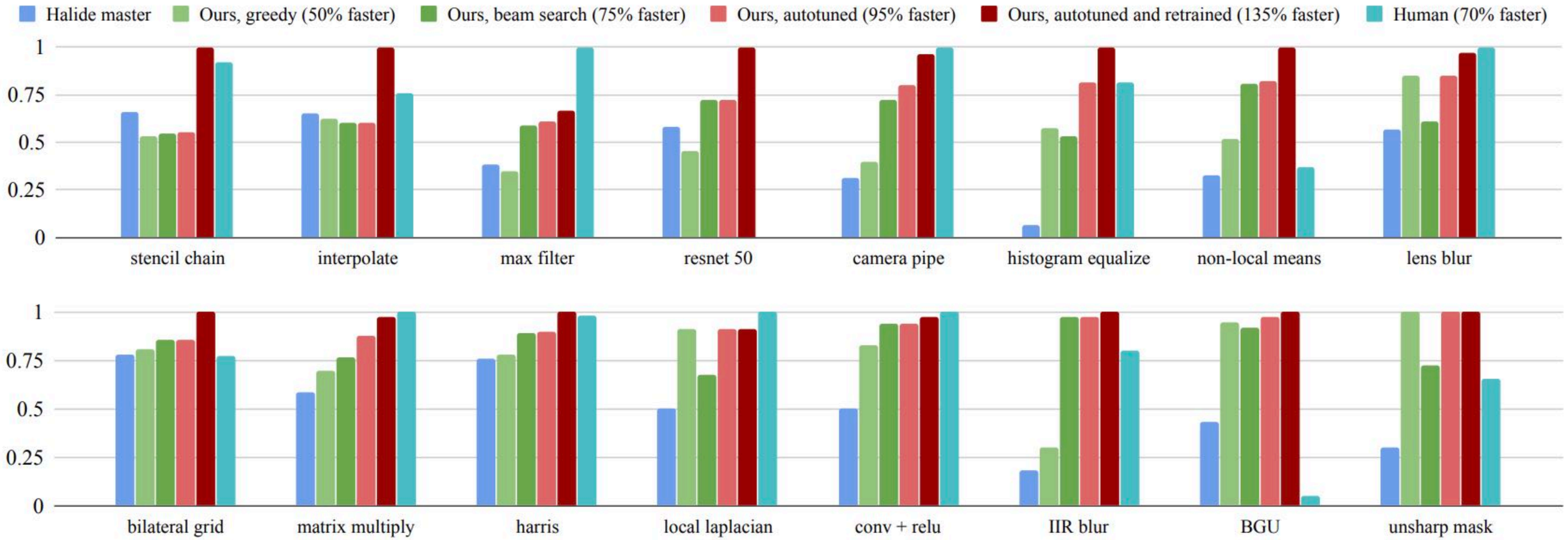




* in practice, doesn't directly compute cost… it outputs 27 coefficients that are plugged into a hand-crafted cost model

