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

Richard Membarth, Arsène Pérard-Gayot, Stefan Lemme, Manuela Schuler, Puya Amiri, Philipp Slusallek (Visual Computing)

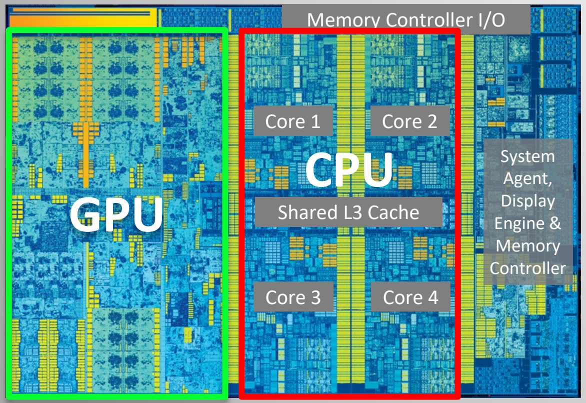
Roland Leißa, Simon Moll, Sebastian Hack (Compiler)

Intel Visual Computing Institute (IVCI) at Saarland University

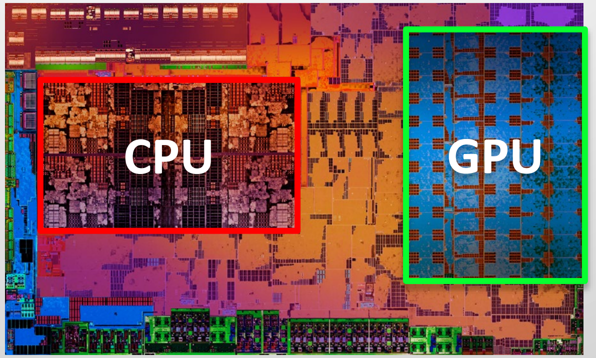
German Research Center for Artificial Intelligence (DFKI)

Many-Core Dilemma

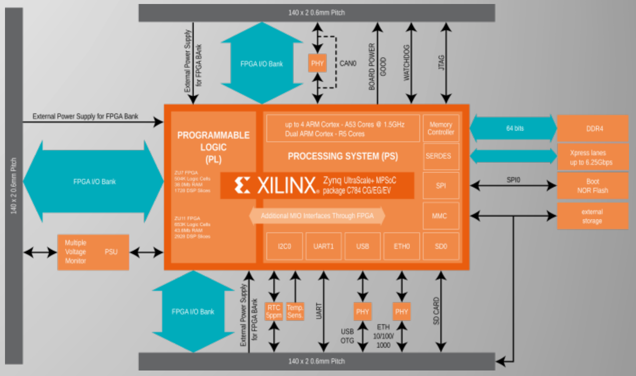
- Many-core hardware is everywhere – but programming it is still hard



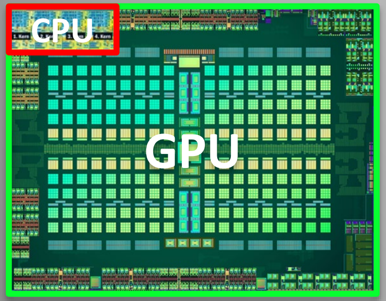
Intel Skylake (1.8B transistors)



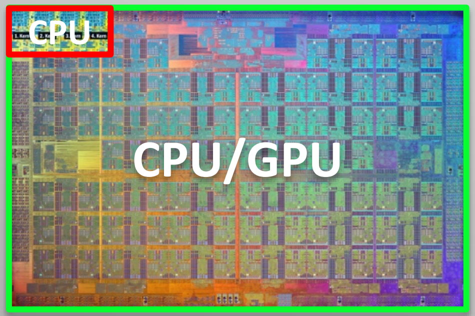
AMD Zen + Vega (4.9B transistors)



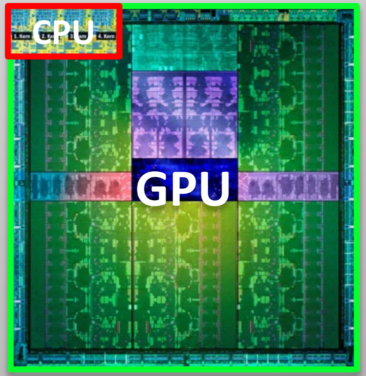
Xilinx Zynq



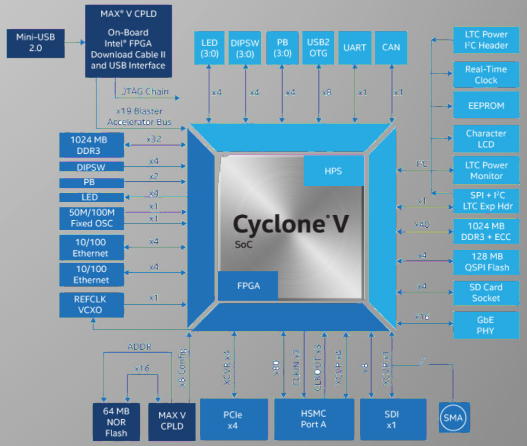
AMD Polaris
(~5.7B transistors)



Intel Knights Landing
(~8B transistors)



NVIDIA Kepler
(~7B transistors)



Intel / Altera Cyclone

intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Still State-of-the-Art …

Math

Pseudo-Code

C sequential

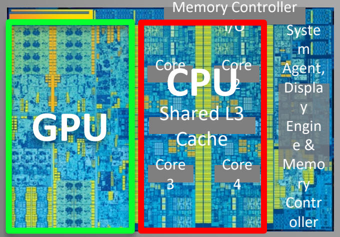
X1 | X2 | X3 | X4

abstract
maintainable
readable
portable
“slow”

manual

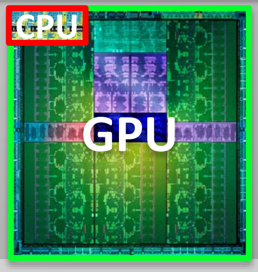
automated

concrete
fast



$X_i \subseteq$ { C, OpenMP, OpenACC, CUDA, OpenCL, OpenCL4X, OpenCL4Y, … }

…

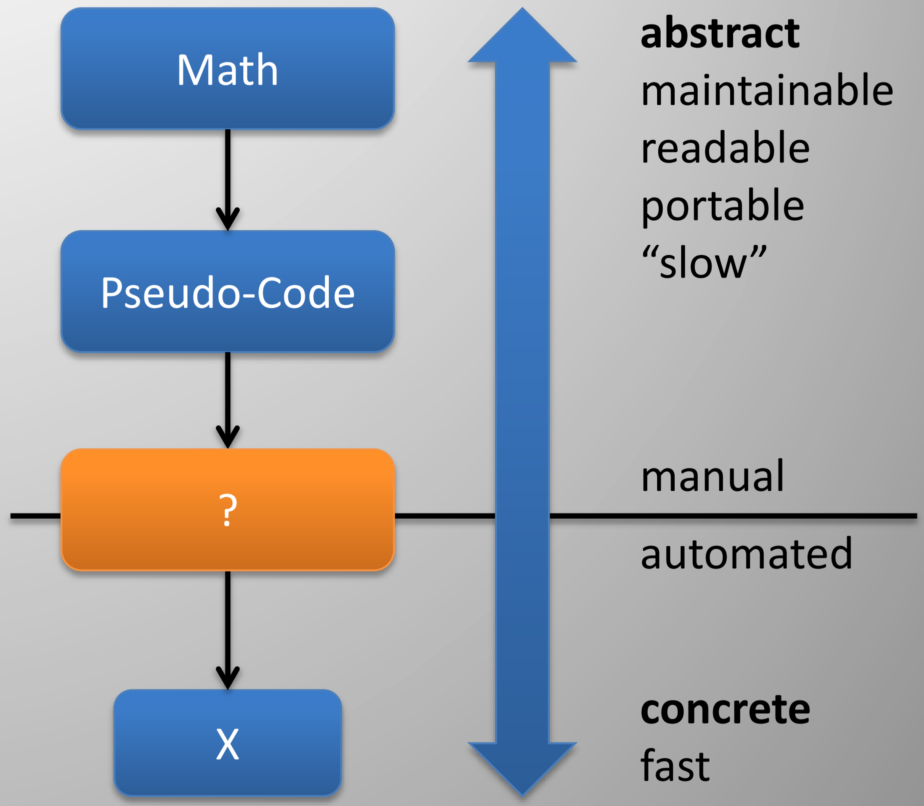


intel
VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

What can we do?

Challenges: **Productivity**, **portability**, and **performance**.

- Manual tuning
  rewrite code yourself
- Annotations
  use the compiler to rewrite code
- Program generation
  use a script to write code
- Meta programming
  write program to rewrite program
- Domain-specific languages
  write compiler to rewrite program



