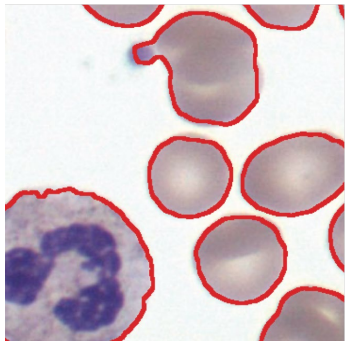
Domain Specific Languages: Halide

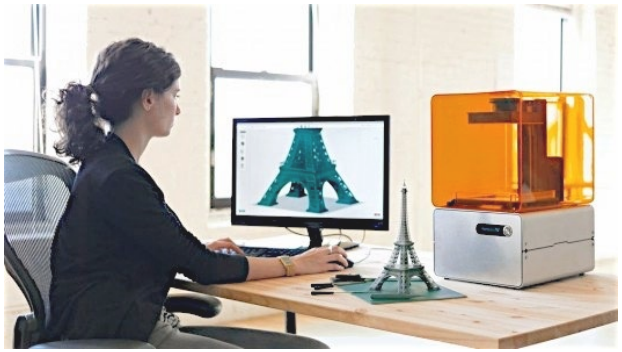
CS267 Lecture 12(a)

Alex Reinking, GSI

Slides adapted from Jonathan Ragan-Kelley

Visual computing is EVERYWHERE…





…but demands EXTREME performance!

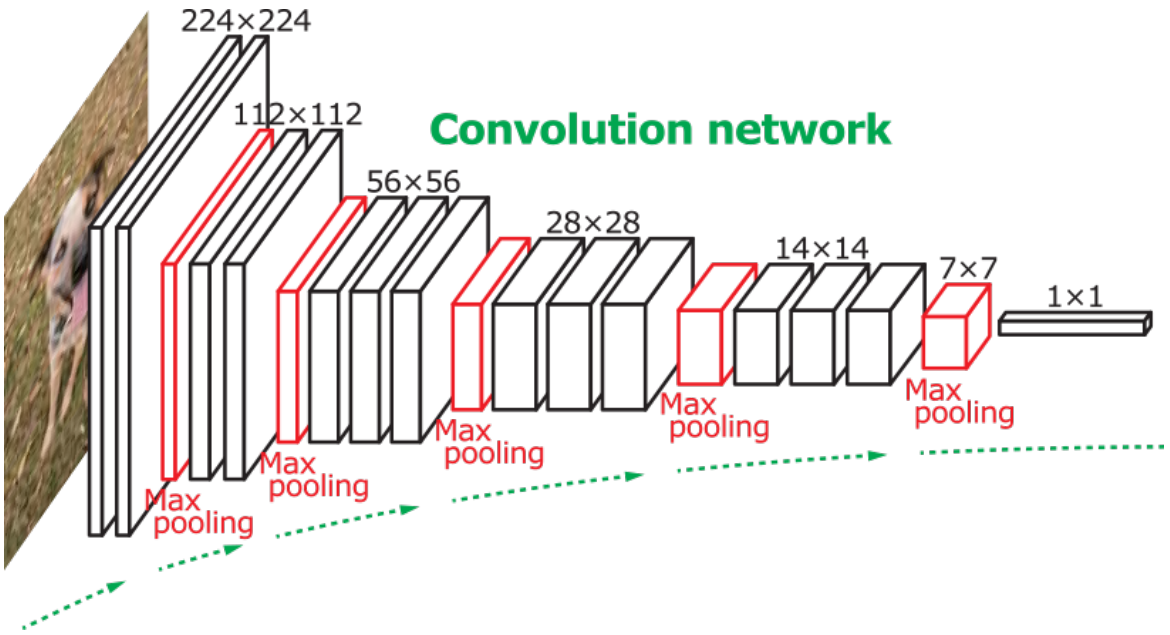
Modern game

Feature Film

10ms

5 hrs

Images by Valve, Weta



Visual data analysis is expensive

One object recognition neural net: **0.05 megapixels at video rate**

(Caffe + cuDNN on K40c running VCG16)
[Simonyan & Zisserman 2015]
[Image: Noh et. al. 2015]

Where does performance come from?

- We've talked about this a *bit*…
  - **Vectorization** (cf. HW1, AVX)
  - **Cache locality** (cf. HW1, tiling/blocking)
  - **Multicore parallelism** (cf. HW2.1, OpenMP)
  - **Communication avoidance** (cf. HW2.2, MPI)

- As we learned with DGEMM, it's not just the algorithm, it's the ***organization of the computation.***
  - Particulars of the hardware matter to picking a good organization

Communication typically dominates compute

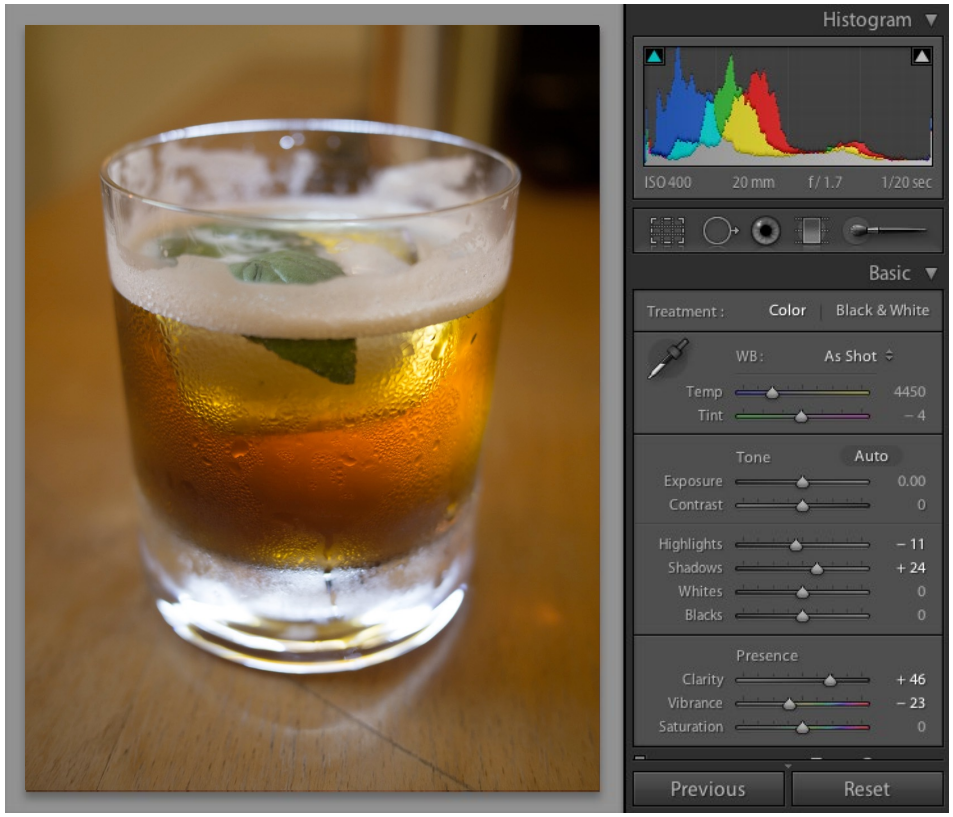
Operation (32bit ops)	Latency	Cost vs. ALU
32bit ADD	0.5 ns	-
L1 cache	1 ns	2x
FMA	2.5 ns	5x
L2 cache	4 ns	8x
L3 cache	40 ns	80x
RAM (DRR DIMMs)	80 ns	160x
NVMe SSD	25 **µs**	50,000x
LTE communication	30 **ms**	60,000,000x

Clearly, shipping everything to the cloud won't cut it!

