Halide: A DSL for Image Processing

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

January 2023 DPP Lecture Slides

A Data-Flow Representation for Stencil Pipelines

Scheduling Stencil Pipelines: Extreme Design Points

Optimized Schedules of Stencil Pipelines

Schedule Representation: Domain Order and Call Schedule

Halide Documentation

The material presented in this lecture was primarily gathered from the paper:

“Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”, Jonathan Ragan-Kelly, Connelly Barnes, Andrew Adams, In procs. of Int. Conf. Programming Language Design and Implementation (PLDI), 2013.

Halide github: https://github.com/halide/Halide

Motivation: Image Processing Applications

- Iterated stencil computations:
  - one or few (small) stencils applied to the same grid over many iterations;
  - well studied, good application for the polyhedral model.

- Image processing applications:
  - graphs of different stencil computations (stencil pipelines);
  - iteration of the same stencil: the exception, not the rule;
  - 1D stencils: handled by stream compilation, but not 2D or 3D;
  - combined with gather/scatter operations, e.g., histograms.

- Main Optimization is stencil fusion; techniques span a complex tradeoff space:
  - producer-consumer locality;
  - parallelism (synchronization);
  - redundant computation.

What Are Stencils?

Each point in a (result) multi-dimensional array is computed by a weighted average of a neighborhood of points of the input array.

Stencil codes defined as a 4-tuple (I, A, s, T):

- Index set $I = \prod_{i=1}^{k} [0, \ldots, n_i]$ defining the array topology;
- $A \colon \mathbb{Z}^k \rightarrow \alpha$ the input array;
- $s \in \prod_{i=1}^{l} \mathbb{Z}^k$ is the stencil; describes the shape of the neighborhood. There are l elements in the stencil.
- $T \colon \alpha^l \rightarrow \alpha$ is the transition function used to determine a cell's new state, depending on its neighbors.

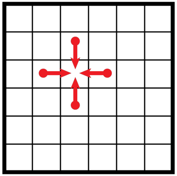
What Are Stencils?

Each point in a (result) multi-dimensional array is computed by a weighted average of a neighborhood of points of the input array.

Stencil codes defined as a 4-tuple (I, A, s, T):

- Index set $I = \prod_{i=1}^{k} [0, \ldots, n_i]$ defining the array topology;
- $A \colon \mathbb{Z}^k \to \alpha$ the input array;
- $s \in \prod_{i=1}^{l} \mathbb{Z}^k$ is the stencil; describes the shape of the neighborhood. There are l elements in the stencil.
- $T \colon \alpha^l \to \alpha$ is the transition function used to determine a cell's new state, depending on its neighbors.

```
float A[M][N], B[M][N];
for(int i=0; i<M; i++) {
    for(int j=0; j<N; j++) {
      if(i>0 && i<M-1 && j>0 && j<N-1) {
          B[i,j] = ( A[i+1,j] + A[i-1,j] +
                     A[i,j+1] + A[i,j-1] ) / 4;
      } else {
          // ... treat boundary conditions
      }
}  }
```



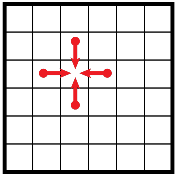
What Are Stencils?

Each point in a (result) multi-dimensional array is computed by a weighted average of a neighborhood of points of the input array.

Stencil codes defined as a 4-tuple (I, A, s, T):

- Index set $I = \prod_{i=1}^{k} [0, \ldots, n_i]$ defining the array topology;
- $A \colon \mathbb{Z}^k \to \alpha$ the input array;
- $s \in \prod_{i=1}^{l} \mathbb{Z}^k$ is the stencil; describes the shape of the neighborhood. There are l elements in the stencil.
- $T \colon \alpha^l \to \alpha$ is the transition function used to determine a cell's new state, depending on its neighbors.

```
float A[M][N], B[M][N];
for(int i=0; i<M; i++) {
    for(int j=0; j<N; j++) {
      if(i>0 && i<M-1 && j>0 && j<N-1) {
          B[i,j] = ( A[i+1,j] + A[i-1,j] +
                     A[i,j+1] + A[i,j-1] ) / 4;
      } else {
          // ... treat boundary conditions
      }
}  }
```



For example $I = [0, \ldots, M-1] \times [0, \ldots, N-1], \alpha = \mathbb{R}$,
$s = ((-1, 0), (+1, 0), (0, +1), (0, -1)), T = (x1 + x2 + x3 + x4)/4$.

