Performance

Will Crichton

CS 242 – 11/26/18

Key questions of performance

1. What programs need to be efficient?

2. How do we know if programs are efficient?

3. How can we make programs efficient?

Resource limits in 1975

Apple II

- 32 KB of memory
- 1 MHz CPU
- 100 KB floppy disks
- $5000+ (w/ inflation)

The average programmer *always* has to care about performance.

Extreme end: game development

Games were mostly engineered in a "data-oriented" way out of sheer necessity. There is not much room for abstraction when your target console has 128KB of RAM (SNES).

Every byte of storage is precious, so mostly games from this era are designed with very predictable manually managed layouts for their entire game state in memory. In the NES / SNES era, there was so little memory that generally the graphical representation of your game (tiles, sprites) and the logical representation of your game are the same,

In Mario 64, the entity structures are all exactly 608 bytes long, and there is a hard limit to 240 of them.

Kyren, RustConf 2018 keynote

Resource limits today

- 3 GHz processor
- 32+ GB of RAM
- GPU (1+ *tera*flop/s)
- 1 *tera*byte of disk

The average programmer *rarely* has to care about performance.

PL paradigms throughout history

- 1975: rise of systems languages
  - Efficiency first: we don't have enough resources, carefully build a system
- 1995: rise of scripting languages
  - Productivity first: systems languages are too hard (and we have the resources), so quickly glue together a system
- 2015: rise of functional languages
  - Correctness first: scripting languages are too buggy, we need to know if our system works

Latency: visual applications

Andy Matuschak
@andy_matuschak

Follow

Replying to @rsnous @danluu

100%! I was part of a project trying to cut one frame of latency out of iOS's touch response. Cutting just a few ms down required changes top-to-bottom in the OS!

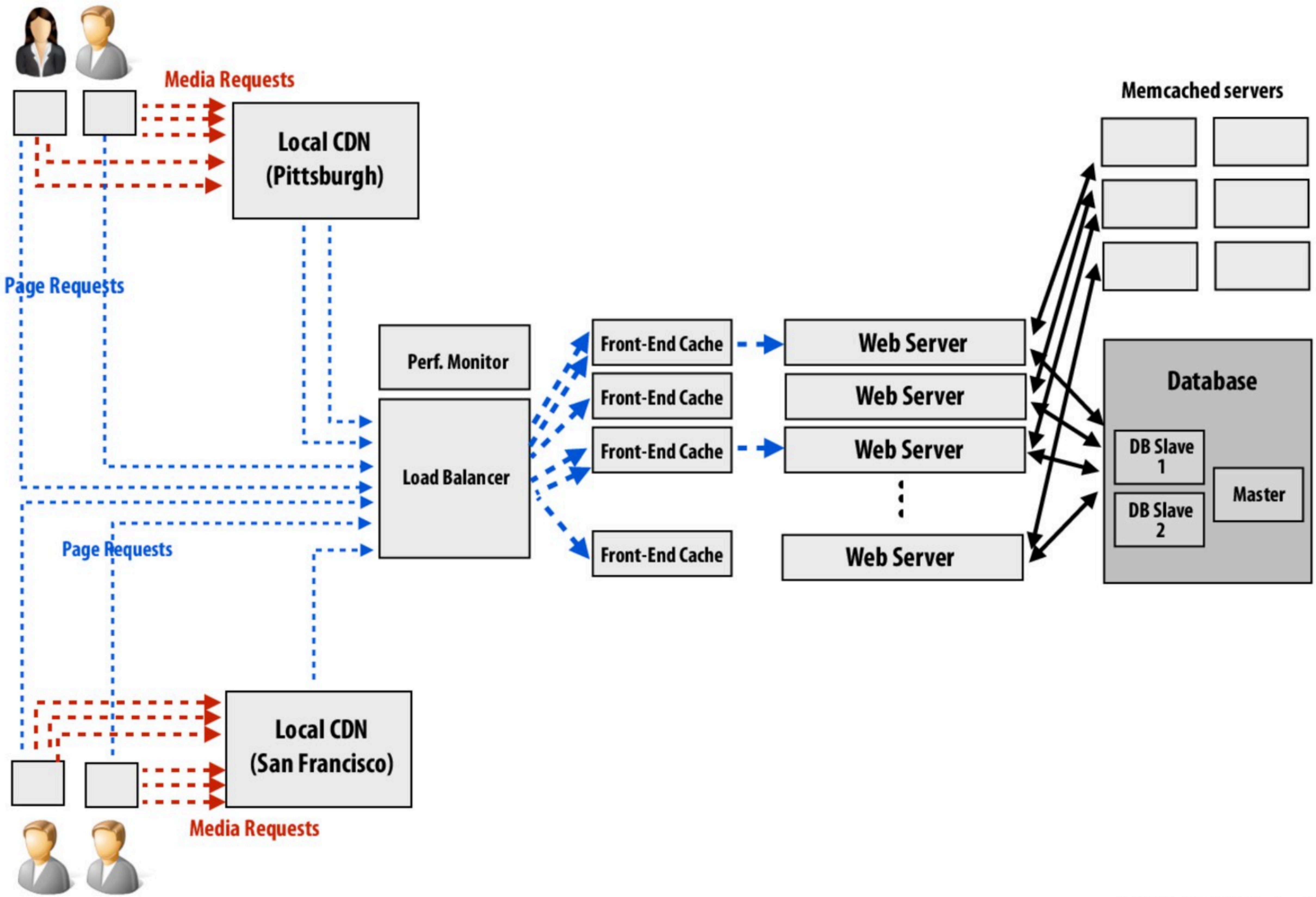
Alt story: iOS touch code had a fun rule—zero dynamic allocations allowed between touch HW event and app code!

