AImotion Bavaria

AnyDSL Compiler Framework

A Partial Evaluation Framework for Programming High-Performance Libraries

Prof. Dr.-Ing. Richard Membarth

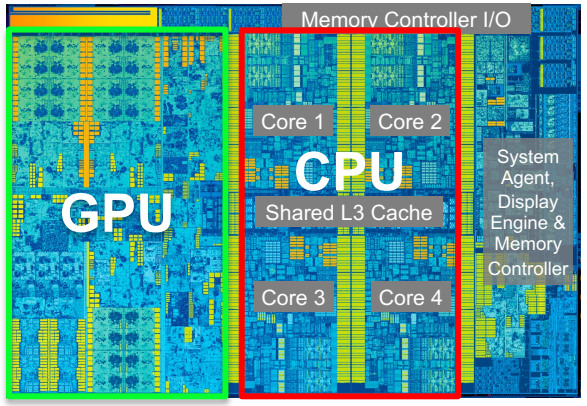
14.06.2023

DFKI

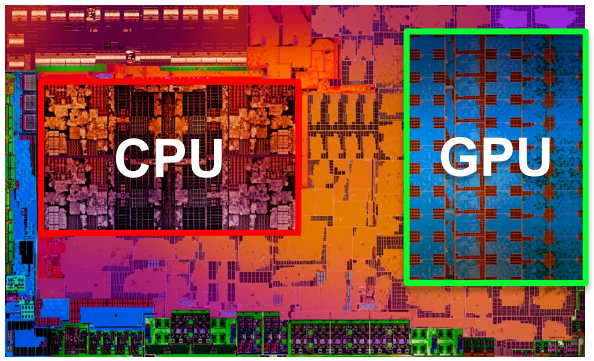
Technische Hochschule Ingolstadt

Many-Core Dilemma

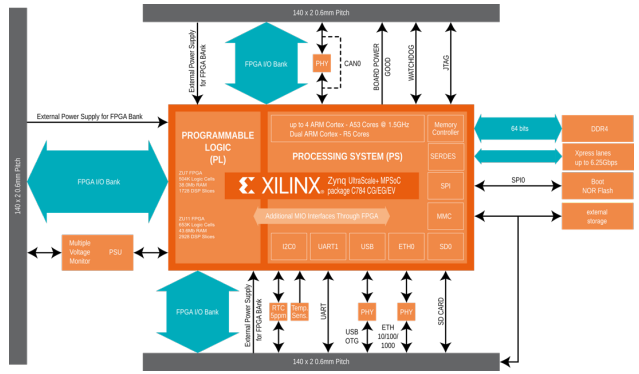
Many-Core Hardware is Everywhere – but Programming it is Still Hard



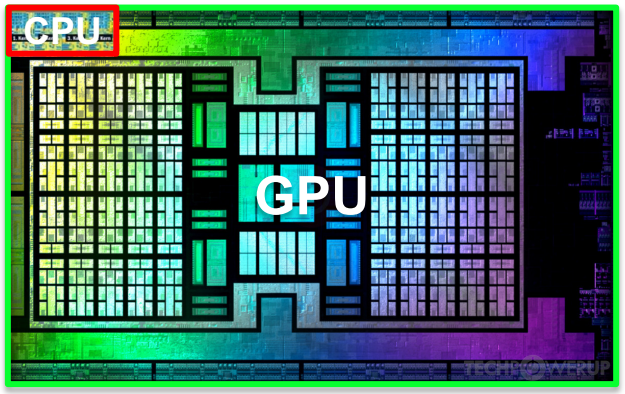
Intel Skylake (1.8B transistors)



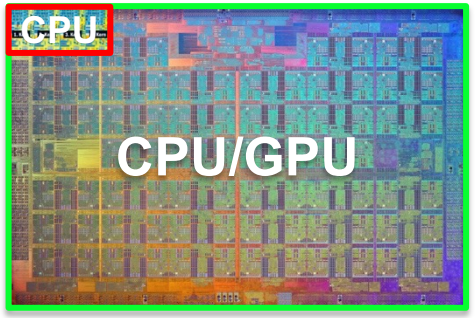
AMD Zen + Vega (4.9B transistors)



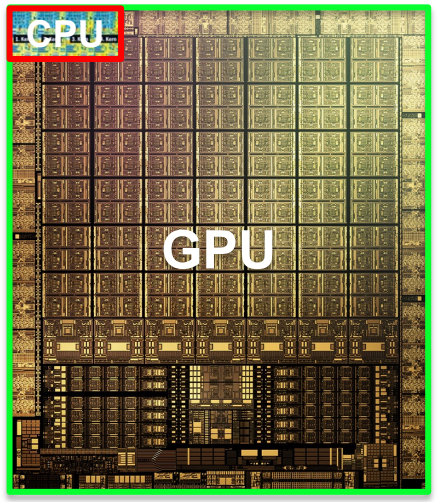
AMD / Xilinx Zync



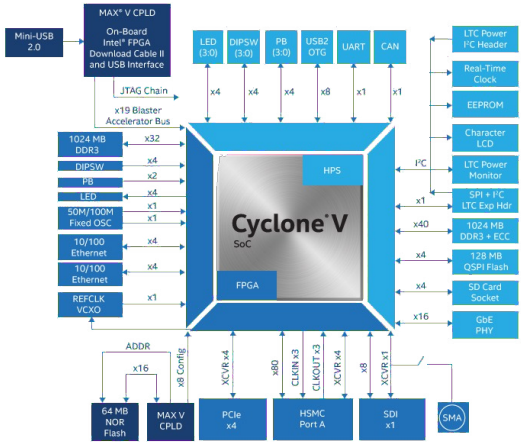
AMD Polaris (26.8B transistors)



Intel Knights Landing
(~8B transistors)

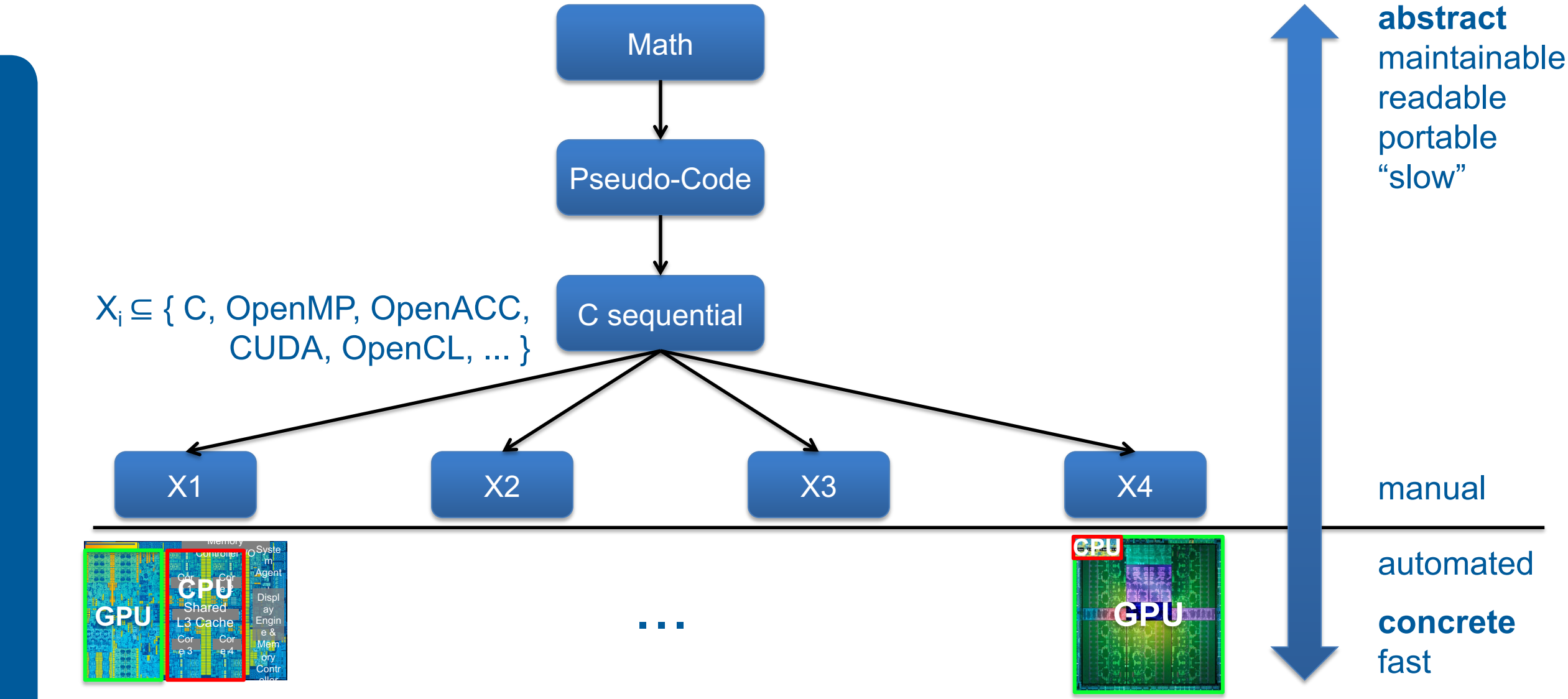


NVIDIA Ampere (28.3B transistors)



Intel / Altera Cyclone

Still State-of-the-Art …



What can we do?