Simplest boundary condition: $(i, j) + s.k \; mod \; (M, N)$

Functional Specification Enable Aggressive Optimizations

Separate the algorithm from the optimization recipe, which can be either specified by the user or searched automatically:

- Data flow specification of the algorithm
  - does not unnecessarily constrains the implementation;
- A scheduling representation that generates optimization recipes along three axes:
  - locality;
  - parallelism;
  - redundant computation;
- Loop Synthesizer and Code Generator for a range of modern hardware;
- Autotuner that can find high-performance schedules for complex pipelines using stochastic search.

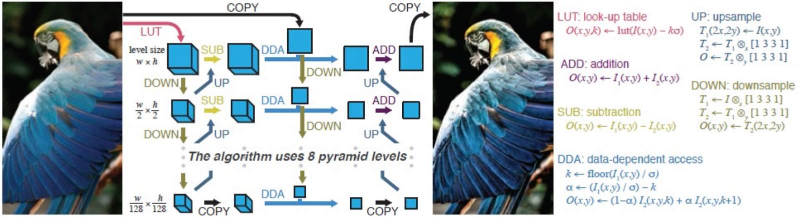
Halide: Data-Flow Functional Representation

- C++ implem of Laplacian filters uses hundreds of code lines;
- hard to statically optimize by a general-purpose compiler.
- Halide version uses 62 lines describing the essential dataflow and computation of the 99-stage pipeline;
- Optimizations are described separately or auto searched.
- mutable arrays $\Rightarrow$ functions from coordinates to values;
- pipelines $\Rightarrow$ chains of functions;
- functions: either simple (pure) expressions or reductions;

Halide: Data-Flow Functional Representation

- C++ implem of Laplacian filters uses hundreds of code lines;
- hard to statically optimize by a general-purpose compiler.
- Halide version uses 62 lines describing the essential dataflow and computation of the 99-stage pipeline;
- Optimizations are described separately or auto searched.
- mutable arrays ⇒ functions from coordinates to values;
- pipelines ⇒ chains of functions;
- functions: either simple (pure) expressions or reductions;

Structure of Local-Laplacian filter. Each box represents intermediate data and each arrow represents functions that define that data. The pipeline includes horizontal and vertical stencils, resampling, data-dependent gathers (e.g., histograms) and map-like functions.



COPY
COPY
LUT
level size $w \times h$
SUB
DDA
ADD
DOWN
UP
DOWN
UP
$\frac{w}{2} \times \frac{h}{2}$
SUB
DDA
ADD
DOWN
UP
DOWN
UP
The algorithm uses 8 pyramid levels
$\frac{w}{128} \times \frac{h}{128}$
COPY
DDA
COPY
LUT: look-up table
$O(x,y,k) \leftarrow \mathrm{lut}(I(x,y) - k\sigma)$
ADD: addition
$O(x,y) \leftarrow I_1(x,y) + I_2(x,y)$
SUB: subtraction
$O(x,y) \leftarrow I_1(x,y) - I_2(x,y)$
UP: upsample
$T_1(2x,2y) \leftarrow I(x,y)$
$T_2 \leftarrow T_1 \otimes_x [1\ 3\ 3\ 1]$
$O \leftarrow T_2 \otimes_y [1\ 3\ 3\ 1]$
DOWN: downsample
$T_1 \leftarrow I \otimes_x [1\ 3\ 3\ 1]$
$T_2 \leftarrow T_1 \otimes_y [1\ 3\ 3\ 1]$
$O(x,y) \leftarrow T_2(2x,2y)$
DDA: data-dependent access
$k \leftarrow \mathrm{floor}(I_1(x,y) / \sigma)$
$\alpha \leftarrow (I_1(x,y) / \sigma) - k$
$O(x,y) \leftarrow (1-\alpha)\, I_2(x,y,k) + \alpha\, I_2(x,y,k+1)$

Halide: Data-Flow Functional Representation

- data-parallel represent. within the domain of each function;
- functions defined over infinite domain ⇒ boundary conds handled (auto) by computing guard bands of extra values;
- if boundary conditions matter, functions can define their own;

A separable 3×3 un-normalized box filter is expressed as a chain of two functions `blurx` and `out` in variables `x` and `y`.