3:53 PM - 31 Aug 2018

4 Retweets **20** Likes

3 4 20

Latency: data transfer



Latency: media processing

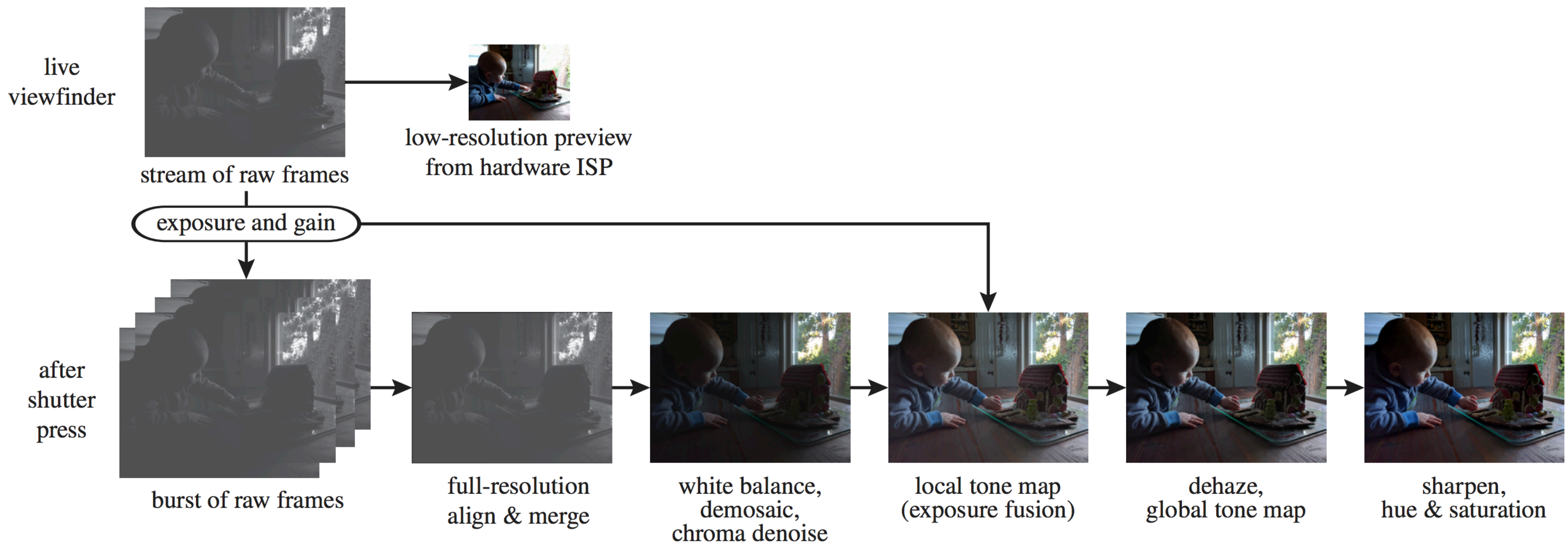
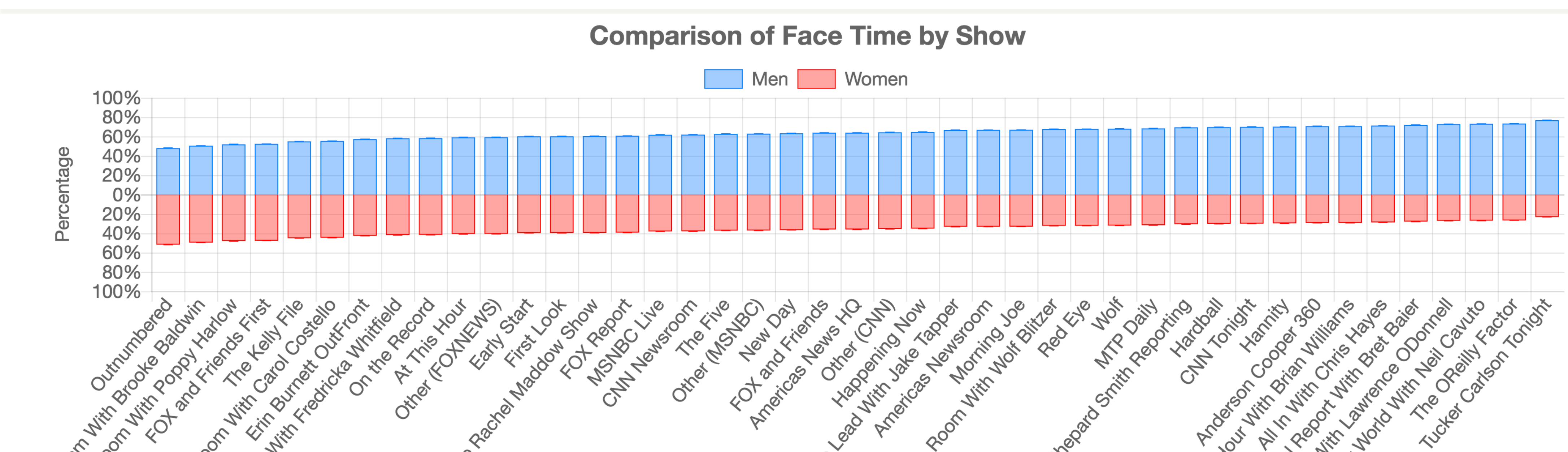


Figure 3: *Overview of our two processing pipelines. The input to both pipelines is a stream of Bayer mosaic (raw) images at full sensor resolution (for example, 12 Mpix) at up to 30 frames per second. When the camera app is launched, only the viewfinder (top row) is active. This pipeline converts raw images into low-resolution images for display on the mobile device's screen, possibly at a lower frame rate. In our current implementation the viewfinder is 1.6 Mpix and is updated at 15–30 frames per second. When the shutter is pressed, this pipeline suspends briefly, a burst of frames is captured at constant exposure, stored temporarily in main memory, and the software pipeline (bottom row) is activated. This pipeline aligns and merges the frames in the burst (sections 4 and 5), producing a single intermediate image of high bit depth, then applies color and tone mapping (section 6) to produce a single full-resolution 8-bit output photograph for compression and storage in flash memory. In our implementation this photograph is 12 Mpix and is computed in about 4 seconds on the mobile device.*

Hasinoff et al. "Burst photography for high dynamic range and low-light imaging on mobile cameras". SIGGRAPH 2016