Challenges: ***Productivity***, ***Portability***, *and* ***Performance***

- **Manual tuning**

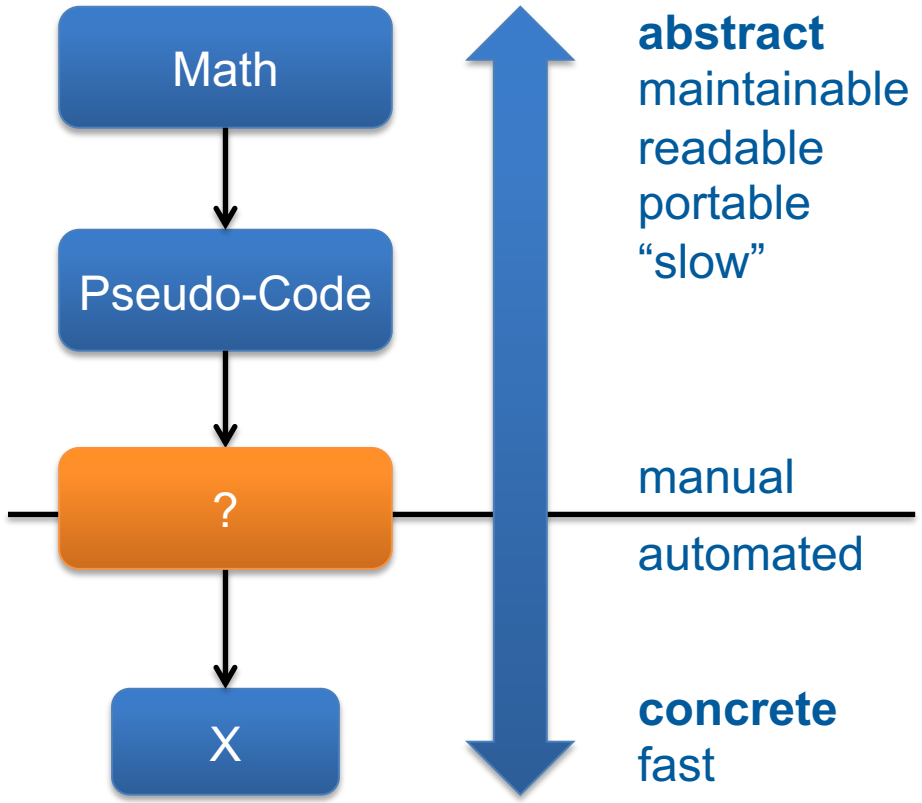
  rewrite code yourself
- **Annotations**

  use the compiler to rewrite code
- **Program generation**

  use a script to write code
- **Domain-specific languages**

  write compiler to rewrite program
- **Metaprogramming**

  write program to rewrite program



Program Specialization

C++ Template Metaprogramming, Partial Evaluation

C++ Template Metaprogramming

- **Pros**
  - Specialize programs at compile time
  - Faster execution times
- **Cons**
  - Duplicates code
  - Limited to primitive types
  - Compile-time constants only
  - Not typed
- **Famous for its error messages**

```
int pow(int b, int x) {
  if (x == 0) {
    return 1;
  } else {
    return b * pow(b, x-1);
  }
}

pow(base, 5);
```

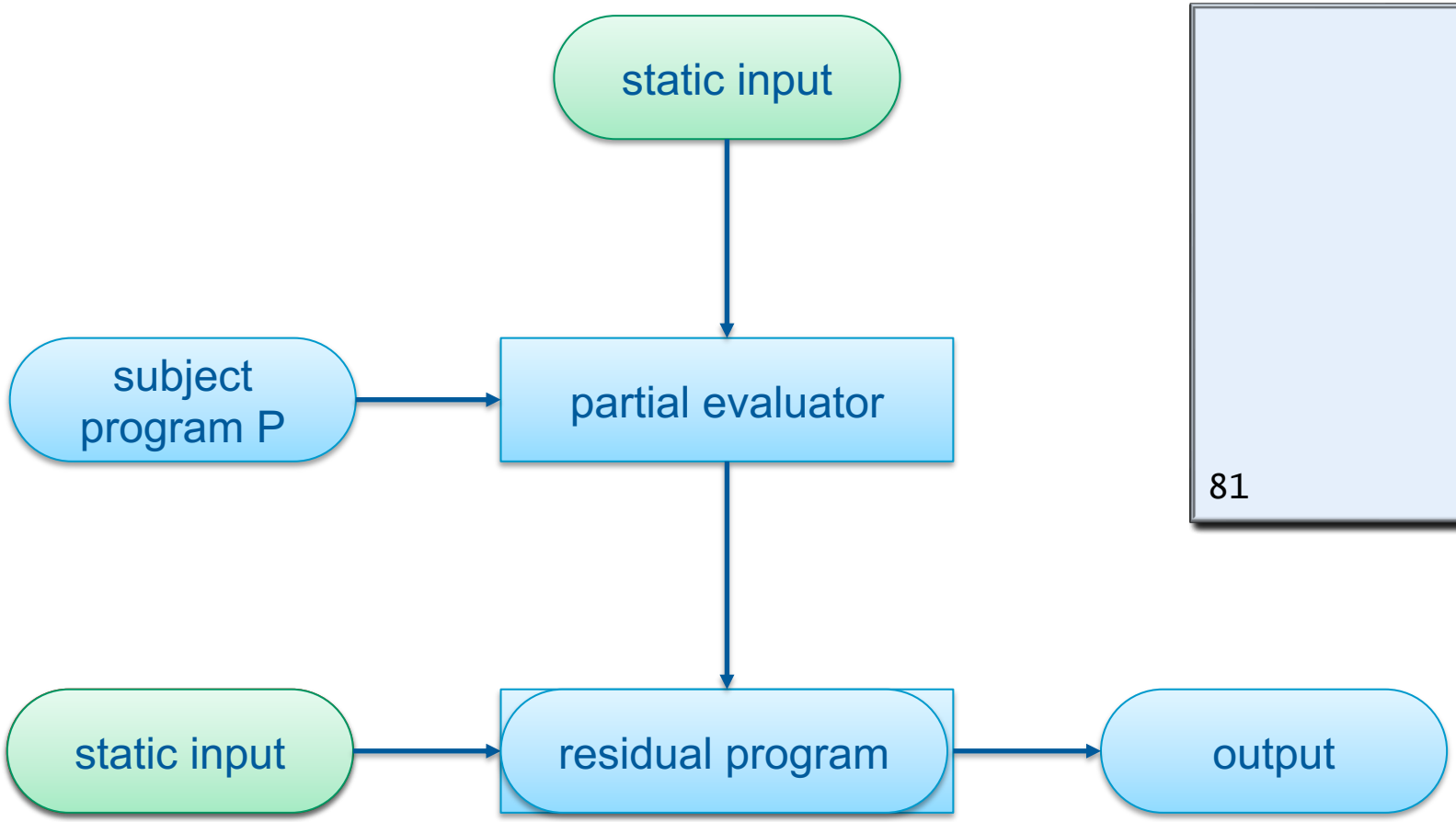
```
template<int x>
int power(int b) {
  return b * power<x-1>(b);
}
template<>
int power<1>(int b) {
  return b;
}
template<>
int power<0>(int b) {
  return 1;
}

power<5>(base);
```

