

High-Performance Image Processing

Frédo Durand
MIT CSAIL

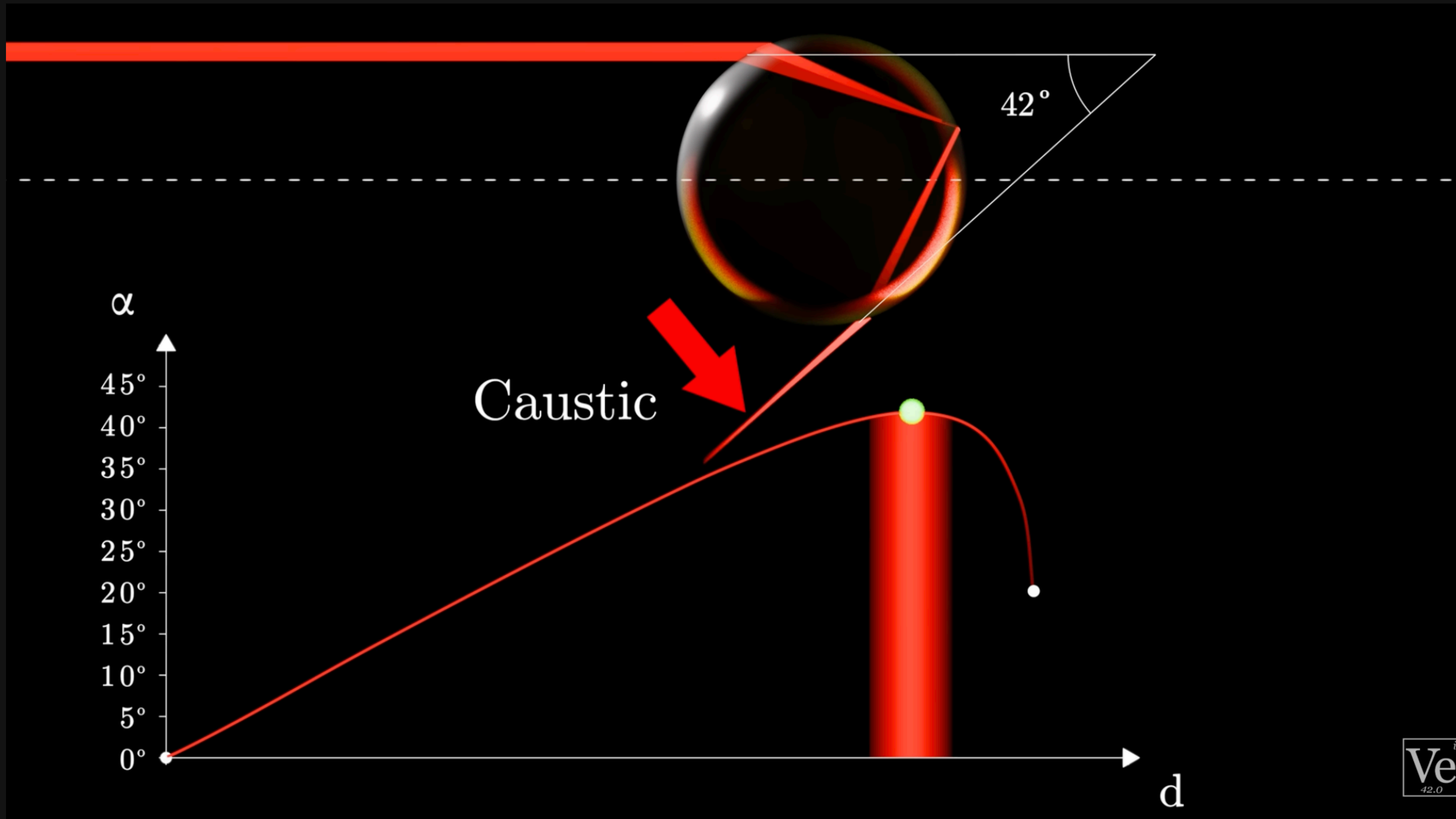
many slides
by Jonathan Ragan-Kelley

The Grading System...

Rainbows



Who has seen the Veritassium video?



