CS162
Operating Systems and Systems Programming
Lecture 24

Research Talks

April 23rd, 2018
Profs. Anthony D. Joseph & Jonathan Ragan-Kelley
http://cs162.eecs.Berkeley.edu

Anthony's Research Projects

- SecML – Secure Machine Learning in the presence of adversaries
- Big Data Genomics – Algorithms and systems programming for population-scale genomics analyses

Adversarial Exploitation of ML

- Machine Learning is everywhere
  - Search rankings, spam detection, financial models, self driving cars, …
- Adversaries everywhere! Traditional approach – *Evading Adversary*
  - Attacker determines decision boundary
  - Crafts (positive instance) content that is classified as negative
- Input data "drifts" over time, so must periodically retrain ML
  - Use previously classified input data
- Newer approach – *Influencing Adversary*
  - *Patient* attacker operates during periodic retraining stage by injecting "tricky" positive instances
  - Shifts decision boundary over time during retraining such that (positive instance) content is eventually classified as negative
- *We need novel adaptive, robust ML techniques to defend against Influencing Adversaries*
  - *Also Active Learning with Humans-in-the-Loop*

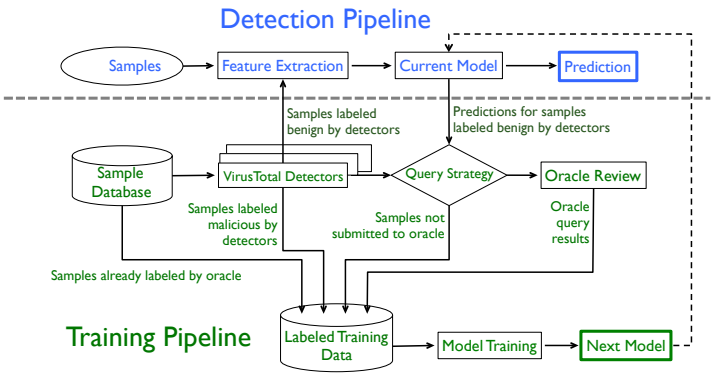
Our Current Domain and Dataset

- Windows malware detection
  - Ever changing, evading adversary with novel attacks
- Over 1 million unique executables from VirusTotal
  - Seen over 5 million times
  - Spanning January 2012 to June 2014
  - ~1% of VirusTotal's executables for the period
- Labeled by 32 AV providers' static engines
  - Threshold of four detections to label as malicious
- 85% malicious (3,000 to 406,000 families)
- *The largest academic malware detection dataset*

An ML Analysis Pipeline: From Labs to Production

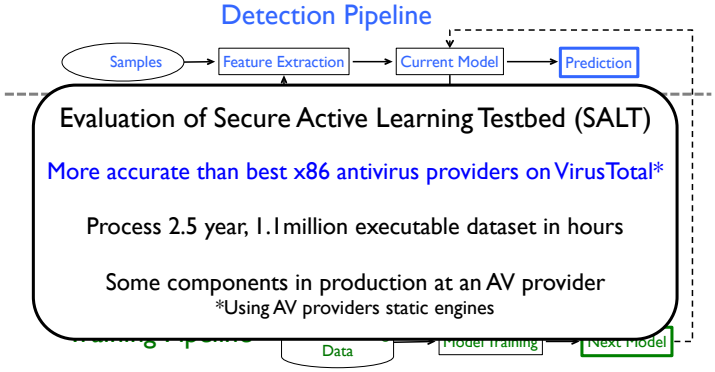
- Goal: research AND production environment
  - *– Moonshot: Detect undetectable Advanced Persistent Threats*
- Significant challenges to deploying ML in practice:
  - Increased human insight – *how the system is making decisions*
  - Scalable, interactive analysis – *large-scale data, human-driven exploration*
  - Time series analysis – *repeatable scientific experiments to build confidence in ML*
  - Resource management – *variable cost analyses and experts*
  - Adversarial resilience – *detecting attempts to manipulate or evade detection*

Secure Active Learning Testbed (SALT)



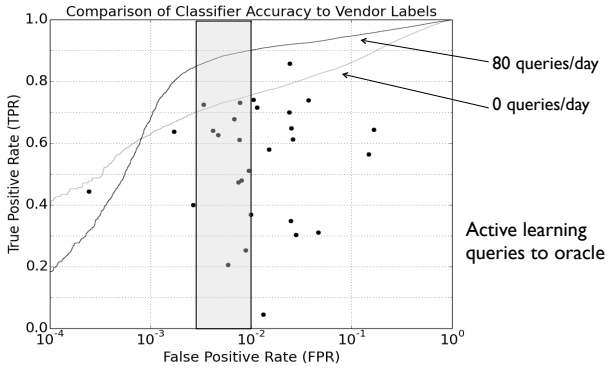
Built on Apache Spark for scalability

Secure Active Learning Testbed (SALT)

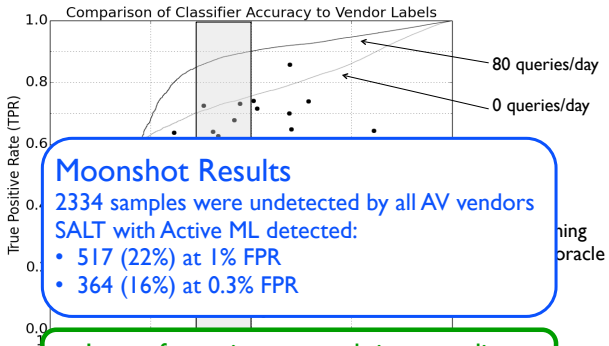


Built on Apache Spark for scalability

Active Learning Results



Active Learning Results



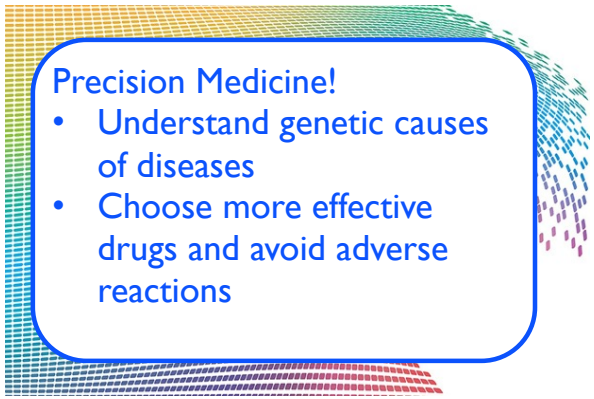
Moonshot Results
2334 samples were undetected by all AV vendors
SALT with Active ML detected:
- 517 (22%) at 1% FPR
- 364 (16%) at 0.3% FPR

