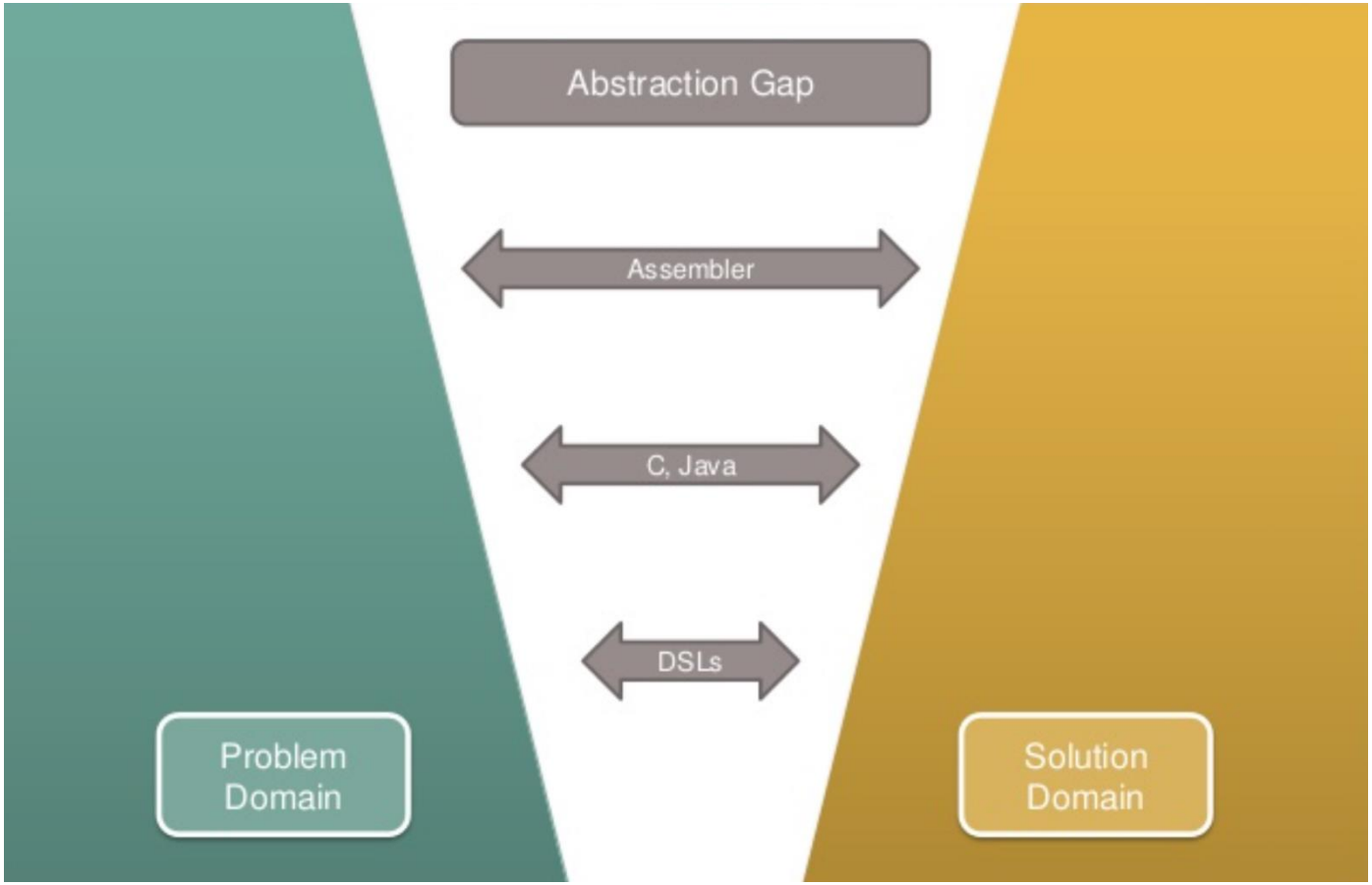
Lecture 24

Domain Specific Languages

I. Overview
II. Delite
III. Halide

I. Overview: What are Domain Specific Languages (DSLs)?



Languages designed to close the abstraction gap
between a problem domain and the code to express the solution

Some Popular DSLs

- SQL

```
CREATE TABLE Employee (
  id INT NOT NULL IDENTITY (1,1) PRIMARY KEY,
  name VARCHAR(50),
  surname VARCHAR(50),
  address VARCHAR(255),
  city VARCHAR(60),
  telephone VARCHAR(15),
)
```

- CSS

```
body {
  text-align: left;
  font-family: helvetica, sans-serif;
}
h1 {
  border: 1px solid #b4b9bf;
  font-size: 35px;}
```

- HTML

```
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <p>Example</p>
  </body>
</html>
```

- LaTeX

```
\ifthenelse{\boolean{showcomments}}
  {\newcommand{\nb}[2]{
    \fcolorbox{gray}{yellow}{
      \bfseries\sffamily\scriptsize#1
    }
    {\sf\small\textit{#2}}
    }
    \newcommand{\version}{\scriptsize$-$working$-$}
  }
  {\newcommand{\nb}[2]{}
    \newcommand{\version}{}
}
```

What was the first DSL you ever used?

Some More Recent DSLs

- **MapReduce** for big data processing
- **Halide** for image processing
- **GraphLab** [CMU] / **Pregel** for graph processing
  - "Think like a vertex"
- **Ligra** [CMU] for shared memory graph processing
  - edgeMap & vertexMap
- **Tensor Flow** for deep neural networks

