AnyDSL: A Compiler-Framework for Domain-Specific Libraries (DSLs)

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

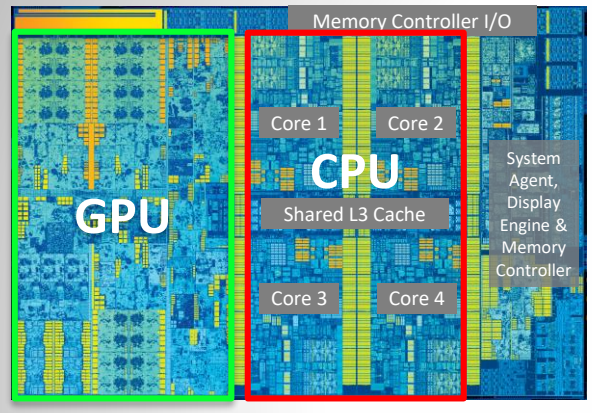
Roland Leißa, Klaas Boesche, Simon Moll, Sebastian Hack (Compiler)

Intel Visual Computing Institute (IVCI) at Saarland University

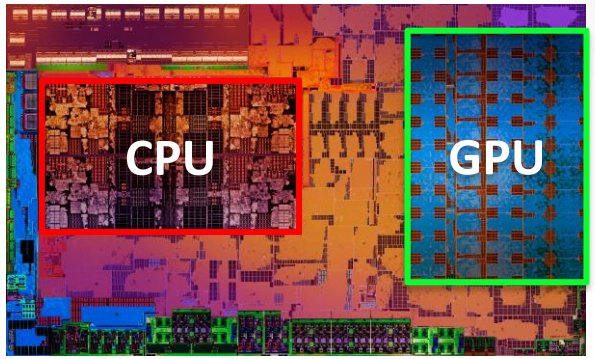
German Research Center for Artificial Intelligence (DFKI)

Many-Core Dilemma

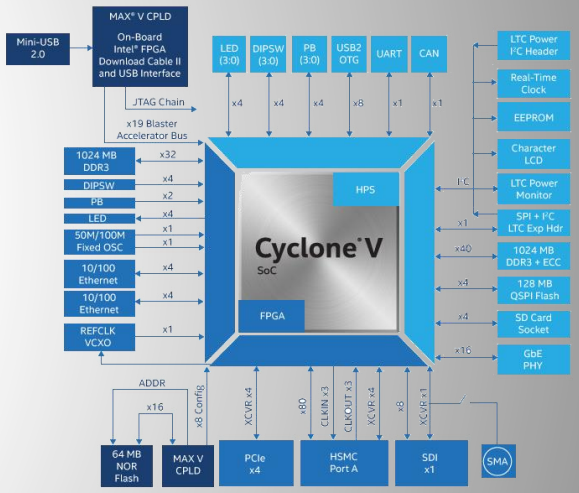
- Many-core hardware is everywhere – but programming it is still hard



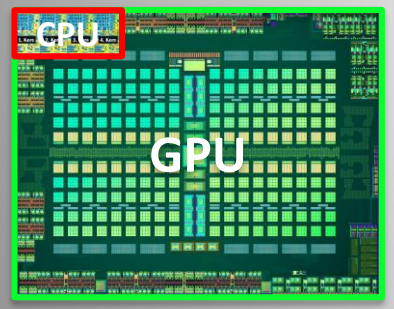
Intel Skylake (1.8B transistors)



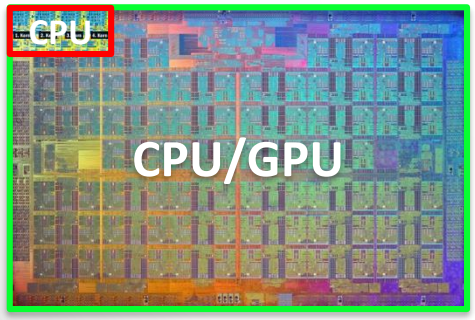
AMD Zen + Vega (4.9B transistors)



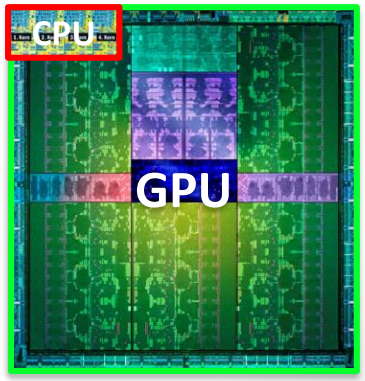
Intel / Altera Cyclone V



AMD Polaris
(~5.7B transistors)



Intel Knights Landing
(~8B transistors)



NVIDIA Kepler
(~7B transistors)

Program Optimization for Target Hardware

- Von Neumann is dead: Programs must be specialized for
  - SIMD instructions & width,
  - Memory layout & alignment,
  - Memory hierarchy & blocking, ...
- Compiler will not solve the problem !!
  - Languages express only a fraction of the domain knowledge
  - Most compiler algorithms are NP-hard
- Our languages are stuck in the `80ies
  - No separation of conceptual abstractions and implementations
  - Implementation aspects easily overgrow algorithmic aspects

Example: Stencil Codes in OpenCV (Image Processing)

- Example: Separable image filtering kernels for GPU (CUDA)
  - Architecture-dependent optimizations (via lots of macros)
  - Separate code for each stencil size (1 .. 32)
  - 5 boundary handling modes
  - Separate implementation for row and column component

    ➔ **2 x 160 explicit code variants all specialized at compile-time**
- Problems
  - Hard to maintain
  - Long compilation times
  - Lots of unneeded code
  - Multiple ***incompatible*** implementations: CPU, CUDA, OpenCL, …

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

Existing Approaches (1)

- Optimizing Compilers
  - Auto-Parallelization or parallelization of annotated code (#pragma)
  - OpenACC, OpenMP, …
- New Languages
  - Introduce syntax to express parallel computation
  - CUDA, OpenCL, X10, …

Existing Approaches (2)

- Libraries of hand-optimized algorithms
  - Hand-tuned implementations for given application (domain) and target architecture(s)
  - IPP, NPP, OpenCV, Thrust, …
- Domain-Specific Languages (DSLs)
  - Compiler & Language (hybrid approach)
  - Concise description of problems in a domain
  - Halide, HIPA^{cc}, LMS, Terra, …
- But good language and compiler construction are really hard problems

Domain-Specific Languages

- Address the needs of different groups of experts working at different levels:
  - Machine expert
    - Provides generic, low-level abstraction of hardware functionality
  - Domain expert
    - Defines a DSL as a set of domain-specific abstractions, interfaces, and algorithms
    - Uses (multiple levels of) lower level abstractions
  - Application developer
    - Uses the provided functionality in an application program

None of them knows about compiler & language construction!

Programmer has no/little influence on compiler transformations!

