Decoupling Performance and Correctness with User-Schedulable Languages

Andrew Adams

About me

Stanford	2004-2010	Computational Photography
MIT	2011-2013	Halide
Google	2013-2017	Halide + Google Pixel camera pipeline
Facebook	2017-2019	Halide
Adobe	2019-	Halide + Photoshop

Part 1:

What are user-schedulable languages and why should we care?

Why care about user-schedulable languages?

A programming language is the interface between devs and hardware

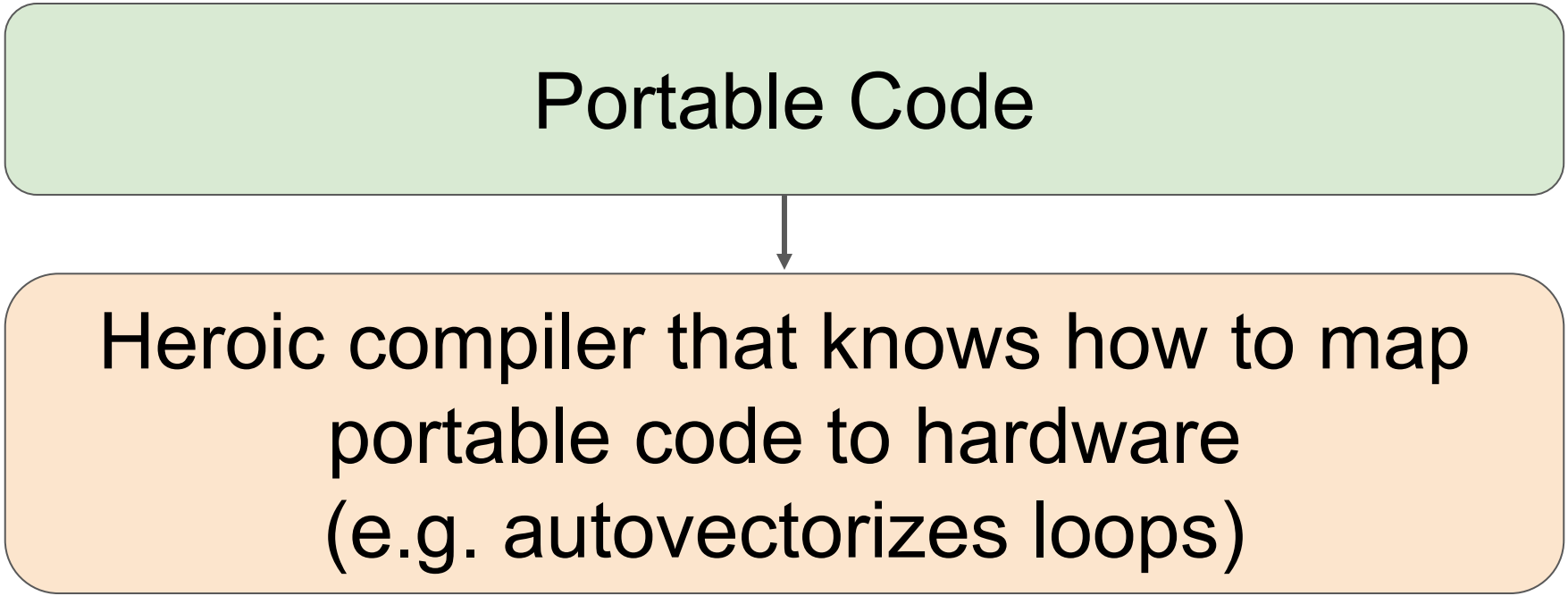
How productive it is can make or break new hardware

Hardware efforts commonly fall into one of two failure modes

User-schedulable languages offer a way out of this dilemma

How should developers write code for my new hardware?

Option 1: Use a portable language (C, OpenCL)



Use a portable language (C, OpenCL)

Failure mode #1

Fast on benchmarks

Slow on real code

Heroic compilers suck

Features unique to novel hardware are hidden behind an optimizer

~5 years of work for an optimizing compiler to do a good job on unseen code

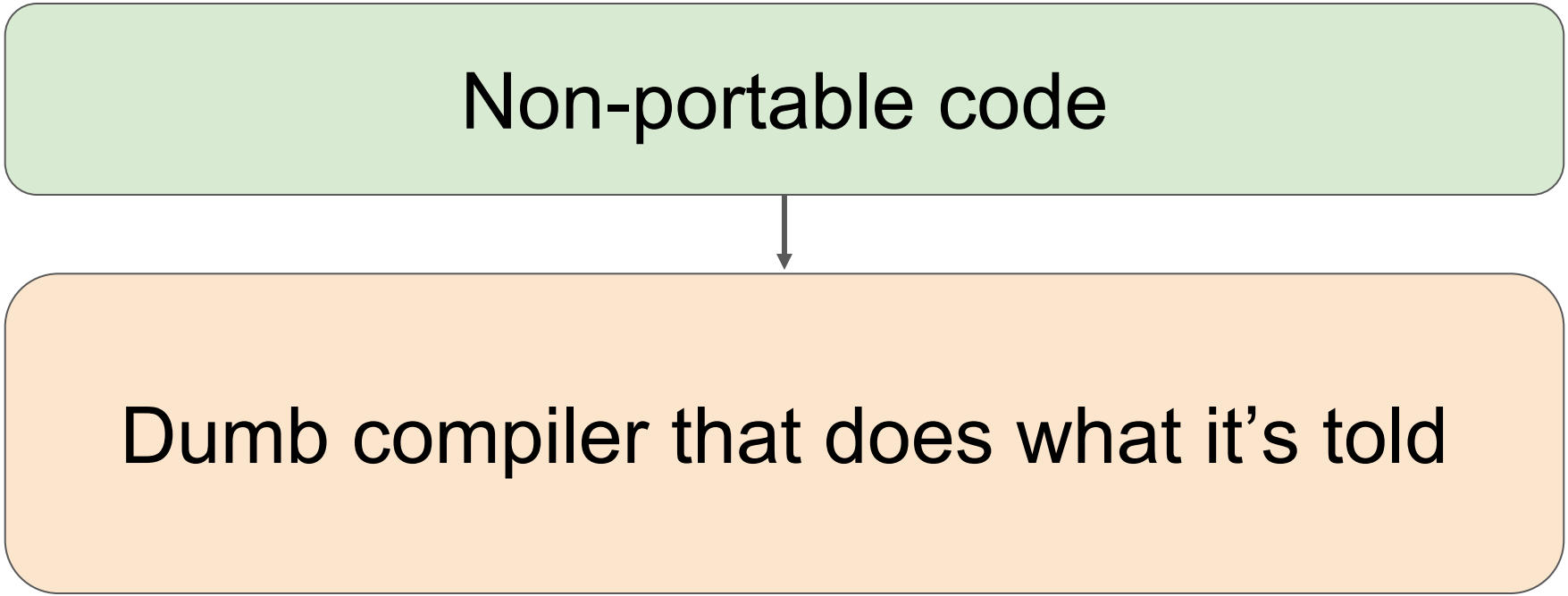
- Until then, compiler engineer in the loop for new code

Even then, their task is often impossible

- Portable languages under-specify behavior
- Portable languages over-specify behavior

Result: Users fight with the compiler. Most lose. Code is slow.