MapReduce

- Popularized by Google
- Open source implementation called Hadoop MapReduce

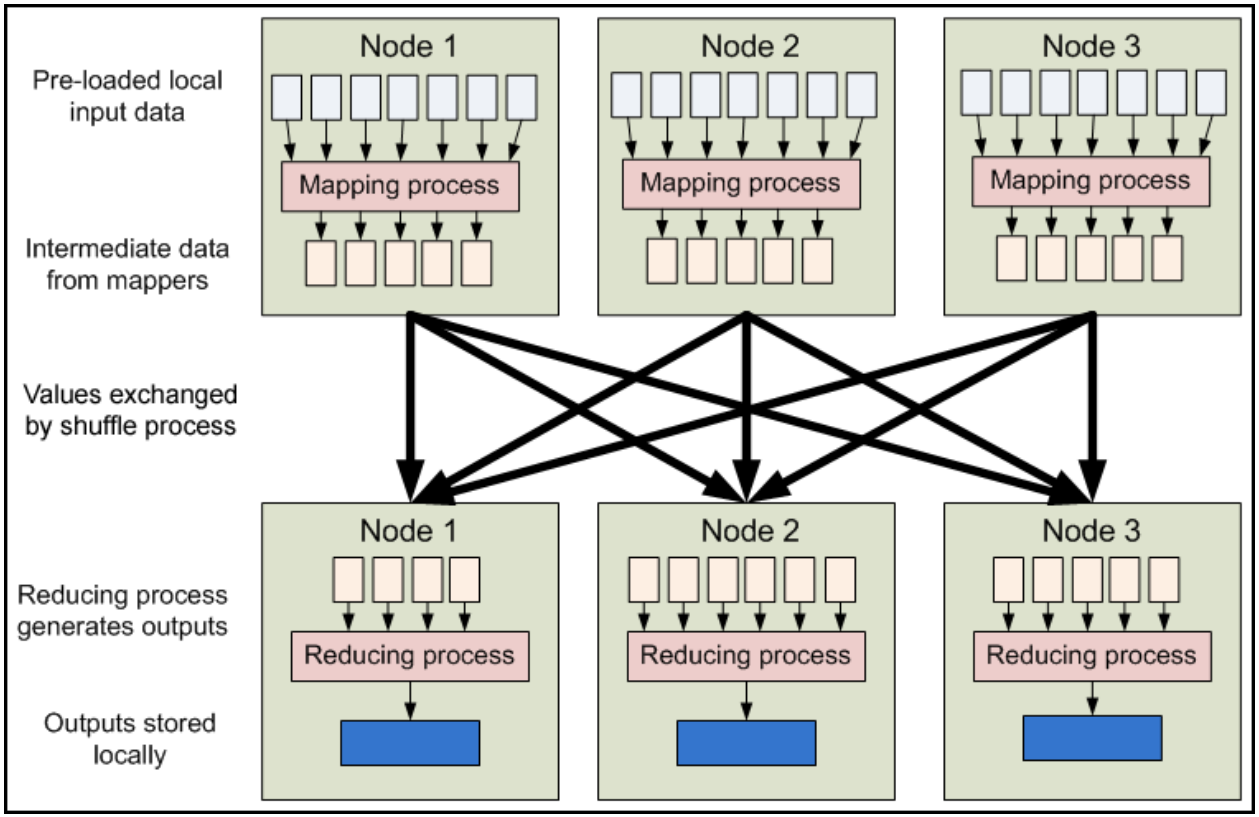


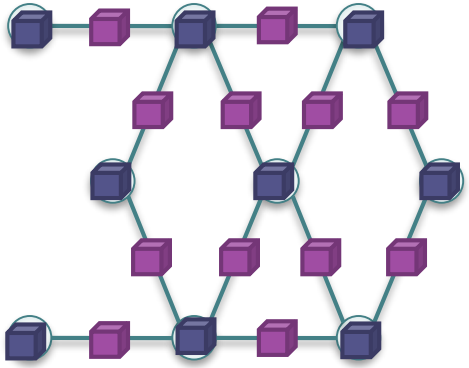
Image from: developer.yahoo.com/hadoop/tutorial/module4.html

GraphLab

Graph Parallel: “Think like a vertex”

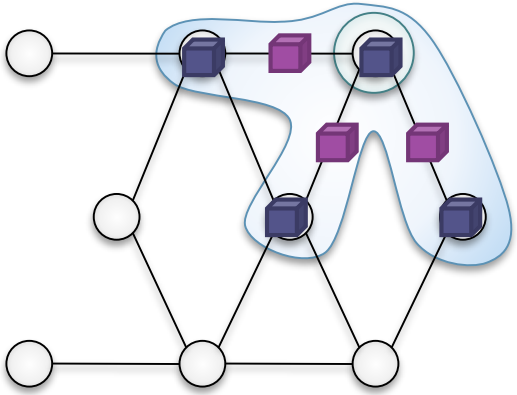
Graph Based

Data Representation

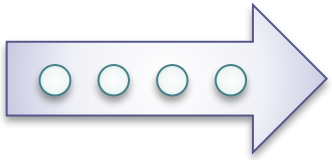


Update Functions

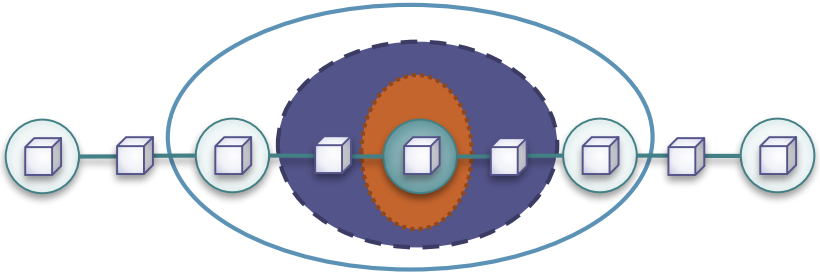
User Computation



Scheduler



Consistency Model



Slide courtesy of Carlos Guestrin

Advantages/Goals of DSLs

- Offer pre-defined abstractions to represent concepts from the application domain
  - Programming accessibility
    - Domain experts can readily write effective programs
    - More clear and intuitive
  - Programmer productivity
    - Fewer lines of code
    - Domain-specific tool support

- DSL compilers optimize the code written for the specific domain
  - High-performance
    - Higher-level (often declarative) abstraction and restrictive language constructs enable more optimizations
  - Portability
    - Across a range of hardware platforms

Design Guidelines for Domain Specific Languages

[Karsai et al, DSM'09]

- Language Purpose
  1. Identify language uses early
  2. Ask questions
  3. Make your language consistent

- Language Realization
  4. Decide carefully whether to use graphical or textual realization
  5. Compose existing languages where possible
  6. Reuse existing language definitions
  7. Reuse existing type systems

- Language Content
  8. Reflect only the necessary domain concepts
  9. Keep it simple

Design Guidelines for Domain Specific Languages

- Language Content (cont.)
  10. Avoid unnecessary generality
  11. Limit the number of language elements
  12. Avoid conceptual redundancy
  13. Avoid inefficient language elements

- Concrete Syntax
  14. Adopt existing notations domain experts use
  15. Use descriptive notations
  16. Make elements distinguishable
  17. Use syntactic sugar appropriately
  18. Permit comments
  19. Provide organizational structures for models
  20. Balance compactness and comprehensibility

Design Guidelines for Domain Specific Languages

- Concrete Syntax (cont.)
  21. Use the same style everywhere
  22. Identify usage conventions

- Abstract Syntax
  23. Align abstract and concrete syntax
  24. Prefer layout which does not affect translation from concrete to abstract syntax
  25. Enable modularity
  26. Introduce interfaces

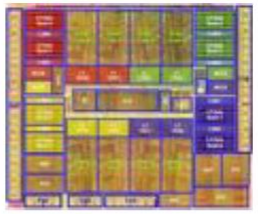
II. Delite

[Brown et al., PACT'11]

Performance
= heterogeneous
+ parallel

Compilers have
often not kept pace

Pthreads
OpenMP

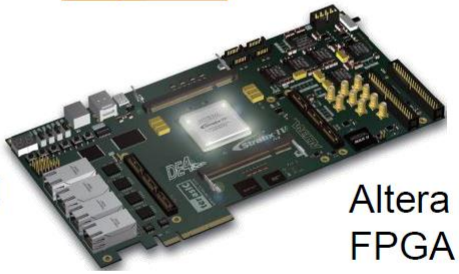


Sun
T2

CUDA
OpenCL

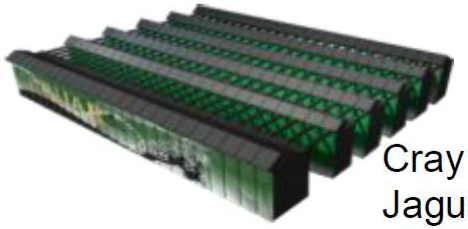
Nvidia
Fermi

Verilog
VHDL



Altera
FPGA

MPI



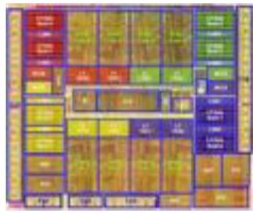
Cray
Jaguar

Programmability Chasm

Applications

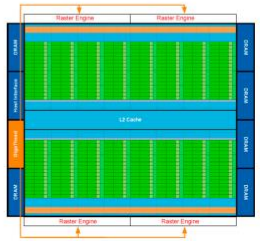
- Scientific Engineering
- Virtual Worlds
- Personal Robotics
- Data informatics