RTfact

RTfact: A DSL for Ray Tracing

- **Data Structures: e.g. paket of rays**

```
template<unsigned int size>
struct RayPacket
{
  Vec3f<size> org;
  Vec3f<size> dir;
  Packet<size, float> tMin;
  Packet<size, float> tMax;
};
```

- **A ray packet can be**
  - Single ray (size == 1)
  - A larger packet of rays (size > 1)
  - A hierarchy of ray packets (size is a multiple of packets of N rays)
  - Several sizes can exist at the same time
  - Can be allocated on the stack (size is know to the compiler)

C++ Concepts (ideally)

- **Like a class declaration – just for templates**
  - Unfortunately, not included in new C++ standard

```
template<class Element>
class KdTree {
public:
  class NodeIterator;
  class ElementIterator;
  // interface for structure creation
  void createInnerNode(NodeIterator node,
                       int axis, float splitValue);
  template<class Iterator>
  void createLeaf(NodeIterator leaf, const BBox& bounds,
                  Iterator begin, Iterator end);
  // interface for structure traversal
  NodeIterator getRoot() const;
  NodeIterator getLeftChild(NodeIterator node) const;
  NodeIterator getRightChild(NodeIterator node) const;
  int getSplitAxis(NodeIterator node) const;
  float getSplitValue(NodeIterator node) const;
  std::pair<ElementIterator, ElementIterator>
    getElementList(NodeIterator leaf) const;
};
```

Composition

```
// data structure
BVH<KdTree<Triangle>> hierarchy;
// corresponding intersector
BVHIntersector<KdTreeIntersector<
  SimpleTriangleIntersector>> intersector;
```

Example: Traversal

```
template<class ElemIsect>    //nested element intersector
template<class KdTree,       //models the KdTree concept
         unsigned int size,  //size of the ray packet
         bool commonOrg,     //common ray origin?
         bool computeInters> //intersection data needed?
void KdTreeIntersector<ObjIsect>::intersect(
  RayPacket<size>& rayPacket,//the actual rays
  KdTree& tree,              //the actual kd-tree
  ElemIsect::Intersection<size>& r)//intersection defined
{                            // by the nested intersector
  typedef BitMask<size>                t_BitMask;
  typedef Packet<size,float>           t_Packet;
  typedef typename t_Packet::Container t_Container;
  /* omitted: initialize traversal */
  KdTree::NodeIterator node = tree.getRoot();
  int splitDim; // split dimension (3 means leaf node)
  while(true) {
    while((splitDim = tree.getSplitAxis(node)) != 3)
      t_Container splitValue =
        t_Container::replicate(tree.getSplitValue(node));
      t_BitMask farChildConditionMask,
                nearChildConditionMask;
      t_Container tSplitFactor;
      if(commonOrg) // compile-time constant decision
        tSplitFactor = splitValue -
          rayPacket.org(0).get(splitDimension);
```

Example: Traversal

```
      for(int i=0;i<RayPacket<size>::CONTAINER_COUNT;++i){
        if(!commonOrg) // compile-time constant decision
          tSplitFactor = splitValue -
            rayPacket.org(i).get(splitDimension);

        const t_Container tSplit = tSplitFactor *
          rayPacket.invDir(i).get(splitDimension);
        farChildConditionMask.setContainer(
          i, (currentTMin(i) > tSplit(i)).getIntMask());
        nearChildConditionMask.setContainer(
          i, (tSplit(i) > currentTMax(i)).getIntMask());
      }
      /*omitted: get first child from masks and descend */
    }
    KdTree::ObjectIterators objIterators =
      aTree.getObjectIterators(currentIterator);

    if(objIterators.first != objIterators.second) {
      do { //invoke nested intersector for leaf elements
        m_inters.intersect<commonOrg, computeInters>(
          rayPacket, *(objIterators.first++), r);
      } while(objIterators.first != objIterators.second);

      //check whether active rays found intersections
      terminationMask |= rayActiveMask &
        (result.dist <= currentTMax).getBitMask();

      if(terminationMask.isTrue()) return;
    }
    /* omitted: pop a node from the stack and mask rays*/
  }
};
```

Example: RT versus Shading

```
class Integrator {
public:
  template<int size> struct Color;

  template<int size, class Scene,
           class AccelStruct, class Intersector>
  Color<size> evaluate(
    RayPacket<size>& rayPacket,//initial ray packet
    AccelStruct& accelStruct,  //top-level accel. struct.
    Intersector& intersector,  //top-level intersector
    Scene& scene);             //shading scene data
```

Listing 5: A concept for an integrator. In the decoupled shading model, all rays are shot by integrators. Scene data is usually needed only for querying materials and light sources. Specific implementations are similar to PBRT's.

Example Ray Tracer

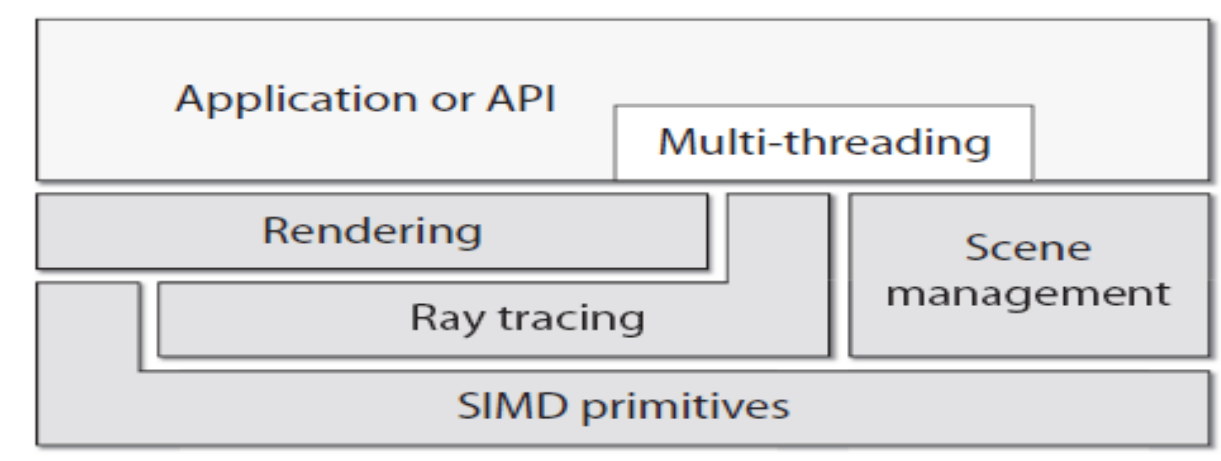
```
template<class PixelSampler, class Integrator>
class RayTracingRenderer {
  PixelSampler sampler;
  Integrator integrator;
public:
  template<int size, // the size of primary ray packets
           class Camera, class Scene, class AccelStruct,
           class Intersector, class Framebuffer>
  void render(
    Scene& scene, Camera& camera,
    Framebuffer& framebuffer, ImageClipRegion& clip,
    AccelStruct& accelStruct, Intersector& intersector)
  {
    PixelSampler::Sample<size> sample;
    PixelSampler::Iterator<size> it =
      sampler.getIterator<size>(clip);

    while(it.getNextSample(sample))
    {
      RayPacket<size> rays = camera.genRay(sample);
      Integrator::Color<size> color = integrator.eval(
        sample, rays, scene, accelStruct, intersector);
      sampler.writeColor(sample, color, framebuffer);
    }
  }
  /* omitted: preprocess() function for pre-integration*/
};
```

