Thorin, Partial Evaluation, and AnyDSL

Russel Arbore

AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries

Shallow Embedding of DSLs via Online Partial Evaluation

A Graph-Based Higher-Order Intermediate Representation

Thorin

Modern programming is functional*

- Almost all modern languages support some form of functional programming
- Manifests as higher order functions (HOFs)
- Implemented as closures
- Imperative languages must convert closures into normal functions and (possibly dynamically allocated) structs

```
void range(int a, int b,
           function<void(int)> f) {
    if (a < b) {
        f(a);
        range(a+1, b, f);
    }
}

void foo(int n) {
    range(0, n, [=] (int i) {
        use(i, n);
    });
}
```

(a) Original C++ program

```
struct closurebase {
    void (*f)(void* c, int i);
};

struct closure {
    closurebase base;
    int n;
};

void lambda(void* c, int i) {
    use(i, (closure* c)->n);
}

void range(int a, int b, void* c) {
    if (a < b) {
        ((closurebase*) c)->f(c, a);
        range(a+1, b, c);
    }
}

void foo(int n) {
    closure c = {{&lambda}, n};
    range(0, n, &c);
}
```

(b) Stylized imperative IR

Can we represent this in the IR?

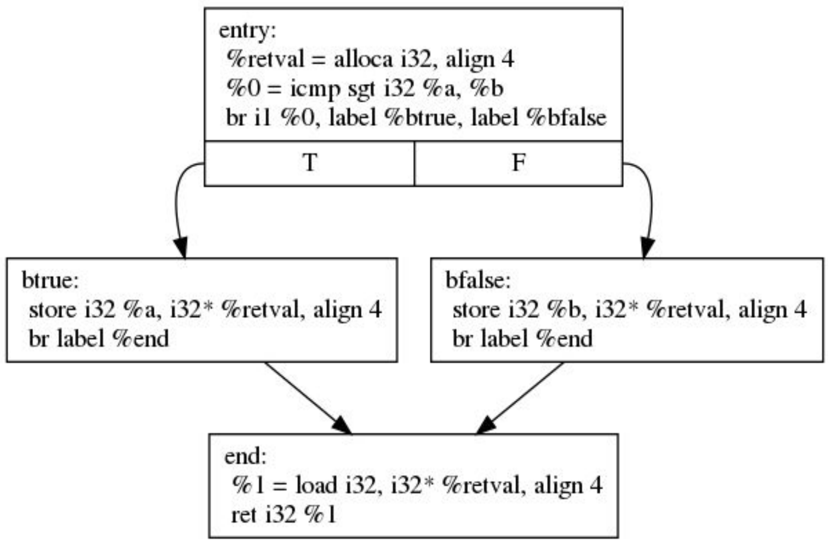
Imperative IRs

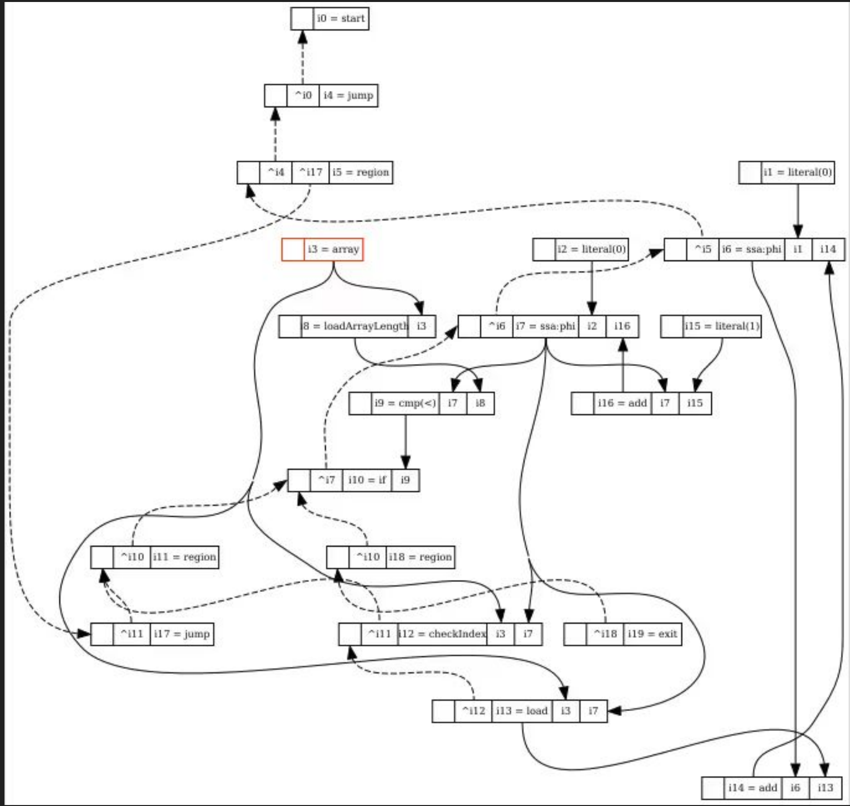
- What is typically used for imperative-first languages
- Can’t represent HOFs directly
- Closures must be lowered into function pointer + struct representation
    - This can sometimes be optimized through inlining and scalar replacement of aggregates
- Cannot reason about recursive HOFs