Lots of ongoing research into evading and influencing adversaries

SecML Acknowledgements

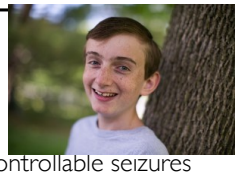
- **UC Berkeley:** David Fifield, Alex Kantchelian, Brad Miller, Qi Zhong, Vaishaal Shankar, Tony Wu, George Yu, Riyaz Faizullabhoy, Doug Tygar, Anthony D. Joseph
- **Drexel:** Eleanor Cawthon, Edwin Dauber, Rachel Greenstadt
- **ICSI:** Sadia Afroz, Michael Tshantz
- **Intel:** Ling Huang, Rekha Bachwani, Catherine Huang
- **Mitsubishi:** Takumi Yamamoto

Why Do We Need Big Genomic Data?

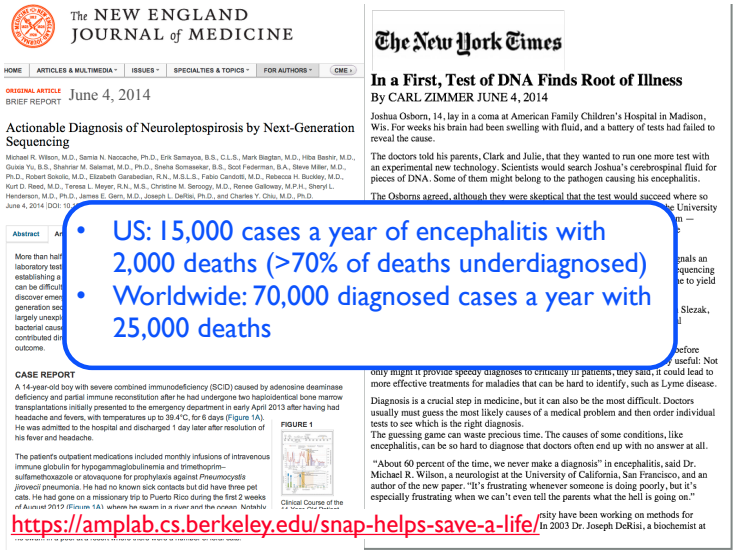


Precision Medicine!
- Understand genetic causes of diseases
- Choose more effective drugs and avoid adverse reactions

A Precision Medicine First



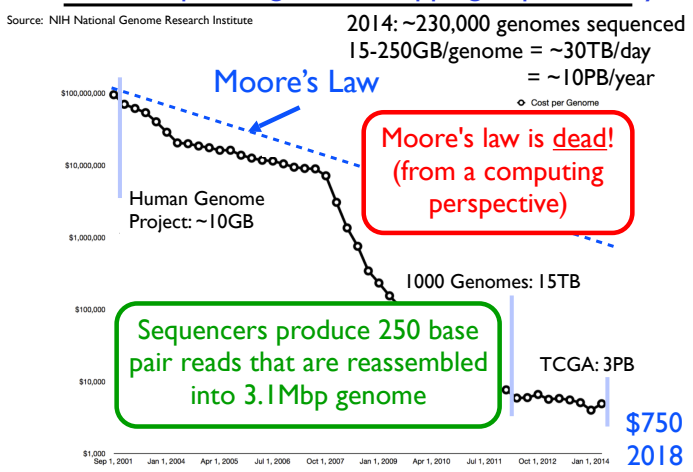
- Joshua Osborn (14 years old)
  - Visited ER 3 times in 4 months with encephalitis (brain swelling)
  - Hospitalized after 3rd visit and placed into a medically induced coma due to uncontrollable seizures
- Doctors tried traditional medicine approaches
  - Over 100 viral/fungal/bacterial pathogens cause encephalitis
  - Lots and lots of tests for different pathogens
  - 1 cm^3 of brain tissue biopsied
- Last resort – sent spinal fluid to UCSF for DNA sequencing
  - Needle in a haystack – throw out human reads (99.984%)
  - 475 reads out of 3M from rare *Leptospira santarosai* bacteria → given antibiotics, out of coma in week, discharged 4 wks



- US: 15,000 cases a year of encephalitis with 2,000 deaths (>70% of deaths underdiagnosed)
- Worldwide: 70,000 diagnosed cases a year with 25,000 deaths

https://amplab.cs.berkeley.edu/snap-helps-save-a-life/

Wet Lab Sequencing Cost Dropping Exponentially



Time to Break from Legacy Compute!