Autoscheduler comparable to best known human schedules

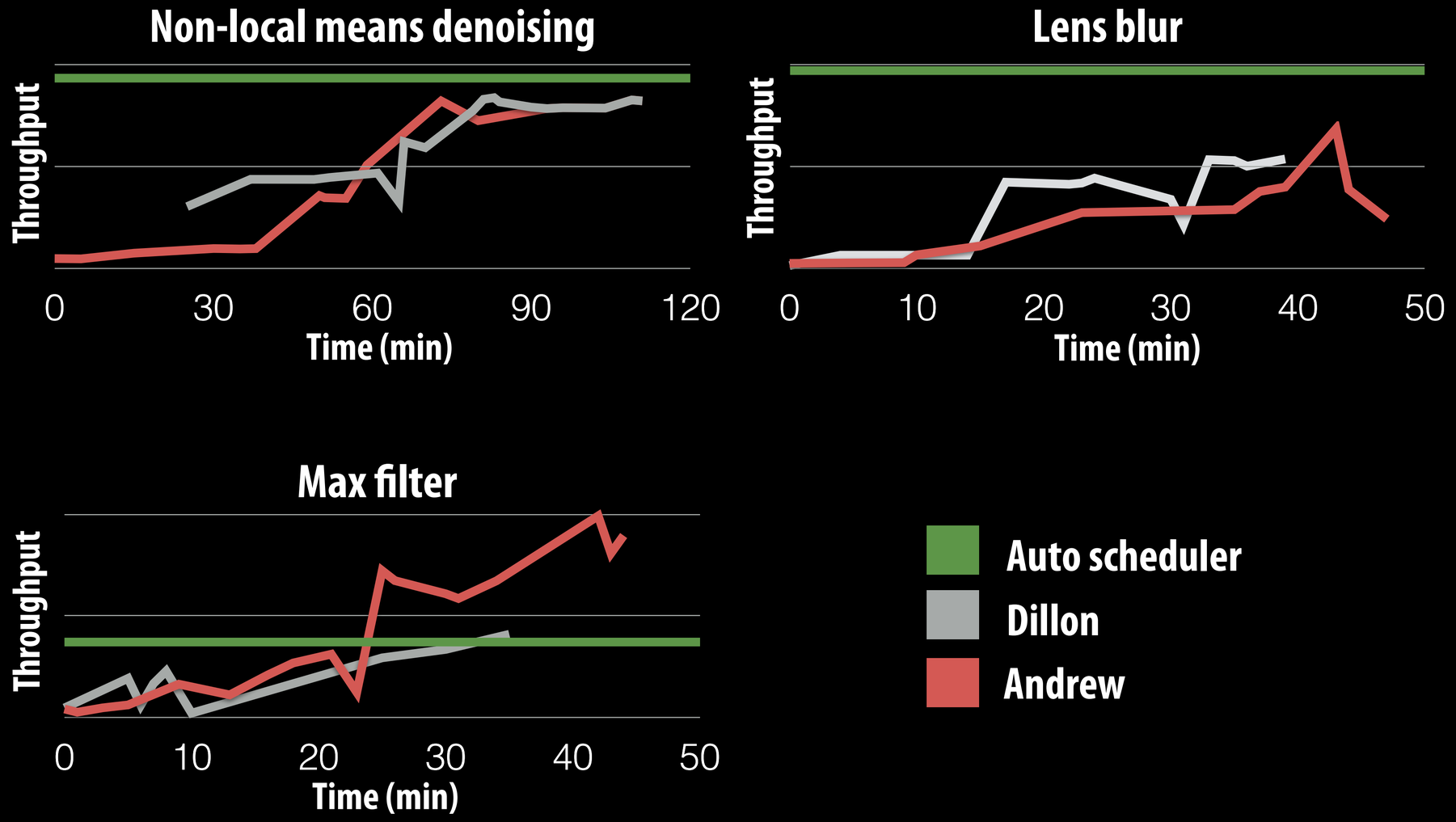
Graphs plot relative throughput (output pixels/second)



TL;DR - [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 on CPUs

Autoscheduler saves time for experts

Earlier results from [Mullapudi 2016], not [Adams 2019]



Takeaways

- Halide scheduling primitives were designed to enhance productivity of expert human programmers that were trying to schedule image processing code

- The high level of abstraction for scheduling also provided a clear way to enumerate the space of all possible schedules, enabling automated search

