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

Richard Membarth, Arsène Pérard-Gayot, Philipp Slusallek (Visual Computing)
Roland Leißa, Klaas Boesche, Sebastian Hack (Compiler)

Intel Visual Computing Institute at Saarland University
German Research Center for Artificial Intelligence (DFKI)
Many-Core Dilemma

Many-core HW is everywhere – but programming it is still hard
Future HW Architectures

Knights Landing Processor Architecture

- Up to 72 Intel Architecture cores based on Silvermont (Intel® Atom processor)
  - Four threads/core
  - Two 512b vector units/core
  - Up to 3x single thread performance improvement over KNC generation

- Full Intel® Xeon processor ISA compatibility through AVX-512 (except TSX)
- 6 channels of DDR4 2400 MHz -up to 384GB
- 36 lanes PCI Express® Gen 3
- 8/16GB of high-bandwidth on-package MCDRAM memory >500GB/sec
- 200W TDP

- On-chip Mesh Network
- On-chip 16 GB DRAM
- MCDRAM
- x86 PCIe3
- HUB
- TILE
- 2VPU
- Core
- 1MB L2

- 288 Threads
- 32x SIMD
- 3 TFLOPS in DP Float
- Std. CPU
- 141 GB/s
- Huge Cache/Mem
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
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, ...
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, HIPA², ...

- 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: A DSL for Ray Tracing

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

```cpp
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

```c++
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(Iterator 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;
};
```
// data structure
BVH<KdTree<Triangle>> hierarchy;
// corresponding intersector
BVHIntersector<KdTreeIntersector<
    SimpleTriangleIntersector>> intersector;
Example: Traversal

```cpp
template<class ElemIssect>  // nested element intersector
    // models the KdTree concept
    unsigned int size,   // size of the ray packet
def class KdTree,
    bool commonOrg,   // common ray origin?
    bool computeInters>:  // intersection data needed?
void KdTree::intersect
    // the actual rays
    RayPacket<dim>& rayPacket,
    // the actual kd-tree
    KdTree<tree,
    // intersection defined
    ElemIssect::Intersection<dim>& r) // intersection defined
    // by the nested intersector
{
    typedef BitMask<dim> t_BitMask;
    typedef Packet<dim,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

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

    const t_Const tSplit = tSplitFactor *
        rayPacket.invDir(i).get(splitDimension);
    forChildConditionMask.setContainer(i, currentTMin(i) > tSplit(i)).getIntMask());
    nearChildConditionMask.setContainer(i, currentTMax(i) < tSplit(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

```c++
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

```java
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<PlaneIntersector> bvhIntersect;
RayTracingRenderer<PixelCenterSampler,
    DirectIlluminationIntegrator> renderer1;
renderer1.render<64>(scene, camera, fb, fb.getClipRegion(),
    bvh, bvhIntersect);

// 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> pointIntersect;
RayTracingRenderer<PixelCenterSampler,
    PointLoDIntegrator> renderer2;
renderer2.render<16>(scene, camera, fb, fb.getClipRegion(),
    tree, lodKdTreeIntersect);
```
Evaluation

Figure 3: The CONFERENCE scene rendered with RTIact at a 1024² 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.
Performance

• Preliminary Performance Comparison
  – Needed common denominator to be able to compare

<table>
<thead>
<tr>
<th></th>
<th>Sponza</th>
<th>Conference</th>
<th>Soda Hall</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenRT K</td>
<td>4.5</td>
<td>4.2</td>
<td>5.1</td>
</tr>
<tr>
<td>Manta K</td>
<td>4.7</td>
<td>4.2</td>
<td>5.4</td>
</tr>
<tr>
<td>RTfact K</td>
<td>6.8</td>
<td>6.4</td>
<td>6.5</td>
</tr>
<tr>
<td>Wald et al. [31] B</td>
<td>n/a</td>
<td>9.3</td>
<td>11.1</td>
</tr>
<tr>
<td>Manta B</td>
<td>4.5</td>
<td>4.8</td>
<td>5.6</td>
</tr>
<tr>
<td>Arauna B</td>
<td>13.2</td>
<td>11.3</td>
<td>n/a</td>
</tr>
<tr>
<td>RTfact B</td>
<td>13.1</td>
<td>11.6</td>
<td>11.4</td>
</tr>
</tbody>
</table>

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

Diagram:
- Computer Vision DSL
- Physics DSL
- Ray Tracing DSL
- Parallel Runtime DSL
- Layered DSLs
- AnyDSL Unified Program Representation
- AnyDSL Compiler Framework (Thorin)
- Various Backends (via LLVM)

Developer
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”
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
- Whole-function vectorizer [CGO’11]
- LLVM-based back ends
  - Full compiler optimization passes
  - Multi-target code generation
    - SPIR, NVVM
    - CPUs, GPUs, MICs, …
Impala: A Base Language for DSL Embedding

- Impala is an imperative & functional language
- A dialect of Rust (http://rust-lang.org)
- Partial evaluation is triggered by annotation @
  - Aggressive execution of all possible instructions at compile time