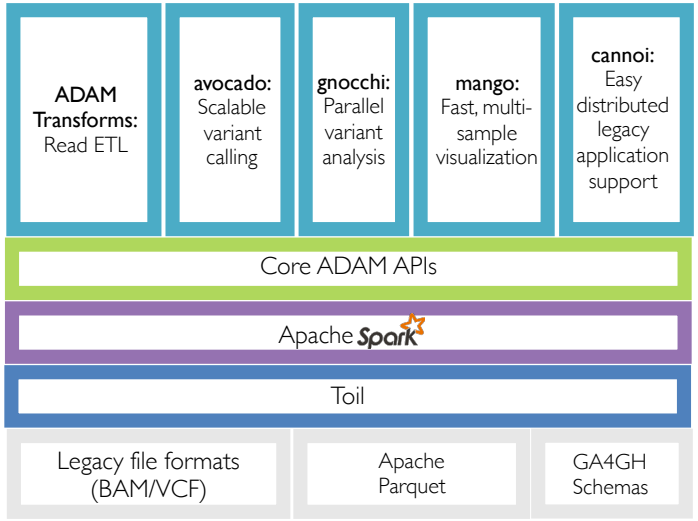
- Large cohorts generate massive amounts of data
  - 100+ GB of sequence data per sample
  - 1+ GB of variant data per sample
  - US Veterans Administration's Million Veterans Project (210 PB)
    - » Enrolled 500,000th veteran in August 2016
  - UK 100K Genome Project (21 PB)
    - » One goal is to analyze rare diseases (<1 in 2,000 people)
  - US National Institutes of Health's All of Us Project (210 PB)
    - » 1+ million ordinary Americans
- Many historical issues:
  - Processing genomic data on expensive, non-scalable HPC
  - Legacy file formats designed for single machine
  - Algorithms tied to file formats

Our Approach

- Bring together cross cutting researchers
  - We have CS, Bio, HPC backgrounds
- Leverage fact that most genomic analysis steps are parallel
- Rethink design of genomics tools for cloud/commodity computing
  - Easy to write and use horizontally scalable genomics algorithms
- Apply Open Source Software ethos
  - We build tools that people freely use and build upon
- Bring emphasis on collaboration:
  - Work closely with UCSC CGL, MSR, OHSU, Stanford, Mt. Sinai

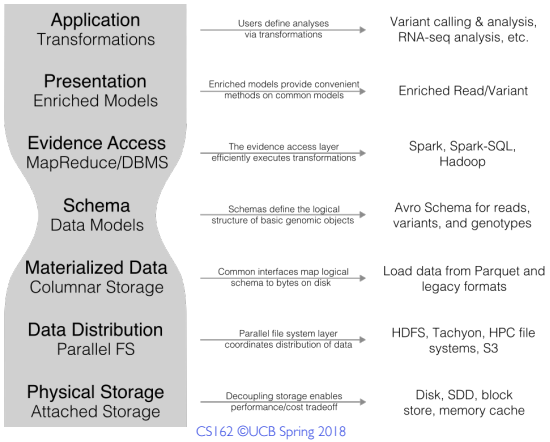
Cloud-scale Genomic Analyses

- *Going from raw genomic reads (100+ GB) to aligned genomes (3+ GB) to the differences between us (1GB) to actionable insights*
- Projects: Toil, ADAM, avocado, mango, gnocchi, cannoli
- Many algorithmic challenges:
  - Complex preprocessing and filtering methods
  - ML algorithms at genome- and population-scale
- Many infrastructure challenges:
  - Volume of data: VA MVP, NIH AoU Project
  - How do we run geo-distributed workflows efficiently?
  - Can bioinformatics be made reproducible?



ADAM: A Stack-based Design

A Well Designed Stack Simplifies Application Design



Define a Schema, Query Genomic Data via Schema

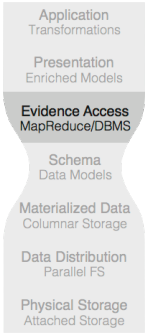
```
record AlignmentRecord {
    union { null, Contig } contig = null;
    union { null, long } start = null;
    union { null, long } end = null;
    union { null, int } mapq = null;
    union { null, string } readName = null;
    union { null, string } sequence = null;
    union { null, string } mateReference = null;
    union { null, long } mateAlignmentStart = null;
    union { null, string } cigar = null;
    union { null, string } qual = null;
    union { null, string } recordGroupName = null;
    union { int, null } basesTrimmedFromStart = 0;
    union { int, null } basesTrimmedFromEnd = 0;
    union { boolean, null } readPaired = false;
    union { boolean, null } properPair = false;
    union { boolean, null } readMapped = false;
    union { boolean, null } mateMapped = false;
    union { boolean, null } firstOfPair = false;
    union { boolean, null } secondOfPair = false;
    union { boolean, null } failedVendorQualityChecks = false;
    union { boolean, null } duplicateRead = false;
    union { boolean, null } readNegativeStrand = false;
    union { boolean, null } mateNegativeStrand = false;
    union { boolean, null } primaryAlignment = false;
    union { boolean, null } secondaryAlignment = false;
    union { boolean, null } supplementaryAlignment = false;
    union { null, string } mismatchingPositions = null;
    union { null, string } origQual = null;
    union { null, string } attributes = null;
    union { null, string } recordGroupSequencingCenter = null;
    union { null, string } recordGroupDescription = null;
    union { null, long } recordGroupRunDateEpoch = null;
    union { null, string } recordGroupFlowOrder = null;
    union { null, string } recordGroupKeySequence = null;
    union { null, string } recordGroupLibrary = null;
    union { null, int } recordGroupPredictedMedianInsertSize = null;
    union { null, string } recordGroupPlatform = null;
    union { null, string } recordGroupPlatformUnit = null;
    union { null, string } recordGroupSample = null;
    union { null, Contig } mateContig = null;
}
```

Logical data representation

Schema
Data Models

Parallelize Common Access Patterns

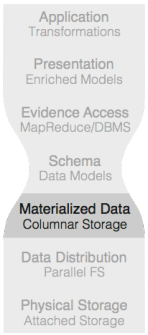
Efficient parallel access methods



- We provide parallel methods for processing genomic data with Spark
- Makes use of genomics specific reference genome structure to accelerate queries
- E.g., region join for high performance overlap queries

Pick Appropriate Storage

Efficient physical storage formats



- When accessing scientific datasets, we frequently slice and dice the dataset:
  - Algorithms may touch subsets of columns
  - We don't always touch the whole dataset
- This is a good match for columnar storage like Apache Parquet
- *Also, can support legacy formats*

Want to Learn More?