Pthreads
OpenMP



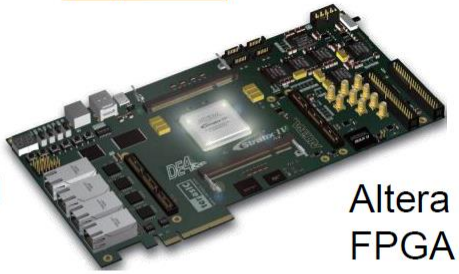
Sun T2

CUDA
OpenCL



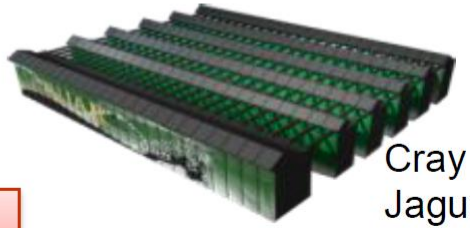
Nvidia Fermi

Verilog
VHDL



Altera FPGA

MPI



Cray Jaguar

Too many different programming models

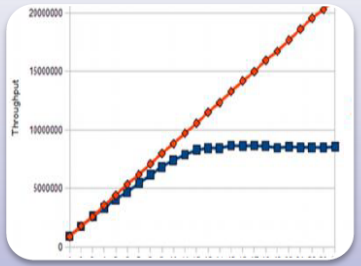
Benefits of Using DSLs for Parallelism

Productivity

- Shield most programmers from the difficulty of parallel programming
- Focus on developing algorithms and applications and not on low level implementation details

Performance

- Match high level domain abstraction to generic parallel execution patterns
- Restrict expressiveness to more easily and fully extract available parallelism
- Use domain knowledge for static/dynamic optimizations



Portability and forward scalability

- DSL & Runtime can be evolved to take advantage of latest hardware features
- Applications remain unchanged
- Allows innovative HW without worrying about application portability

DSLs: Compiler vs. Library

- *A Domain-Specific Approach to Heterogeneous Parallelism*, Chafi et al.
  - A framework for parallel DSL libraries
  - Used data-parallel patterns and deferred execution (transparent futures) to execute tasks in parallel

- Why write a compiler?
  - Static optimizations (both generic and domain-specific)
  - All DSL abstractions can be removed from the generated code
  - Generate code for hardware not supported by the host language
  - Full-program analysis

Common DSL Framework

- Building a new DSL
  - Design the language (syntax, operations, abstractions, etc.)
  - Implement compiler (parsing, type checking, optimizations, etc.)
  - Discover parallelism (understand parallel patterns)
  - Emit parallel code for different hardware (optimize for low-level architectural details)
  - Handle synchronization, multiple address spaces, etc.
- Need a DSL infrastructure
  - Embed DSLs in a common host language
  - Provide building blocks for common DSL compiler & runtime functionality

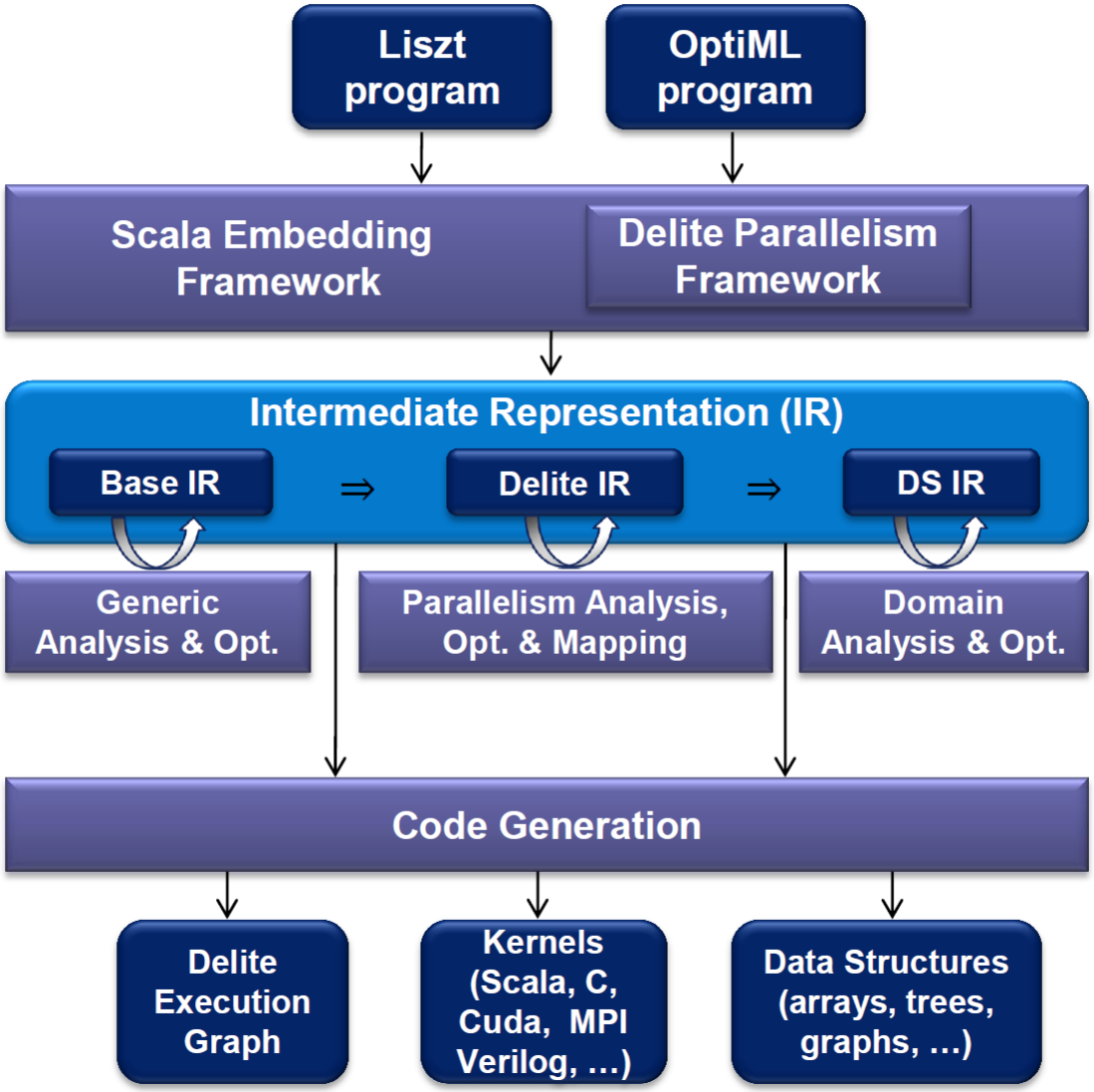
Need to simplify the process of developing DSLs for parallelism

- **Delite** provides a framework for creating heterogeneous parallel DSLs
- Performs generic, parallel, and domain-specific optimizations in a single system

Delite Ops

- Encode known parallel execution patterns
  - Map, filter, reduce, …
  - Bulk-synchronous foreach
  - Divide & conquer
- Delite provides implementations of these patterns for multiple hardware targets
  - e.g., multi-core, GPU
- DSL author maps each domain operation to the appropriate pattern
  - Delite handles parallel optimization, code generation, and execution for all DSLs