Example: Ray Tracer

```
PinholeCamera camera;
OpenGLFrameBuffer fb;

//surface rendering with a BVH (Fig. 5)
BasicScene<Triangle> scene; //initialization omitted
BVH<Triangle> bvh;
BVHBuilder builder;
builder.build(bvh, scene.prim.begin(),scene.prim.end());
BVHIntersector<PlueckerTriangleIntersector> bvhIsect;
RayTracingRenderer<PixelCenterSampler,
                   DirectIlluminationIntegrator> renderer1;
renderer1.render<64>(scene, camera, fb, fb.getClipRegion(),
                     bvh, bvhIsect);
```

```
//level-of-detail point cloud rendering (Fig. 1, 4 right)
BasicScene<Point> scene; //initialization omitted
LoDKdTree<Point> kdtree;
LoDKdTreeBuilder builder;
builder.build(kdtree, scene.prim.begin(),scene.prim.end());
LoDKdTreeIntersector<PointIntersector> pointIntersector;
RayTracingRenderer<PixelCenterSampler,
                   PointLoDIntegrator> renderer2;
renderer2.render<16>(scene, camera, fb, fb.getClipRegion(),
                     tree, lodKdTreeIsect);
```

Framework


Application or API
Multi-threading
Rendering
Scene management
Ray tracing
SIMD primitives

Evaluation

Figure 3: The CONFERENCE scene rendered with RTfact at a 1024^2 resolution on a notebook Core 2 Duo processor (8.2 fps).

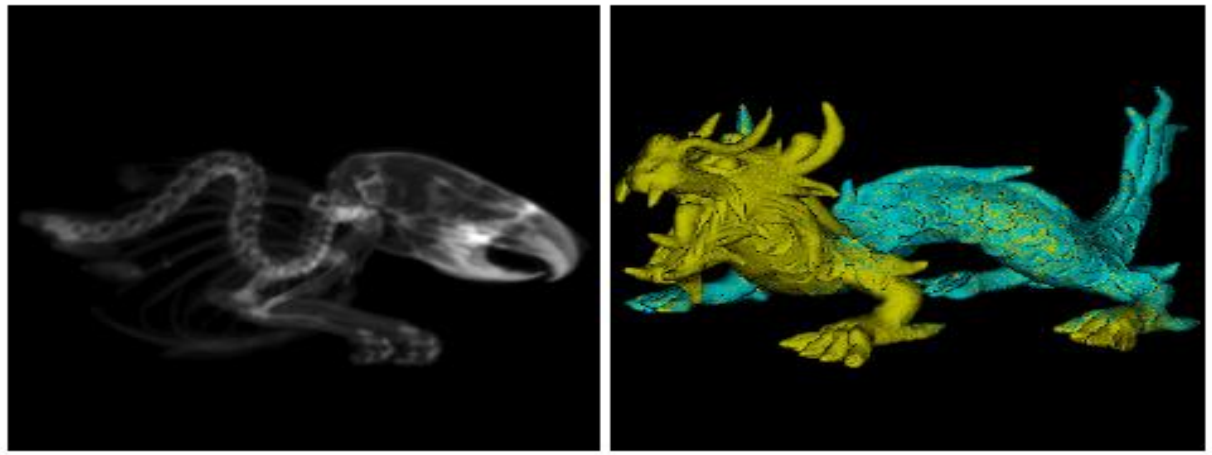


Figure 4: Direct volume and point-based rendering. Left: The Skeleton dataset. Right: The Asian Dragon model with level of detail.

Volume

Points

Performance

- **Preliminary Performance Comparison**
  - Needed common denominator to be able to compare

	SPONZA	CONFERENCE	SODA HALL
OpenRT K	4.5	4.2	5.1
Manta K	4.7	4.2	5.4
RTfact K	6.8	6.4	6.5
Wald et al. [31] B	n/a	9.3	11.1
Manta B	4.5	4.8	5.6
Arauna B	13.2	11.3	n/a
RTfact B	13.1	11.6	11.4

Figure 5: The SPONZA scene rendered with RTfact at a 1024^2 resolution on a notebook Core 2 Duo processor with three light sources (7.1 fps).