Making image processing faster

Making image processing faster

Faster algorithms

e.g. separable blur, FFT, recursive, bilateral grid

Faster Hardware

More GHz, bigger better CPU/GPUs

Making image processing faster

Faster algorithms

Faster Hardware

Better leverage Existing Hardware

Remove unnecessary operations
e.g. if statements in loops, memory copy

Parallelism

Memory behavior

Inefficiencies in our/your code

IFs in loops

Checking image boundaries for each pixel
test which channel we're manipulating

Create lots of intermediate images

e.g. `im2=im1*2.0`

vectors of images: copies image each time we access

Some operations are cheaper than others

divide is more costly than multiply by inverse

Recomputing stuff over and over

take out of loop, precompute tabulated version

Modern CPUs - compute & parallelism

Moore's law, frequency scaling, power

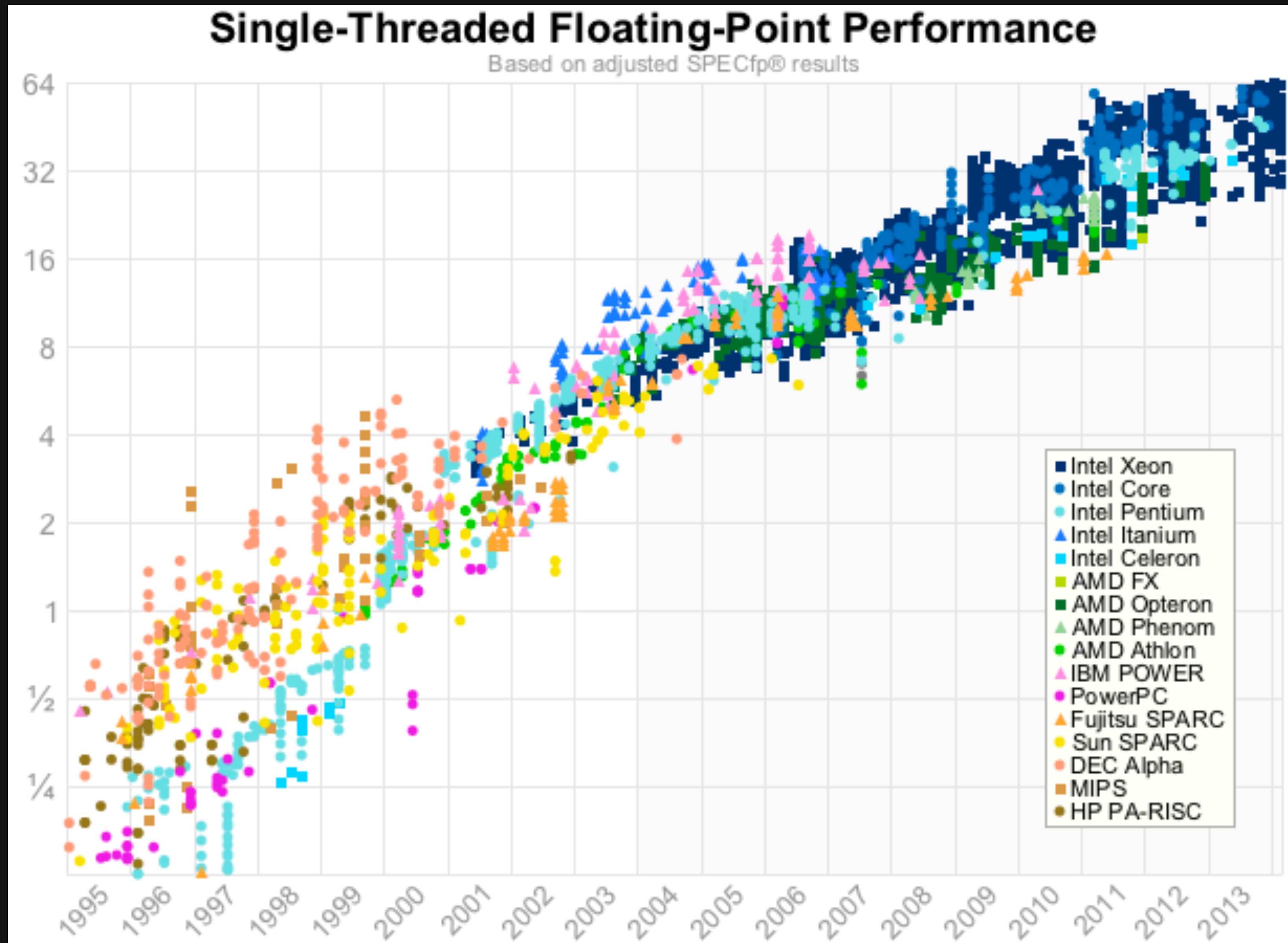
pipeline

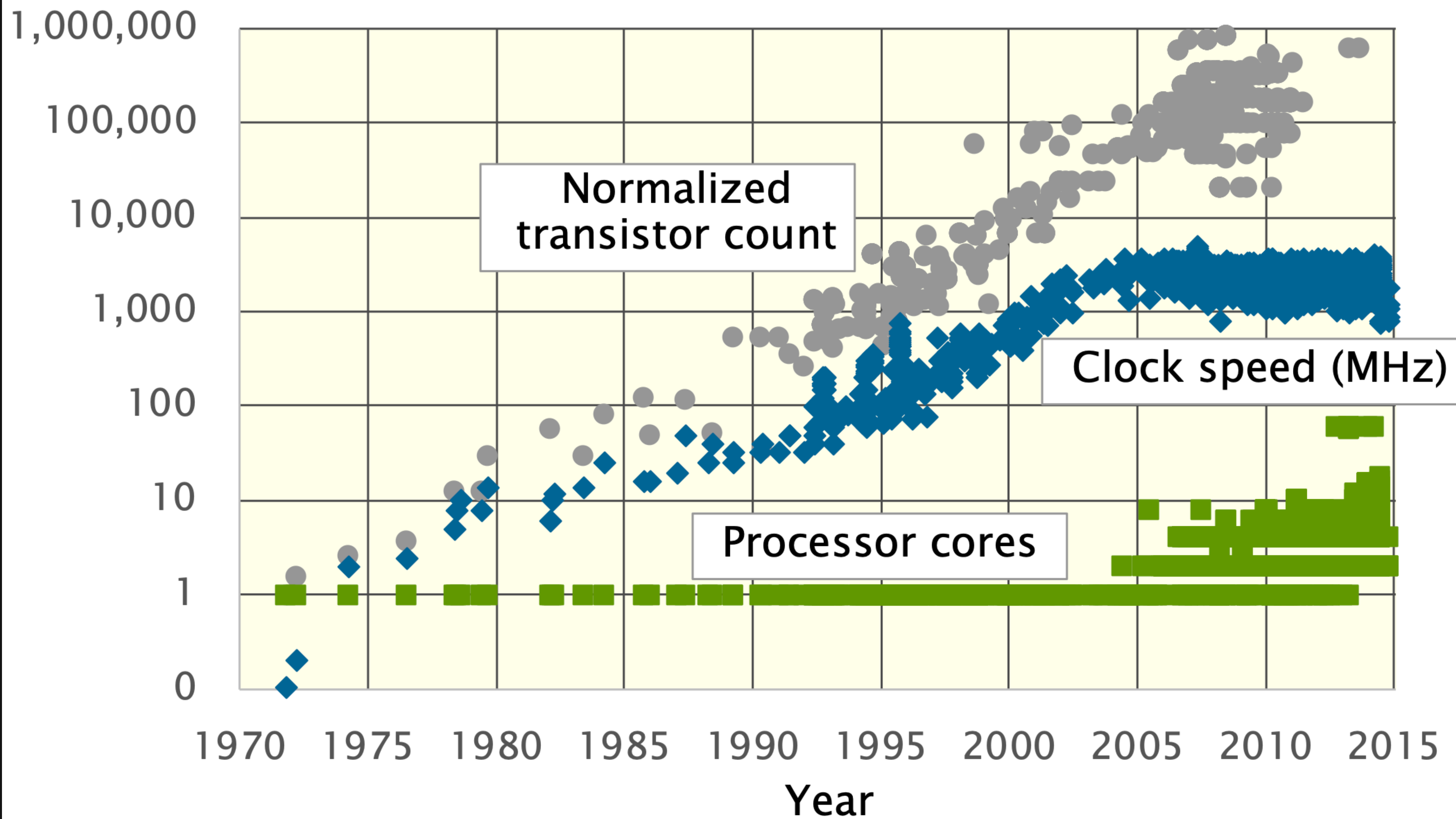
multicore

vector

see e.g. 6.172, or <http://www.lighterra.com/papers/modernmicroprocessors/>

<https://imgur.com/a/jbu7l>



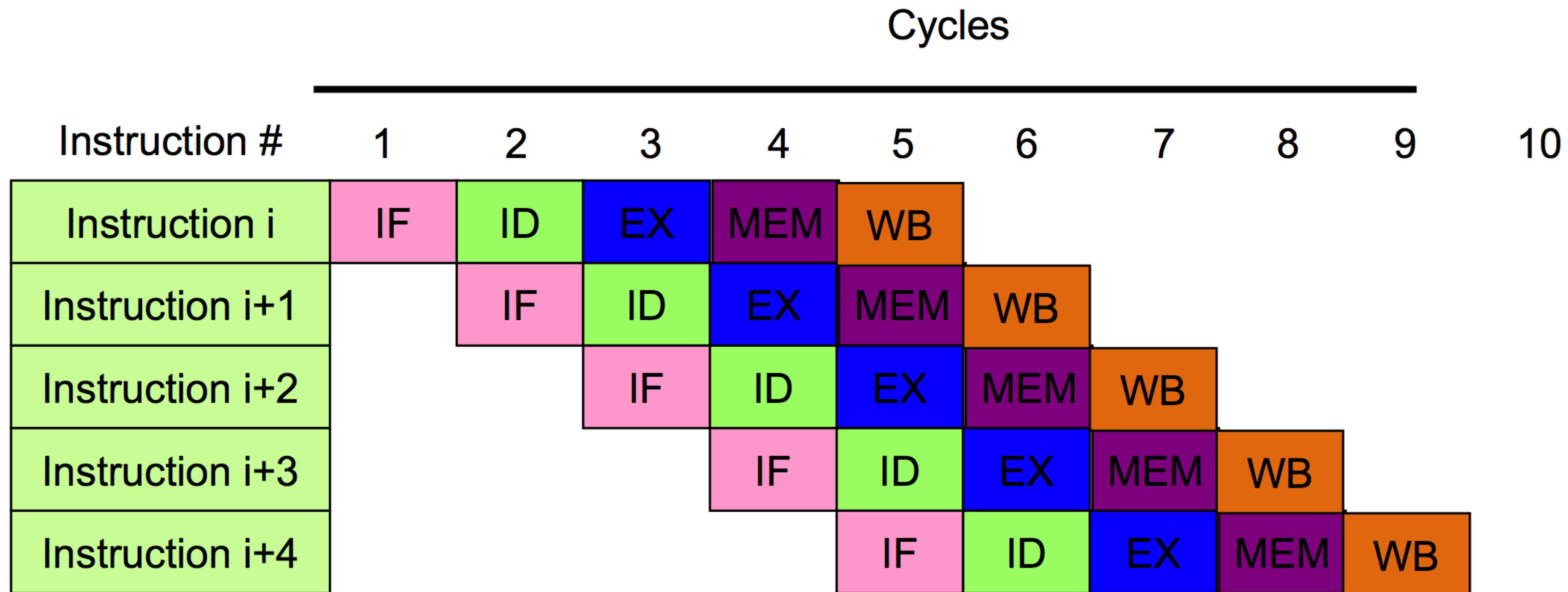


Processor data from Stanford's CPU DB [DKM12].

Pipeline (Instruction Level Parallelism)

IF: Instruction fetch
EX : Execution
WB : Write back

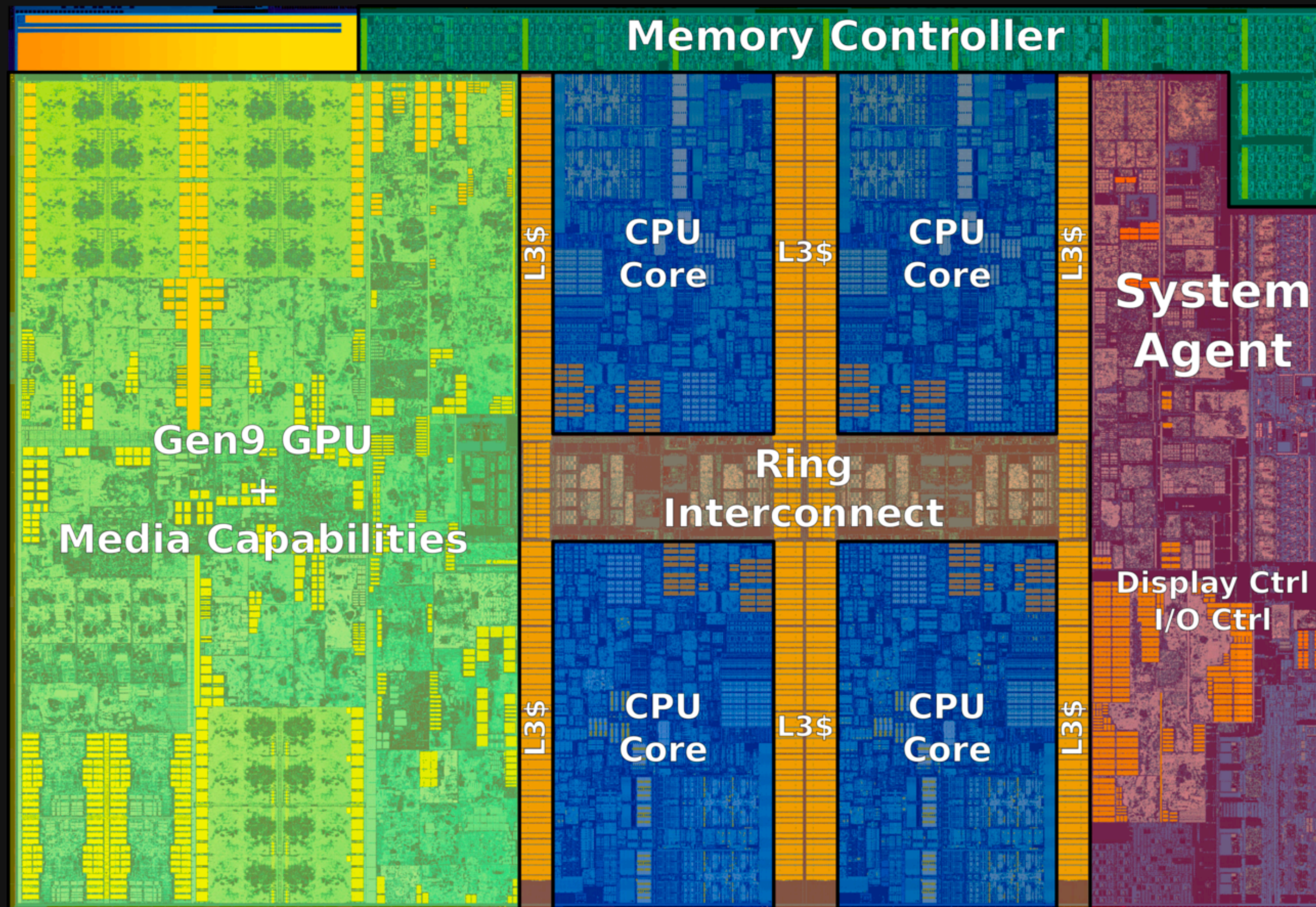
ID : Instruction decode
MEM: Memory access



Limited by dependencies

Multicore (Skylake here)

Multiple cores perform independent computation
aka MIMD, Multiple Instructions Multiple Data



Vector units

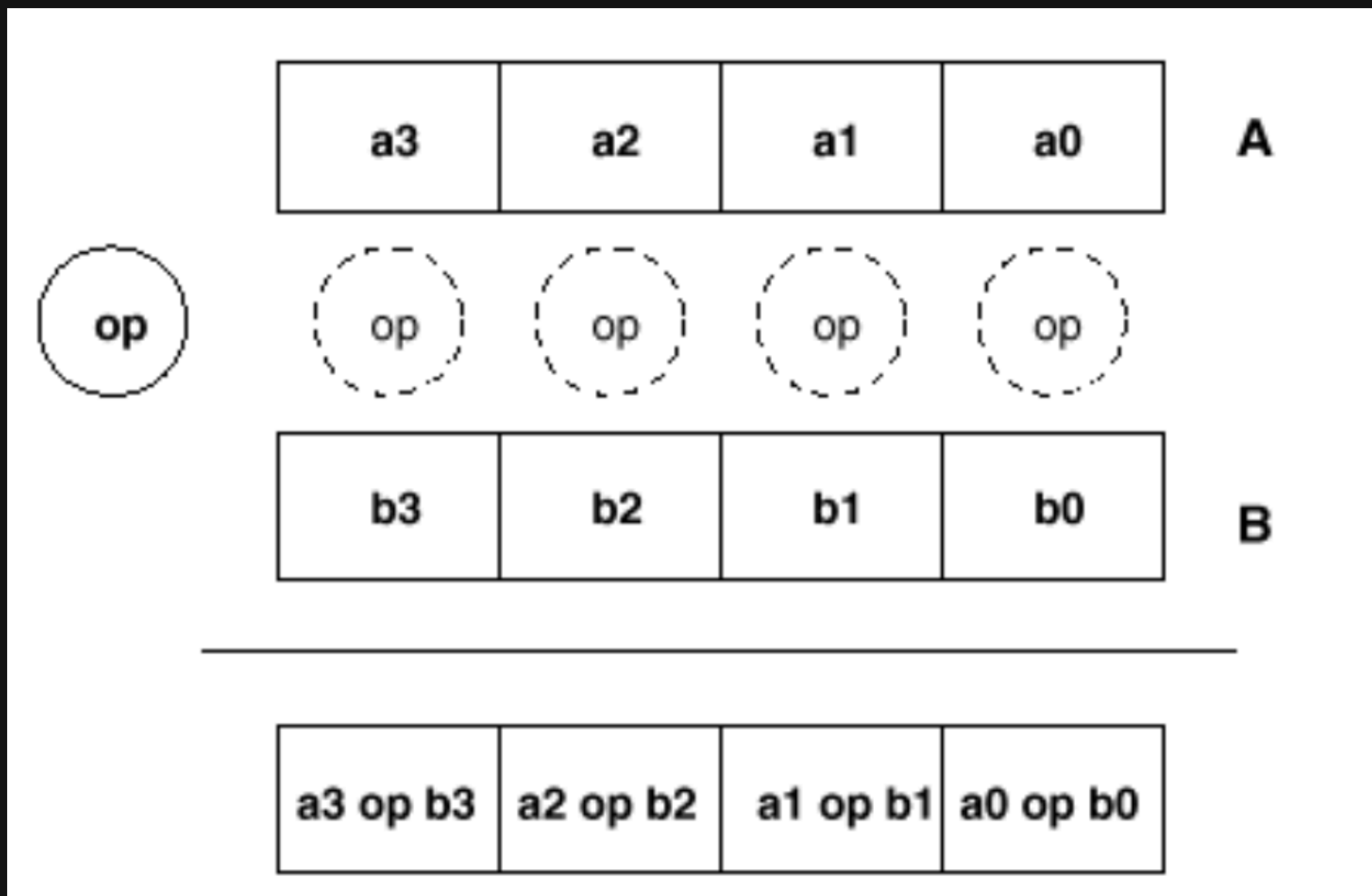
Perform same operation on multiple data

e.g. add two 3D vectors component by component in parallel

benefit: share control hardware

Vector units

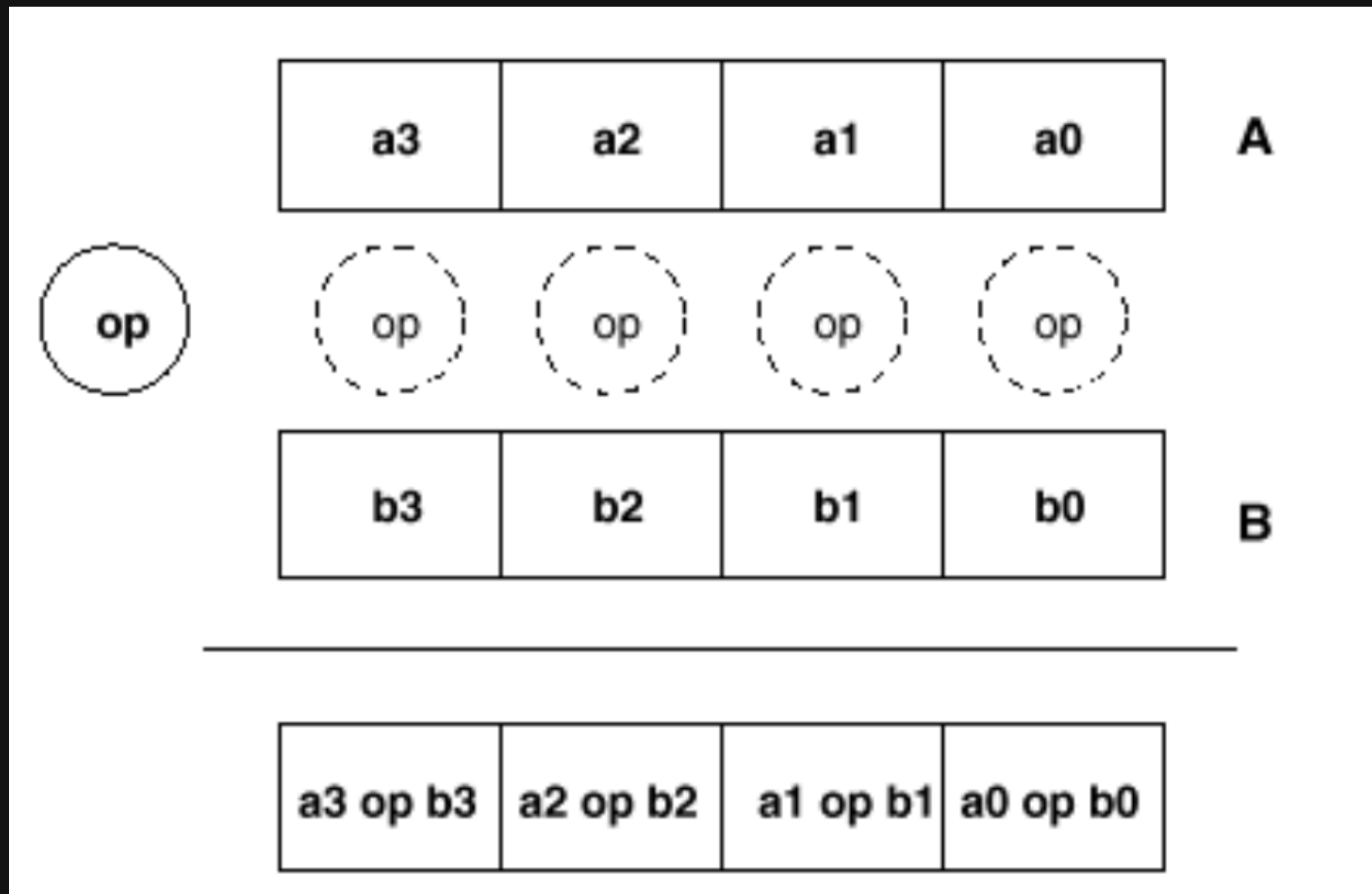
Single Instruction Multiple Data (SIMD)