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

- `blurx`: horizontal blur by averaging over a 3×1 window.
- `out`: isotropic blur by averaging over a 1×3 window.

Halide: Iterative Computations

Iterative/recursive computations like summation, histogram, scan. “Reductions” are defined in two steps:

- Initial value function: specifies a value at each point in the output domain;
- Recursive reduction function: redefines the values at points in terms of prior values of the function;
- Reduction Domain: bounded by the min/max exps for each dim; specifies the order in which the reduction fun is applied.

Halide: Iterative Computations

Iterative/recursive computations like summation, histogram, scan. “Reductions” are defined in two steps:

- Initial value function: specifies a value at each point in the output domain;
- Recursive reduction function: redefines the values at points in terms of prior values of the function;
- Reduction Domain: bounded by the min/max exps for each dim; specifies the order in which the reduction fun is applied.

```
UniformImage in(Uint(8), 2)
RDom r(0..in.width(), 0..in.height()), ri(0..255)
Var x, y, i
Func histogram(i) = 0; histogram(in(x,y))++
Func cfd(i) = 0; cfd(ri) = cfd(ri-1) + histogram(ri)
Func out(x,y) = cfd(in(x,y))
```

Iteration bounds for histogram and scan are explicitly expressed by means of the reduction domain (RDom).

A Data-Flow Representation for Stencil Pipelines

Scheduling Stencil Pipelines: Extreme Design Points

Optimized Schedules of Stencil Pipelines

Schedule Representation: Domain Order and Call Schedule

Breadth First (Data Parallel) Scheduling

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

Think about scheduling from the perspective of the output stage!

Extreme Case 1: each stage executes breadth-first across its input before passing its entire output to the next stage.
Applied to a $2K \times 3K$ image:

Breadth First (Data Parallel) Scheduling

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

Think about scheduling from the perspective of the output stage!

Extreme Case 1: each stage executes breadth-first across its input before passing its entire output to the next stage.

Applied to a $2K \times 3K$ image:

```
alloc blurx[2048][3072]
for each y in 0..2048:
  for each x in 0..3072:
    blurx[y][x] = in[y][x-1] + in[y][x] + in[y][x+1]

alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    out[y][x] = blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

Breadth First (Data Parallel) Scheduling

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

Think about scheduling from the perspective of the output stage!

Extreme Case 1: each stage executes breadth-first across its input before passing its entire output to the next stage.

Applied to a $2K \times 3K$ image:

```
alloc blurx[2048][3072]
for each y in 0..2048:
  for each x in 0..3072:
    blurx[y][x] = in[y][x-1] + in[y][x] + in[y][x+1]

alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    out[y][x] = blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

+ high parallelism degree,
- low producer-consumer locality.

Total-Fusion / Fully-Fused Scheduling

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

Think about scheduling from the perspective of the output stage!

Extreme Case 2: compute each point in `blurx` immediately before the point using it, without storing the intermediate result across uses. Applied to a $2K \times 3K$ image:

Total-Fusion / Fully-Fused Scheduling

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

Think about scheduling from the perspective of the output stage!

Extreme Case 2: compute each point in `blurx` immediately before the point using it, without storing the intermediate result across uses. Applied to a $2K \times 3K$ image:

```
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    alloc blurx[-1..1]
    for each i in -1..1:
      blurx[i] = in[y+i][x-1] + in[y+i][x] + in[y+i][x+1]
    out[y][x] = blurx[-1] + blurx[0] + blurx[1]
```

Total-Fusion / Fully-Fused Scheduling

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

Think about scheduling from the perspective of the output stage!

Extreme Case 2: compute each point in `blurx` immediately before the point using it, without storing the intermediate result across uses. Applied to a $2K \times 3K$ image:

```
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    alloc blurx[-1..1]
    for each i in -1..1:
      blurx[i] = in[y+i][x-1] + in[y+i][x] + in[y+i][x+1]
    out[y][x] = blurx[-1] + blurx[0] + blurx[1]
```

+ high parallelism degree;
+ maximizes locality (small producer-consumer reuse distance);
- a lot of redundant work (`blurx` not reused across iterations) 10 accesses to global memory rather than 8 previously.