Throughput: big data analytics

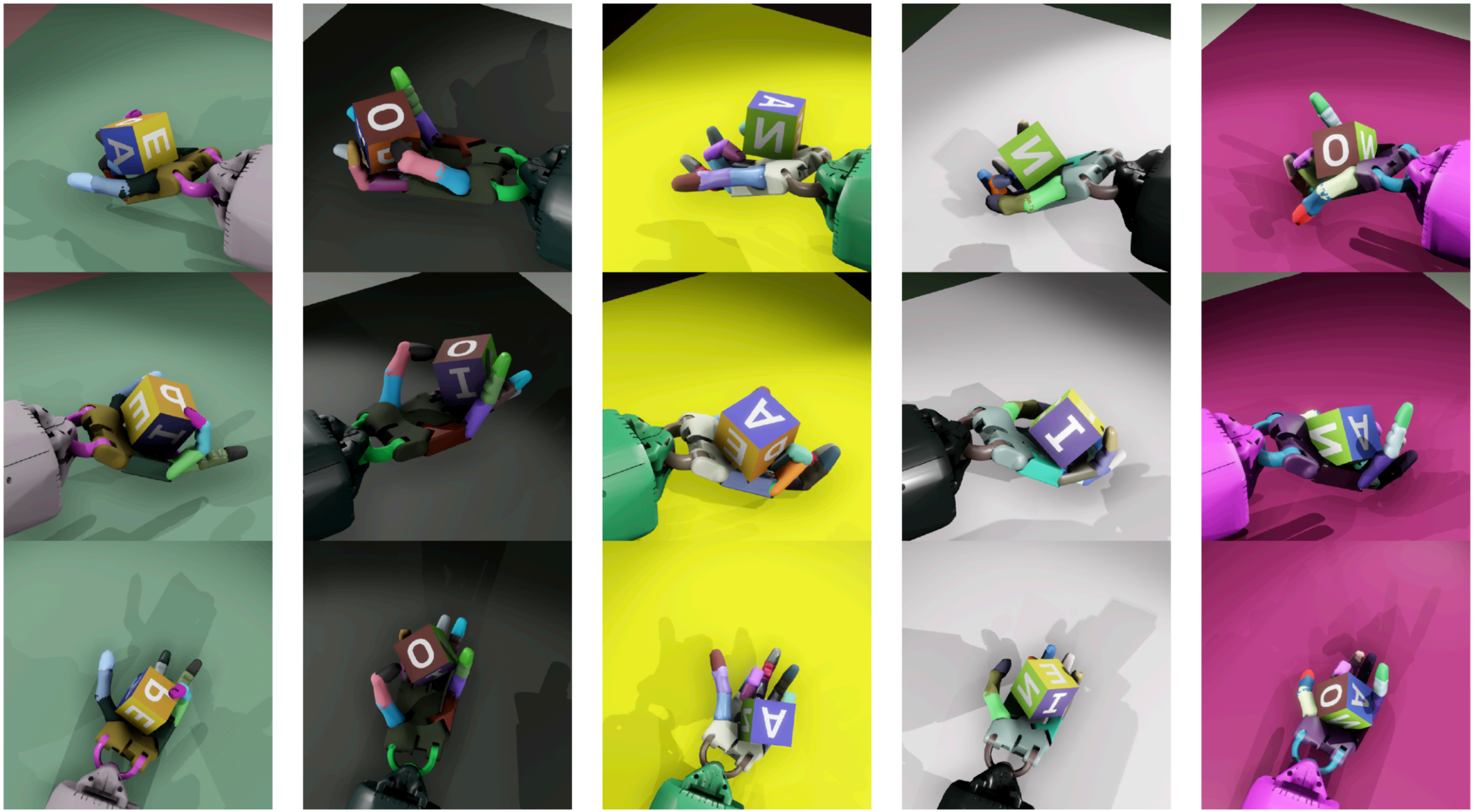


Huge perf gap for most programs

	Immutable	Mutable	Double Only	No Objects	In C	Transposed	Tiled	Vectorized	BLAS MxM	BLAS Parallel
ms	17,094,152	77,826	32,800	15,306	7,530	2,275	1,388	511	196	58
	219.7x			2.2x		3.4x		2.8x		3.5x
			2.4x		2.1x		1.7x		2.7x	
	219.7x									
	522x									
		1117x								
			2271x							
			7514x							
				12316x						
				33453x						
					87042x					
					296260x					
Cycles/OP	8,358	38	16	7	4	1	1/2	1/5	1/11	1/36

Saman Amarasinghe, MIT 6.172 “Performance Engineering”. 2009

Throughput: simulation



OpenAI. "Learning Dexterity". 2018

Resource usage: memory

Process Name	Memory
Slack Helper	365.8 MB
Slack Helper	234.9 MB
Slack	91.8 MB
Slack Helper	89.8 MB

700 MB!

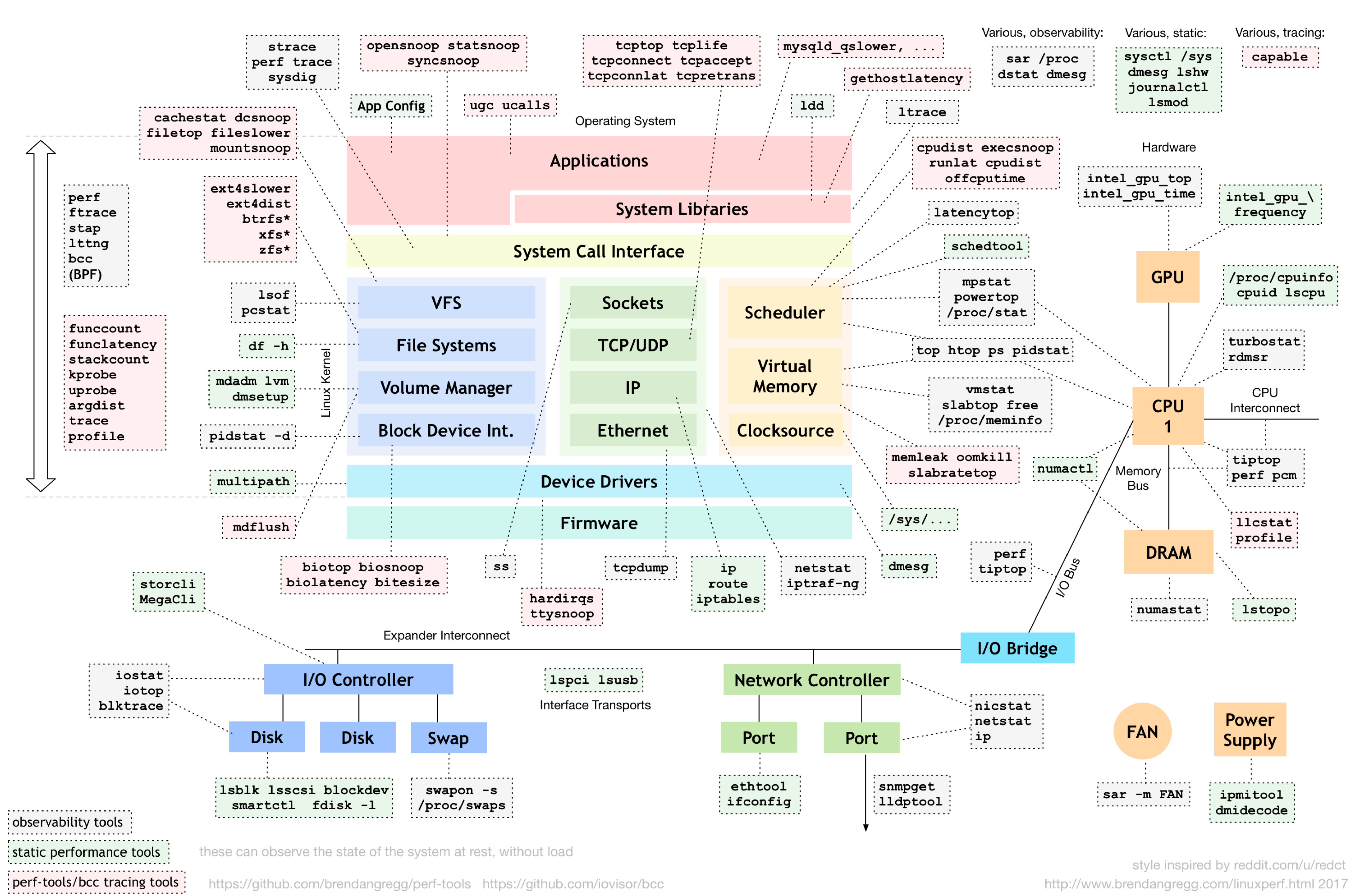
Process Name	Memory	Co
Google Chrome Helper	738.8 MB	
Google Chrome Helper	419.8 MB	
Google Chrome Helper	399.4 MB	
Google Chrome Helper	366.4 MB	
Google Chrome	317.8 MB	
Google Chrome Helper	57.2 MB	
Google Chrome Helper	54.9 MB	
Google Chrome Helper	53.5 MB	
Google Chrome Helper	45.2 MB	
Google Chrome Helper	44.5 MB	
Google Chrome Helper	44.0 MB	
Google Chrome Helper	43.8 MB	
Google Chrome Helper	43.6 MB	
Google Chrome Helper	43.5 MB	
Google Chrome Helper	43.3 MB	
Google Chrome Helper	43.2 MB	
Google Chrome Helper	43.1 MB	
Google Chrome Helper	42.8 MB	
Google Chrome Helper	40.0 MB	
Google Chrome Helper	30.1 MB	
Google Chrome Helper	18.8 MB	
Google Chrome Helper	18.4 MB	
Google Chrome Helper	17.3 MB	

3 GB!!!

Performance profiling is a dark art

- Perf, cProfile, etc. are rarely taught in school
- Debuggers are great, but a lot of profiling can be pretty close to "printf debugging"
- A lot more work to do here!
- (Claim: "Profiling", "Debugging" should be required courses in any CS curriculum.)