Vector units

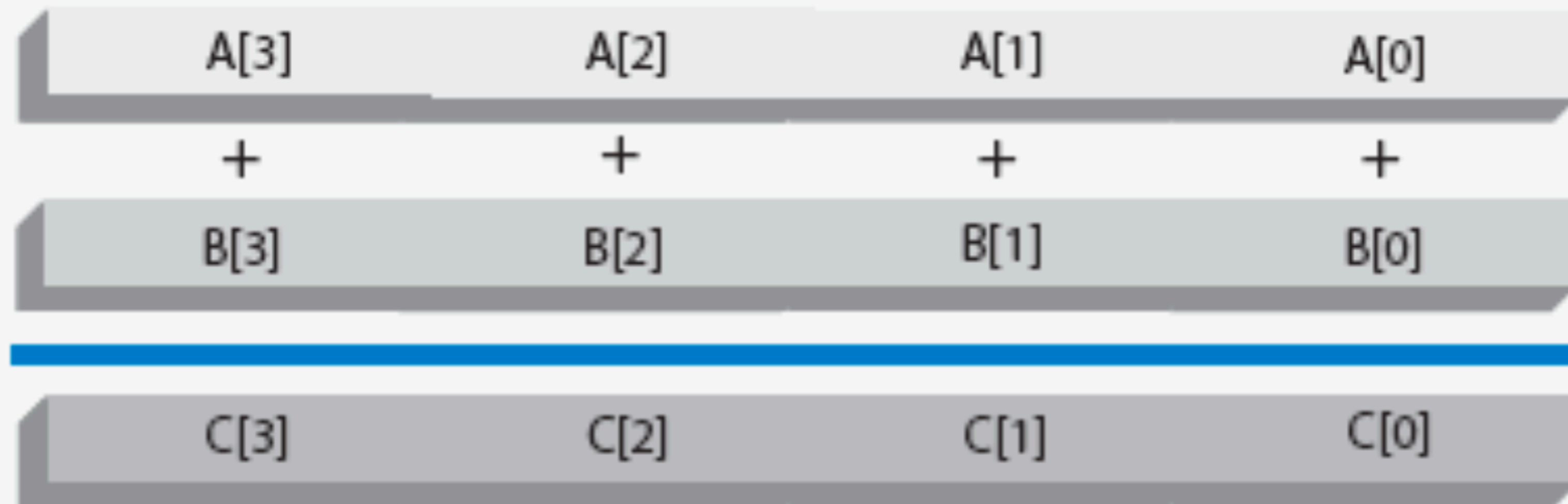
Single Instruction Multiple Data (SIMD)

Data needs to be packed (contiguous, aligned)



Vectorization Converts Loops

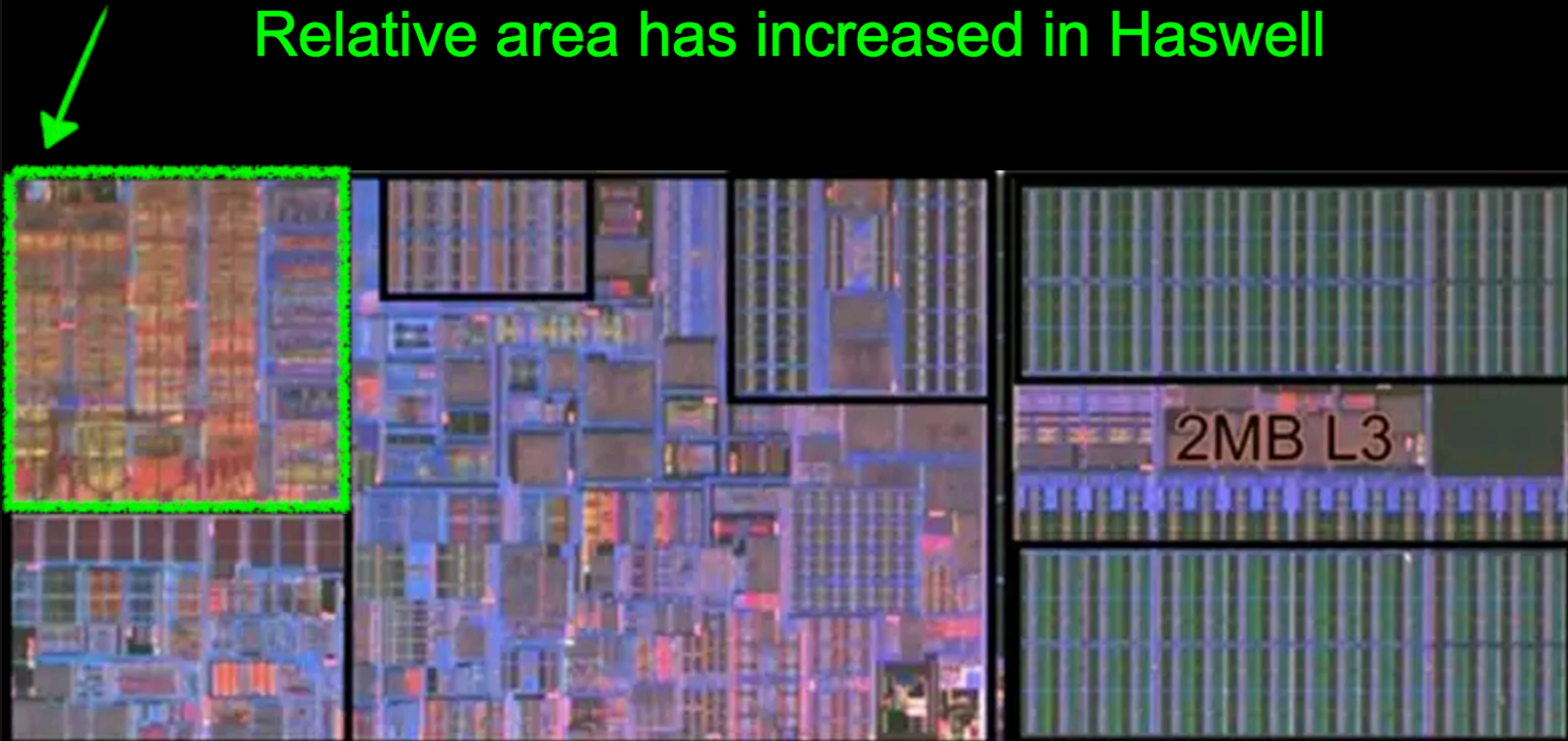
```
for (l=0; l<=MAX; l++)  
  c[l]=a[l]+b[l];
```



Vector units

Advanced Vector Extension (AVX) unit

Smaller than L3 cache, but larger than L2 cache.
Relative area has increased in Haswell



Die closeup of one Sandy Bridge core

Example: 3x3 blur — simple code

```
void box_filter_3x3(const Image &in, Image &blury) {  
    Image blurx(in.width(), in.height()); // allocate blurx array  
  
    for (int x = 0; x < in.width(); x++)  
        for (int y = 0; y < in.height(); y++)  
            blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;  
  
    for (int x = 0; x < in.width(); x++)  
        for (int y = 0; y < in.height(); y++)  
            blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;  
}
```

Using vector units

Special instruction set

Pack data

Reorganize loop in
chunks of 8 or 16

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

Recap parallelism

Pipelining (ILP) — careful of dependencies/bubbles

You don't need to worry about it, done for you by CPU designers

Multicore (MIMD) from 2 to 22 cores on CPUs, more on GPUs

Your responsibility

Coarse granularity, lots of flexibility, usually outer loop

Vector units (SIMD) from 8 to 16 lanes

Your responsibility

Fine granularity, constrained, usually inner loop

Not discussed: superscalar, reordering, etc.

Communication dominates computation in both energy and time

Operation (32-bit operands)	Energy/Op (28 nm)	<i>Cost</i> (vs. <i>ALU</i>)
ALU op		
Load from SRAM		
Move 10mm on-chip		
Send off-chip		
Send to DRAM		
Send over LTE		

data from John Brunhaver, Bill Dally, Mark Horowitz

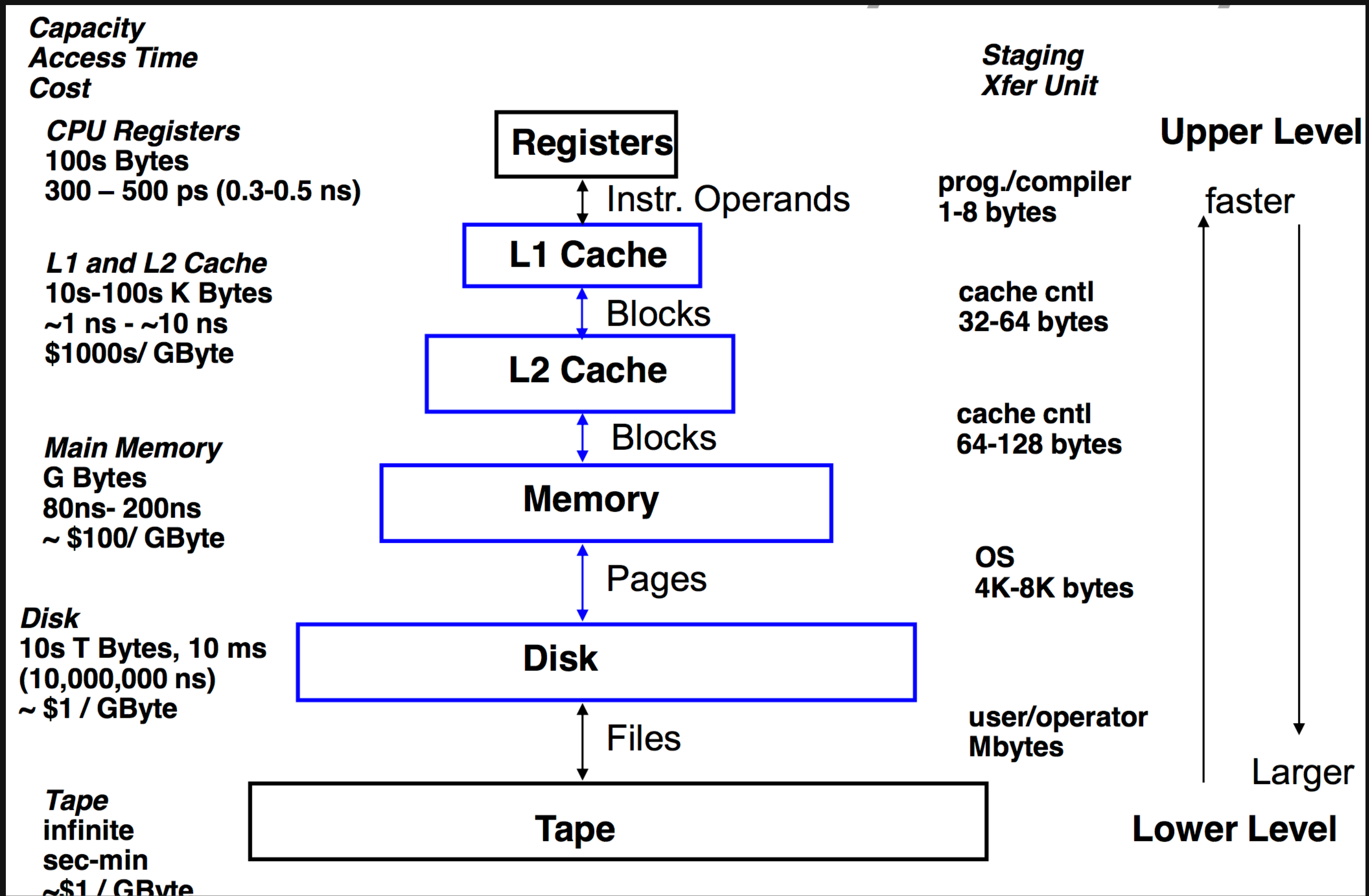
Communication dominates computation in both energy and time

Operation (32-bit operands)	Energy/Op (28 nm)	Cost (vs. ALU)
ALU op	1 pJ	-
Load from SRAM	1-5 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	>10 μ J	10,000,000x

data from John Brunhaver, Bill Dally, Mark Horowitz



Memory hierarchy



Locality

Temporal locality

use produced data as fast as possible

Spatial locality

Use adjacent data

Locality in Image Processing

Extra Problems with our Image class

Lots of copies, creations of intermediate data storage with each function operation and return

In particular no in-place operations

Multithreading

Execution model that splits work within a process into concurrent tasks

Can be at different granularity

Multiple potential goals

Make tasks independence and non-blocking
(e.g. UI does not stop when process is computing)

Hide memory latency
(others threads keeps working while one waits for data from main memory)

Parallelism (MIMD)
(different threads are run in parallel on different cores)

Enabled/facilitated by OS and CPU

Cost of thread switching may vary

Concurrency vs. Parallelism

In computer science, concurrency refers to the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome. [wikipedia]

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. [wikipedia]

Concurrency is about *potentially* running at the same time (or out of order), parallelism is *actually* running at the same time.

