Fast Image Processing using Halide

Andrew Adams (Google)
Jonathan Ragan-Kelley (Berkeley)
Zalman Stern (Google)
Steven Johnson (Google)
Dillon Sharlet (Google)
Patricia Suriana (Google)

This talk

- The challenges involved in fast image processing
- How we tackled these problem using a language
- How that worked out for us in practice at Google
- General lessons learned

Pixel smartphone camera review: At the top

By David Cardinal - Tuesday October 04 2016

Mobile Review

DxOMark Mobile 89

Mobile Scores

89	Google Pixel
88	HTC 10
	Samsung Galaxy S7 Edge
	Sony Xperia X Perf.
87	Moto Z Force

“The highest-rated smartphone camera we have ever tested”

Writing fast image processing pipelines is hard.

Halide is a language that makes it easier.

Big idea: separate algorithm from optimization
programmer defines both
algorithm becomes simple, modular, portable
exploring optimizations is much easier

C/C++ is slow

```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());  // allocate temporary 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;
}
```

9.96 ms/megapixel
(quad core x86)

An optimized implementation is 11x faster

```
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** than a naïve implementation*

0.9 ms/megapixel
(quad core x86)

An optimized implementation is 11x faster

```
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

distribute across threads

SIMD parallel vectors

0.9 ms/megapixel

(quad core x86)

An optimized implementation is 11x faster

```
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

distribute across threads
SIMD parallel vectors

locality

compute in tiles
interleave tiles of blurx, blury
store blurx in local cache

0.9 ms/megapixel
(quad core x86)

An optimized implementation is 11x faster

```
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

distribute across threads
SIMD parallel vectors

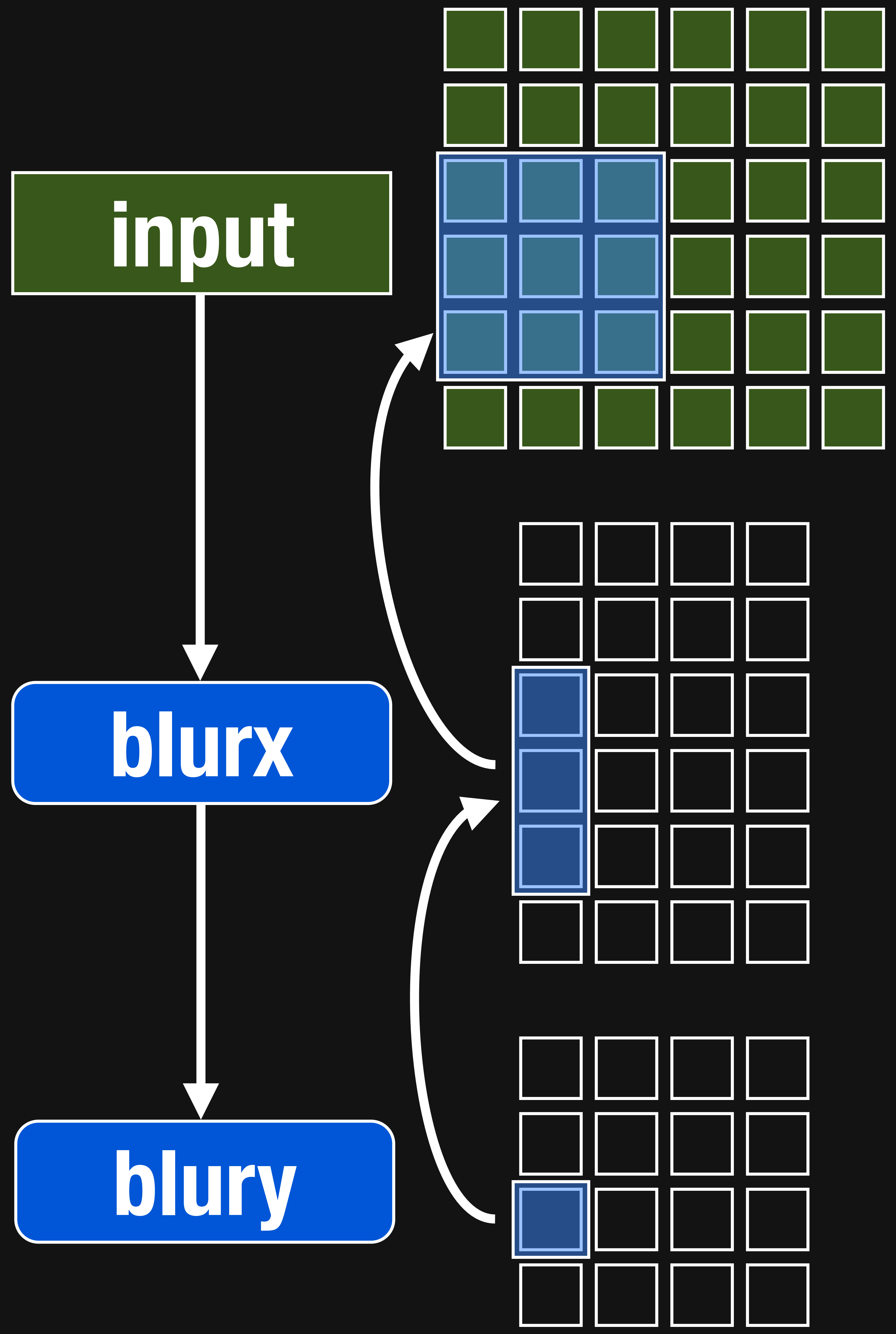
locality

compute in tiles
interleave tiles of blurx, blury
store blurx in local cache

0.9 ms/megapixel
(quad core x86)