Sliding-Window Scheduling

Think about scheduling from the perspective of the output stage!

Extreme Case 3: compute each point in `blurx` immediately before the point using it, but points in `blurx` are stored and reused. Applied to a $2K \times 3K$ image:

```
alloc out[2046][3072]
alloc blurx[3][3071]
for y in -1..2047:
  for each x in 0..3072:
    blurx[(y+1)%3][x] = in[y+1][x-1] + in[y+1][x] + in[y+1][x+1]
    if y < 1: continue
    out[y][x] = blurx[(y-1)%3][x] + blurx[y%3][x] + blurx[(y+1)%3][x]
```

Sliding-Window Scheduling

Think about scheduling from the perspective of the output stage!

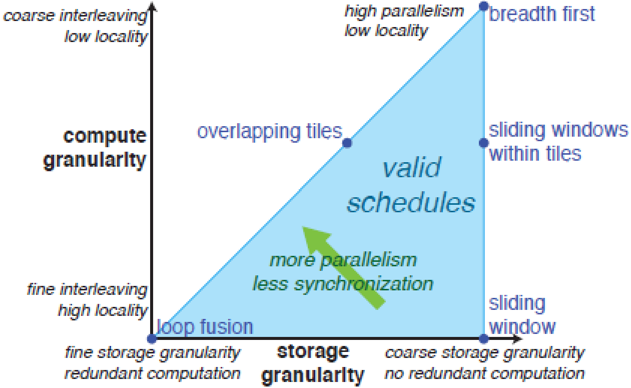
Extreme Case 3: compute each point in `blurx` immediately before the point using it, but points in `blurx` are stored and reused. Applied to a $2K \times 3K$ image:

```
alloc out[2046][3072]
alloc blurx[3][3071]
for y in -1..2047:
  for each x in 0..3072:
    blurx[(y+1)%3][x] = in[y+1][x-1] + in[y+1][x] + in[y+1][x+1]
    if y < 1: continue
    out[y][x] = blurx[(y-1)%3][x] + blurx[y%3][x] + blurx[(y+1)%3][x]
```

Code interleaves the computation in a sliding window with `out` trailing `blurx` by the stencil radius (one scanline);

- + loop `for y` is sequential: a given iteration of `out` depends on the last three outer loop iterations of `blurx`;
- + okish locality: maximum distance between a value produced in `blurx` and consumed in `out` proportional with the stencil height (three scanlines);
- + no redundant work (each point in `blurx` computed once)

Tradeoff Space



Visualization in terms of granularity of storage and of computation:

- Total fusion requires redundant work:
  fine-grained computation in fine-grained storage (small temp buffers);
- Sliding window constrains parallelism: allocates enough storage for the entire intermediate stage but computes it in fine-grained chunks as late as possible;
- Breadth first has poor locality: coarse-storage and coarse interleaving;
- Best strategies tend to be mixed and lie in the middle of the space!

Tiled-Fusion Scheduling

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

Interleave the computation of `blurx` and `out` at the level of tiles!

```
alloc out[2046][3072]
for each ty in 0..2048 by 32:
  for each tx in 0..3072 by 32:
    // what slice of blurx is used below?
    // compute it here instead!
    for each y in 0..32:
      for each x in 0..32:
        out[ty+y][tx+x] = blurx[ty+y-1][tx+x] +
                          blurx[ty+y  ][tx+x] +
                          blurx[ty+y+1][tx+x]
```

Tiled-Fusion Scheduling

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

Interleave the computation of `blurx` and `out` at the level of tiles!

```
alloc out[2046][3072]
for each ty in 0..2048 by 32:
  for each tx in 0..3072 by 32:
    // what slice of blurx is used below?
    // compute it here instead!
    for each y in 0..32:
      for each x in 0..32:
        out[ty+y][tx+x] = blurx[ty+y-1][tx+x] +
                          blurx[ty+y  ][tx+x] +
                          blurx[ty+y+1][tx+x]
```

Needed Slice Is: `blurx[ty-1 :  ty+32+1][tx :  tx+32]`

Insert the code that computes the slice; it fits in a 2D array of size 34 × 32!