AnyDSL

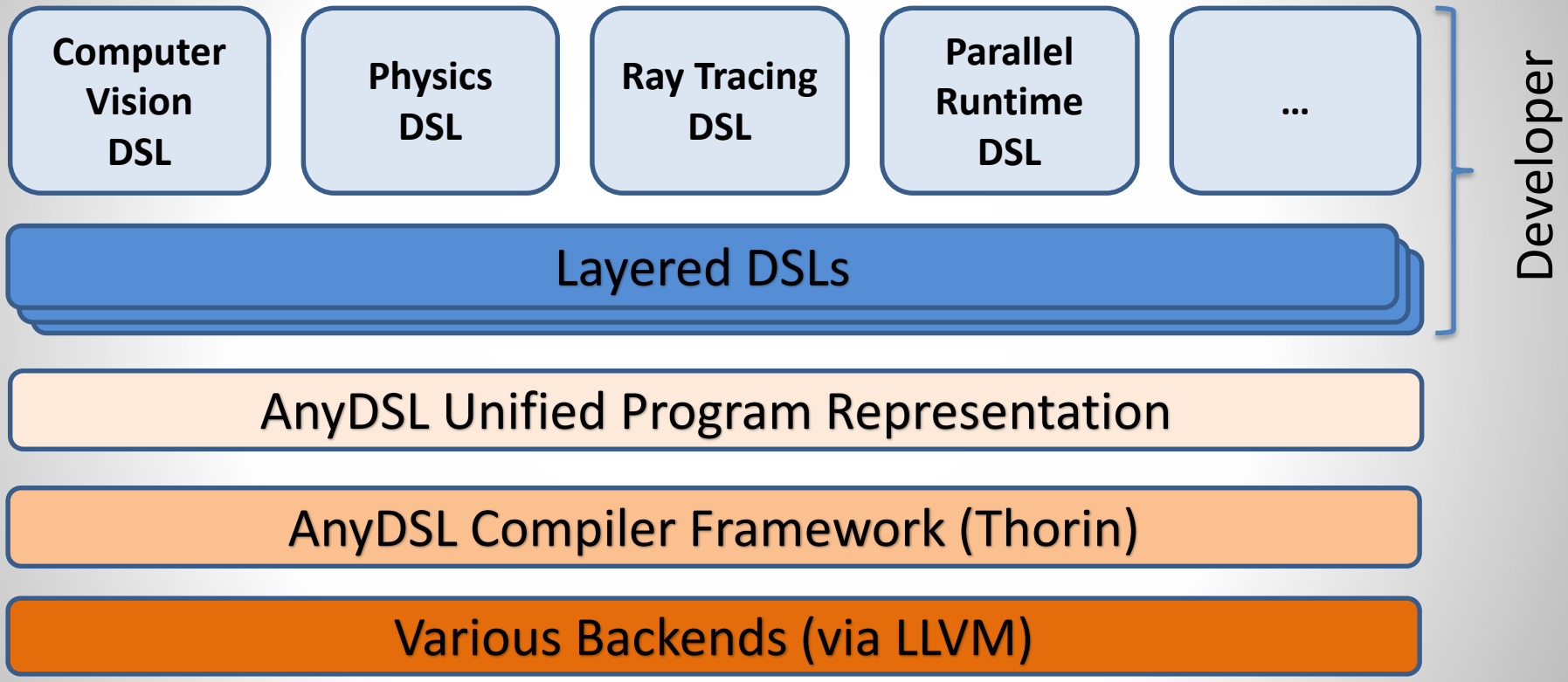
AnyDSL Goals

- Bring back control to the programmer
- Features:
  - Enable ***hierarchies of abstractions for any set of domains*** within the same language
  - Use ***refinement*** to specify ***efficient transformation*** to HW or lower-level abstractions
  - Provide ***configuration and parameterization data*** at ***each level of abstraction***
- Optimization:
  - Developer-driven ***aggressive specialization across all levels of abstraction***
  - Also provide functionality for explicit vectorization, target code generation, …

- AnyDSL:
  - Ability to define your own high-performance **Domain-Specific Libraries (DSL)**

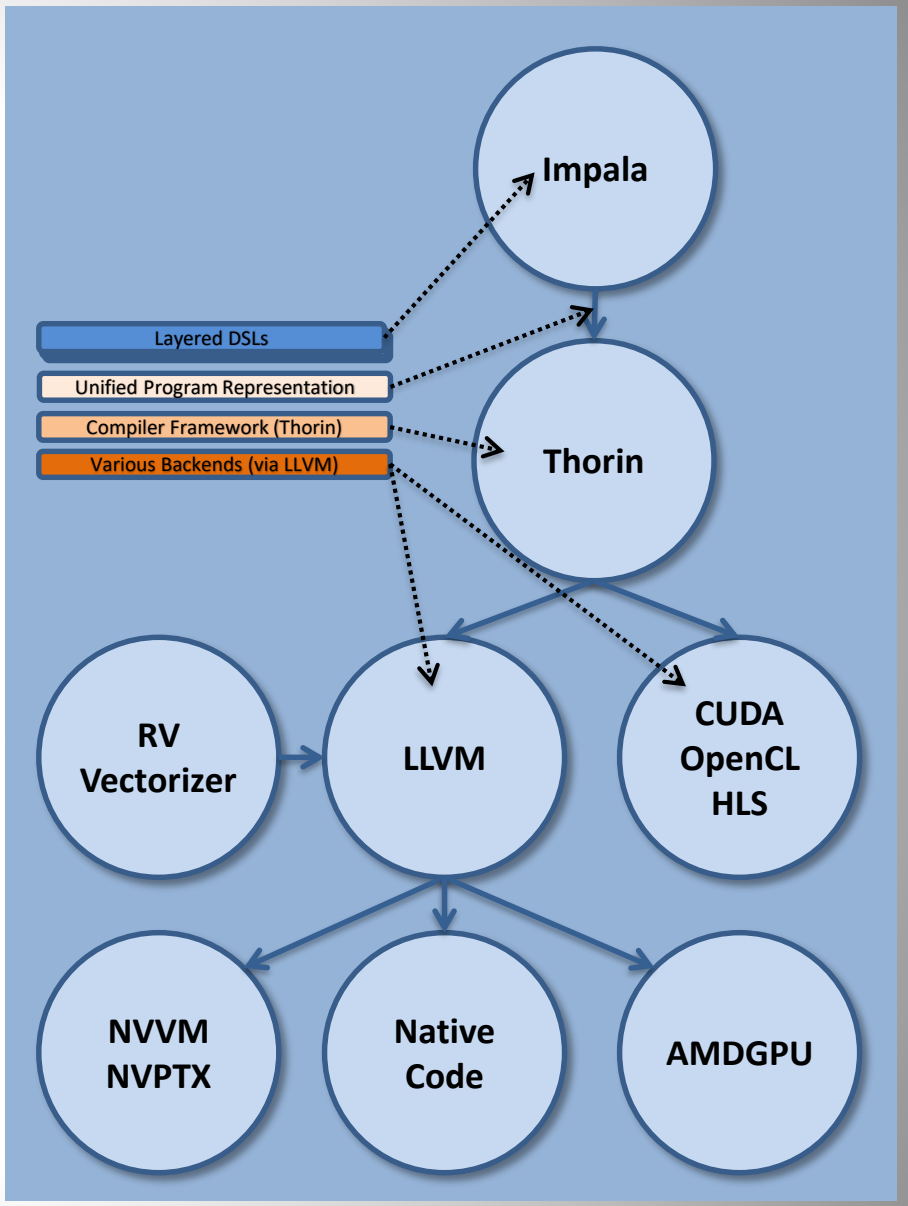
Our Approach

- AnyDSL framework



Compiler Framework

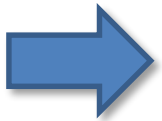
- Impala language (Rust dialect)
  - Functional & imperative language
- Thorin compiler [GPCE'15 ***best paper award**]
  - Higher-order functional IR [CGO'15]
    - Special optimization passes
    - No overhead during runtime
- Region vectorizer, extends WFV [CGO'11]
- LLVM-based back ends
  - Full compiler optimization passes
  - Multi-target code generation
    - NVVM, AMDGPU
    - CPUs, GPUs, Xeon Phis, FPGAs, …