Partial Evaluation

Normal Program Execution

Partial Evaluation

Execution with Program Specialization



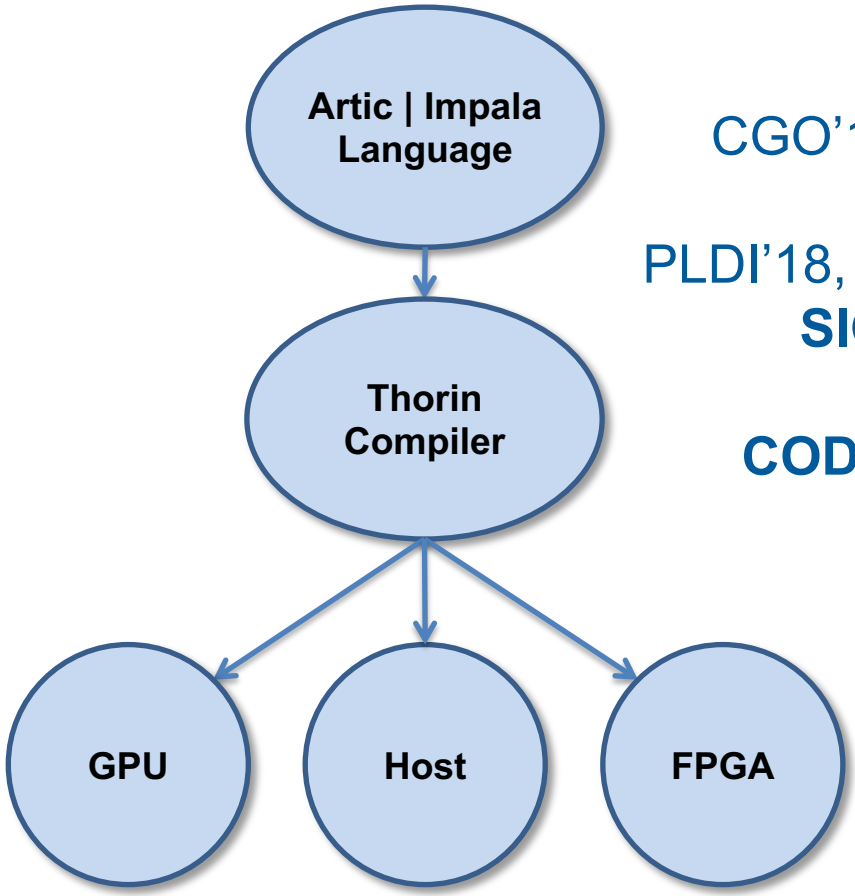
81

AnyDSL

Compiler Framework

- **Artic | Impala language: Rust dialect**
  - Functional & imperative
  - Filters to guide specialization
  - Triggered code generation
- **Thorin compiler**
  - Higher-order functional IR
  - Built-in partial evaluator
  - Multi-target code generation
    - Host: vectorization & parallelization
    - GPU: CUDA, OpenCL, AMDGPU, NVVM
    - FPGA: HLS

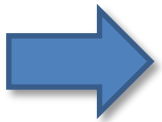
https://anydsl.github.io
https://github.com/AnyDSL



GPCE'14
CGO'15, **GPCE'15**
GPCE'17
PLDI'18, **OOPSLA'18**
SIGGRAPH'19
IPDPS'20
CODES+ISSS'20
FPT'21
ICS'22
IPDPS'23

- **A dialect of Rust (https://rust-lang.org)**
- **Specialization triggered when instantiating @-annotated functions [OOPSLA'18]**
- **Partial evaluation executes all possible instructions at compile time**

```
fn @(?n) dot(n: i32, u: &[f32], v: &[f32]) -> f32 {
  let mut sum = 0;

  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

Selected Application Domains

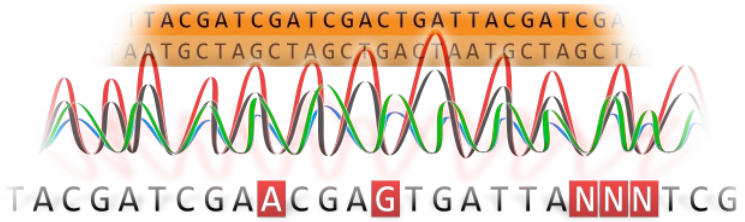
Stincilla: Image Processing [OOPSLA'18]

Blur (OpenCV): +40% (CPU) +50% (GPU)
Blur (Halide): +12% (CPU) +7% (GPU)
Harris (Halide): +37% (CPU) +44% (GPU)
Runs also on FGPA!

Rodent: Ray Tracing [SIGGRAPH'19]

Embree (CPU): +1% to +23% (WF)
OptiX (GPU): +2% to +31% (MK)
OptiX (GPU): +29% to +42% (WF)
Runs also on ARM and AMD GPU!

AnySeq: Genome Sequence Alignment [IPDPS'20]



SeqAn (CPU): -7.2% to +7.8%
Parasail (CPU): -2.1% to +90.0%
NVBIO (GPU): +5.3% to +10.4%
Runs also on FPGA!

AnySeq/GPU [ICS'22]

Median speedup of 19.2x vs. fastest available GPU implementations!

Case Study: Stencil Codes [GPCE'15, OOPSLA'18]

Stincilla – A DSL for Stencil Codes
https://github.com/AnyDSL/stincilla

Stincilla

Application Code

- **Applications**
  - Image processing
  - Scientific computing
  - Deep learning

- Example: Image processing, specifically Gaussian blur

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

Stincilla

Stencil Library

- Gaussian blur implementation using generic apply_stencil
- iterate function iterates over image (provided for each platform)

```
fn @gaussian_blur(img: Img) -> Img {
  let mut out = Img { data   = alloc(img.width*img.height),
                      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_stencil(x, y, img, filter);
  }

  out
}
```

Stincilla

Stencil Library

- Gaussian blur implementation using generic apply_stencil
- iterate function iterates over image (provided for each platform)
- for syntax: syntactic sugar for lambda function as last argument

```
fn @gaussian_blur(img: Img) -> Img {
  let mut out = Img { data   = alloc(img.width*img.height),
                      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_stencil(x, y, img, filter);
  });

  out
}
```

- **apply_stencil: generic function that applies a given stencil to a single pixel**
- **Partial evaluation**
  - Unrolls stencil
  - Propagates constants
  - Inlines function calls

```
fn @apply_stencil(x: i32, y: i32,
                  img: Img, filter: [f32]
                  ) -> f32 {
  let mut sum = 0:f32;
  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
}
```

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

```
fn @iterate(body: fn(i32, i32)->()) = @|img: Img| {
  for y in range(0, img.height) {
    for x in range(0, img.width) {
      body(x, y);
    }
  }
}
```

Stincilla

Mapping to Target Hardware (CPU)

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

```
fn @iterate(body: fn(i32, i32)->()) = @|img: Img| {
  let num_threads = 4;       // for 4 core CPU
  let num_vector_lanes = 8; // for 256-bit AVX

  for y in parallel(num_threads, 0, img.height) {
    for x in range_step(0, img.width, num_vector_lanes) {
      for lane in vectorize(num_vector_lanes) {
        body(x + lane, y);
      }
    }
  }
}
```

- **Scheduling & mapping provided by machine expert**
  - Exposed code generation, here: OpenCL
  - Last argument of opencl is function we generate GPU kernel code for

```
fn @iterate(body: fn(i32, i32)->()) = @|img: Img| {
  let grid  = (img.width, img.height, 1);
  let block = (32, 4, 1);

  with opencl(grid, block) {
    let x = get_global_id(0);
    let y = get_global_id(1);
    body(x, y);
  }
}
```

Stincilla

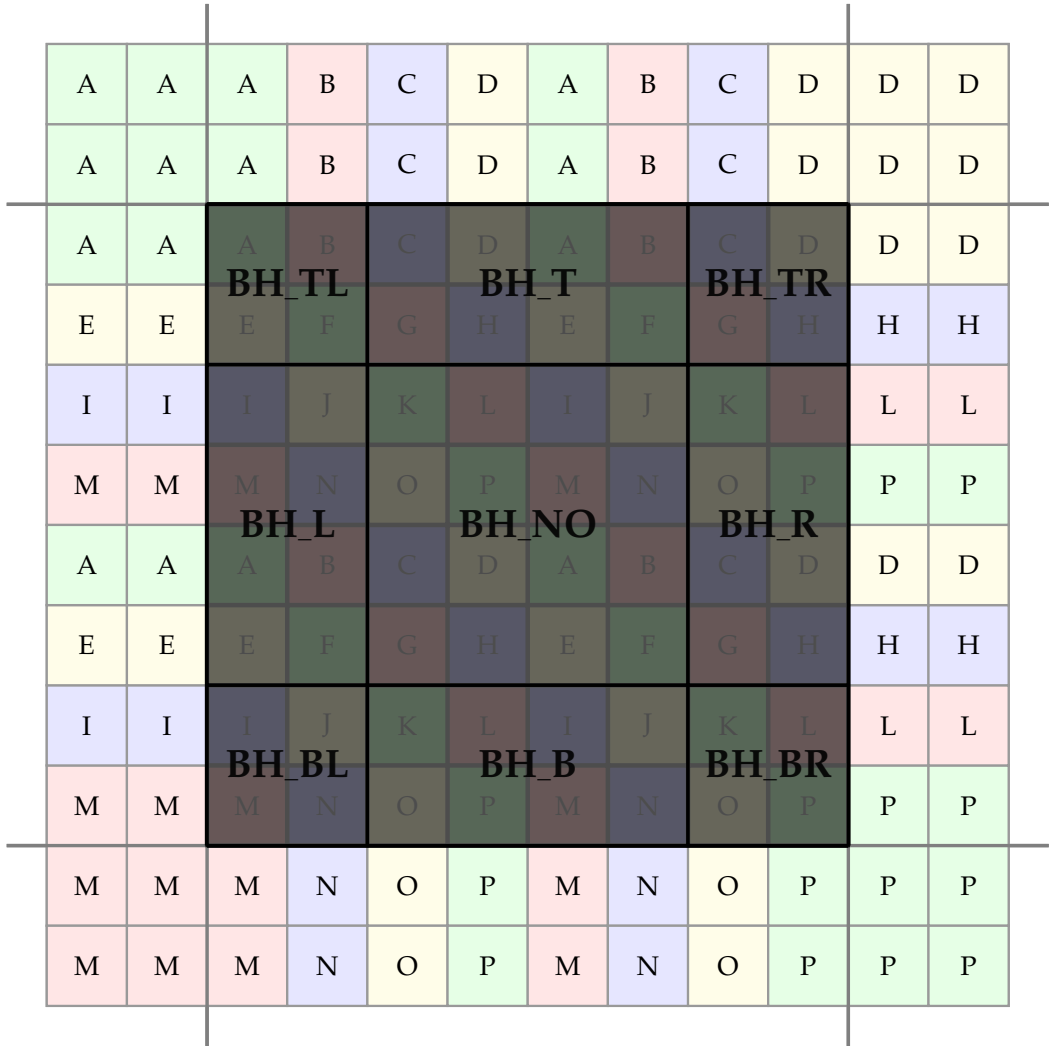
Mapping to Target Hardware (FPGA)

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