Linux Performance Tools



How to improve program performance:

1. Change the program
2. Change the programmer

Automatic optimization

- Promise of automatic optimizers: don't worry about performance, we'll take care of everything
- Type checks, memory management, parallelization
- Issue is <u>consistency</u>

Algebraic simplification

- Some statements can be deleted

  x := x + 0

  x := x * 1

- Some statements can be simplified

  x := x * 0 ⇒ x := 0

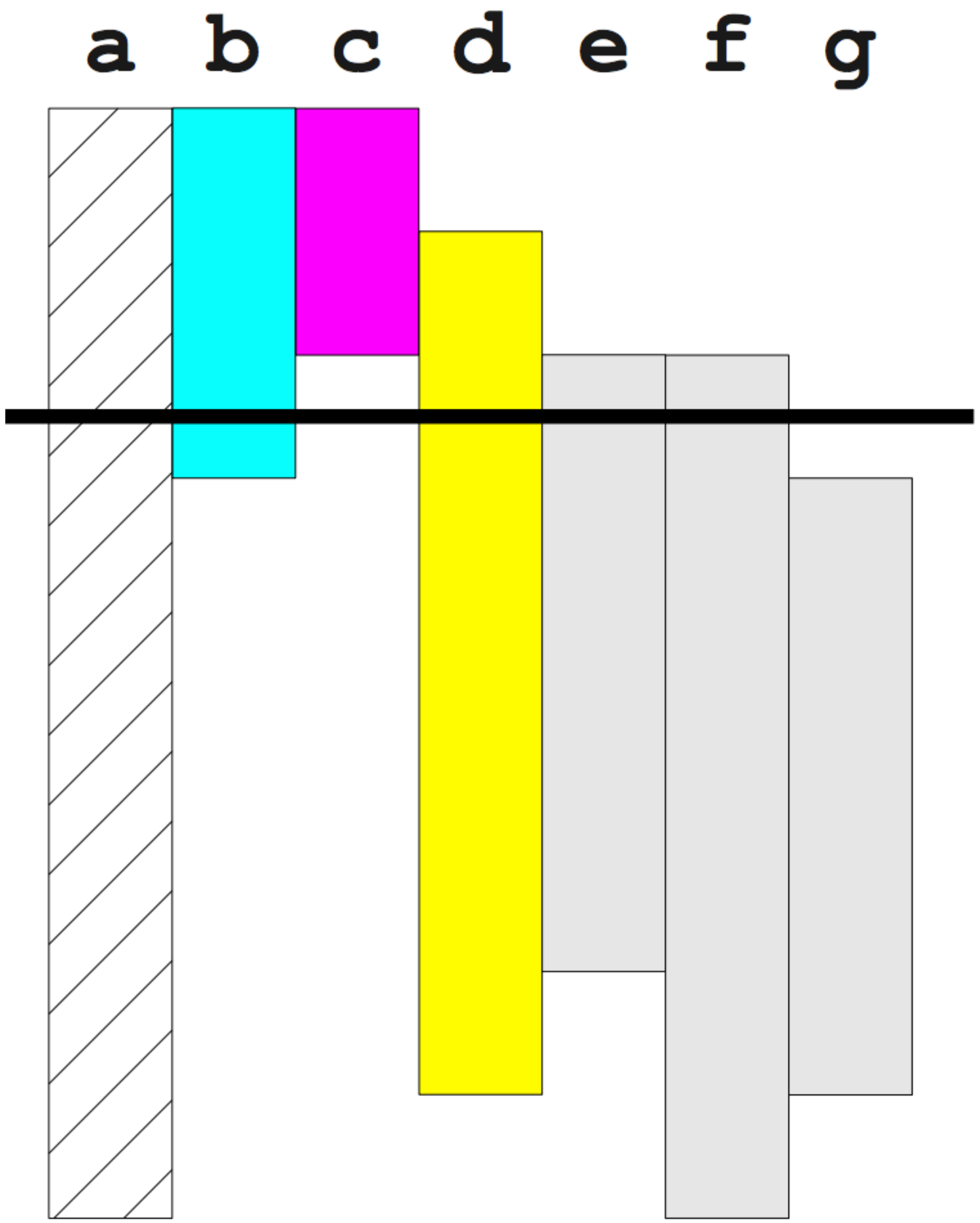
  y := y ** 2 ⇒ y := y * y

  x := x * 8 ⇒ x := x << 3

  x := x * 15 ⇒ t := x << 4; x := t - x

  (on some machines << is faster than *; but not on all!)