- Consider searching over all possible permutations of a C++ program 😱

LLM code generation

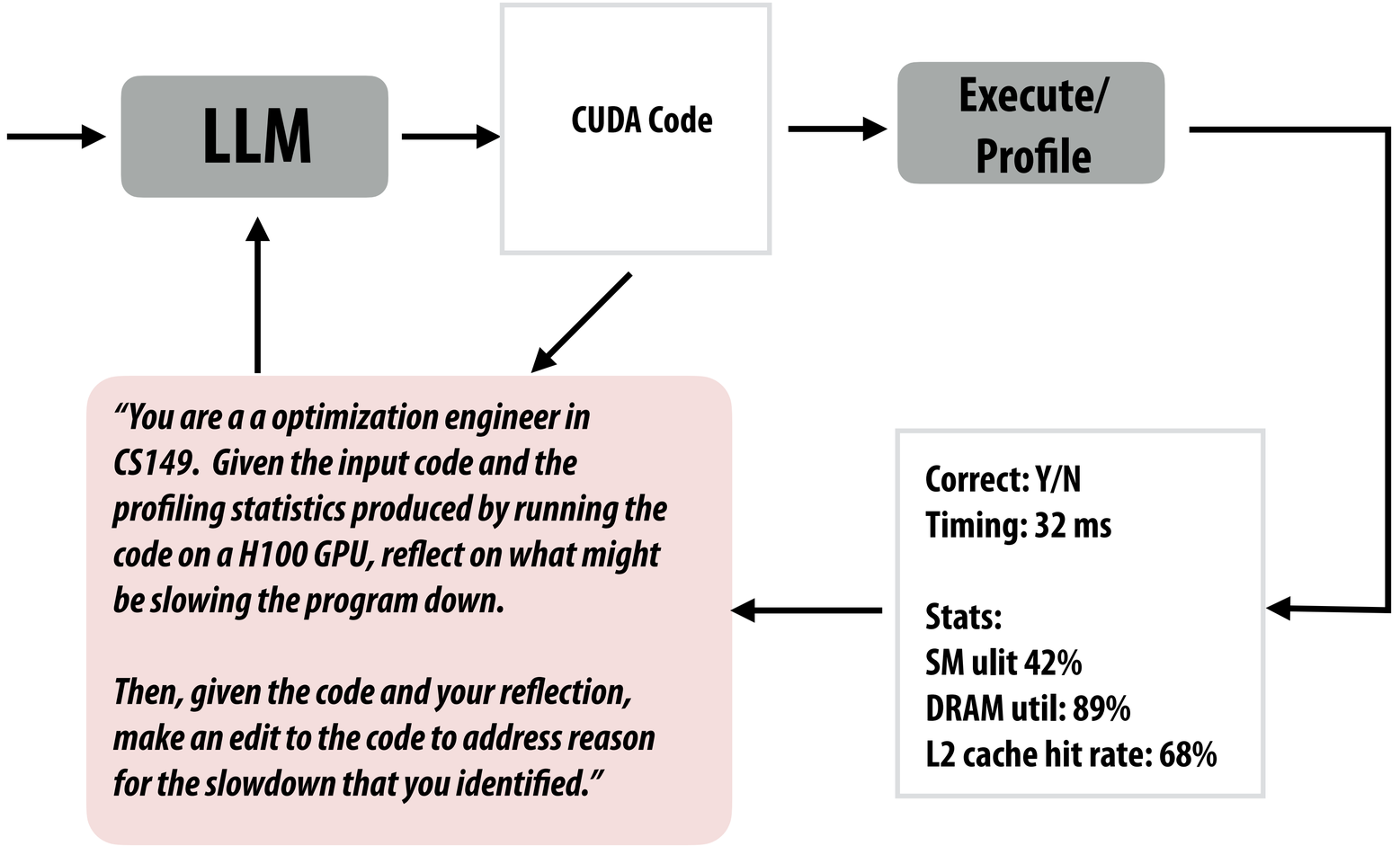
Trial and error via reflection

Starting code (e.g., PyTorch) + LLM prompt

```python
class Model(nn.Module):
    """
    Simple model that performs a matrix multiplication, scales the result, and applies ba
    """
    def __init__(self, in_features, out_features, scale_shape, eps=1e-5, momentum=0.1):
        super(Model, self).__init__()
        self.gemm = nn.Linear(in_features, out_features)
        self.scale = nn.Parameter(torch.randn(scale_shape))
        self.bn = nn.BatchNorm1d(out_features, eps=eps, momentum=momentum)

    def forward(self, x):
        x = self.gemm(x)
        x = x * self.scale
        x = self.bn(x)
        return x

batch_size = 16384
in_features = 4096
out_features = 4096
scale_shape = (out_features,)
```

"You are a performance optimization engineer in CS149. Please rewrite the following PyTorch code as high performance code in CUDA."

Keep in mind the following code optimization principles we discussed in class…

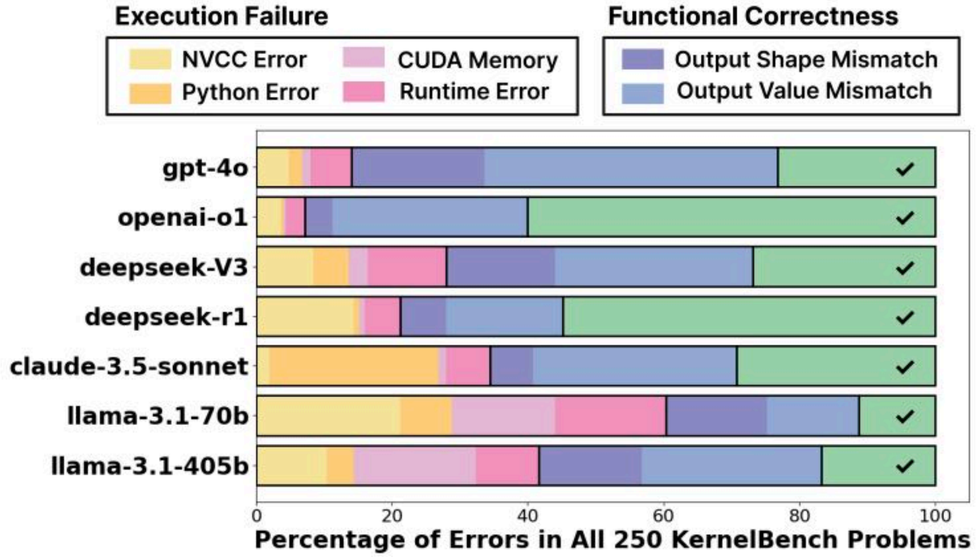


KernelBench

- **A benchmark of hundreds of PyTorch kernels**
- **LMM agent's goal is to automatically produce fast and correct CUDA kernels**