Recap: Better exploit hardware

Less work

Parallelism

Good cache coherence (locality)

Requires to reorganize computation!

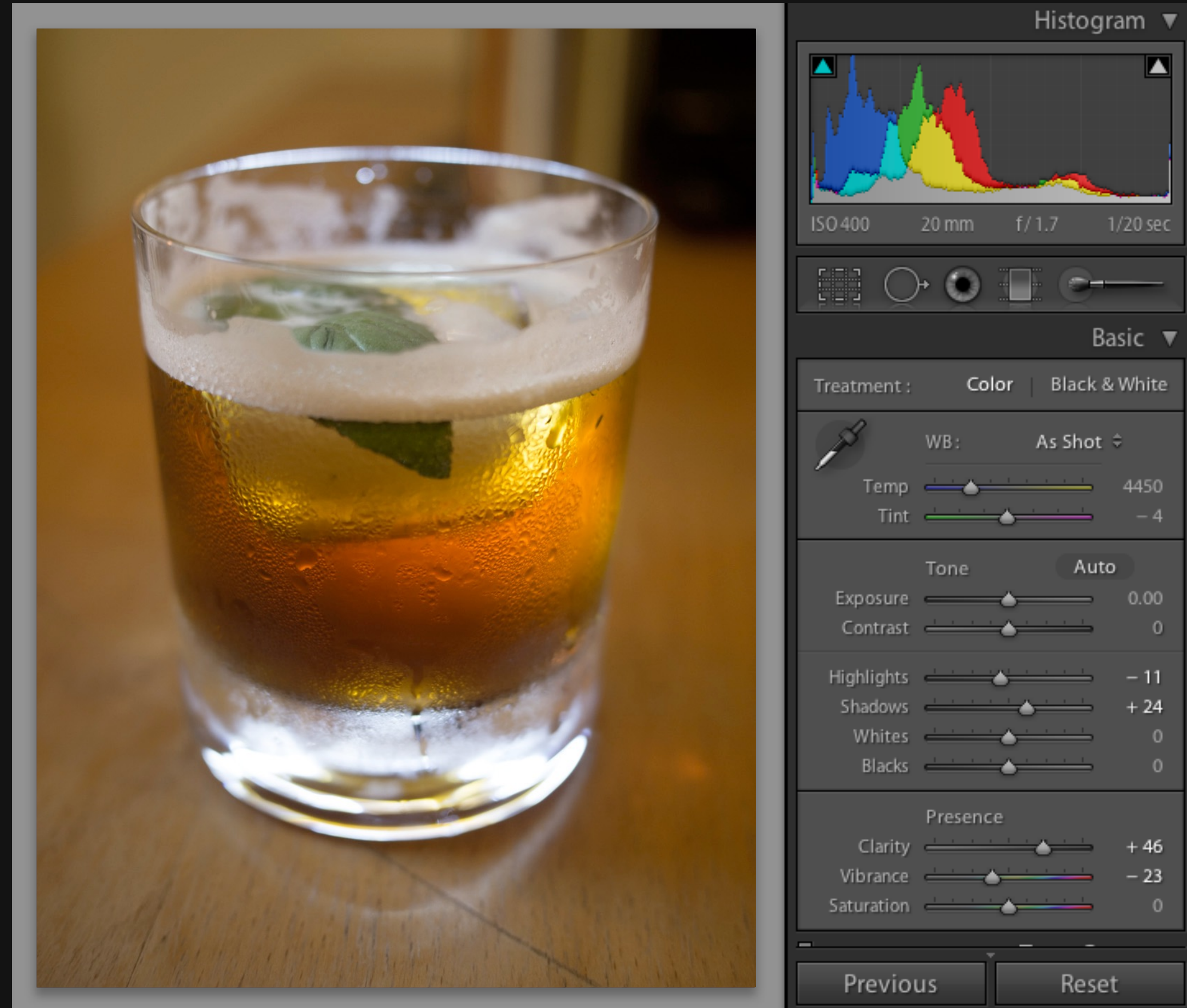
Halide motivation, e.g. Local Laplacian

Reference: 300 lines C++

Adobe: 1500 lines

3 months of work

10x faster (vs. reference)



Spoiler: e.g. Local Laplacian Filtering

Reference: 300 lines C++

Adobe: 1500 lines

3 months of work

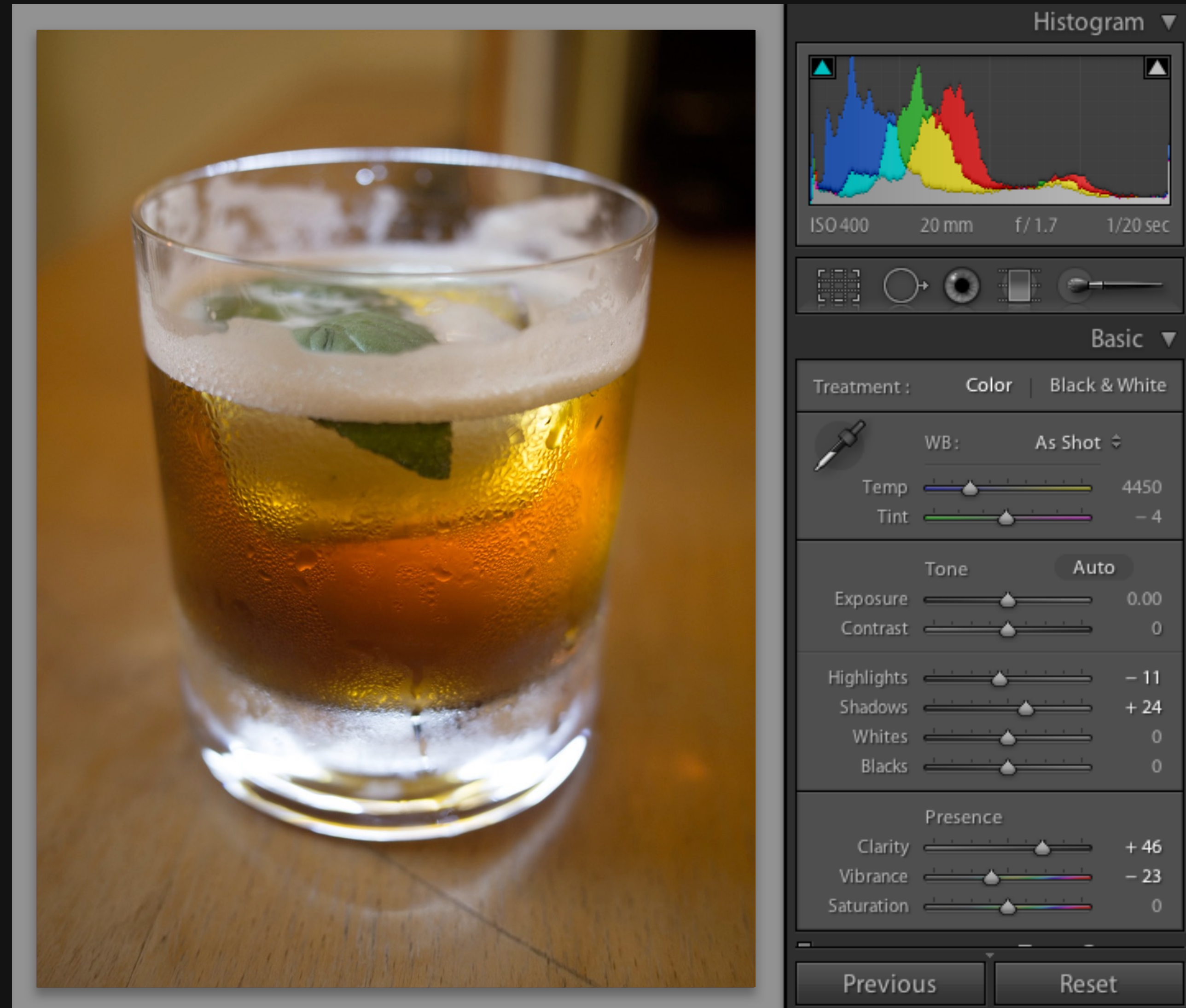
10x faster (vs. reference)

Parallelize (multicore)

Parallelize (SIMD vectorization)

Organized into tiles to maximize locality

Other tricks



Spoiler: Simpler, Faster, Scalable

Reference: 300 lines C++

Adobe: 1500 lines

3 months of work

10x faster (vs. reference)

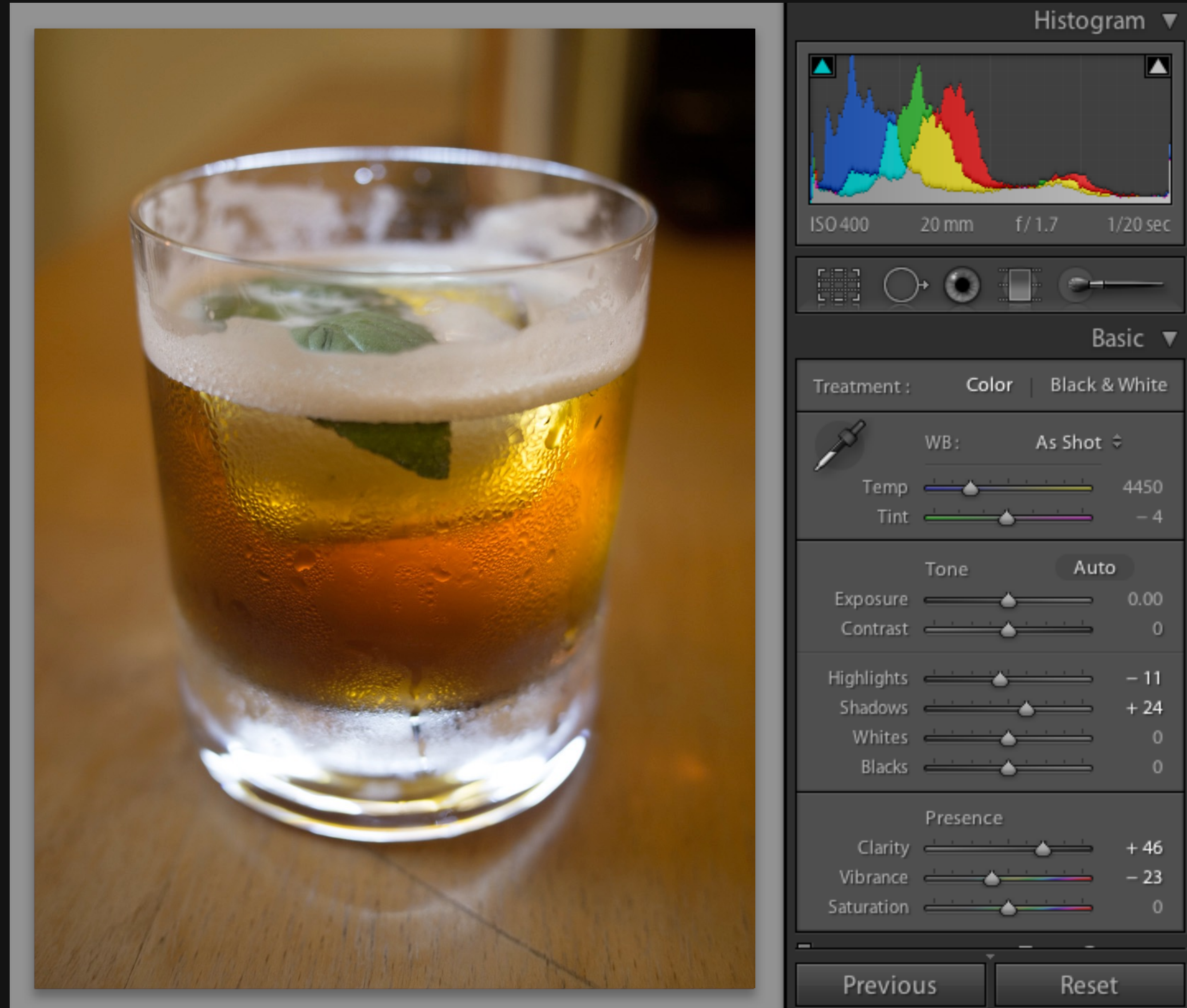
Halide: 60 lines

1 intern-day

20x faster (vs. reference)

2x faster (vs. Adobe)

GPU: 70x faster (vs. reference)



