Lecture 21:
Domain-Specific Programming Systems

Parallel Computer Architecture and Programming

CMU 15-418/15-618, Fall 2024

What we have learnt

- Design computer systems that can **scale**
  - Running faster given more resources

- Design computer systems that are **efficient**
  - Running faster under resource constraints

- Techniques discussed
  - Exploiting parallelism in applications
  - Exploiting locality in applications
  - Leveraging hardware specialization

Various programming models to abstract hardware

Machines with very different performance characteristics

- CPUs, GPUs, TPUs, systolic arrays

Different technologies and performance characteristics within the same machine at different scales

- **Within a core**: SIMD, multi-threading: fine grained sync and comm
  - **Abstractions**: SPMD programming (ISPC, CUDA, OpenCL)
- **Across cores**: coherent shared memory via fast on-chip network
  - **Abstractions**: OpenMP pragma, Cilk
- **Across racks**: distributed memory, multi-stage network
  - **Abstractions**: message passing (MPI, Go, Spark, Legion, Charm++)

Credit: Pat Hanrahan

Various programming models to abstract hardware

Machines with very different performance characteristics

- CPUs, GPUs, TPUs, systolic arrays

Different technologies and performance characteristics within the same machine at different scales

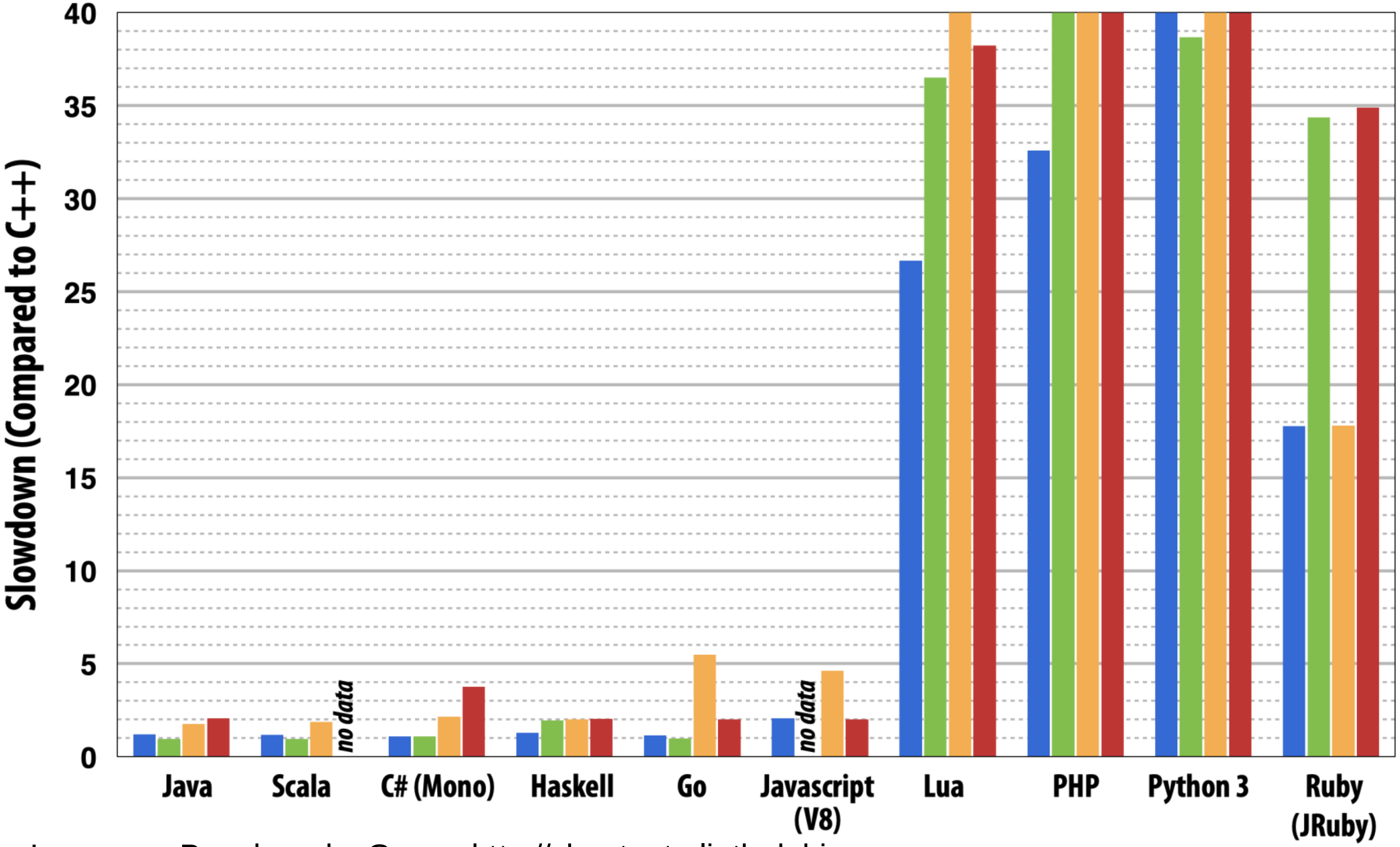
To be efficient, software must be optimized for HW characteristics

- Difficult even in the case of one level of one machine
- Combinatorial complexity of optimizations when considering a complex machine, or different machines
- Loss of software portability

Credit: Pat Hanrahan

Most software systems use hardware inefficiently

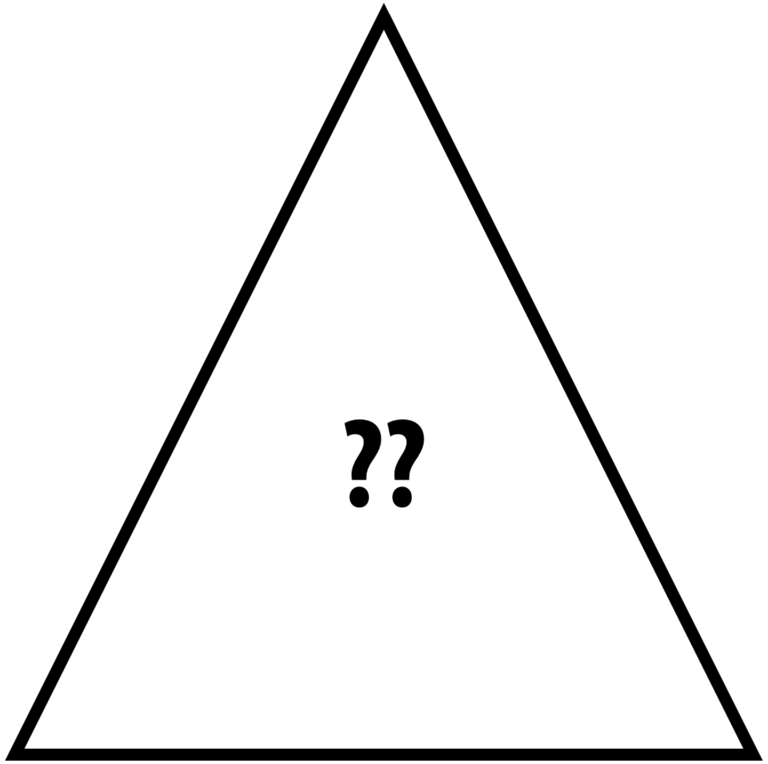
Compared against GCC -o3 (no manual vector optimizations)



