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.
Still State-of-the-Art ...

Math

Pseudo-Code

C sequential

X1 \subseteq \{ C, \text{OpenMP, OpenACC, CUDA, OpenCL, OpenCL4X, OpenCL4Y, ... } \}

abstr\text{act maintainable readable portable "slow"}

\text{manual}

\text{automated}

\text{concrete fast}
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

Layered DSLs

Unified Program Representation

Compiler Framework (Thorin)

Various Backends (via LLVM)

Developer

Computer Vision DSL
Physics DSL
Ray Tracing DSL
Parallel Runtime DSL

...
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, 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

```rust
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);
```
**Case Study:** Image Processing

[GPCE’15, OOPSLA’18]

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

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

```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]];

    for x, y in iterate(out) {
        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
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
}
```
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: Image,
    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
}
```
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);
        }
    }
}
```
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

```haskell
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);
            }
        }
    }
}
```
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

```plaintext
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)**

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>D</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>I</td>
<td>I</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>B H</td>
<td>B H</td>
<td>B H</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>I</td>
<td>I</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>B H</td>
<td>B H</td>
<td>B H</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
</tbody>
</table>

- **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

```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: 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)

```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 unroll(0, 3) {
        for y in outer_loop(0, img.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
  - Speedup over OpenCV: 25% (Intel GPU), 50% (AMD GPU), 45% (NVIDIA GPU)

```javascript
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);
        }
    }
}
```
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

```scala
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);
      }
    }
  }
}
```
Other Domains [OOPSLA‘18]

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

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

Genome Sequence Alignment
- SeqAn: -19% to -7%
- NVBIO: -8% to -2%

Ray Traversal
Separation of Concerns

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

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

```rust
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
}
```

```rust
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
### Features

<table>
<thead>
<tr>
<th>OptiX (NVIDIA)</th>
<th>Rodent</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GPU only</td>
<td>NVIDIA &amp; AMD GPUs</td>
</tr>
<tr>
<td>Generates megakernel (MK)</td>
<td>Megakernel &amp; wavefront (WF)</td>
</tr>
<tr>
<td>Not easy to extend (closed source)</td>
<td>Open source</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Embree + ispc (Intel)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>amd64 only</td>
<td>amd64 &amp; ARM support</td>
</tr>
<tr>
<td>Low-level, write-only code</td>
<td>High-level, textbook style code</td>
</tr>
</tbody>
</table>
Test Scenes
Performance Results

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

<table>
<thead>
<tr>
<th>Scene</th>
<th>CPU (Intel&lt;sup&gt;TM&lt;/sup&gt; i7 6700K)</th>
<th>GPU (NVIDIA&lt;sup&gt;TM&lt;/sup&gt; Titan X)</th>
<th>GPU (AMD&lt;sup&gt;TM&lt;/sup&gt; R9 Nano)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rodent&lt;sup&gt;WF&lt;/sup&gt;</td>
<td>Embree&lt;sup&gt;WF&lt;/sup&gt;</td>
<td>Rodent&lt;sup&gt;MK&lt;/sup&gt;</td>
</tr>
<tr>
<td>Living Room</td>
<td>9.77 (+23%)</td>
<td>7.94</td>
<td>38.59 (+25%)</td>
</tr>
<tr>
<td>Bathroom</td>
<td>6.65 (+13%)</td>
<td>5.90</td>
<td>27.06 (+31%)</td>
</tr>
<tr>
<td>Bedroom</td>
<td>7.55 (+4%)</td>
<td>7.24</td>
<td>30.25 (+9%)</td>
</tr>
<tr>
<td>Dining Room</td>
<td>7.08 (+1%)</td>
<td>7.01</td>
<td>30.07 (+5%)</td>
</tr>
<tr>
<td>Kitchen</td>
<td>6.64 (+12%)</td>
<td>5.92</td>
<td>22.73 (+2%)</td>
</tr>
<tr>
<td>Staircase</td>
<td>4.86 (+8%)</td>
<td>4.48</td>
<td>20.00 (+18%)</td>
</tr>
</tbody>
</table>

Msamples/s (higher is better). MK: Megakernel, WF: Wavefront.
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)
Scene Statistics: Compile Time & Shader Fusion

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

### Compilation times

<table>
<thead>
<tr>
<th>Room</th>
<th>Initial</th>
<th>Unique</th>
<th>Fused</th>
</tr>
</thead>
<tbody>
<tr>
<td>Living Room</td>
<td>19</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>Bathroom</td>
<td>16</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>Dining Room</td>
<td>58</td>
<td>51</td>
<td>28</td>
</tr>
<tr>
<td>Kitchen</td>
<td>129</td>
<td>95</td>
<td>19</td>
</tr>
<tr>
<td>Staircase</td>
<td>31</td>
<td>27</td>
<td>11</td>
</tr>
<tr>
<td>Bedroom</td>
<td>41</td>
<td>38</td>
<td>13</td>
</tr>
</tbody>
</table>

![Compilation Time Chart](image-url)
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
Performance Results

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

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

<table>
<thead>
<tr>
<th>Lang</th>
<th>HW</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatLab</td>
<td>Intel Core i5</td>
<td>6 min</td>
</tr>
<tr>
<td>AnyDSL</td>
<td>Tegra X1 CPU</td>
<td>2 s</td>
</tr>
<tr>
<td>AnyDSL</td>
<td>Tegra X1 GPU</td>
<td>36 ms</td>
</tr>
<tr>
<td>AnyDSL</td>
<td>Drive PX2 GPU</td>
<td>15 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lang</th>
<th>HW</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatLab</td>
<td>Intel Core i5</td>
<td>16.5 s</td>
</tr>
<tr>
<td>AnyDSL</td>
<td>Tegra X1 CPU</td>
<td>0.3 s</td>
</tr>
<tr>
<td>AnyDSL</td>
<td>Tegra X1 GPU</td>
<td>8 ms</td>
</tr>
<tr>
<td>AnyDSL</td>
<td>Drive PX2 GPU</td>
<td>12 ms</td>
</tr>
</tbody>
</table>
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!