```
fn @iterate(body: fn(i32, i32)->()) = @|img: Img| {
  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);
      }
    }
  }
}
```

Stincilla

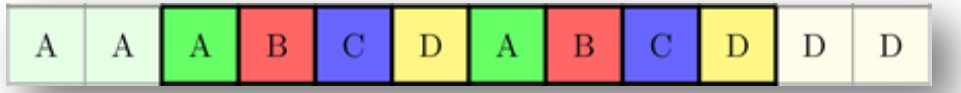
Exploiting Boundary Handling



- **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




```
fn @access(mut x: i32, y: i32,
           img: Img,
           region,
           bh_lower: fn(i32, i32) -> i32,
           bh_upper: fn(i32, i32) -> i32,
          ) -> f32 {
  if region == left  { x = bh_lower(x, 0); }
  if region == right { x = bh_upper(x, img.width); }
  img(x, y)
}
```

Stincilla

Specialized Boundary Handling: CPU & AVX

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

```
fn @iterate(body: fn(i32, i32, i32)->()) = @|img: Img| {
  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);
      }
    }
  }
}
```

Stincilla

Specialized Boundary Handling: GPU

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

```
fn @iterate(body: fn(i32, i32, i32)->()) = @|img: Img| {
  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);
    }
  }
}
```

Stincilla

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
}
```