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

The Magical Ideal Parallel Programming System

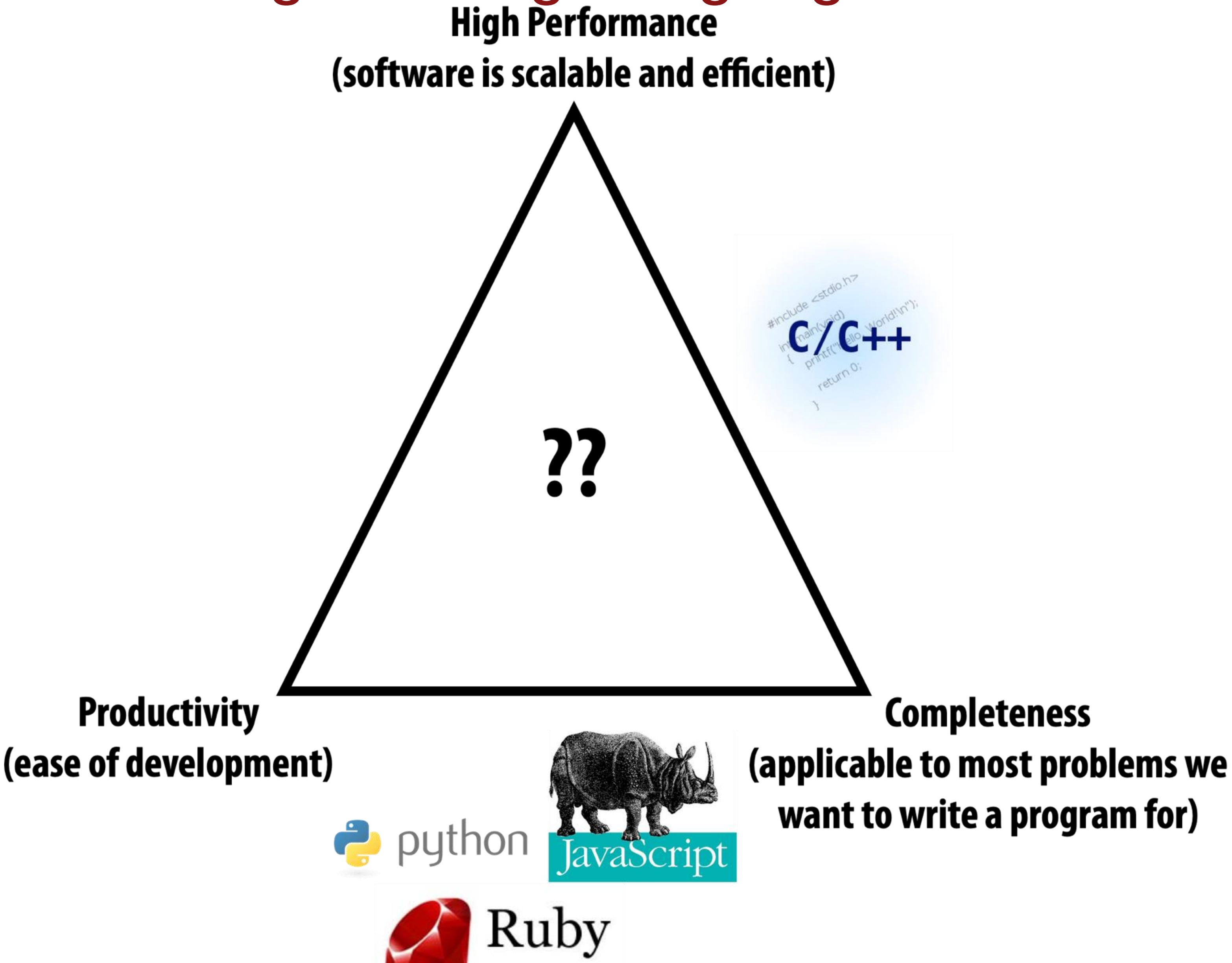
High Performance
(software is scalable and efficient)



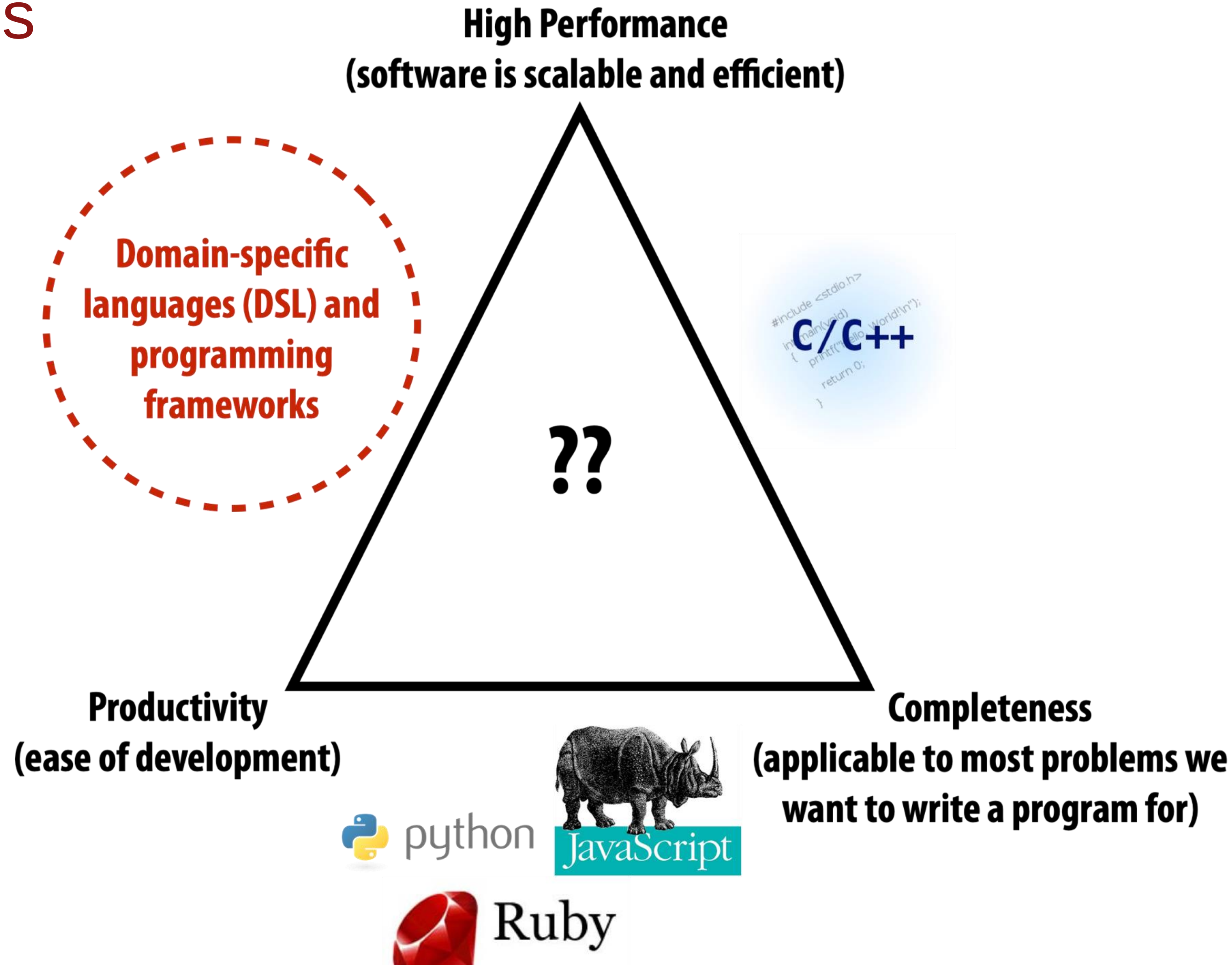
Productivity
(ease of development)

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

Widely Used Programming Languages



Growing Interest in Domain-Specific Programming Systems



Domain-Specific Programming Systems

Key idea: raise level of abstraction for expressing programs

Introduce high-level programming primitives specific to an application domain

- **Productive**: intuitive to use, portable across machines, primitives correspond to behaviors frequently used to solve problems in targeted domain
- **Performant**: system uses domain knowledge to provide efficient, optimized implementation(s)
  - Given a machine: system knows what algorithms to use, parallelization strategies to employ for this domain

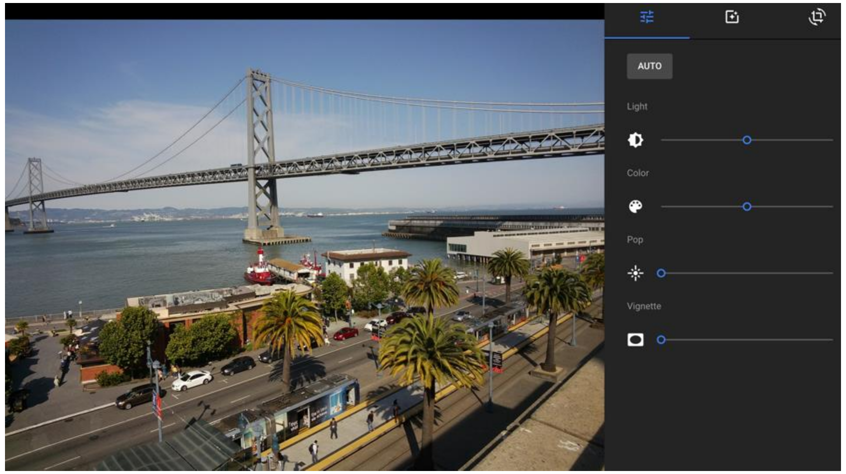
Cost: loss of generality/completeness

Two Domain-Specific Programming Systems

1. Halide: for image processing
2. TVM: for deep learning

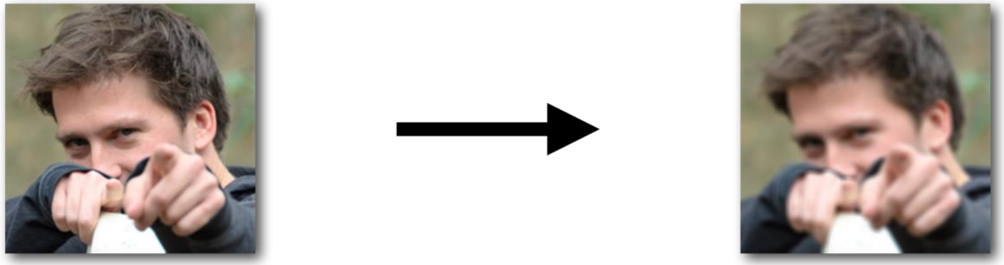
Halide:
a Domain-Specific Language for Image Processing