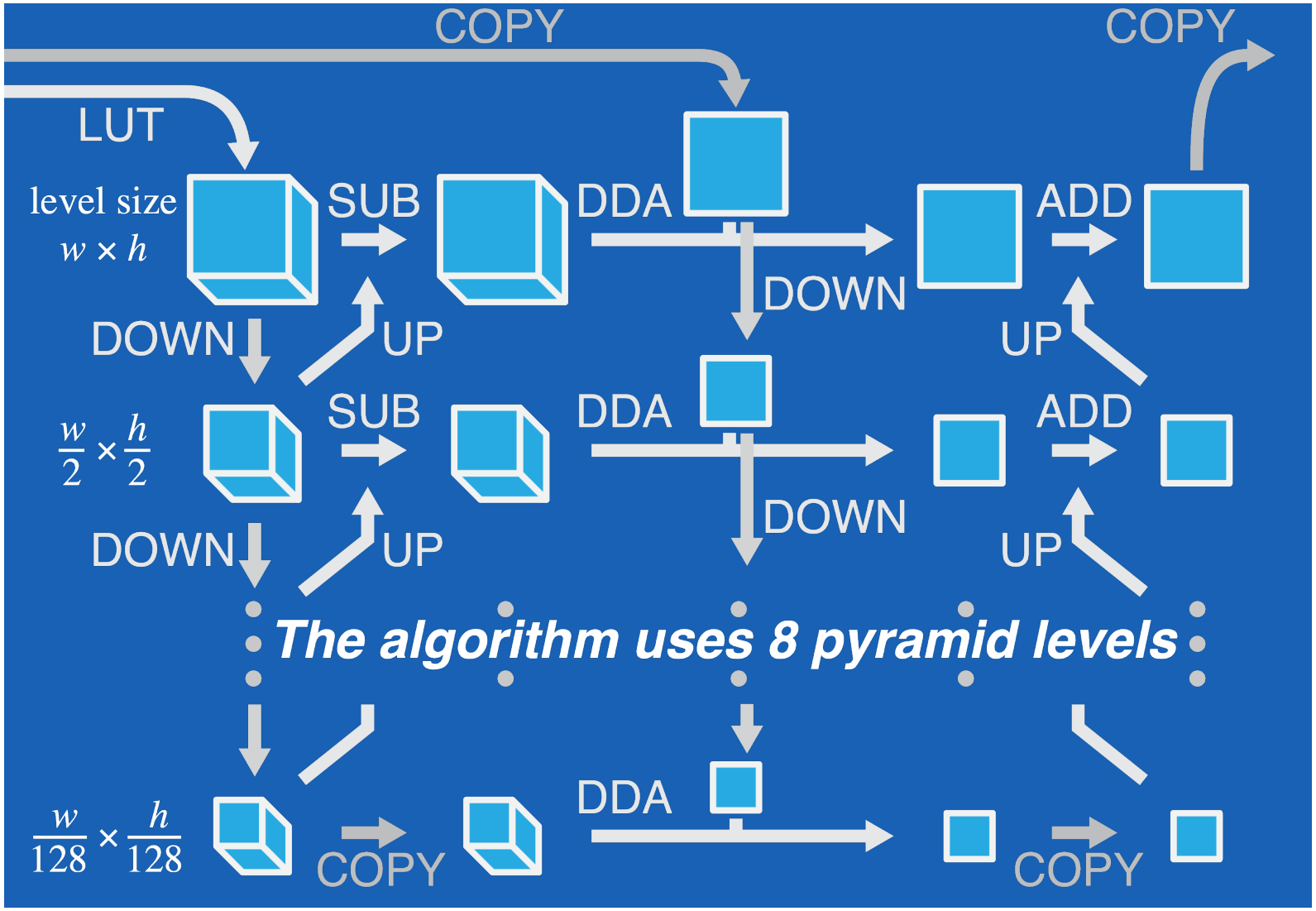
https://software.intel.com/en-us/articles/memory-performance-in-a-nutshell
https://www.agner.org/optimize/instruction_tables.pdf
https://www.storagereview.com/samsung_960_pro_m2_nvme_ssd_review
http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html

Writing high-performance code is hard!

- Reference:
  - 300 lines of C++
- Adobe:
  - 1500 lines
  - 3 months of work
  - 10x faster
- Same algorithm, different organization!



Global reorganization breaks modularity



Introducing…

Halide: a Language for Numerical Processing

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;
```

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;
```

Same algorithm, only now it's 15x faster!

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

Same algorithm, only now it's 15x faster!

Key idea: organize computation *across* the loops

Hand-optimized C++

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

- Tiling, fusing, SSE
- Multi-threaded
- Redundant computation
- Near roofline optimum
- 11x faster (quad core)