```rust
fn dot(n: int,
       u: &[float],
       v: &[float]
       ) -> float {
    let mut sum = 0.0f;
    for i in range(n) {
        sum += u(i)*v(i);
    }
    sum
}

// specialization at call-site
result = @dot(3, a, b);
```
Impala: Language Features

- Data types:
  - Signed: i8, i16, i32 (int), i64
  - Unsigned: u8, u16, u32, u64
  - Floating point: f32 (float), f64 (double)
  - Bool: bool; Void: ()

- Constant literals use type suffix, e.g. 5i8
- Mutable information as variable annotation:
  - let mut var;

- Support for arrays:
  - Definitive arrays: [f32 * 3]
  - Indefinite arrays: [f32]
Impala: Language Features

- Data allocation:
  - let arr: [f32] = ~[size:f32];

- Reference to data (pointer):
  - let ref: &[f32] = &arr;

- Support for structs:

```rust
struct Img {
  data : &[float],
  width : int,
  height : int
}

... let img = Img { data: ~[width*height:float],
  width: 4096,
  height: 4096};
```
Impala: Language Features

- Support for higher-order functions
  - Pass function as argument
    
    ```
    fn fun(int) -> int;
    fn bar(a: int, foo: fn(int) -> int) -> int {
      foo(a)
    }
    fn main(a: int) -> int {
      bar(42, fun)
    }
    ```

- Interaction with C++ code
  - Make function callable from C++: extern fn foo(...) {}
  - Declare C++ function: extern “C” { fn bar(...); }
Case Study: Image Processing

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

```rust
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

```rust
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 range(-half, half+1) {
        for j in range(-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
- Gaussian blur implementation using generic `apply_convolution`
- `iterate` function iterates over image (provided by machine expert)

```rust
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

```rust
defn 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
}
```
Sample DSL: Stencil Codes in Impala

- Higher level domain-specific code: DSL implementation
- Compiler exposes partial evaluation through @
  - Unrolls stencil
  - Propagates constants

```rust
def 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 (1)

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

```python
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 (2)

- 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

```rust
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 (3)

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

```rust
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
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

```rust
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 (3)

- Specialized implementation
- Mapping and scheduling: CPU & AVX
- outer_loop maps to parallel and inner_loop calls either range (CPU) or vectorize (AVX)

```rust
def iterate(img: Img, body: fn(int, int, int) -> ()) -> ()
    let offset = filter.size / 2;
    let L = [0, img.width - offset, offset];
    let U = [offset, img.width, img.width - offset];
    for region in range(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 (4)

- Specialized implementation
- Mapping and scheduling: CPU & AVX
- @ triggers image region specialization; $ prevents specialization across entire image

```rust
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 @range(0, 3) {
        for y in $range(0, out.height) {
            for x in $inner_loop(L(region), U(region)) @{
                ...
                body(x, y, region);
            }
        }
    }
}
```
Exploiting Boundary Handling (cont.)

- Specialized implementation
- Mapping and scheduling: GPU

```rust
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 range(0, 3) {
        let grid = (U(region) - L(region), img.height, 1);
        with nvvm(grid, (128, 1, 1)) {
            ...
            body(L(region) + x, y, region);
        }
    }
}
```
Exploiting Boundary Handling (cont.)