Register allocation

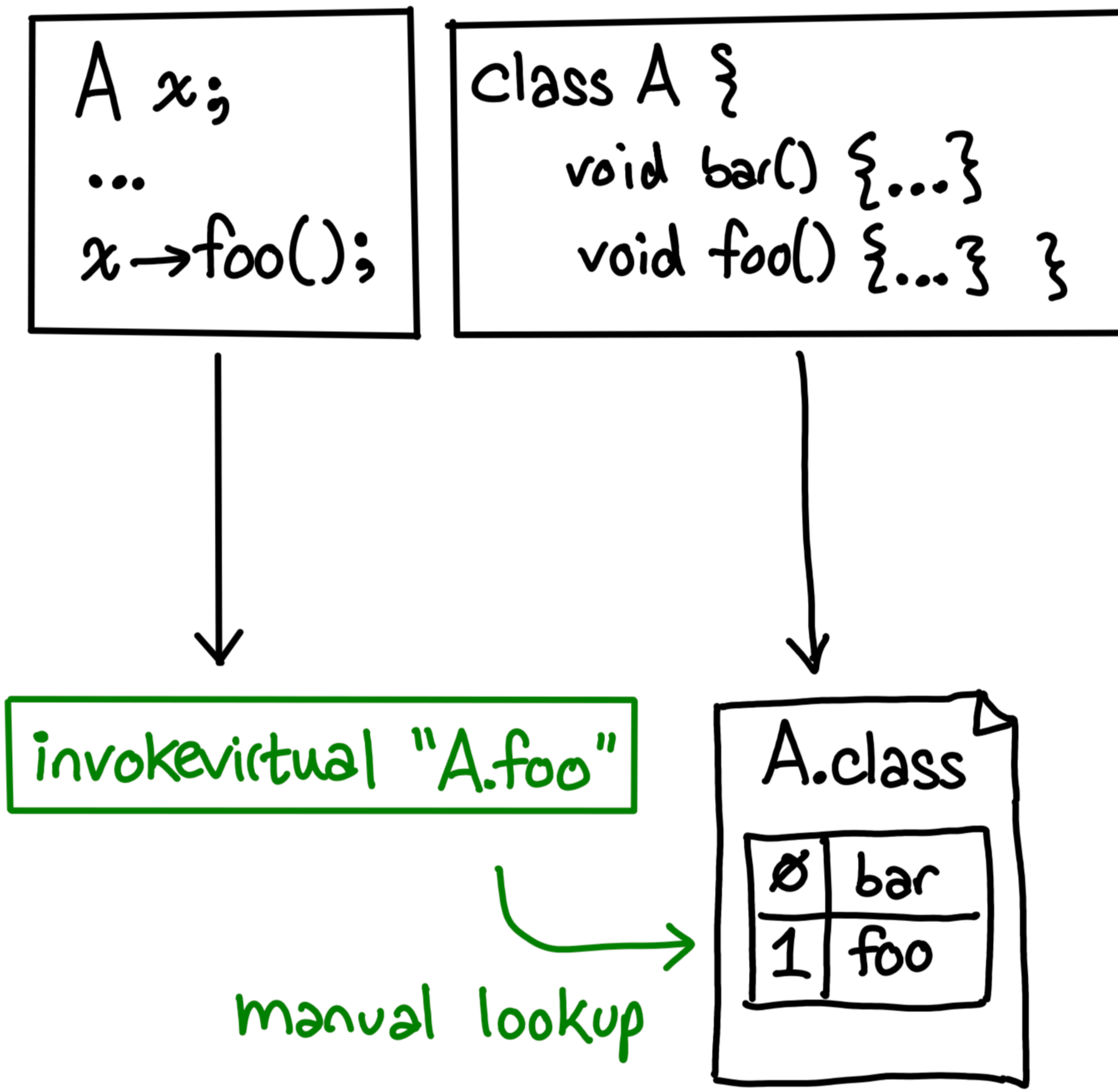


Hot loop compilation

- Runtime profiling to identify hot spots in code
- Execute compiler during runtime
- Swap in compiled code for runtime code
- Monomorphization for dynamically typed code