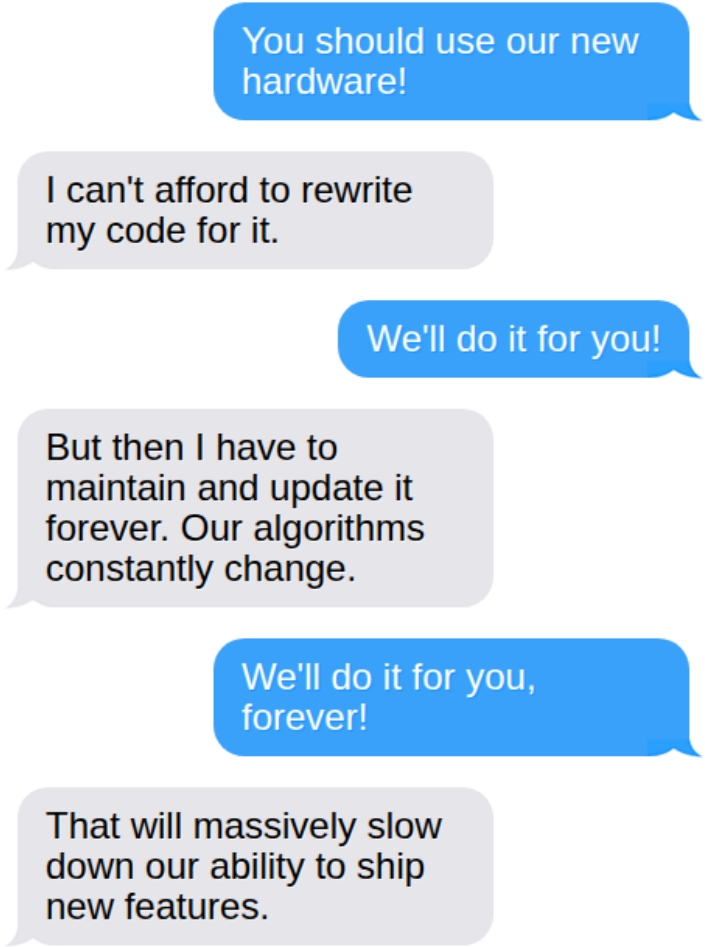
Option 2: Use a non-portable language (C with intrinsics)



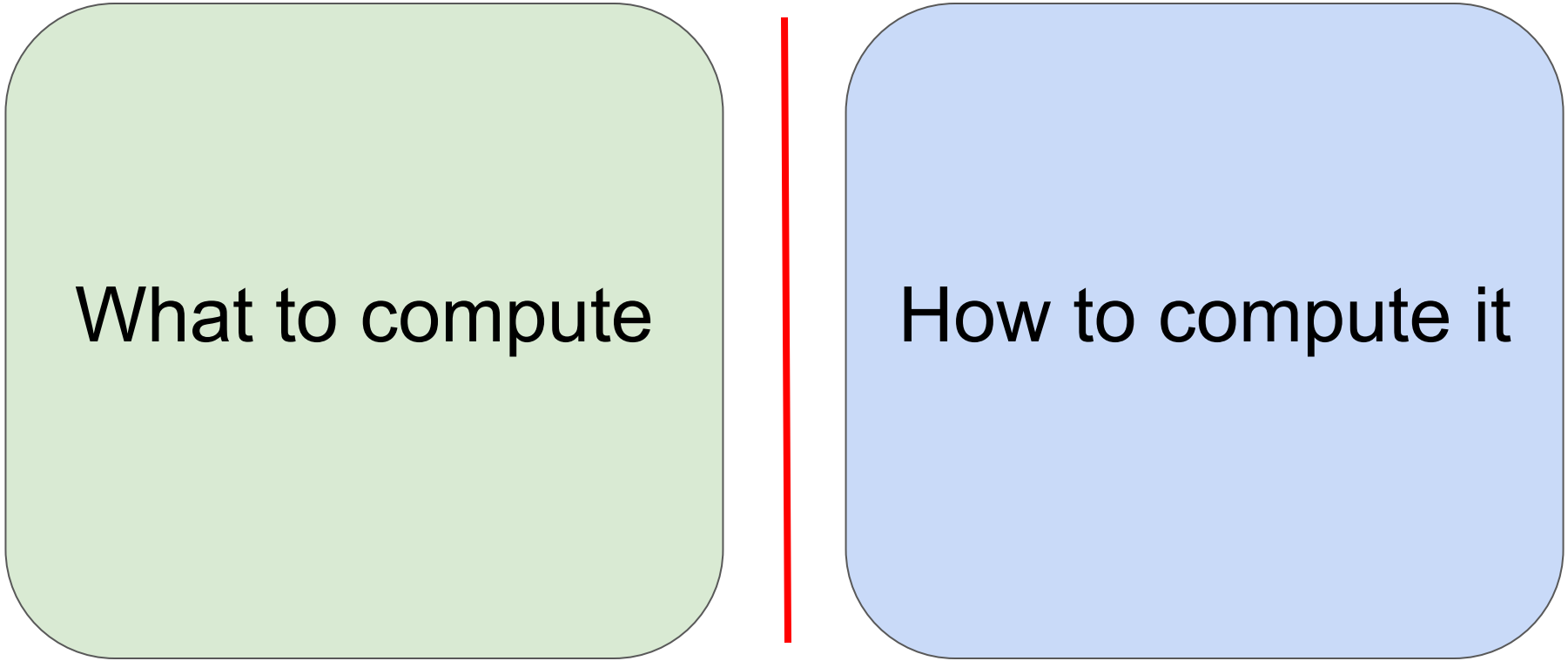
Use a non-portable language (e.g. C with intrinsics)