Impala: A Base Language for DSL Embedding

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

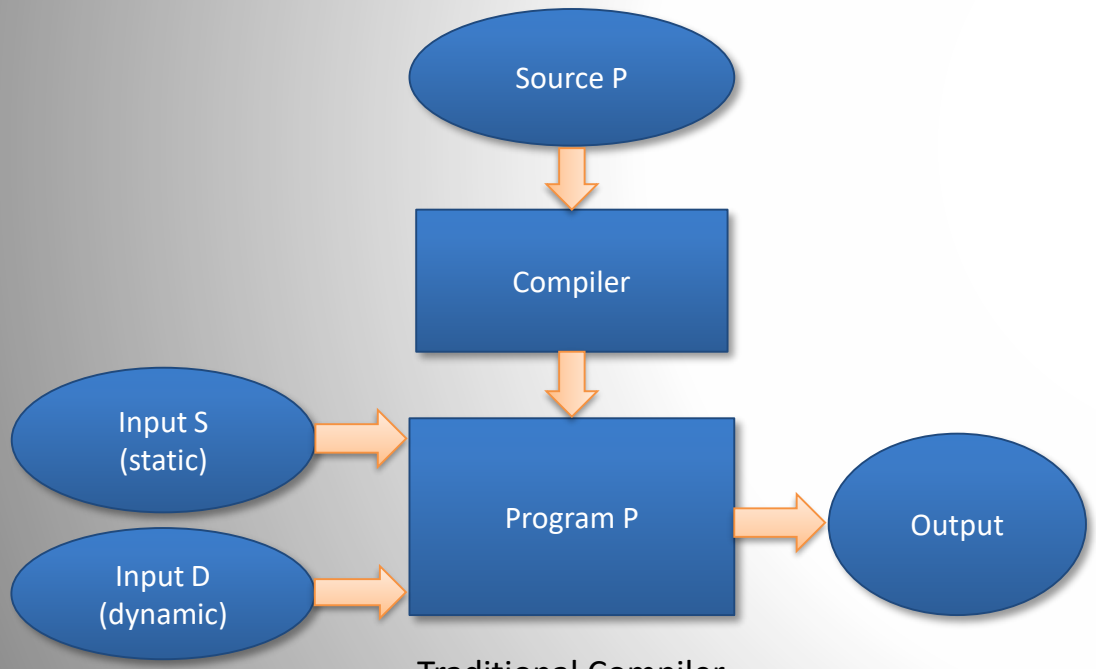
```
fn @(?n)dot(n: int,
            u: &[float],
            v: &[float]
            ) -> float {
  let mut sum = 0.0f;

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

  sum
}

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



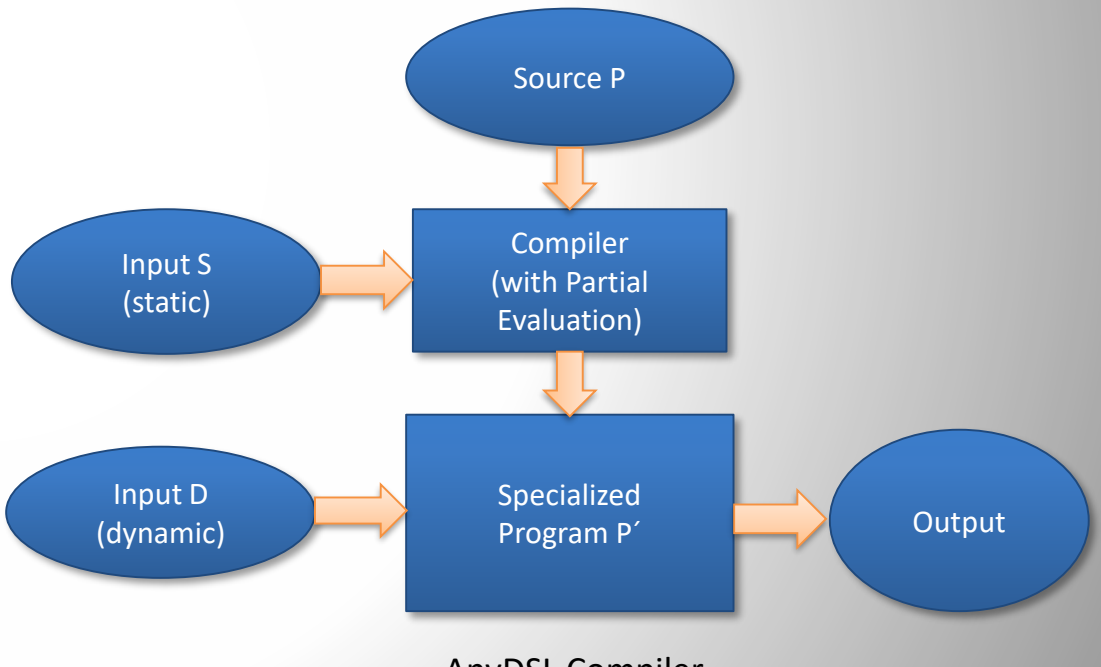
```
// specialized code for dot-call
result = 0;
result += a(0)*b(0);
result += a(1)*b(1);
result += a(2)*b(2);
```

AnyDSL Key Feature: Partial Evaluation (in a Nutshell)

- Left: Normal program execution
- Right: Execution with program specialization (PE)
  - PE as part of normal compilation process!!



Traditional Compiler



AnyDSL Compiler

Case Study: Image Processing [GPCE'15]

Stincilla – A DSL for Stencil Codes

https://github.com/AnyDSL/stincilla

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);
}
```

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
  - Allows for partial evaluation of function (via "@"):
    - Unrolls stencil
    - Propagates constants
    - Inlines function calls
  - Can control what data is used for PE
    - Also conditional PE
    - PE applied only where info is available to the compiler

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

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

  sum
}
```

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

  out
}
```

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

  out
}
```

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, out.height) {
    for x in range(0, out.width) {
      body(x, y);
    }
  }
}
```

Mapping to Target Hardware: CPU

- 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 whole-function vectorization

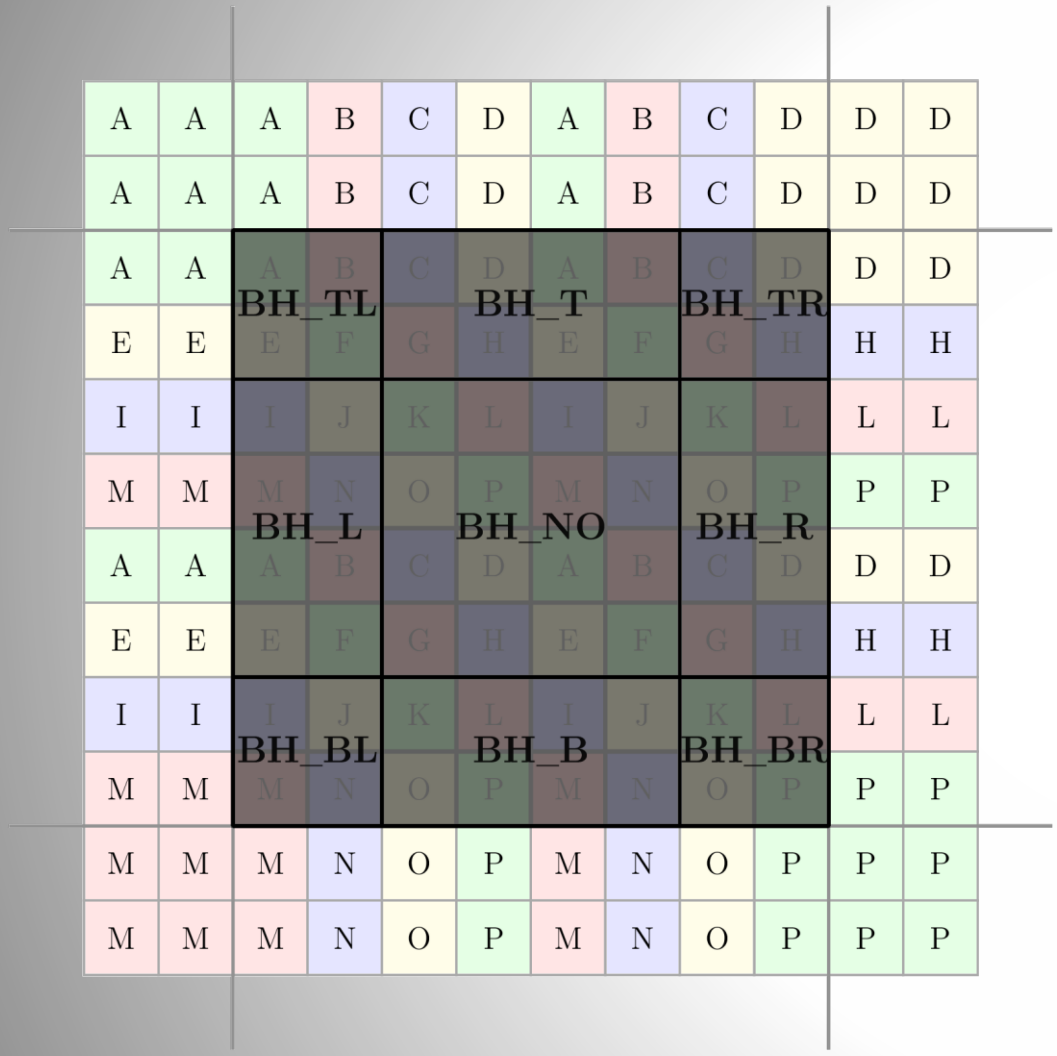
```
fn @iterate(img: Img, body: fn(int, int) -> ()) -> () {
  let thread_number = 4;
  let vector_length = 8;
  for y in parallel(thread_number, 0, img.height) {
    for x in vectorize(vector_length, 0, img.width) {
      body(x, y);
    }
  }
}
```