Functional IRs

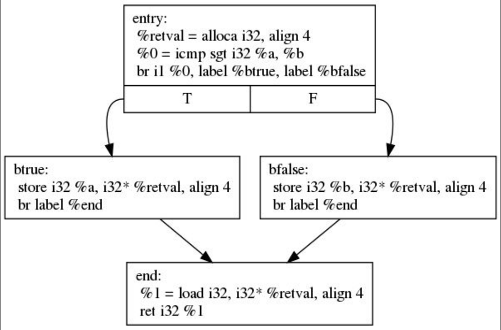
- Can reason about HOFs explicitly
- Not obvious how to lower a C++ or Rust into a functional IR
- Employs scope nesting to bind variables
    - Tricky to manipulate due to need to rename variables during transformations

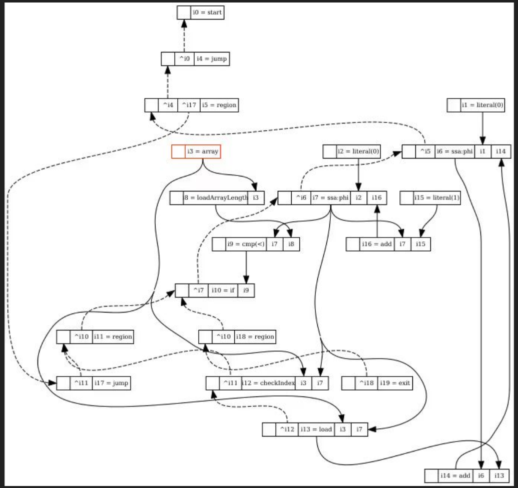
What about graph representations?





What about graph representations?

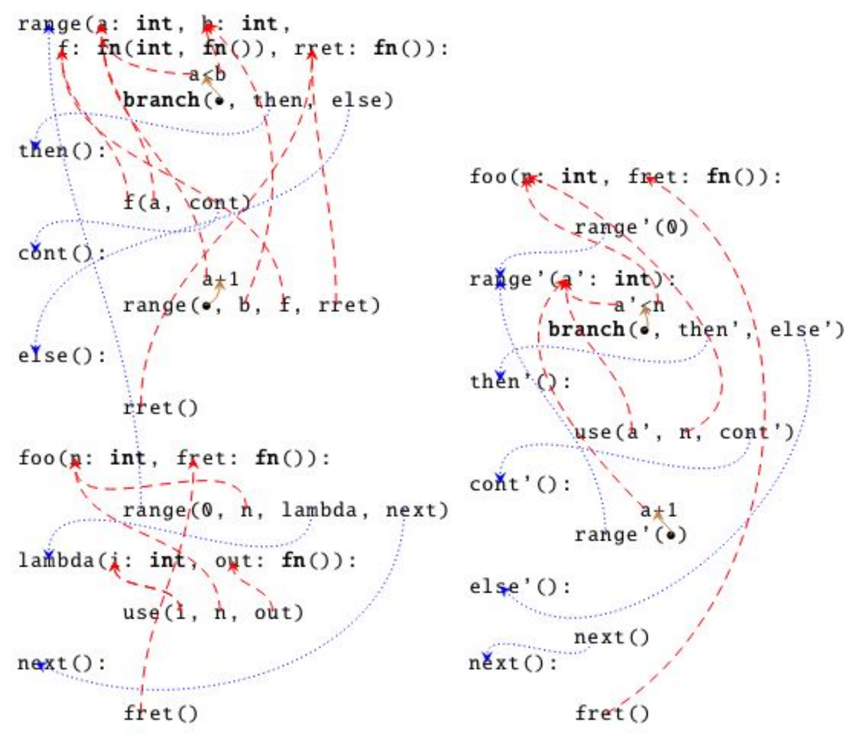




Neither of these are interprocedural!

Thorin IR

- Use CPS to represent all control flow (branches, function calls, longjmp)
- Implicit scope nesting - graph based
  - All “names” are graph edges
- in this paper we nevertheless use names in Thorin programs to make the presentation more accessible for humans. Names have no meaning otherwise.