Failure mode #2

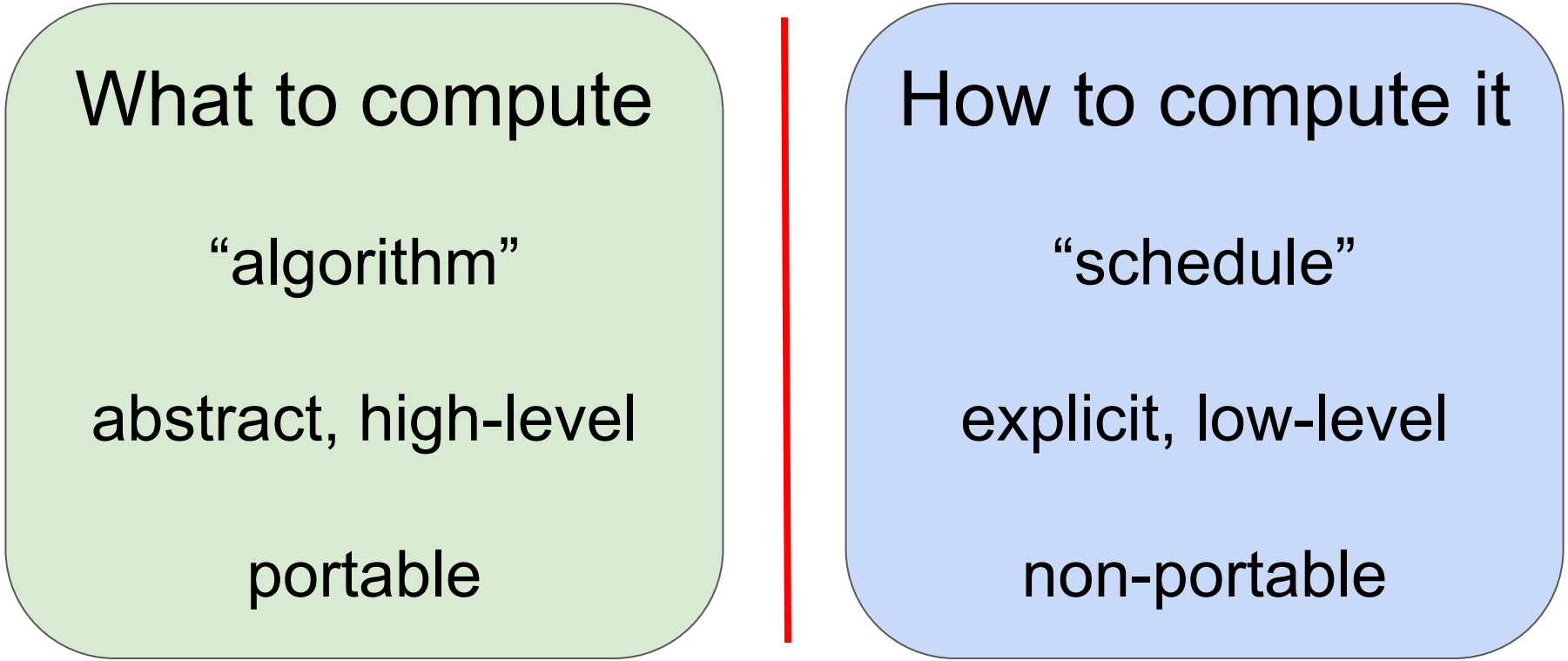
Nobody wants to write code for it


You should use our new hardware!
I can't afford to rewrite my code for it.
We'll do it for you!
But then I have to maintain and update it forever. Our algorithms constantly change.
We'll do it for you, forever!
That will massively slow down our ability to ship new features.

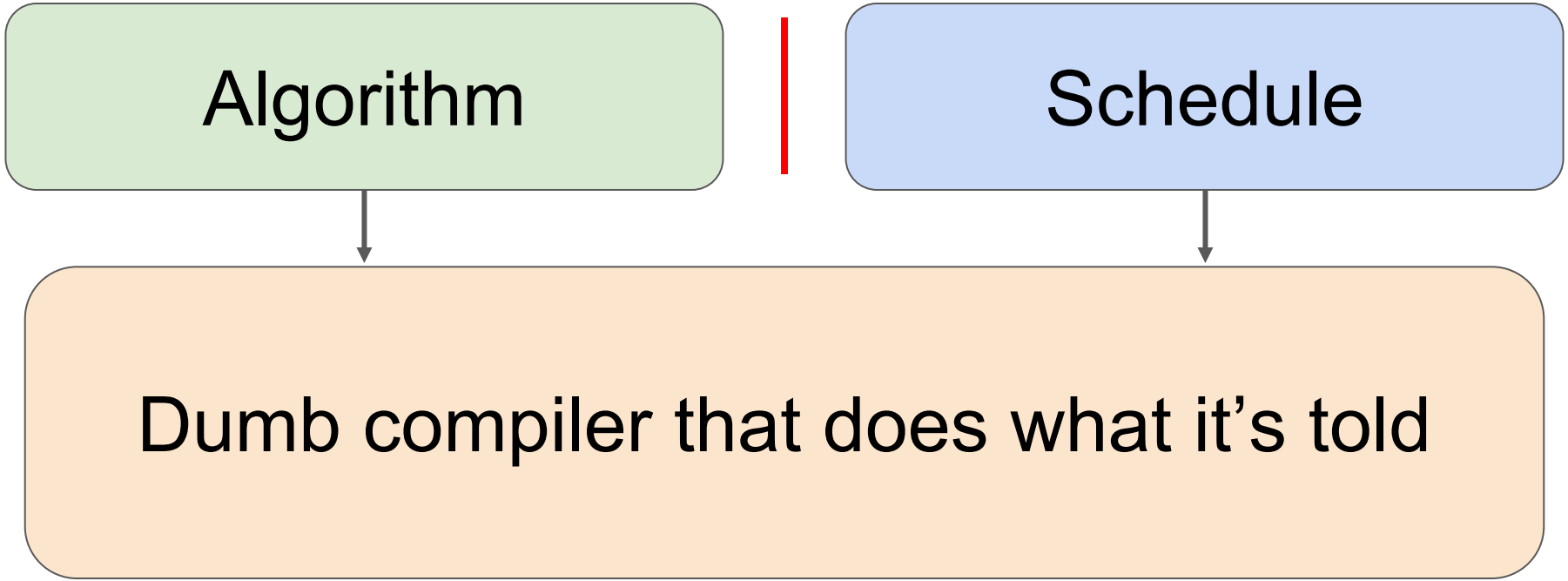
User-schedulable languages offer an alternative