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);
  }
}
```

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 region variable (here only in horizontally)
  - Specialization discards unnecessary checks


```
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

```
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, out.height) {
      for x in inner_loop(L(region), U(region)) {
        ...
        body(x, y, region);
      }
    }
  }
}
```

Exploiting Boundary Handling: GPU

- Specialized implementation
  - unroll triggers image region specialization
  - Generates multiple GPU kernels for each image region

```
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);
    }
  }
}
```

Performance: Gaussian Blur Filter (Intel Haswell: Intel Iris 5100)

- Specialized implementation for
  - Given stencil (SS)
  - Boundary handling (BH)
  - Scratchpad memory (SM)

	OpenCL		Speedup
	Simple	Unrolled	
Gaussian	n/a	n/a	
SS	17.49	17.15	
SS + BH	17.00	16.87	
SS + SM	24.21	**12.84**	**~-25%**
OpenCV 2.4	18.55		
OpenCV 3.0 (ref.)	**16.61**		

Image of 4096x4096, kernel window size of 5x5, runtime in ms, OpenCV 2.4.12 & 3.00

Performance: Gaussian Blur Filter (Intel Haswell: Intel Core i5-4288U)

- Specialized implementation for
  - Given stencil (SS)
  - Boundary handling (BH)
  - Scratchpad memory (SM)
- Much better performance than hand-tuned OpenCV implementation
- More than **1500 LoC** for vectorized implementations in OpenCV

	CPU Simple	AVX Simple	Speedup
Gaussian	85.34	202.74	
SS	85.69	155.57	
SS + BH	23.56	23.23	
SS + BH + SM	16.67	**15.98**	**~-40%**
OpenCV 2.4	**27.21**		
OpenCV 3.0 (ref.)	**26.63**		

Image of 4096x4096, kernel window size of 5x5, runtime in ms, OpenCV 2.4.12 & 3.00

Performance: Gaussian Blur Filter (AMD Radeon R9 290X)

- Specialized implementation for
  - Given stencil (SS)
  - Scratchpad memory (SM)
  - Boundary handling (BH)

	SPIR		OpenCL		Speedup
	Simple	Unrolled	Simple	Unrolled	
Gaussian	n/a	n/a	n/a	n/a	
SS	1.02	0.97	1.02	0.97	
SS + BH	1.05	0.99	1.04	0.99	
SS + SM	0.82	**0.76**	0.82	**0.75**	**~-50%**
OpenCV 2.4	**0.89**				
OpenCV 3.0 (ref.)	**1.42**				

Image of 4096x4096, kernel window size of 5x5, runtime in ms, OpenCV 2.4.12 & 3.00, Crimson 15.11

Performance: Gaussian Blur Filter (NVIDIA GTX 970)

- Specialized implementation for
  - Given stencil (SS)
  - Scratchpad memory (SM)
  - Boundary handling (BH)

	NVVM		OpenCL		Speedup
	Simple	Unrolled	Simple	Unrolled	
Gaussian	n/a	n/a	n/a	n/a	
SS	2.34	2.26	2.34	2.26	
SS + BH	2.36	2.30	2.38	2.28	
SS + SM	1.61	**1.28**	1.67	**1.27**	**~-45%**
OpenCV 2.4	**2.24**		**2.17**		
OpenCV 3.0 (ref.)	**2.24**		**2.11**		

Image of 4096x4096, kernel window size of 5x5, runtime in ms, OpenCV 2.4.12 & 3.00, CUDA 7.5

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
}
```

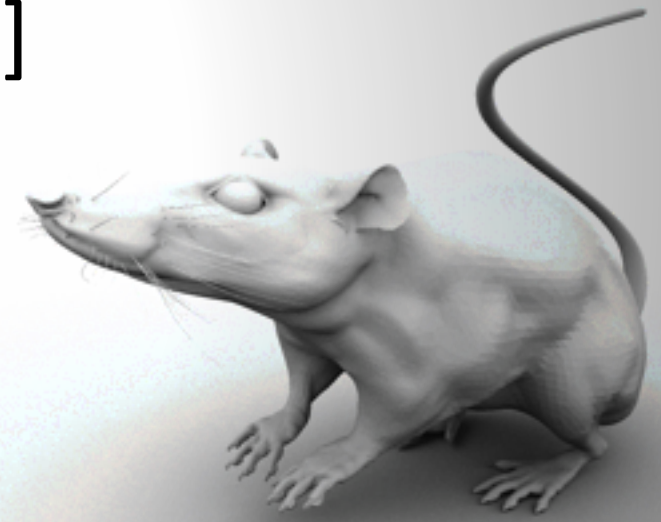
Machine expert

```
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 Traversal [GPCE'17]

RaTrace – A DSL for Ray Traversal

https://github.com/AnyDSL/traversal



Ray Traversal

- *Ray traversal* is the process of traversing an acceleration structure in order to find the intersection of a ray and a mesh

- High performance implementations have been developed for each hardware platform
  - They are written in extremely low-level code
  - They take advantage of every hardware feature
  - Often “write-only code”