- Check out the code!
  - ADAM: https://github.com/bigdatagenomics/adam
- Check out a demo!
  - https://databricks.com/blog/2016/05/24/genome-sequencing-in-a-nutshell.html
- Run ADAM in Databricks Community Edition!
  - http://goo.gl/xK8x7s

Acknowledgements

- **UC Berkeley:** Matt Massie, Timothy Danford, André Schumacher, Jey Kottalam, Karen Feng, Eric Tu, Niranjan Kumar, Ananth Pallaseni, Michael Heuer, Alyssa Morrow, Frank A. Nothaft, Devin Petersohn, Gunjan Baid, Justin Paschall, Anthony D. Joseph, Dave Patterson
- **Mt. Sinai:** Arun Ahuja, Neal Sidhwaney, Ryan Williams, Michael Linderman, Jeff Hammerbacher
- **GenomeBridge:** Carl Yeksigian
- **Cloudera:** Uri Laserson, Tom White
- **Microsoft Research:** Ravi Pandya, Bill Bolosky
- **UC Santa Cruz:** Benedict Paten, David Haussler, Hannes Schmidt, Beau Norgeot, Audrey Musselman-Brown, John Vivian, Jacob Pfeil
- And many other open source contributors, especially Neil Ferguson, Andy Petrella, Xavier Tordior
- Total of >60 contributors to ADAM/BDG from >12 institutions

Administrivia

- Midterm 3 coming up on **Wed 4/25 6:30-8PM**
  - All topics up to and including Lecture 23
    - » Focus will be on Lectures 17 – 23 and associated readings, and Projects 3
    - » But expect 20-30% questions from materials from Lectures 1-16
  - LKS 245, Hearst Field Annex A1, VLSB 2060, Barrows 20, Wurster 102 (see Piazza for your room assignment)
  - Closed book
  - 2 pages hand-written notes both sides
- Please fill out the online course evaluation

BREAK

Systems & Compilers for Graphics and Visual Computing

Jonathan Ragan-Kelley

Visual computing demands orders of magnitude more performance

Rendering:
insatiable demand for computation

Modern game
2 Mpixels
1 Mpolys
10 ms/frame

10 ms → 5 hrs

Tintin, Avatar
8 Mpixels
5 Gpolys
5 hrs/frame

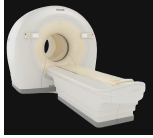
6 orders of magnitude more computation

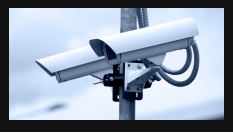
images by Valve, Weta

The biggest data is visual

YouTube: 400 hrs uploaded / min
[Brewer et al. 2016]
1.5 Terapixels/sec

250 M surveillance cameras,
2.5 B cell phone cameras, …





Pervasive sensing: "the cloud" is not enough
data transfer ≫ capture

Sensor + Read out
5 Mpixels
~1 mJ/frame

transmit

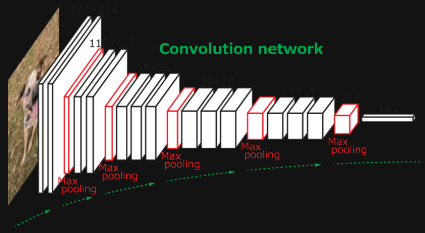
LTE radio
50 Mbit/sec
1 W
~1 J/frame

[Wu et al. 2012]

transmission power costs 1,000x capture

Visual data analysis is expensive

One object recognition neural net:
250 Watt GPU → 0.05 megapixels at video rate



(Caffe + cuDNN on K40c running VGG16)

[Simonyan & Zisserman 2015]
image: Noh et al. 2015

Your data-intensive problem here…

Programmer productivity has exploded

1990s → **2010s**

C/C++

node.js

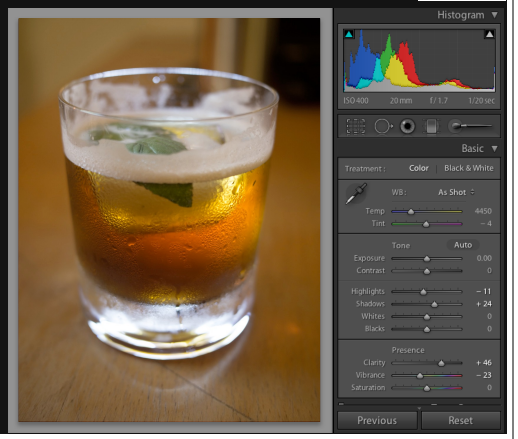
amazon web services™

Writing high-performance code is hard

Adobe

Reference:
300 lines C++

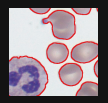
Adobe: 1500 lines
3 months of work
10x faster



My group's research:

Compilers, systems, architectures, and algorithms for high-performance graphics & visual computing.



Reorganize computations & data.

- Simpler programs
- Order of magnitude faster
- Scalable on future architectures

applications
algorithms
data structures
languages
compilers
hardware

How can we increase performance and efficiency?

Parallelism

"Moore's law" scaling requires exponentially more parallelism.

Locality

Data should move as little as possible.

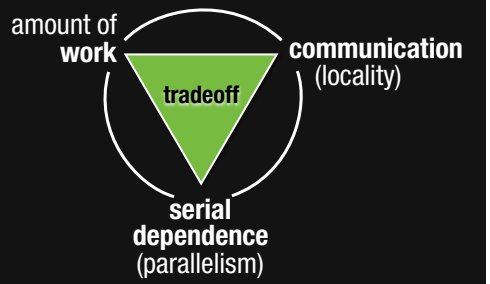
Communication dominates computation

in both energy and time

Operation (32-bit operands)	**Energy/Op** (28 nm)	***Cost*** *(vs. ALU)*
ALU op	1 pJ	-
Load from SRAM	5 pJ	*5x*
Move 10mm on-chip	32 pJ	*32x*
Send off-chip	500 pJ	*500x*
Send to DRAM	1 nJ	*1,000x*
Send over LTE	> 50 µJ	*50,000,000x*

data from John Brunhaver, Bill Dally, Mark Horowitz

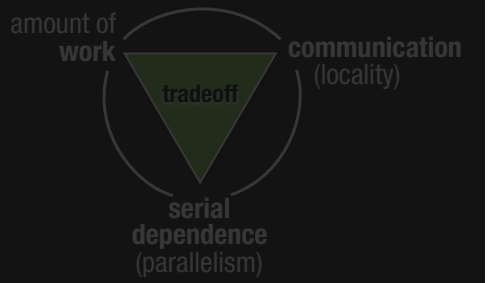
Message #1: Performance requires complex tradeoffs



Where does performance come from?