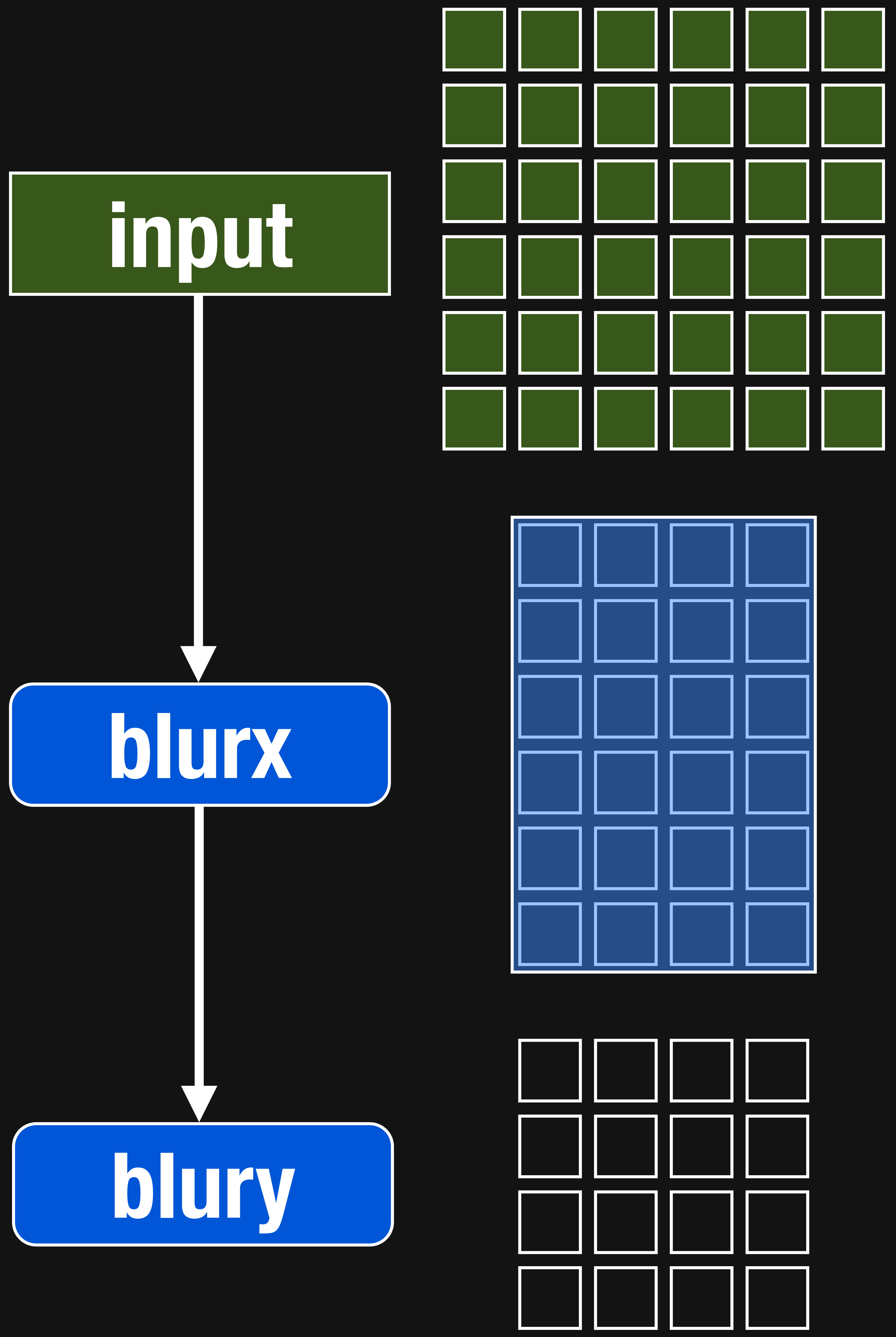
Executing the pipeline

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



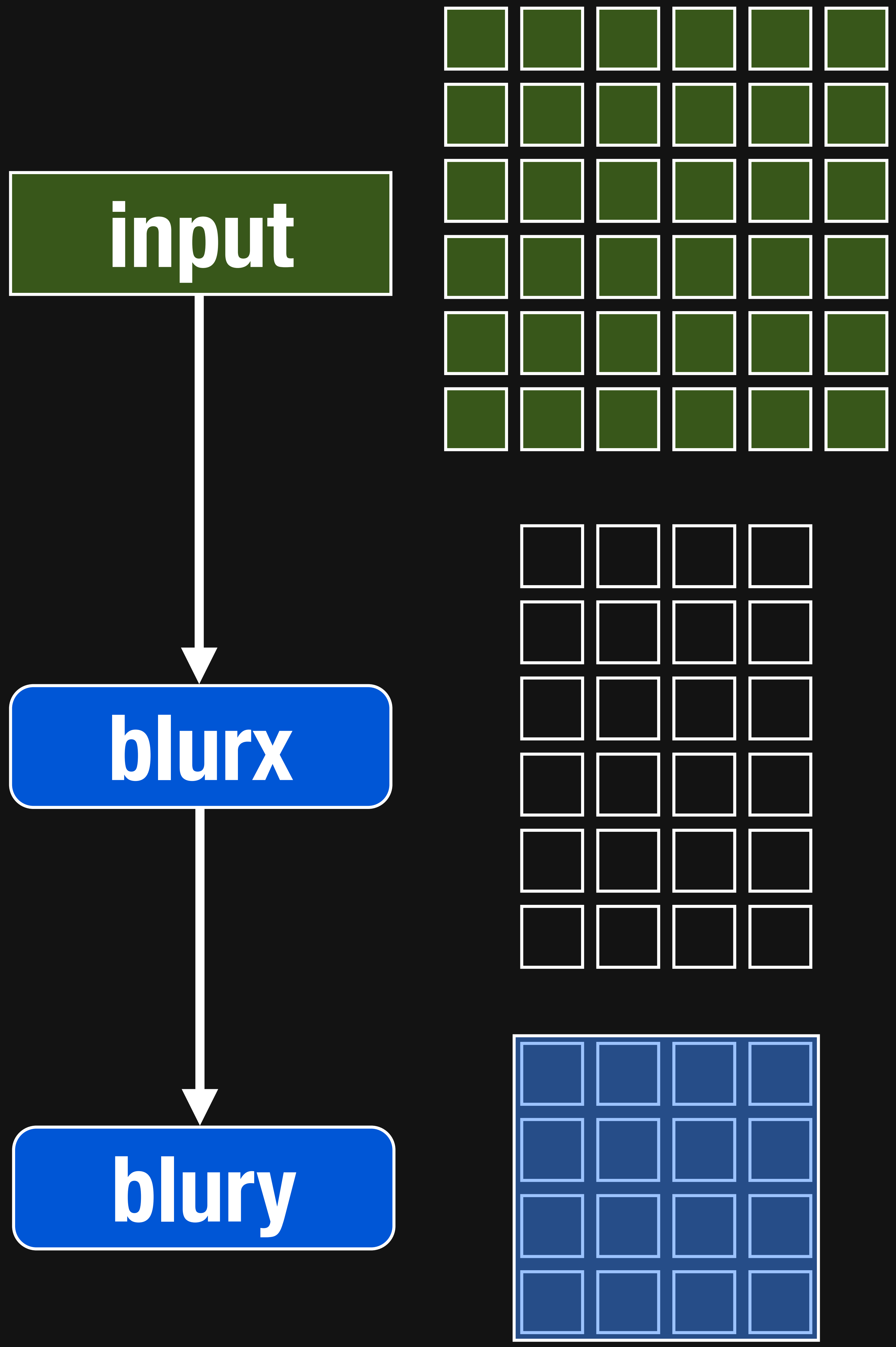
Executing the pipeline

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



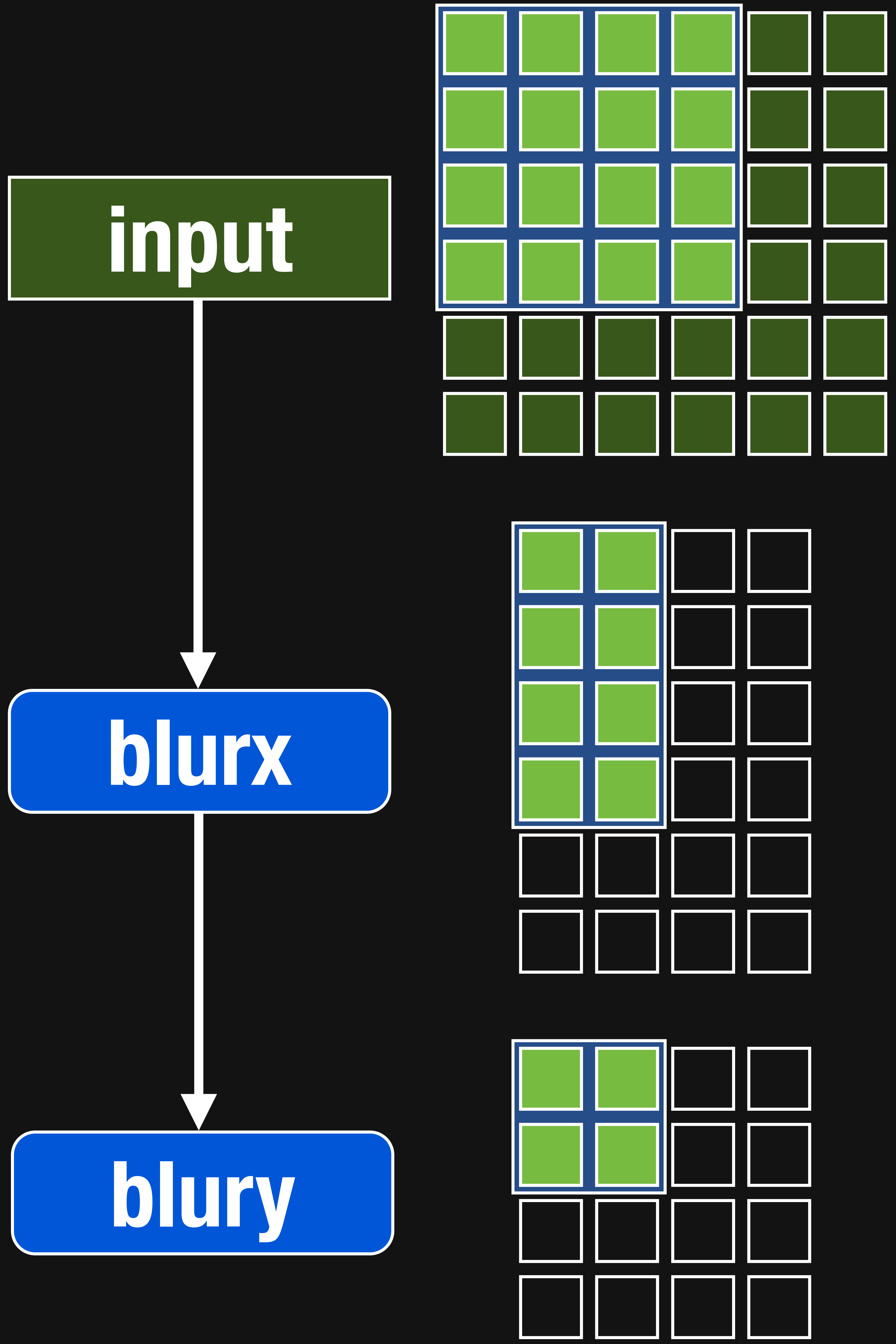
Executing the pipeline

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



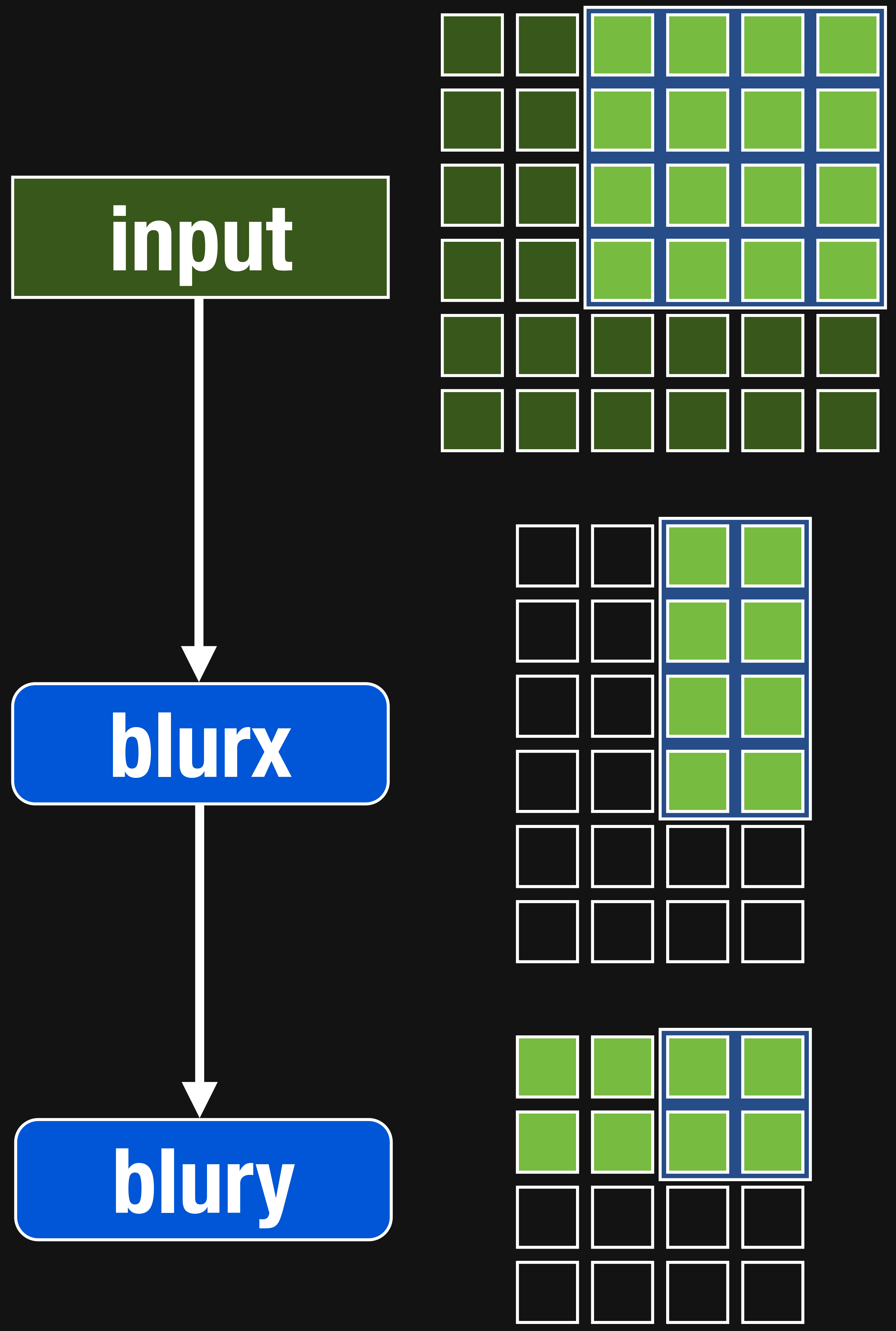
Fusing stages globally interleaves execution