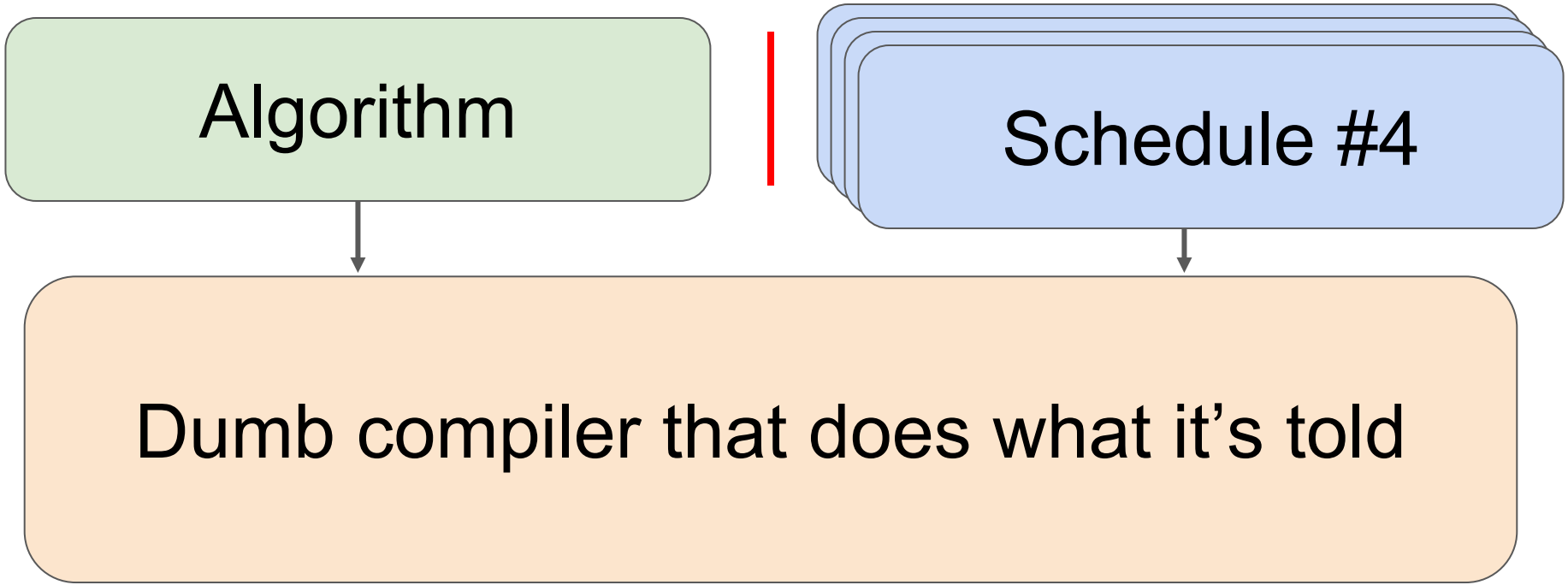
User-schedulable languages offer an alternative



User-schedulable languages offer an alternative



Can have a dumb compiler without forking algorithm code



The core guarantee:

Changing the schedule does not change the output

The core guarantee:

Changing the schedule does not change the output

Implications:

No schedule can cause a race condition

No schedule can cause out-of-bounds access

No schedule can cause uninitialized loads

No schedule can leak, crash …

User-schedulable languages make optimization easier

The core guarantee means you can rapidly try many schedules with needing to debug correctness

With Halide it is not uncommon to beat C with intrinsics

A Halide programmer can rapidly try many alternative optimization strategies

An C-with-intrinsics programmer only has time to try a few

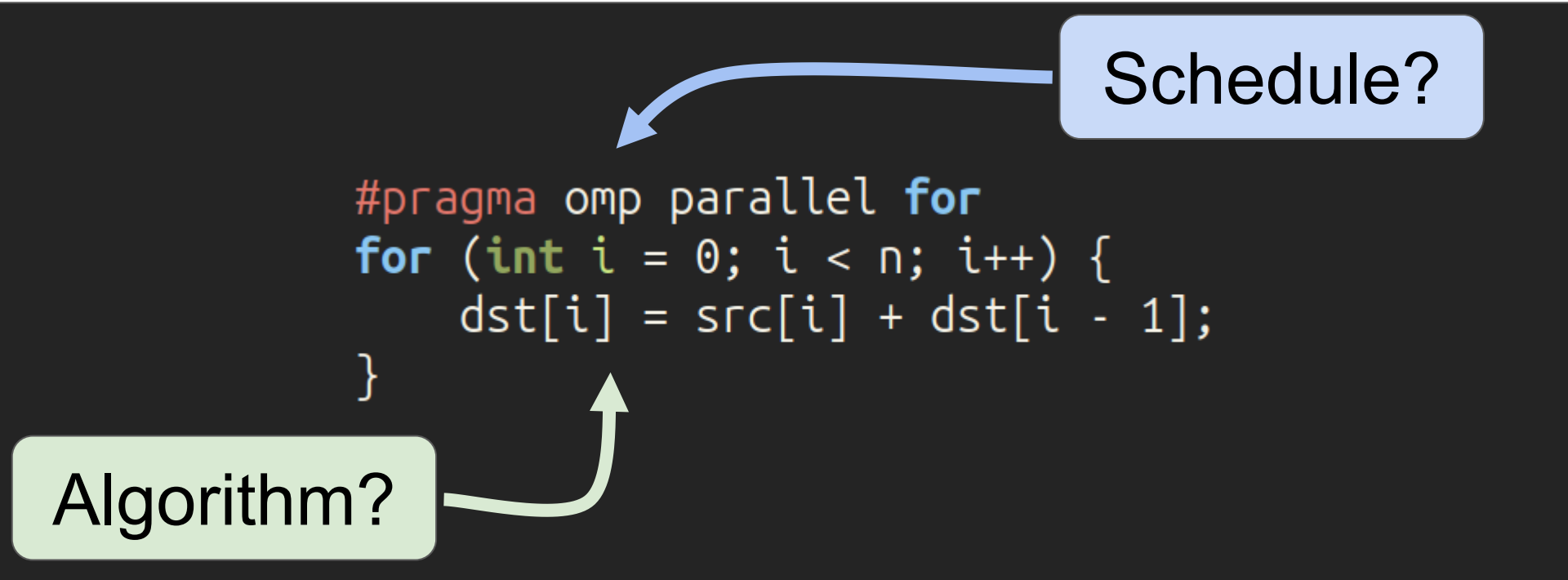
C-with-intrinsics rots quickly relative to lifespan of successful product

Different people can write the algorithm and the schedule

Many algorithm engineers assisted by few performance engineers

Pop quiz!

Are openmp pragmas a user-schedulable language?