Tiled-Fusion Scheduling

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

Interleave the computation of `blurx` and `out` at the level of tiles!

```
alloc out[2046][3072]
for each ty in 0..2048 by 32:
  for each tx in 0..3072 by 32:
    alloc blurx[-1..33][32]
    for each y in -1..33:
      for each x in 0..32:
        blurx[y][x]= in[ty+y][tx+x-1]+in[ty+y][tx+x]+in[ty+y][tx+x+1]
    for each y in 0..32:
      for each x in 0..32:
        out[ty+y][tx+x] = blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

“On modern X86 this is 10×??? faster than breadth-first using the same amount of multithreading and vector parallelism.”

+ high parallelism degree: both within and across tiles;

+ good locality (producer-consumer reuse distance is a tile);

- small amount of redundant work on tile boundaries (ghost zones).

Sliding Window Within Tiles Scheduling

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1)+ blurx(x,y)+ blurx(x,y+1)
```

Interleave the computation of the 2 stages using a sliding window, but tile the scanlines and keep dependencies within the tile!

```
alloc out[2046][3072]
for each ty in 0..2048 by 8:
  alloc blurx[-1..1][3072]
  for y in -2..8:
    for each x in 0..3072:
      blurx[(y+1)%3][x]= in[ty+y+1][x-1]+in[ty+y+1][x]+in[ty+y+1][x+1]
    if y < 0: continue
    for each x in 0..3072:
      out[ty+y][x]= blurx[(y-1)%3][x]+blurx[y%3][x]+blurx[(y+1)%3][x]
```

“On modern X86 this is –10% to 10% faster than tiled fusion”

+ sequentializes only `for y in -2..8`;
+ good locality (prod-consumer reuse distance: 3 scanlines);
- sacrifices 2 scanlines of redundant work for every 8 scanlines.

Comparison between Optimization Recipes

Strategy	Parallelism	Max Reuse Dist.	Work Ampl.
Breadth-First	3072×2046	$3072 \times 2048 \times 3$	1.0×
Full Fusion	3072×2046	3×3	2.0×
Sliding Window	3072	$3072 \times (3 + 3)$	1.0×
Tiled	3072×2046	$34 \times 32 \times 3$	1.0625×
Sliding Tiles	$3072 \times 2048/8$	$3072 \times (3 + 3)$	1.25×

A Model for the Scheduling Choice Space

The space of choices in scheduling stencil pipelines can be summarized by answering the following questions for each stage in the pipeline:

- At what granularity to *compute* each of the inputs for each stage?
- At what granularity to *store* each input for reuse?
- Within those grains, in *what order* its domain should be traversed?

The Domain Order

Defines the order in which the required region of each function's domain is traversed:

- Each dimension can be traversed *sequentially* or *in parallel*;
- Constant-size dimensions can be *unrolled* or *vectorized*;
- Dimensions can be *reordered*, e.g., loop interchange;
- Dimensions can be *split* by a factor , i.e., loop stripmining.

Splitting recursively enables multiple optimization choices:

- vectorization: split by the vector width factor + schedule the new dimension as vectorized;
- unrolling: split by an unrolling factor + schedule the new dimension as unrolled;
- tiling: splitting dimensions + reordering them.

Code synthesized by simpler interval analysis rather than more complex polyhedral analysis, introduced in another lecture.

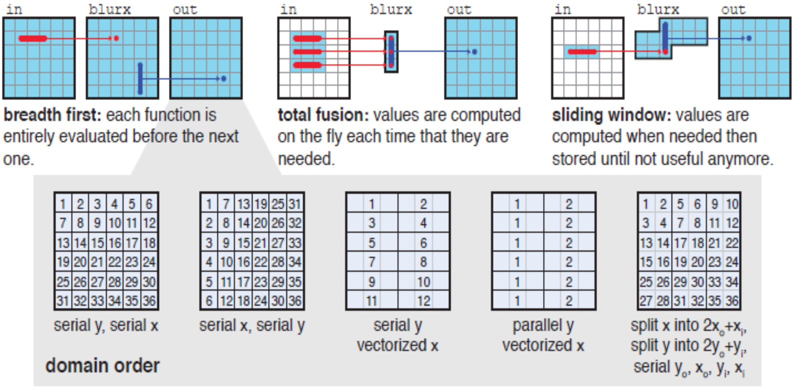
The Call Schedule

Each function's call schedule is defined as the points in the loop nest of its callers where it is *stored* and *computed*, for example