(c) Thorin version

(d) Optimized Thorin version

SSA vs. CPS vs. Thorin

```
fn fac(n: int) → int {
  if (n ≤ 1)
    return 1;

  r: int = 1;
  for (i: int = 2; i ≤ n; ++i)
    r *= i;

  return r;
}
```

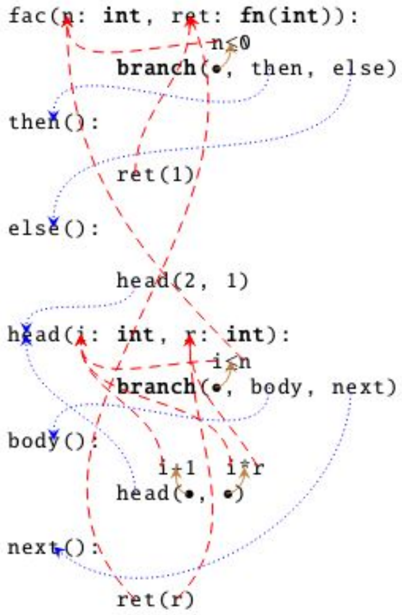
(a) Original program

```
fn fac(n: int) → int {
  branch(n ≤ 1, then, else)
then:
  return 1;
else:
  r0: int = 1;
  i0: int = 2;
head:
  r1 = φ(r0 [else], r2 [body]);
  i1 = φ(i0 [else], i2 [body]);
  branch(i1 ≤ n, body, next)
body:
  r2 = r1 * i1;
  i2 = i1 + 1;
  goto head;
next:
  return r1;
}
```

(b) SSA-form version

```
fac(n: int, ret: int → ⊥):
  let
    then():
      ret(1)
    else():
      letrec
        head(i: int, r: int):
          let
            body():
              head(i + 1, i * r)
            next():
              ret(r)
          in
            branch(i ≤ n, body, next)
      in
        head(2, 1)
  in
    branch(n ≤ 1, then, else)
```

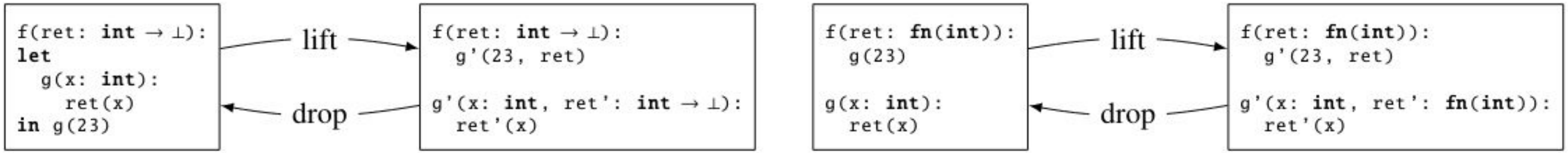
(c) Classic CPS version



(d) Thorin version (blockless)

Lambda Mangling

- In CPS, there is lambda lifting and dropping
  - Lifting removes a free parameter of a let function by adding an explicit argument
  - Dropping removes an explicit argument of a function by adding a free parameter from a caller
- In Thorin, only explicit modification is adding / removing an explicit parameter



(a) Classic CPS version (b) Thorin version

Lambda Mangling

```
f(x: int, y: int, ret: fn(int)):
  branch(..., calcx, calcy)
pow(a: int, b: int):
  branch(b = 0, then, else)
then():
  ret(1)
else():
  head(0, a)
head(i: int, r: int):
  branch(i < b, body, next)
body():
  head(i+1, r*a)
next():
  ret(r)
calcx():
  pow(x, 3)
calcy():
  pow(y, 3)
```

(a) The nested pow computes a^b.

```
f(x: int, y: int, ret: fn(int)):
  branch(..., calcx, calcy)
pow_d(a_d: int):
  head(0, a_d)
head(i: int, r: int):
  branch(i < 3, body, next)
body():
  head(i+1, r*a_d)
next():
  ret(r)
calcx():
  pow_d(x)
calcy():
  pow_d(y)
```

(b) Dropped pow_d computes a_d^3.

```
pow_l(a_l: int, b_l: int,
      ret_l: fn(int)):
  branch(b_l = 0, then, else)
then():
  ret_l(1)
else():
  head(0, a_l)
head(i: int, r: int):
  branch(i < b_l, body, next)
body():
  head(i+1, r*a_l)
next():
  ret_l(r)

f(x: int, y: int, ret: fn(int)):
  branch(..., calcx, calcy)
calcx():
  pow_l(x, 3, ret)
calcy():
  pow_l(y, 3, ret)
```

(c) Lifted pow_l doesn't use free variables.

```
pow_m(a_m: int,
      ret_m: fn(int)):
  head(0, a_m)
head(i: int, r: int):
  branch(i < 3, body, next)
body():
  head(i+1, r*a_m)
next():
  ret_m(r)

f(x: int, y: int, ret: fn(int)):
  branch(..., calcx, calcy)
calcx():
  pow_m(x, ret)
calcy():
  pow_m(y, ret)
```

(d) Dropped and lifted pow_m.

Lambda Mangling

```
foo(i: int, ret: fn(bool)):
  iseven(i, ret)

iseven(ei: int, eret: fn(bool)):
  branch(ei>0, ethen, eelse)
ethen():
  isodd(ei-1, eret)
eelse():
  eret(true)

isodd(oi: int, oret: fn(bool)):
  branch(oi>0, othen, oelse)
othen():
  iseven(oi-1, oret)
oelse():
  oret(false)
```

(a) Functions `iseven` and `isodd` are first-order recursive.

```
foo(i: int, ret: fn(bool)):
  iseven'(i)

iseven'(ei': int):
  branch(ei'>0, ethen', eelse')
ethen'():
  isodd'(ei'-1)
eelse'():
  ret(true)

isodd'(oi': int):
  branch(oi'>0, ethen, oelse)
othen'():
  iseven'(oi'-1)
oelse'():
  ret(false)
```

(b) The optimized version consists of a loop.