Part 2:

A brief look at some user-schedulable languages

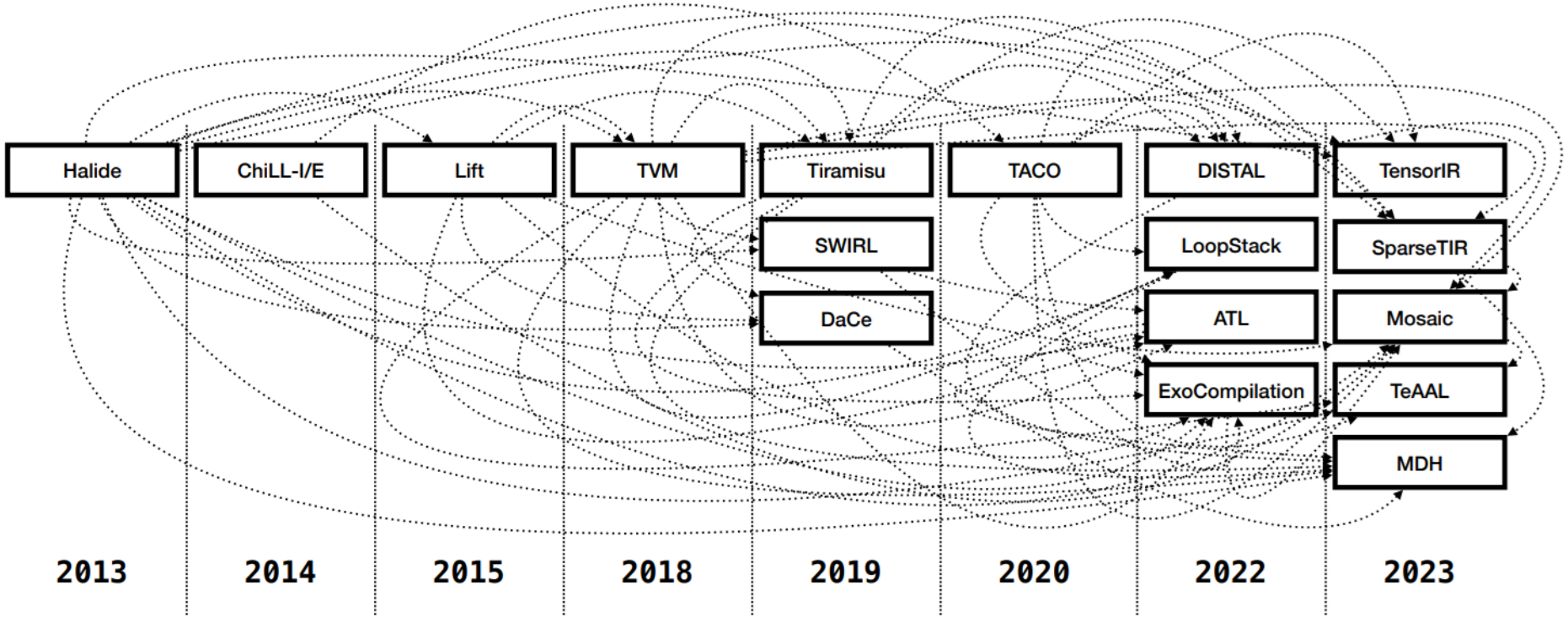


Fig. 5. Evolution of a number of scheduling approaches (arrows indicate citation).

Scheduling Languages: A Past, Present, and Future Taxonomy

Mary Hall, Cosmin Oancea, Anne C. Elster, Ari Rasch, Sameeran Joshi, Amir Mohammed Tavakkoli, Richard Schulze

Halide

Algorithm

```
blur_x(x, y) = (input(x-1, y) +
                input(x, y) +
                input(x+1, y));

blur_y(x, y) = (blur_x(x, y-1) +
                blur_x(x, y) +
                blur_x(x, y+1));
```