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;
```

Problem: optimized C++ code is not *readable*, *portable*, or *maintainable*.

Hand-optimized C++

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

Parallelism across threads and SIMD vector lanes.

Hand-optimized C++

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

Parallelism across threads and SIMD vector lanes.

Locality in tiled computation and fusion of the two blurs.

(Re)organizing computation is hard!

- Optimizing for parallelism and locality requires **transforming the program and its data structures.**

- What transformations are ***legal***?

- What transformations are ***beneficial***?

- Highly tuned libraries don't solve this:
  - BLAS, IPP, MKL, OpenCV, MATLAB
  - Optimized kernels compose into inefficient pipelines
  - Why? No fusion!

Halide's answer

Decouple the ***algorithm*** from the ***schedule***!

Algorithm:

what is computed

Schedule:

where and ***when*** it is computed

Algorithm: define pipelines as pure functions

- Pipeline stages are **pure 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

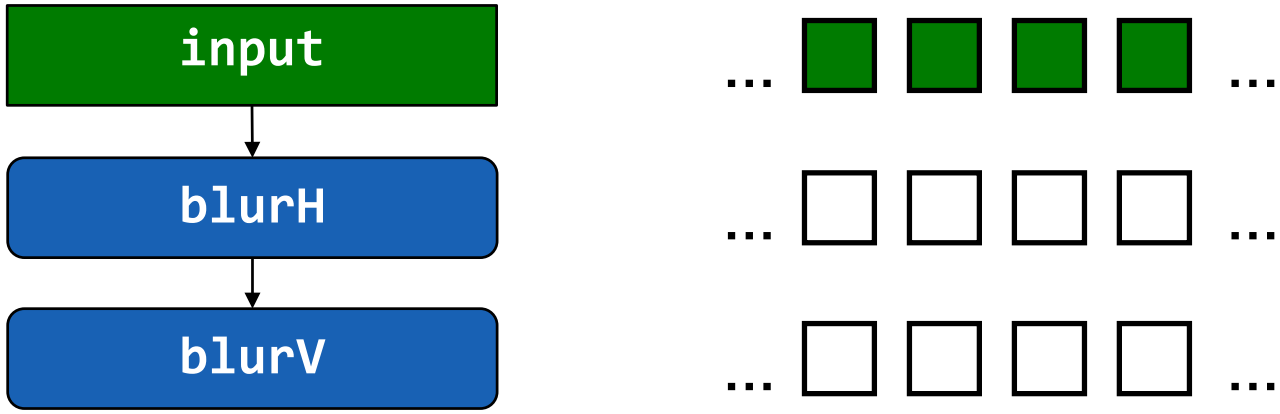
- All computation is over **multidimensional arrays**
- Only **feed-forward pipelines** (with escape hatches)
- **Iteration** must have **bounded depth**

- Dependence must be **inferable**
  - User can clamp to impose tight bounds, if needed
- Long, **heterogeneous** pipelines
  - Complex graphs, deeper than traditional stencils

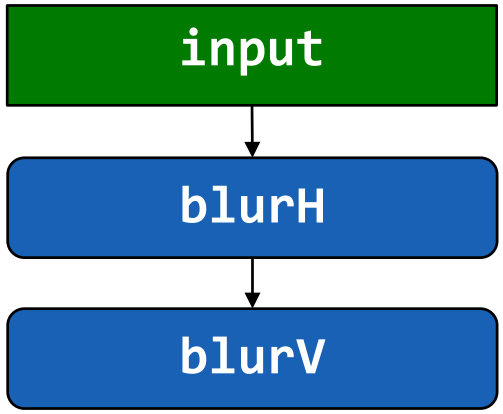
The question is then:

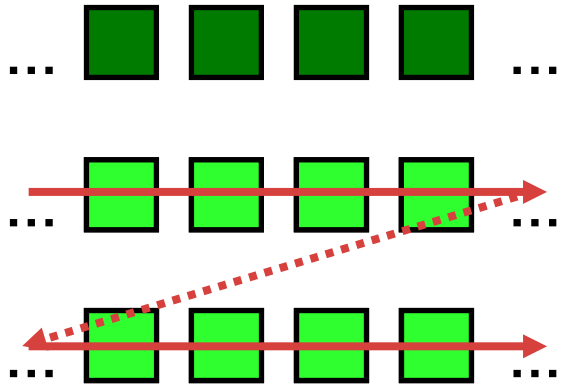
How can we schedule this?

Organizing a data-parallel pipeline

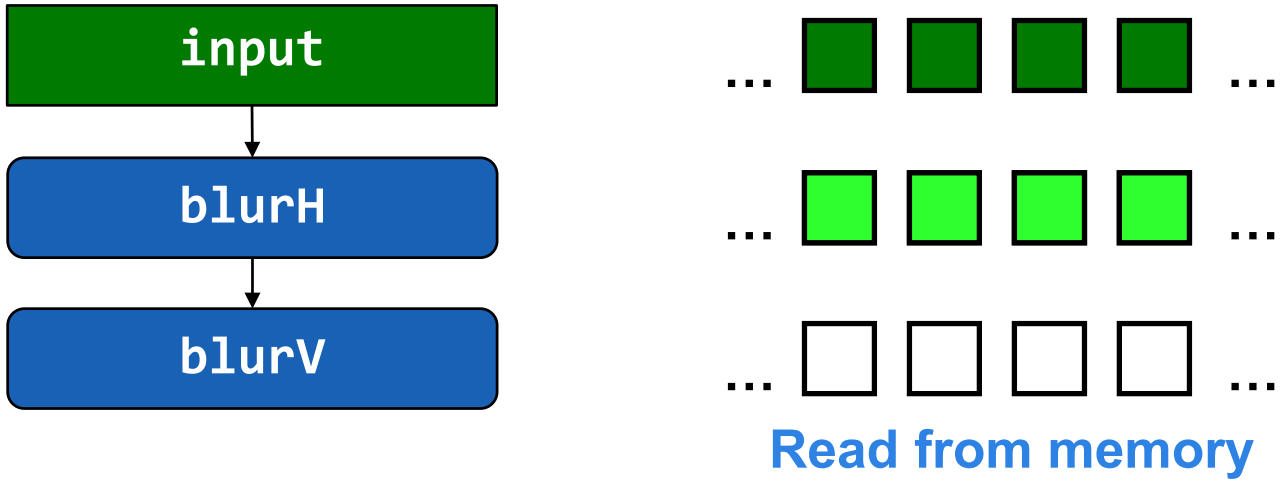


Breadth-first execution sacrifices locality.

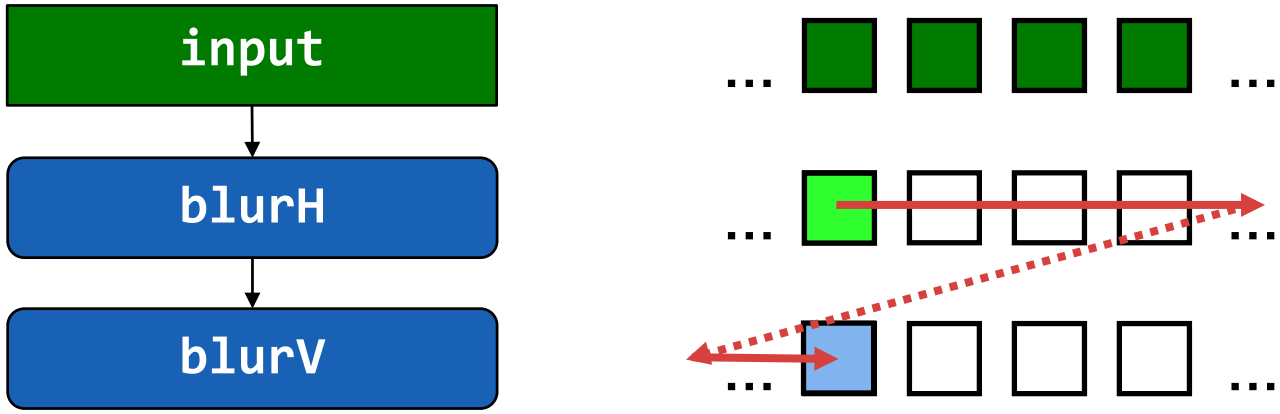




Breadth-first execution sacrifices locality.

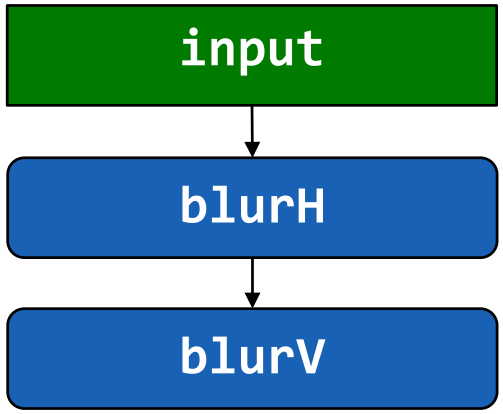


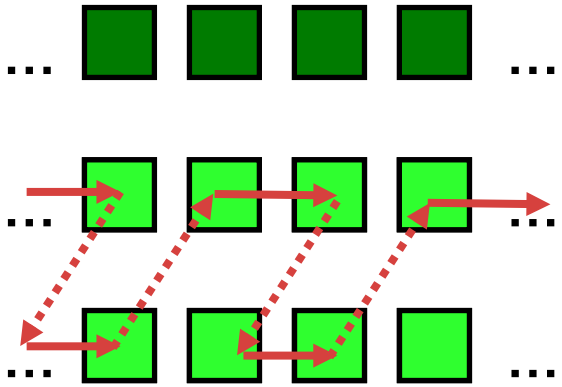
Breadth-first execution sacrifices locality.