With a bit of portable Halide code you get

faster matrix multiply than Eigen

faster Gaussian blur than Intel Performance Primitives

faster Fourier transform than FFTW

faster Local Laplacian than Adobe's

Plus

it runs on device and in the cloud

is supported on Linux, Windows, OSX, iOS, Android

compiles to x86, ARM, MIPS, native client, OpenCL,

OpenGL, CUDA, JavaScript, RenderScript (ISPs soon)

slide by Marc Levoy



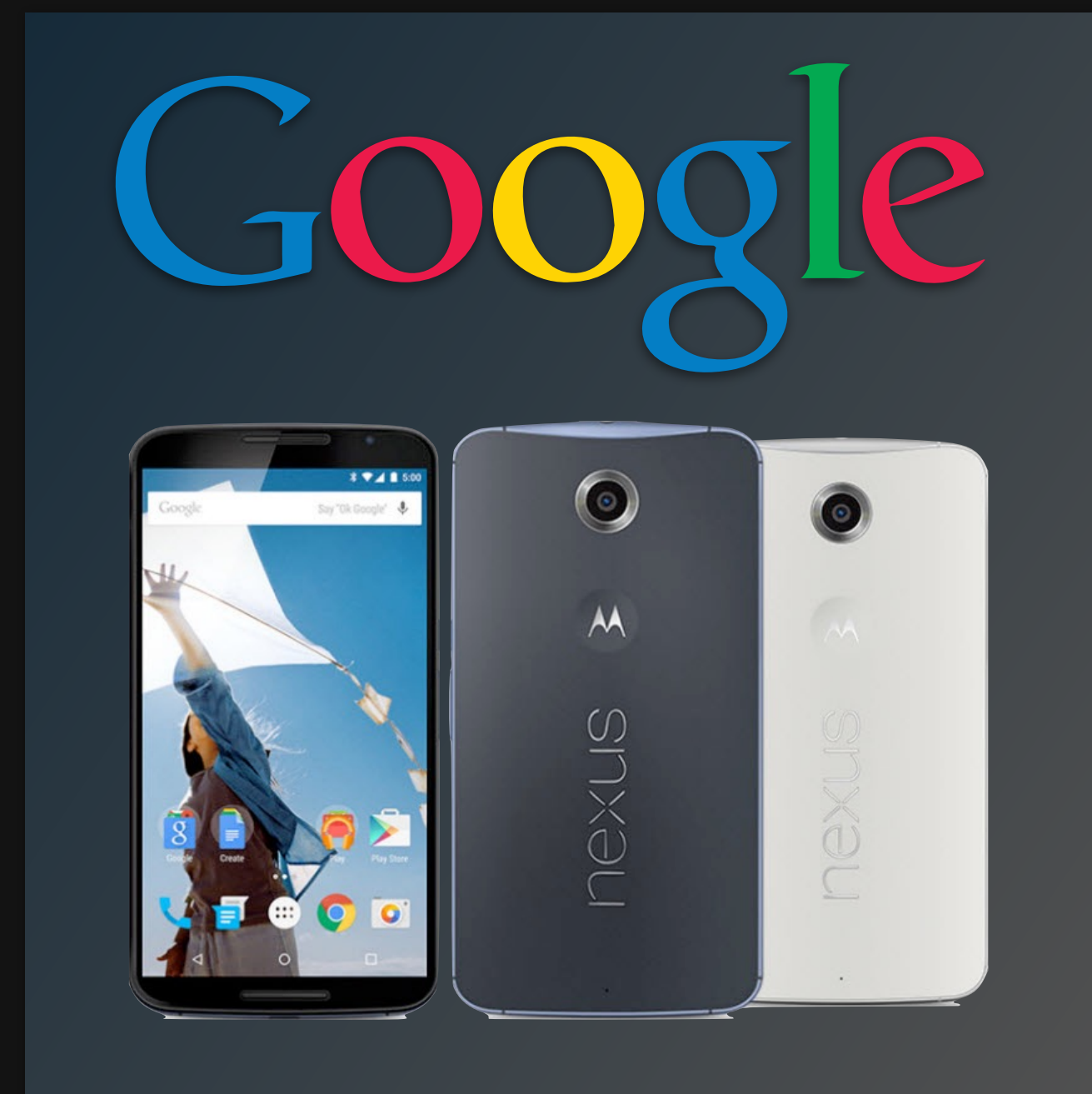
Real-world adoption

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



Google

> 200 pipelines 10s of kLOC in production



YouTube

Halide touches every video uploaded.
65B frames/day



Core features of Photoshop 2018 are compiled with Halide
Photoshop iPad enabled by

How can we get there?

Parallelism

“Moore’s law” growth will require exponentially more parallelism.

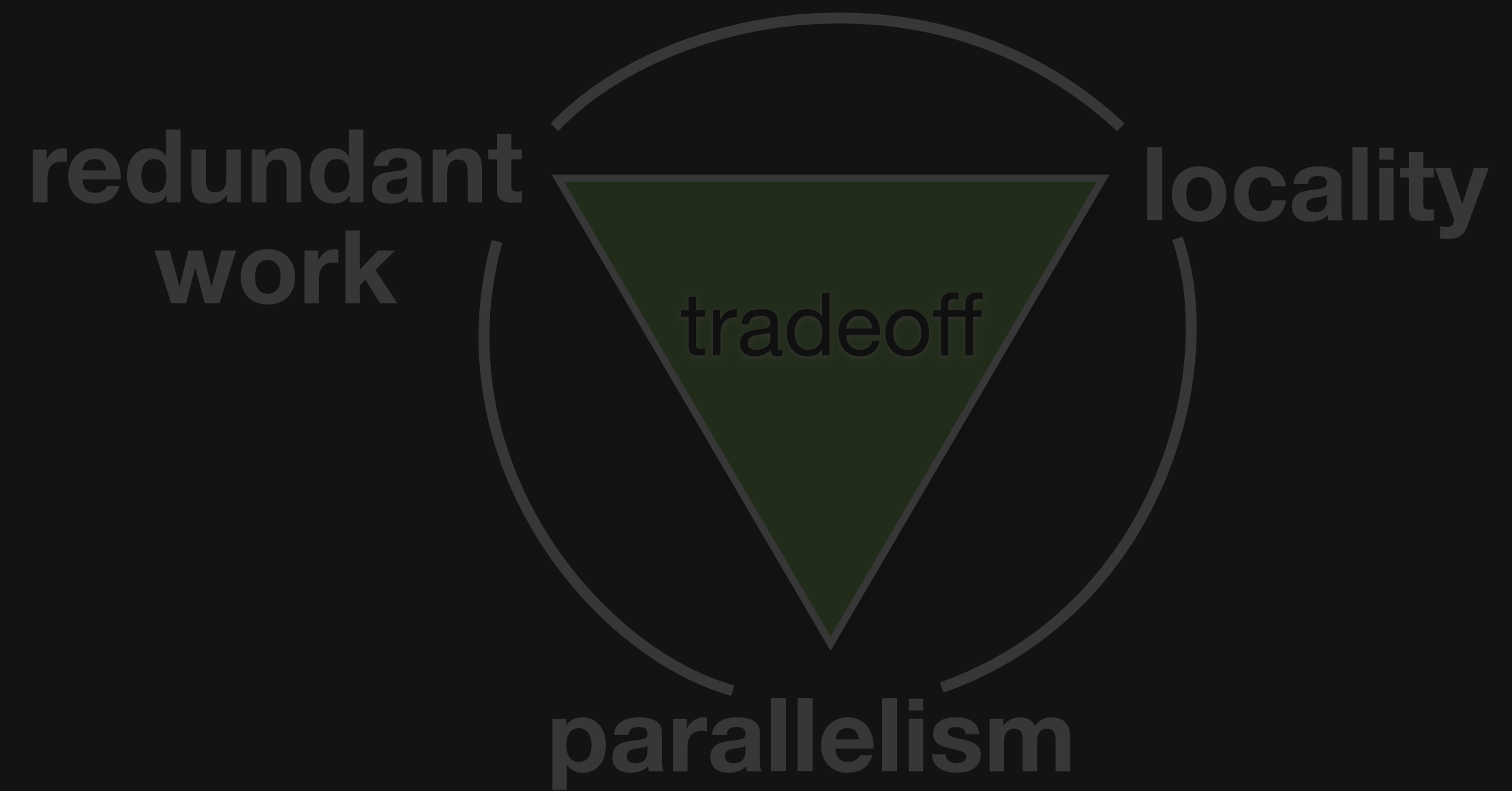
Locality

Data should move as little as possible.

Where does performance come from?

Program

Hardware



Organization of computation is a first-class issue

Program:

Algorithm

**Organization of
computation**

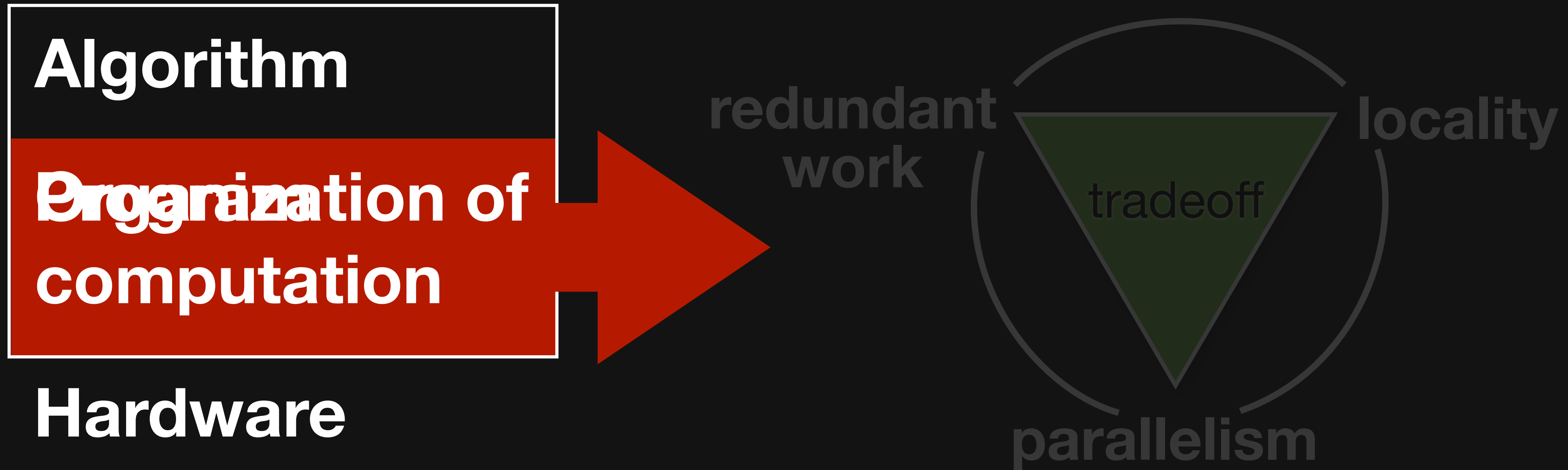
Hardware

redundant
work

locality

tradeoff

parallelism



Algorithm vs. Organization: 3x3 blur

```
void box_filter_3x3(const Image &in, Image &blury) {  
    Image blurx(in.width(), in.height()); // allocate blurx array  
  
     for (int x = 0; x < in.width(); x++)  
        for (int y = 0; y < in.height(); y++)  
            blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;  
  
     for (int x = 0; x < in.width(); x++)  
        for (int y = 0; y < in.height(); y++)  
            blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;  
}
```

Algorithm vs. Organization: 3x3 blur

```
void box_filter_3x3(const Image &in, Image &blury) {  
    Image blurx(in.width(), in.height()); // allocate blurx array  
  
    for (int y = 0; y < in.height(); y++)  
        for (int x = 0; x < in.width(); x++)  
            blurx(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++)  
            blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;  
}
```

Same algorithm, different organization

One of them is 15x faster

Why swapping loops make things faster/slower

Memory behavior

Images are laid out linearly in memory.

More coherence along scanlines

In general

Reorganize computation to maximize parallelism & locality

e.g. compute in tiles, merge pipeline stages

e.g. compute blur_x and blur_y for a full tile, compute tiles in parallel,
leverage SIMD vector units

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*

(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, IPP, MKL, OpenCV, MATLAB

optimized kernels compose into inefficient pipelines (no fusion)

Halide's answer: *decouple* algorithm from schedule

Halide's answer: *decouple* algorithm from schedule

Algorithm: *what* is computed

Schedule: *where* and *when* it's computed

Easy for programmers to build pipelines

Easy to specify & explore optimizations

manual or automatic search

Easy for the compiler to generate fast code

Halide *algorithm*:

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;

Halide *algorithm*:

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;

Halide *schedule*:

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);

Recall 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*

Final 3x3 blur: add parallelism and vectorization

```
Var x, y;  
Fun blur_x, blur_y ;  
  
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;  
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;  
  
Var xi, yi ;  
blur_y.tile(x, y, xi, yi, 8, 4) \  
    .parallel(y) \  
    .vectorize(xi, 8);  
blur_x.compute_at(blur_y, x) \  
    .vectorize(x, 8);  
  
output=blur_y.realize(input.width()-2, input.height()-2);
```

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

tile 256 x 256 + interleave

took 1.7552740097 seconds

tile 256 x 256 + parallel+vector

took 0.550438785553 seconds

tile 256 x 256 + parallel+vector without interleaving

took 1.10970659256 seconds

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

tile 256 x 256 + interleave

took 1.7552740097 seconds

tile 256 x 256 + parallel+vector

took 0.550438785553 seconds

tile 256 x 256 + parallel+vector without interleaving

took 1.10970659256 seconds

Note the exact doubling (memory bound)

Same as Hand-optimized C++

9.9 → 0.9 ms/megapixel

11x faster
(quad core x86)

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

Tiled, fused

Vectorized

Multithreaded

Redundant
computation

*Near roof-line
optimum*

The Halide Language

References

<http://halide-lang.org/>

<http://people.csail.mit.edu/jrk/halide12>

<http://people.csail.mit.edu/jrk/halide-pldi13.pdf>

<http://www.connellybarnes.com/documents/halide/>

The algorithm defines pipelines as pure functions

Pipeline stages are functions from coordinates to values

Execution order and storage are unspecified

3x3 blur as a Halide *algorithm*:

Func **blurx**, **blury**;

Var x, y;

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**;

Basic Halide program (default schedule)

```
Image<float> input = load<float>("images/rgb.png");
```

```
Var x, y;
```

```
Func blur_x;
```

```
Func blur_y;
```

```
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;
```

```
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;
```

```
Image<float> output = blur_y.realize(input.width()-2,  
input.height()-2);
```

Metaprogramming

Halide is an embedded language (in C++)

Create C++ objects that describe a Halide program

Metaprogramming

```
Image<float> input = load<float>("images/rgb.png") ;  
Var x, y;  
Func blur_x;  
Func blur_y;  
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;  
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;
```

Metaprogramming

Create C++ objects that describe a Halide program

Essentially algebraic trees (Abstract Syntax Tree, AST)

Once the representation is constructed, call `.realize()` to compile and execute

This calls the C++ Halide compiler, creates binary, executes it

Metaprogramming

Makes it easy to embed in an existing language and codebase

Avoids the need to parse

Basic Halide program (default schedule)

```
Image<float> input = load<float>("images/rgb.png");
```

```
Var x, y;
```

```
Func blur_x;
```

```
Func blur_y;
```

```
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;
```

```
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;
```

```
Image<float> output = blur_y.realize(input.width()-2,  
input.height()-2);
```

Loops are implicit !!!!!

```
Image<float> input = load<float>("images/rgb.png") ;
```

```
Var x, y;  
Func blur_x;  
Func blur_y;
```

```
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;  
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;
```

```
Image<float> output = blur_y.realize(input.width()-2,  
input.height()-2) ;
```

Halide is a purely Functional language

No side effect, no environments!