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



A Quick Tutorial on High-Performance Image Processing

Image Blur



(Zoom view)

3x3 Image Blur (a convolution with predefined wights)

Total work per image: 9 * WIDTH * HEIGHT
For NxN filter: N * N * WIDTH * HEIGHT

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

float weights[] = {1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9};

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;
  }
}
```

Two-Pass 3x3 Blur

Total work per image: 6 * WIDTH * HEIGHT
For NxN filter: 2 * N * WIDTH * HEIGHT
Extra memory: WEIGHT * NEIGHT
3x lower arithmetic intensity

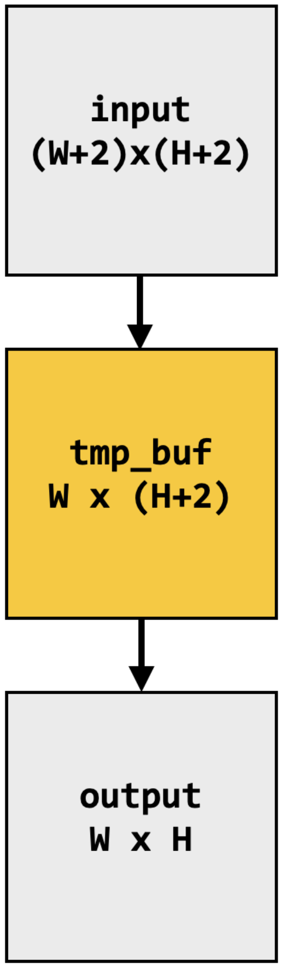
```
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];
```

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

1D horizontal blur

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

1D vertical blur



Two-Pass 3x3 Blur (Chunked)

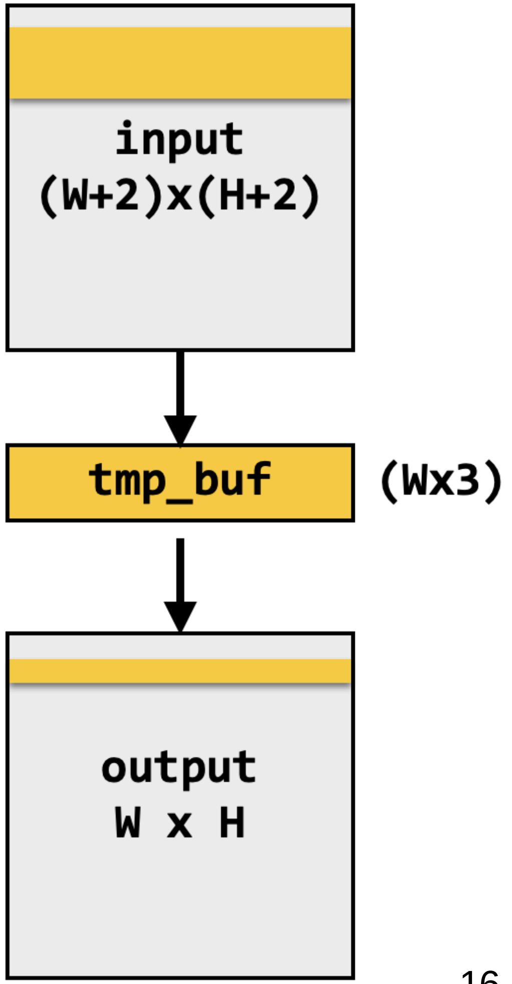
Total work per row of output:
- **step 1: 3 x 3 x WIDTH work**
- **step 2: 3 x WIDTH work**

Total work per image = 12 x WIDTH x HEIGHT
Loads from tmp_buffer are cached

```
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * 3];
float output[WIDTH * HEIGHT];
float weights[] = {1.0/3, 1.0/3, 1.0/3};

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;
  }
}
```

Produce 3 rows of tmp_buf

Combine them together to get one row of output



Conflicting goals (once again…)

- Want to be computationally efficient (perform fewer operations)
- Want to take advantage of locality when possible
  - Otherwise computationally efficient code will be bandwidth bound
- Want to execute in parallel (multi-core, SIMD within core)

Optimized C++ code: 3x3 image blur

```
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);
}}}}}
```

+ 10x faster than the original code
- Specific to SSE (not AVX2), CPU-code only
- Lacks readability, portability, modularity

Halide: Decouple Algorithm from Schedule

- Algorithm: *what* to do
- Schedule: *how* to do

```
Func halide_blur(Func in) {
 Func tmp, blurred;
 Var x, y, xi, yi;

 // The algorithm
 tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

 // The schedule
 blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
 tmp.chunk(x).vectorize(x, 8);

 return blurred;
}
```

Why Decoupling Algorithm from Schedule?

- Algorithm: *what* to do
- Schedule: *how* to do

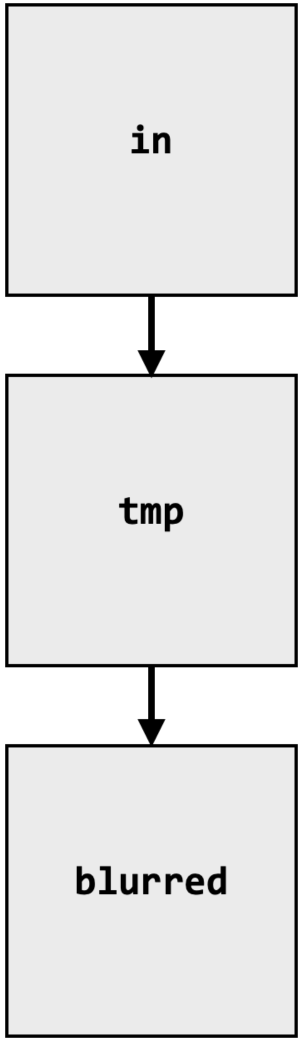
- Easy for programmers to build pipelines
- Easy for programmers to specify & explore optimizations
- Easy for compilers to generate fast code

Algorithm: Pure Functional

- Declarative specification
- Pipeline stages are pure functions from coordinates to values
- No explicit bounds
- No loops or traversal orders
- Only feed forward pipelines

```
// The algorithm
tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
```

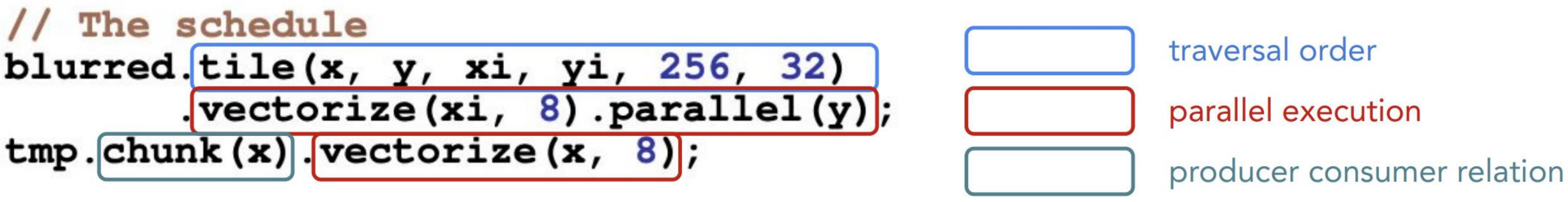
Think of a Halide algorithm as a pipeline



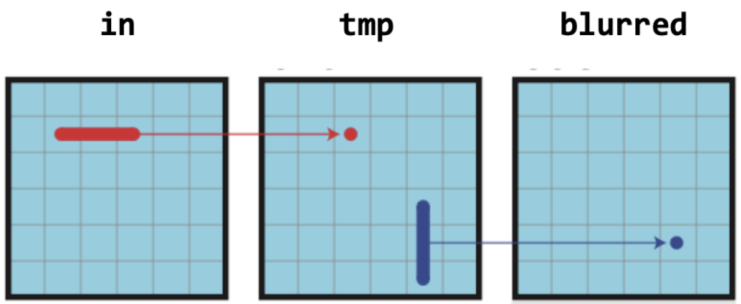
Schedule: describe how to execute a pipeline

- Defines intra-stage order and inter-stage interleaving
- For each stage:

1) In which order should we compute its values?
2) How to map onto parallel execution resources like SIMD units and GPU blocks?

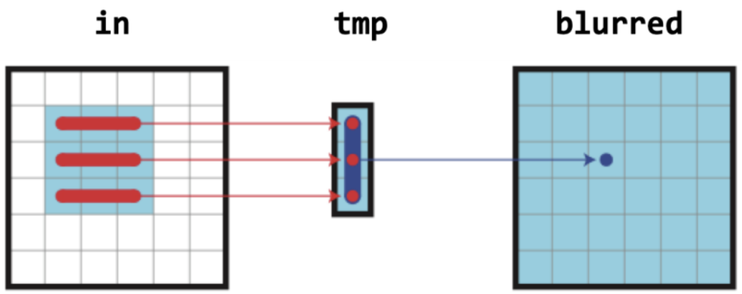


Producer/Consumer Scheduling Primitives



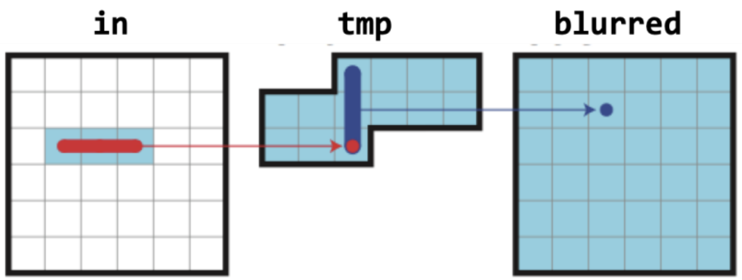
breadth first: each function is entirely evaluated before the next one.

"Root"



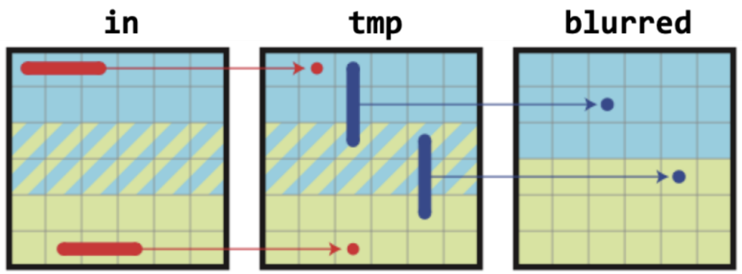
total fusion: values are computed on the fly each time that they are needed.

"Inline"



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

"Sliding Window"



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

"Chunked"

Producer/Consumer Scheduling Primitives