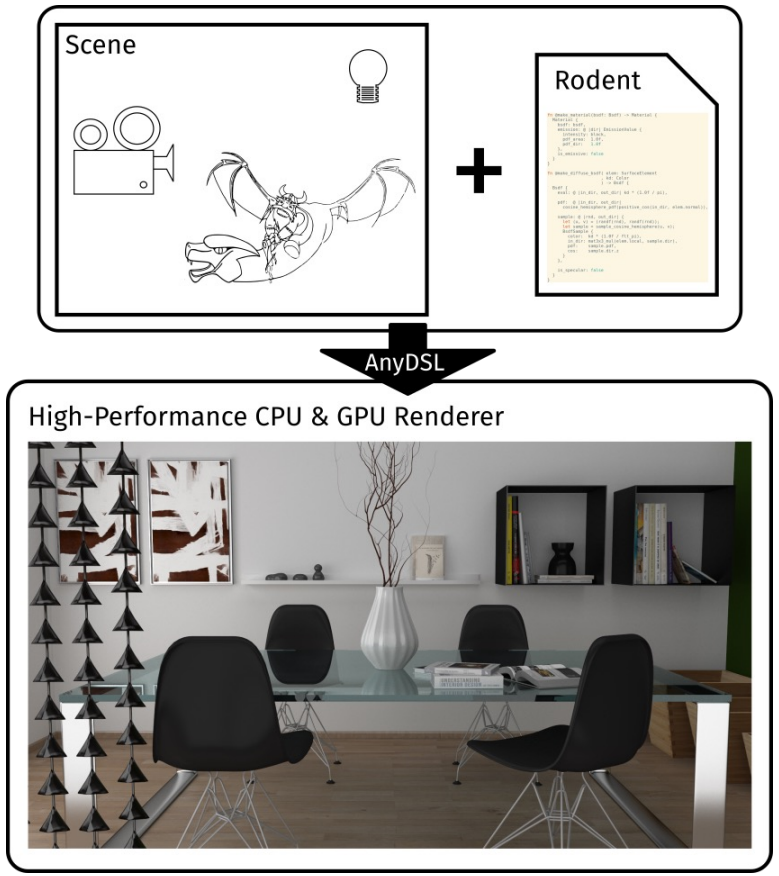
Machine expert

```
fn @iterate(body: fn(i32, i32)->()) = @|img: Img| {
  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



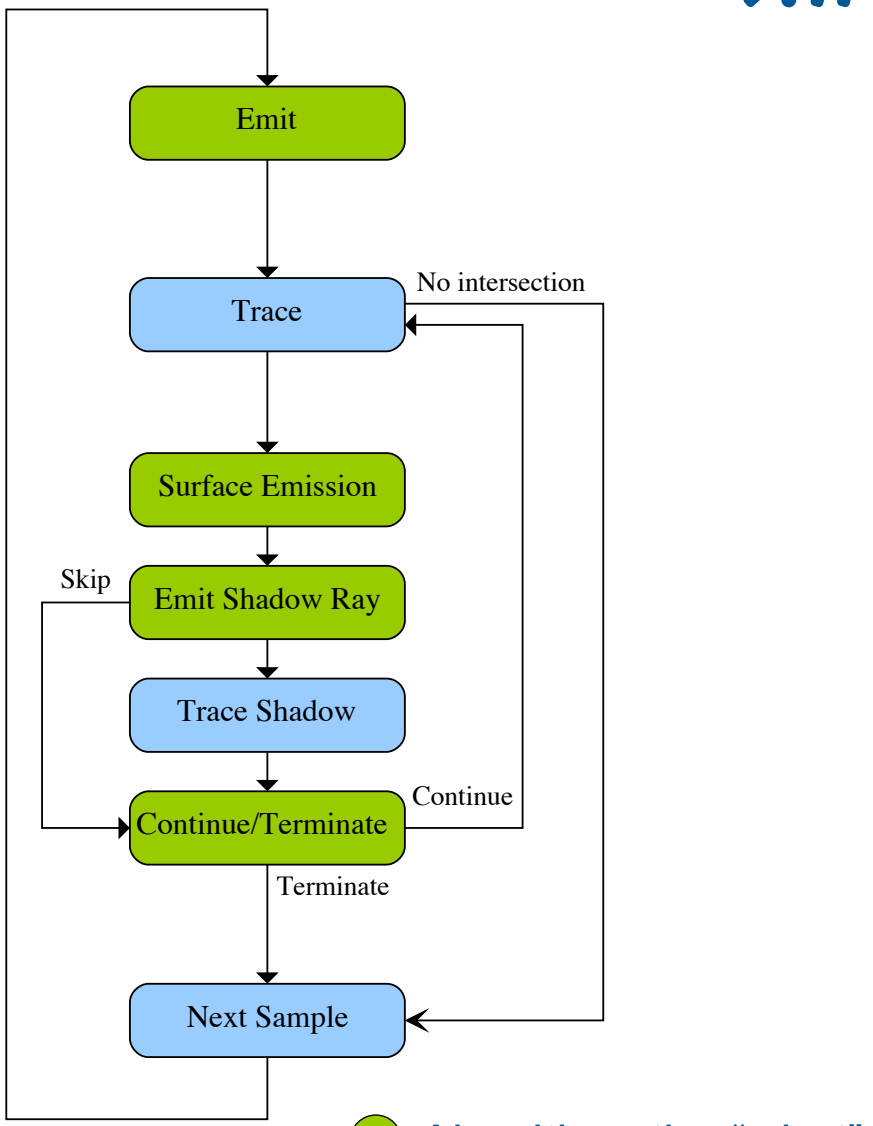
Rodent

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**
  - For example, from within Blender via exporter
  - Code triggers code generation



Algorithm, the “what”

Schedule, the “how”

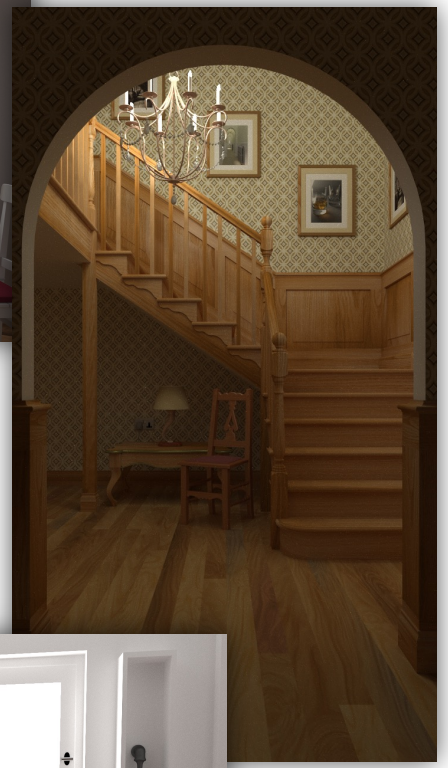
Rodent

Features

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

Rodent

Test Scenes



Rodent

Performance Results

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

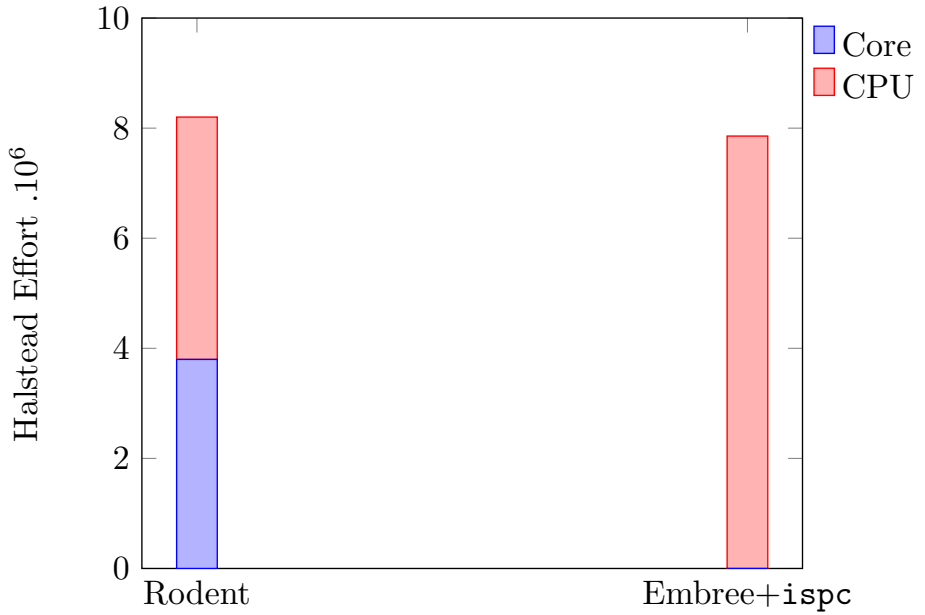
	CPU (Intel™ i7 6700K)		GPU (NVIDIA™ Titan X)			GPU (AMD™ Radeon RX 5700 XT) (50th Anniversary Edition)	
Scene	Rodent	Embree	Rodent: MK	Rodent: WF	OptiX	Rodent: MK	Rodent: WF
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	56.46	67.93
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	41.16	54.61
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	44.68	63.30
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	31.77	62.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	35.56	55.34
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	33.82	44.96

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

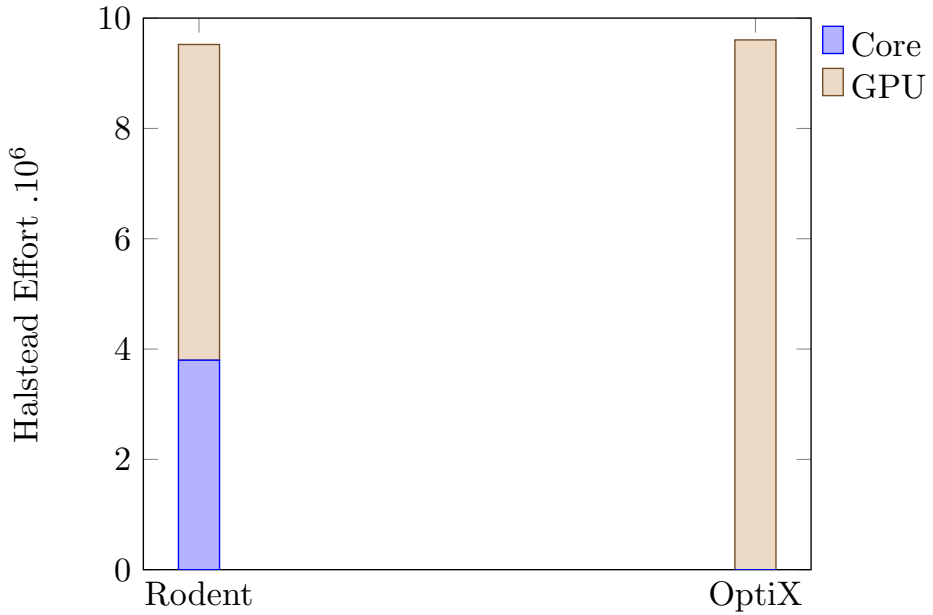
Rodent

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)



Rodent

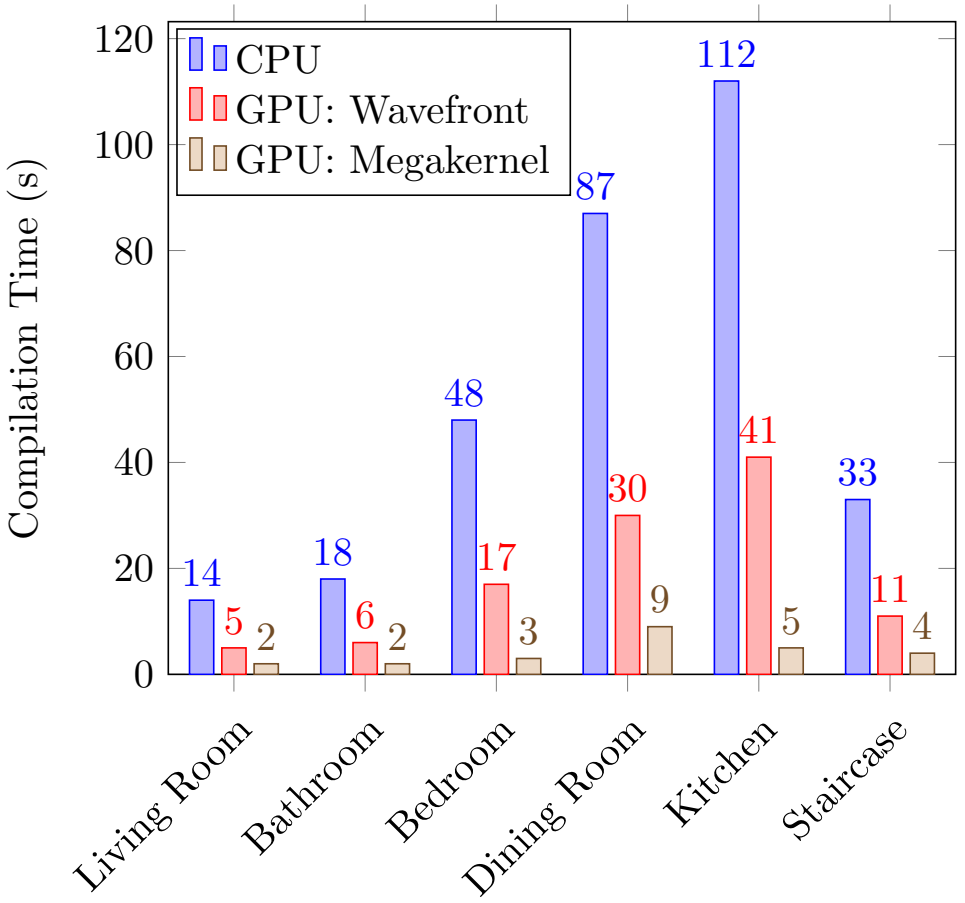
Scene Statistics: Compile Time & Shader Fusion

- **Megakernel only: shader fusion**

  #initial → #unique → #fused

  - Living room: 19 → 16 → 6
  - Bathroom: 16 → 15 → 5
  - Dining room: 58 → 51 → 28
  - Kitchen: 129 → 95 → 19