Code Generation

- Treat first order functions like basic blocks
- Treat second order functions as “returning” functions
- Lower as follows:
  - All “returning” functions become normal SSA functions
    - Calls to the second order parameter become returns
  - Each basic block like functions becomes a basic block
    - Each parameter turns into a phi node
  - Calls to “returning” functions become normal calls, calls to basic block functions become jumps
    - Value that would’ve been passed to “returning” function’s continuation becomes the return value

```
foo(i: int, ret: fn(bool)):
  iseven(i, ret)

iseven(ei: int, eret: fn(bool)):
  branch(ei>0, ethen, eelse)
ethen():
  isodd(ei-1, eret)
eelse():
  eret(true)

isodd(oi: int, oret: fn(bool)):
  branch(oi>0, othen, oelse)
othen():
  iseven(oi-1, oret)
oelse():
  oret(false)
```

```
foo(i: int, ret: fn(bool)):
  iseven'(i)

iseven'(ei': int):
  branch(ei'>0, ethen', eelse')
ethen'():
  isodd'(ei'-1)
eelse'():
  ret(true)

isodd'(oi': int):
  branch(oi'>0, ethen, oelse)
othen'():
  iseven'(oi'-1)
oelse'():
  ret(false)
```

```
define i1 @iseven(i32 %0) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %6, label %3

3:
  %4 = add i32 %0, -1
  %5 = call i1 @isodd(i32 %4)
  br label %6

6:
  %7 = phi i1 [ %5, %3 ], [ true, %1 ]
  ret i1 %7
}

define i1 @isodd(i32 %0) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %6, label %3

3:
  %4 = add i32 %0, -1
  %5 = call i1 @iseven(i32 %4)
  br label %6

6:
  %7 = phi i1 [ %5, %3 ], [ false, %1 ]
  ret i1 %7
}

define i1 @foo(i32 %0) {
  %2 = call i1 @iseven(i32 %0)
  ret i1 %2
}
```

```
define i1 @foo(i32 %0) {
  br label %2

2:
  %.01 = phi i32 [ %0, %1 ], [ %10, %9 ]
  %3 = icmp ne i32 %.01, 0
  br i1 %3, label %4, label %12

4:
  %5 = add i32 %.01, -1
  br label %7

7:
  %8 = icmp ne i32 %5, 0
  br i1 %8, label %9, label %12

9:
  %10 = add i32 %5, -1
  br label %2

12:
  %.0 = phi i1 [ false, %11 ], [ true, %6 ]
  ret i1 %.0
}
```

Partial Evaluation

What is partial evaluation?

- Evaluate static parts of a program, given some fixed static parameters
- Use PE results to specialize other parts of the program
- May diverge…
  - True divergence: the program actually doesn’t terminate
  - Hidden divergence: dynamically unreachable code is divergent, put PE may reach it
  - Induced divergence: the partial evaluator is “too greedy”

DSLs: deep vs. shallow embedding

Deep Embedding

- Compiler for DSL is written in host language
- Code for DSL is a data structure in host language
- Easy to implement
- Hard for programmer to reason about
- Think PyTorch / Tensorflow / (old?) Halide

Shallow Embedding

- DSL is truly part of the host language
- Better programming experience
- Cannot reason about DSL directly
- One either needs a partial evaluator in the host language, or one needs to significantly modify the host language compiler
- Think SYCL / Hetero-C++

Embedding DSLs in Impala

```
fn vectorize(L: int, a: int, b: int, body: fn(int) -> ()) -> ();
```