- But the *essence of the traversal algorithm* is the same

Generic Ray Tracing Implementation (shortened)

```
for tmin, tmax, org, dir, record_hits in iterate_rays(ray_list, hit_list, ray_count) {
  // Allocate a stack for the traversal
  let stack = allocate_stack();

  // Traversal loop
  stack.push_top(root, tmin);
  while !stack.is_empty() {
    let node = stack.top();

    // Step 1: Intersect children and update the stack
    for min, max, hit_child in iterate_children(node, stack) {
      intersect_ray_box(org, idir, tmin, t, min, max, hit_child);
    }

    // Step 2: Intersect the leaves
    while is_leaf(stack.top()) {
      let leaf = stack.pop();

      for id, tri in iterate_triangles(leaf, tris) {
        let (mask, t0, u0, v0) = intersect_ray_tri(org, dir, tmin, t, tri);
        t      = select(mask, t0, t);
        u      = select(mask, u0, u);
        v      = select(mask, v0, v);
        tri_id = select(mask, id, tri_id);
      }

      stack.pop();
    }
  }
  record_hits(tri_id, t, u, v);
}
```

The iteration over the set of rays is abstract:
- Can be vectorized with AVX instructions on the CPU
- Can be done in single-ray fashion on the GPU
- Returns abstract function (record_hits) to handle rays

The iteration over the children of a node is abstract:
- Different branching factors &
- Different traversal heuristics implemented

The iteration over the triangles in a leaf is abstract:
- Uses a different data layout, depending on the target hardware
- May use an indexed representation to save some space

Mapping: CPU with AVX – Iterating Over Rays

Common part

```
struct Vec3 {x: Real, y: Real, z: Real}
// Abstraction: Iterate of all rays (may use single or packets of rays)
for tmin, tmax, org, dir, record_hit in iterate_rays(rays, hits, ray_count) { /* ... */ }
```

CPU mapping

```
static vector_size = 8;
type Real = simd[float * vector_size];

fn @iterate_rays(rays: &[Ray], hits: &mut[Hit], ray_count: int,
                 body: fn(tmin: Real, tmax: Real, org: Vec3, dir: Vec3, record_hit: HitFn) -> ()) -> () {
  for i in range_step(0, ray_count, vector_size) {
    // Convert ray from AoS to SoA*8 format
    let mut org: Vec3; let mut dir: Vec3; let mut tmin: Real; let mut tmax: Real;
    for in unroll(0, vector_size) {
      org.x(k) = rays(i + k).org.x;  org.y(k) = rays(i + k).org.y;  org.z(k) = rays(i + k).org.z;  tmin(k)  = rays(i + k).org.w;
      dir.x(k) = rays(i + k).dir.x;  dir.y(k) = rays(i + k).dir.y;  dir.z(k) = rays(i + k).dir.z;  tmax(k)  = rays(i + k).dir.w;
    }

    // Execute body with a specific function to record hit points
    body(tmin, tmax, org, dir, |tri, t, u, v| {
      for j in unroll(0, vector_size) {
        hits(i + j).tri_id = tri(j);
        hits(i + j).tmax = t(j);
        hits(i + j).u = u(j);
        hits(i + j).v = v(j);
      }
    });
  }
}
```

Mapping: GPU with NVVM – Iterating Over Rays

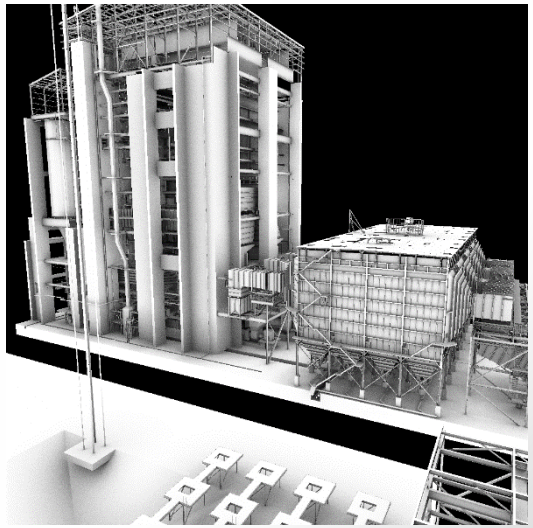
Common part

```
struct Vec3 {x: Real, y: Real, z: Real}
// Abstraction: Iterate of all rays (may use single or packets of rays)
for tmin, tmax, org, dir, record_hit in iterate_rays(rays, hits, ray_count) { /* ... */ }
```

GPU mapping

```
type Real = float;

fn @iterate_rays(rays: &[Ray], hits: &mut[Hit], ray_count: int,
                 body: fn(tmin: Real, tmax: Real, org: Vec3, dir: Vec3, record_hit: HitFn) -> ()) -> () {
  // Setup GPU iteration space (grid size and block size)
  let grid = (ray_count / block_h, block_h, 1); let block = (block_w, block_h, 1);

  acc.exec(grid, block, |exit| { // Triggers GPU code generation
    // Typical GPU conversion from grid/block to linear index
    let id = acc_tidx() + acc_bdimx()*(acc_tidy() + acc_bdimy()*(acc_bidx() + acc_gdimx()*acc_bidy()));
    if id > ray_count {
      exit()
    }

    // Loading of ray data (use GPU intrinsics for optimal memory access)
    let ray_ptr = &rays(id) as &[float];
    let ray0 = ldg4_f32(&ray_ptr(0));
    let ray1 = ldg4_f32(&ray_ptr(4));

    body(vec3(ray0(0), ray0(1), ray0(2)),
         vec3(ray1(0), ray1(1), ray1(2)), ray0(3), ray1(3), |tri, t, u, v| {
      *(&hits(id) as &mut simd[float * 4]) = simd[bitcast[float](tri), t, u, v]; // Optimized store
    });
  });
  acc.sync();
}
```

Performance Results



Performance Results

		CPU			GPU	
Scene	**Ray Type**	**Embree (icc)**	**Embree (clang)**	**Ours**	**Aila et al.**	**Ours**
San Miguel 7880K tris.	Primary	4.90	4.31	4.81 (-1.84%, +11.60%)	114.75	132.48 (+15.45%)
	Shadow	4.35	3.90	4.17 (-4.14%, +6.92%)	101.30	122.54 (+20.97%)
	Random	1.52	1.38	1.49(-1.97%, +7.97%)	90.63	105.27 (+16.15%)
Sibenik 75K tris.	Primary	18.17	15.06	17.80 (-2.04%, +18.19%)	336.47	405.01 (+20.37%)
	Shadow	23.93	19.54	23.48 (-1.88%, +20.16%)	459.04	560.44 (+22.09%)
	Random	2.48	2.29	2.39 (-3.63%, +4.37%)	154.83	177.48 (+14.63%)
Sponza 262K tris.	Primary	7.77	6.60	7.46 (-3.99%, +13.03%)	189.45	223.34 (+17.89%)
	Shadow	10.13	8.13	9.82 (-3.06%, **+20.79%**)	304.17	359.47 (+18.18%)
	Random	2.62	2.41	2.52 (-3.82%, +4.56%)	121.46	141.20 (+16.25%)
Conference 331K tris.	Primary	27.43	23.24	26.80 (-2.30%, +15.32%)	427.96	514.26 (+20.17%)
	Shadow	20.00	16.98	19.96 (**-0.70%**, +16.96%)	358.66	433.65 (+20.91%)
	Random	5.01	4.61	4.82 (-3.79%, +4.56%)	169.07	181.16 (**+7.15%**)
Power Plant 12759K tris.	Primary	8.53	7.65	8.43 (-1.17%, +10.20%)	261.13	301.57 (+15.49%)
	Shadow	8.22	7.41	7.77 (**-5.47%**, +4.86%)	301.02	339.34 (+12.73%)
	Random	4.49	4.22	4.40 (-2.00%, **+4.27%**)	193.34	242.22 (**+25.28%**)