Program

Hardware



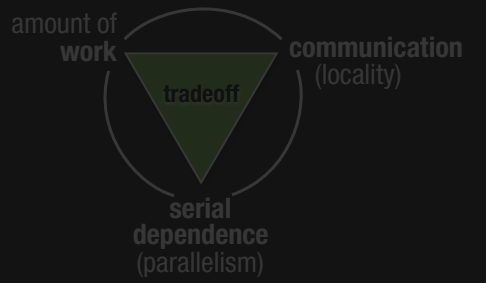
Message #2: organization of computation is a first-class issue

Program:

Algorithm

Organization of computation

Hardware



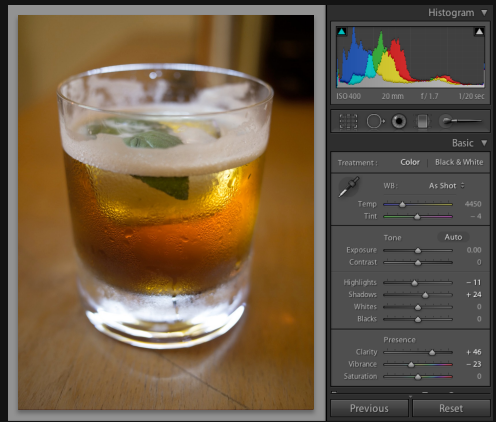
Reorganizing computation is painful

Adobe

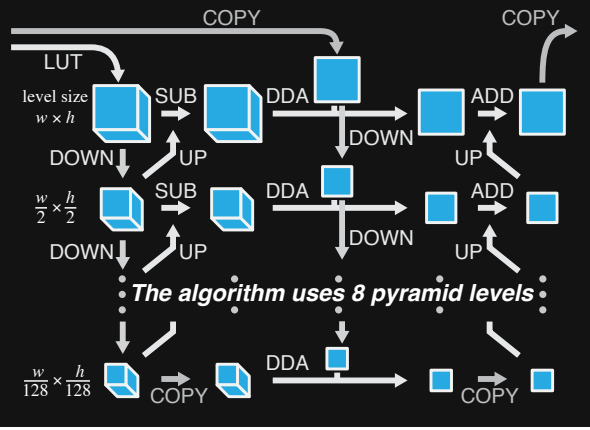
Reference:
300 lines C++

Adobe: 1500 lines
3 months of work
10x faster (vs. reference)

Same algorithm, Different organization



Global reorganization breaks modularity



Halide

a language and compiler
for image processing & vision

[SIGGRAPH 2012,
PLDI 2013,
SIGGRAPH 2016,
SIGGRAPH 2018, …]

Algorithm vs. Organization: 3x3 blur

```
for (int x = 0; x < input.width(); x++)
  for (int y = 0; y < input.height(); y++)
    blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;

for (int x = 0; x < input.width(); x++)
  for (int y = 0; y < input.height(); y++)
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

Algorithm vs. Organization: 3x3 blur

```
for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;

for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

Same algorithm, different organization
One of them is 15x faster

Algorithm vs. Organization: 3x3 blur

```
for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;

for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

Hand-optimized C++

9.9 → 0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blury) {
  __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 blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        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(blurxPtr++, avg);
         inPtr += 8;
        }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

11x faster
(quad core x86)

Tiled, fused

Vectorized

Multithreaded

Redundant computation

Near roof-line optimum

Traditional languages **conflate algorithm** & **organization**

```
void box_filter_3x3(const Image &in, Image &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        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(blurHPtr++, avg);
         inPtr += 8;
        }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

```
void box_filter_3x3(const Image &in, Image &blurV) {
  Image blurH(in.width(), in.height());  // allocate blurH array

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
}
```

not readable

architecture-specific

hard to change organization *or* algorithm

Optimized 3x3 blur in C++

```
void box_filter_3x3(const Image &in, Image &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        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(blurHPtr++, avg);
         inPtr += 8;
        }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

parallelism

distribute across threads

SIMD parallel vectors

Optimized 3x3 blur in C++

```
void box_filter_3x3(const Image &in, Image &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        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(blurHPtr++, avg);
         inPtr += 8;
        }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

parallelism

distribute across threads

SIMD parallel vectors

locality

reorganize computation:

fuse two blurs,

compute in tiles

(Re)organizing computation is hard

Optimizing parallelism, locality requires
transforming program & data structure.

What transformations are *legal?*

What transformations are *beneficial?*

libraries don't solve this:
BLAS, MKL, OpenCV, PyTorch, TensorFlow
optimized kernels compose into inefficient pipelines (no fusion)