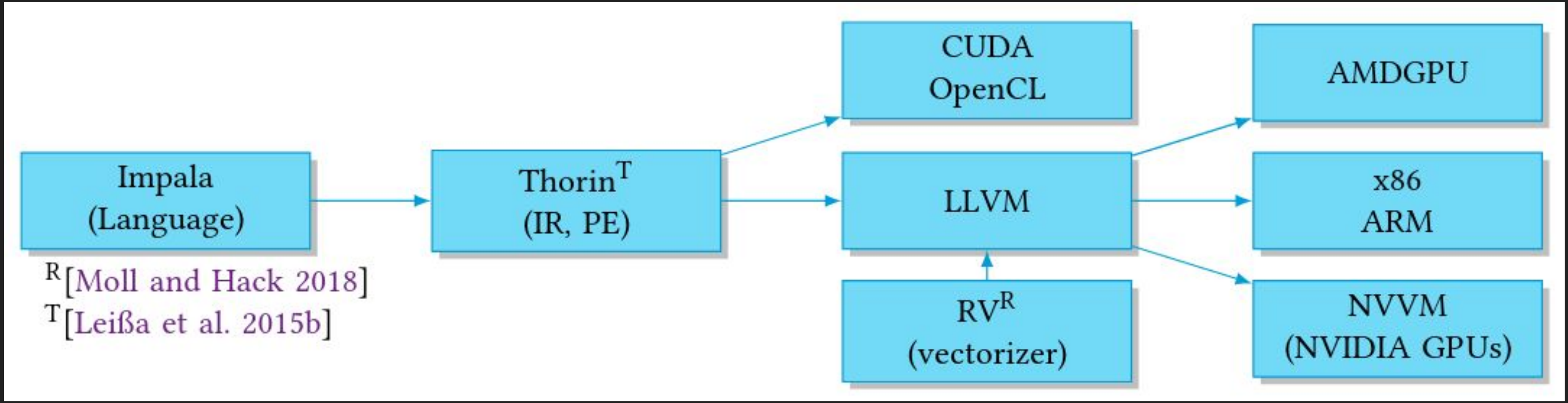
```
fn nvvm(grid: (int, int, int), block: (int, int, int),
        body: fn() -> ()) -> ();
```

```
fn apply_stencil(region: int, /*...*/,
        bh_lower: fn(int, int, int, fn(float)) -> int),
        bh_upper: fn(int, int, int, fn(float)) -> int)) -> float {
  // ...
  if region==0 { x = bh_lower(x, 0, arr.cols, return); } // left
  if region==2 { x = bh_upper(x, 0, arr.cols, return); } // right
  // ...
}

fn iterate(/*...*/) -> () {
  let limits = /* lower and upper limits for each region */;
  for y in $range(0, out.rows)
    for region in @range(0, 3)   // left, center, right
      let bounds = limits(region);
      for x in $range(bounds(0), bounds(1))
        @body(x, y, region);
}
```

```
fn iterate(out: Field, body: fn(int, int) -> ()) -> () {
  let unroll_factor = 4;
  let grid  = (out.cols, out.rows/unroll_factor, 1);
  let block = (128, 1, 1);
  nvvm(grid, block, || {
    let x = tid_x() + ntid_x()*ctaid_x();
    let y = tid_y() + ntid_y()*ctaid_y()*unroll_factor;
    for i in @range(0, unroll_factor)
      body(x, y + i * ntid_y());
  });
}
```

AnyDSL

Putting it all together



```
// user code
let blur_x = |x, y| (img.get(x-1, y) + img.get(x, y) + img.get(x+1, y)) / 3;
let blur_y = |x, y| ( blur_x(x, y-1) +  blur_x(x, y) +  blur_x(x, y+1)) / 3;

let seq = combine_xy(range, range);
let opt = tile(512, 32, vec(8), par(16));
let gpu = tile_cuda(32, 4);

compute(out_img_seq, seq, blur_y);
compute(out_img_opt, opt, blur_y);
compute(out_img_gpu, gpu, blur_y);

// implementation
type BinOp  = fn(i32, i32) -> i32;
type Loop1D = fn(i32, i32, fn(i32) -> ()) -> ();
type Loop2D = fn(i32, i32, i32, i32, fn(i32, i32) -> ()) -> ();

fn compute(out: Img, loop: Loop2D, op: BinOp) -> BinOp {
    for x, y in loop(0, 0, img.width, img.height) {
        out.set(x, y, op(x, y))
    }
    |x, y| out.get(x, y)
}

fn combine_xy(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |xa, ya, xb, yb, f|
        loop_y(ya, yb, |y|
            loop_x(xa, xb, |x| f(x, y)))
}

fn tile(xs: i32, ys: i32, loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |xa, ya, xb, yb, f|
        loop_y(0, (yb-ya)/ys, |ly|
            range(ly*ys+ya, (ly+1)*ys+ya, |ry|
                range(0, (xb-xa)/xs, |rx|
                    loop_x(rx*xs+xa, (rx+1)*xs+xa, |lx| f(lx, ry)))))
}

fn tile_cuda(xs: i32, ys: i32) -> Loop2D {
    |xa, ya, xb, yb, f| {
        let (grid, block)  = ((xb - xa, yb - ya, 1), (xs, ys, 1));
        cuda(grid, block, || f(cuda_gid_x(), cuda_gid_y()))
    }
}

fn @vec(vec_length:  i32) -> Loop1D { |a, b, f| vectorize(vec_length, a, b, f) }
fn @par(num_threads: i32) -> Loop1D { |a, b, f| parallel(num_threads, a, b, f) }
```

```
// user code
let blur_x = |x, y| (img.get(x-1, y) + img.get(x, y) + img.get(x+1, y)) / 3;
let blur_y = |x, y| ( blur_x(x, y-1) +  blur_x(x, y) +  blur_x(x, y+1)) / 3;

let seq = combine_xy(range, range);
let opt = tile(512, 32, vec(8), par(16));
let gpu = tile_cuda(32, 4);

compute(out_img_seq, seq, blur_y);
compute(out_img_opt, opt, blur_y);
compute(out_img_gpu, gpu, blur_y);

// implementation
type BinOp  = fn(i32, i32) -> i32;
type Loop1D = fn(i32, i32, fn(i32) -> ()) -> ();
type Loop2D = fn(i32, i32, i32, i32, fn(i32, i32) -> ()) -> ();

fn compute(out: Img, loop: Loop2D, op: BinOp) -> BinOp {
    for x, y in loop(0, 0, img.width, img.height) {
        out.set(x, y, op(x, y))
    }
    |x, y| out.get(x, y)
}

fn combine_xy(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |xa, ya, xb, yb, f|
        loop_y(ya, yb, |y|
            loop_x(xa, xb, |x| f(x, y)))
}

fn tile(xs: i32, ys: i32, loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |xa, ya, xb, yb, f|
        loop_y(0, (yb-ya)/ys, |ly|
            range(ly*ys+ya, (ly+1)*ys+ya, |ry|
                range(0, (xb-xa)/xs, |rx|
                    loop_x(rx*xs+xa, (rx+1)*xs+xa, |lx| f(lx, ry)))))
}

fn tile_cuda(xs: i32, ys: i32) -> Loop2D {
    |xa, ya, xb, yb, f| {
        let (grid, block)  = ((xb - xa, yb - ya, 1), (xs, ys, 1));
        cuda(grid, block, || f(cuda_gid_x(), cuda_gid_y()))
    }
}

fn @vec(vec_length:  i32) -> Loop1D { |a, b, f| vectorize(vec_length, a, b, f) }
fn @par(num_threads: i32) -> Loop1D { |a, b, f| parallel(num_threads, a, b, f) }
```