  - Staircase: 31 → 27 → 11
  - Bedroom: 41 → 38 → 13

- **Compilation times**



Pixel-based Denoising of Monte-Carlo Noise

4 spp

800 spp

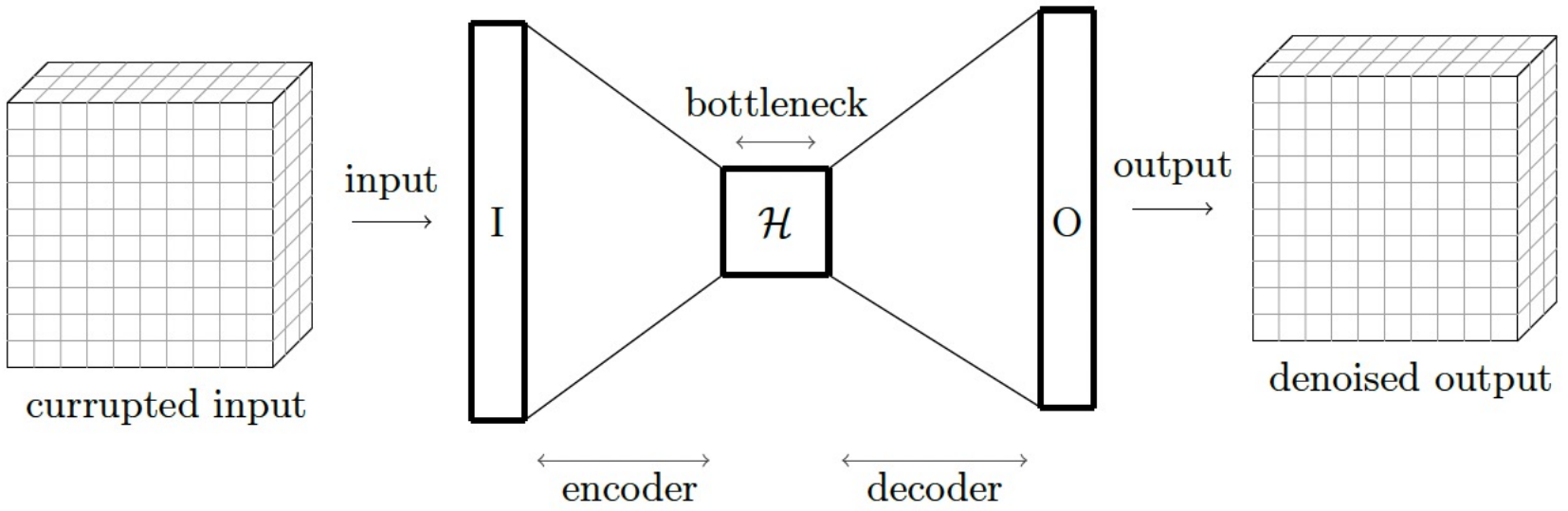
8000 spp

spp → ∞

image denoising

denoised

- **OptiX (NVIDIA)**
  - NVIDIA GPU only
  - Trained in TensorFlow
  - AutoEncoder network
  - Inference acceleration with cuDNN
- **Open Image Denoise (Intel)**
  - CPU only with at least SSE4.1
  - Trained in PyTorch
  - AutoEncoder and U-Net architecture
  - Inference acceleration with oneAPI DNN

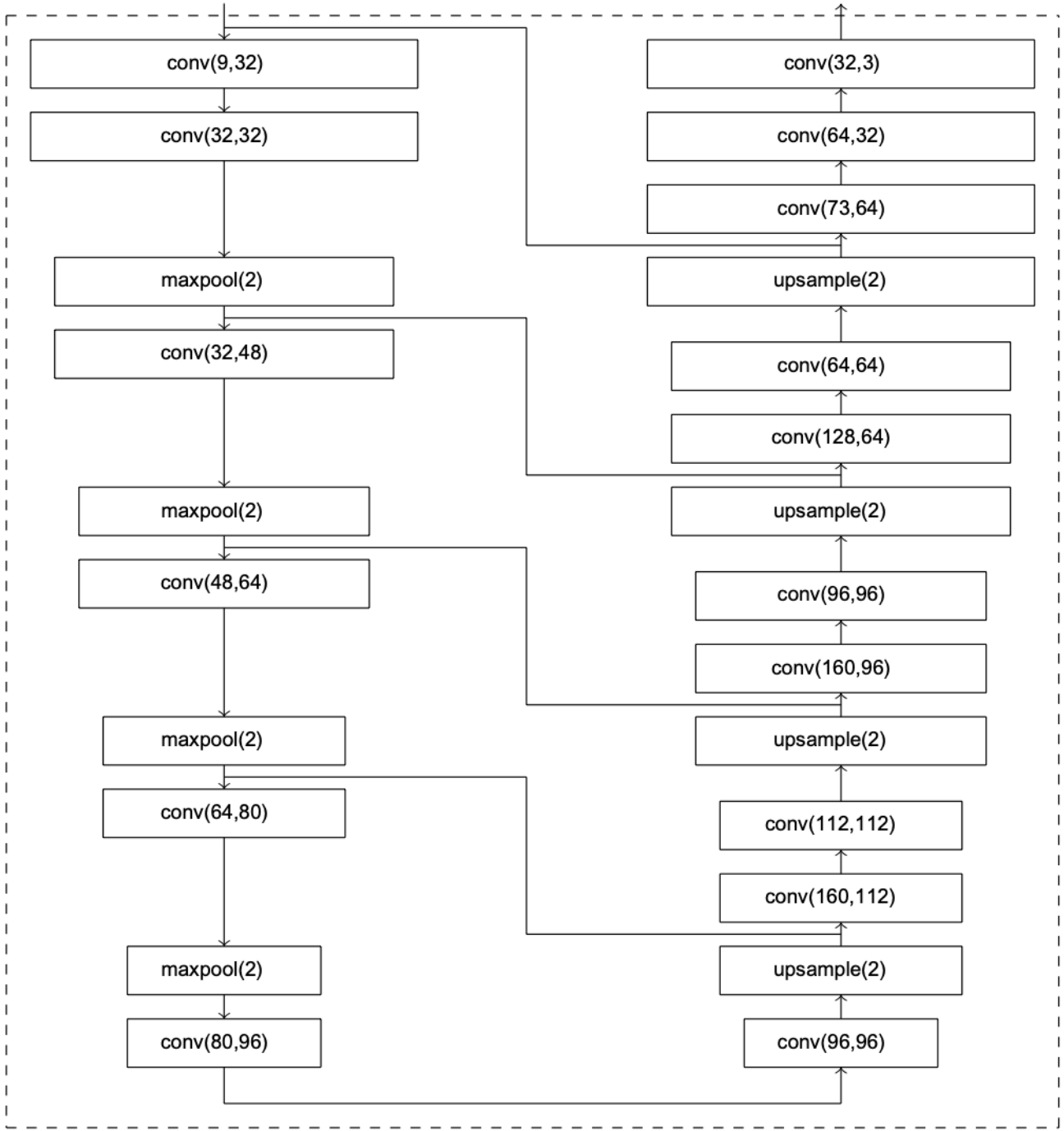


Intel Open Image Denoise

OIDN

U-Net Architecture

- **Limited number of operators**
  - Pooling
  - Upsampling
  - Bias and activation
  - Convolution
- **Cross-Connections**



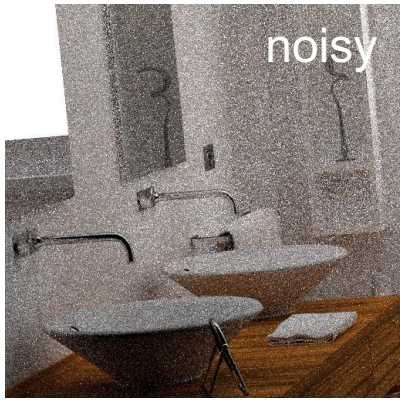
Auxiliary Feature Images

width x height x 3

image denoising

width x height x 3

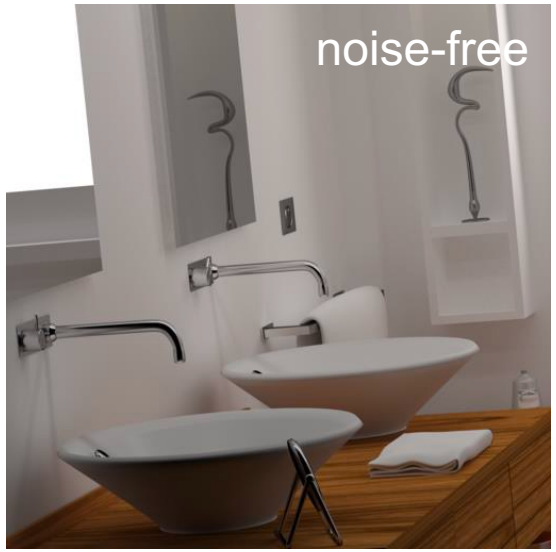
Auxiliary Feature Images



width x height x 9

image denoising

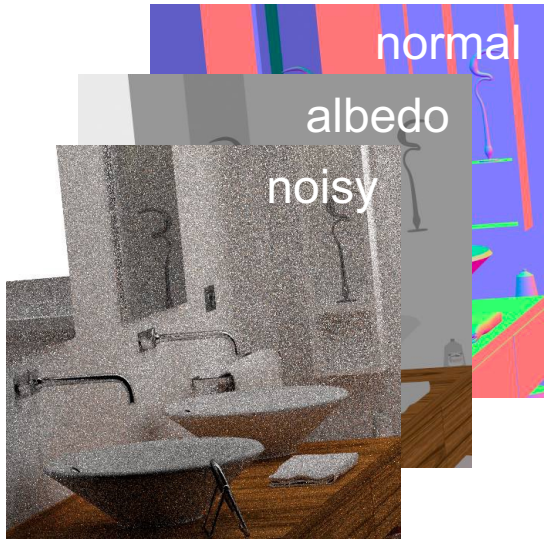
width x height x 3

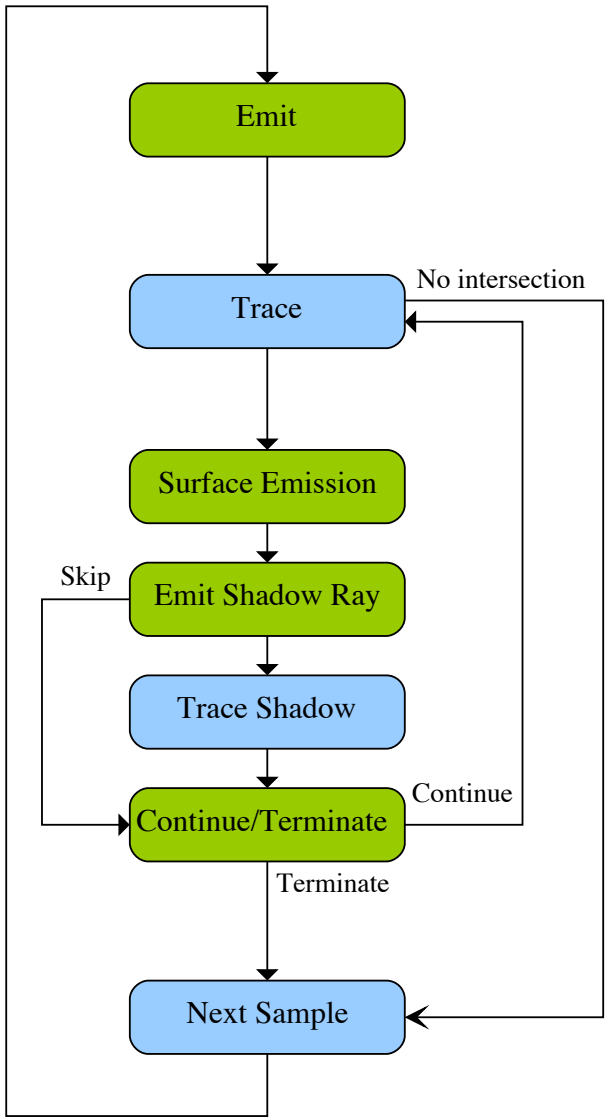


Rodent

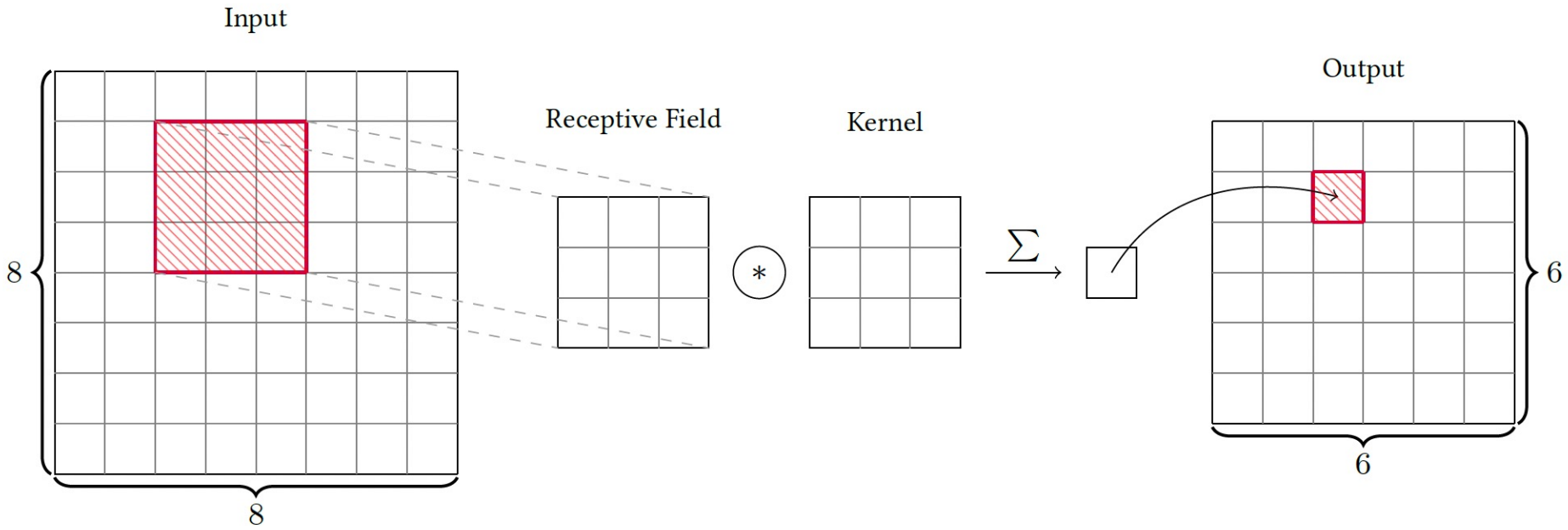
Integration

- **Denoising as a post-processing step**
  - Render normal, albedo, and noisy images
  - Pass buffer to denoiser
  - Display returned buffer



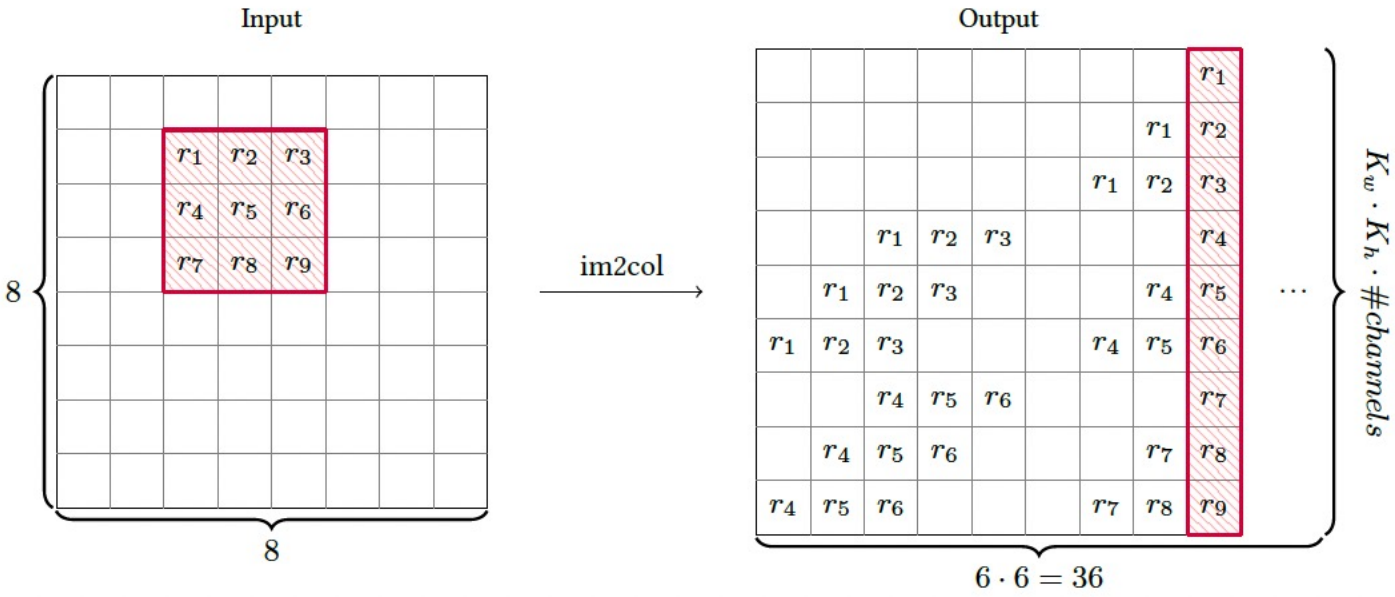


- **Convolutions are at the heart of DL**
- **Efficient implementation crucial**
- **How can we get efficient implements?**



- **Im2col transformation**
  - Rearrange memory layout
  - Allows to use one matrix multiplication for all convolutions in one convolutional layer

    → use fast BLAS libraries



Rodent Implementation

Matrix Multiplication