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



Fusing stages globally interleaves execution

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

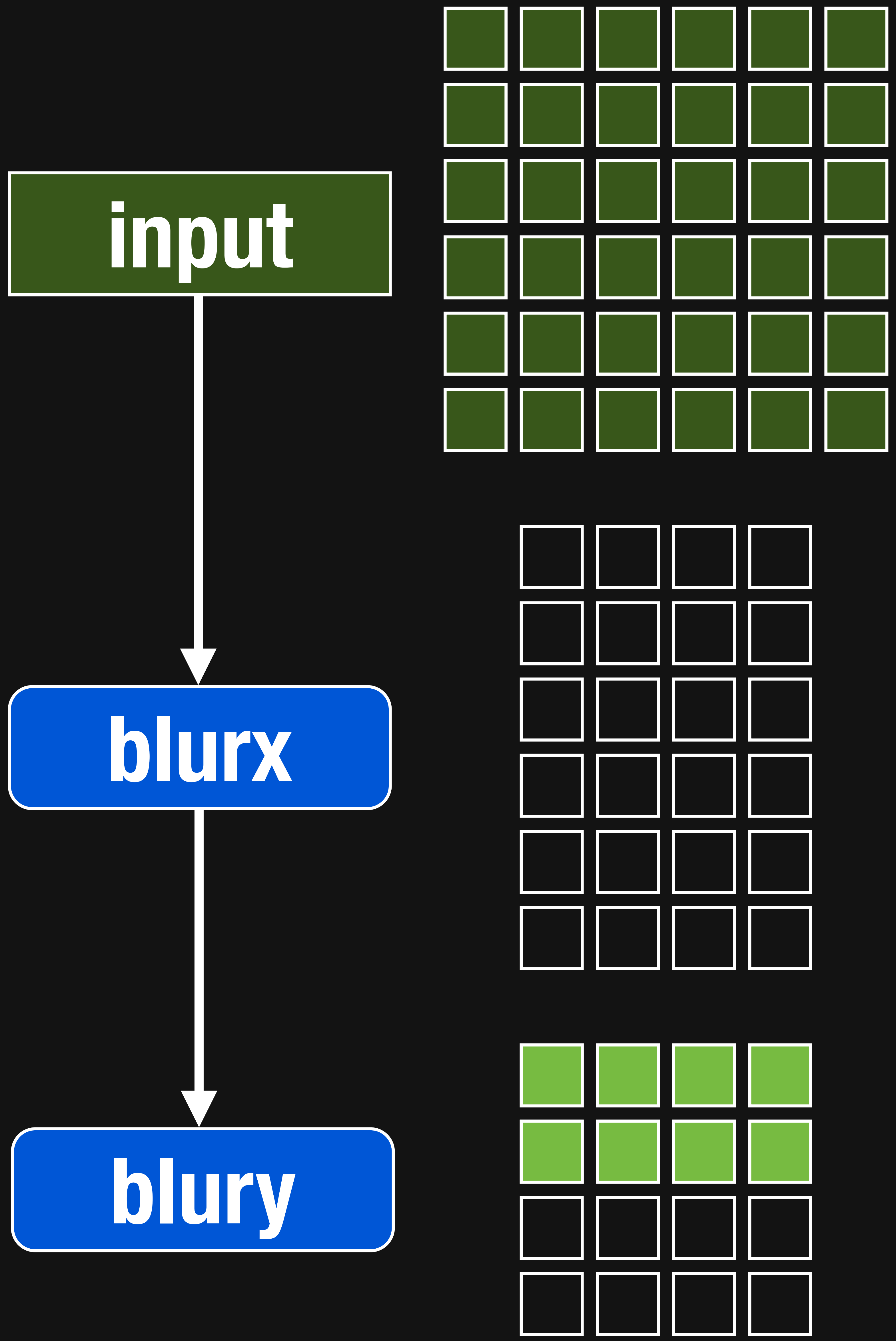

input

blurx

blury

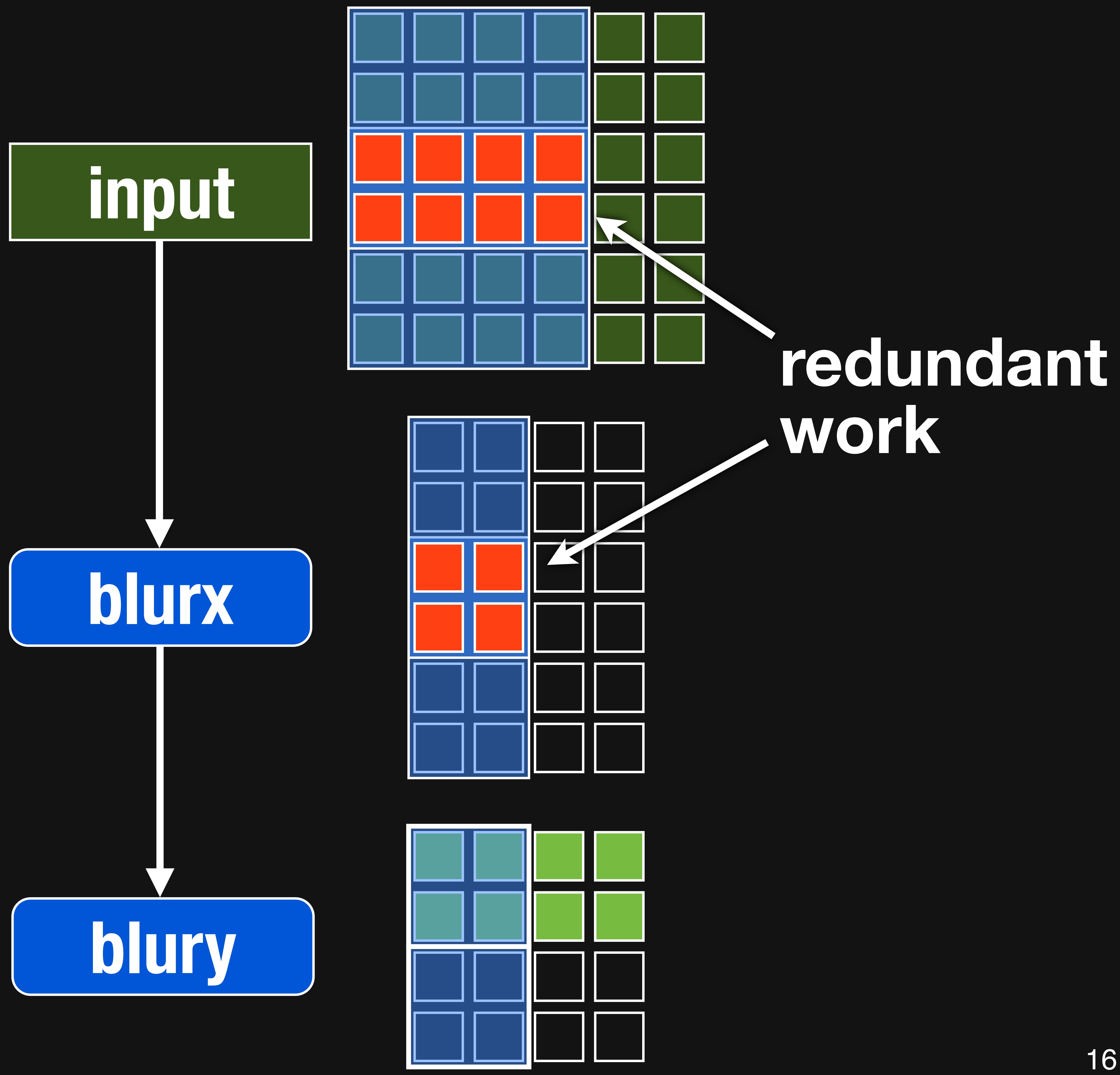
Fusion is a complex *tradeoff*

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



Fusion is a complex *tradeoff*

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



input

blurx

blury

redundant work

The choice space

For each stage:

Question 1) In what order should it compute its values?

In what order should I compute my values?

Serial y,
Serial x

In what order should I compute my values?

Serial x,

Serial y

In what order should I compute my values?

Serial y,
Vectorize x by 4

In what order should I compute my values?

Parallel y,
Vectorize x by 4

In what order should I compute my values?

Split x by 4**,**
Split y by 4**.**
Serial $\mathbf{y}_{\text{outer}}$,
Serial $\mathbf{x}_{\text{outer}}$,
Serial $\mathbf{y}_{\text{inner}}$,
Serial $\mathbf{x}_{\text{inner}}$

The choice space

For each stage:

Question 1) In what order should it compute its values?

Question 2) When should it compute its inputs?

When should I compute my inputs?

Poor locality

All at once, ahead of time

When should I compute my inputs?

Redundant recompute

As needed, discarding after use

When should I compute my inputs?