Implement DSL as a library, not a new compiler

Preventing divergence

- Program author must annotate where specialization can occur
- @ sign denotes a set of *filters*
- A set of filters can be applied to an entire function, or individually per parameter
- ?n evaluates to true if n is constant
- $n yields n, but isn't constant
- Can contain arbitrary expression (n < 5)
- No @ is sugar for @(false), just an @ is sugar for @(true)

```
fn @(?n) pow(x: i32, n: i32) -> i32 {
    if n == 0 {
        1
    } else if n % 2 == 0 {
        let r = pow(x, n/2);
        r * r
    } else {
        x * pow(x, n-1)
    }
}
```

Accelerator support

```
for i in parallel(num_threads, a, b) { array(i) = f(x); }
parallel(num_threads, a, b, |i: i32| { array(i) = f(x); }); // desugared variant
```

```
void anydsl_parallel_for(int a, int b, void* args, void (*fun)(void*, int)) {
    tbb::parallel_for(tbb::blocked_range<int>(a, b), [=] (auto& range) {
        for (int i = range.begin(); i < range.end(); ++i) fun(args, i);
    });
}
```

```
    vectorize(vec_length, a, b, align, |i: i32| array(i) = f(x));
--> vectorize(vec_length, a, b, align, |i: i32, array: &mut[i32], x: i32| array(i) = f(x));
```

```
for i in range_step(a, b, vec_length) { B_simd(i, array, x); }
```

```
    cuda(device, grid, block, |i: i32| array(i) = f(x));
--> cuda(device, grid, block, |i: i32, array: &mut[i32], x: i32| array(i) = f(x));
```

```
__device__ int f(int x) { /* ... */ }
__global__ void lambda(int* array, int x) { /* ... */ }
```

```
void anydsl_launch_kernel(DevId device, const char* file, const char* kernel, const uint* grid,
  const uint* block, void** args, const uint* sizes, const Type* types, uint num);
```

Is it fast?

	BVH4						BVH8					
	Primary		AO		Diffuse		Primary		AO		Diffuse	
Scene	Ours	Embree	Ours	Embree	Ours	Embree	Ours	Embree	Ours	Embree	Ours	Embree
Sponza	34.73 (-4%)	36.35	76.34 (+8%)	70.66	9.78 (-12%)	11.07	34.84 (-4%)	36.40	76.73 (+13%)	67.81	11.46 (-10%)	12.74
Crown	102.51 (+5%)	97.86	40.28 (-9%)	44.26	19.48 (-12%)	22.20	95.48 (+6%)	89.92	42.12 (-5%)	44.25	21.04 (-9%)	23.16
San-Miguel	22.06 (-4%)	23.04	13.82 (-13%)	15.91	6.46 (-12%)	7.33	18.74 (-2%)	19.13	14.77 (-10%)	16.32	6.98 (-8%)	7.62
Powerplant	49.34 (-3%)	50.63	102.89 (+8%)	95.42	11.86 (-15%)	13.88	43.02 (-4%)	44.82	98.10 (+11%)	88.04	13.29 (-9%)	14.66

(a) CPU: Skylake i7 6700K

	BVH4					
	Primary		AO		Diffuse	
Scene	Vec.	Scalar	Vec.	Scalar	Vec.	Scalar
Sponza	2.75 (+100%)	1.38	5.36 (+101%)	2.66	0.95 (-5%)	1.00
Crown	9.82 (+69%)	5.80	3.65 (+21%)	3.01	1.87 (-2%)	1.91
San-Miguel	2.07 (+94%)	1.07	1.49 (+14%)	1.31	0.72 (-8%)	0.78
Powerplant	4.44 (+71%)	2.59	8.19 (+102%)	4.06	1.09 (-11%)	1.23