Define your algorithm as a pipeline of pure functions

Loops are NOT specified in the Halide algorithm

That's the job of the schedule

Main Halide elements

Func: pure functions defined over integer domain

These are the central objects in Halide
e.g. blur, brighten, harris, etc.

Var: abstract variable representing the domain

e.g. x, y, c

Expr: Algebraic expression of Halide Funcs and Vars

e.g. $x^{**2} + y + \text{blur}(x, y, c)$

Most operators and functions you expect are available (+, -, *, /, **, sqrt, cos..)

Image: represents inputs and output image

Can be created from our numpy arrays.

Careful: convention is that order is x, y

Under the hood

Func, Var, Expr, Image are C++ classes representing Halide programs

Declare Halide Funcs, Vars, etc. by calling the C++ constructor of these classes

Define Funcs and Expr with operator overloading: (), =, +, *, etc.

Manipulate Halide schedules with methods of the Func class

Types are more complicated/strict than C

Realize

Func.realize calls the Halide compiler on the C++ representation

Lots of magic happens behind the scene

Will depend on the schedule

More later

then executes your code and returns a Halide Image

Always call on last stage

unless you're debugging earlier stages

Swept under the rug so far

Reductions

sum, histogram, general convolution

Types and precision

floats, 16 bits, 8 bits, etc.

It's annoying to work with 8-bit integers because of overflow

Clamping to image boundary

like black/edge padding. easy.

and of course, scheduling not explained yet

Halide good news and bad news

Halide syntax is easy

Halide is cross platform (Linux, Mac, Windows)

Although some better than others (I'm looking at you Windows)

Supports many targets (x86, GPU, ARM)

Halide generates really fast executables

Halide is easy to embed in C++

Halide is not Turing complete and cannot represent everything

e.g. the iteration loop of conjugate gradient would have to be in the host language, not Halide

Halide doesn't always give useful error messages

Recap & Questions?

Functional (no side effect, no loop in algorithms (mostly))

Pipeline of functions

Embedded in C++:

Under the hood: C++ classes represent the Halide program

Simple syntax: Func, Var, Expr

Call realize to compile and execute

```
Image<float> input = load<float>("images/rgb.png");  
Var x, y;  
Func blur_x, blur_y;  
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;  
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;  
Image<float> output = blur_y.realize(input.width()-2, input.height()-2);
```


Schedule

Given an algorithm defined as a pipeline of Func

Specifies the order of computation

Schedule

Given an algorithm defined as a pipeline of Func

Specifies the order of computation

within each Func

across Func (the more interesting part)

Boils down to specifying nested loops

e.g. tiled computation for one stage:

```
for y_tile_index:  
    for x_tile_index:  
        for y_within_tile:  
            for x_within_tile:  
                out(y_tile_index*tile_height+y_within_tile,  
                    x_tile_index*tile_width+x_within_tile)=...
```

Schedule across stages

e.g. separable blur: blur_y of blur_x

blur_y (later stage) is the **consumer**

blur_x (earlier) is the **producer**

Schedule across stages

e.g. separable blur: blur_y of blur_x

blur_y (later stage) is the **consumer**

blur_x (earlier) is the **producer**

Locality: have values consumed soon after they are produced

Scheduling is driven by the consumer

Specify when the producer is computed with respect to the consumer

Why start with consumer

In general, schedule from the output to the input

Because consumer knows dependencies, producer doesn't
while data flow from producer to consumer,
dependencies flow from consumer to producer

essentially pull perspective rather than push

Compute_root

Root schedule (across)

Root schedule

Most similar to what you'd do in C++

Compute the producer on the entire image before starting the consumer

Specified with `Func.compute_root()`

Default and only option for the output (last stage)

```
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3;  
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3;  
blur_y.compute_root();  
blur_x.compute_root();
```

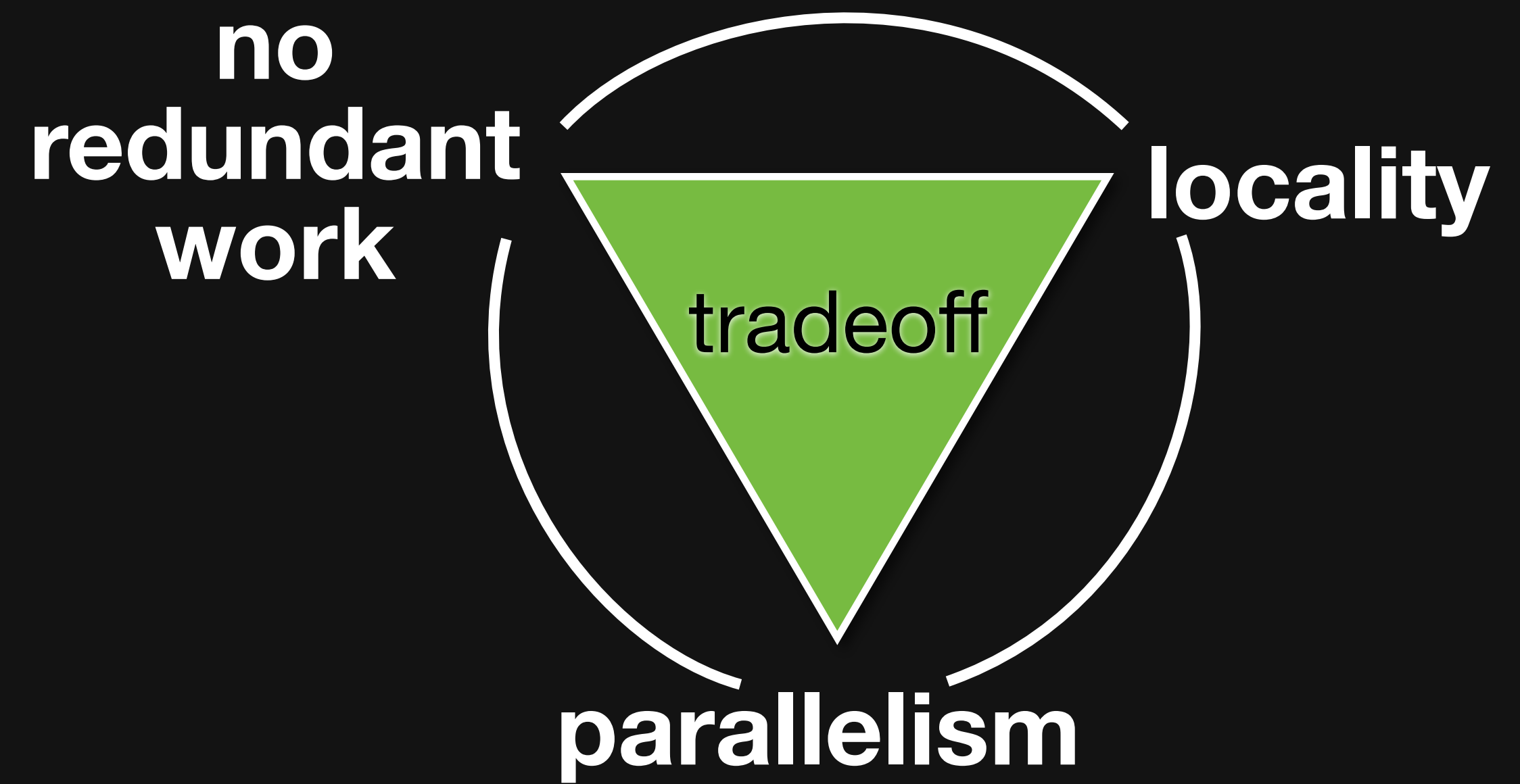
Root schedule equivalent

Root schedule equivalent

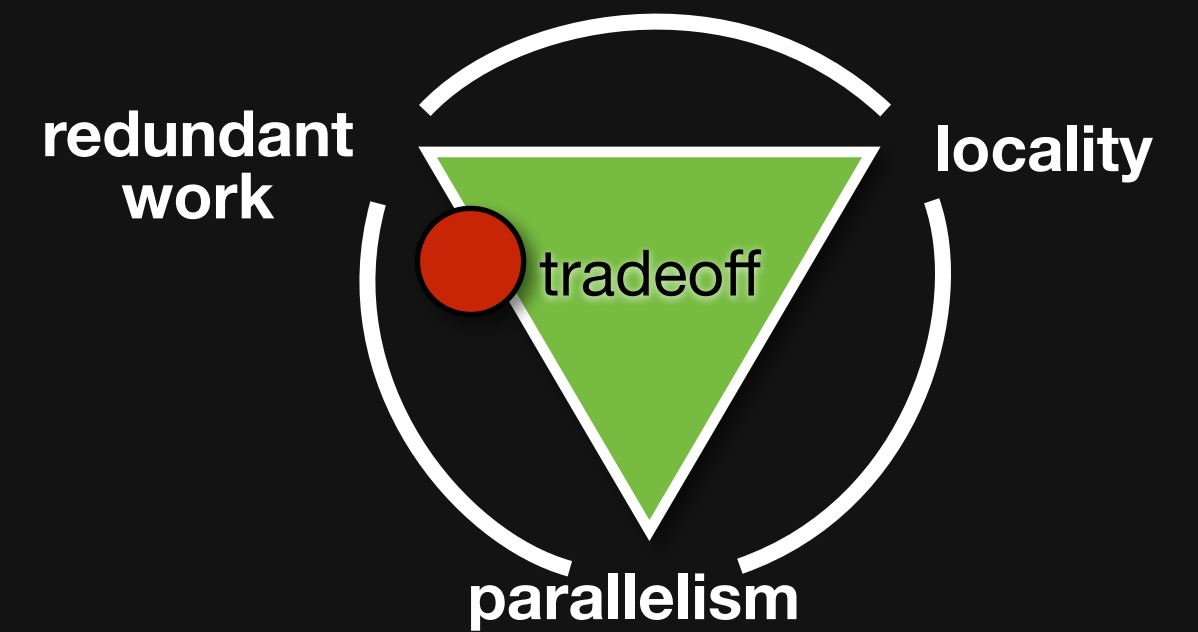
```
allocate tmp(width, height+2)
for y in(0..height+2):
  for x in(0..width):
    tmp[x,y]=(input[x,y]+input[x+1,y]+input[x+2,y])/3

for y in(0..height):
  for x in (0..width):
    out[x,y] = (tmp[x,y]+tmp[x,y+1]+tmp[x,y+2])/3
```

Root pros and cons



Recap: root



Most similar to what you'd do in C++

Compute the producer on the entire image before starting the consumer

Specified with `Func.compute_root()`

Default and only option for the output (last stage)

```
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3;  
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3;  
blur_y.compute_root();  
blur_x.compute_root();
```

inline

Inline schedule (across)

Inline schedule

**Opposite of root: compute producer right when needed,
for each value of the consumer**

No loop for the producer, all inside consumer

```
blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3
blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3
blur_y.compute_root()
blur_x.compute_inline()
# both could be skipped since they are the default
```

Inline schedule equivalent

```
//reminder, Halide algorithm:  
// blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3  
// blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3  
  
allocate out (width, height)
```

Inline schedule equivalent

```
//reminder, Halide algorithm:  
// blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3  
// blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3  
  
allocate out (width, height)  
for y in (0..height):  
    for x in(0..width):  
        out[x,y] = ((inputP[x,y]+inputP[x+1,y]+inputP[x+2,y])/3  
                    + (inputP[x,y+1]+inputP[x+1,y+1]+inputP[x+2,y+1])/3  
                    + (inputP[x,y+2]+inputP[x+1,y+2]+inputP[x+2,y+2])/3 )/3
```