Schedule

```
blur_y.compute_root()
   .tile(x, y, xi, yi, 32, 8)
   .vectorize(xi, 8)
   .parallel(y);

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

Halide

Compute all of blur_y entirely, before proceeding to the next stage in the imaging pipeline

blur_y

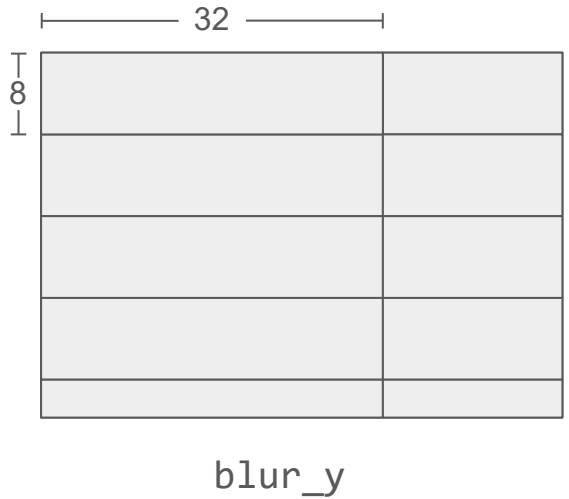
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute it in 32 x 8 tiles



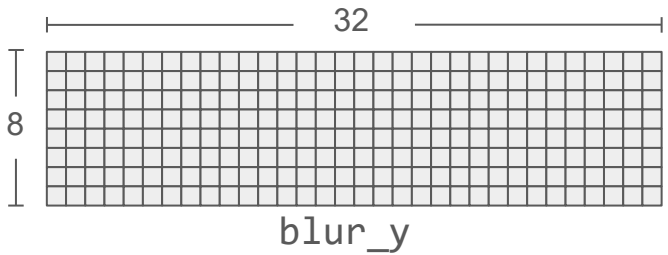
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Zooming in on one tile…



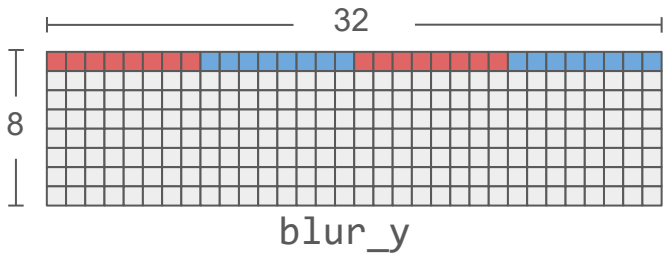
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Within those tiles, compute it in SIMD vectors of size 8



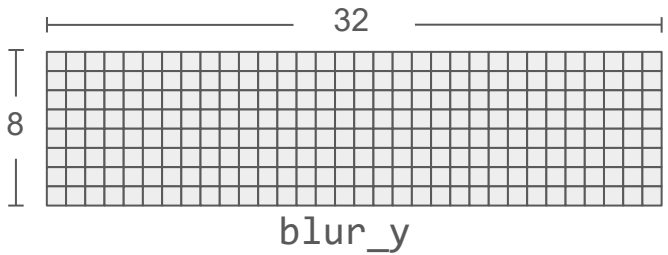
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute rows of tiles in parallel



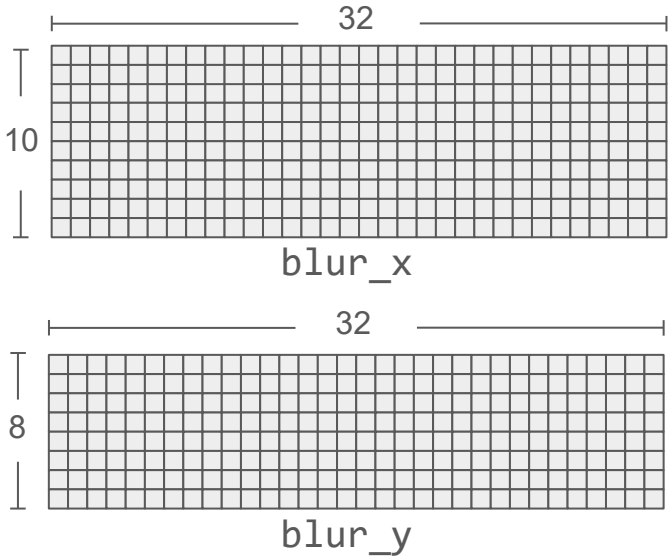
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Allocate scratch space to store enough of blur_x to compute one tile of blur_y



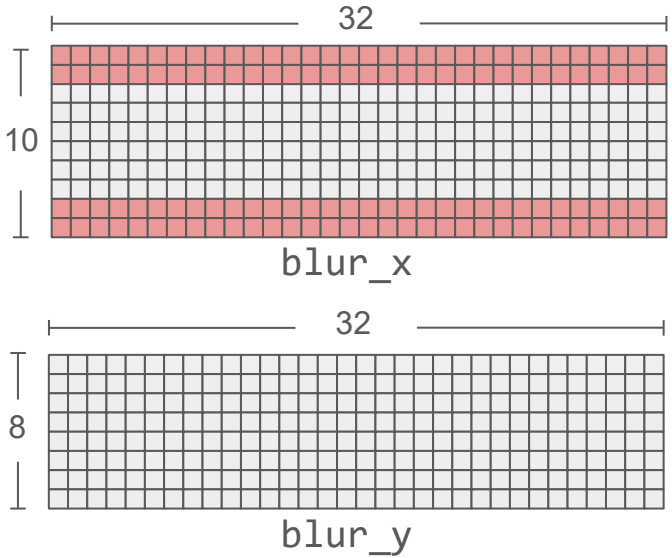
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Redundant recompute!
Other tiles of blur_y also need these values



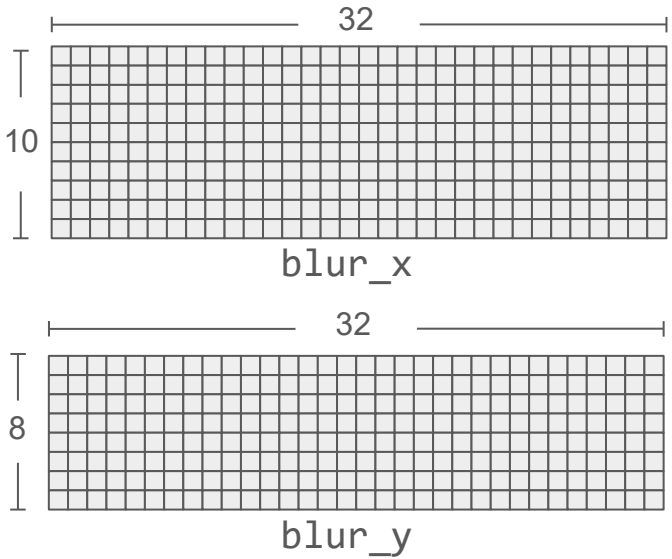
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute blur_x (into that scratch space) as needed per row of blur_y



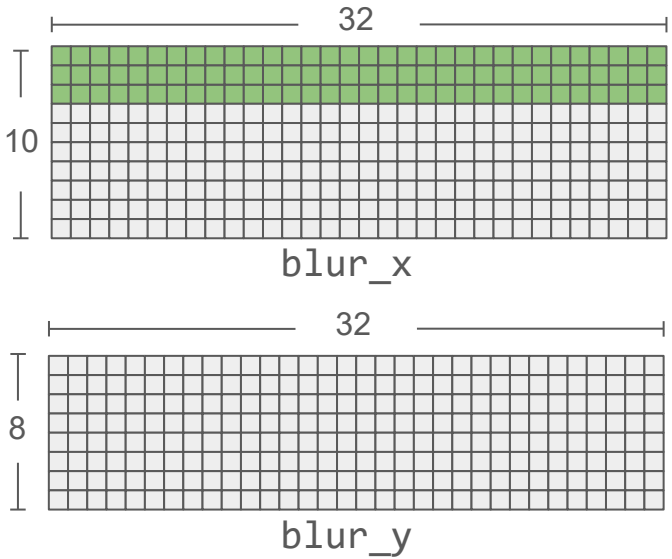
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute blur_x (into that scratch space) as needed per row of blur_y



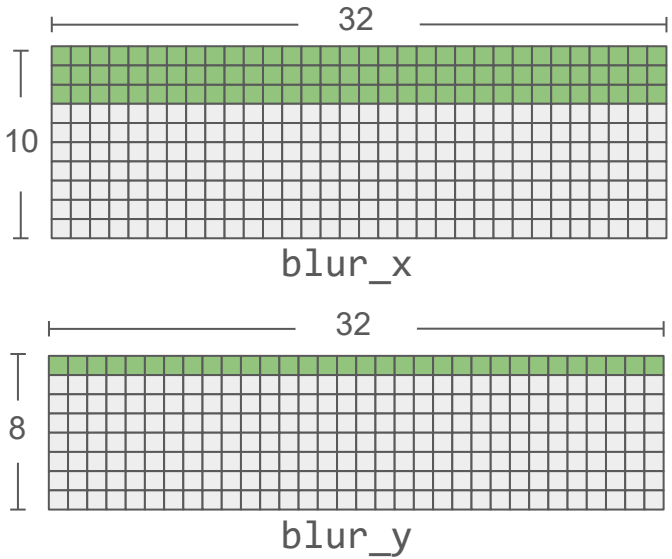
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute blur_x (into that scratch space) as needed per row of blur_y



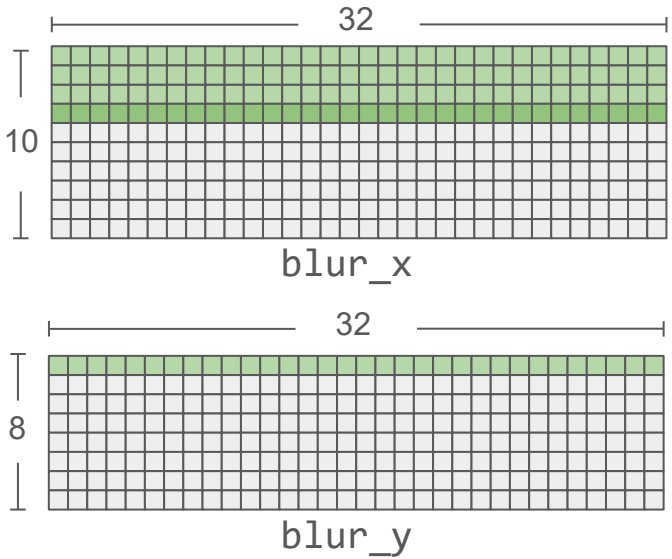