Halide's answer:
decouple algorithm from schedule

Algorithm: *what* is computed
Schedule: *where* and ***when*** it's computed

The algorithm defines pipelines as pure functions

Pipeline stages are functions from coordinates to values

Execution order and storage are unspecified
no explicit loops or arrays

3x3 blur as a Halide algorithm:

```
blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

Domain scope
of the programming model

All computation is over **regular grids**.

not Turing complete

- Only **feed-forward pipelines**
  Iterative computations are a (partial) escape hatch.
- **Iteration** must have **bounded depth**.

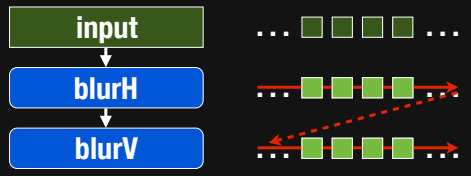
Dependence must be inferable.
User-defined clamping can impose tight bounds, when needed.

Long, heterogeneous pipelines.
Complex graphs, deeper than traditional stencil computations.

How can we organize this computation?

Organizing a data-parallel pipeline

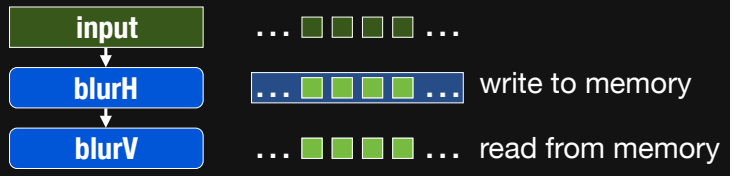
Simple loops execute **breadth-first** across stages

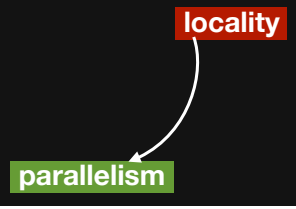


Simple loops execute **breadth-first** across stages

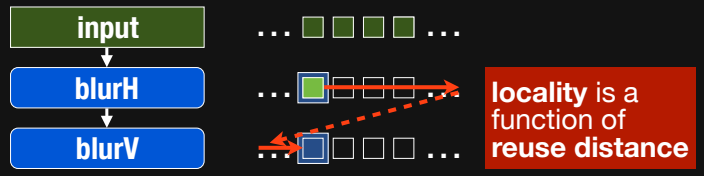
parallelism

Breadth-first execution **sacrifices locality**



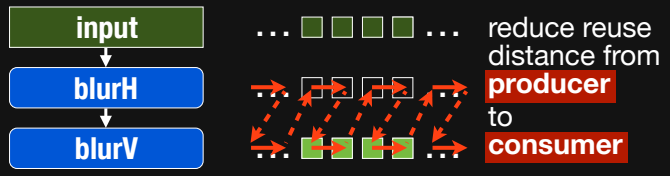


Breadth-first execution **sacrifices locality**



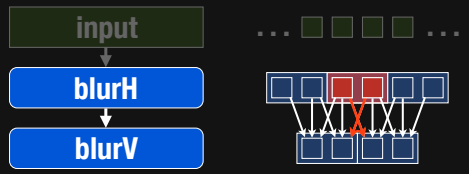
Interleaved execution (fusion) **improves locality**

*fusion **globally interleaves** computation*

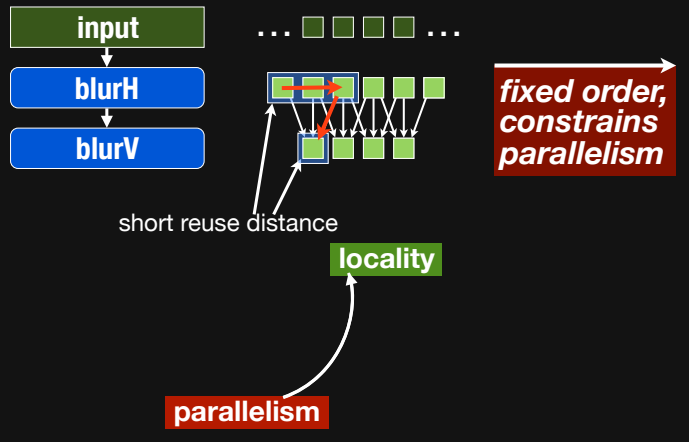


Understanding dependencies

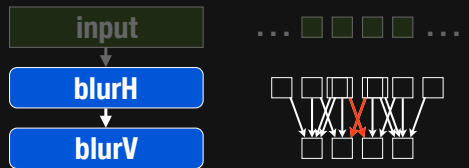
Stencils have overlapping dependencies



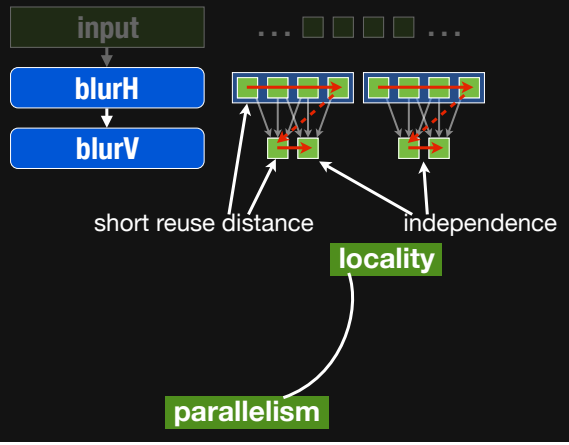
Sliding window execution **sacrifices parallelism**



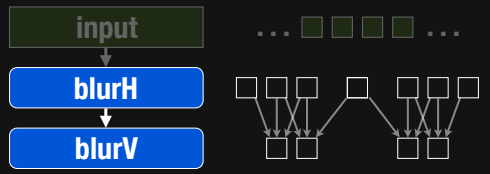
Breaking dependencies with tiling



Decoupled tiles optimize **parallelism & locality**



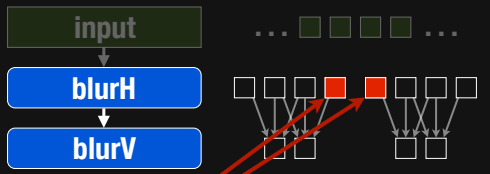
Breaking dependencies introduces **redundant work**



locality

parallelism

Breaking dependencies introduces **redundant work**

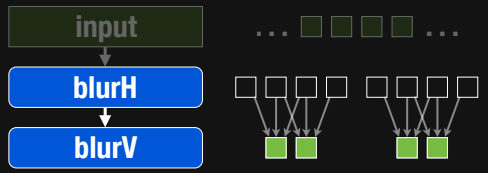


redundant work

locality

parallelism

Message #1: performance requires **tradeoffs**



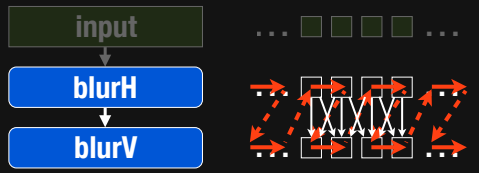
trade off with *granularity of fusion*

redundant work

locality

parallelism
trade off by *constraining order*

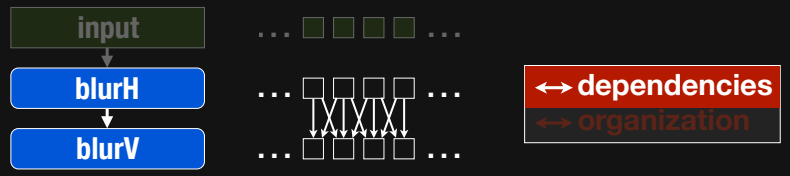
Message #2: algorithm vs. **organization**



↔ algorithm
↔ organization

order and interleaving
radically alter performance
of the *same algorithm*

Dependencies limit choices of organization



A language of schedules

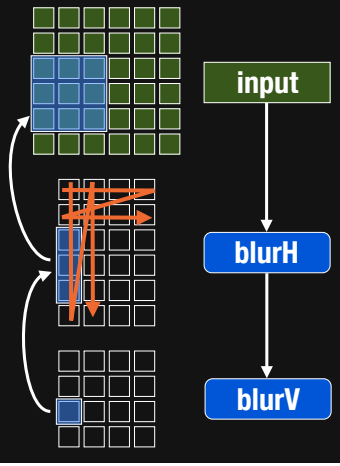
The schedule defines intra-stage order, inter-stage interleaving

For each stage:

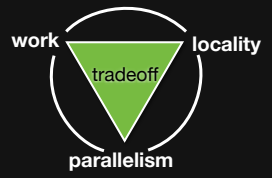
1) In **what order** should we **compute** its **values**?

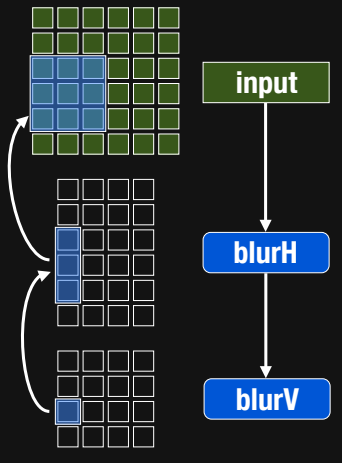
2) **When** should we **compute** its **inputs**?

This is a **language** for scheduling choices.



The schedule defines intra-stage order, inter-stage interleaving





Schedule primitives **compose** to create many organizations

Schedule primitives **compose** to create many organizations