Inline schedule

**Opposite of root: compute producer right when needed,
for each value of the consumer**

No loop for the producer, all inside consumer

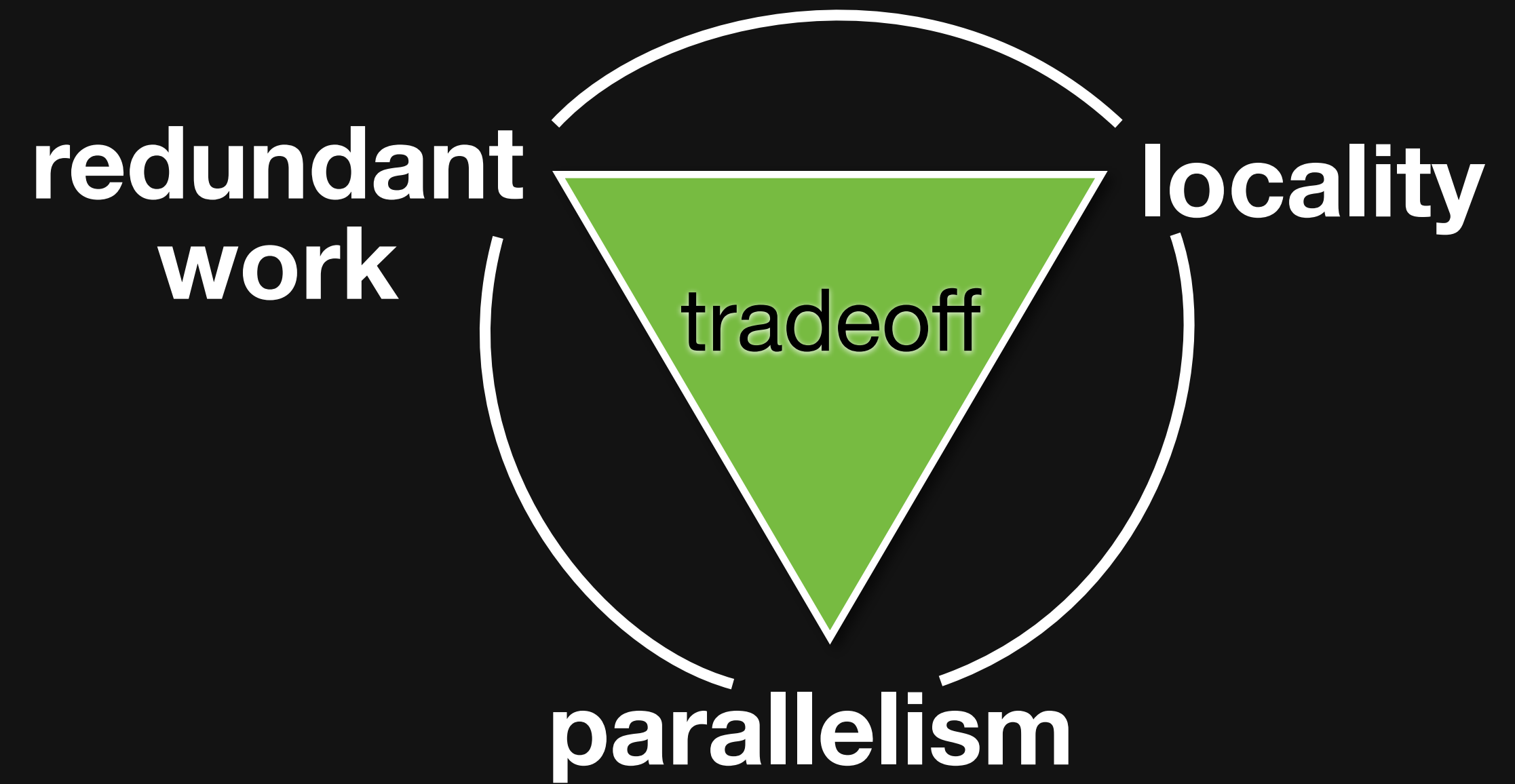
Specified with `Func.compute_inline()`

Default schedule (except output)

This makes it easier to use many intermediate Funcs for complex algebraic calculations.

```
blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3
blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3
blur_y.compute_root()
blur_x.compute_inline()
# both could be skipped since they are the default
```

Inline pros and cons

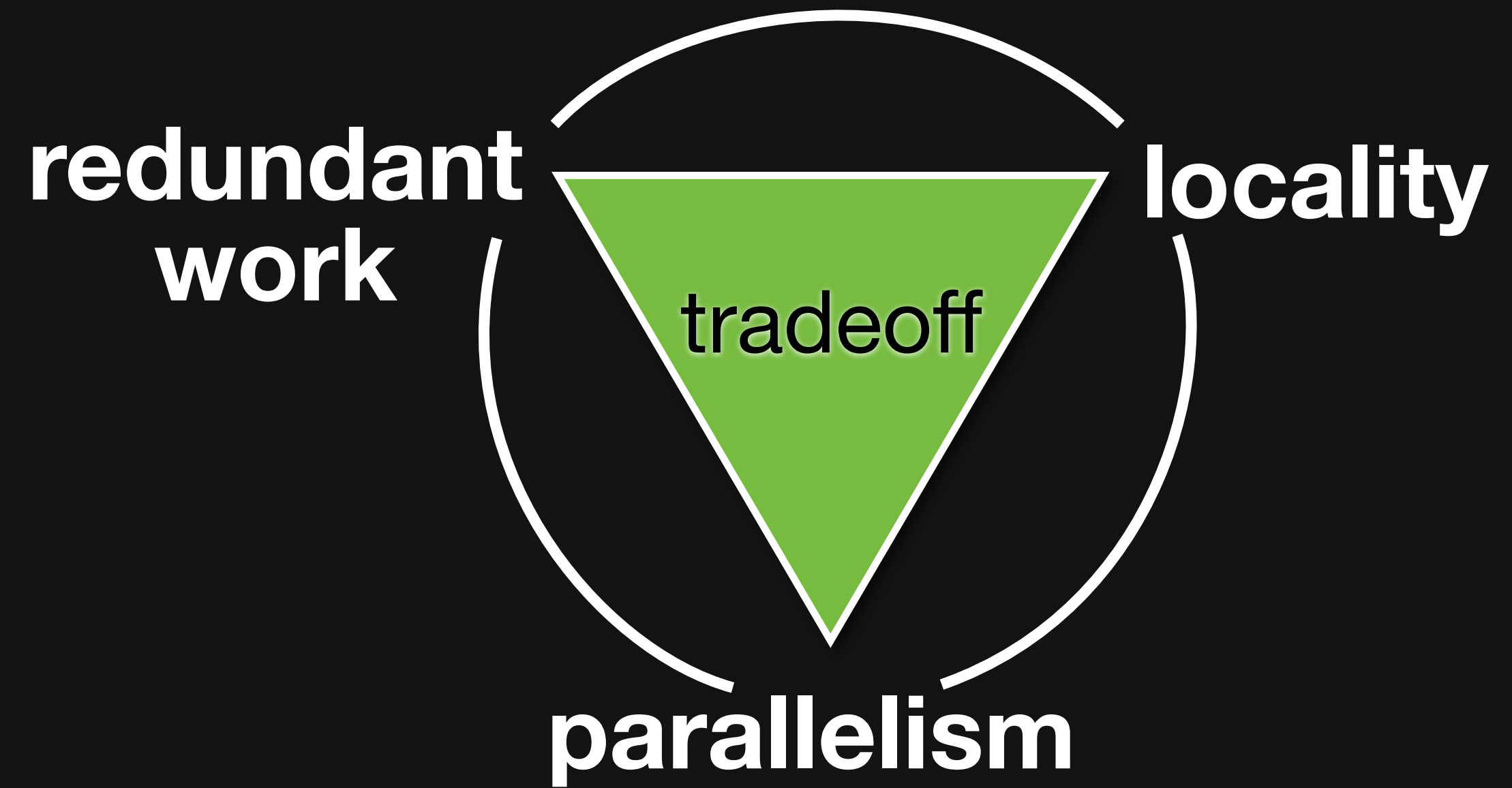


Root vs. Inline

Scheduling is often about finding a compromise halfway between root and inline

good locality like inline

limit redundancy like root



More general box blur, 5x5, 35 MPixels on 12

default schedule (inline)

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

Producer-consumer

First schedule consumer (start with output)

so far trivial, root (more interesting soon)

For producer, at what granularity of the consumer is it computed?

root: prepare all producer values needed for entire image

inline: produce on the fly, for each individual consumer pixel
(lots of redundancy but super local)

more options soon

Once you specify consumer granularity

Halide figures out how much is needed, allocates buffers, etc.

Then maybe schedule within producer

Tiling

Tiling (within)

Split the domain into inner and outer loops

Specified with `Func.tile()`

requires new Vars for inner and outer indices

Tiling (within stage)

Split the domain into inner and outer loops

Specified with `Func.tile()`

requires new Vars for inner and outer indices

Let's look at a single-stage pipeline function that only blurs vertically

```
Var x, y; Func blur_y;
```

```
blur_y(x,y) = (input(x,y)+input(x,y+1)+input(x,y+2))/3;
```

```
Var xo, yo, xi, yi;
```

```
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32);
```

Tiling: equivalent

allocate out (width, height)

Tiling: equivalent

```
allocate out (width, height)
for yo in (0..height/32): #assuming height%32=0
  for xo in (0..width/256): #assuming width%256=0
    for yi in (0..32):
      y=yo*32+yi
      for xi in (0..256):
        x=xo*256+xi
        out[x,y] = (input[x,y]+input[x,y+1]+input[x,y+2])/3
```

Tiling: equivalent for non-integer multiple

```
width, height = input.width()-2, input.height()-2
out=numpy.empty((width, height))
for yo in (0..(height+31)/32): #+31 to get the ceiling int
    for xo in(0..(width+255)/256): #+255 to get the ceiling int

        for yi in(0..32):
            y=yo*32+yi
            # make sure y doesn't go beyond height
            # worst case we'll recompute the last rows multiple times
            if y>=height: y=height-1
            for xi in(0..256):
                x=xo*256+xi
                # make sure x doesn't go beyond width
                # worst case we'll recompute the last columns
                if x>=width: x=width-1
                out[x,y] = (input[x,y]+input[x,y+1]+input[x,y+2])/3
```

Why tile?

Parallelism

Soon: parallelism+locality+limited redundant work

Parallel

func.parallel(var)

turns the for of this var into a parallel for

vectorize

vectorize(var)

always innermost var

Use systematically

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

tile 256 x 256 + parallel + vector

took 1.10970659256 seconds

Tiling under the hood

Tiling under the hood

Tiling is a sugar for split+reorder

split y into y_o, y_i

split x into x_o, x_i

reorder from y_o, y_i, x_o, x_i

to y_o, x_o, y_i, x_i

Tiling & fusion

Tiling II (within+across)

Back to 2-stage pipeline, tile the consumer (blur_y)

what about the producer, blur_x?

Tiling II

Back to 2-stage pipeline, tile the consumer (blur_y)

what about the producer, blur_x?

default: inline

could do : compute_root + tile

but Halide will compute all tiles of blur_x before starting blur_y
bad locality

note that tile without compute_root doesn't make sense:
you can't tile something that's inline

**Instead, we want to compute each tile of blur_x before the
corresponding tile of blur_Y**

Tiling and Fusion (within and across tiles)

Split consumer into tiles and compute producer for the whole tile just before computing the consumer tile.

Specified using **.tile()** on consumer and **.compute_at()** on producer

compute_at inserts the producer loop at a specified level of the nested consumer loops

Halfway between root and inline:

~root within tile: produce full tile before consuming it

~inline across tile: compute a tile when it is needed

Tiling and Fusion (within and across tiles)

Often requires to enlarge the producer tile

Because the consumer footprint might need extra pixels

e.g. `blur_y` needs extra pixels vertically

Inferred automatically by Halide

Tiling and Fusion (within and across tiles)

Split consumer into tiles and compute producer for the whole tile just before computing the consumer tile.