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute blur_x (into that scratch space) as needed per row of blur_y



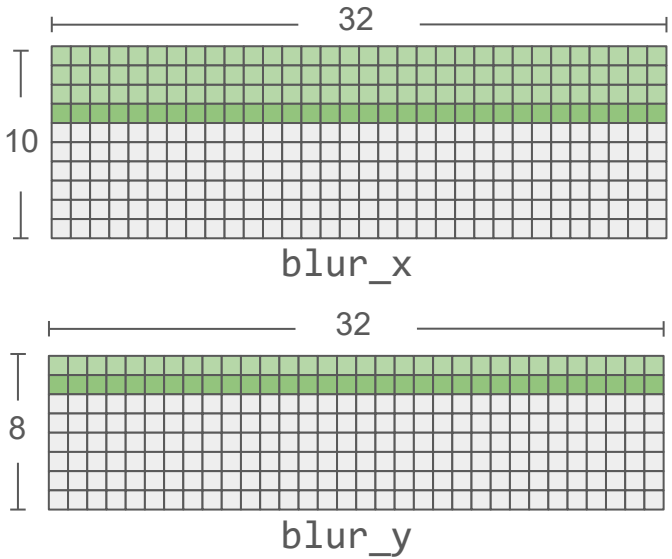
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute blur_x (into that scratch space) as needed per row of blur_y



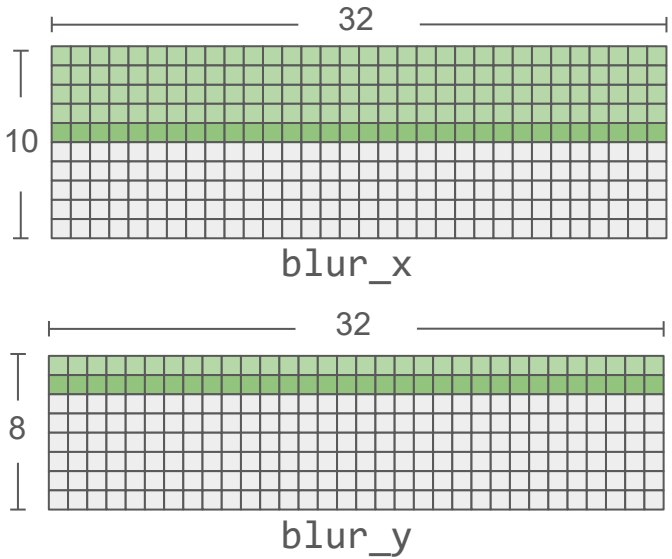
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute blur_x (into that scratch space) as needed per row of blur_y



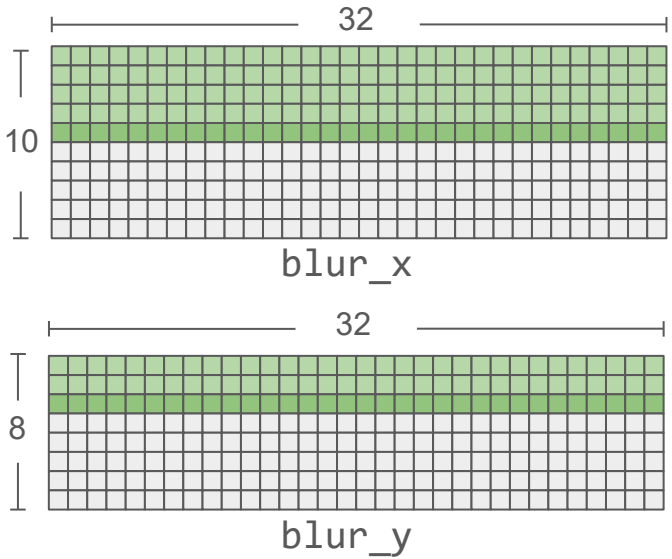
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute blur_x (into that scratch space) as needed per row of blur_y



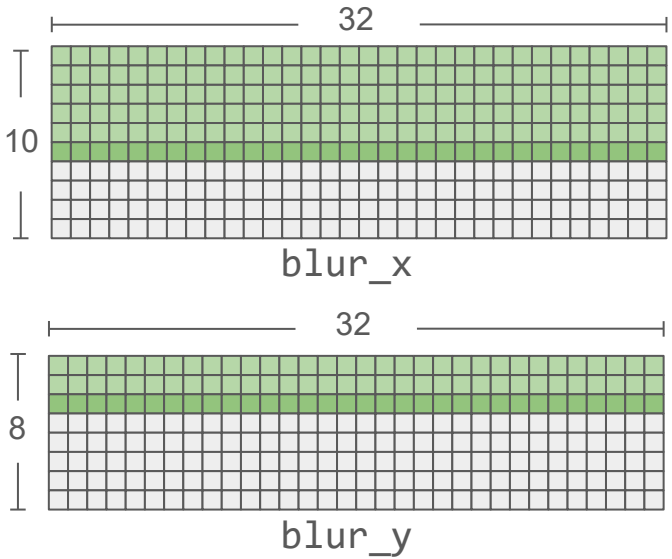
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute blur_x (into that scratch space) as needed per row of blur_y



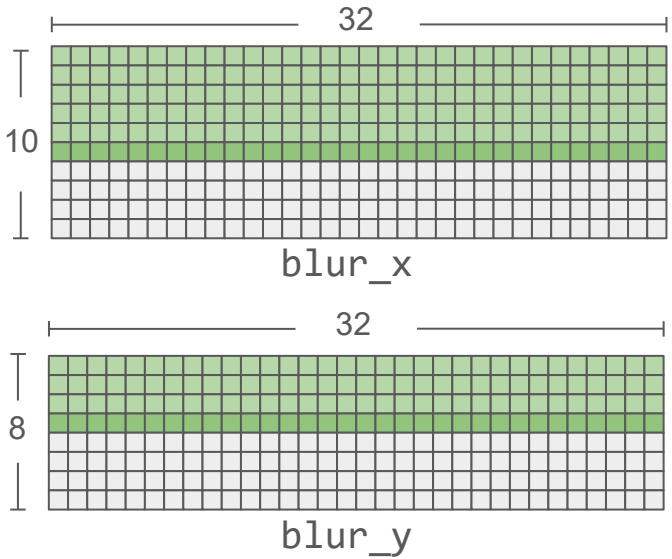
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute blur_x (into that scratch space) as needed per row of blur_y



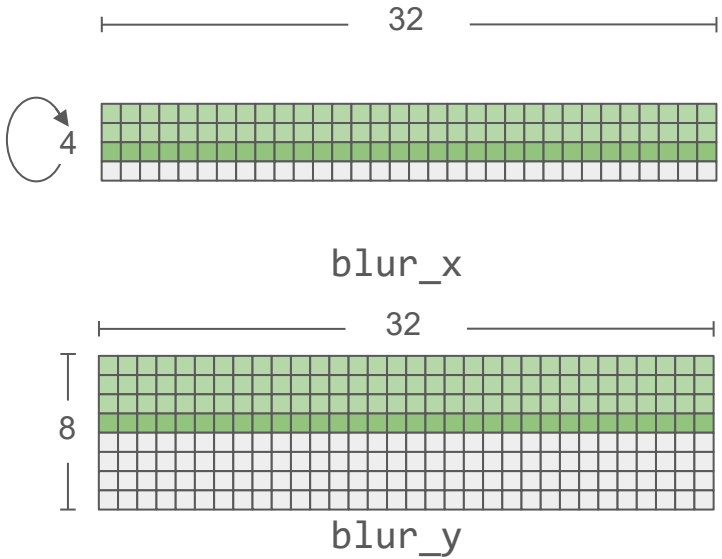
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

We only need a small sliding window of rows of blur_x, so we can fold it into a circular buffer.



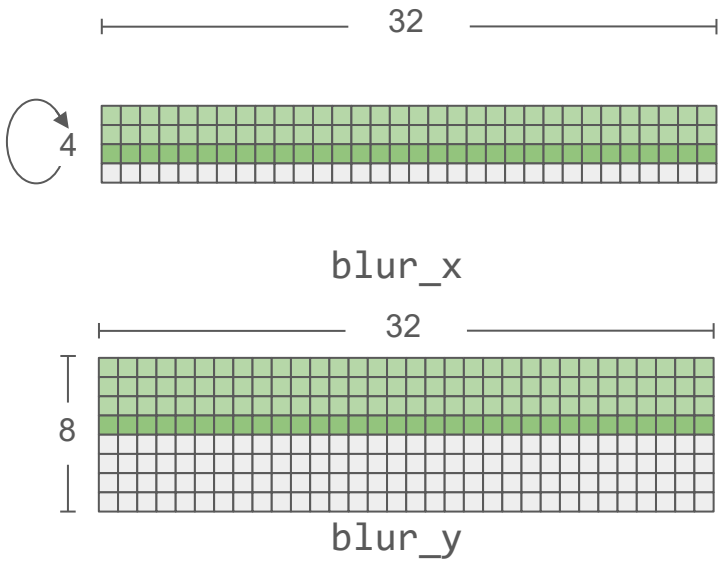
Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Compute blur_x in SIMD vectors of size 8



Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

Boring loop transformations

Lots of systems do this

Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

Halide

What made Halide unique

The ability to trade off redundant recompute, locality, and parallelism by controlling the granularity at which things get computed

The machinery that makes it work: Symbolic bounds inference

Schedule