```
blurH.compute_at(blurV, x)
      .vectorize(x, 4)

blurV.tile(x, y, xi, yi, 8, 8)
      .parallel(y)
      .vectorize(xi, 4)
```

redundant work / locality / parallelism

```
void box_filter_3x3(const Image &in, Image &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        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(blurHPtr++, avg);
          inPtr += 8;
        }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Halide

0.9 ms/megapixel

```
Func box_filter_3x3(Func in) {
  Func blurH, blurV;
  Var x, y, xi, yi;

  // The algorithm - no storage, order
  blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
  blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blurV.tile(x, y, xi, yi, 256, 32)
       .vectorize(xi, 8).parallel(y);
  blurH.compute_at(blurV, x).store_at(blurV, x).vectorize(x, 8);

  return blurV;
}
```

Halide

0.9 ms/megapixel

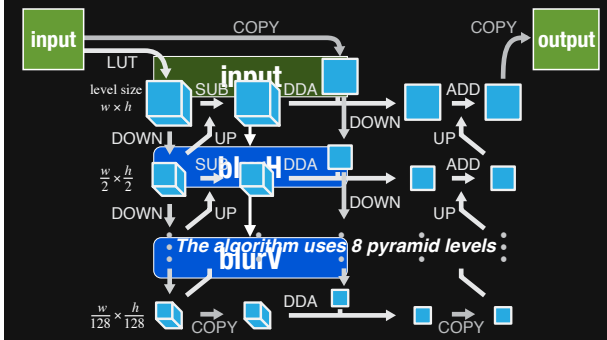
```
Func box_filter_3x3(Func in) {
  Func blurH, blurV;
  Var x, y, xi, yi;

  // The algorithm - no storage, order
  blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
  blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blurV.tile(x, y, xi, yi, 256, 32)
       .vectorize(xi, 8).parallel(y);
  blurH.compute_at(blurV, x).store_at(blurV, x).vectorize(x, 8);

  return blurV;
}
```

C++

0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        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(blurHPtr++, avg);
          inPtr += 8;
        }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Organization requires global tradeoffs



LUT: look-up table
$O(x,y,k) \leftarrow \text{lut}(I(x,y) - k\sigma)$

ADD: addition
$O(x,y) \leftarrow I_1(x,y) + I_2(x,y)$

SUB: subtraction
$O(x,y) \leftarrow I_1(x,y) - I_2(x,y)$

DDA: data-dependent access
$k \leftarrow \text{floor}(I_1(x,y) / \sigma)$
$\alpha \leftarrow (I_1(x,y) / \sigma) - k$
$O(x,y) \leftarrow (1-\alpha)\, I_2(x,y,k) + \alpha\, I_2(x,y,k+1)$

UP: upsample
$T_1(2x,2y) \leftarrow I(x,y)$
$T_2 \leftarrow T_1 \otimes_x [1\ 3\ 3\ 1]$
$O \leftarrow T_2 \otimes_y [1\ 3\ 3\ 1]$

DOWN: downsample
$T_1 \leftarrow I \otimes_x [1\ 3\ 3\ 1]$
$T_2 \leftarrow T_1 \otimes_y [1\ 3\ 3\ 1]$
$O(x,y) \leftarrow T_2(2x,2y)$