```
// Halide program definition
Func halide_blur(Func in) {

  Func blurx, out;
  Var  x, y, xi, yi

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

  // “the schedule” (how to do it)
  blurx.compute_at(ROOT);
  return out;
}
```

“Root”:

compute all points of the producer, then run consumer (minimal locality)

```
void halide_blur(uint8_t* in, uint8_t* out) {
    uint8_t blurx[WIDTH * HEIGHT];

    for (int y=0; y<HEIGHT; y++) {
      for (int x=0; y<WIDTH; x++) {
          blurx[] = ...

    for (int y=0; y<HEIGHT; y++) {
      for (int x=0; y<WIDTH; x++) {
          out[] = ...
}
```

```
// Halide program definition
Func halide_blur(Func in) {

  Func blurx, out;
  Var  x, y, xi, yi

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

  // “the schedule” (how to do it)
  blurx.inline();
  return out;
}
```

“Inline”:

revaluate producer at every use site in consumer (maximal locality)

```
void halide_blur(uint8_t* in, uint8_t* out) {
    for (int y=0; y<HEIGHT; y++) {
      for (int x=0; y<WIDTH; x++) {
          out[] = (((in[(y-1)*WIDTH+x-1] +
                     in[(y-1)*WIDTH+x] +
                     in[(y-1)*WIDTH+x+1]) / 3) +
                    ((in[y*WIDTH+x-1] +
                     in[y*WIDTH+x] +
                     in[y*WIDTH+x+1]) / 3) +
                    ((in[(y+1)*WIDTH+x-1] +
                     in[(y+1)*WIDTH+x] +
                     in[(y+1)*WIDTH+x+1]) / 3));
}
```

Schedule: describe how to execute a pipeline

```
// Halide program definition
Func halide_blur(Func in) {

  Func blurx, out;
  Var  x, y, xi, yi

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

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

```
void halide_blur(uint8_t* in, uint8_t* out) {
   #pragma omp parallel for
   for (int y=0; y<HEIGHT; y+=32) {       // tile loop
      for (int x=0; y<WIDTH; x+=256) {  // tile loop

         // buffer
         uint8_t* blurx[34 * 256];

         // produce intermediate buffer
         for (int yi=0; yi<34; yi++) {
            // SIMD vectorize this loop (not shown)
            for (int xi=0; xi<256; xi++) {
               blurx[yi*256+xi] =
                   (in[(y+yi-1)*WIDTH+x+xi-1] +
                   in[(y+yi-1)*WIDTH+x+xi] +
                   in[(y+yi-1)*WIDTH+x+xi+1]) / 3.0;
            }
         }

         // consumer intermediate buffer
         for (int yi=0; yi<32; yi++) {
            // SIMD vectorize this loop (not shown)
            for (int xi=0; xi<256; xi++) {
               out[(y+yi)*256+(x+xi)] =
                   (blurx[yi*256+xi] +
                   blurx[(yi+1)*256+xi] +
                   blurx[(yi+2)*256+xi]) / 3.0;
            }
         }
      } // loop over tiles
   }  // loop over tiles
}
```

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

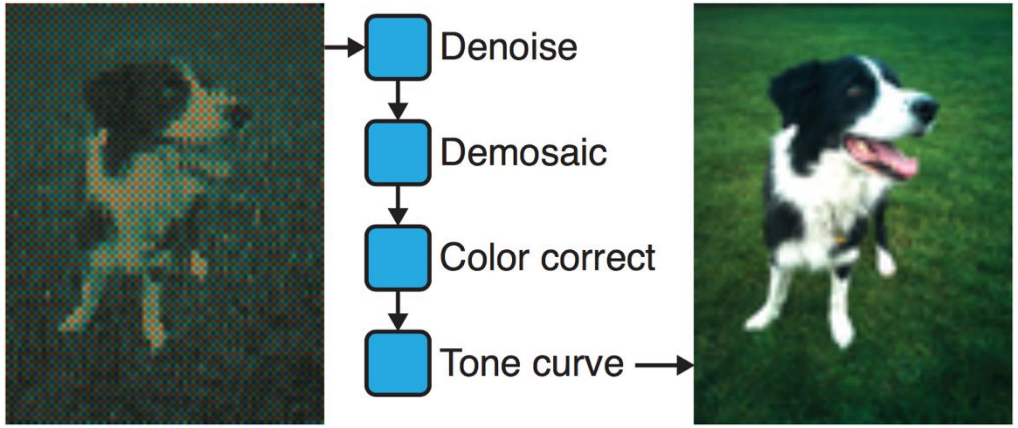
Halide: two domain-specific co-languages

- **Functional language** for describing image processing operations
- **Domain-specific language** for describing schedules
- **Design principle**: separate “algorithm specification” from its schedule
  - Programmer’s responsibility: provide a high-performance schedule
  - Compiler’s responsibility: carry out mechanical process of generating threads, SIMD instructions, managing buffers, etc.
  - Result: enable programmer to rapidly explore space of schedules
    - (e.g., “tile these loops”, “vectorize this loop”, “parallelize this loop across cores”)
- **Domain scope**:
  - All computation on regular N-D coordinate spaces
  - Only feed-forward pipelines
  - All dependencies inferable by compiler

Example Halide Results

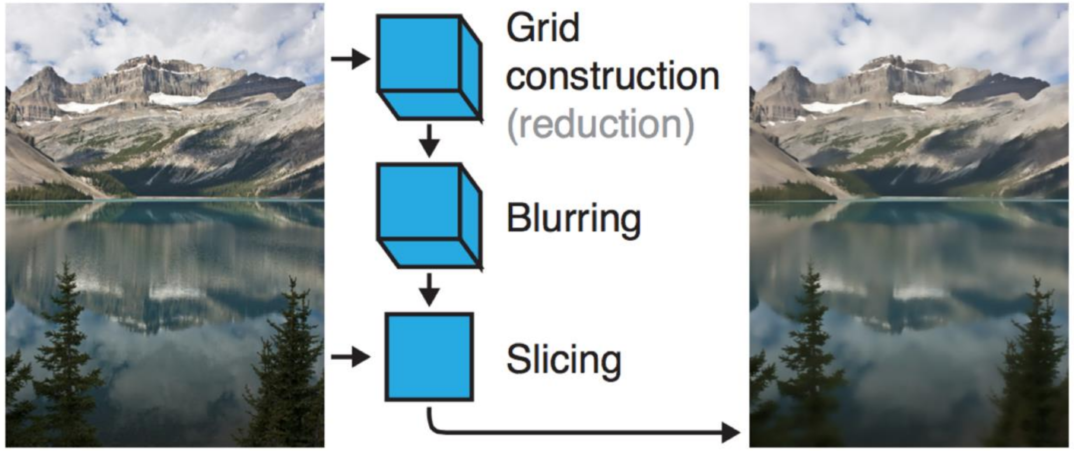
Camera RAW processing pipeline
(Convert RAW sensor data to RGB)

- Original: **463** lines of hand-tuned ARM NEON assembly
- Halide: **2.75x** less code, **5%** faster