Locality is a function of reuse distance!

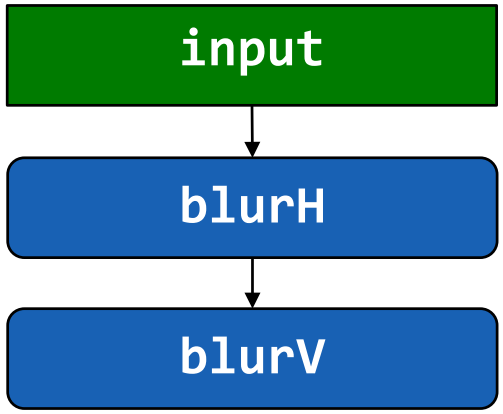
Interleaving (fusion) improves locality

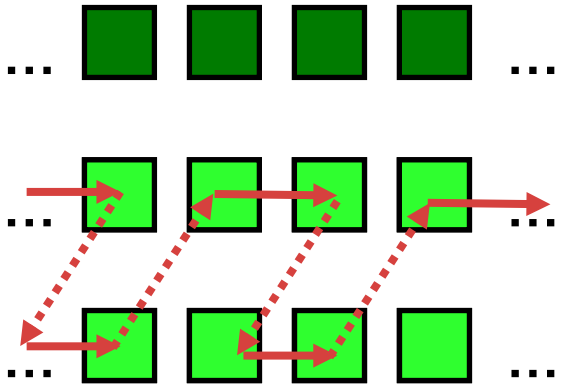




Reduce reuse distance from *producer* to *consumer!*

Interleaving (fusion) improves locality

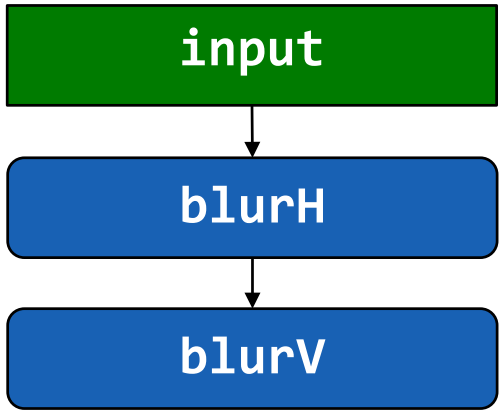


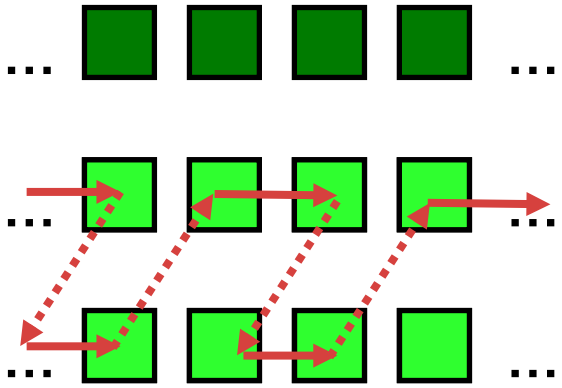


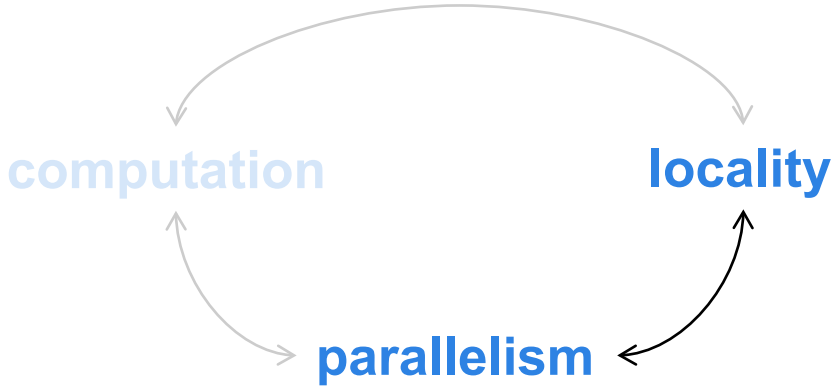
Reduce reuse distance from *producer* to *consumer!*

But this makes optimization a global problem of carefully interleaving the whole pipeline!

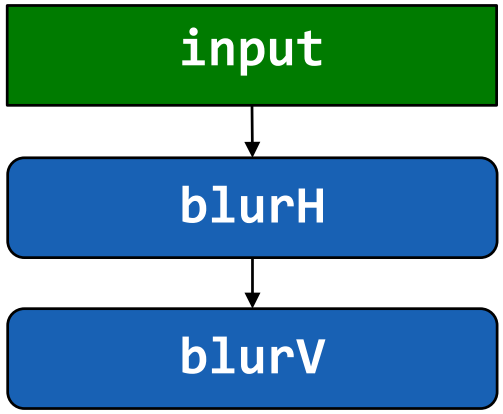
Interleaving (fusion) improves locality

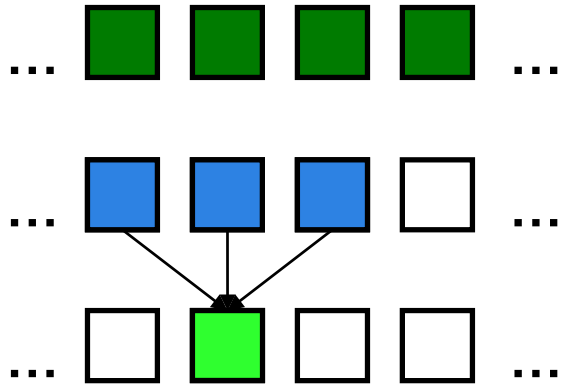


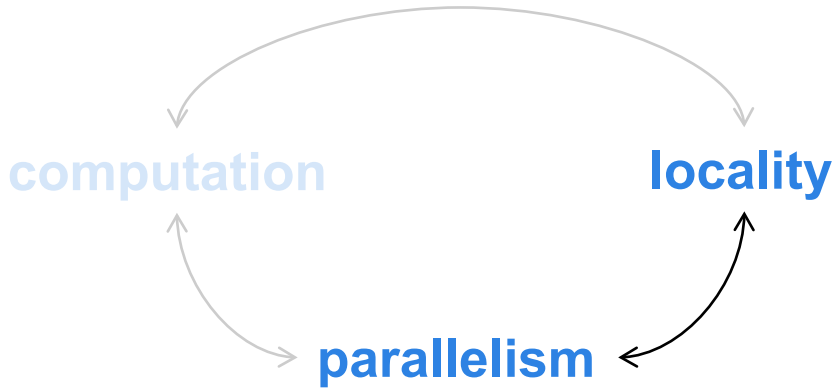




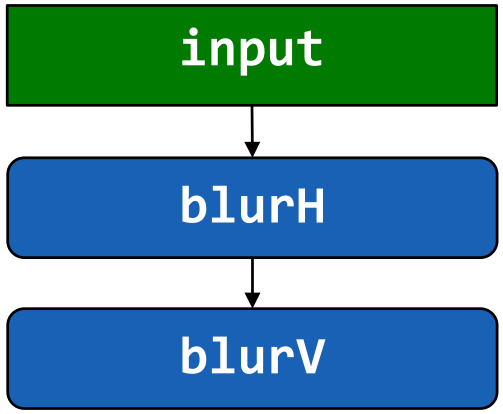
Understanding dependencies

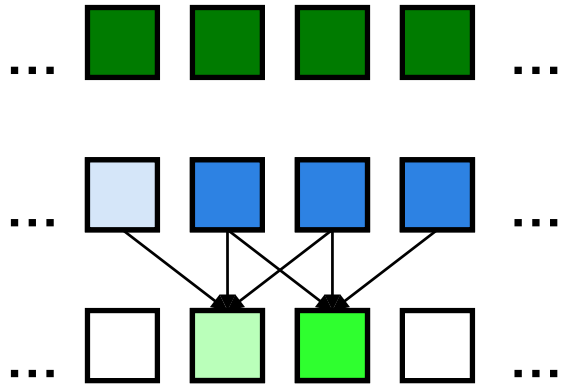


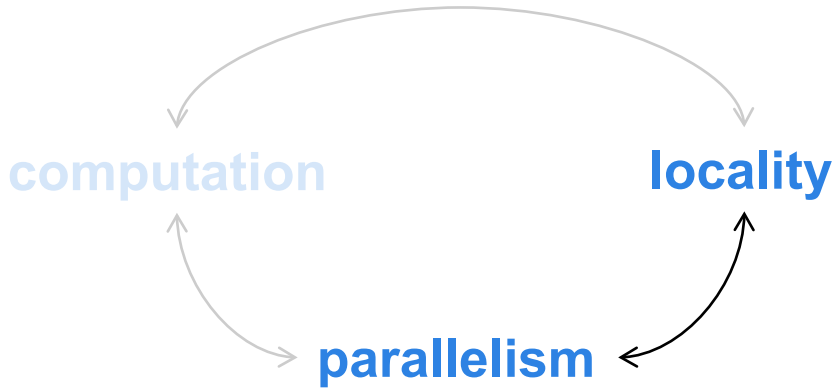




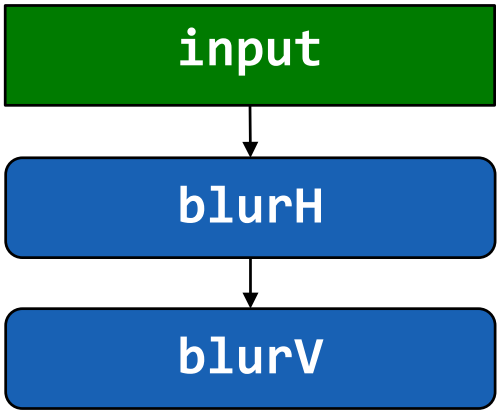
Understanding dependencies

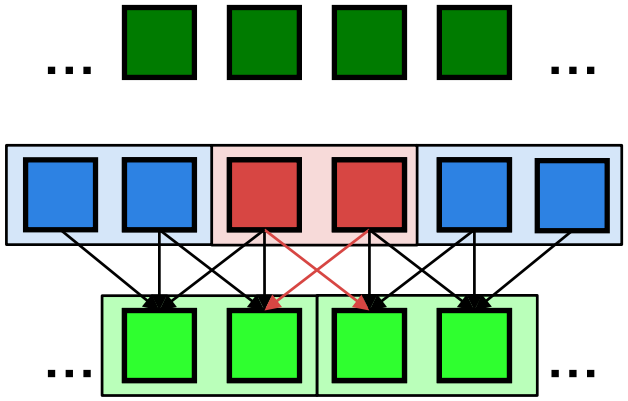