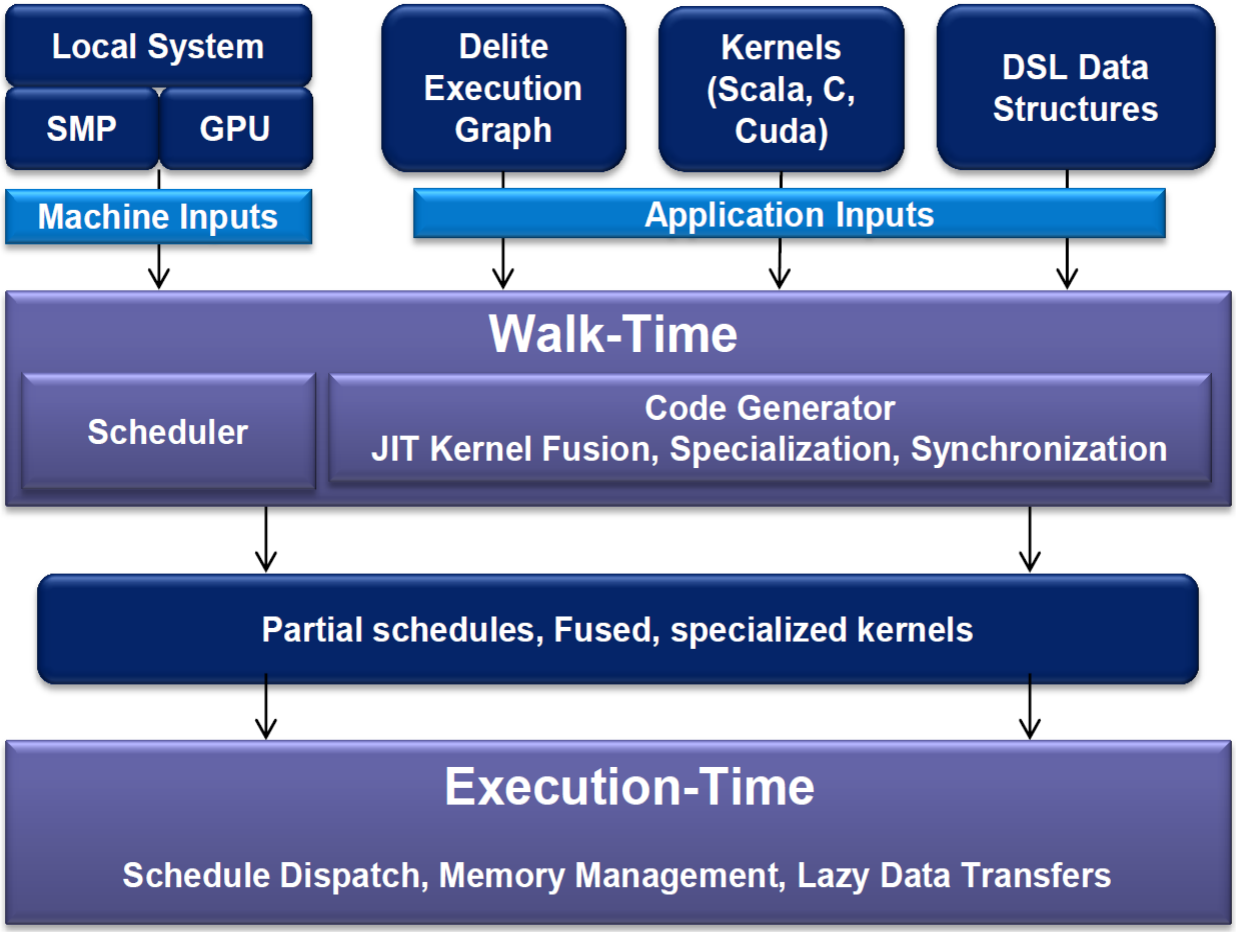
Delite DSL Compiler



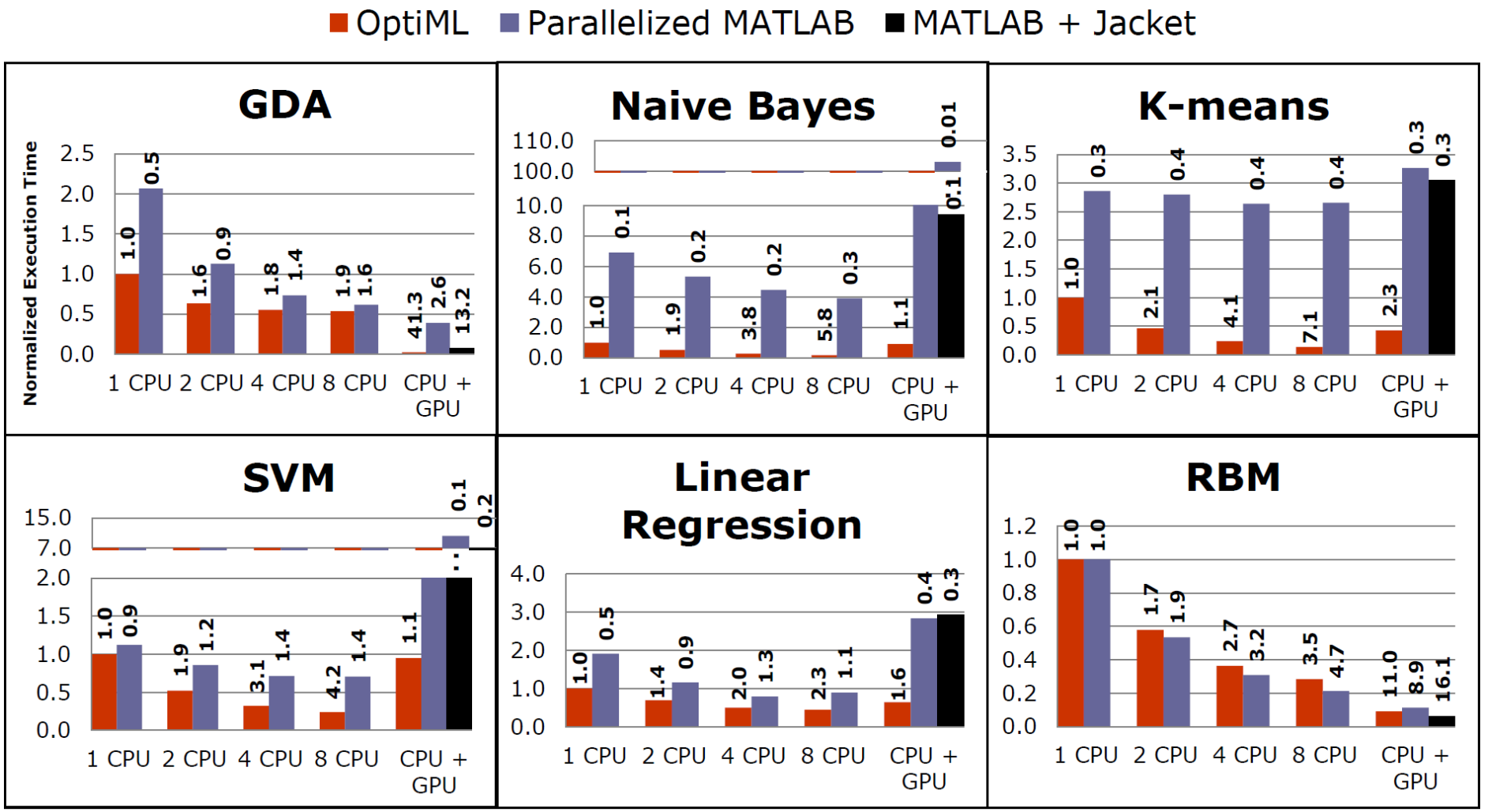
Generic IR

- Optimizations
  - Common subexpression elimination (CSE)
  - Dead code elimination (DCE)
  - Constant folding
  - Code motion (e.g., loop hoisting)
- Side effects and alias tracking
- All performed at the granularity of DSL operations
  - e.g., MatrixMultiply