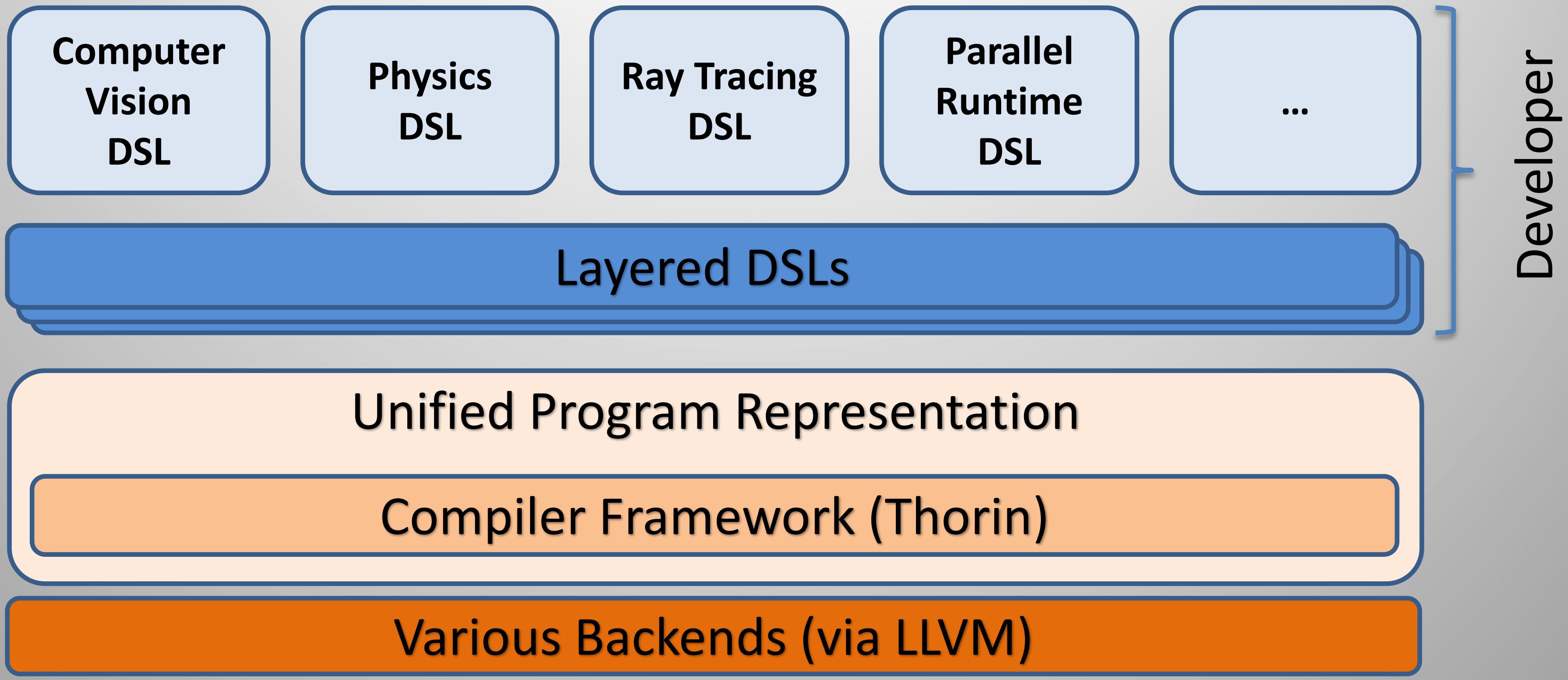
The Vision

- Single high-level representation of our algorithms
- Simple transformations to wide range of target hardware architectures

- First step: RTfact [HPG'08]
  - Use of C++ Template Metaprogramming
  - Great performance (-10%) – but largely unusable due to template syntax

- AnyDSL: New compiler technology, enabling arbitrary Domain-Specific *Libraries* (DSLs)
  - High-level algorithms + HW mapping of used abstractions + cross-layer specialization
  - **Computer Vision:** 10x shorter code, 25-50% *faster* than OpenCV on GPU & CPU
  - **Ray Tracing:** First cross-platform algorithm, beating best code on CPUs & GPUs

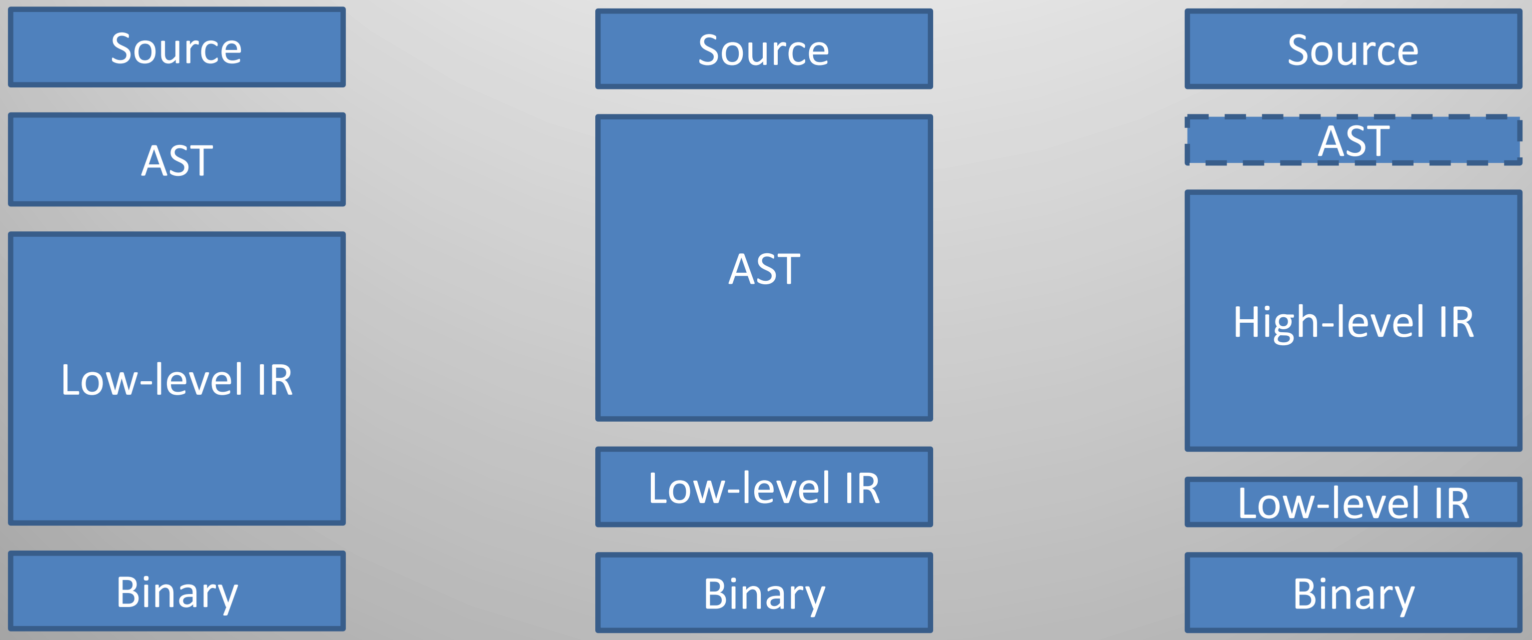
AnyDSL: Overview



intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
max planck institut informatik
Max Planck Institute for Software Systems

High-Level Program Representation

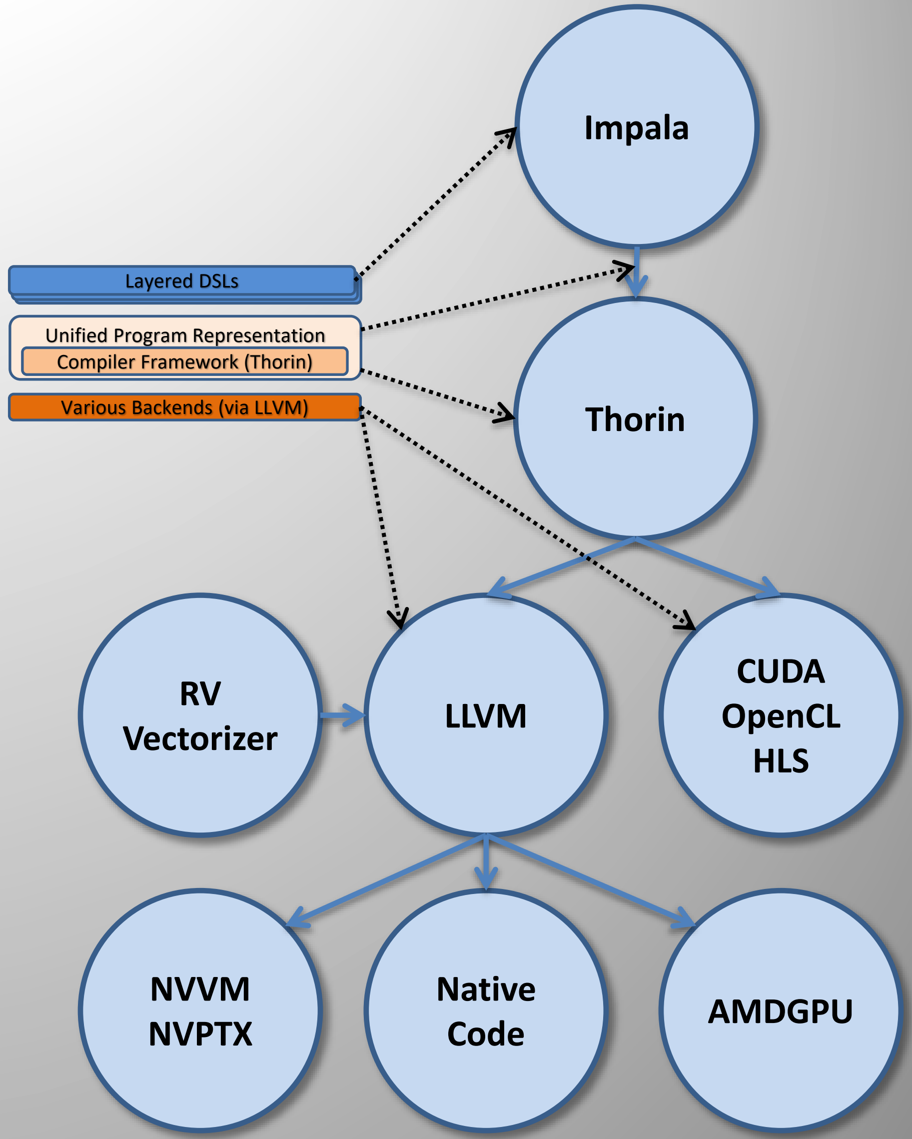
- Uses functional Continuation Passing Style (CPS) and graph-based structure
  - All language constructs as higher-order functions
  - Structure well suited for transformations using “lambda mangling”



Traditional Compilers „Functional“ Compilers AnyDSL

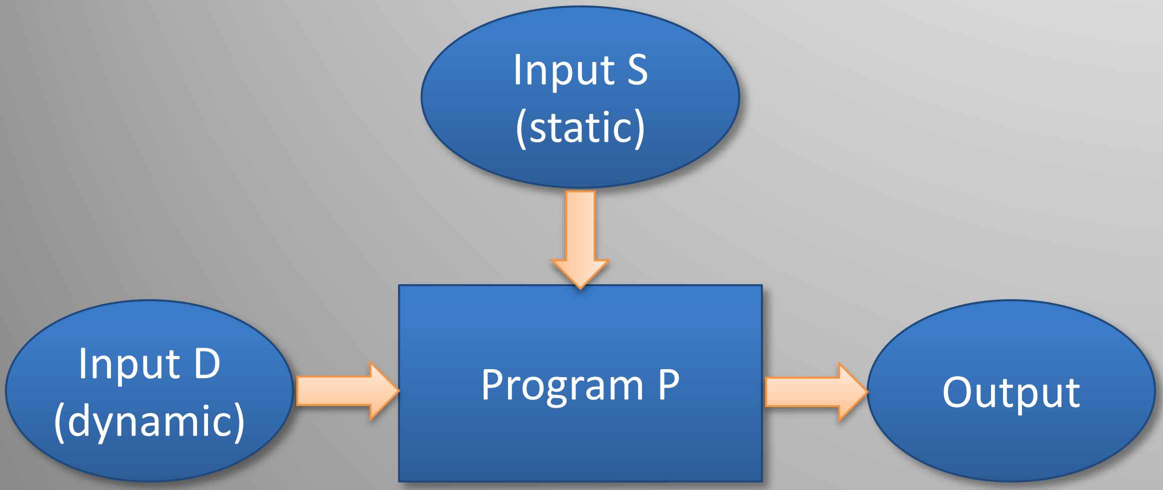
Compiler Framework

- Impala language (Rust dialect)
  - Functional & imperative language
- Thorin compiler [GPCE'15, OOPSLA'18]
  - Higher-order functional IR [CGO'15]
    - Special optimization passes
    - No overhead during runtime
- Region Vectorizer [PLDI'18]
- LLVM-based back ends
  - Full compiler optimization passes
  - Multi-target code generation
    - NVVM/NVPTX, AMDGPU
    - CPUs, GPUs, FPGAs, SX-Aurora, …