- Specialized implementation
  - Mapping and scheduling: GPU
  - `@` triggers image region specialization

```rust
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 @range(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)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>OpenCL Simple</th>
<th>OpenCL Unrolled</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td>17.49</td>
<td>17.15</td>
<td></td>
</tr>
<tr>
<td>SS + BH</td>
<td>17.00</td>
<td>16.87</td>
<td></td>
</tr>
<tr>
<td>SS + SM</td>
<td>24.21</td>
<td>12.84</td>
<td>~25%</td>
</tr>
<tr>
<td>OpenCV 2.4</td>
<td>18.55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenCV 3.0 (ref.)</td>
<td>16.61</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>CPU Simple</th>
<th>AVX Simple</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian</td>
<td>85.34</td>
<td>202.74</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td>85.69</td>
<td>155.57</td>
<td></td>
</tr>
<tr>
<td>SS + BH</td>
<td>23.56</td>
<td>23.23</td>
<td></td>
</tr>
<tr>
<td>SS + BH + SM</td>
<td>16.67</td>
<td>15.98</td>
<td>~-40%</td>
</tr>
<tr>
<td>OpenCV 2.4</td>
<td>27.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenCV 3.0 (ref.)</td>
<td>26.63</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th></th>
<th>SPIR Simple</th>
<th>SPIR Unrolled</th>
<th>OpenCL Simple</th>
<th>OpenCL Unrolled</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>~1.02</td>
</tr>
<tr>
<td>SS</td>
<td>1.02</td>
<td>0.97</td>
<td>1.02</td>
<td>0.97</td>
<td>~1.02</td>
</tr>
<tr>
<td>SS + BH</td>
<td>1.05</td>
<td>0.99</td>
<td>1.04</td>
<td>0.99</td>
<td>~1.04</td>
</tr>
<tr>
<td>SS + SM</td>
<td>0.82</td>
<td>0.76</td>
<td>0.82</td>
<td>0.75</td>
<td>~1.04</td>
</tr>
</tbody>
</table>

- 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)

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

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

```rust
fn iterate(img: Img, ...) {
    let grid = (img.width, img.height);
    nvm(grid, (128, 1, 1), || {
        ... body(x, y);
    })
}