```
let c_size = 64; // tile size for b
let i_size = 16; // tile size for a
let mut m : [f32 * 64]; // temporary memory for sum

for cc in range_step(0, b.cols, c_size) {
    for ii in range_step(0, a.cols, i_size) {
        for r in range(0, m.rows) {
            if ii == 0 {
                m(j) = 0;
            } else {
                m(j) = c.read(r, c + j);
            }
            for i in range(ii, ii + i_size) {
                m(j) += a.read(r, i) * b.read(i, c + j);
            }
            c.write(r, c + j, m(j));
        }
    }
}
```

- **Code specialization via partial evaluation**
  - Parallelization
  - Vectorization
  - Specialization of i loop
    - load bias
    - apply activation function

Rodent Denoiser

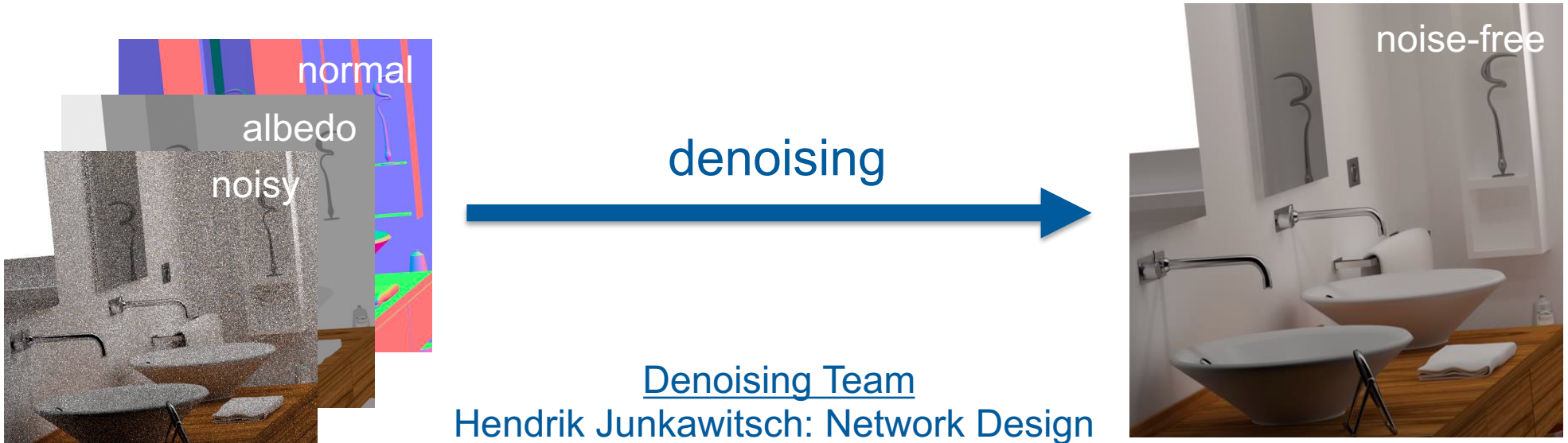
Results

Benchmark	Matrix Multiplication	Time			
		Average	im2col	matmul	other
Denoise 512	AnyDSL (CPU)	575 ms	52%	46%	< 1%
	oneAPI	508 ms	57%	41%	< 1%
	AnyDSL (GPU)	108 ms	12%	87%	< 1%
	cuBLAS	38 ms	34%	64%	< 1%
Denoise 1024	AnyDSL (CPU)	2427 ms	63%	36%	< 1%
	oneAPI	2262 ms	66%	33%	< 1%
	AnyDSL (GPU)	449 ms	11%	88%	< 1%
	cuBLAS	136 ms	38%	60%	< 1%
SuperRes 512	AnyDSL (CPU)	620 ms	72%	27%	< 1%
	oneAPI	603 ms	71%	28%	< 1%
	AnyDSL (GPU)	80 ms	16%	83%	< 1%
	cuBLAS	30 ms	42%	57%	< 1%
SuperRes 1024	AnyDSL (CPU)	2692 ms	76%	23%	< 1%
	oneAPI	2511 ms	79%	20%	< 1%
	AnyDSL (GPU)	342 ms	14%	84%	< 1%
	cuBLAS	114 ms	44%	55%	< 1%

Time in ms (lower is better): AMD Ryzen 5 5900X and NVIDIA GeForce GTX 1080.

Thank you for your attention.

Questions?



Denoising Team
Hendrik Junkawitsch: Network Design
Joshua Meyer: Rodent Integration

Publications