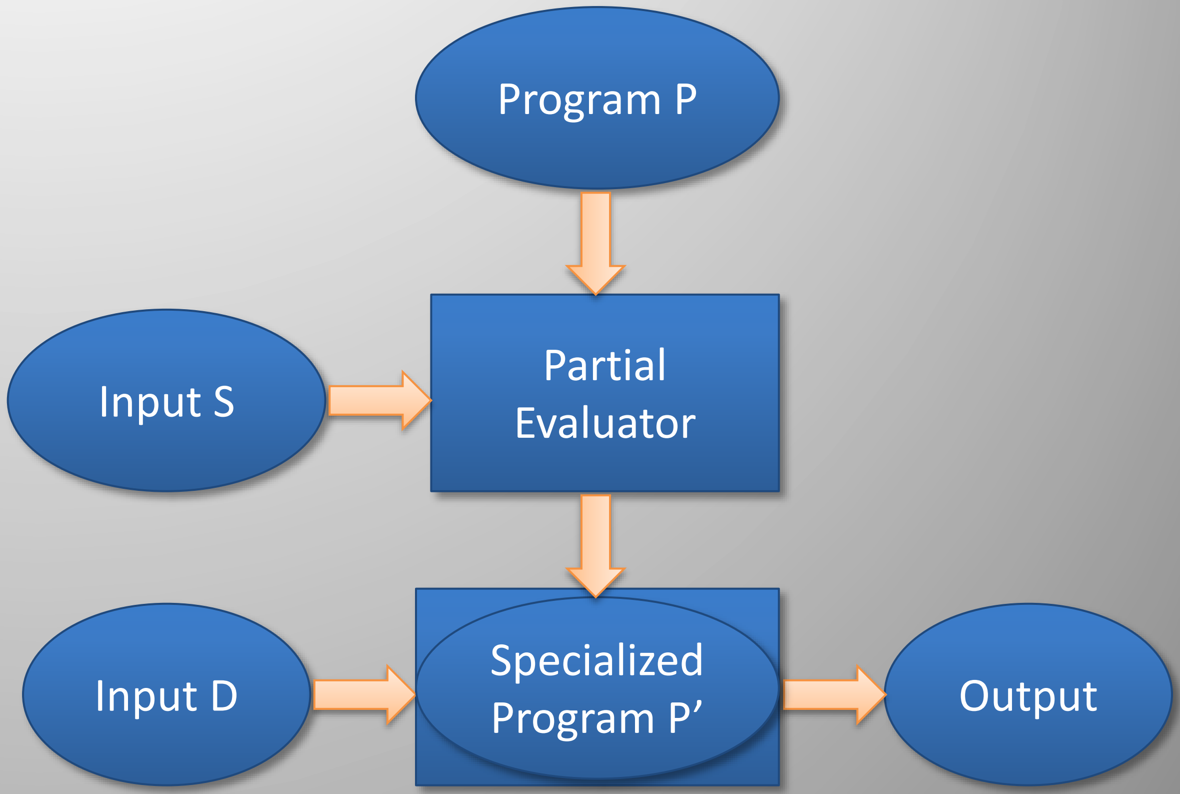


AnyDSL Key Feature: Partial Evaluation (in a Nutshell)

- Normal program execution



- Execution with program specialization
  - PE as part of normal compilation process!!



Impala: A Base Language for DSL Embedding

- Impala is an imperative & functional language
  - A dialect of Rust (https://rust-lang.org)
  - Specialization when instantiating @-annotated functions [OOPSLA'18]
  - Partial evaluation executes all possible instructions at compile time

```
fn @(?n) dot(n: int,
             u: &[float],
             v: &[float]
             ) -> float {
  let mut sum = 0.0f;

  for i in unroll(0, n) {
    sum += u(i)*v(i);
  }

  sum
}

// specialization at call-site
result = dot(3, a, b);
```

```
// specialized code for dot-call
result = 0;
result += a(0)*b(0);
result += a(1)*b(1);
result += a(2)*b(2);
```


intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpi max planck institut informatik
Max Planck Institute for Software Systems

Case Study: Image Processing

[GPCE’15, OOPSLA’18]

Stincilla – A DSL for Stencil Codes

https://github.com/AnyDSL/stincilla

intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Sample DSL: Stencil Codes in Impala

- Application developer: Simply wants to use a DSL
  - Example: Image processing, specifically Gaussian blur
  - Using OpenCV as reference

```
fn main() -> () {
  let img = read_image("lena.pgm");
  let result = gaussian_blur(img);
  show_image(result);
}
```


Intel Visual Computing Institute
Universität des Saarlandes
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Sample DSL: Stencil Codes in Impala

- Higher level domain-specific code: DSL implementation
  - Gaussian blur implementation using generic apply_convolution
  - iterate function iterates over image (provided by machine expert)

```
fn @gaussian_blur(img: Img) -> Img {
  let mut out = Img { data:   ~[img.width*img.height:float],
                      width:  img.width,
                      height: img.height };
  let filter = [[0.057118f, 0.124758f, 0.057118f],
                [0.124758f, 0.272496f, 0.124758f],
                [0.057118f, 0.124758f, 0.057118f]];

  for x, y in iterate(out) {
    out.data(x, y) = apply_convolution(x, y, img, filter);
  }

  out
}
```


intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Sample DSL: Stencil Codes in Impala

- Higher level domain-specific code: DSL implementation
  - for syntax: syntactic sugar for lambda function as last argument

```
fn @gaussian_blur(img: Img) -> Img {
  let mut out = Img { data:   ~[img.width*img.height:float],
                      width:  img.width,
                      height: img.height };
  let filter = [[0.057118f, 0.124758f, 0.057118f],
                [0.124758f, 0.272496f, 0.124758f],
                [0.057118f, 0.124758f, 0.057118f]];

  iterate(out, |x, y| -> () {
    out.data(x, y) = apply_convolution(x, y, img, filter);
  });

  out
}
```


intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
max planck institut informatik
Max Planck Institute for Software Systems

Sample DSL: Stencil Codes in Impala

- Domain-specific code: DSL implementation for image processing
  - Generic function that applies a given stencil to a single pixel
  - Partial evaluation
    - Unrolls stencil
    - Propagates constants
    - Inlines function calls

```
fn @apply_convolution(x: int, y: int,
                      img: Img,
                      filter: [float]
                     ) -> float {
  let mut sum = 0.0f;
  let half = filter.size / 2;

  for j in unroll(-half, half+1) {
    for i in unroll(-half, half+1) {
      sum += img.data(x+i, y+j) * filter(i, j);
    }
  }

  sum
}
```


intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpi max planck institut informatik
Max Planck Institute for Software Systems

Mapping to Target Hardware: CPU

- Scheduling & mapping provided by machine expert
  - Simple sequential code on a CPU
  - body gets inlined through specialization at higher level

```
fn @iterate(img: Img, body: fn(int, int) -> ()) -> () {
  for y in range(0, img.height) {
    for x in range(0, img.width) {
      body(x, y);
    }
  }
}
```


intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Mapping to Target Hardware: CPU with Optimization

- Scheduling & mapping provided by machine expert
  - CPU code using parallelization and vectorization (e.g. AVX)
  - parallel is provided by the compiler, maps to TBB or C++11 threads
  - vectorize is provided by the compiler, uses region vectorization

```
fn @iterate(img: Img, body: fn(int, int) -> ()) -> () {
  let thread_number = 4;
  let vector_length = 8;
  for y in parallel(thread_number, 0, img.height) {
    for x in range_step(0, img.width, vector_length) {
      for lane in vectorize(vector_length) {
        body(x + lane, y);
      }
    }
  }
}
```


intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
max planck institut informatik
Max Planck Institute for Software Systems