We construct KernelBench to have 4 Levels of categories:

- **Level 1** 🧱: Single-kernel operators (100 Problems) The foundational building blocks of neural nets (Convolutions, Matrix multiplies, Layer normalization)
- **Level 2** 🔗: Simple fusion patterns (100 Problems) A fused kernel would be faster than separated kernels (Conv + Bias + ReLU, Matmul + Scale + Sigmoid)
- **Level 3** ⚛️: Full model architectures (50 Problems) Optimize entire model architectures end-to-end (MobileNet, VGG, MiniGPT, Mamba)
- **Level 4** 🤗: Level Hugging Face Optimize whole model architectures from HuggingFace



Domain specific languages for writing DNN programs help automation as well

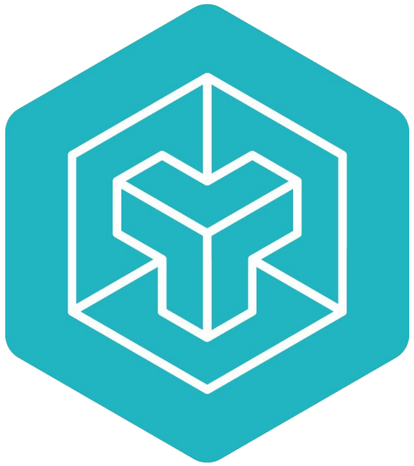
- Good:
  - LLM is now assembling high-performance primitives, not writing low-level CUDA
  - Less likely for correctness mistakes/hallucinations
- Challenge:
  - DNNs can struggle to write correct code in less-used languages (less data to train on… will resolve over time)

ThunderKittens: A Simple Embedded DSL for AI kernels

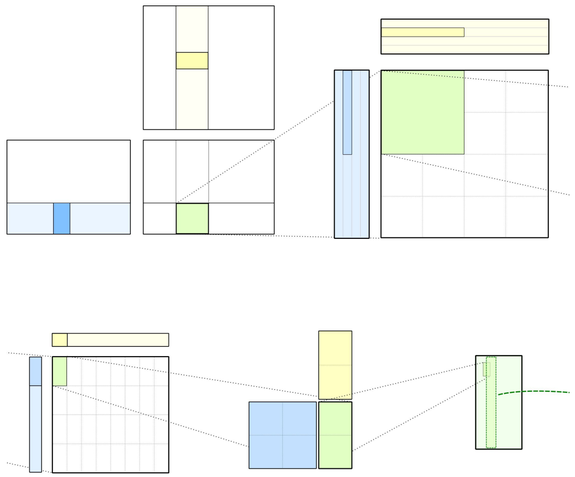
Benjamin Spector, Aaryan Singhal, Simran Arora, Chris Re

(This post is an extended introduction to a longer post we're releasing concurrently.)

Triton



CUTLASS/CuTe



TileLang

TileLang
TILE LANGUAGE

Open question:

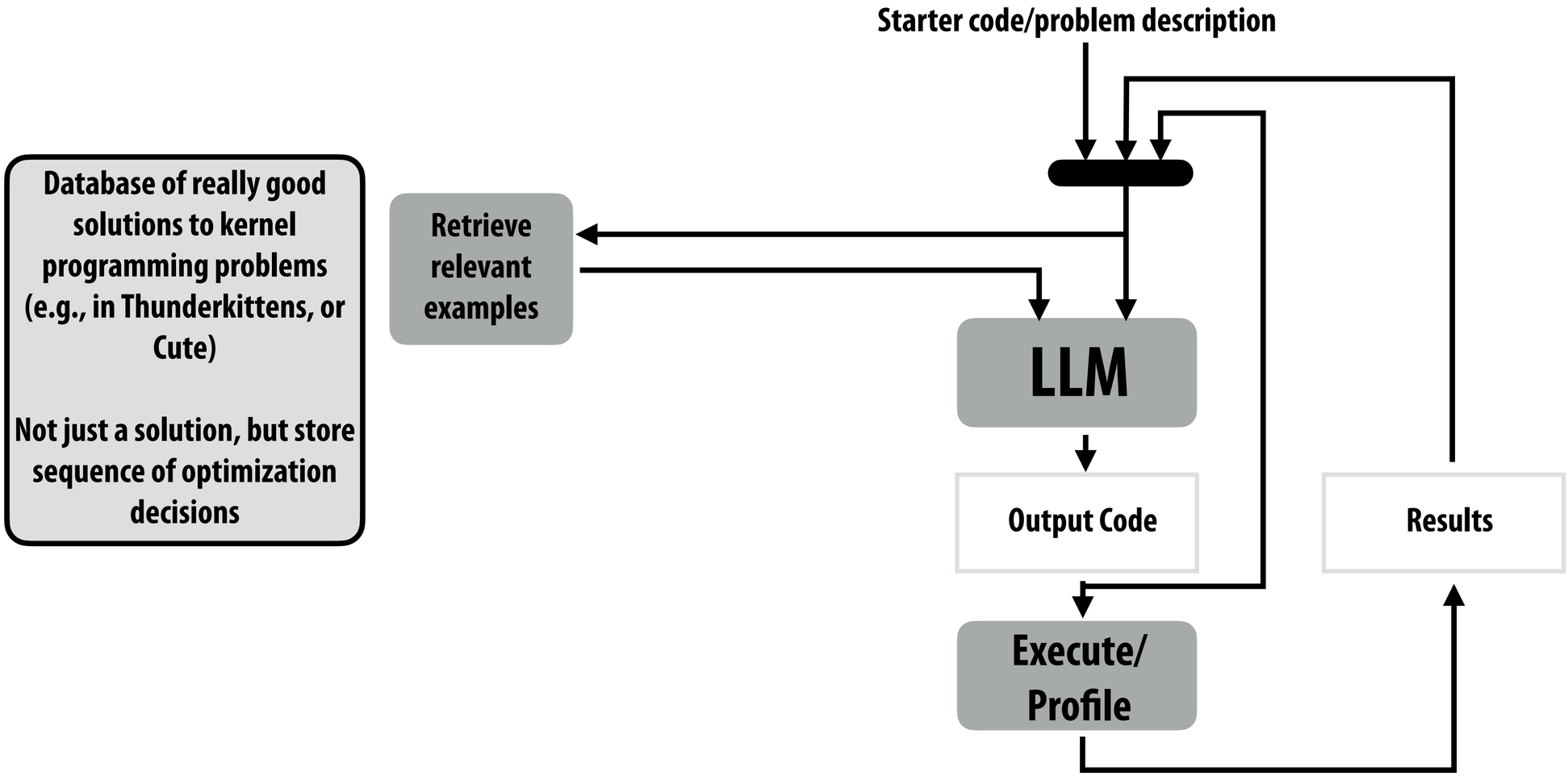
Can an LLM agent serve as a great CS149 student?
At what token cost?

Idea 1: fine-tune LLMs based on experience

- Use experience to fine-tune a custom LLM for a partial type of programming task