Bilateral filter

- 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



Recap: Halide is a DSL

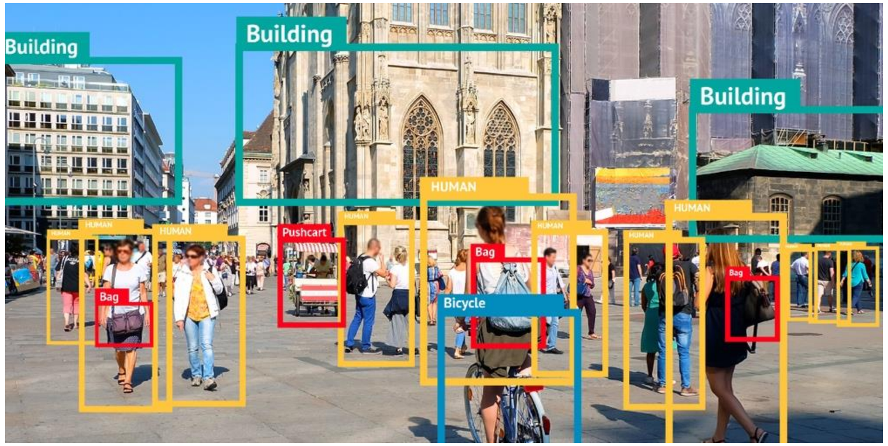
For helping developers optimize image processing code more rapidly

- Halide doesn't decide how to optimize a program for a novice programmer
- Halide provides primitives for a programmer to rapidly express what optimizations the system should apply
- **Halide carries out the nitty-gritty of mapping that strategy to a machine**

Two Domain-Specific Programming Systems

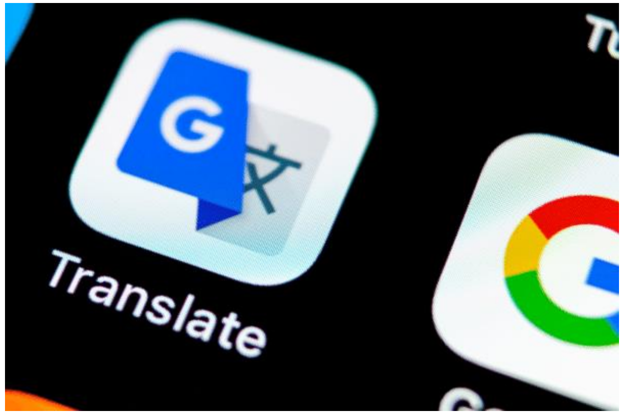
1. Halide: for image processing
2. TVM: for deep learning

The Success of Machine Learning Today

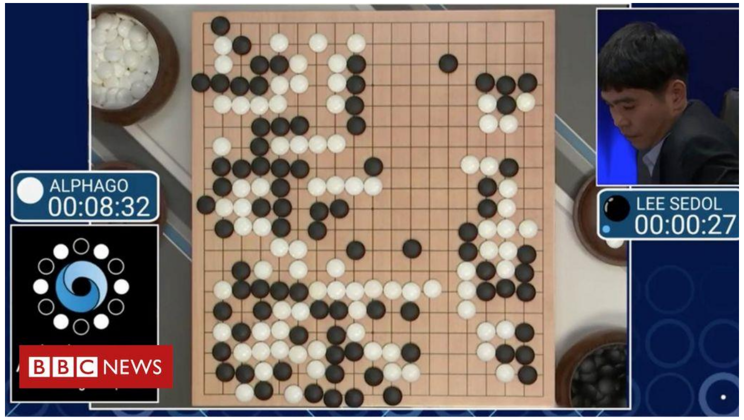


Object detection

Autonomous vehicles



Machine translation

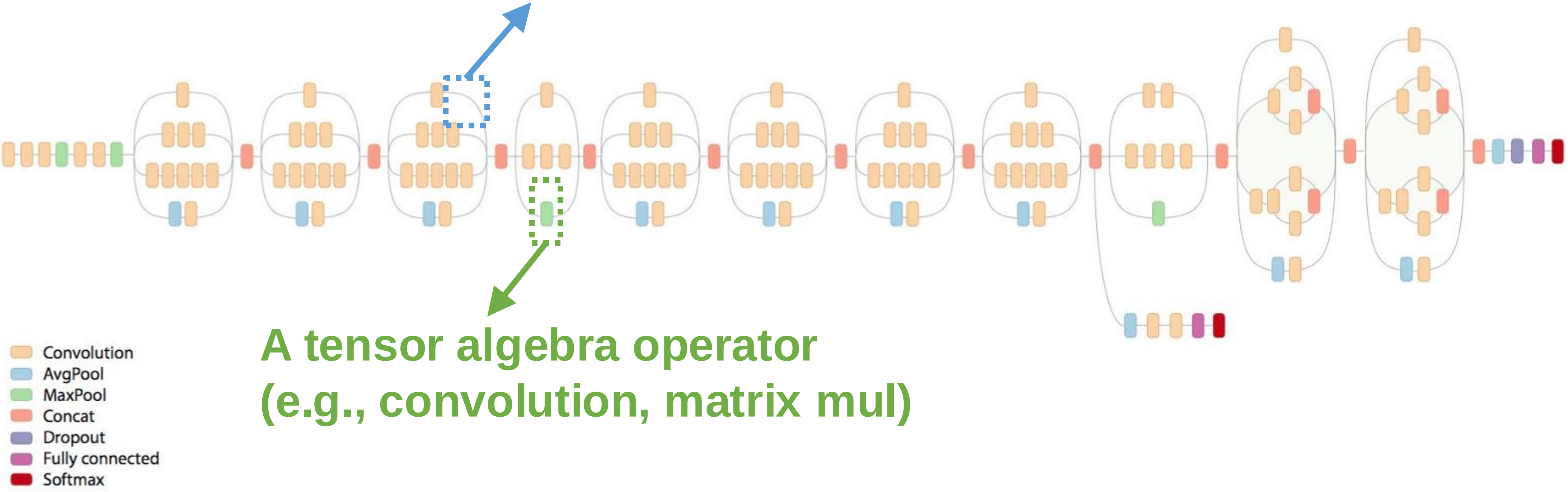


Game playing

Deep Neural Network

- Collection of simple trainable mathematical units that work together to solve complicated tasks

A tensor (i.e., *n*-dimensional array)



A tensor algebra operator (e.g., convolution, matrix mul)

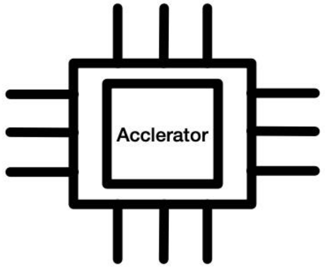
TVM: A Learning-based Compiler for Deep Learning

Frameworks

Explosion of models and frameworks

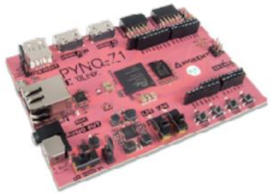
Goal: efficiently deploy deep learning on modern hardware platforms