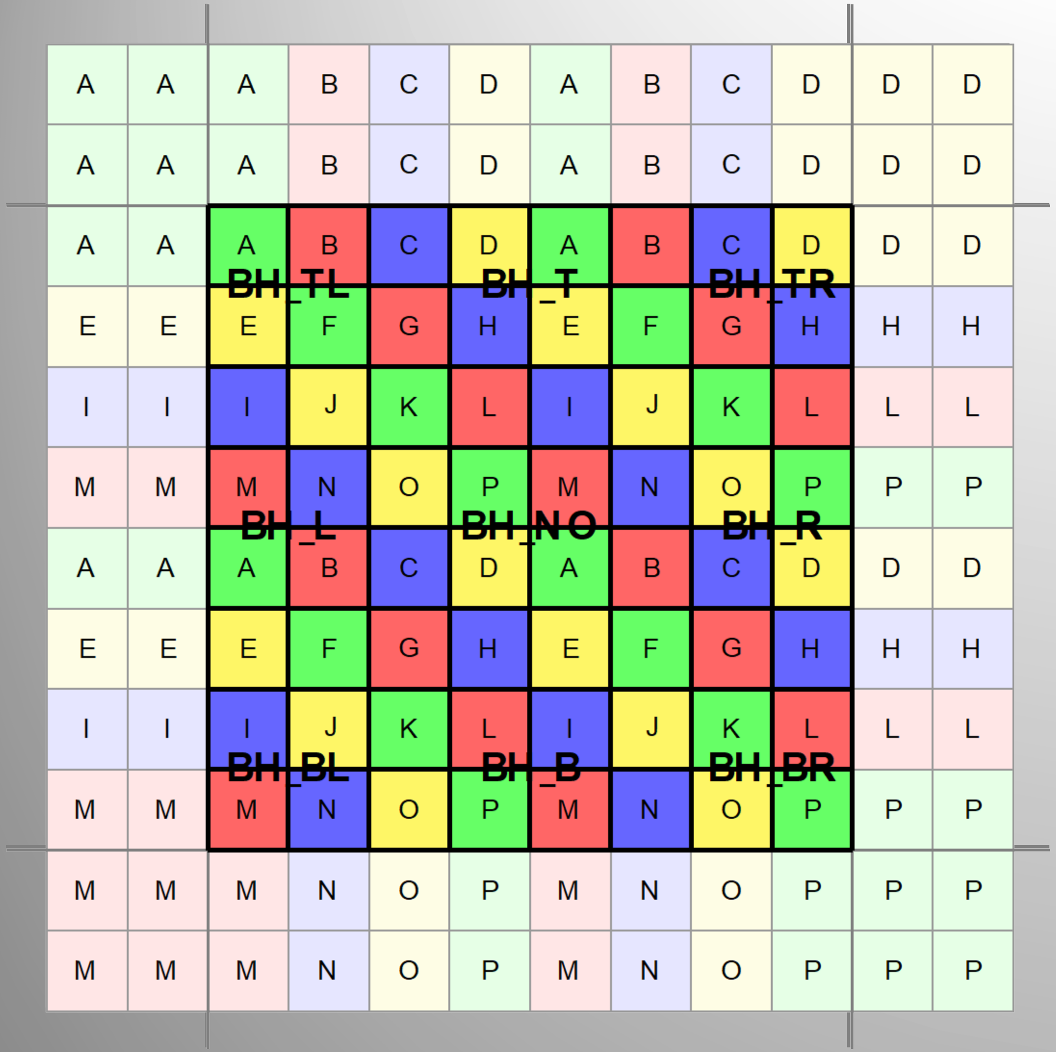
Mapping to Target Hardware: GPU

- Scheduling & mapping provided by machine expert
  - Exposed NVVM (CUDA) code generation
  - Last argument of nvvm is function we generate NVVM code for

```
fn @iterate(img: Img, body: fn(int, int) -> ()) -> () {
  let grid  = (img.width, img.height, 1);
  let block = (32, 4, 1);

  with nvvm(grid, block) {
    let x = nvvm_tid_x() + nvvm_ntid_x() * nvvm_ctaid_x();
    let y = nvvm_tid_y() + nvvm_ntid_y() * nvvm_ctaid_y();
    body(x, y);
  }
}
```


intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpi max planck institut informatik
Max Planck Institute for Software Systems

Exploiting Boundary Handling (1)



- Boundary handling
  - Evaluated for all points
  - Unnecessary evaluation of conditionals
- Specialized variants for different regions
- Automatic generation of variants
  → Partial evaluation

Exploiting Boundary Handling (2)

- Specialized implementation
  - Wrap memory access to image in an access() function
    - Distinction of variant via <u>region</u> variable (here only in horizontally)
  - Specialization discards unnecessary checks

A	A	A	B	C	D	A	B	C	D	D	D

```
fn @access(mut x: int, y: int,
           img: Img,
           region,
           bh_lower: fn(int, int) -> int,
           bh_upper: fn(int, int) -> int,
          ) -> float {
  if region == left  { x = bh_lower(x, 0); }
  if region == right { x = bh_upper(x, img.width); }
  img(x, y)
}
```

Exploiting Boundary Handling: CPU & AVX

- Specialized implementation
  - outer_loop maps to parallel and inner_loop calls either range (CPU) or vectorize (AVX)
  - unroll triggers image region specialization
  - Speedup over OpenCV: 40% (Intel CPU, vectorized)

```
fn @iterate(img: Img, body: fn(int, int, int) -> ()) -> () {
  let offset = filter.size / 2;
  //        left    right                center
  let L = [0,       img.width - offset, offset];
  let U = [offset, img.width,            img.width - offset];

  for region in unroll(0, 3) {
    for y in outer_loop(0, img.height) {
      for x in inner_loop(L(region), U(region)) {
        ...
        body(x, y, region);
      }
    }
  }
}
```


intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Exploiting Boundary Handling: GPU

- Specialized implementation
  - unroll triggers image region specialization
  - Generates multiple GPU kernels for each image region
  - Speedup over OpenCV: 25% (Intel GPU), 50% (AMD GPU), 45% (NVIDIA GPU)

```
fn @iterate(img: Img, body: fn(int, int, int) -> ()) -> () {
  let offset = filter.size / 2;
  //        left    right                 center
  let L = [0,       img.width - offset, offset];
  let U = [offset, img.width,           img.width - offset];

  for region in unroll(0, 3) {
    let grid = (U(region) - L(region), img.height, 1);
    with nvvm(grid, (128, 1, 1)) {
      ...
      body(L(region) + x, y, region);
    }
  }
}
```


intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Mapping to Target Hardware: FPGA (WIP)

- Scheduling & mapping provided by machine expert
  - Exposed AOCL code generation via opencl
  - Exposed VHLS code generation via hls
  - Mapping for simple point operators

```
fn @iterate(img: Img, body: fn(int, int) -> ()) -> () {
  with opencl((1, 1, 1), (1, 1, 1)) {
    for y in range(0, img.height) {
      for x in range(0, img.width) {
        body(x, y);
      }
    }
  }
}
```


intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpi max planck institut informatik
Max Planck Institute for Software Systems

Other Domains [OOPSLA'18]

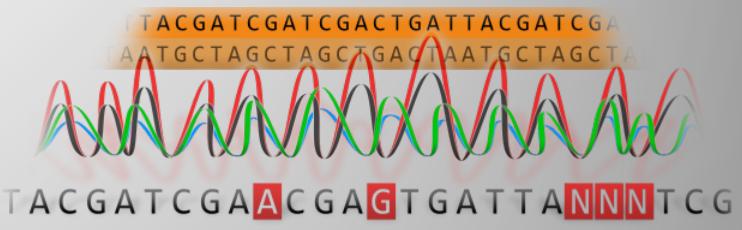
Image Processing

OpenCV: +45% to +50% (Blur)
Halide: +7% to +12 (Blur)
Halide: +37% to +44% (Harris Corner)

Ray Tracing

Ray Traversal
Embree: -15% to +13%
OptiX: -19% to -2%

Genome Sequence Alignment



SeqAn: -19% to -7%
NVBIO: -8% to -2%

intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Separation of Concerns

- Separation of concerns through code refinement
  - Higher-order functions
  - Partial evaluation
  - Triggered code generation

Application developer

```
fn main() {
  let result = gaussian_blur(img);
}
```

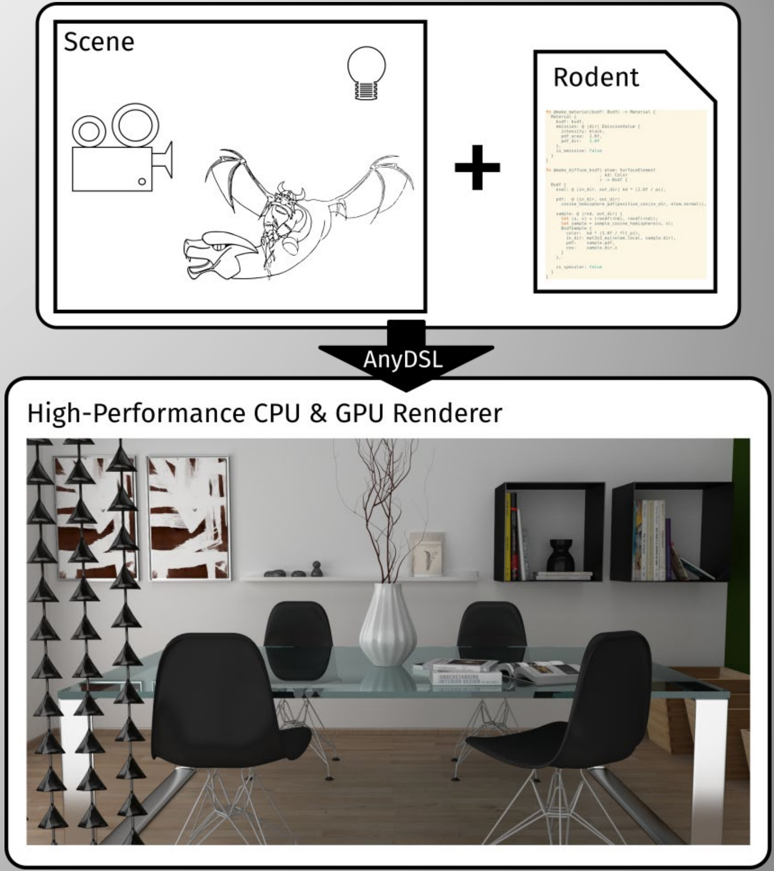
DSL developer

```
fn @gaussian_blur(img: Img) -> Img {
  let filter = /* ... */; let mut out = Img { /* ... */ };

  for x, y in iterate(out) {
    out(x, y) = apply(x, y, img, filter);
  }
  out
}
```

Machine expert

```
fn @iterate(img: Img, body: fn(int, int) -> ()) -> () {
  let grid  = (img.width, img.height);
  let block = (128, 1, 1);

   with nvvm(grid, block) {
    let x = nvvm_tid_x() + nvvm_ntid_x() + nvvm_ctaid_x();
    let y = nvvm_tid_y() + nvvm_ntid_y() + nvvm_ctaid_y();
    body(x, y);
  }
}
```

Case Study: Ray Tracing [SIGGRAPH’19]