Specified using **.tile()** on consumer and **.compute_at()** on producer

```
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3;  
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3;  
Var xo, yo, xi, yi;  
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32);  
blur_x.compute_at(blur_y, xo);
```

compute_at inserts the producer loop at a given level of the nested consumer loops (xo here, x coordinate of tile)

Tiling and Fusion: equivalent

Tiling and Fusion: equivalent

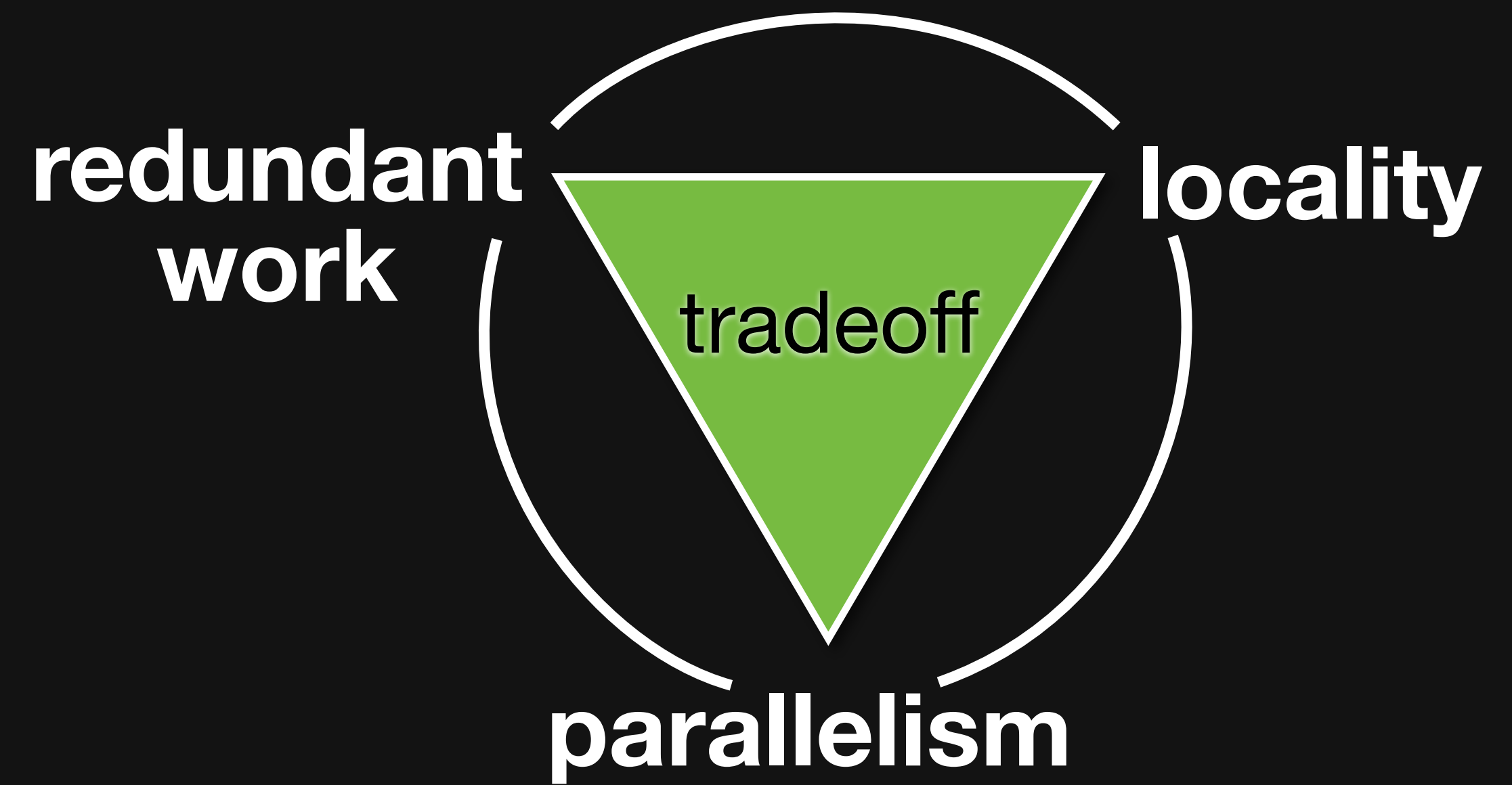
```
width, height = input.width()-2, input.height()-2
allocate out((width, height))
for yo in (0..(height+31)/32):
  for xo in (0..(width+255)/256):
    allocate tmp((256, 32+2))
    for yi in (0..32+2):
      y=yo*32+yi
      if y>=height: y=height-1
      for xi in (0..256):
        x=xo*256+xi
        if x>=width: x=width-1
        tmp[xi,yi]=(inputP[x,y]+inputP[x+1,y]+inputP[x+2,y])/3
    for yi in (0..32):
      y=yo*32+yi
      if y>=height: y=height-1
      for xi in (0..256):
        x=xo*256+xi
        if x>=width: x=width-1
        out[x,y] = (tmp[xi,yi]+tmp[xi,yi+1]+tmp[xi,yi+2])/3
```

Tiling and Fusion: equivalent

```
width, height = input.width()-2, input.height()-2
allocate out((width, height))
for yo in (0..(height+31)/32):      #loops over tile
  for xo in (0..(width+255)/256):
    allocate tmp((256, 32+2)) #tile to store blur_x. Note +2 for enlargement
    for yi in (0..32+2):      #loops for blur_x nested inside xo yo of blur_y
      y=yo*32+yi
      if y>=height: y=height-1 # for boundary tiles
      for xi in (0..256):
        x=xo*256+xi
        if x>=width: x=width-1 # for boundary tiles
        tmp[xi,yi]=(inputP[x,y]+inputP[x+1,y]+inputP[x+2,y])/3
                                #computation on x,y but store at xi, yi
    for yi in (0..32):      #loops for blur_y
      y=yo*32+yi
      if y>=height: y=height-1 # for boundary tiles
      for xi in (0..256):
        x=xo*256+xi
        if x>=width: x=width-1 # for boundary tiles
        out[x,y] = (tmp[xi,yi]+tmp[xi,yi+1]+tmp[xi,yi+2])/3
                                #computation on xi,yi but store at x, y (opposite of blur_x)
```

*indexing code affected by
schedule in green*

Tiling and fusion pros and cons

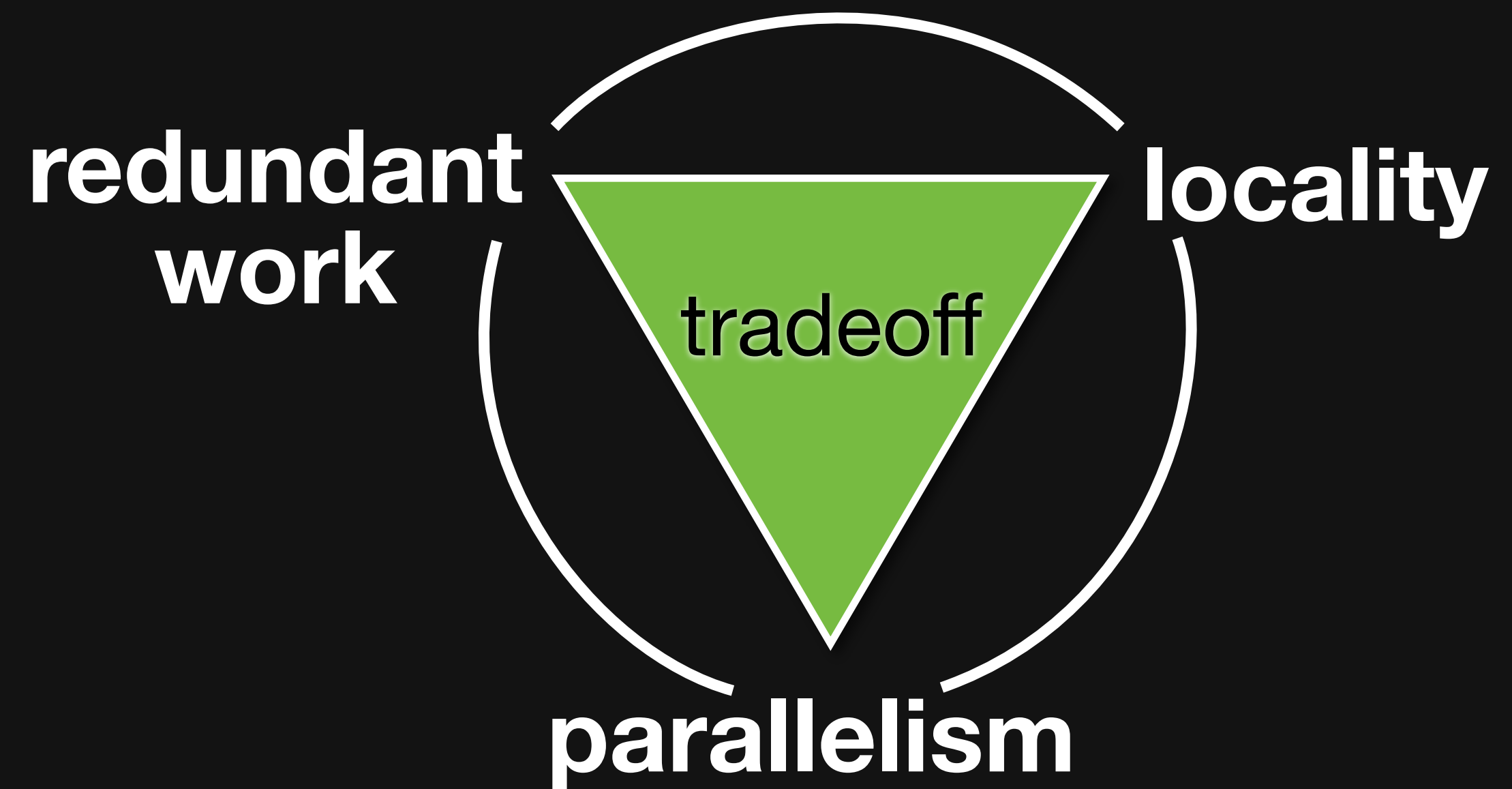


Tiling and fusion pros and cons

Scheduling is often about finding a compromise halfway between root and inline

good locality like inline

limit redundancy like root



Tile size controls the tradeoff

Performance

Final 3x3 blur: add parallelism and vectorization

```
Var x, y;
Fun blur_x, blur_y ;

blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;

Var xi, yi ;
blur_y.tile(x, y, xi, yi, 8, 4) \
    .parallel(y) \
    .vectorize(xi, 8);
blur_x.compute_at(blur_y, x) \
    .vectorize(x, 8);

output=blur_y.realize(input.width()-2, input.height()-2);
```

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

tile 256 x 256 + interleave

took 1.7552740097 seconds

tile 256 x 256 + parallel+vector

took 0.550438785553 seconds

tile 256 x 256 + parallel+vector without interleaving

took 1.10970659256 seconds

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

tile 256 x 256 + interleave

took 1.7552740097 seconds

tile 256 x 256 + parallel+vector

took 0.550438785553 seconds

tile 256 x 256 + parallel+vector without interleaving

took 1.10970659256 seconds

Note the exact doubling (memory bound)

Same as 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*

TODO

compute_at

figure out loop level

figure out consumer rectangle

figure out bound expansion for producer

introduce loops for this rectangle (as a function of within scheduling)

Warning

There are two hard things about Halide:

compute_at

reductions (histogram, convolutio, etc.)

compute_at : across

producer.compute_at(some_consumer, var)

compute_at

producer.compute_at(some_consumer, var)

**given the nested loops of the consumer,
performs computation of producer at loop for var**

**Halide infers everything needed by consumer for loops below
var**

i.e. rectangle xmin, xmax, ymin, ymax (and other coordinates if applicable)

The resulting rectangle can then be scheduled within

producer.compute_at(consumer, var_k)

```
for var1
```

```
...
```

```
for var_k
```

```
    //Insert computation of producer here
```

```
    // whatever is needed for the consumer loops below
```

```
    allocate temp buffer of appropriate size
```

```
    for producer_var1
```

```
        for producer_var2
```

```
            ...
```

```
            temp(var1, var2, ...) = ...
```

```
    for var_k+1
```

```
        ...
```

```
        for var_n
```

```
            consumer(index1, index2,...) =...
```

compute_at example

```
blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3;  
blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3;  
Var xo, yo, xi, yi;  
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32);  
blur_x.compute_at(blur_y,    );
```

compute_at example

```
blur_x[x,y] = (input[x,y]+input[x+1,y]+input[x+2,y])/3;  
blur_y[x,y] = (blur_x[x,y]+blur_x[x,y+1]+blur_x[x,y+2])/3;  
Var xo, yo, xi, yi;  
blur_y.tile(x, y, xo, yo, xi, yi, 256, 32);  
blur_x.compute_at(blur_y,    );
```

compute_at

producer.compute_at(some_consumer, var)

**given the nested loops of the consumer,
performs computation of producer at loop for var**

**Halide infers everything needed by consumer for loops below
var**

i.e. rectangle xmin, xmax, ymin, ymax (and other coordinates if applicable)

The resulting rectangle can then be scheduled within

Could be any consumer above the producer

typically in multi-stage pipeline,
there are key consumers that everybody compute_at

multistage(>2) pipeline

root & inline as compute_at

Extra scheduling options

Reorder

e.g. for x for y => for y for x

Split

e.g. for x => for x₀ for x_i

Tiling combines a set of splits and a reorder

Unroll

Vectorize

Parallel

some CUDA-specific

Final 3x3 blur: add parallelism and vectorization

```
Var x, y;  
Fun blur_x, blur_y ;  
  
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3.0;  
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3.0;  
  
Var xi, yi ;  
blur_y.tile(x, y, xi, yi, 8, 4) \  
    .parallel(y) \  
    .vectorize(xi, 8);  
blur_x.compute_at(blur_y, x) \  
    .vectorize(x, 8);  
  
output=blur_y.realize(input.width()-2, input.height()-2);
```

Parallel

func.parallel(var)

turns the for of this var into a parallel for

vectorize

Don't worry too much about vectorize for 6.815/6.865

It can be finicky

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*

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

tile 256 x 256 + interleave

took 1.7552740097 seconds

tile 256 x 256 + parallel+vector

took 0.550438785553 seconds

tile 256 x 256 + parallel+vector without interleaving

took 1.10970659256 seconds

More general box blur, 5x5, 35 MPixels on 12

default schedule

took 5.80736255646 seconds

root first stage

took 1.99803357124 seconds

tile 256 x 256 + interleave

took 1.7552740097 seconds

tile 256 x 256 + parallel+vector

took 0.550438785553 seconds

tile 256 x 256 + parallel+vector without interleaving

took 1.10970659256 seconds

Note the exact doubling (memory bound)