- breadth first: schedules both *stores* and *computes* `blurx` at the *coarsest granularity* (root level, outside any other loops);
- fully fused: schedules both *stores* and *computes* `blurx` at the *finest granularity (inside the innermost* `x` *loop of* `out`*);*
  - *values are produced and consumed in the same iteration, but must be reallocated/recomputed at each iteration.*
- *sliding window: schedules* stores at the root granularity*, while* computing at the finest granularity.
  - `blurx` *values are computed in the same iteration as their first use, but persist across iterations.*
  - *requires the (outer) loops between the storage and computation levels to be sequential!*

The Call Graph and The Domain Order define an algebra for scheduling stencil pipelines on rectangular grids.

Graphical Rep of Domain Order and Call Schedule



breadth first: each function is entirely evaluated before the next one.

total fusion: values are computed on the fly each time that they are needed.

sliding window: values are computed when needed then stored until not useful anymore.

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

serial y, serial x

1	7	13	19	25	31
2	8	14	20	26	32
3	9	15	21	27	33
4	10	16	22	28	34
5	11	17	23	29	35
6	12	18	24	30	36

serial x, serial y

1	2
3	4
5	6
7	8
9	10
11	12

serial y
vectorized x

1	2
1	2
1	2
1	2
1	2
1	2

parallel y
vectorized x

1	2	5	6	9	10
3	4	7	8	11	12
13	14	17	18	21	22
15	16	19	20	23	24
25	26	29	30	33	34
27	28	31	32	35	36

split x into $2x_o+x_i$,
split y into $2y_o+y_i$,
serial y_o, x_o, y_i, x_i

domain order

Schedule Example: Tiled Fused

```
alloc out[2046][3072]
for each ty in 0..2048 by 32:
  for each tx in 0..3072 by 32:
    alloc blurx[-1..33][32]
    for each y in -1..33:
      for each x in 0..32:
        blurx[y][x]= in[ty+y][tx+x-1]+in[ty+y][tx+x]+in[ty+y][tx+x+1]
    for each y in 0..32:
      for each x in 0..32:
        out[ty+y][tx+x] = blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

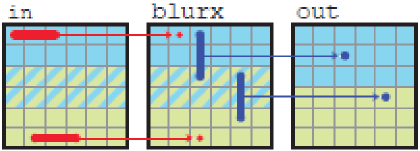
- domain order of out is *split*(`y,32`)→ `ty,y;` *split*(`x,32`)→ `tx,x;` *order*(`ty,tx,y,x`);
- all dimensions are *parallel*;
- call schedule of `blurx` is set to both store and compute for each iteration `tx` of out
- the domain of `blurx` under `tx` is scheduled *order*(`y,x`) and x is vectorized for performance.

Schedule Example: Tiled Sliding Window

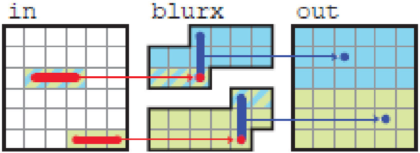
```
alloc out[2046][3072]
for each ty in 0..2048 by 8:
  alloc blurx[-1..1][3072]
  for y in -2..8:
    for each x in 0..3072:
      blurx[(y+1)%3][x]= in[ty+y+1][x-1]+in[ty+y+1][x]+in[ty+y+1][x+1]
    if y < 0: continue
    for each x in 0..3072:
      out[ty+y][x]= blurx[(y-1)%3][x]+blurx[y%3][x]+blurx[(y+1)%3][x]
```

- domain order of out is *split*`(y,8)`$\rightarrow$ `ty,y;` *order*`(ty,y,x)`;
- `ty` may be parallel, `y` must be sequential (to enable reuse), `x` is vectorized for performance;
- the domain of `blurx` under `y` only has dimension `x`, which is vectorized for performance.

Graphical Rep of Optimized Call Schedules

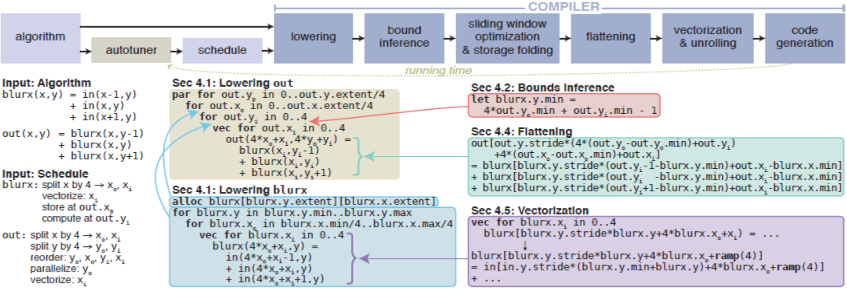


tiles: overlapping regions are processed in parallel, functions are evaluated one after another.



sliding windows within tiles: tiles are evaluated in parallel using sliding windows.

Compiler Pipeline



Input: Algorithm