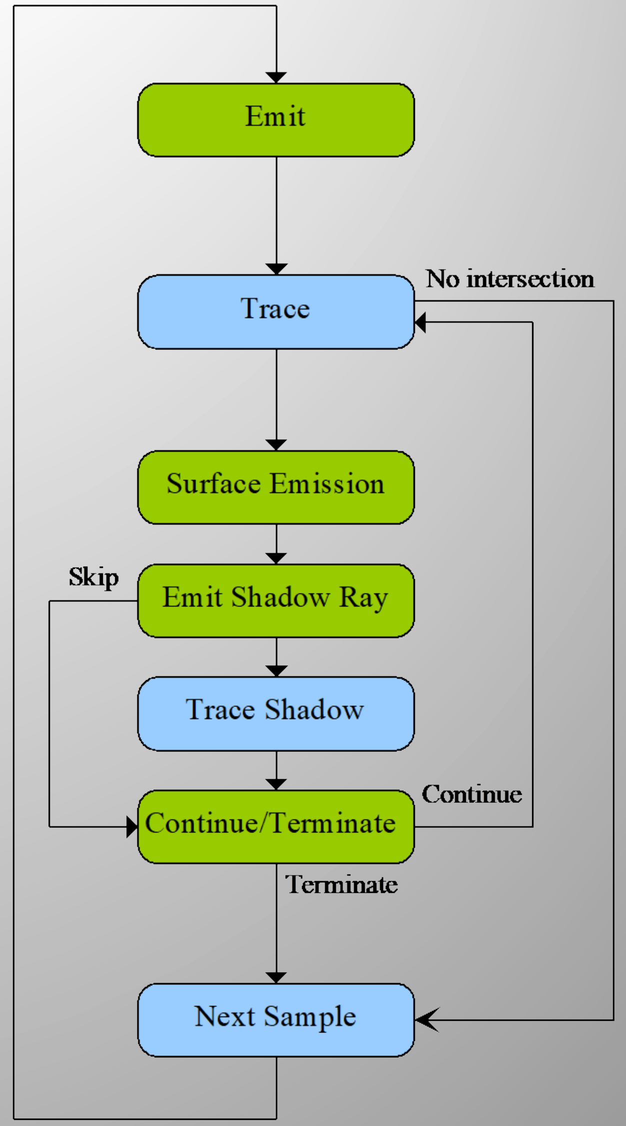


Rodent: Generating Renderers without Writing a Generator

https://github.com/AnyDSL/rodent

Rodent: Renderer + Traversal Library

- Renderer-generating library:
  Generate renderer that is optimized/specialized for a given input scene (or a class of scenes)
  - Generic, high-level, textbook code for
    - Shaders, lights, geometry, integrator, …
  - No low-level aspects
    - Strategy, scheduling, data layout, …
  - Separate mapping for each hardware
- 3D scenes are converted into code
  - E.g. from within Blender via exporter
  - Code triggers code generation



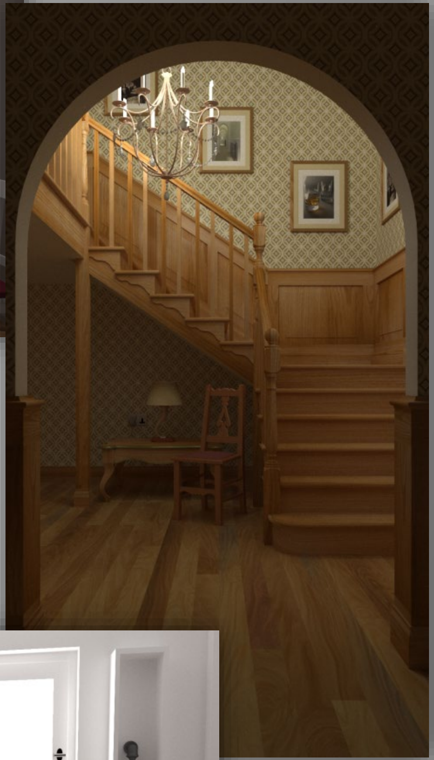
intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Features

- OptiX (NVIDIA)
  - NVIDIA GPU only
  - Generates megakernel (MK)
  - Not easy to extend (closed source)
- Embree + *ispc* (Intel)
  - amd64 only
  - Low-level, write-only code
- Rodent
  - NVIDIA & AMD GPUs
  - Megakernel & wavefront (WF)
  - Open source
  - amd64 & ARM support
  - High-level, textbook style code

intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
max planck institut informatik
Max Planck Institute for Software Systems

Test Scenes



intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Performance Results

- Cross-layer specialization (traversal + shading)
  - ~20% speedup vs. no specialization
- Optimal scheduling for each device
  - Megakernel vs. wavefront

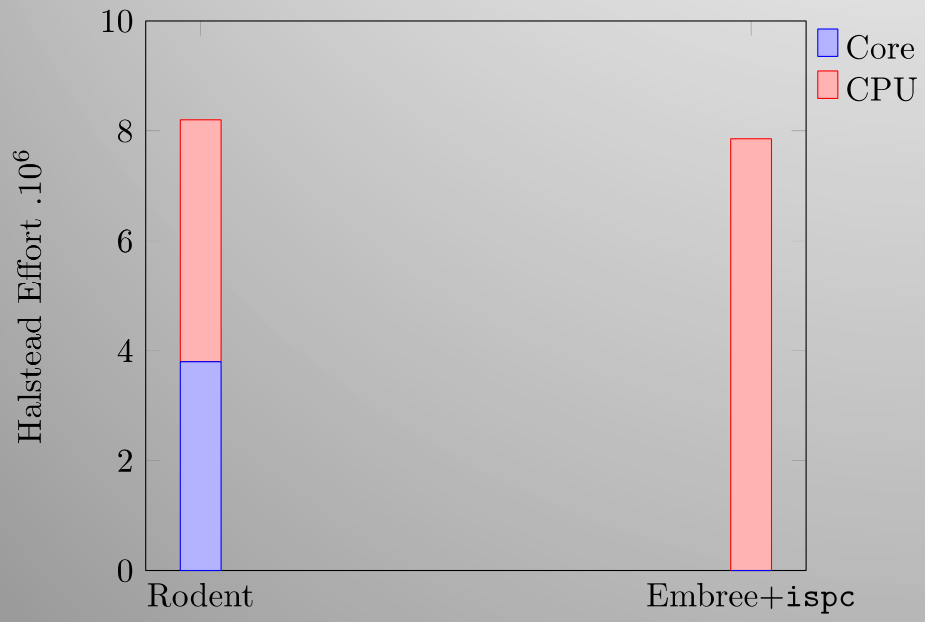
	CPU (Intel^{TM} i7 6700K)		GPU (NVIDIA^{TM} Titan X)			GPU (AMD^{TM} R9 Nano)	
Scene	Rodent^{WF}	Embree^{WF}	Rodent^{MK}	Rodent^{WF}	OptiX^{MK}	Rodent^{MK}	Rodent^{WF}
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.87	35.11
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.95	27.31
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	19.25	32.90
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.22	30.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.68	28.13
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.74	22.21

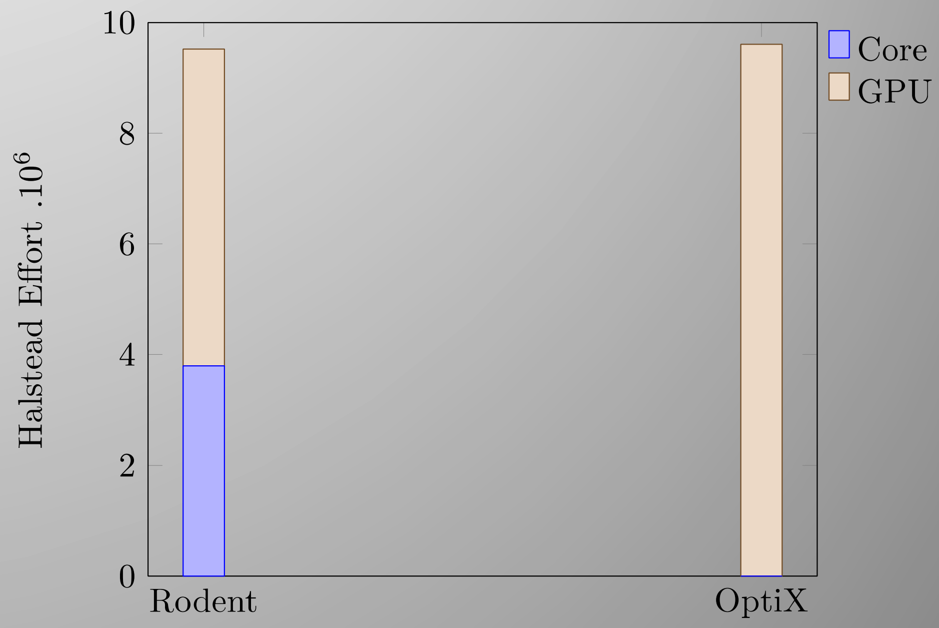
Msamples/s (higher is better). MK: Megakernel, WF: Wavefront.

intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Code Complexity

- Halstead's complexity measures
  - Reusable renderer core
  - More accurate than LoC
- Polyvariant and nested vectorization
  - Reusable code across architectures
  - Change vector width within vectorized region (e.g. hybrid traversal)



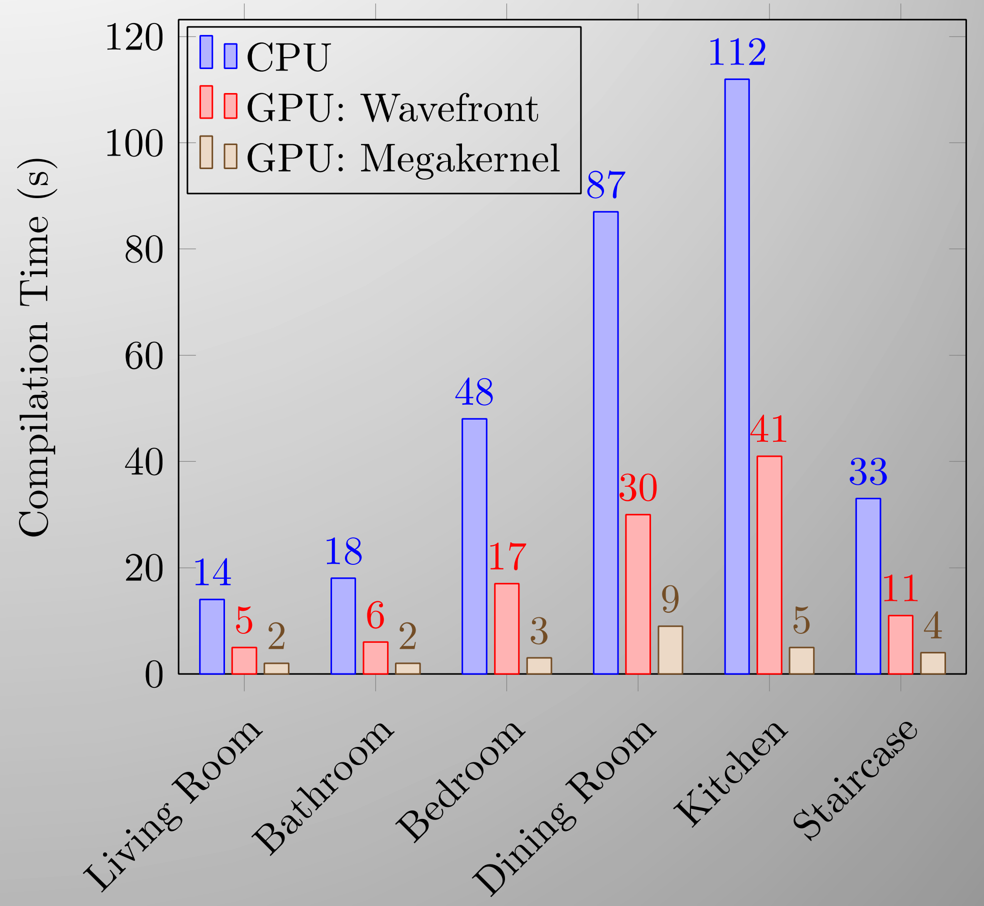


intel
VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpi max planck institut informatik
Max Planck Institute for Software Systems