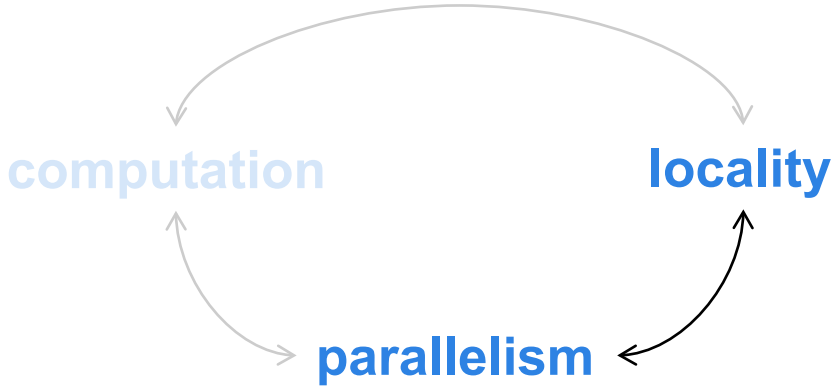




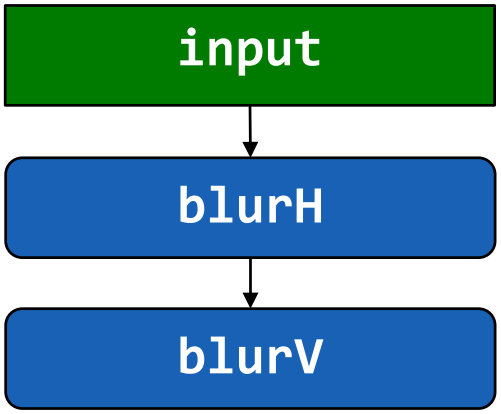
Understanding dependencies

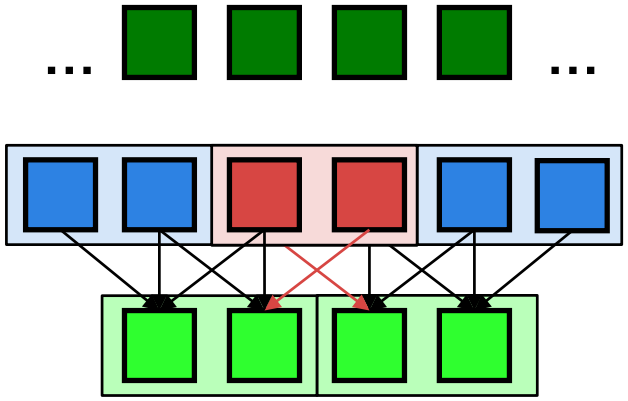


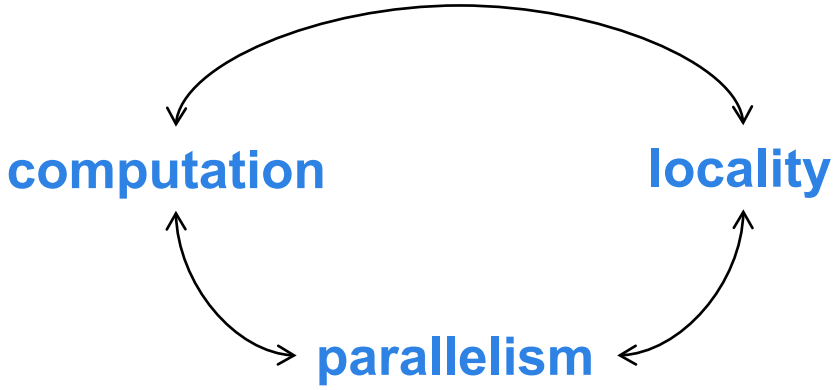




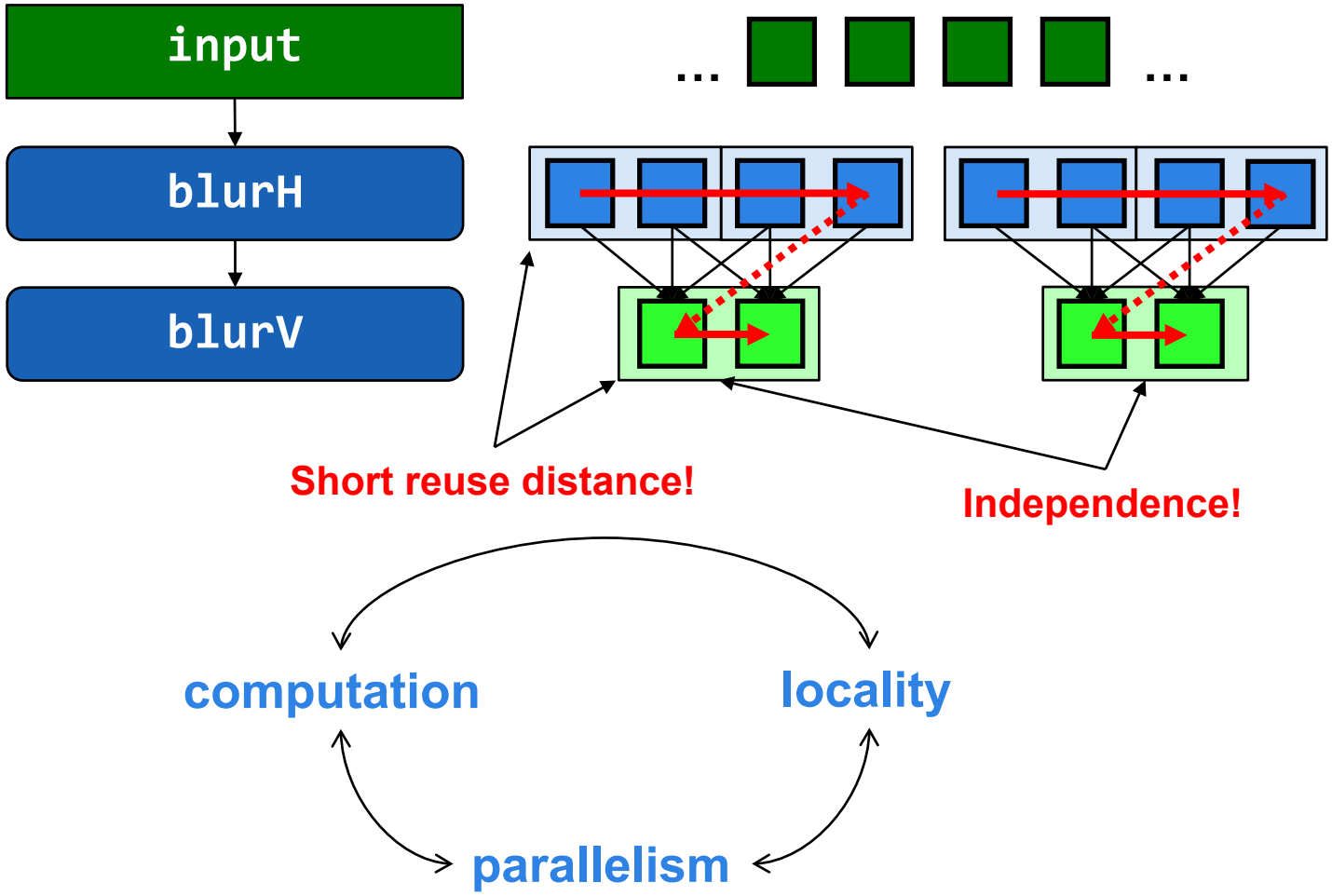
Understanding dependencies



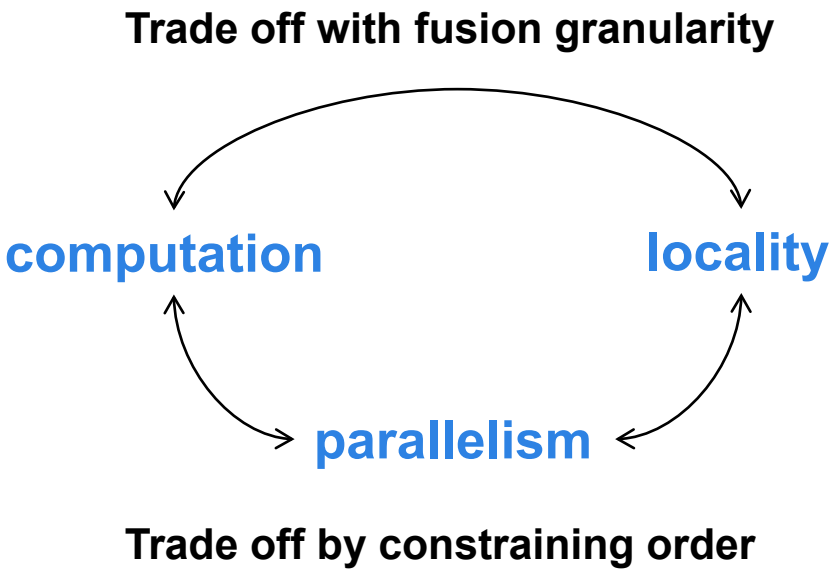




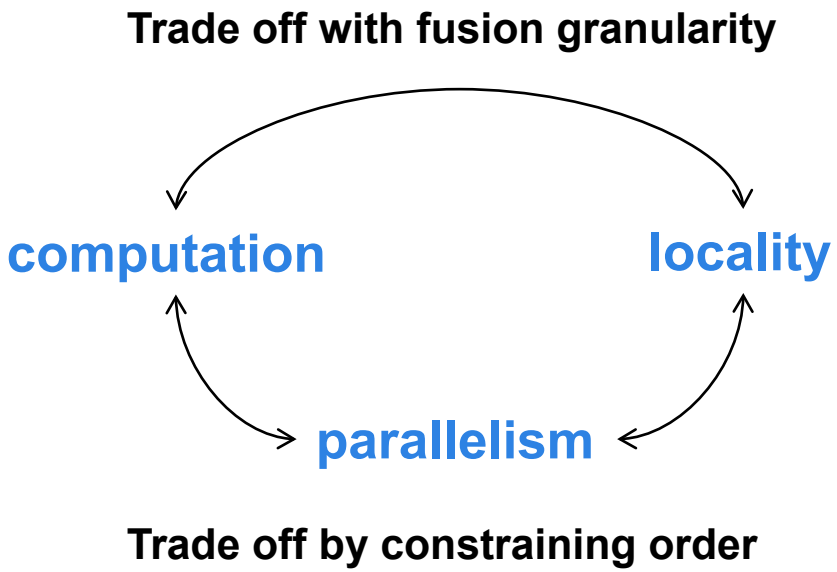
Breaking dependencies means extra work



Main lesson: performance requires trade-offs



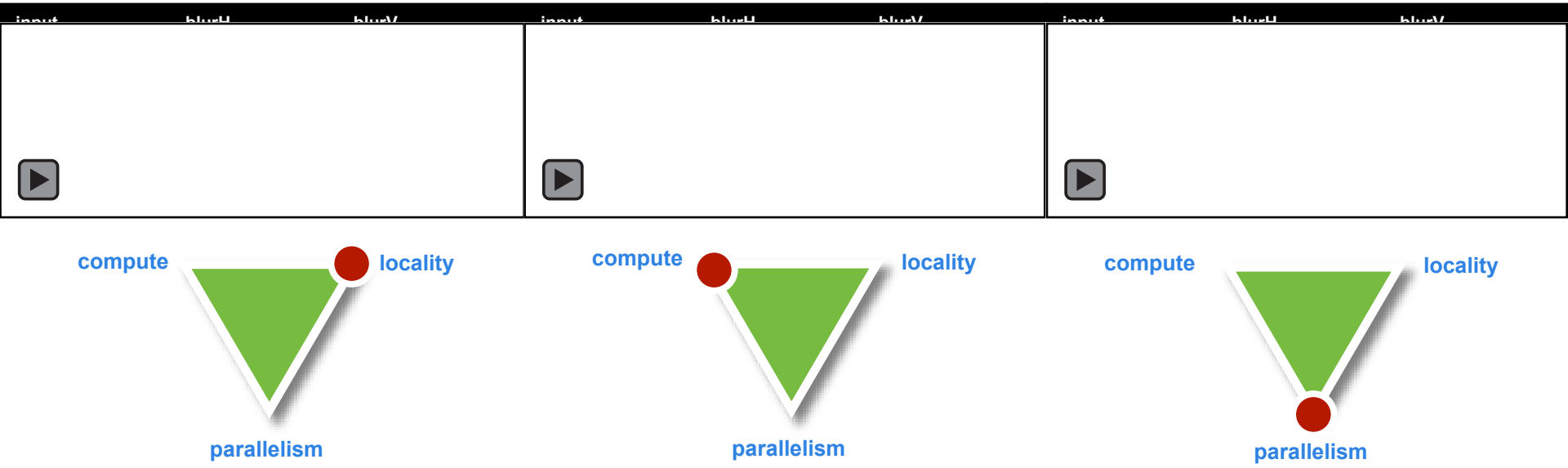
Main lesson: performance requires trade-offs

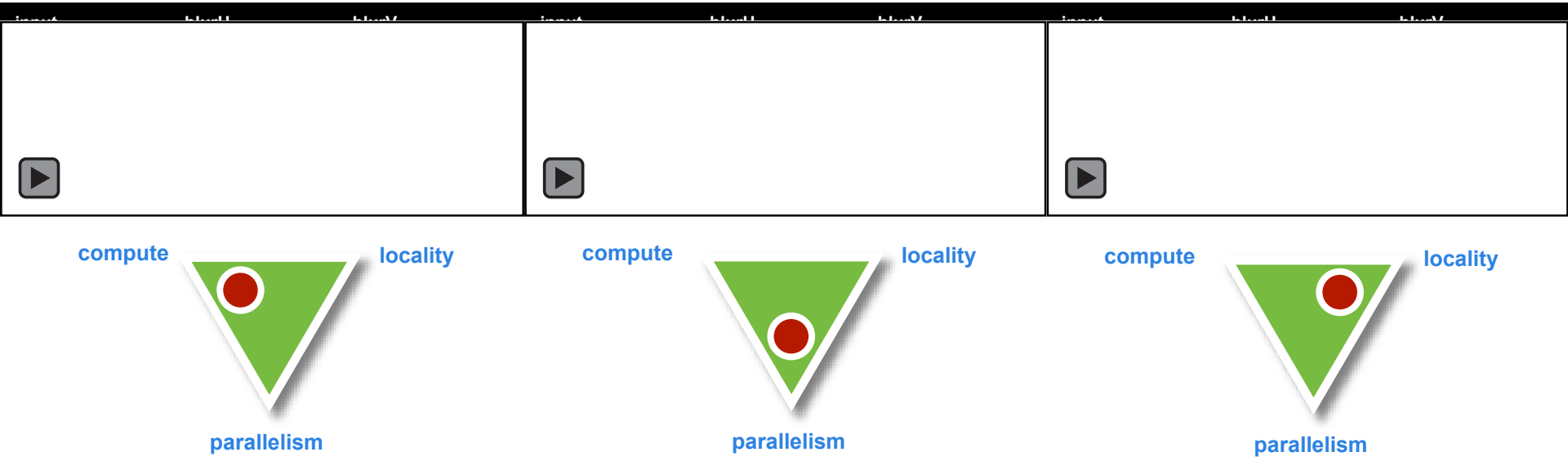


Trade offs determined by organization.
Organizations constrained by dependencies.

A Language of Schedules

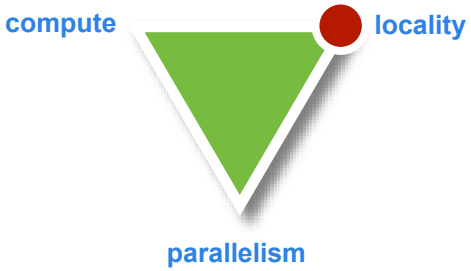
A multitude of organizations



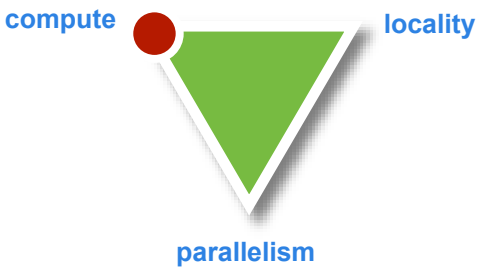


A multitude of organizations

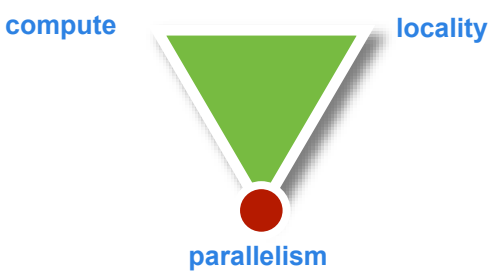
```
blurH.compute_root()
```




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

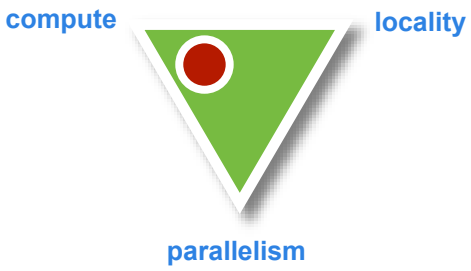