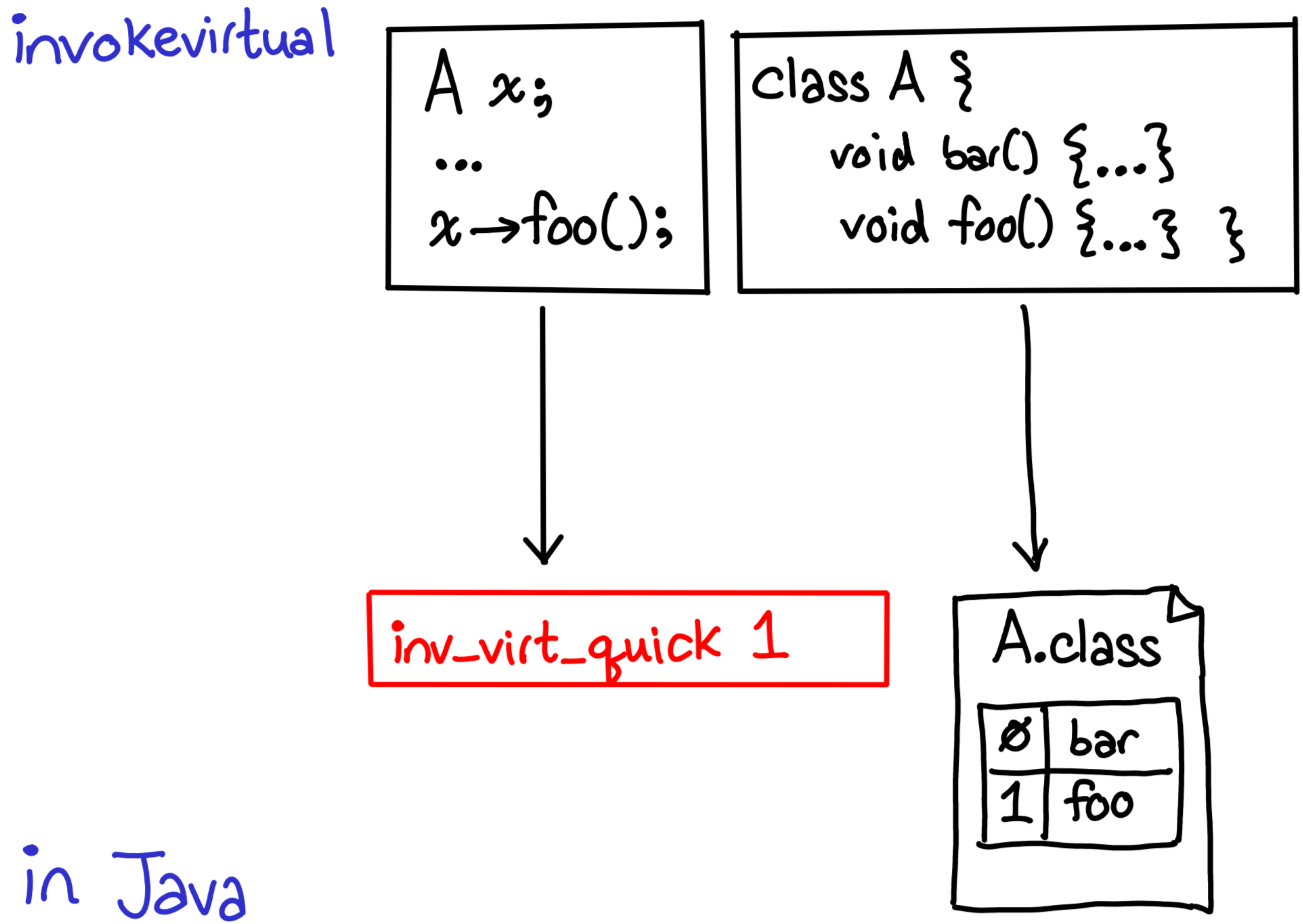
Virtual method call optimization

invokevirtual

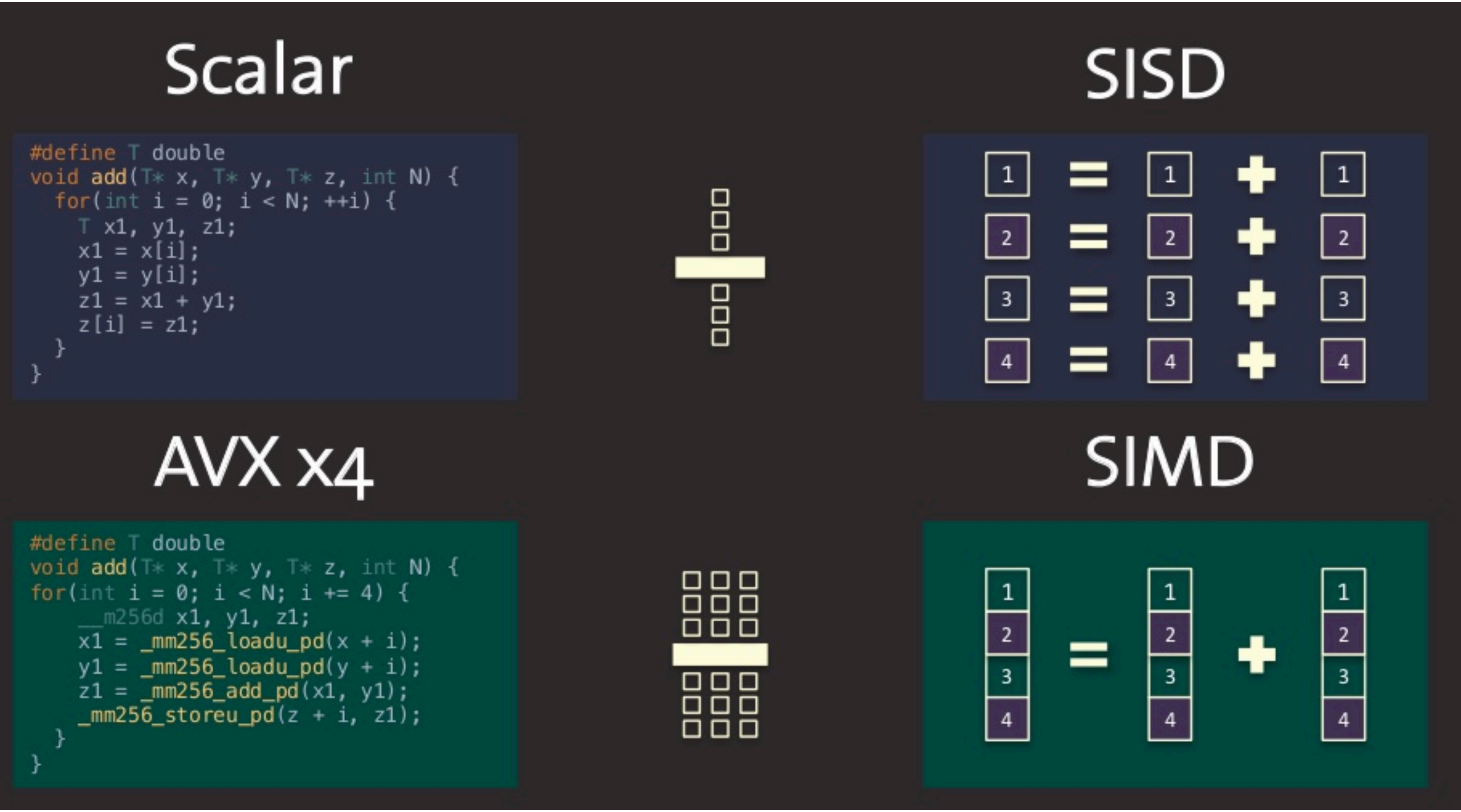


in Java

Virtual method call optimization




Javasc
function cloneSort(com
let templ
1.92% v8::internal::StringTable::LookupStrin
- v8::internal::StringTable::LookupStringIfE
+ 99.80% Builtin:KeyedLoadIC_Megamorphi
1.52% v8::internal::(anonymous namesp
v8::internal::(anonymous namespace)
v8::internal::StringTab
ists_NoAlloca
locate
teKey::IsMatch
Match
I trawled V8 issue tracker and found few
Issue 6391: StringCharCodeAt slow
Issue 7092: High overhead of Stri
Issue 7326: Performance degrad
Overhead Symbo
17.02% *doQuickSo
11.20% Builtin:ArgumentsAdaptorTramp
7.17% *compareByOriginalPositions ../dist/source-map.j
4.49% Builtin:CallFunction_ReceiverIsNullOrUndefined
3.58% *compareByGeneratedPositionsDeflated ../dist/source-map.js:1063
2.73% *SourceMapConsumer_parseMappings ../dist/source-map.js:1894
2.11% Builtin:StringEqual
1.93% *SourceMapConsumer_parseMappings ../dist/source-map.js:1894
1.66% *doQuickSort ../dist/source-map.js:2752
1.25% v8::internal::StringTable::LookupStringIfExists_NoAllocate
1.22% *SourceMapConsumer_parseMappings ../dist/source-map.js:18
1.21% Builtin:StringCharAt
1.16% Builtin:Call_ReceiverIsNullOrUndefined
1.14% v8::internal::(anonymous namespace)::StringTableNoAl
0.90% Builtin:StringPrototypeSlice
0.86% Builtin:KeyedLoadIC_Megamorphic
0.82% v8::internal::(anonymous namespace)::MakeStringThin
0.80% v8::internal::(anonymous namespace)::CopyObjectToObjectElements
0.76% v8::internal::Scavenger::ScavengeObject
0.72% v8::internal::String::VisitFlat<v8::internal::IteratingStringHasher>
0.68% *SourceMapConsumer_parseMappings ../dist/source-map.js:1894
0.64% *doQuickSort ../dist/source-map.js:2752
0.56% v8::internal::IncrementalMarking::RecordWriteSlow
function Mapping(memory) {
this._memory = memory;
this.pointer = 0;
}
Mapping.prototype = {
get generatedLine () {
return this._memory[this.pointer + 0];
},
get generatedColumn () {
return this._memory[this.pointer + 1];
},
get source () {
return this._memory[this.pointer + 2];
},
get originalLine () {
return this._memory[this.pointer + 3];
},
get originalColumn () {
return this._memory[this.pointer + 4];
},
get name () {
return this._memory[this.pointer + 5];
},
set generatedLine (value) {



Alen Stojanov. “Vectorization with LMS: SIMD Intrinsics.” 2017

Auto-vectorization

Steeped in generating great code for regular loops that performed dense matrix math, for a long time most of the Intel compiler team denied that anything more than their auto-vectorizer was needed to take care of vector unit utilization. We quickly fell into a cycle:

- They'd inform the graphics folks that they'd improved their auto-vectorizer in response to our requests and that it did everything we had asked for.
- We'd try it and find that though it was better, boy was it easy to write code that wasn't actually compiled to vector code—it'd fail unpredictably.
- We'd give them failing cases, a few months would would pass and they'd inform us that the latest version solved the problem.

And so on.

It didn't take much to fall off the vectorization path. They tried to patch things up at first but eventually they threw up their hands and came up with `#pragma simd`, which would disable the "is it safe to vectorize this" checks in the auto-vectorizer and vectorize the following loop no matter what. (Once a `#pragma` is proposed to solve a hard problem, you know things aren't in a good place.)