Poor parallelism

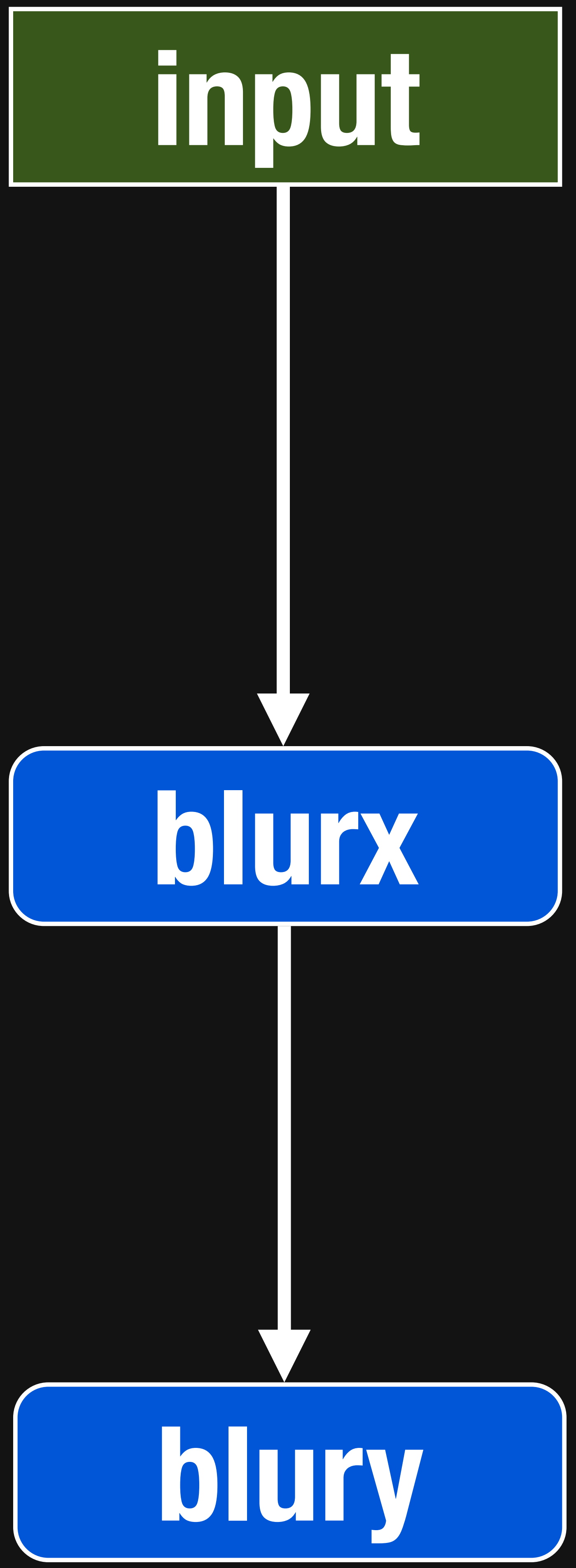
As needed, reusing old values

Some more points within the choice space

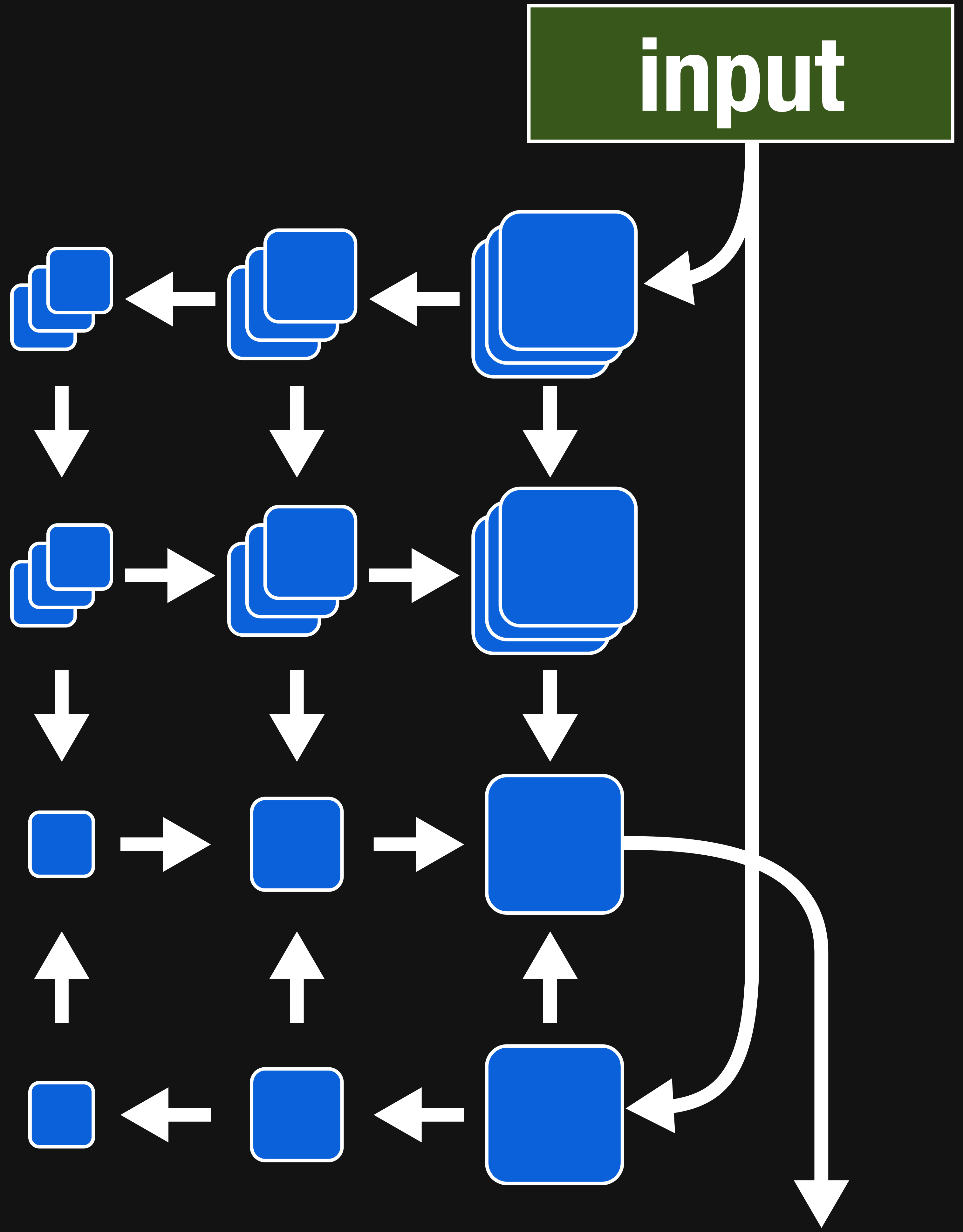
Some more points within the choice space

Some more points within the choice space

Scheduling is a complex *tradeoff*



3x3 box filter



local Laplacian filters

[Paris et al. 2010, Aubry et al. 2011]

Existing languages make optimizations hard

Parallelism	**Locality**
vectorization	***fusion***
multithreading	***tiling***

C - parallelism + tiling + fusion are hard to write *or* automate

CUDA, OpenCL, shaders - data parallelism is easy, fusion is hard

libraries don't help:
BLAS, IPP, MKL, OpenCV, MATLAB
optimized kernels compose into inefficient pipelines (no fusion)

Halide: *decouple* algorithm from schedule

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

Easy for programmers to build pipelines
simplifies algorithm code
improves modularity

Easy for programmers to specify & explore optimizations
fusion, tiling, parallelism, vectorization
can't break the algorithm

Easy for the compiler to generate fast code

The algorithm: pipelines as pure functions

Pipeline stages are functions from coordinates to values

no side effects

coordinates span an infinite domain

boundaries and required regions are inferred

Execution order and storage are unspecified

points can be evaluated (or reevaluated) in any order

results can be cached, duplicated, or recomputed anywhere

3x3 blur as a Halide *algorithm*:

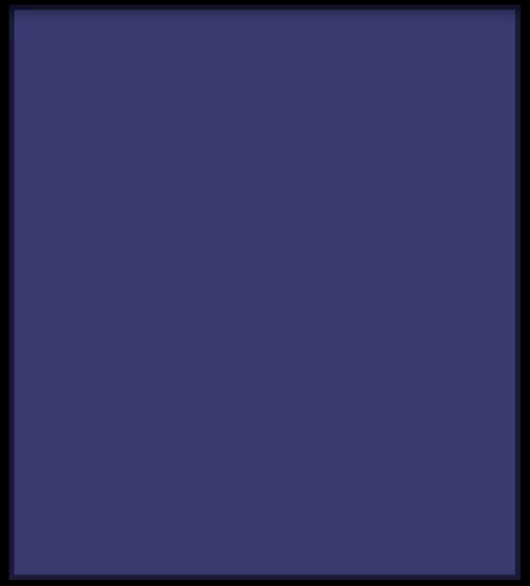
```
Func blurx, blury;
Var x, y;
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
```

The schedule: producer-consumer interleaving

For each stage:

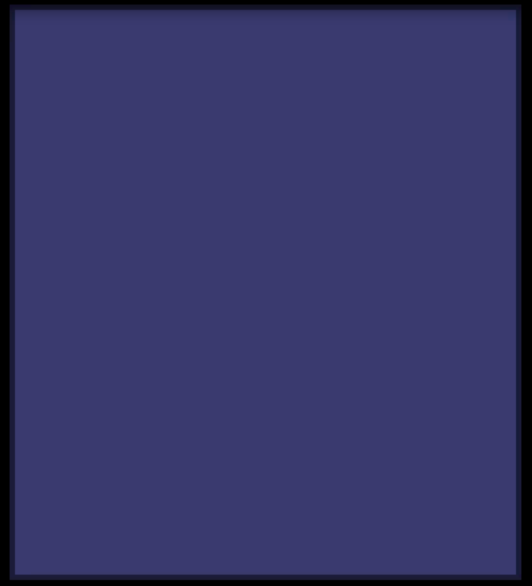
Question 1) In what order should it compute its output?

Question 2) When should it compute its inputs?



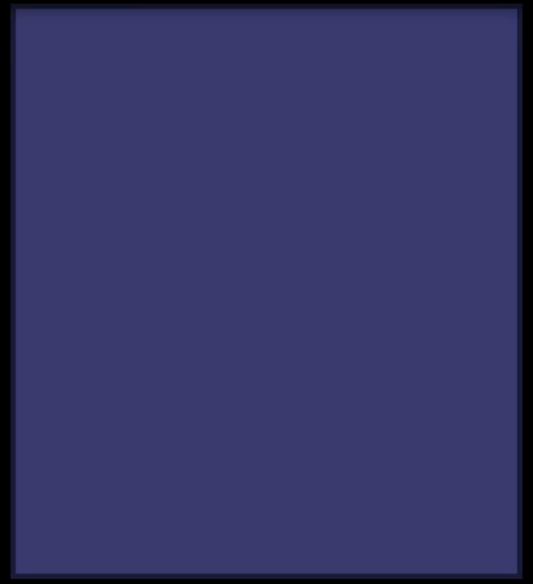
```
blur_x.compute_root();
```

```
blur_x.compute_at(blur_y, x);
```



```
blur_x.store_root().compute_at(blur_y, x);
```

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



```
blur_x.store_root()
      .compute_at(blur_y, y)
      .split(x, x, xi, 8)
      .vectorize(xi, 4).parallel(x);
blur_y.split(x, x, xi, 8)
      .vectorize(xi, 4).parallel(x);
```

```
blur_x.store_at(blur_y, y)
      .compute_at(blur_y, yi)
      .vectorize(x, 4);
blur_y.split(y, y, yi, 8)
      .vectorize(x, 4)
      .parallel(y);
```