Delite Runtime



Experiments on ML kernels



OptiML+Delite outperforms MATLAB

[Ragan-Kelly et al., PLDI'13]

III. Halide

- **Open-source DSL for the complex image processing pipelines** found in modern computational photography and vision applications

- A systematic model of the tradeoffs between locality, parallelism, and redundant recomputation in stencil pipelines;
- a scheduling representation that spans this space of choices;
- a DSL compiler based on this representation that combines Halide programs and schedule descriptions to synthesize points anywhere in this space, using a design where the choices for how to execute a program are separated not just from the definition of what to compute, but are pulled all the way outside the black box of the compiler;
- a loop synthesizer for data parallel pipelines based on simple interval analysis, which is simpler and less expressive than polyhedral model, but more general in the class of expressions it can analyze;
- a code generator that produces high quality vector code for image processing pipelines, using machinery much simpler than the polyhedral model;
- an autotuner that can infer high performance schedules—up to 5 faster than hand-optimized programs written by experts—for complex image processing pipelines using stochastic search.

We are surrounded by computational cameras

Enormous opportunity,
demands extreme optimization
parallelism & locality limit
performance and energy

Camera: 8 Mpixels
(96MB/frame as *float*)

CPUs: 15 GFLOP/sec

GPU: 115 GFLOP/sec

Required arithmetic intensity > 40:1

Methodology Prior to Halide

C++ w/multithreading, SIMD
CUDA/OpenCL
OpenGL/RenderScript

Optimization requires manually
transforming program & data structure
for locality and parallelism.

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

Local Laplacian Filters

in Adobe Photoshop Camera Raw / Lightroom

1500 lines of expert-optimized C++

multi-threaded, SSE

3 months of work

***10x faster** than reference C++*

Halide: 60 lines

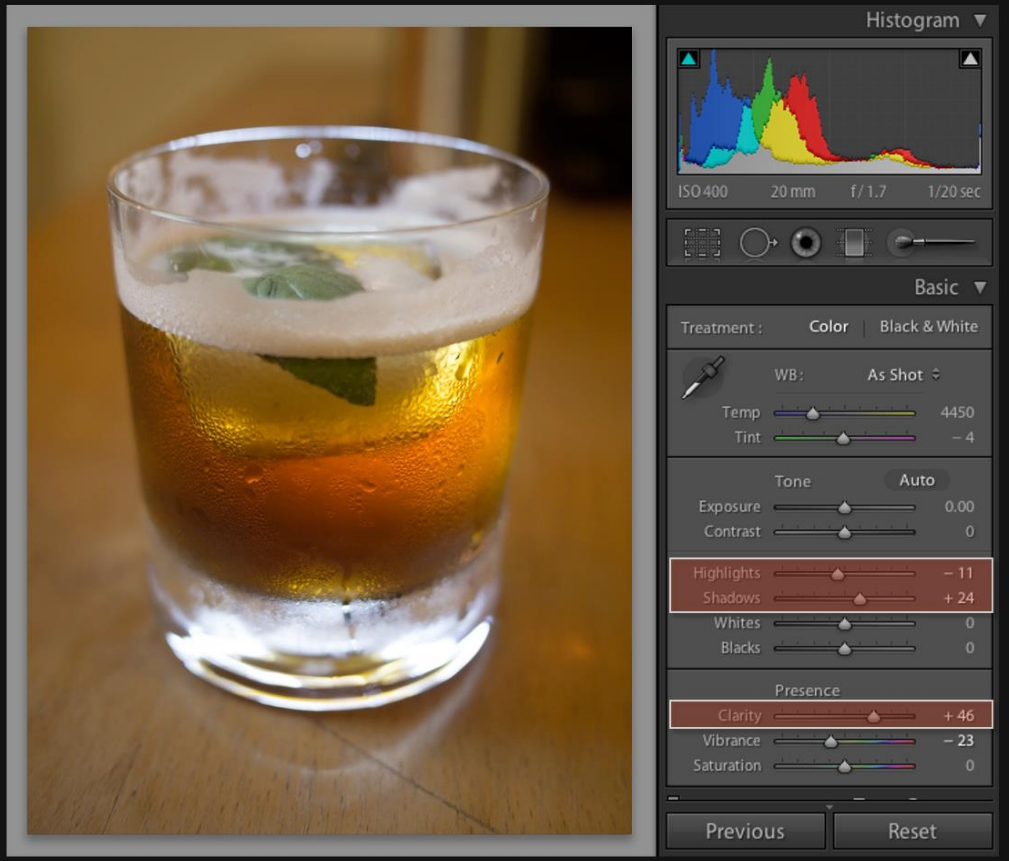
1 intern-day

20x faster (vs. reference)

2x faster (vs. Adobe)

GPU: 70x faster (vs. reference)

7x faster (vs. Adobe)



A simple example: 3x3 blur

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


  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)

      blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
```

Hand-optimized C++

9.9 → 0.9 ms/megapixel

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

11x faster
(quad core x86)

Tiled, fused

Vectorized

Multithreaded

Redundant computation

Near roof-line optimum

Halide's answer: *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 defines pipelines as pure functions

Pipeline stages are *functions* from coordinates to values

Execution order and storage are unspecified

3x3 blur as a Halide *algorithm*:

```
Var x, y; Func blurx, blury;
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;
```

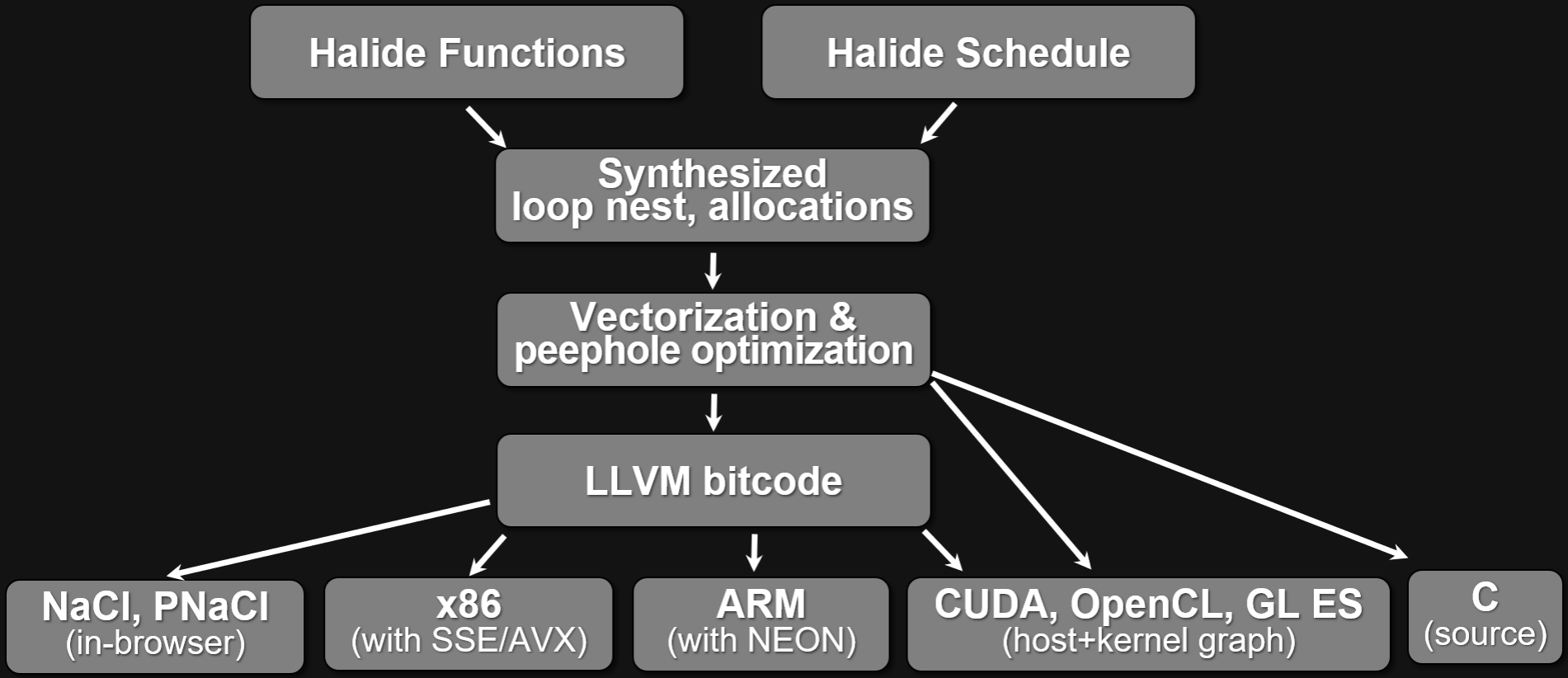
Domain scope of the programming model

All computation is over **regular grids**.

not Turing complete

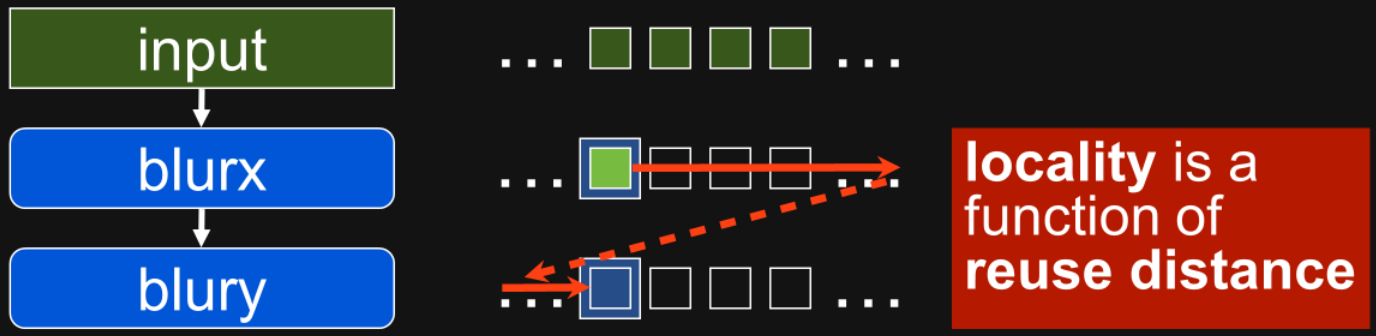
- Only **feed-forward pipelines**
  Recursive/reduction computations are a (partial) escape hatch.
- **Recursion** must have **bounded depth.**

The Halide Compiler



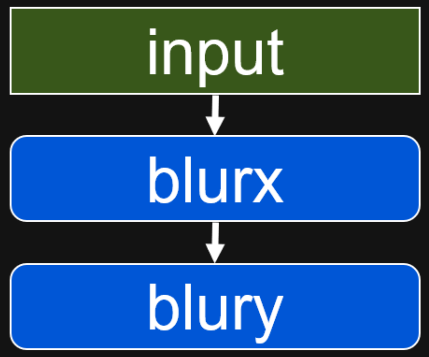
Parallelism vs. Locality

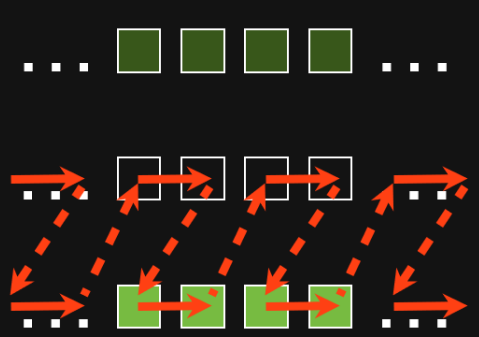
Breadth-first execution **sacrifices locality**



Interleaved execution (fusion) **improves locality**

fusion ***globally interleaves*** *computation*