Scene Statistics: Compile Time & Shader Fusion

- Megakernel only: shader fusion #initial → #unique → #fused
  - Living room: 19 → 16 → 6
  - Bathroom: 16 → 15 → 5
  - Dining room: 58 → 51 → 28
  - Kitchen: 129 → 95 → 19
  - Staircase: 31 → 27 → 11
  - Bedroom: 41 → 38 → 13

- Compilation times



VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
max planck institut informatik
Max Planck Institute for Software Systems

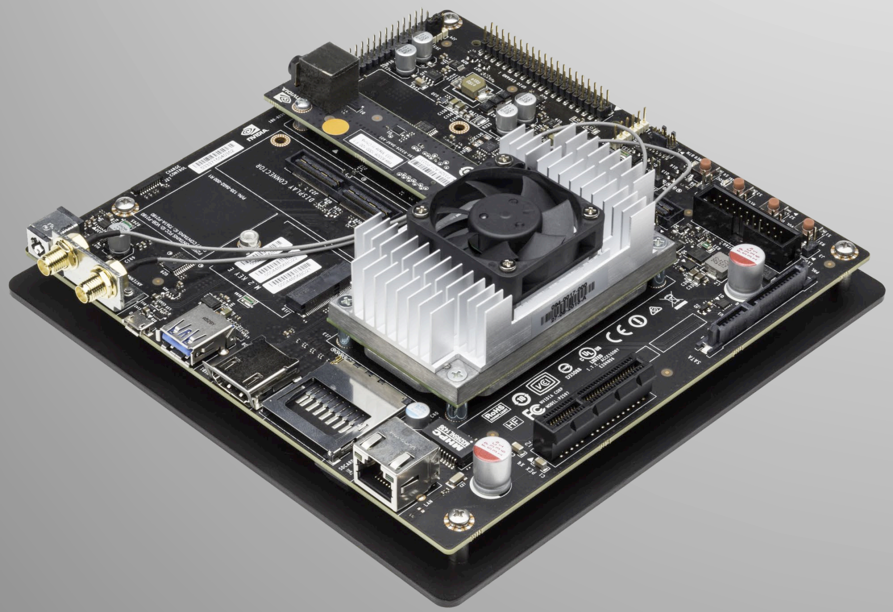
Thank you for your attention.
Questions?

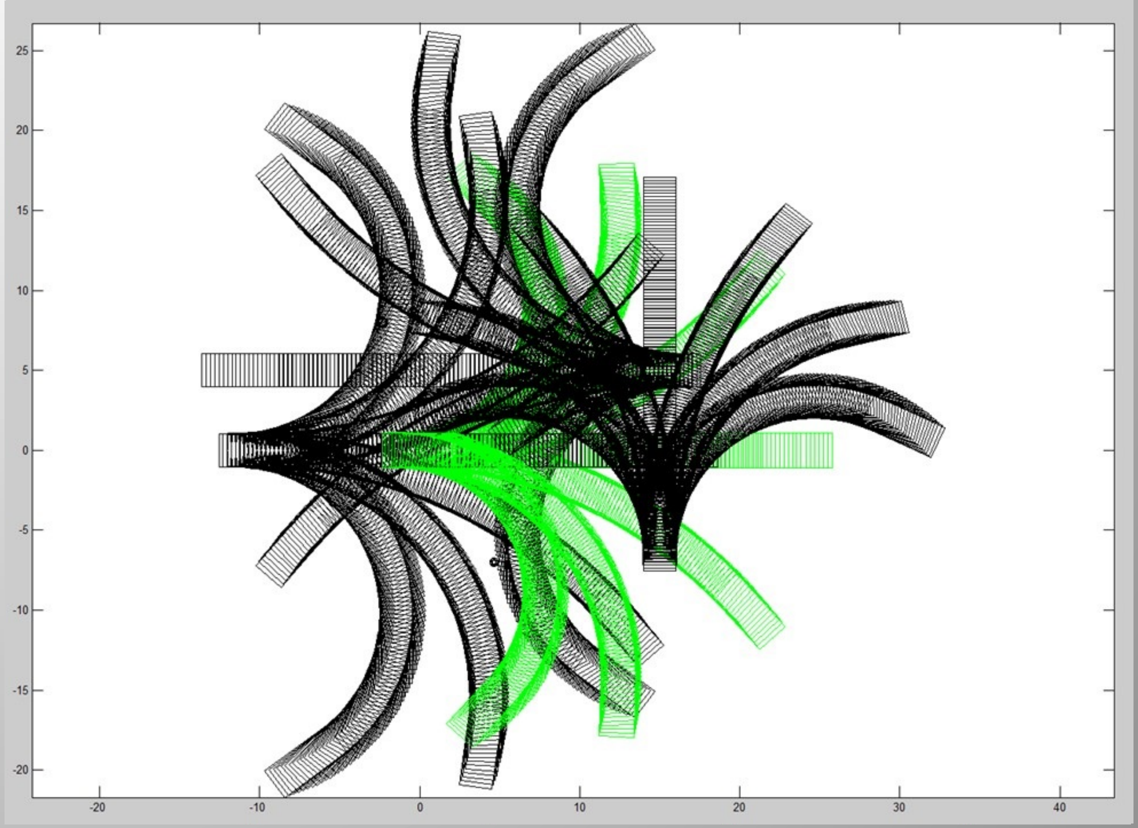
Case Study: Collision Avoidance & Crash Impact Point Optimization [GTC'16,IV'19]

Joint Project with Audi and THI

Prediction Approach to Environment Analysis

- Objects are described by their physical properties
- Movement is sampled and extrapolated
- All object hypotheses are combined with each other





intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

Performance Results

- Collision Avoidance
  - 8.6 million hypotheses combinations per collision object
- Scenario: 3 collision object + EGO vehicle
  - 26 million hypotheses combinations

Lang	HW	Time
MatLab	Intel Core i5	6 min
AnyDSL	Tegra X1 CPU	2 s
AnyDSL	Tegra X1 GPU	36 ms
AnyDSL	Drive PX2 GPU	15 ms

- Crash Impact Point Optimization
  - 0.9 million hypotheses combinations per collision object
- Scenario: 2 critical objects + EGO vehicle
  - 1.8 million hypotheses combinations

Lang	HW	Time
MatLab	Intel Core i5	16.5 s
AnyDSL	Tegra X1 CPU	0.3 s
AnyDSL	Tegra X1 GPU	8 ms
AnyDSL	Drive PX2 GPU	12 ms

DreamSpace

Case Study: DreamSpace EU Project

High Quality Rendering of Virtual Production Scenes

Key Achievements

Goals:

- High quality, global illumination rendering for real-time use
- With quality allowing creative use already during onset work
- Fully integrated into the Dreamspace ecosystem

Technology Developments:

- Improve and use of novel compiler framework (AnyDSL)
- Optimize core ray traversal and intersection engine
- Design a scalable, high-performance rendering architecture
- Create real-time distribution framework

Conclusion

- AnyDSL Framework
  - High-level, higher-order functional program representation
  - Novel code-refinement concept
  - Control over partial evaluation, vectorization, target code-generation

- Sample high-performance, domain-specific libraries (DSLs)
  - Stincilla: Stencil codes, image processing
  - RaTrace: Ray traversal kernels
  - Rodent: Renderer generator
  - AnySeq: Genome sequence alignment

Future Work

- Other high-performance libraries
  - Deep learning
  - Computer vision pipelines
  - Simulation, string matching, …
- Hardware synthesis as a backend
  - Very promising results with FPGAs!

intel VISUAL COMPUTING INSTITUTE
UNIVERSITÄT DES SAARLANDES
DFKI
mpii max planck institut informatik
Max Planck Institute for Software Systems

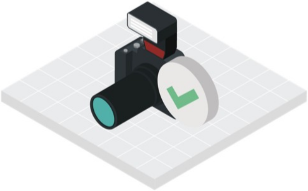
thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY – 1 AUGUST

RODENT: GENERATING RENDERERS WITHOUT WRITING A GENERATOR

A. Pérard-Gayot, R. Membarth,
R. Leissa, S. Hack, P. Slusallek

UNIVERSITÄT DES SAARLANDES
Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

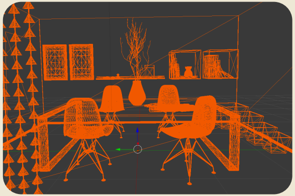
© 2019 SIGGRAPH. ALL RIGHTS RESERVED.



PHOTOGRAPHY & RECORDING ENCOURAGED

© 2019 SIGGRAPH. ALL RIGHTS RESERVED.