Matt Pharr. "The story of ispc". 2018

Play with the compiler flags

- icc –help
- Find the best flags
  - icc -c -O3 -xT -msse3 mxm.c
- Use information from icc
  - icc -vec-report5 …
- Generate assembly and stare!
  - Icc -S -fsource-asm -fverbose-asm…

Tweaked the program until the compiler is happy ☹

```
;;; for(j2 = 0; j2 < N; j2 += BLOCK_X)
        xorl      %edx, %edx
        xorl      %eax, %eax
        xorps     %xmm0, %xmm0
;;; for(k2 = 0; k2 < N; k2 += BLOCK_Y)
;;; for(i = 0; i < N; i++)
        xorl      %ebx, %ebx
        xorl      %ecx, %ecx
;;; for(j = 0; j < BLOCK_X; j++)
        xorl      %r9d, %r9d
;;; for(k = 0; k < BLOCK_Y; k++)
;;; IND(A,i,j+j2,N)+=IND(B,i,k+k2,N)* IND(Cx,j+j2,k+k2,N);
        movslq    %ecx, %r8
        lea       (%rdx,%rcx), %esi
        movslq    %esi, %rdi
        shlq      $3, %rdi
        movslq    %eax, %rsi
        shlq      $3, %rsi
..B1.13:
        movaps    %xmm0, %xmm2
        movsd     A(%rdi), %xmm1
        xorl      %r10d, %r10d
..B1.14:
        movaps    B(%r10,%r8,8), %xmm3
        mulpd     Cx(%r10,%rsi), %xmm3
        addpd     %xmm3, %xmm1
        movaps    16+B(%r10,%r8,8), %xmm4
        mulpd     16+Cx(%r10,%rsi), %xmm4
        addpd     %xmm4, %xmm2
        movaps    32+B(%r10,%r8,8), %xmm5
        mulpd     32+Cx(%r10,%rsi), %xmm5
        addpd     %xmm5, %xmm1
        movaps    48+B(%r10,%r8,8), %xmm6
        mulpd     48+Cx(%r10,%rsi), %xmm6
        addpd     %xmm6, %xmm2
        movaps    64+B(%r10,%r8,8), %xmm7
        mulpd     64+Cx(%r10,%rsi), %xmm7
        addpd     %xmm7, %xmm1
        movaps    80+B(%r10,%r8,8), %xmm8
        mulpd     80+Cx(%r10,%rsi), %xmm8
        addpd     %xmm8, %xmm2
        movaps    96+B(%r10,%r8,8), %xmm9
        mulpd     96+Cx(%r10,%rsi), %xmm9
        addpd     %xmm9, %xmm1
        movaps    112+B(%r10,%r8,8), %xmm10
        mulpd     112+Cx(%r10,%rsi), %xmm10
        addpd     %xmm10, %xmm2
        addq      $128, %r10
        cmpq      $8192, %r10
        jl        ..B1.14         # Prob 99%
```

Inner loop: SSE instructions

Auto-vectorization is not a programming model!

Thesis:

The future of performance optimization is better programming models, not better optimizers.

Parallel processing of large datasets with Spark

Word count

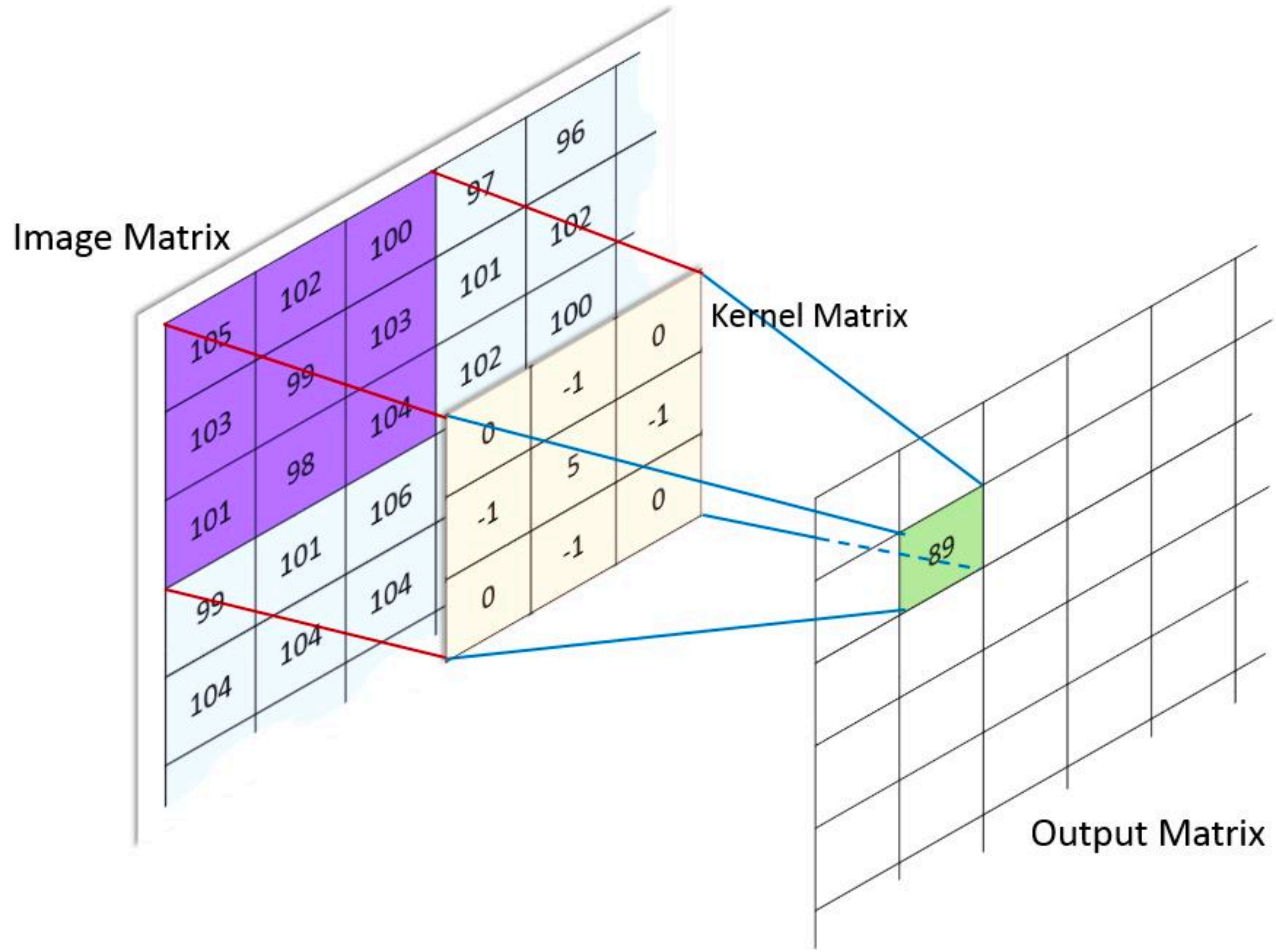
```
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
counts = lines.flatMap(lambda x: x.split(' ')) \
              .map(lambda x: (x, 1)) \
              .reduceByKey(add)
output = counts.collect()
for (word, count) in output:
    print("%s: %i" % (word, count))
```

Parallel processing of large datasets with Spark

K-means clustering

```
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
data = lines.map(parseVector).cache()
K = int(sys.argv[2])
convergeDist = float(sys.argv[3])

kPoints = data.takeSample(False, K, 1)
tempDist = 1.0

while tempDist > convergeDist:
    closest = data.map(
        lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(
        lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(
        lambda st: (st[0], st[1][0] / st[1][1])).collect()

    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)

    for (iK, p) in newPoints:
        kPoints[iK] = p
```