Halide

0.9 ms/megapixel

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

  // The 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;

  // The schedule
  blury.tile(x, y, xi, yi, 256, 32)
       .vectorize(xi, 8).parallel(y);
  blurx.compute_at(blur_y, x).vectorize(x, 8);

  return blury;
}
```

Halide

0.9 ms/megapixel

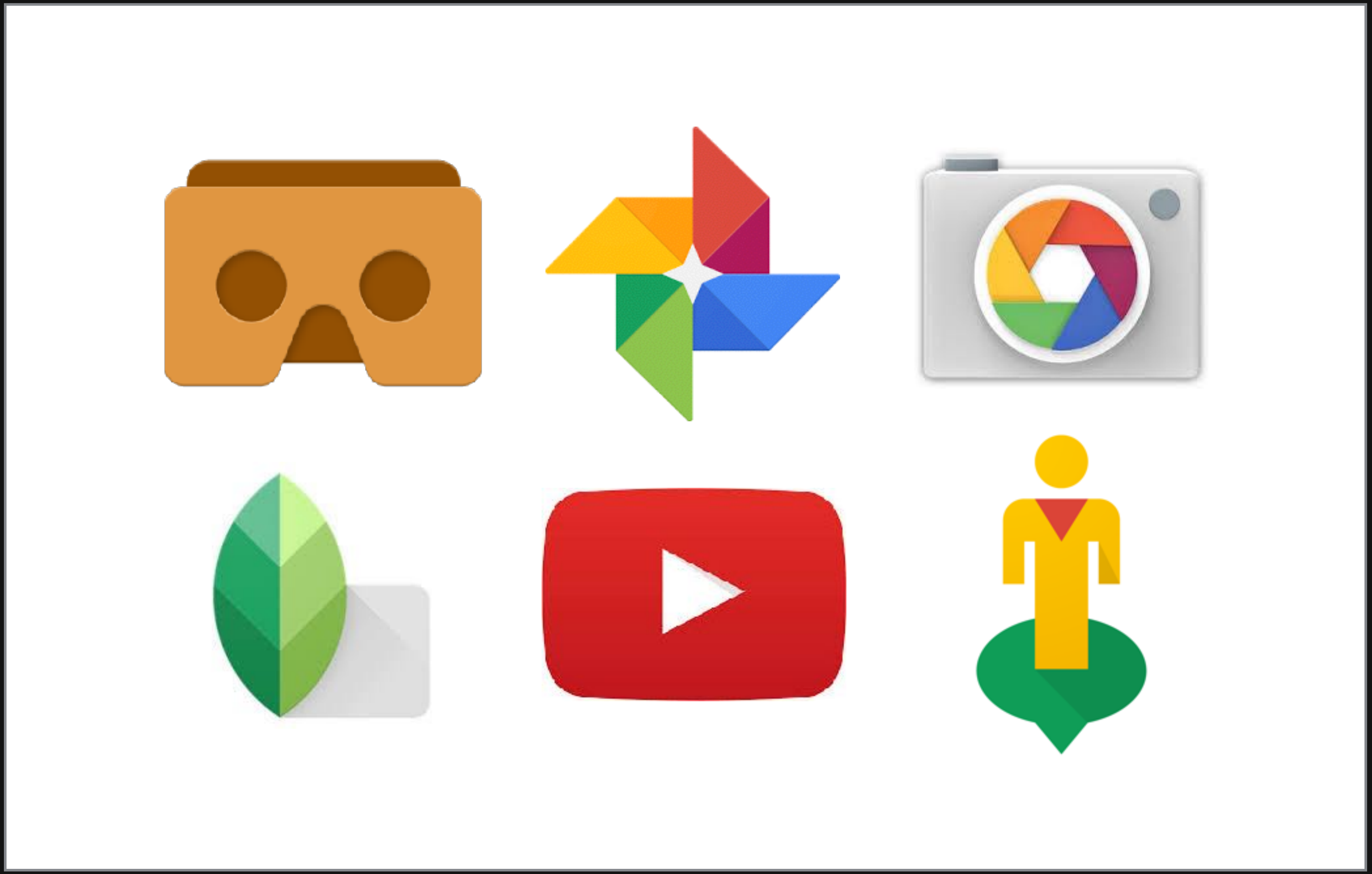
```
Func box_filter_3x3(Func in) {
  Func blurx, blury;
  Var x, y, xi, yi;