Overview



Scene

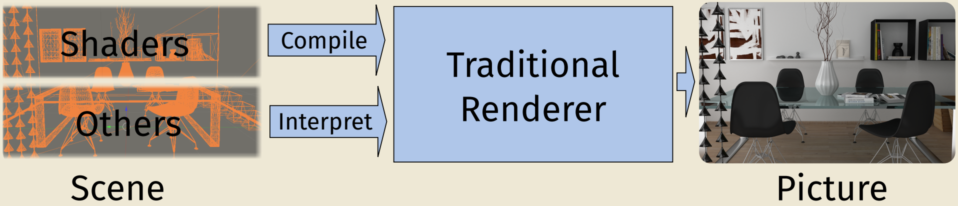
Traditional Renderer

Picture

What this talk is about

- Generating renderers from high-level, textbook-like code
- Specialized/optimized for a scene type
- High-performance: Up to 40%/20% faster than OptiX/Embree+`ispc`

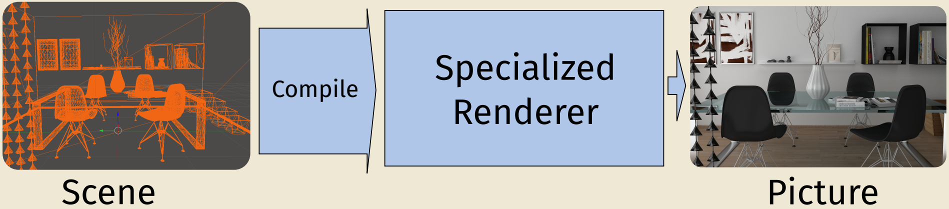
Rendering



In a traditional renderer

- Shaders are compiled by a (shader) compiler
  - Standard compiler optimizations
- Rest of the scene is interpreted during rendering
  - `if`/`else` branches (e.g. for renderer config/options)
  - Virtual function calls (e.g. for geometry types)
  - …

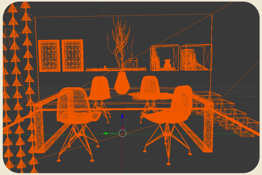
Rendering



In Rodent

- We compile the entire scene into a renderer
- We only use the scene type, not the actual scene data
  - No benefit from knowing e.g. the position of triangle 544
- We use Partial Evaluation
  - To avoid writing a *Renderer Generator*

Traditional Execution vs. Partial Evaluation



Scene

Traditional Renderer

Picture

Traditional program execution

Traditional Execution vs. Partial Evaluation

High-level Rendering Code

Partial Evaluation

Traditional Execution vs. Partial Evaluation

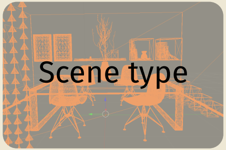
High-level Rendering Code

Rodent

Traditional Execution vs. Partial Evaluation

High-level Rendering Code

Rodent

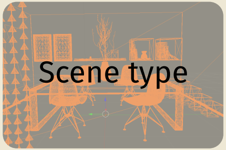


Partial Evaluation

Traditional Execution vs. Partial Evaluation

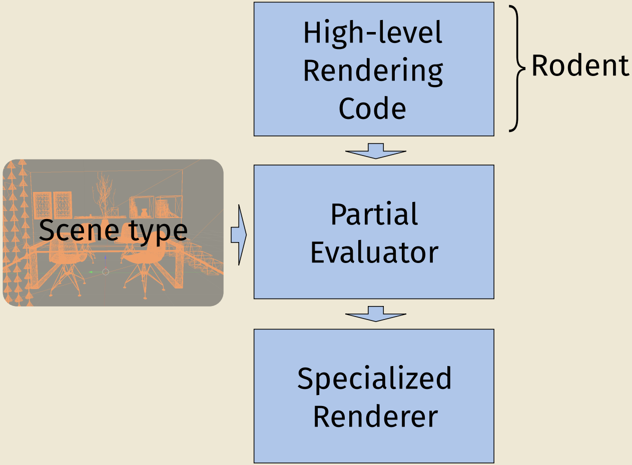
High-level Rendering Code

Rodent



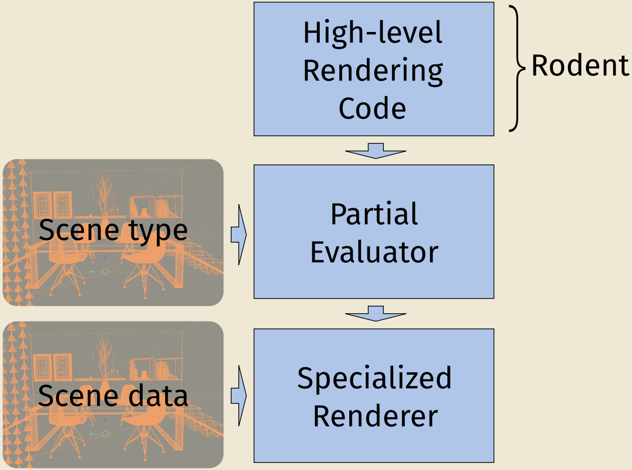
Partial Evaluator

Traditional Execution vs. Partial Evaluation



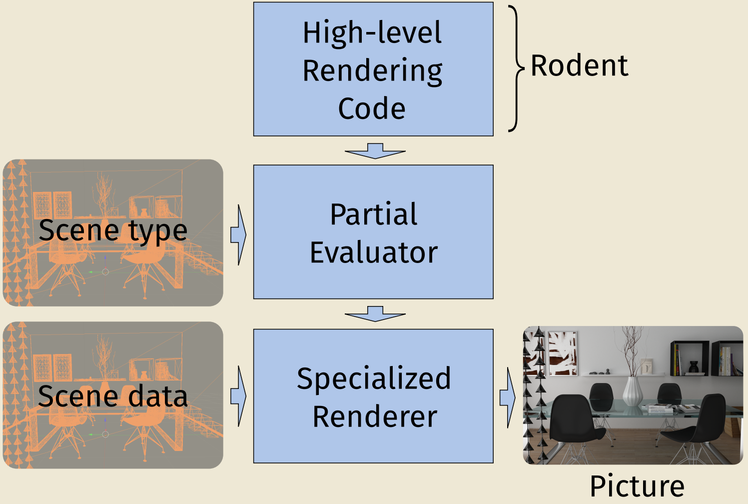
Partial Evaluation

Traditional Execution vs. Partial Evaluation



Partial Evaluation

Traditional Execution vs. Partial Evaluation



Partial Evaluation

AnyDSL

- This work leverages the AnyDSL compiler framework
  - `https://github.com/AnyDSL`
- Provides user-guided Partial Evaluation
- High-performance code generation using LLVM
- Can target/optimize for CPUs or GPUs
  - Intel/AMD/NVIDIA/ARM/...

Rendering Library Design

- High-level, textbook-like
  - In the spirit of PBRT
- Descriptive and modular
  - Separate the algorithm (”what”) from the schedule/hardware mapping (”how”)
- High-performance
  - Different hardware *mappings*
  - CPUs/GPUs have different execution models
  - Need efficient and flexible abstractions

The ”What”

BSDF

```
struct Bsdf {
  // Evaluation of the function given a pair of directions
  eval: fn (Vec3, Vec3) -> Color,

  // Probability density function used during sampling
  pdf: fn (Vec3, Vec3) -> f32,

  // Samples a direction (importance sampled according to this BSDF)
  sample: fn (Vec3) -> BsdfSample,
}
```

Example: Diffuse BSDF

```
fn @make_diffuse_bsdf(surf: SurfaceElement, kd: Color) -> Bsdf {
  Bsdf {
    eval: @ |in_dir, out_dir| kd * (1.0f / pi),
    pdf:  @|in_dir, out_dir|
      cosine_hemisphere_pdf(positive_cos(in_dir, surf.normal)),
    sample: @ |out_dir| {
      let sample = sample_cosine_hemisphere(rand(), rand());
      let color  = kd * (1.0f / pi);
      make_bsdf_sample(surf, sample, color)
    }
  }
}
```

- `@` triggers partial evaluation/specializes the function
- Replaces the function by its contents at the call site to allow optimizations

Defining a scene with Rodent

- BSDFs:

```
let diff = make_diffuse_bsdf(kd);
let spec = make_phong_bsdf(ns, ks);
let bsdf = make_mix_bsdf(spec, diff, k);
```

Defining a scene with Rodent

- BSDFs:

```
let diff = make_diffuse_bsdf(kd);
let spec = make_phong_bsdf(ns, ks);
let bsdf = make_mix_bsdf(spec, diff, k);
```

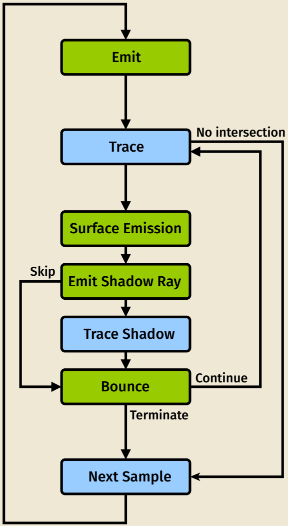
- Light sources, textures, geometric objects, ...

Rodent is a Scene Description Language

```
let renderer = make_path_tracing_renderer(/* ... */);
let geometry = make_tri_mesh_geometry(/* ... */);

let tex = make_image_texture(/* ... */);

let shader = |ray, hit, surface| {
    let uv = surface.attribute(0).as_vec2;
    make_diffuse_bsdf(surface, tex(uv1));
};

let scene = make_scene(geometry, /* ... */);
```

BSDF DSL + Light DSL + Geometry DSL + ... = Scene language embedded in AnyDSL

Abstracting the Rendering Process




```
struct Tracer {
  on_emit:   OnEmitFn,
  on_hit:    OnHitFn,
  on_shadow: OnShadowFn,
  on_bounce: OnBounceFn,
}
```

- Can also be used for bidir. algorithms
- Green nodes: the algorithm
  What should be computed
- Blue nodes: the schedule
  How it should be computed

The ”How”

Mapping Renderers to Hardware

- The `Device` contains hardware-specific routines:

```
struct Device {
  trace: fn (Scene, Tracer) -> (),
  /* ... */
}
```

- Schedule renderers differently depending on the platform
  - Wavefront: Batches (larger than SIMD width) of rays together
  - Megakernel: Large compute kernel, one ray at a time (used in OptiX)
- Rodent implements 3 devices:
  1. CPU: Wavefront
  2. GPU: Megakernel
  3. GPU: Wavefront

Wavefront Devices

On CPUs

- Processes a small (∼1000 rays) batch of rays together
  - Maximize cache efficiency
- Sort rays by shader and process contiguous ranges
- Uses vectorization and specialization, simplified:

```
for shader in unroll(0, scene.num_shaders) {
  // Get the range of rays for this shader
  let (begin, end) = ray_range_by_shader(shader);
  for i in vectorize(vector_width, begin, end) {
    // Scalar code using on_hit(), on_shadow(), ...
    // => automatically vectorized
  }
}
```