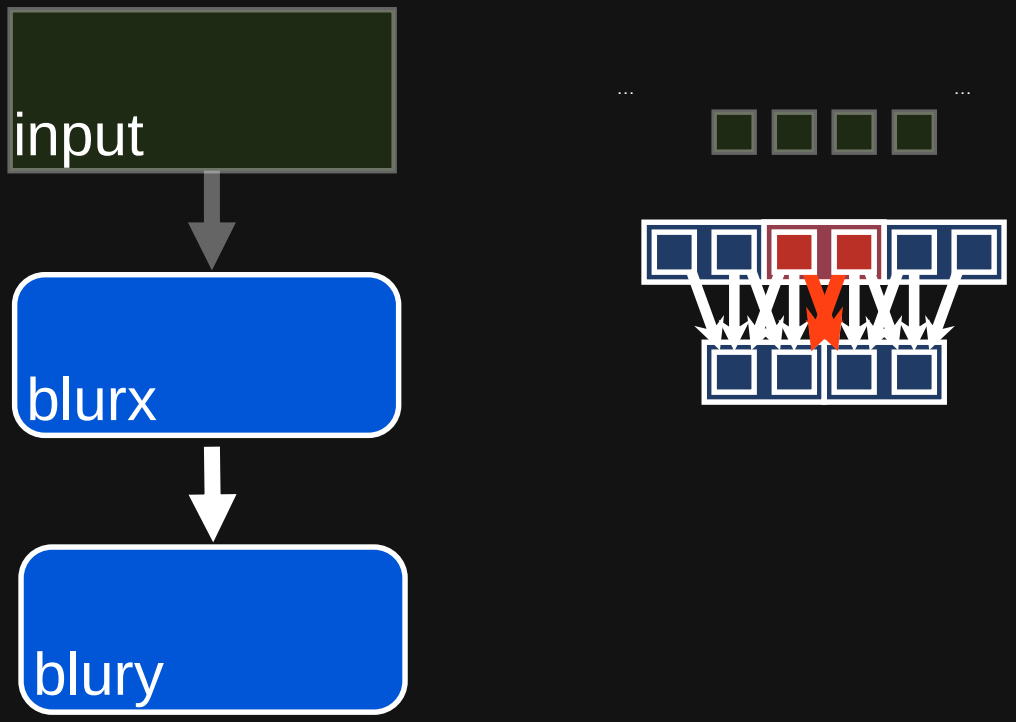


reduce reuse distance from **producer** to **consumer**

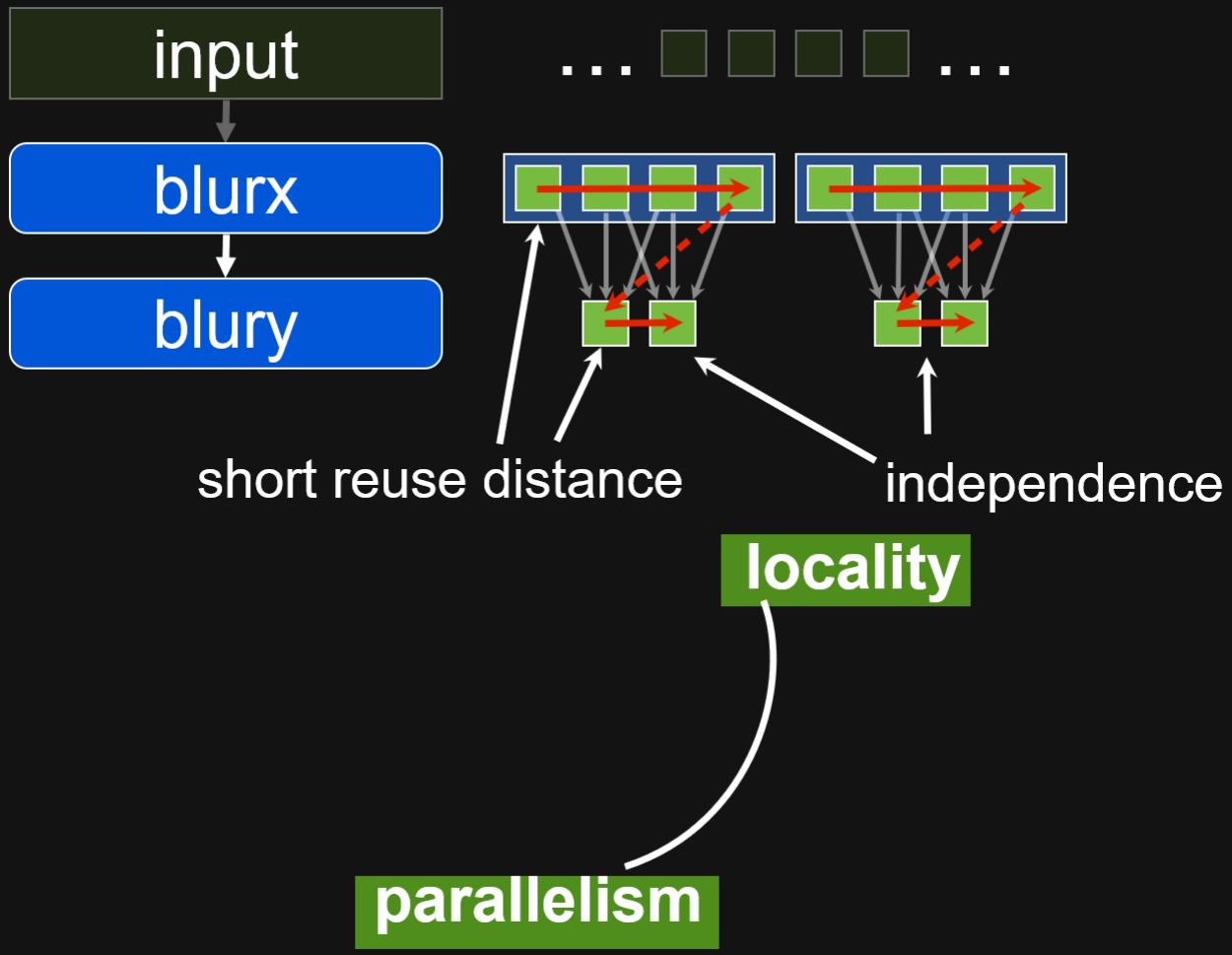
locality

parallelism

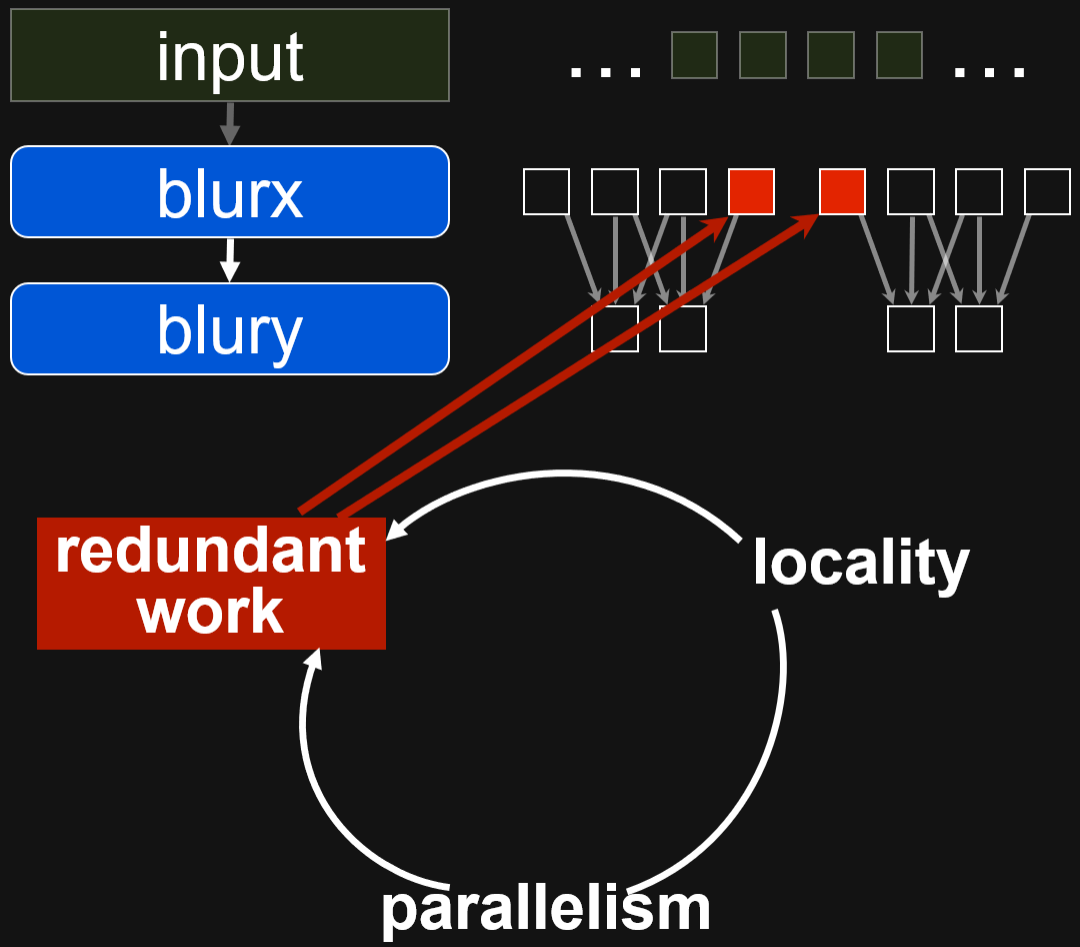
Stencils have overlapping dependencies



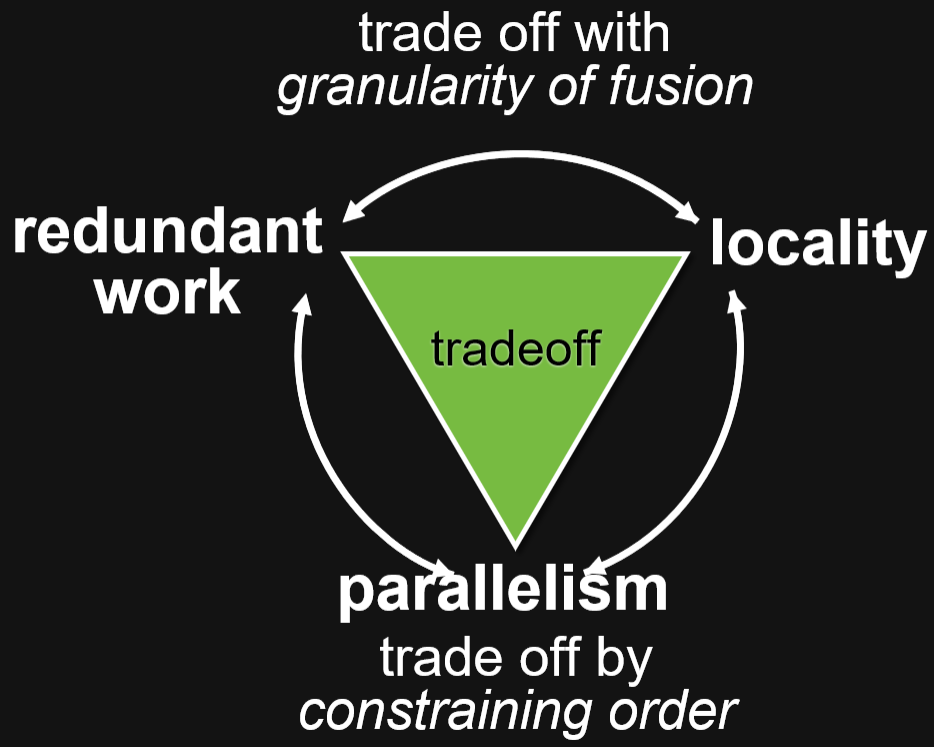
Decoupled tiles optimize **parallelism & locality**



Breaking dependencies introduces **redundant work**



Stencil pipelines require **tradeoffs** determined by **organization of computation**



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



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

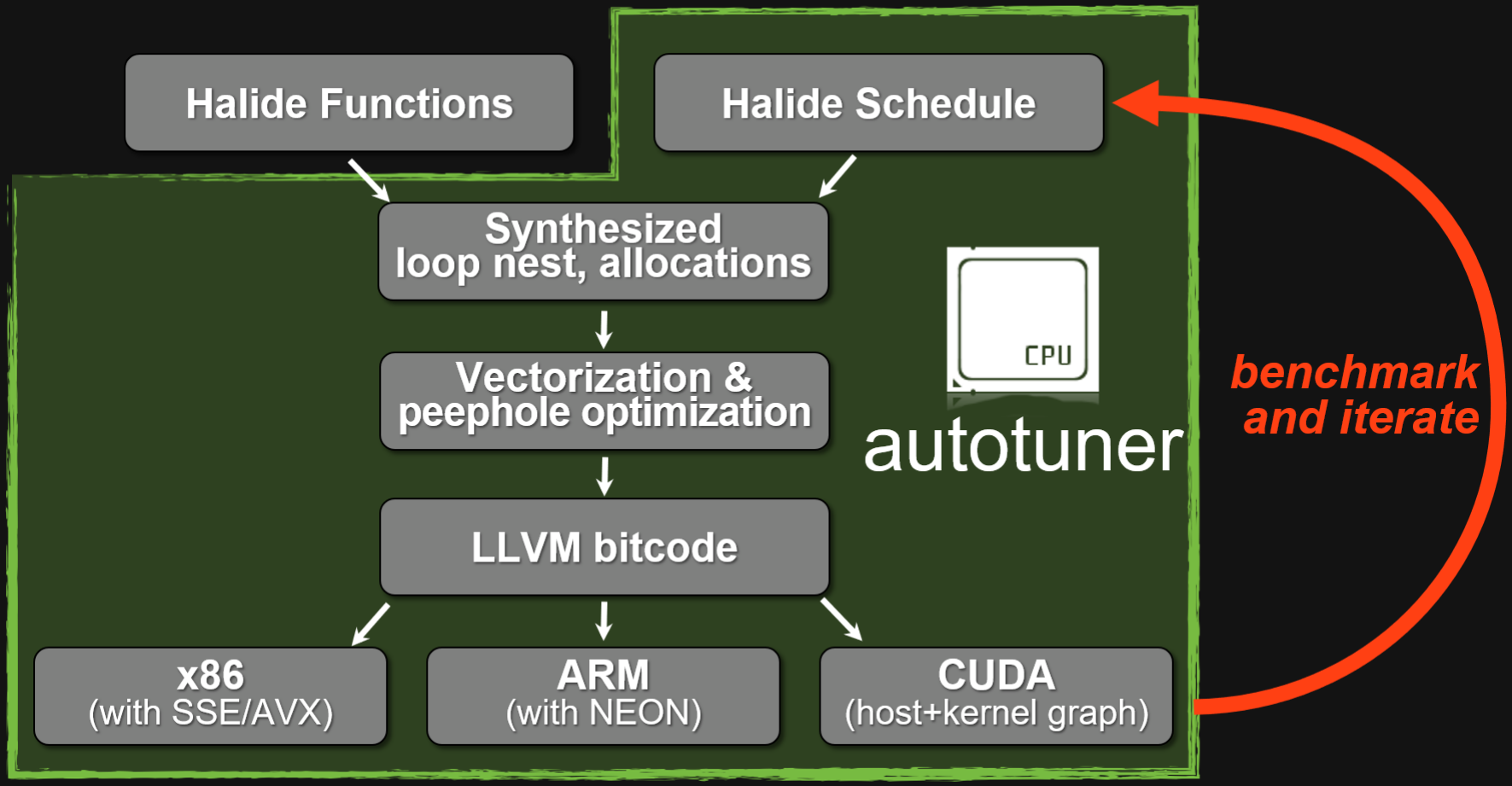
```
blur_x.store_root().compute_at(blur_y, x);
```