  // 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).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);
}}}}}
```

Halide at Google



Halide’s Development Philosophy

All Halide development and design is done in the open.

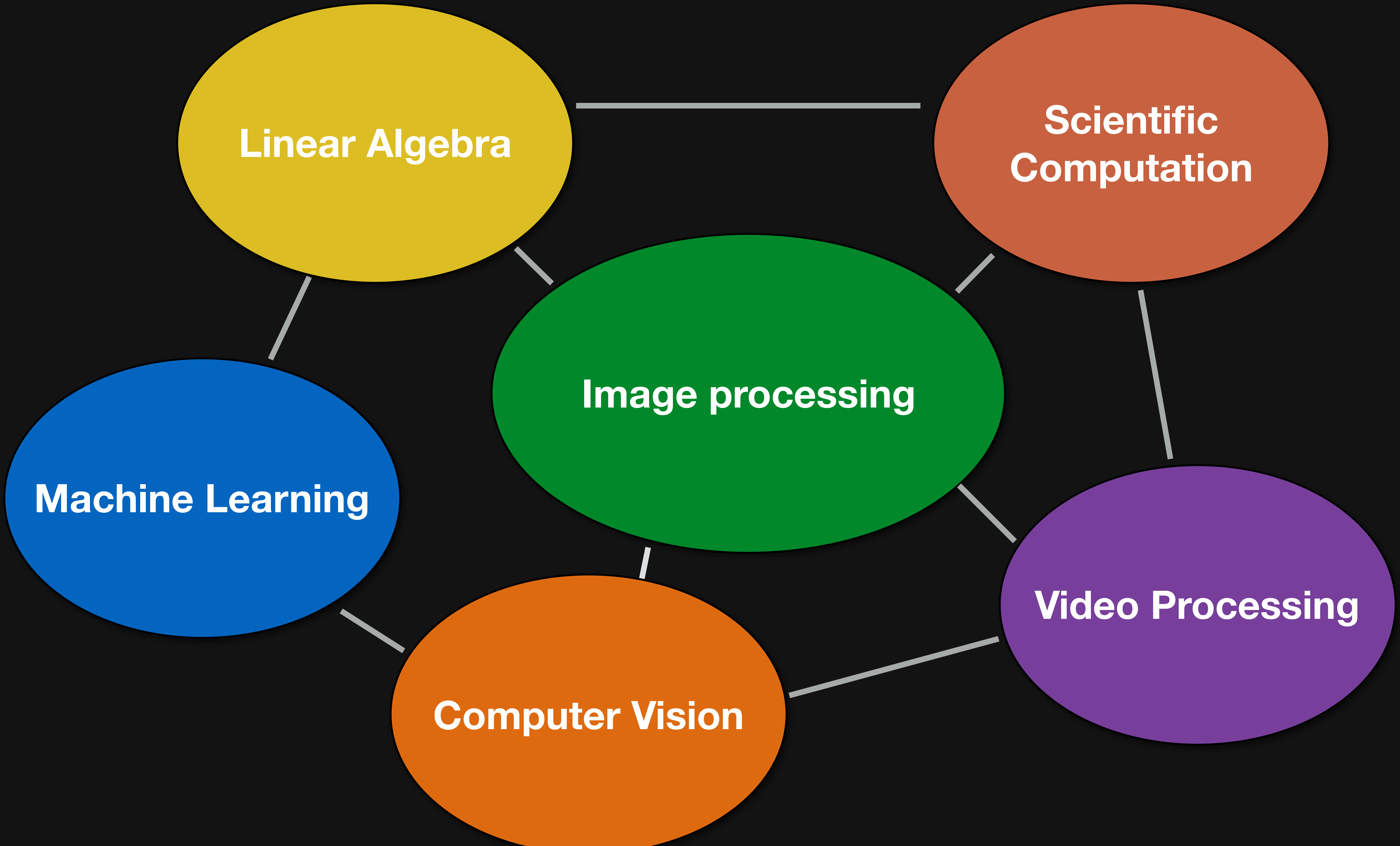
We also support the things we don’t care about at Google.

We grow by telling engineers about Halide, and helping them to evaluate if it’s a useful tool for them.

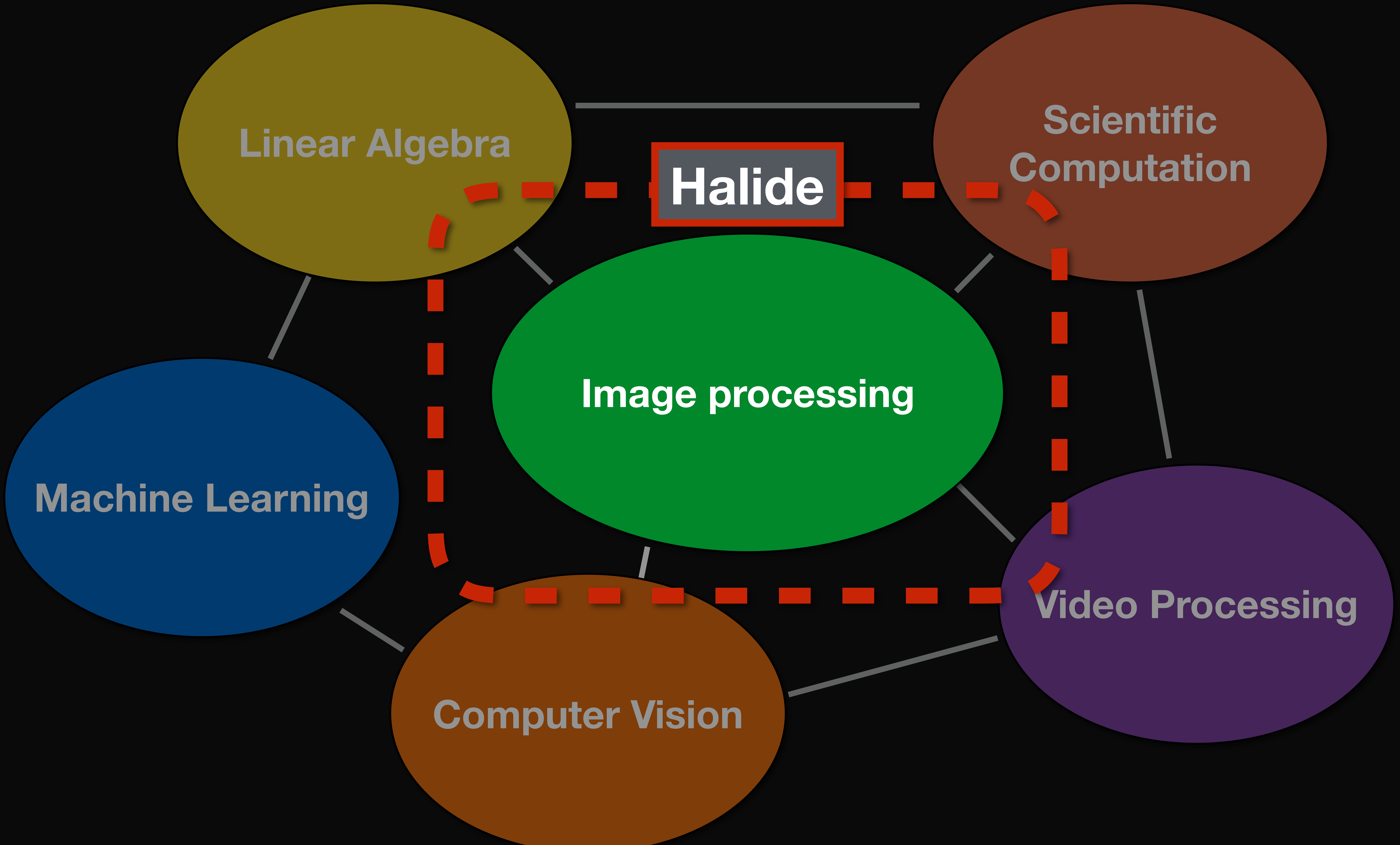
We are supportive of and interoperate with other languages and tools in the same domain.

Problems with Halide

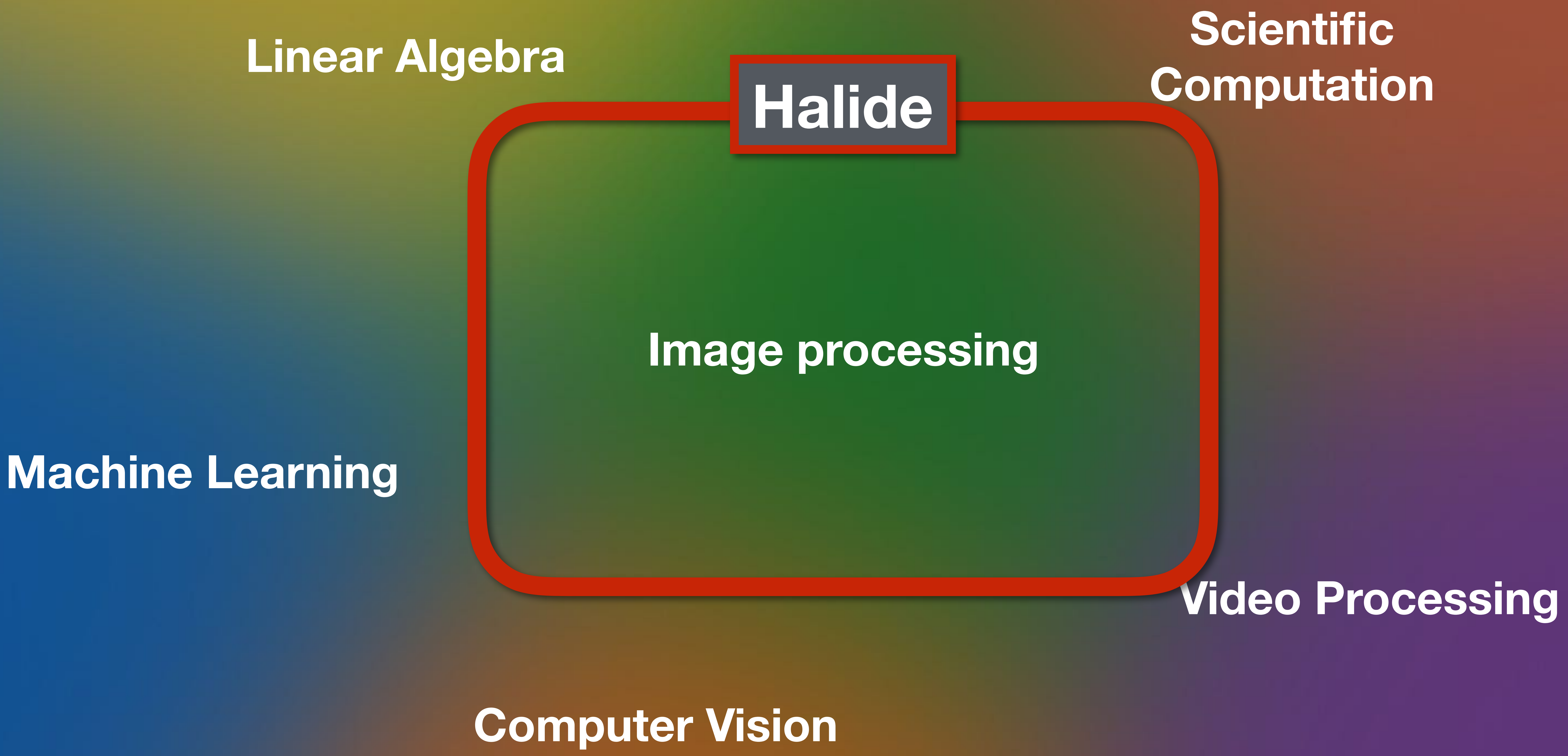
Domains are amorphous



Domains are amorphous



Domains are amorphous



Linear Algebra

Scientific
Computation

Halide

Image processing

Machine Learning

Video Processing

Computer Vision

Metaprogramming is weird

Halide is embedded in C++. This means:

You write a C++ program