- Need large numbers of tasks, ability to fine tune larger LLMs

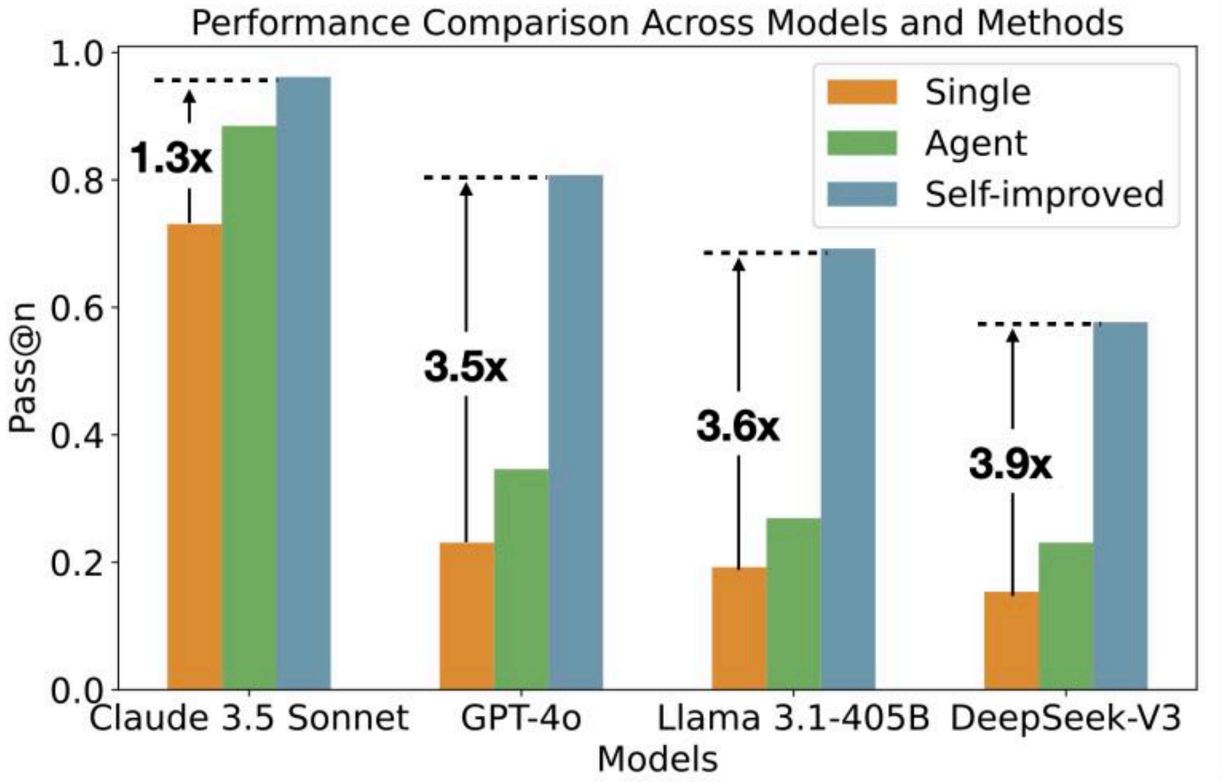
Idea 2: LLM agent self-improves by building up a DB of "example solutions"