Performance Results

		CPU			GPU	
Scene	**Ray Type**	**Embree (icc)**	**Embree (clang)**	**Ours**	**Aila et al.**	**Ours**
San Miguel 7880K tris.	Primary	4.90	4.31	4.81 (-1.84%, +11.60%)	114.75	132.48 (+15.45%)
	Shadow	4.35	3.90	4.17 (-4.14%, +6.92%)	101.30	122.54 (+20.97%)
	Random	1.52	1.38	1.49(-1.97%, +7.97%)	90.63	105.27 (+16.15%)
Sibenik 75K tris.	Primary	18.17	15.06	17.80 (-2.04%, +18.19%)	336.47	405.01 (+20.37%)
	Shadow	23.93	19.54	23.48 (-1.88%, +20.16%)	459.04	560.44 (+22.09%)
	Random	2.48	2.29	2.39 (-3.63%, +4.37%)	154.83	177.48 (+14.63%)
Sponza 262K tris.	Primary	7.77	6.60	7.46 (-3.99%, +13.03%)	189.45	223.34 (+17.89%)
	Shadow	10.13	8.13	9.82 (-3.06%, **+20.79%**)	304.17	359.47 (+18.18%)
	Random	2.62	2.41	2.52 (-3.82%, +4.56%)	121.46	141.20 (+16.25%)
Conference 331K tris.	Primary	27.43	23.24	26.80 (-2.30%, +15.32%)	427.96	514.26 (+20.17%)
	Shadow	20.00	16.98	19.96 (**-0.70%**, +16.96%)	358.66	433.65 (+20.91%)
	Random	5.01	4.61	4.82 (-3.79%, +4.56%)	169.07	181.16 (**+7.15%**)
Power Plant 12759K tris.	Primary	8.53	7.65	8.43 (-1.17%, +10.20%)	261.13	301.57 (+15.49%)
	Shadow	8.22	7.41	7.77 (**-5.47%**, +4.86%)	301.02	339.34 (+12.73%)
	Random	4.49	4.22	4.40 (-2.00%, **+4.27%**)	193.34	242.22 (**+25.28%**)

Performance Results

		CPU			GPU	
Scene	**Ray Type**	**Embree (icc)**	**Embree (clang)**	**Ours**	**Aila et al.**	**Ours**
San Miguel 7880K tris.	Primary	4.90	4.31	4.81 (-1.84%, +11.60%)	114.75	132.48 (+15.45%)
	Shadow	4.35	3.90	4.17 (-4.14%, +6.92%)	101.30	122.54 (+20.97%)
	Random	1.52	1.38	1.49(-1.97%, +7.97%)	90.63	105.27 (+16.15%)
Sibenik 75K tris.	Primary	18.17	15.06	17.80 (-2.04%, +18.19%)	336.47	405.01 (+20.37%)
	Shadow	23.93	19.54	23.48 (-1.88%, +20.16%)	459.04	560.44 (+22.09%)
	Random	2.48	2.29	2.39 (-3.63%, +4.37%)	154.83	177.48 (+14.63%)
Sponza 262K tris.	Primary	7.77	6.60	7.46 (-3.99%, +13.03%)	189.45	223.34 (+17.89%)
	Shadow	10.13	8.13	9.82 (-3.06%, **+20.79%**)	304.17	359.47 (+18.18%)
	Random	2.62	2.41	2.52 (-3.82%, +4.56%)	121.46	141.20 (+16.25%)
Conference 331K tris.	Primary	27.43	23.24	26.80 (-2.30%, +15.32%)	427.96	514.26 (+20.17%)
	Shadow	20.00	16.98	19.96 (**-0.70%**, +16.96%)	358.66	433.65 (+20.91%)
	Random	5.01	4.61	4.82 (-3.79%, +4.56%)	169.07	181.16 (**+7.15%**)
Power Plant 12759K tris.	Primary	8.53	7.65	8.43 (-1.17%, +10.20%)	261.13	301.57 (+15.49%)
	Shadow	8.22	7.41	7.77 (**-5.47%**, +4.86%)	301.02	339.34 (+12.73%)
	Random	4.49	4.22	4.40 (-2.00%, **+4.27%**)	193.34	242.22 (**+25.28%**)

Performance Results

		CPU			GPU	
Scene	**Ray Type**	**Embree (icc)**	**Embree (clang)**	**Ours**	**Aila et al.**	**Ours**
San Miguel 7880K tris.	Primary	4.90	4.31	4.81 (-1.84%, +11.60%)	114.75	132.48 (+15.45%)
	Shadow	4.35	3.90	4.17 (-4.14%, +6.92%)	101.30	122.54 (+20.97%)
	Random	1.52	1.38	1.49(-1.97%, +7.97%)	90.63	105.27 (+16.15%)
Sibenik 75K tris.	Primary	18.17	15.06	17.80 (-2.04%, +18.19%)	336.47	405.01 (+20.37%)
	Shadow	23.93	19.54	23.48 (-1.88%, +20.16%)	459.04	560.44 (+22.09%)
	Random	2.48	2.29	2.39 (-3.63%, +4.37%)	154.83	177.48 (+14.63%)
Sponza 262K tris.	Primary	7.77	6.60	7.46 (-3.99%, +13.03%)	189.45	223.34 (+17.89%)
	Shadow	10.13	8.13	9.82 (-3.06%, **+20.79%**)	304.17	359.47 (+18.18%)
	Random	2.62	2.41	2.52 (-3.82%, +4.56%)	121.46	141.20 (+16.25%)
Conference 331K tris.	Primary	27.43	23.24	26.80 (-2.30%, +15.32%)	427.96	514.26 (+20.17%)
	Shadow	20.00	16.98	19.96 (**-0.70%**, +16.96%)	358.66	433.65 (+20.91%)
	Random	5.01	4.61	4.82 (-3.79%, +4.56%)	169.07	181.16 (**+7.15%**)
Power Plant 12759K tris.	Primary	8.53	7.65	8.43 (-1.17%, +10.20%)	261.13	301.57 (+15.49%)
	Shadow	8.22	7.41	7.77 (**-5.47%**, +4.86%)	301.02	339.34 (+12.73%)
	Random	4.49	4.22	4.40 (-2.00%, **+4.27%**)	193.34	242.22 (**+25.28%**)

Code Complexity