```
blurH.compute_at(blurV, x)
     .store_root()
```

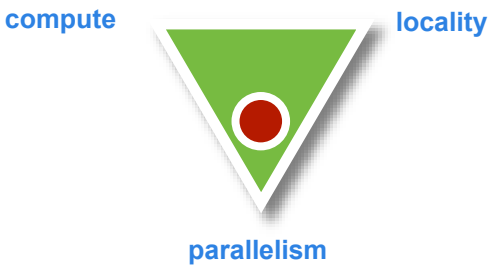



```
blurH.compute_at(blurV, x)
     .vectorize(x, 4)

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

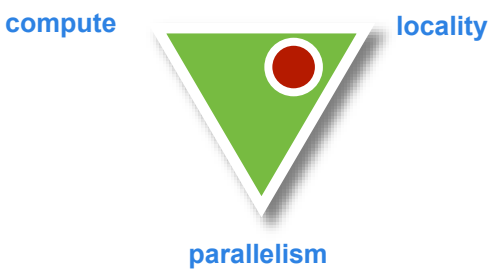



```
blurH.compute_at(blurV, y)
     .store_root()
     .split(x, x, xi, 8)
     .vectorize(xi, 4)
     .parallel(x)
blurV.split(x, x, xi, 8)
     .vectorize(xi, 4)
     .parallel(x)
```




```
blurH.compute_at(blurV, y)
     .store_at(blurV, yi)
     .vectorize(x, 4)

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



Algorithm stages are loop nests

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

```
for blurV.y:
  for blurV.x:
    blurV(x, y) = /* … */
```

An algorithm stage like this one corresponds to the loop nest on the right.

Algorithm stages are loop nests

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

```
for blurV.y:
  for blurV.x:
    blurV(x, y) = /* … */
```

The actual computation isn't touched by scheduling

Algorithm stages are loop nests

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

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

Schedules modify these loop nests. Tiling introduces new loops, for instance.