Explosion of hardware backends



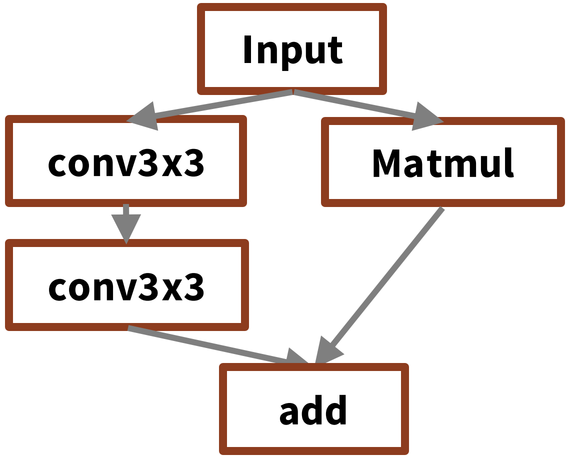






4

Existing Approach: Engineer Optimized Tensor Operators



Matmul: Operator Specification

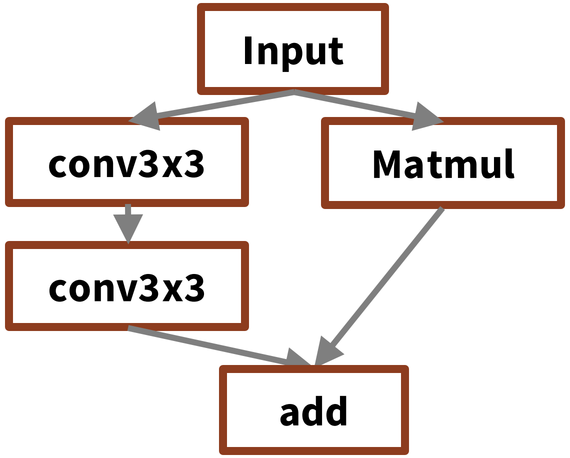
```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Vanilla Code

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

* Slides from Tianqi Chen

Existing Approach: Engineer Optimized Tensor Operators



Matmul: Operator Specification

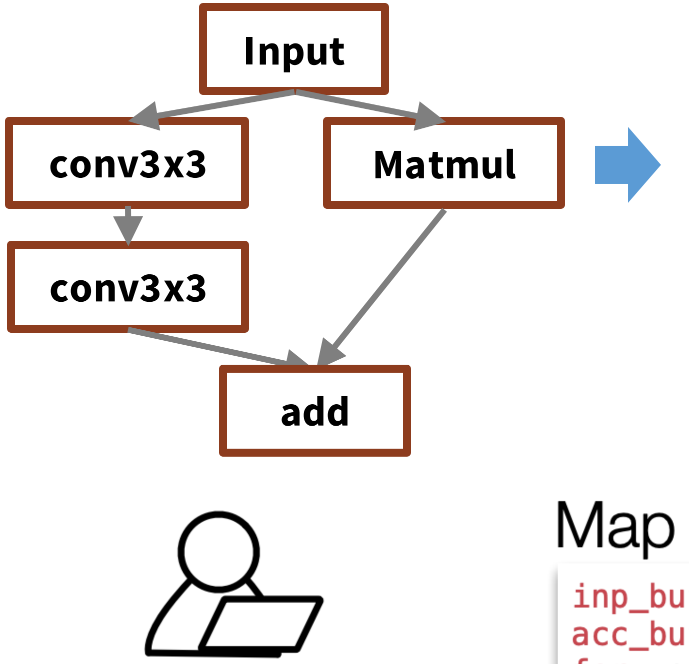
```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Loop Tiling for Locality

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
               A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

* Slides from Tianqi Chen

Existing Approach: Engineer Optimized Tensor Operators



Matmul: Operator Specification

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

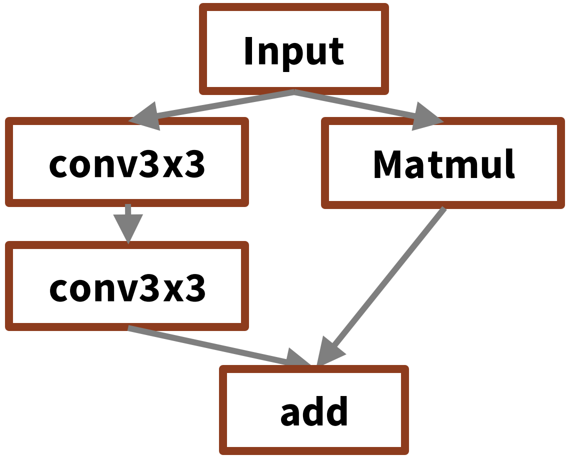
Map to Accelerators

```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

Human exploration of optimized code

* Slides from Tianqi Chen

Challenge: Billions of Possible Optimization Choices in the Search Space




```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Search Space of Possible Program Optimizations

Low-level Program Variants

```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
                A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

* Slides from Tianqi Chen

TVM: Learning-based Compiler for Deep Learning

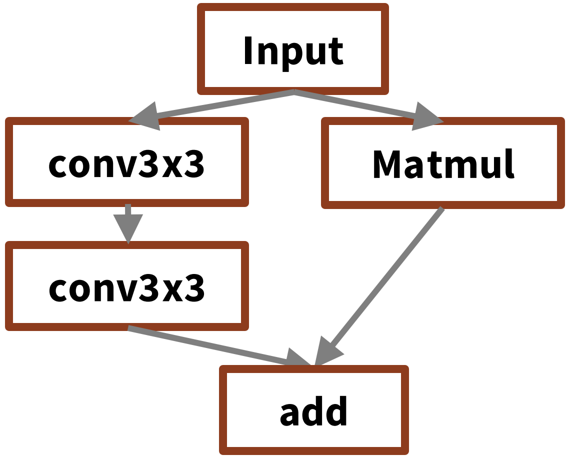
Frameworks

Hardware-aware Search Space of Optimized Tensor Programs

Machine Learning based Program Optimizer

4

Hardware-aware Search Space



≈ Halide's algorithm

Tensor Expression Language (Specification)

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

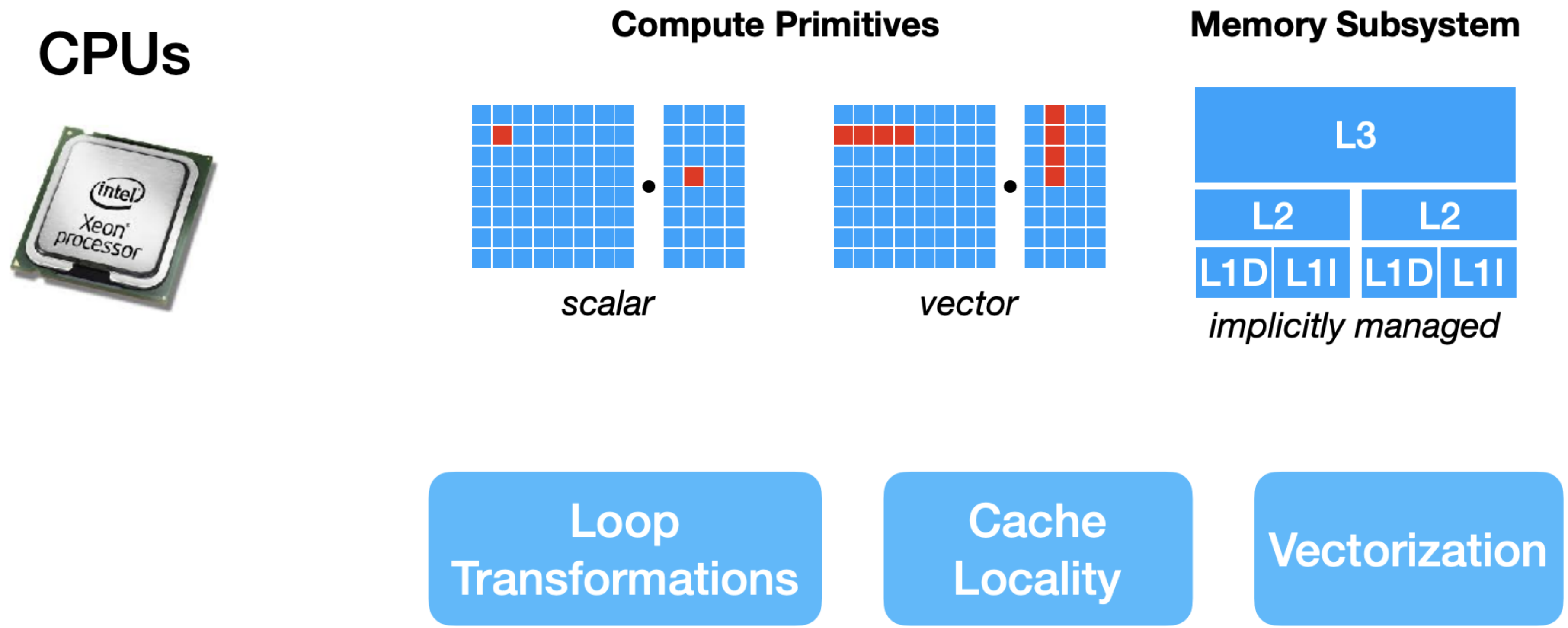
Define search space of hardware aware mappings from expression to hardware program

Based on Halide's compute/schedule separation

Hardware

Hardware-aware Search Space

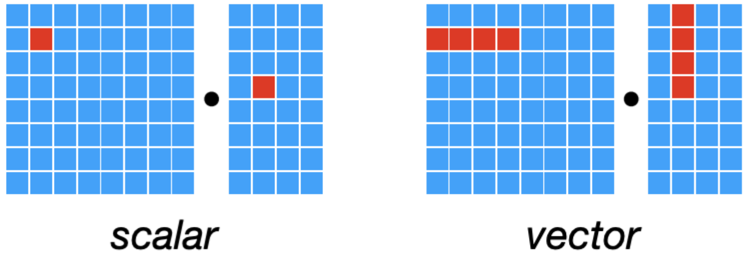
Reuse primitives from Halide



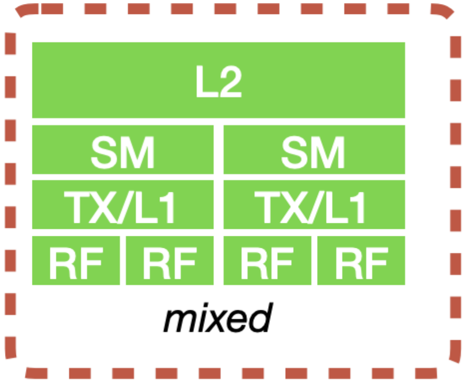
Hardware-aware Search Space

GPUs

Compute Primitives



Memory Subsystem



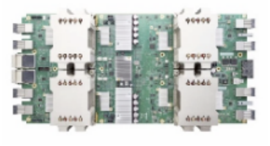
Shared memory among compute cores

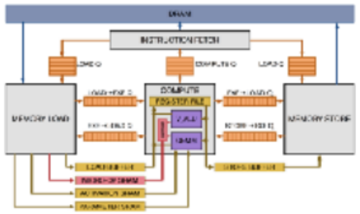
Use of Shared Memory

Thread Cooperation

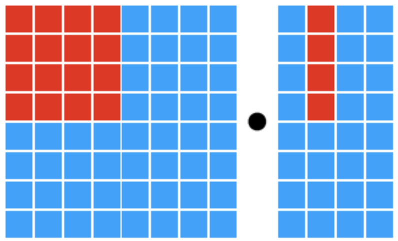
Hardware-aware Search Space

TPU-like Specialized Accelerators



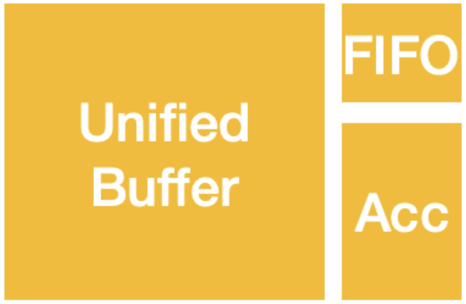


Compute Primitives



tensor

Memory Subsystem

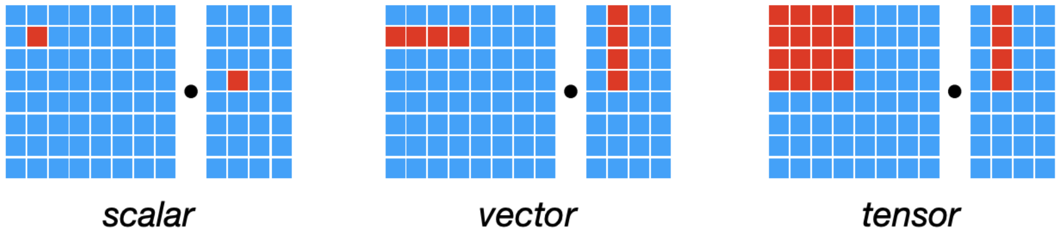


explicitly managed

Tensorization

Tensorization Challenge

Compute primitives



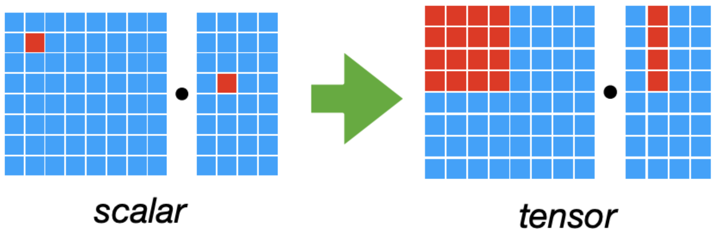
Hardware designer: declare tensor instruction interface with Tensor Expression