which you must compile

when you run it, it builds a Halide pipeline in memory

then compiles it to a .o file

which you then link into another C++ program

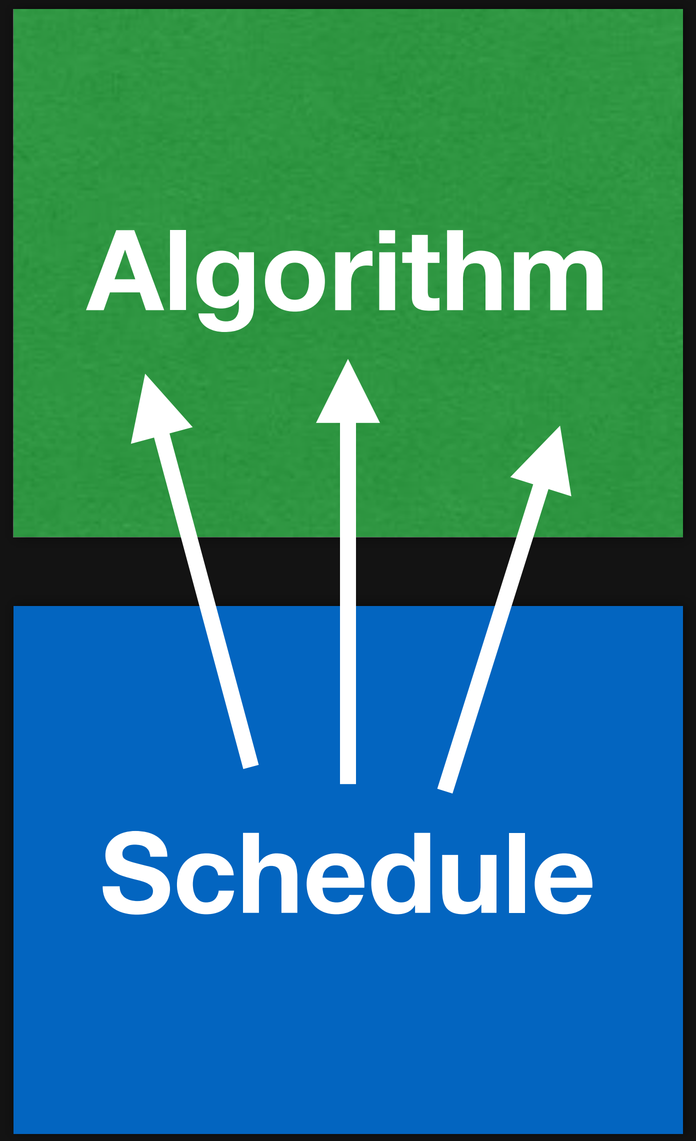
which you deploy

Aspect-oriented programming is weird

Algorithm

Schedule

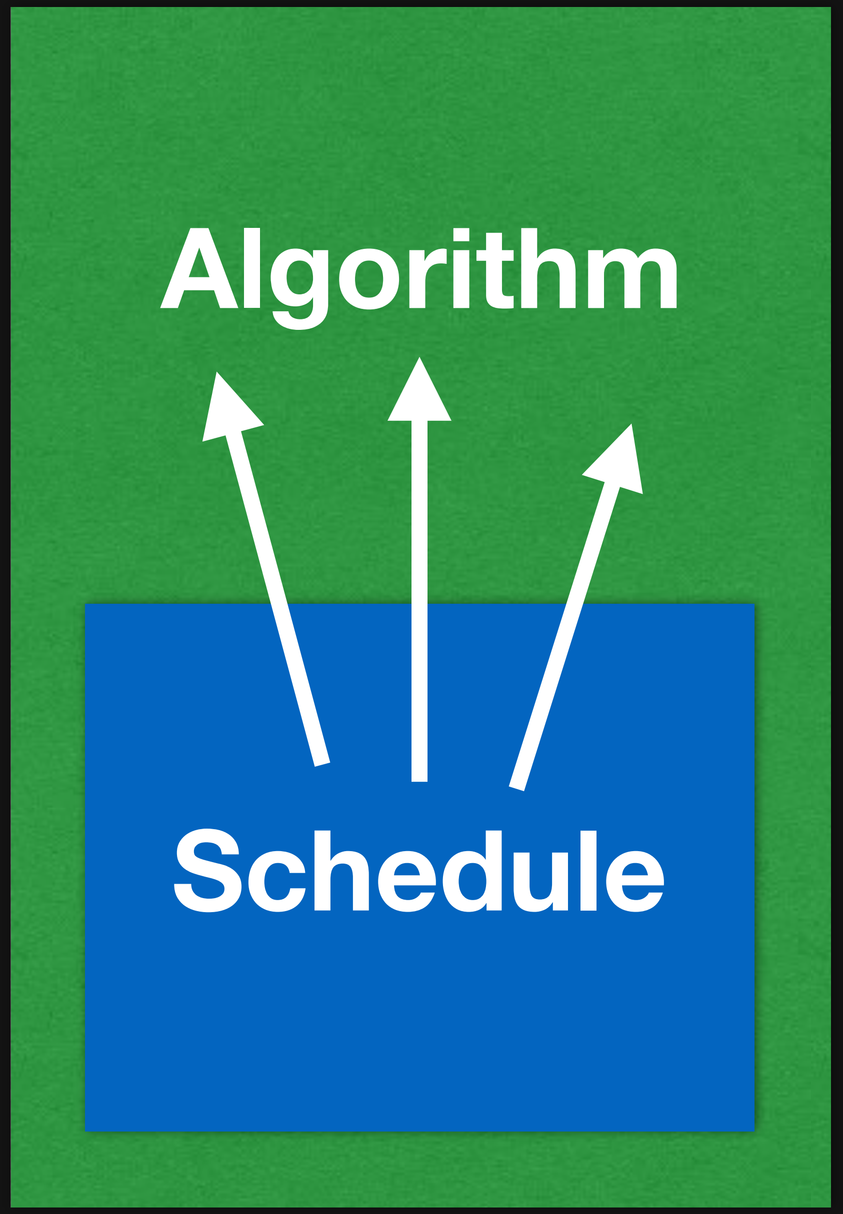
Aspect-oriented programming is weird



Algorithm

Schedule

Aspect-oriented programming is weird


Algorithm

Schedule

Aspect-oriented programming is weird

Algorithm

Schedule

Algorithm

Schedule

Algorithm

Schedule

Algorithm

Schedule

Algorithm

Schedule

Aspect-oriented programming is weird

Aspect-oriented programming is weird

Aspect-oriented programming is weird

Define algorithms

Algorithm

Schedule

Algorithm

Schedule

Algorithm

Schedule

Algorithm

Schedule

Algorithm

Schedule

Define schedules

Scheduling is hard

Easy

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

  return blury;
}
```