```
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's Autotuner Stochastically Searches for High-Performance Code



Prior work*

*a tiny sample. Thousands have come before us.

Streaming languages
Ptolemy [Buck et al. 1993]
StreamIt [Thies et al. 2002]
Brook [Buck et al. 2004]

Loop transformation
Systolic arrays [Gross & Lam 1984]
Polyhedral model [Ancourt & Irigoin 1991, Amarasinghe & Lam 1993]

Parallel work scheduling
Cilk [Blumhofe et al. 1995]
NESL [Blelloch et al. 1993]

Region-based languages
ZPL [Chamberlain et al. 1998]
Chapel [Callahan et al. 2004]

Stencil optimization & DSLs
[Frigo & Strumpen 2005]
[Krishnamoorthy et al. 2007]
[Kamil et al. 2010]

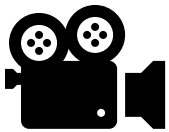
Mapping-based languages & DSLs
SPL/SPIRAL [Püschel et al. 2005]
Sequoia [Fatahalian et al. 2006]

Shading languages
RSL [Hanrahan & Lawson 1990]
Cg, HLSL [Mark et al. 2003; Blythe 2006]

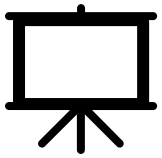
Image processing systems
[Shantzis 1994], [Levoy 1994]
PixelBender, CoreImage

Today's Class: Domain Specific Languages

I. Overview
II. Delite
III. Halide



Coming Attractions



- No more lectures!
- Wednesday 4/11: Exam topics posted on Piazza
- Friday 4/13: Project Milestone Report due midnight
- Wednesday 4/18: In-class Exam