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

let result = gaussian_blur(img);
```
Case Study: Ray Traversal

RaTrace – A DSL for Ray 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
  - Write-only code

- But the essence of the traversal algorithm is the same
for tmin, tmax, org, dir, record_hit 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.top();
            for id, v0, v1, v2 in iterate_triangles(leaf, tris) {
                intersect_ray_tri(org, dir, tmin, v0, v1, v2, |cond, t0, u0, v0| {
                    t = select(cond, t0, t);
                    u = select(cond, u0, u);
                    v = select(cond, v0, v);
                    tri = select(cond, id, tri);
                });
            }
            stack.pop();
        }
    }
    record_hit(tri, t, u, v);
}
struct Vec3 {x: Real, y: Real, z: Real}

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

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

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

  // Execute body with a specific function to record hit points
  @body(org, dir, tmin, tmax, |tri, t, u, v| {
    for j in range(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);
    }
  });
}
}
// Abstraction: Intersect children and update stack
for min, max, hit_box in iterate_children(nodes, stack.top(), stack) { /* ... */ }

fn iterate_children(nodes: &[Node], node_id: int, t: simd[float * 8],
                    stack: Stack, body: IntersectionFn) -> ()

    let tmin = stack.tmin();
    stack.pop();

    if any(t > tmin)
        let node = nodes(node_id);
        for i in range(0, 4) {
            if node.children(i) == 0 { break; }
            let min = Vec3(node.min_x(i), node.min_y(i), node.min_z(i));
            let max = Vec3(node.max_x(i), node.max_y(i), node.max_z(i));

            body(min, max, |tentry, texit| {
                let t = select(texit >= tentry, tentry, flt_max);
                if any(texit >= tentry) {
                    if any(stack.tmin() > t) {
                        stack.push_top(node.children(i), t)
                    } else {
                        stack.push(node.children(i), t)
                    }
                }
            });
        }
    }
Mapping: CPU with AVX – Iterating Over Triangles

// Abstraction: Iterate (sequentially) over all triangles in leaf node
for id, v0, v1, v2 in iterate_triangles(nodes, stack.top(), tris) { /* ... */ }

// Iterate over BVH-4 structure
fn iterate_triangles(nodes: &[Node], node_id: i32, mut tris: &[Vec4],
    body: fn (Intr, Vec3, Vec3, Vec3) -> () -> () {
    let mut tri_id = !node_id; // Negative node_id marks leafs, id then is index in vertex array
    while true {
        // Load triangle data (could use different representation here as well (e.g. indexed))
        // Should be better abstracted
        let v0 = *(tris(tri_id + 0) as &simd[f32 * 4]); let v1 = ...; let v2 = ...;

        @body(intr(loop_id),
            Vec3(Real(v0(0)), Real(v0(1)), Real(v0(2))), // Convert to SIMD
            Vec3(Real(v1(0)), Real(v1(1)), Real(v1(2))),
            Vec3(Real(v2(0)), Real(v2(1)), Real(v2(2))));

        if (movmskps128(v2) & 8) != 0 { break() } // Stop iteration at marked triangle
        tri_id += 3;
    }
}
// Abstraction: Iterate of all rays (may use single or packets of rays)
for org, dir, tmin, tmax, record_hit in iterate_rays(rays, hits, ray_count) { /* ... */ }

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

    acc(dev, grid, block, |exit| { // Triggered 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 (currently needs some GPU intrinsics for optimal memory access)
        let ray0 = ldg4_f32((&rays(id) as &[float])(0) as &simd[float * 4]); let ray1 = ...;

        @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 &simd[float * 4]) = simd[bitcast_i32_f32(tri), t, u, v]; // Optimized store
        });
    });
}
// Abstraction: Intersect children and update stack
for min, max, hit_box in iterate_children(nodes, stack.top(), stack) { /* ... */ }

fn iterate_children(mut nodes: &[Node], node_id: int, stack: Stack,
body: fn (min: Vec3, max: Vec3, hit_box: IntersectionFn) -> () -> () {
  let mut node_ptr = &nodes(node_id) as &[float];
  let bb0 = ldg4_f32(&node_ptr(0) as &simd[float * 4]); let bb1 = ...; let bb2 = ...;
  let mut chldn = ldg4_i32(&node_ptr(12) as &simd[int * 4]);

  // Intersect with both children of BVH-2 structure, sort and insert intersected BVH nodes
  body(vec3(bb0(0), bb0(2), bb1(0)), vec3(bb0(1), bb0(3), bb1(1)), |t00, t01| {
    body(vec3(bb1(2), bb2(0), bb2(2)), vec3(bb1(3), bb2(1), bb2(3)), |t10, t11| {
      let hit0 = t00 <= t01; let hit1 = t10 <= t11;
      if !hit0 && !hit1 { stack.pop(); }
      else {
        if hit0 && hit1 {
          if t00 < t10 {
            chldn(0) ^= chldn(1); chldn(1) ^= chldn(0); chldn(0) ^= chldn(1);
            stack.push(|i| { chldn(i) }, 2)
          } else { stack.push(|i| { if hit0 { chldn(0) } else { chldn(1) } }, 1) }
        }
      }
    });
  });
}
// Abstraction: Iterate (sequentially) over all triangles in leaf node
for id, v0, v1, v2 in iterate_triangles(nodes, stack.top(), tris) { /* ... */ }
Performance Results
## Performance Results

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

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

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

<table>
<thead>
<tr>
<th>Scene</th>
<th>Ray Type</th>
<th>CPU</th>
<th>GPU</th>
<th>Aila et al.</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Miguel 7880K tris.</td>
<td>Primary</td>
<td>Embree (icc)</td>
<td>4.90</td>
<td>4.81 (-1.84%, +11.60%)</td>
<td>114.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Embree (clang)</td>
<td>4.31</td>
<td>4.17 (-4.14%, +6.92%)</td>
<td>101.30</td>
</tr>
<tr>
<td></td>
<td>Shadow</td>
<td></td>
<td>4.35</td>
<td>3.90</td>
<td>4.17 (-1.97%, +7.97%)</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td>1.52</td>
<td>1.38</td>
<td>1.49 (-1.84%, +11.60%)</td>
</tr>
<tr>
<td>Sibenik 75K tris.</td>
<td>Primary</td>
<td></td>
<td>18.17</td>
<td>15.06</td>
<td>17.80 (-2.04%, +18.19%)</td>
</tr>
<tr>
<td></td>
<td>Shadow</td>
<td></td>
<td>23.93</td>
<td>19.54</td>
<td>23.48 (-1.88%, +20.16%)</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td>2.48</td>
<td>2.29</td>
<td>2.39 (-3.63%, +4.37%)</td>
</tr>
<tr>
<td>Sponza 262K tris.</td>
<td>Primary</td>
<td></td>
<td>7.77</td>
<td>6.60</td>
<td>7.46 (-3.99%, +13.03%)</td>
</tr>
<tr>
<td></td>
<td>Shadow</td>
<td></td>
<td>10.13</td>
<td>8.13</td>
<td>9.82 (-3.06%, +20.79%)</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td>2.62</td>
<td>2.41</td>
<td>2.52 (-3.82%, +4.56%)</td>
</tr>
<tr>
<td>Conference 331K tris.</td>
<td>Primary</td>
<td></td>
<td>27.43</td>
<td>23.24</td>
<td>26.80 (-2.30%, +15.32%)</td>
</tr>
<tr>
<td></td>
<td>Shadow</td>
<td></td>
<td>20.00</td>
<td>16.98</td>
<td>19.96 (-0.70%, +16.96%)</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td>5.01</td>
<td>4.61</td>
<td>4.82 (-3.79%, +4.56%)</td>
</tr>
<tr>
<td>Power Plant 12759K tris.</td>
<td>Primary</td>
<td></td>
<td>8.53</td>
<td>7.65</td>
<td>8.43 (-1.17%, +10.20%)</td>
</tr>
<tr>
<td></td>
<td>Shadow</td>
<td></td>
<td>8.22</td>
<td>7.41</td>
<td>7.77 (-5.47%, +4.86%)</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td>4.49</td>
<td>4.22</td>
<td>4.40 (-2.00%, +4.27%)</td>
</tr>
</tbody>
</table>
Using Halstead’s metric, our code is **4-10x faster to write** than Embree or Aila’s code.

There is a **clean separation** between the **core logic** and the **implementation details**.

Our code can easily be **extended** and **ported** to a different platform.

- Not possible for Embree or Aila’s code without a reimplementation.

**More to come!**

- Shading and renderer logic can also benefit from this approach.
- A lot of room left for performance optimizations.
Memory Mapping for Heterogeneous Systems

- Explicit mapping/unmapping of memory to
  - Devices: CPU & GPU
  - Address spaces: global, constant, texture, and shared memory