- Agent builds up a database of example solutions ("e.g. practice problems")
- Given new problem to solve, it retrieves solutions to most relevant practice problems

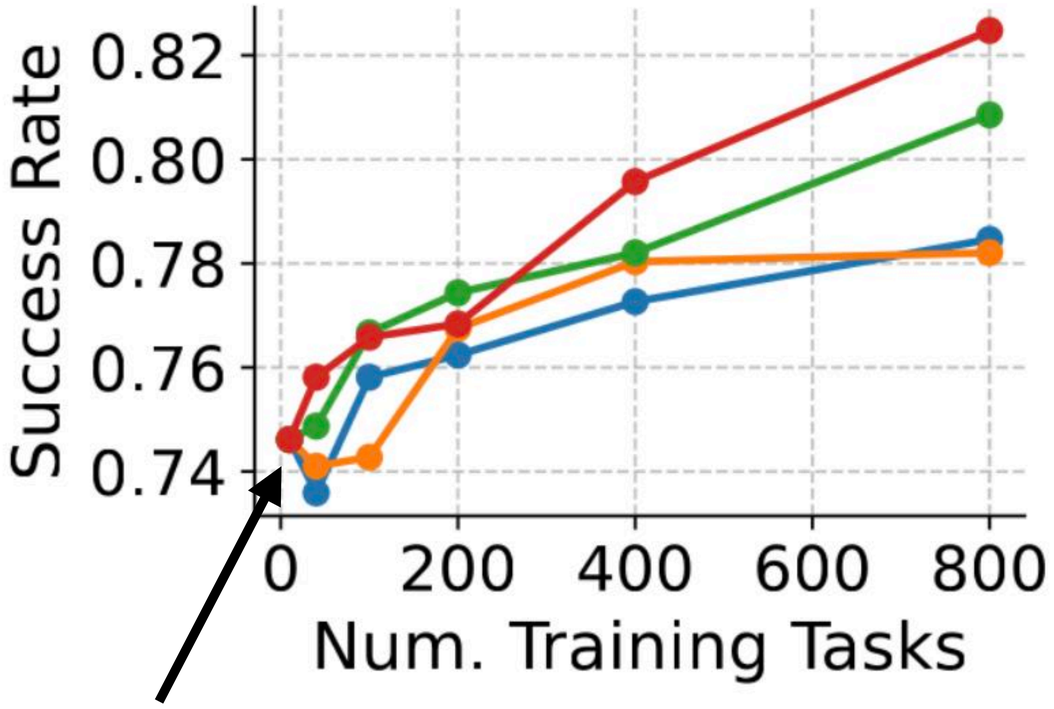


Benefit of database-driven self-improving agent

Writing ML Library Functions for a Domain Specific Accelerator



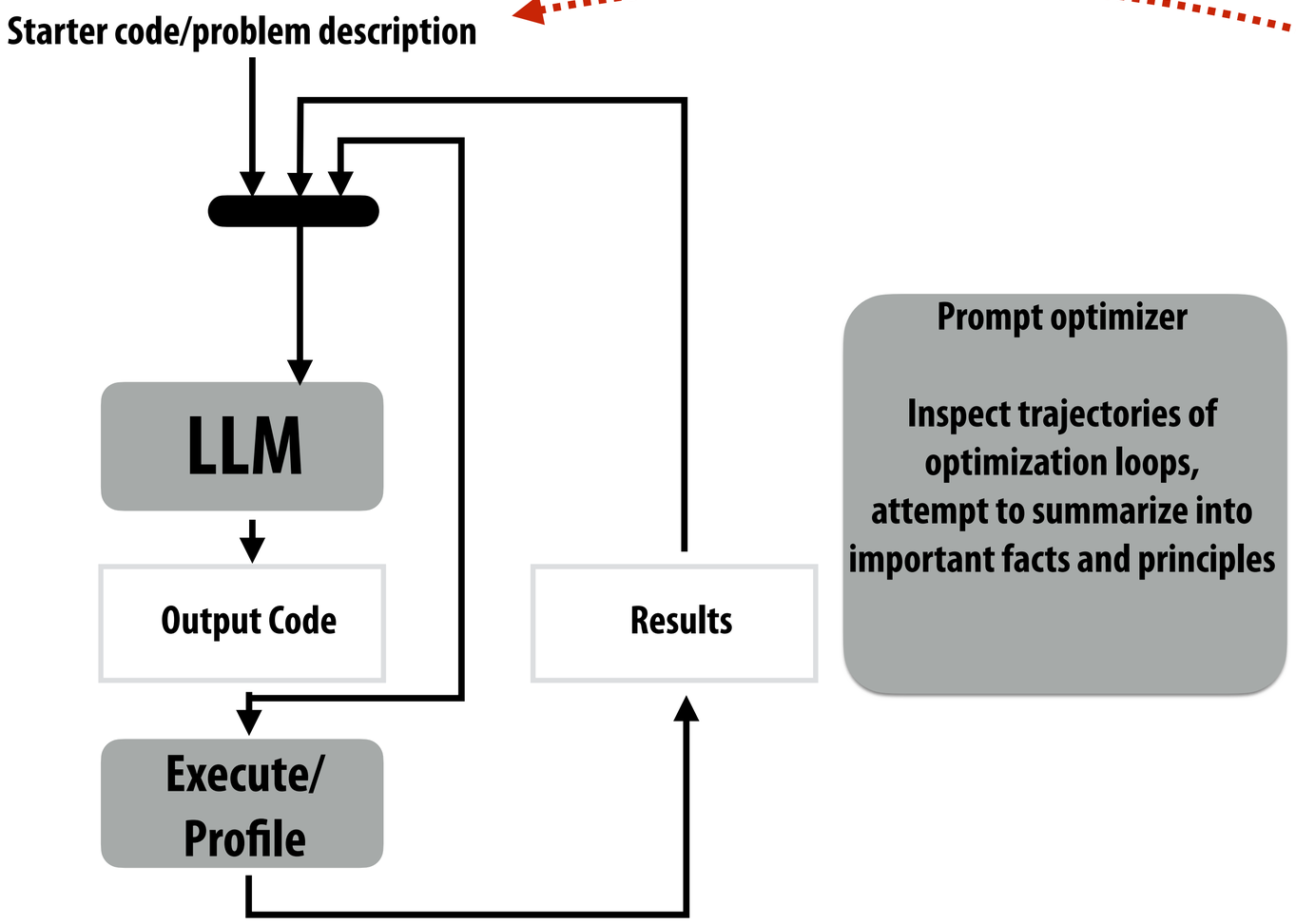
Database Programming Tasks
(Graph Plots Number of Problems Successfully Answered, not Performance)



LLM starting point (no database)

Idea 3: self-improvement via optimizing prompts from experience

Same idea as before, but now update the prompt given to the LLM based on the prior experience, don't just provide relevant examples



Prompt optimizer

Inspect trajectories of optimization loops, attempt to summarize into important facts and principles

The Generated ACE Playbook on AppWorld

STRATEGIES AND HARD RULES

[shr-00009]

When processing time-sensitive transactions involving specific relationships: always resolve identities from the correct source app (phone contacts), use proper datetime range comparisons instead of string matching, and verify all filtering criteria (relationship + time) are met before processing items. This ensures accurate identification and processing of the right transactions.