```
blur_y.compute_root()
    .tile(x, y, xi, yi, 32, 8)
    .vectorize(xi, 8)
    .parallel(y);

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

A longer talk on Halide

https://youtu.be/1ir_nEfKQ7A

Me actually using Halide

https://youtu.be/UeyWo42_PS8

Taco

A language for sparse tensor algebra

The algorithm:

Individual sparse linear algebra kernels (SpMV, SpMM, etc.)

The schedule:

Sparse matrix formats, loop transformations

Productive because it generates fast kernels for a combinatorial combination of input and output sparse matrix formats from a single “algorithm”

The Tensor Algebra Compiler, Fredrik Kjolstad et al., OOPSLA 2017

TVM

Originally Halide-derived, but for ML

The algorithm:

- Neural network layers, including invocations of tensor accelerators

The schedule:

- Halide like, with some twists (reverse_compute_at!)

TVM became a full solution for neural networks that adds layers on top of this

- Scheduling language is now almost an internal implementation detail

TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, Tianqi Chen et al., OSDI 2018

Taichi

A language for kernels over hierarchical sparse voxel data structures in graphics, for rendering and simulation

The algorithm:

Sparse kernels over space (e.g. for fluid sim)

The schedule:

The hierarchical data structure (e.g. an octree of dense grids of hash tables)

Automatically exploits GPU-style (SPMD) parallelism

Now popular as a general way to accelerate graphics kernels written in python

Taichi: a language for high-performance computation on spatially sparse data structures, Yuanmin Hu et al. TOG 2019

Exo

Introduced the idea of “exocompilation” - making the backend something the user supplies, to support novel hardware accelerators

The algorithm:

Numerical programs on dense arrays (e.g. sgemm)

The schedule:

Rewrite rules on algorithm, with correctness checked automatically by z3 (*)

A library of Halide-style loop transformations can be built from these rewrite rules

Instruction selection for novel hardware is just more rewrite rules

Exocompilation for Productive Programming of Hardware Accelerators, Ikarashi et al. PLDI 2022

Future user-schedulable languages?

A language for quantizing numerical code

Schedule: Data types for different intermediate values

Weakened guarantee: Output is correct to some precision

A language for parallel tree traversal

Schedule: Mapping from a batch of tree lookups onto a GPU tree traversal

A language for Monte Carlo integration

Schedule: Sampling strategy used to estimate intermediate values

Weakened guarantee: Output is correct in expectation

Drawbacks of user-schedulable languages

Doesn’t lower expertise required, just increases productivity

Metaprogramming can be difficult and confusing

A rich scheduling space requires limiting the algorithm space

The compiler can only “just follow orders” to a point

- It must maintain the correctness guarantee
  - Sometimes results in fights with the user

LLMs are bad at generating code for them

Takeaways

User-schedulable languages separate the what (algorithm) from the how (schedule)

Algorithm is portable, high-level

Schedule specifies mapping from algorithm to hardware

Compiler can be dumb - just does what it’s told

Language guarantee required for productivity:

- All schedules produce the same output

Should the schedule be written by an AI agent?