(b) CPU: Cortex-A53

BVH2					
Primary		AO		Diffuse	
Ours	Aila et al.	Ours	Aila et al.	Ours	Aila et al.
330.41 (-2%)	336.50	884.33 (-2%)	899.62	123.46 (-9%)	135.80
695.41 (-11%)	778.52	315.09 (-14%)	366.28	133.85 (-19%)	165.71
181.67 (-5%)	190.78	132.85 (-7%)	142.13	54.06 (-15%)	63.40
465.42 (-12%)	528.21	998.02 (-4%)	1040.93	138.57 (-11%)	155.57

(c) GPU: GeForce GTX Titan X (Maxwell)

Fig. 7. Performance of our traversal kernels on different architectures, in **Mrays/s** (mega rays per second, higher is better). Speed-ups (slow-downs) with respect to the reference are indicated in parentheses. On CPUs (GPUs), we perform 10 (100) warmup iterations and report the average of 50 (500) runs. *Primary* rays start from the camera, *AO* rays compute Ambient Occlusion, and *Diffuse* rays compute purely diffuse reflections.

	CPU		GPU	
	Ours	Halide	Ours	Halide
Blur	1.99 (+12%)	1.77	14.22 (+7%)	13.31
Harris Corner	1.14 (+37%)	0.83	8.39 (+44%)	5.83

Fig. 6. Median pixel throughput in **Gpixels/s** (giga pixels per second, higher is better) for the blur filter of Figure 2 on `i32` pixel type and the Harris corner detector for `f32` pixel type, both for an image of 4096 × 4096 pixels. CPU execution on a Skylake i7 6700K and GPU execution on a GeForce GTX 970. The execution time on the GPU for Halide are the average numbers reported by nvprof.

		CPU			GPU	
		Ours	SeqAn	Parasail	Ours	NVBIO
Score only	linear	11.9 (-3%)	12.3	n/a	148 (+10%)	135
	affine	10.8 (-8%)	11.8	11.0	133 (-3%)	136
Traceback	linear	8.1 (-9%)	8.9	n/a	112 (+5%)	107
	affine	7.7 (+11%)	n/a	6.9	106 (+4%)	102

Fig. 9. Median runtime performance in **GCUPS** (giga cell updates per second, higher is better) for aligning pairs of six DNA sequences with 4.4 to 50 million characters. CPU execution on two Xeon E5-2683v4 CPUs with 32 threads and GPU execution on a Titan Xp.

Related work and thoughts…

34.1.2. Compile-Time Variables

In Zig, the programmer can label variables as `comptime`. This guarantees to the compiler that every load and store of the variable is performed at compile-time. Any violation of this results in a compile error.

This combined with the fact that we can `inline` loops allows us to write a function which is partially evaluated at compile-time and partially at run-time.

For example:

test_comptime_evaluation.zig

```zig
const expect = @import("std").testing.expect;

const CmdFn = struct {
    name: []const u8,
    func: fn(i32) i32,
};

const cmd_fns = [_]CmdFn{
    CmdFn {.name = "one", .func = one},
    CmdFn {.name = "two", .func = two},
    CmdFn {.name = "three", .func = three},
};
fn one(value: i32) i32 { return value + 1; }
fn two(value: i32) i32 { return value + 2; }
fn three(value: i32) i32 { return value + 3; }

fn performFn(comptime prefix_char: u8, start_value: i32) i32 {
    var result: i32 = start_value;
    comptime var i = 0;
    inline while (i < cmd_fns.len) : (i += 1) {
        if (cmd_fns[i].name[0] == prefix_char) {
            result = cmd_fns[i].func(result);
        }
    }
    return result;
}

test "perform fn" {
    try expect(performFn('t', 1) == 6);
    try expect(performFn('o', 0) == 1);
    try expect(performFn('w', 99) == 99);
}
```