USEFUL CODE SNIPPETS AND TEMPLATES

[code-00013]

For efficient artist aggregation when processing songs, use defaultdict(list) to map song titles to artist names:

```
from collections import defaultdict artist_map = defaultdict(list) for song in songs:
artist_map[song['title']].extend([artist['name'] for artist in song['artists']])
```

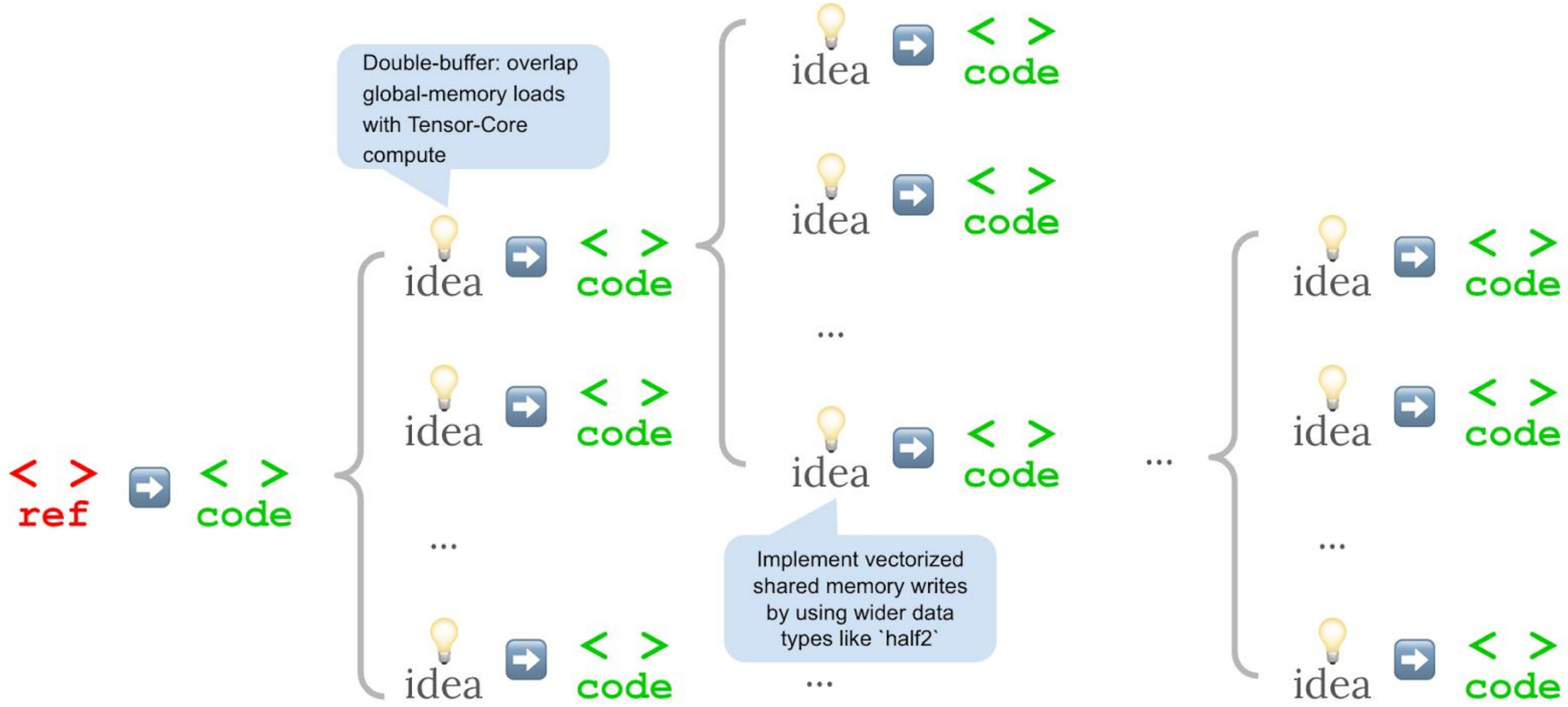
TROUBLESHOOTING AND PITFALLS

[ts-00003]

If authentication fails, troubleshoot systematically: try phone number instead of email as username, clean credentials from supervisor, check API documentation for correct parameters etc. Do not proceed with workarounds.

Idea 4:

- Combine exhaustive search based techniques (like the Halide autotune), with the LLM agentic ideas above
- Extremely high optimization cost, but some of the best results



Summary

- Performance optimization requires a high level of expertise
- And even for experts it's tedious and hard
- And have to repeat it for new machines, slightly different problems
- And companies are spending 10's to 100's of millions of dollars a year or more on AI compute costs

- Seems like a great case for automation

- The best cs149 students of the future will likely be able to work in tangent with automatic agents to accelerate their thinking and their work
- Interesting debates on whether the real value that leads to success is in the DSL design, or the LLM agent!!!!