```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
              t.sum(w[i, k] * x[j, k], axis=k))

def gemm_intrin_lower(inputs, outputs):
   ww_ptr = inputs[0].access_ptr(“r")
   xx_ptr = inputs[1].access_ptr("r")
   zz_ptr = outputs[0].access_ptr("w")
   compute = t.hardware_intrin("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
   reset = t.hardware_intrin("fill_zero", zz_ptr)
   update = t.hardware_intrin("fuse_gemm8x8_add", ww_ptr, xx_ptr, zz_ptr)
   return compute, reset, update

gemm8x8 = t.decl_tensor_intrin(y.op, gemm_intrin_lower)
```

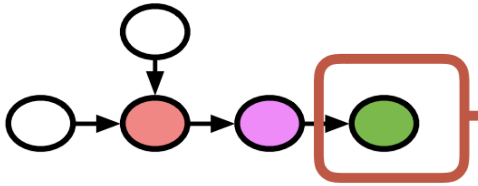
declare behavior

lowering rule to generate hardware intrinsics to carry out the computation

Tensorize: transform program to use tensor instructions



Hardware-aware Search Space



Tensor Expression Language

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

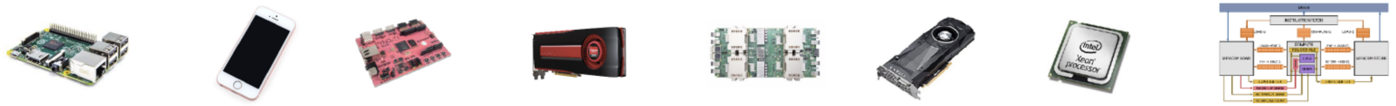
Primitives in prior work: Halide, Loopy

- Loop Transformations
- Thread Bindings
- Cache Locality

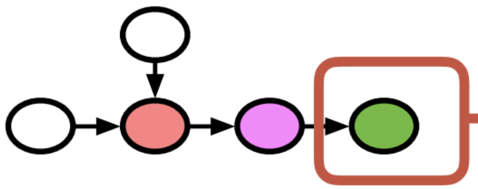
New primitives for GPUs, and enable TPU-like Accelerators

- Thread Cooperation
- Tensorization
- Latency Hiding
- ...

Hardware



Hardware-aware Search Space



Tensor Expression Language

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Billions of Possible Optimization Choices in the Search Space