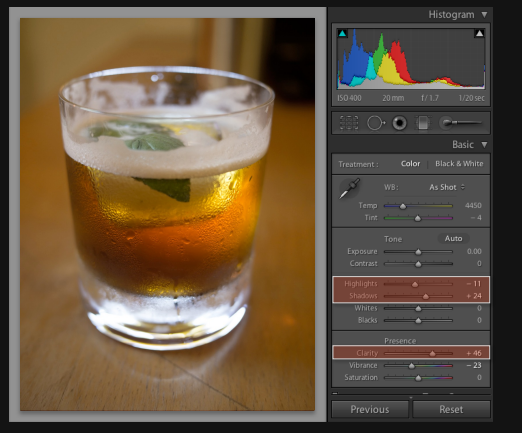
local Laplacian filters
3x3 box filter
[Paris et al. 2010, Adobe et al. 2011]

Adobe: 1500 lines
expert-tuned C++
multi-threaded, SSE
3 months of work
10x faster than original C++

Halide: 60 lines
1 day

Halide vs. Adobe:
2x faster on same **CPU**
10x faster** on **GPU

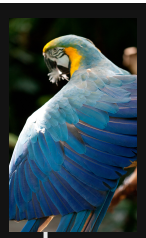
Local Laplacian Filters
in Adobe Photoshop

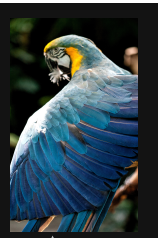


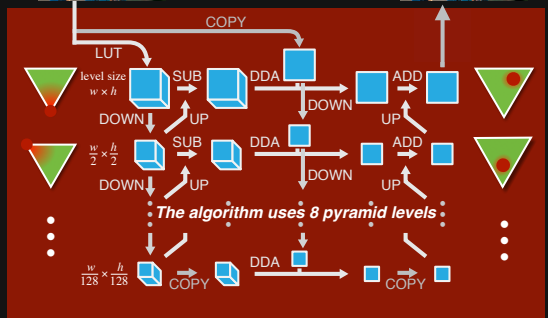
Adobe: 1500 lines
expert-tuned C++
multi-threaded, SSE
3 months of work
10x faster than original C++

Halide: 60 lines
1 day

Halide vs. Adobe:
2x faster on same **CPU**
10x faster** on **GPU







Halide is all around you!

open source at http://halide-lang.org

Adobe

Google
> 1000 pipelines
10s of kLOC in production

Microsoft Cognitive Toolkit

facebook

PYTORCH

Tensor Comprehensions

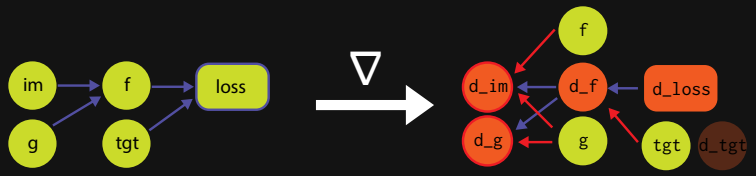
intel

YouTube

What's next?

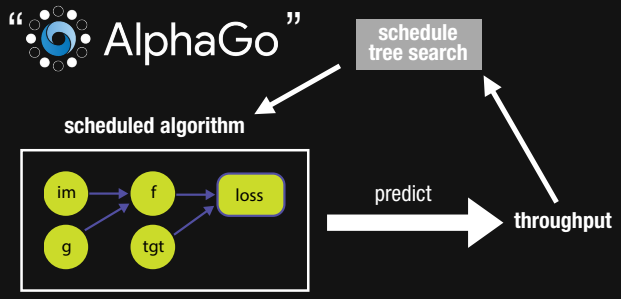
What's next?

Differentiable programming languages:
bringing gradient-based optimization
to arbitrary programs.



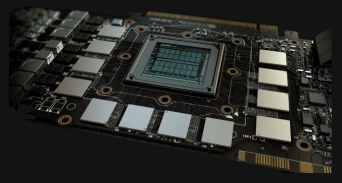
What's next?

Training compilers to **automatically optimize organization** using **reinforcement learning**.



What's next?

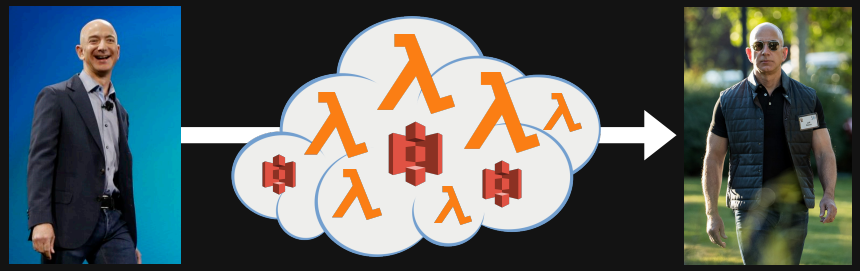
Building the **"programmable GPU"**
of image processing & vision.



RISC-V

What's next?

Turning **AWS Lambda** into a **supercomputer** accessible from your laptop



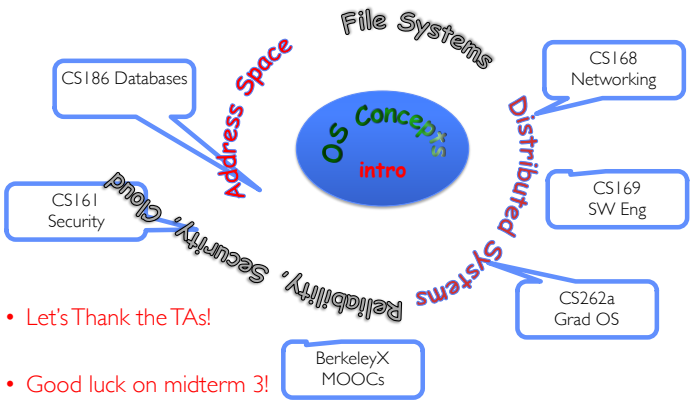
Where you can learn more:

CS 164 - Compilers & Programming Languages

CS 294-jrk - Domain-Specific Languages

Talk to me about research!

Thank you!



- Let's Thank the TAs!
- Good luck on midterm 3!