Wavefront Devices

On CPUs

- Processes a small (∼1000 rays) batch of rays together
  - Maximize cache efficiency
- Sort rays by shader and process contiguous ranges
- Uses vectorization and specialization, simplified:

```
for shader in unroll(0, scene.num_shaders) {
  // Get the range of rays for this shader
  let (begin, end) = ray_range_by_shader(shader);
  for i in vectorize(vector_width, begin, end) {
    // Scalar code using on_hit(), on_shadow(), ...
    // => automatically vectorized
  }
}
```

```
i∈unroll(0,3)
└j∈vectorize(w,begin(i),end(i))
                ⬇
j0∈vectorize(w,begin(0),end(0))
j1∈vectorize(w,begin(1),end(1))
j2∈vectorize(w,begin(2),end(2))
```

Wavefront Devices

On GPUs

- Processes a larger ($\sim$1M rays) batch of rays
  - Maximize parallelism
- Sort rays by shader and process contiguous ranges
- Generates one kernel per shader, with specialization, simplified:

```
for shader in unroll(0, scene.num_shaders) {
  // Get the range of rays for this shader
  let (begin, end) = ray_range_by_shader(shader);
  let grid = (round_up(end - begin, block_size), 1, 1);
  let block = (block_size, 1, 1);
  with work_item in cuda(grid, block) {
    // Use on_hit(), on_shadow(), ...
  }
}
```

Wavefront Devices

On GPUs

- Processes a larger (~1M rays) batch of rays
  - Maximize parallelism
- Sort rays by shader and process contiguous ranges
- Generates one kernel per shader, with specialization, simplified:

```
for shader in unroll(0, scene.num_shaders) {
  // Get the range of rays for this shader
  let (begin, end) = ray_range_by_shader(shader);
  let grid = (round_up(end – begin, block_size), 1, 1);
  let block = (block_size, 1, 1);
  with work_item in cuda(grid, block) {
    // Use on_hit(), on_shadow(), ...
  }
}
```

```
i∈unroll(0,3)
 ┗cuda(grid(i),block(i))
          ⬇
cuda(grid(0),block(0))
cuda(grid(1),block(1))
cuda(grid(2),block(2))
```

Megakernel GPU Device

- Rays are local to the current execution thread
- Rendering loop *inside* the kernel, simplified:

```
fn trace(scene: Scene, tracer: Tracer) -> () {
  with work_item in cuda(grid, block) {
    let (x, y) = (work_item.gidx(), work_item.gidy());
    let (ray, state) = tracer.on_emit(x, y);
    let mut terminated = false;
    while !terminated {
      // Trace + use on_hit(), on_shadow(), ...
    }
  }
}
```

Evaluation

- Versus high-performance, state-of-the-art frameworks:
  - Embree + `ispc`: only for x86/amd64
  - OptiX: only for CUDA hardware
- Built custom, simple renderers based on those frameworks
  - Following documentation
  - Only implemented features required to render the test scenes
- Measured:
  - Performance
  - Code complexity
- Workflow: Convert scene to AnyDSL $\Rightarrow$ compile $\Rightarrow$ render

Scenes

786k tris./ 13 mats.

1.231M tris./14 mats.

545k tris./35 mats.

718k tris./44 mats.

612k tris./61 mats.

263k tris./23 mats.

Scenes by Wig42, nacimus, SlykDrako, MaTTeSr, Jay-Artist, licensed under CC-BY 3.0/CC0 1.0. See paper for details.

Results: Performance

Scene	CPU (Intel™ i7 6700K)		GPU (NVIDIA™ Titan X)			GPU (AMD™ R9 Nano)	
	Rodent[2]	Embree	Rodent[1]	Rodent[2]	OptiX	Rodent[1]	Rodent[2]
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.87	35.11
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.95	27.31
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	19.25	32.90
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.22	30.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.68	28.13
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.74	22.21

(1) Megakernel, (2) Wavefront

Results: Performance

	CPU (Intel™ i7 6700K)		GPU (NVIDIA™ Titan X)			GPU (AMD™ R9 Nano)	
Scene	Rodent[2]	Embree	Rodent[1]	Rodent[2]	OptiX	Rodent[1]	Rodent[2]
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.87	35.11
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.95	27.31
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	19.25	32.90
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.22	30.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.68	28.13
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.74	22.21

(1) Megakernel, (2) Wavefront

- Between $+1 - 23\%$ vs. Embree
    - Around $60 - 70\%$ of the time tracing rays
    - Traversal algorithms in Embree are already specialized
    - Rodent's shading alone is around $2\times$ faster than with `ispc`

Results: Performance

Scene	CPU (Intel™ i7 6700K)		GPU (NVIDIA™ Titan X)			GPU (AMD™ R9 Nano)	
	Rodent[2]	Embree	Rodent[1]	Rodent[2]	OptiX	Rodent[1]	Rodent[2]
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.87	35.11
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.95	27.31
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	19.25	32.90
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.22	30.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.68	28.13
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.74	22.21

(1) Megakernel, (2) Wavefront

- Between $+1 - 23\%$ vs. Embree
  - Around $60 - 70\%$ of the time tracing rays
  - Traversal algorithms in Embree are already specialized
  - Rodent's shading alone is around $2\times$ faster than with `ispc`
- Between $+2 - 31\%$ vs OptiX (Megakernel)

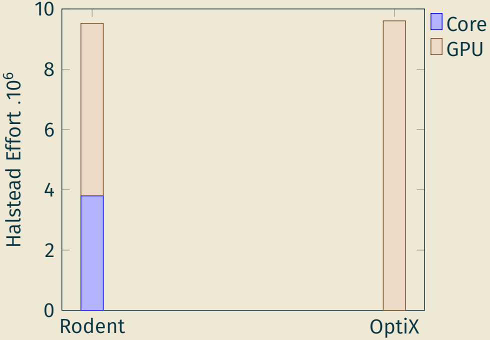
Results: Performance

	CPU (Intel™ i7 6700K)		GPU (NVIDIA™ Titan X)			GPU (AMD™ R9 Nano)	
Scene	Rodent[2]	Embree	Rodent[1]	Rodent[2]	OptiX	Rodent[1]	Rodent[2]
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.87	35.11
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.95	27.31
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	19.25	32.90
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.22	30.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.68	28.13
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.74	22.21

(1) Megakernel, (2) Wavefront

- Between $+1-23\%$ vs. Embree
  - Around $60-70\%$ of the time tracing rays
  - Traversal algorithms in Embree are already specialized
  - Rodent's shading alone is around $2\times$ faster than with `ispc`
- Between $+2-31\%$ vs OptiX (Megakernel)
- Between $+29-42\%$ vs OptiX (Wavefront)
  - Wavefront scales better with shader complexity
  - Not limited by register pressure

Results: Code Complexity



- Embree: only on x86/amd64
- Rodent: also on ARM
  + other LLVM targets (RISC-V?)

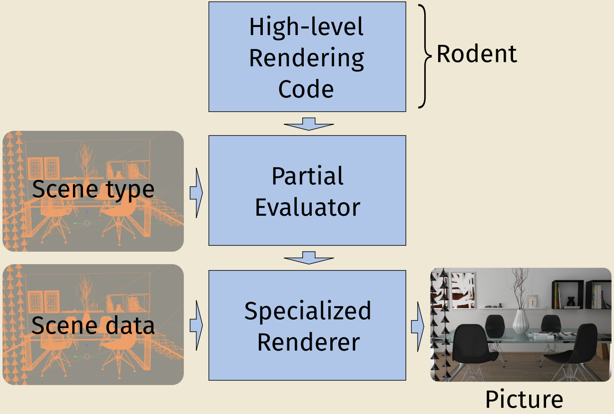
- OptiX: only Megakernel, only CUDA hw.
- Rodent: also on AMD™ GPUs
  + other LLVM targets (Intel™ GPU?)

Conclusion

Rodent *generates* high-performance renderers *without writing a generator*

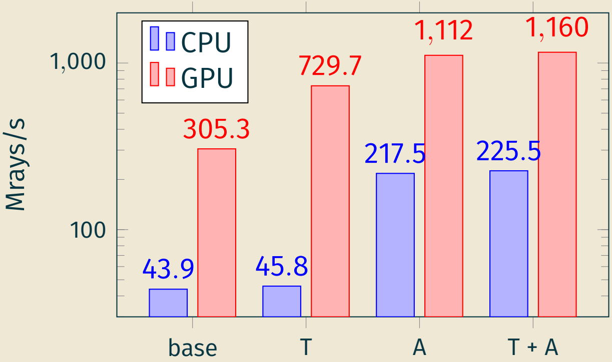
- Defines textbook-like, generic algorithms
- Provides tailored hardware schedules for different CPUs and GPUs
- Specializes code according to the scene via AnyDSL
- Runs up to 40% faster than state-of-the-art

Questions?



https://github.com/AnyDSL/rodent

Results: Impact of Specialization

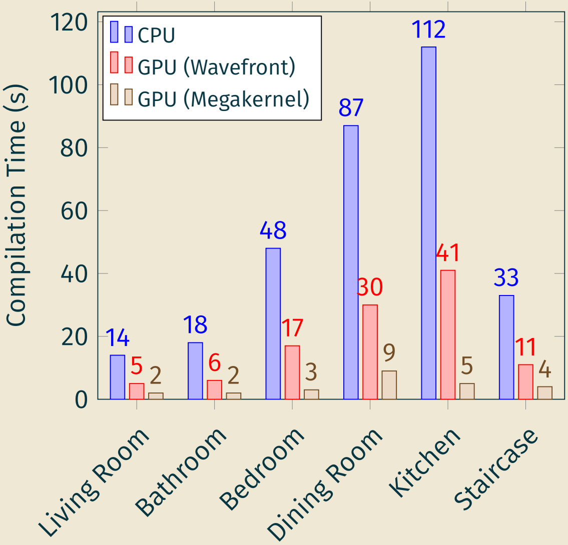


- Base: No specialization
- T: Specialize the interface (shader $\longleftrightarrow$ texturing function)
- A: Specialize the interface (shader $\longleftrightarrow$ mesh attribute)

Specialization: Caveats

- Specialization may lead to increased compilation times
- Specializing to much may increase register pressure
  - Dangerous for the megakernel device
  - Not a problem for the wavefront device
- Rodent fuses simple/similar shaders together
  - Only for the megarkernel device
  - Mitigates problems of divergence and reg. pressure

Results: Compilation Times



Improving Compilation Times

- The more there is to specialize, the slower
- Compiler itself is not particularly optimized for speed
- Parts of the renderer can be pre-compiled
- Does not need to know *everything* in the scene
  - The less is known the less specialization will happen
  - Automatically done by the compiler thanks to annotations
  - Can be exploited to make compilation faster