Hard

Scheduling is hard

All programming is done to an abstract machine.

To write code, you must be able to mentally emulate that abstract machine.

What does “x = 5;” mean in C?

A Halide algorithm is a program for an abstract machine that does simple arithmetic on scalar types.

Easy to mentally emulate.

A Halide schedule is a program for an abstract machine that builds and manipulates loop nests.

Hard to hold a loop nest in your head.

Scheduling is hard

Halide guarantees that changing the schedule will not change the result.

If Halide did not have correctness guarantees, you could express the desired loop nest more directly.

Instead, the schedule specifies a sequence of correctness-preserving mutations of a loop nest.

The space of *valid* schedules is a complex subspace of the space of all syntactically-correct schedules.

… and relies on over-conservative analysis!

Four design questions for DSL writers

Q) Why a language rather than a library?

A) Fusion across component boundaries

✓

Q) How are you going to handle applications that aren’t quite expressible?

A) Well-defined interface for external code to behave as a Halide pipeline stage.

✓

Q) How will people learn your language?

A) Basic Halide usage easy, but learning to exploit the full power of Halide scheduling is very difficult.

✗

Q) How will usability scale with problem size?

A) Complex monolithic pipelines work well, but it is hard to write reusable components.

✗

The hard open questions

How do we scale up aspect-oriented programming?

How do we explicitly specify low-level behavior in a safe language?

Conclusion

Public website at http://halide-lang.org

Tutorials at http://halide-lang.org/tutorials

We welcome contributions
http://github.com/Halide/halide

Fast image processing is hard because you need to optimize for locality *and* parallelism

Halide helps, by separating the algorithm from the optimizations (the *schedule*)

code becomes more modular, readable, and portable

makes it easier to explore different optimizations

Get the compiler at http://halide-lang.org