```plaintext
let img = ...
let out = ...

let gpu_tex = mmap<GPU0, TEX>(img);
let gpu_out = mmap<GPU0, GMEM>(out);

with nvvm(GPU0, grid, block) {
    body(tid_x, tid_y, gpu_tex, gpu_out);
}

munmap(gpu_tex);
munmap(gpu_out);
```

- Support for (real) unified memory on multiple devices
Conclusion & Future Work

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

- Sample DSLs
  - Stincilla: Stencil codes, image processing
  - RaTrace: Ray traversal kernels
  - Working on several utility DSLs (parallelization, scheduling, communication, etc.)

- Next steps
  - Other high-performance codes (e.g. simulation, data base, multi-agents, ...)
  - Hardware synthesis as a backend?
Thank you for your attention.
Questions?
Gaussian Blur Filter (Intel Xeon Phi 7120P)

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

- Intel’s SPIR compiler segfaults for non-trivial examples

<table>
<thead>
<tr>
<th></th>
<th>SPIR Simple</th>
<th>SPIR Unrolled</th>
<th>OpenCL Simple</th>
<th>OpenCL Unrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian</td>
<td>Segfault</td>
<td>Segfault</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>SS</td>
<td>Segfault</td>
<td>Segfault</td>
<td>4.69</td>
<td>8.29</td>
</tr>
<tr>
<td>SS + BH</td>
<td>Segfault</td>
<td>Segfault</td>
<td>4.44</td>
<td>7.77</td>
</tr>
<tr>
<td>SS + SM</td>
<td>Segfault</td>
<td>Segfault</td>
<td>7.33</td>
<td>6.90</td>
</tr>
<tr>
<td>OpenCV</td>
<td></td>
<td></td>
<td>10.96</td>
<td></td>
</tr>
</tbody>
</table>

Image of 4096x4096, kernel window size of 5x5, runtime in ms, OpenCV 2.4.9
Gaussian Blur Filter

- Hardware-specific mapping (Impala)

```python
fn iterate(img: Img, body: fn(int, int) -> () -> ()
    let unroll = N;
    let grid  = (img.width, img.height / unroll, 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() * unroll;

        for i in @range(0, unroll) {
            body(x, y + i * nvvm_ntid_y());
        }
    }
}
```
Outlook

- Original Work on Real-Time Ray-Tracing
  - Low-level coding (intrinsics, at assembly level)
    - Highly time-consuming
  - High-level coding with C++ Template Metaprogramming [RTfact, 2008]
    - Excellent performance but hard to write and read
- Goal: Develop high-level DSL for Ray-Tracing (and Rasterization)
  - Flexible mapping onto different HW architectures (GPUs, CPUs, HSA, MICs, etc.)
- Vision: Real-time virtual production on stage (EU project “Dreamspace”)
  - With real-time global illumination and special effects