```
blurx(x,y) = in(x-1,y)
           + in(x,y)
           + in(x+1,y)

out(x,y) = blurx(x,y-1)
         + blurx(x,y)
         + blurx(x,y+1)
```

Input: Schedule

```
blurx: split x by 4 → x_o, x_i
       vectorize: x_i
       store at out.x_o
       compute at out.y_i

out: split x by 4 → x_o, x_i
     split y by 4 → y_o, y_i
     reorder: y_o, x_o, y_i, x_i
     parallelize: y_o
     vectorize: x_i
```

Sec 4.1: Lowering out

```
par for out.y_o in 0..out.y.extent/4
  for out.x_o in 0..out.x.extent/4
    for out.y_i in 0..4
      vec for out.x_i in 0..4
        out(4*x_o+x_i,4*y_o+y_i) =
          blurx(x_i,y_i-1)
          + blurx(x_i,y_i)
          + blurx(x_i,y_i+1)
```

Sec 4.1: Lowering blurx

```
alloc blurx[blurx.y.extent][blurx.x.extent]
for blurx.y in blurx.y.min..blurx.y.max
  for blurx.x_o in blurx.x.min/4..blurx.x.max/4
    vec for blurx.x_i in 0..4
      blurx(4*x_o+x_i,y) =
        in(4*x_o+x_i-1,y)
        + in(4*x_o+x_i,y)
        + in(4*x_o+x_i+1,y)
```

Sec 4.2: Bounds Inference

```
let blurx.y.min =
  4*out.y_o.min + out.y_i.min - 1
```

Sec 4.4: Flattening

```
out[out.y.stride*(4*(out.y_o-out.y_o.min)+out.y_i)
    +4*(out.x_o-out.x_o.min)+out.x_i]
= blurx[blurx.y.stride*(out.y_i-1-blurx.y.min)+out.x_i-blurx.x.min]
+ blurx[blurx.y.stride*(out.y_i  -blurx.y.min)+out.x_i-blurx.x.min]
+ blurx[blurx.y.stride*(out.y_i+1-blurx.y.min)+out.x_i-blurx.x.min]
```

Sec 4.5: Vectorization

```
vec for blurx.x_i in 0..4
  blurx[blurx.y.stride*blurx.y+4*blurx.x_o+x_i) = ...
        ↓
blurx[blurx.y.stride*blurx.y+4*blurx.x_o+ramp(4)]
= in[in.y.stride*(blurx.y.min+blurx.y)+4*blurx.x_o+ramp(4)]
+ ...
```

Figure 5. Our compiler is driven by an autotuner, which stochastically searches the space of valid schedules to find a high performance implementation of the given Halide program. The core of the compiler lowers a functional representation of an imaging pipeline to imperative code using a *schedule*. It does this by first constructing a loop nest producing the final stage of the pipeline (in this case **out**), and then recursively injecting the storage and computation of earlier stages of the pipeline at the loop levels specified by the schedule. The locations and sizes of regions computed are symbolic at this point. They are resolved by the subsequent bound inference pass, which injects interval arithmetic computations in a preamble at each loop level that set the region produced of each stage to be at least as large as the region consumed by subsequent stages. Next, sliding window optimization and storage folding remove redundant computation and excess storage where the storage granularity is above the compute granularity. A simple flattening transform converts multidimensional coordinates in the infinite domain of each function into simple one-dimensional indices relative to the base of the corresponding buffer. Vectorization and unrolling passes replace loops of constant with k scheduled as *vectorized* or *unrolled* with the corresponding k-wide vector code or k copies of the loop body. Finally, backend code generation emits machine code for the scheduled pipeline via LLVM.