Low-level optimization of image processing with Halide



machinelearninguru.com. "Image Filtering"

Low-level optimization of image processing with Halide

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

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

  return blur_y;
}
```

Low-level optimization of image processing with Halide

Fredo Durand, "High-Performance Image Processing"

Halide

0.9 ms/megapixel

```
Func box_filter_3x3(Func in) {
  Func blurx, blury;
  Var x, y, xi, yi;

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

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

  return blury;
}
```

C++

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);
}}}}}
```

Low-level optimization of image processing with Halide

Fredo Durand, “High-Performance Image Processing”

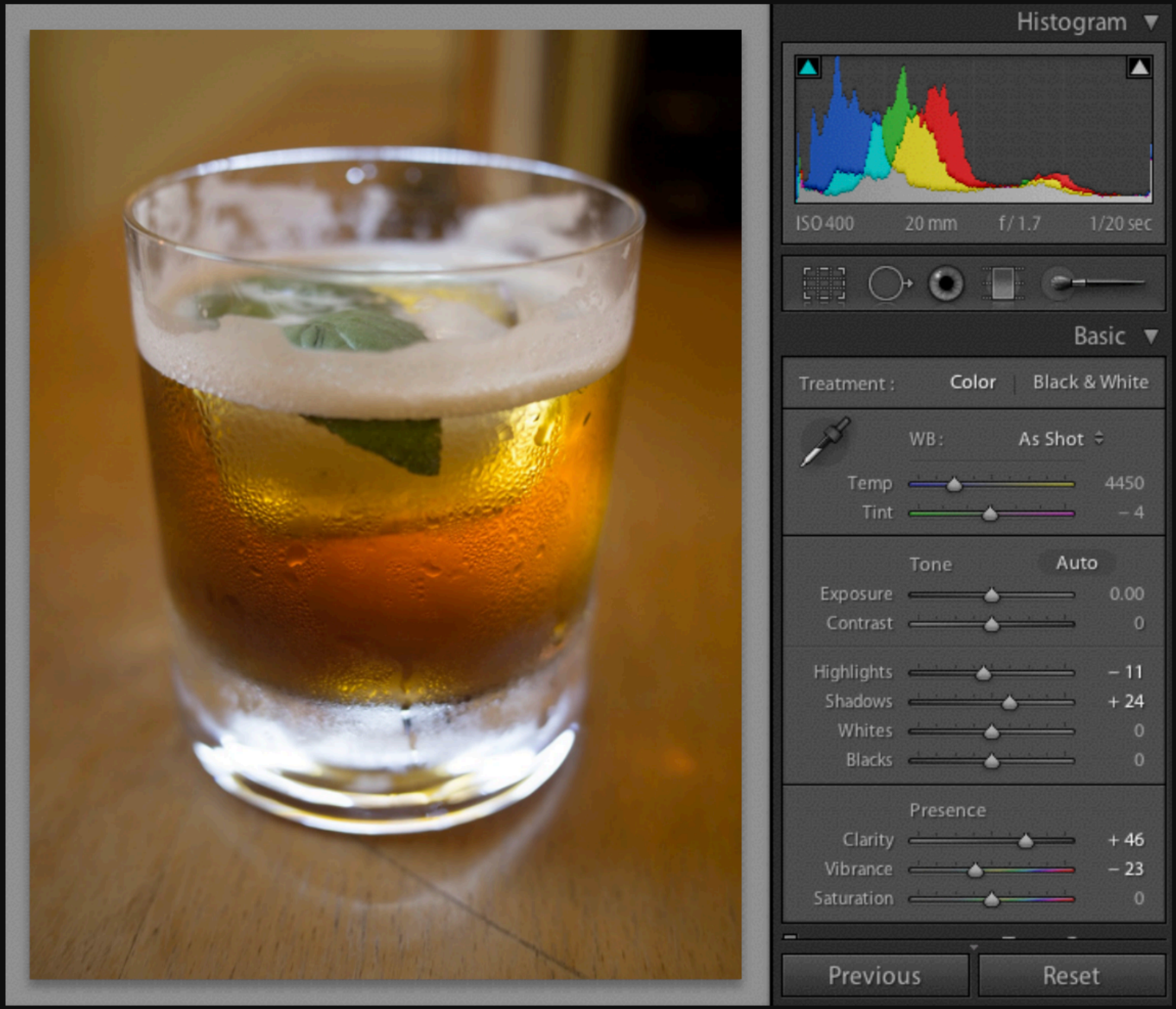
Reference: 300 lines C++

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

Halide: 60 lines
1 intern-day

20x faster (vs. reference)
2x faster (vs. Adobe)

GPU: 70x faster (vs. reference)



Efficient DOM updates with React

```
$('button').click(function() {
  if (!$('#elem1').is(':visible')) {
    $('#elem1').show();
    $('#elem2').show();
  } else {
    $('#elem1').hide();
    $('#elem2').hide();
  }
});
```

```
var toggled = false;
$('button').click(function() {
  toggled = !toggled;
  render();
});

function render() {
  let html = toggled
    ? '<div id="elem1">Elem 1</div>' +
      '<div id="elem2">Elem 2</div>'
    : '';

  $('#container').html(html);
}
```

```
class Container extends React.Component {
  state = {toggled: false}

  render() {
    return <div id="container">
      <button onClick={
        () => this.setState({toggled: !this.state.toggled})}>
        Click me
      </button>
      {this.state.toggled
       ? <div><div>Elem 1</div><div>Elem 2</div></div>
       : <div />}
    </div>;
  }
}

ReactDOM.render(<Container />, document.getElementById('container'));
```

Domain models aren't a panacea

The published work on big data systems has fetishized scalability as the most important feature of a distributed data processing platform… Contrary to the common wisdom that effective scaling is evidence of solid systems building, any system can scale arbitrarily well with a sufficient lack of care in its implementation.

We offer a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread. The COST of a given platform for a given problem is the hardware configuration required before the platform outperforms a competent single-threaded implementation.

Frank McSherry and Derek Murray. "Scalability, but at what COST?" HotOS 2015

Domain models aren't a panacea

```
fn PageRank20(graph: GraphIterator, alpha: f32) {
  let mut a = vec![0f32; graph.nodes()];
  let mut b = vec![0f32; graph.nodes()];
  let mut d = vec![0f32; graph.nodes()];

  graph.map_edges(|x, y| { d[x] += 1; });

  for iter in 0..20 {
    for i in 0..graph.nodes() {
      b[i] = alpha * a[i] / d[i];
      a[i] = 1f32 - alpha;
    }

    graph.map_edges(|x, y| { a[y] += b[x]; });
  }
}
```

← 16 lines of Rust

scalable system	cores	twitter	uk-2007-05
GraphChi [12]	2	3160s	6972s
Stratosphere [8]	16	2250s	-
X-Stream [21]	16	1488s	-
Spark [10]	128	857s	1759s
Giraph [10]	128	596s	1235s
GraphLab [10]	128	249s	833s
GraphX [10]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

Takeaways

- Despite hardware improvements, performance still matters
- Changing the programmer > changing the program
- General optimizations are hard, domain optimizations are easier