```
// for each tile
for blurV.yo:
  for blurV.xo:
    // for each pixel in tile
    for blurV.yi:
      for blurV.xi:
        let x = /* … */
        let y = /* … */
        blurV(x, y) = /* … */
```

Algorithm stages are loop nests

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

blurV.tile(x, y, xo, yo, xi, yi,
           32, 32);

blurH(x, y) = (in(x - 1, y)
             + in(x, y)
             + in(x + 1, y))/3;

blurH.compute_at(blurV, xo);
```

```
// for each tile
for blurV.yo:
  for blurV.xo:
    for blurH.y:
      for blurH.x:
        blurH(x, y) = /* … */
    // for each pixel in tile
    for blurV.yi:
      for blurV.xi:
        let x = /* … */
        let y = /* … */
        blurV(x, y) = /* … */
```

“Compute-at” controls the level of granularity

Algorithm stages are loop nests

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

blurV.tile(x, y, xo, yo, xi, yi,
           32, 32);

blurH(x, y) = (in(x - 1, y)
             + in(x, y)
             + in(x + 1, y))/3;

blurH.compute_at(blurV, xo);

blurV.parallel(yo);
```

```
// for each tile
parallel for blurV.yo:
  for blurV.xo:
    for blurH.y:
      for blurH.x:
        blurH(x, y) = /* … */
    // for each pixel in tile
    for blurV.yi:
      for blurV.xi:
        let x = /* … */
        let y = /* … */
        blurV(x, y) = /* … */
```

Can add OpenMP-style parallelism.

Algorithm stages are loop nests

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

blurV.tile(x, y, xo, yo, xi, yi,
           32, 32);

blurH(x, y) = (in(x - 1, y)
             + in(x, y)
             + in(x + 1, y))/3;

blurH.compute_at(blurV, xo);

blurV.parallel(yo);

blurH.vectorize(x, 8);
```

```
// for each tile
parallel for blurV.yo:
  for blurV.xo:
    for blurH.y:
      for blurH.x:
        blurH(x, y) = /* … */
    // for each pixel in tile
    for blurV.yi:
      vector for blurV.xi:
        let x = /* … */
        let y = /* … */
        blurV(x:x+8, y) = /* … */
```

Can automatically vectorize innermost loops.

Algorithm stages are loop nests

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

blurV.tile(x, y, xo, yo, xi, yi,
           32, 32);

blurH(x, y) = (in(x - 1, y)
             + in(x, y)
             + in(x + 1, y))/3;

blurH.compute_at(blurV, xo);

blurV.parallel(yo);

blurH.vectorize(x, 8);
```

```
// for each tile
parallel for blurV.yo:
  for blurV.xo:
    for blurH.y:
      for blurH.x:
        blurH(x, y) = /* … */
    // for each pixel in tile
    for blurV.yi:
      vector for blurV.xi:
        let x = /* … */
        let y = /* … */
        blurV(x:x+8, y) = /* … */
```

Handles boundary by either (a) switching to serialized code; (b) shifting the loop inwards; (c) rounding up and overcomputing.

Halide vs. C++

```
Func box_filter3x3(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 - order, locality, 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;
}
```

```
void box_filter_3x3(const Image &in, Image
  __m128i one_third = _mm_set1_epi16(21846
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height();
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allo
    for (int xTile = 0; xTile < in.width()
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__m128i*)(i
          b = _mm_loadu_si128((__m128i*)(i
          c = _mm_load_si128((__m128i*)(in
          sum = _mm_add_epi16(_mm_add_epi1
          avg = _mm_mulhi_epi16(sum, one_t
          _mm_store_si128(blurxPtr++, avg)
          inPtr += 8;
        }
      }
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(bl
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*2
          b = _mm_load_si128(blurxPtr+256/
          c = _mm_load_si128(blurxPtr++);
```

Halide vs. C++

```
Func box_filter3x3(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 - order, locality, 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;
}
```

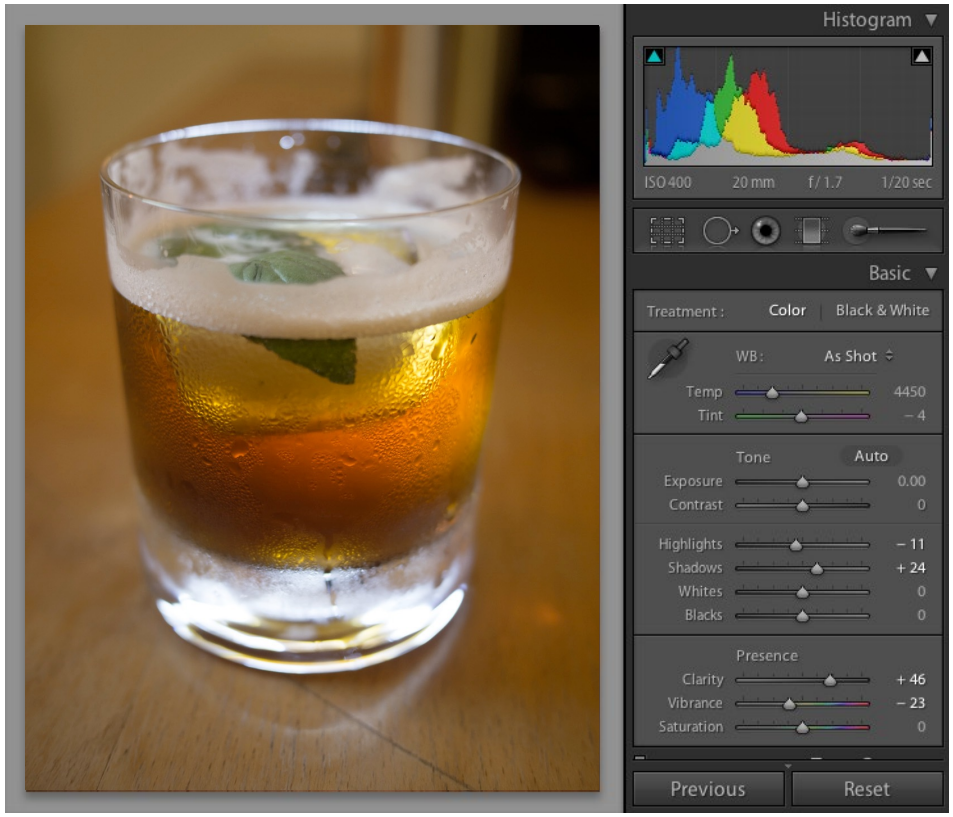
```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32)
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 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);
        }
      }
    }
  }
}
```

What does the scheduling language promise?

- Programmers can write schedules without fear of introducing a bug
- This is because the algorithm language is restricted.
- “Pure” means recomputation is always possible and so parallelism is achievable (upon request)
- **Caveat:** All this contingent on bounds analysis doing the right thing, but it does in practice.

Writing high-performance code is (less) hard!

- Adobe:
  - 1500 lines
  - 3 months of work
  - 10x faster than original C++
- Halide: 60 lines.
  - 1 intern-day
- Halide 2x faster still
- 10x faster on GPU



Halide in sum

- Decouples algorithm from organization through a correctness-preserving scheduling language
- Simpler programs that are easier to maintain.
- Faster than hand-tuned C/C++ code.
- Changing out the schedule lets you quickly target new hardware/architectures.

Learn more at:

https://halide-lang.org

Thanks! Questions?

Bonus slide: cutting edge

- ML-based autoscheduler [Adams 2019]
  - Based on AlphaGo. Outperforms human experts.
- Translating assembly into Halide [Ahmad 2019]
  - Certifies that the translation is correct using formal methods
- Halide-to-ASICs/FPGAs [Vocke 2017]
  - Co-optimizing hardware and schedule = big wins!
- Differentiable Halide [Li 2018]
  - Fast neural network training!

Bonus slide: matrix multiply

Halide is able to express matrix multiplication through a special feature called *reduction domains*

The idea is that you can run an impure loop innermost without messing up too much of the scheduling language.

Try to schedule:

```
RDom k(0, n);
C(i, j) = 0; // initialize to 0
C(i, j) += A(i, k) * B(k, j); // accumulate
```