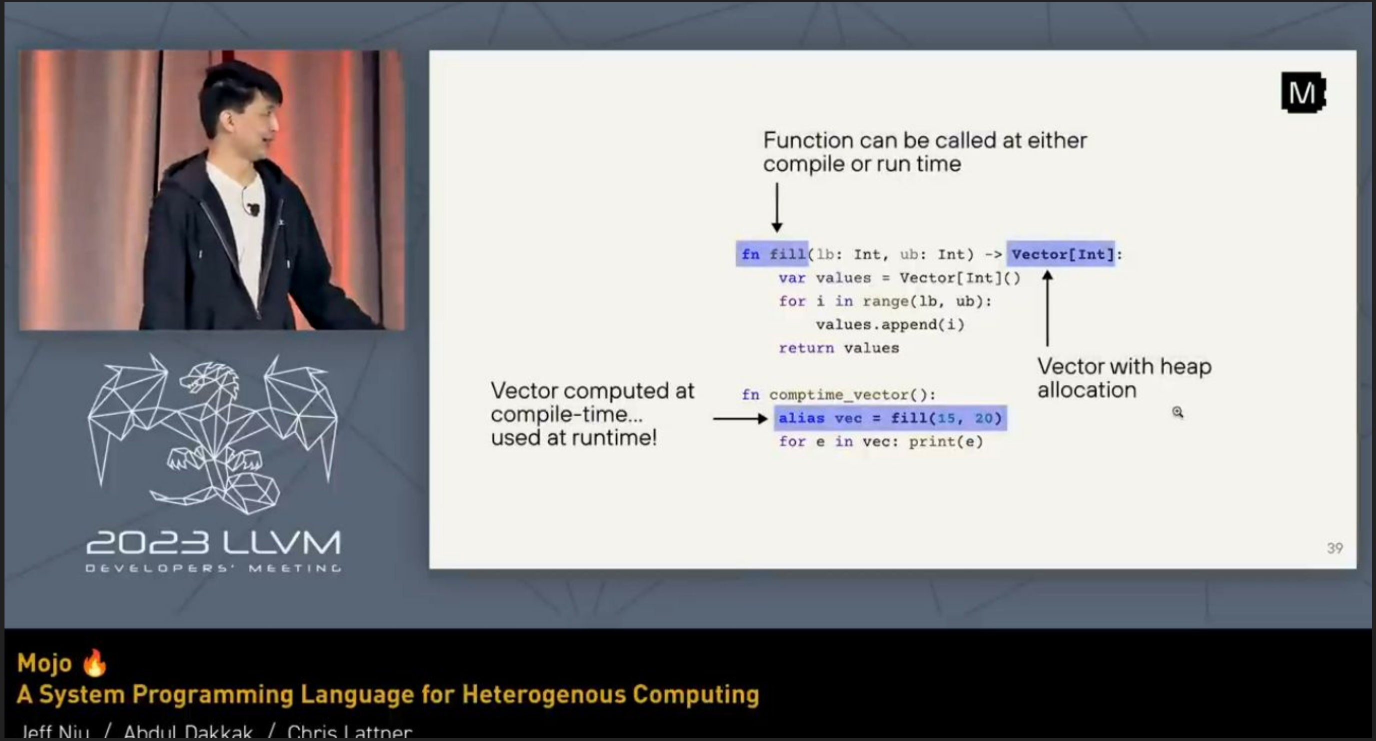
38.113. @Vector

```
@Vector(len: comptime_int, Element: type) type
```

test_vector.zig

```zig
const std = @import("std");
const expectEqual = std.testing.expectEqual;

test "Basic vector usage" {
    // Vectors have a compile-time known length and base type.
    const a = @Vector(4, i32){ 1, 2, 3, 4 };
    const b = @Vector(4, i32){ 5, 6, 7, 8 };

    // Math operations take place element-wise.
    const c = a + b;

    // Individual vector elements can be accessed using array indexing syntax.
    try expectEqual(6, c[0]);
    try expectEqual(8, c[1]);
    try expectEqual(10, c[2]);
    try expectEqual(12, c[3]);
}

test "Conversion between vectors, arrays, and slices" {
    // Vectors and fixed-length arrays can be automatically assigned back and forth
    const arr1: [4]f32 = [_]f32{ 1.1, 3.2, 4.5, 5.6 };
    const vec: @Vector(4, f32) = arr1;
    const arr2: [4]f32 = vec;
    try expectEqual(arr1, arr2);

    // You can also assign from a slice with comptime-known length to a vector using .*
    const vec2: @Vector(2, f32) = arr1[1..3].*;

    const slice: []const f32 = &arr1;
    var offset: u32 = 1; // var to make it runtime-known
    _ = &offset; // suppress 'var is never mutated' error
    // To extract a comptime-known length from a runtime-known offset,
    // first extract a new slice from the starting offset, then an array of
    // comptime-known length
    const vec3: @Vector(2, f32) = slice[offset..][0..2].*;
    try expectEqual(slice[offset], vec2[0]);
    try expectEqual(slice[offset + 1], vec2[1]);
    try expectEqual(vec2, vec3);
}
```

Mojo🔥


Function can be called at either compile or run time
fn fill(lb: Int, ub: Int) -> Vector[Int]:
    var values = Vector[Int]()
    for i in range(lb, ub):
        values.append(i)
    return values
Vector with heap allocation
Vector computed at compile-time... used at runtime!
fn comptime_vector():
    alias vec = fill(15, 20)
    for e in vec: print(e)
39
2023 LLVM
DEVELOPERS' MEETING
Mojo 🔥
A System Programming Language for Heterogenous Computing
Jeff Niu / Abdul Dakkak / Chris Lattner

Metaprogramming vs. Schedules

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

Accelerator support?

- GPUs are still generally programmable
- Ray tracing cores?
- DL accelerators?
- HDC accelerators?
- Dynamic scheduling?