Loop Transformations

Thread Bindings

Cache Locality

Thread Cooperation

Tensorization

Latency Hiding

...

Hardware

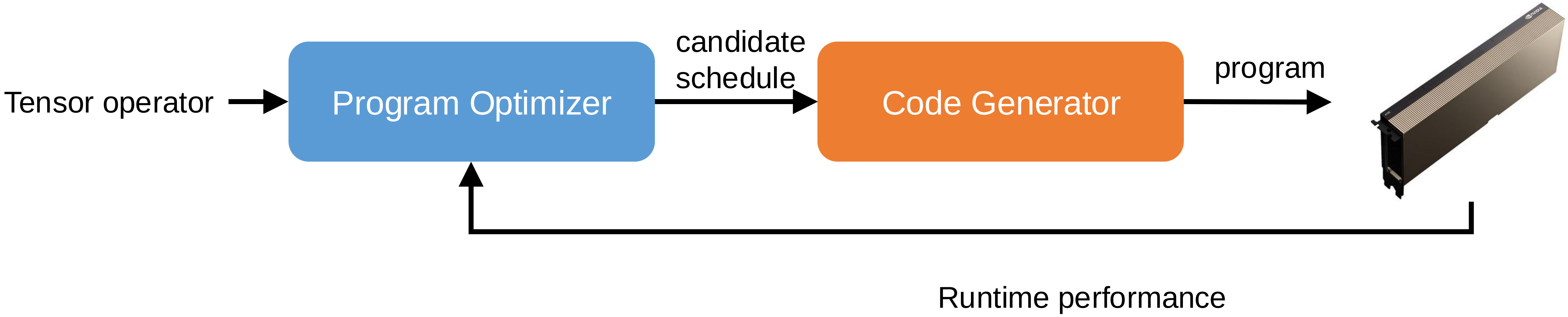
TVM: Learning-based Compiler for Deep Learning

Frameworks

Hardware-aware Search Space of Optimized Tensor Programs

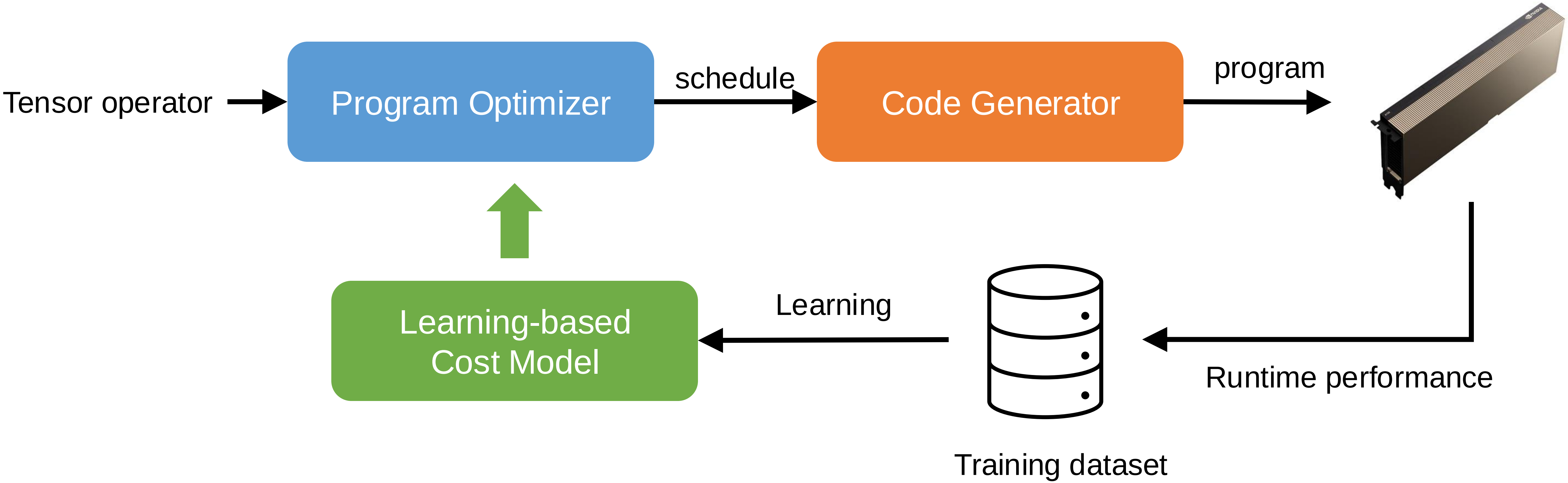
Learning based Program Optimizer

Learning-based Program Optimizer



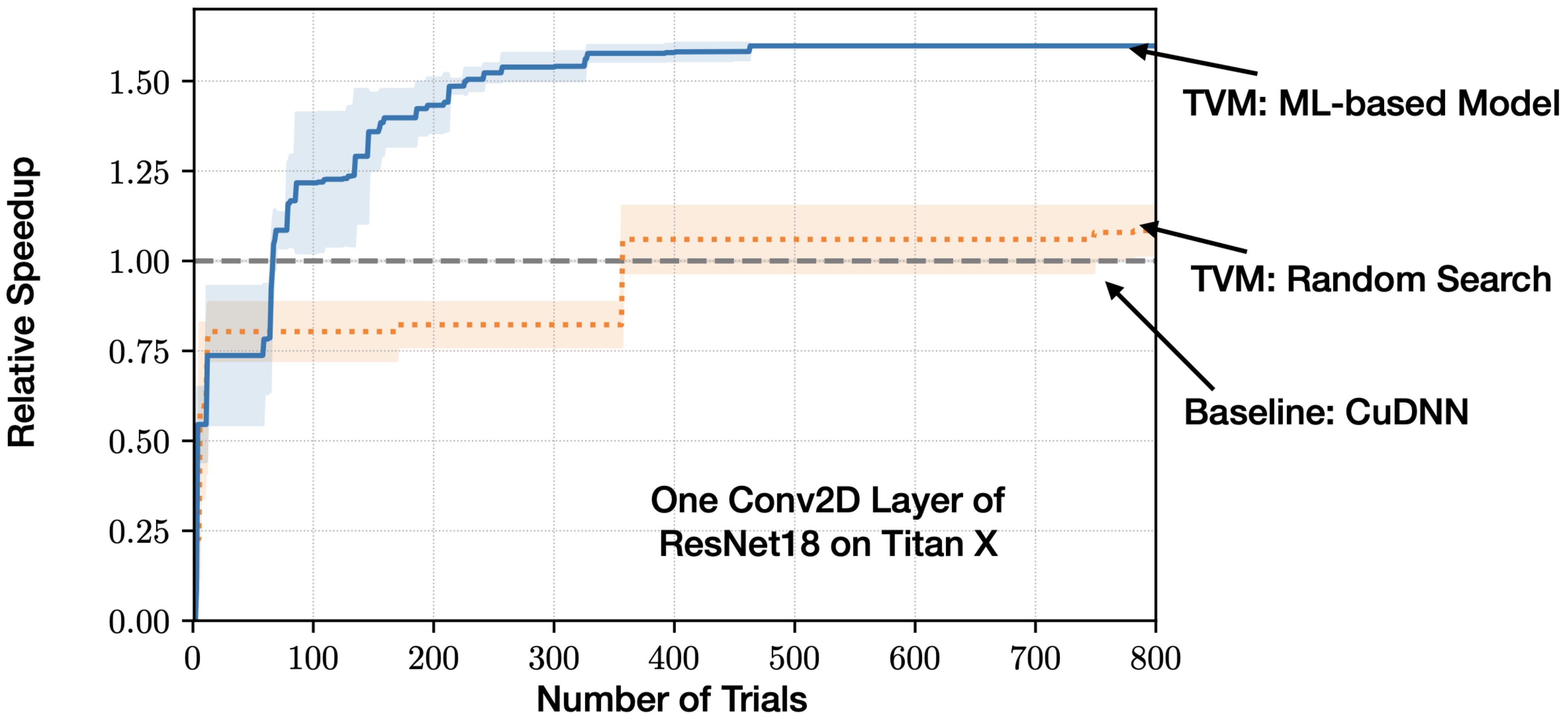
Issue: high experiment cost, each trial takes seconds

Learning-based Program Optimizer

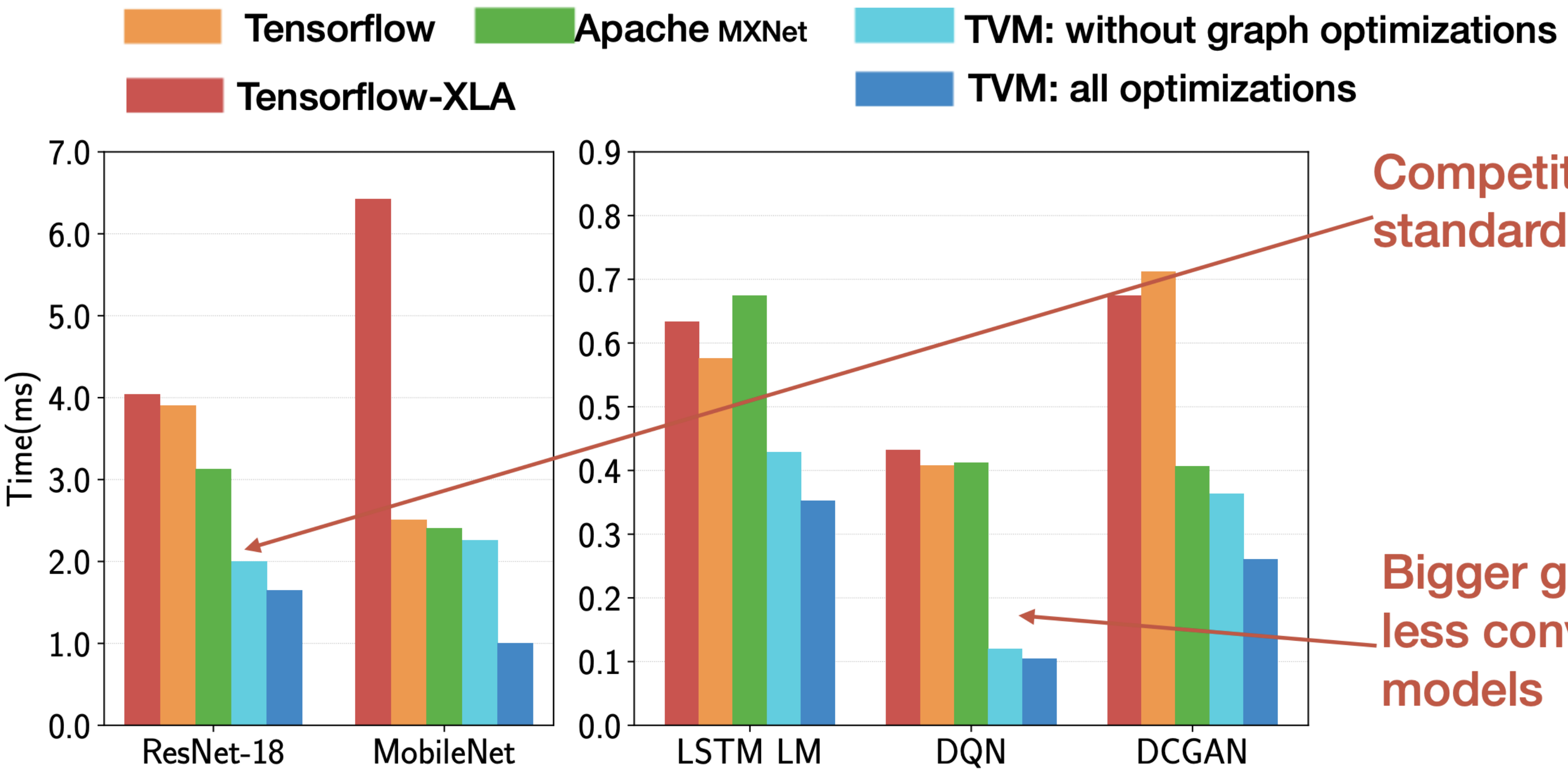


Adapt to hardware by learning, make prediction in milliseconds

Efficient ML-based Cost Model



End-to-end Inference Performance



Competitive on standard models

Bigger gap on less conventional models

Discussion: Halide and TVM

- What are the similarities?

- What are the key differences?

Summary

- Modern machines are parallel and heterogeneous
  - Only way to increase compute capability in energy-constrained world
- Most software uses small fraction of peak capability of machine
  - Challenging to tune programs to these machines
  - Tuning efforts not portable across machines
- DSLs trade-off **generality** to achieve **productivity, performance, portability**
  - Case studies: Halide, TVM