```
let rec pow x n =
    if n = 0 then
        .<1>.
    else if even n then
        let r = pow x (n/2)
            in r * r
    else
        .<.~x * .~(pow x (n-1))>.
```

(a) MetaOCaml

```
function pow(x, n)
    if n == 0 then
        return 1
    elseif n % 2 == 0 then
        local r=pow(x, n/2)
        return `[r]*[r]
    else
        return `[x]*[pow(x,n-1)]
    end
end
```

(b) Terra

```
def pow(x: Rep[Int], n: Int): Rep[Int]= {
    if (n == 0) {
        1
    } else if (n % 2 == 0) {
        val r = pow(x, n/2);
        r * r
    } else {
        x * pow(x, n-1)
    }
}
```

(c) Scala/LMS

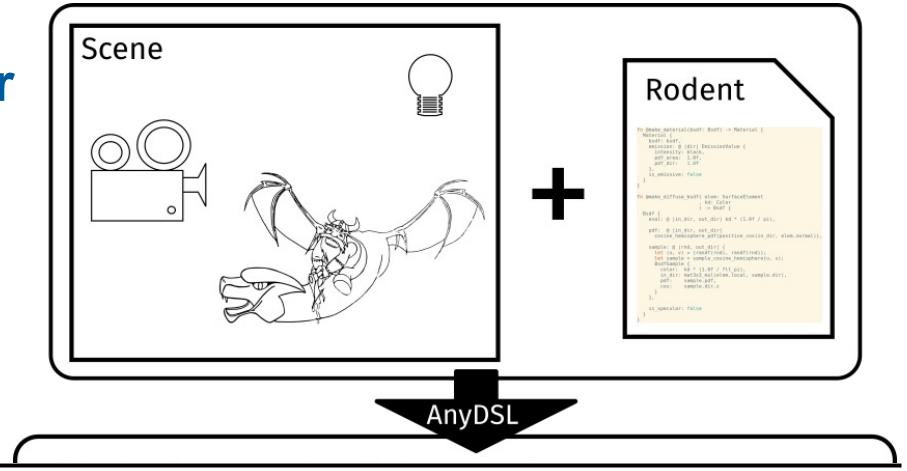
```
fn @(?n) pow(x: i32, n: i32) -> i32 {
    if n == 0 {
        1
    } else if n % 2 == 0 {
        let r = pow(x, n/2);
        r * r
    } else {
        x * pow(x, n-1)
    }
}
```

(d) Impala

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

- **User-guided partial evaluation (via filters)**
- **High-performance domain-specific libraries:**
  - Image processing
  - Ray tracing
  - Genome sequence alignment

Rodent: Generating Renderers without Writing a Generator



- Textbook-like, generic algorithms
- Tailored hardware schedules (CPU/GPU)
- Specialized code for a scene via AnyDSL

	CPU (Intel™ i7 6700K)		GPU (NVIDIA™ Titan X)			GPU (AMD™ Radeon RX 5700 XT) (50th Anniversary Edition)	
Scene	Rodent	Embree	Rodent: MK	Rodent: WF	OptiX	Rodent: MK	Rodent: WF
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	56.46	67.93
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	41.16	54.61
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	44.68	63.30
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	31.77	62.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	35.56	55.34
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	33.82	44.96

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

- Academy Technical Achievement Award 2020 for Intel's Embree

Parallel Multi-Hypothesis Algorithm for Criticality Estimation in Traffic and Collision Avoidance

- Parallel algorithm for embedded multi- and many-core hardware
- Millions of hypothesis combinations in real time (26 mio in 15ms)
- Joint work with Audi and THI

Parallel Multi-Hypothesis Algorithm for Criticality Estimation in Traffic and Collision Avoidance

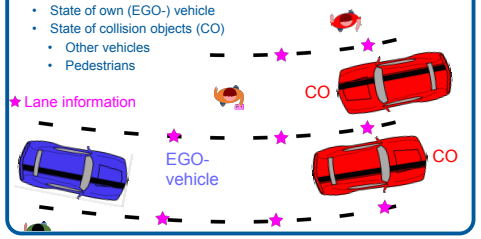
Eduardo Sanchez, Michael Botsch, Andreas Gaull
Ingolstadt Univ. of Applied Sciences
Alexander Kammenhuber, Christoph Lauer, Tobias Dirndorfer
Audi AG
Richard Membarth, Philipp Slusallek
German Research Center for Artificial Intelligence (DFKI)

1. Overview

- Prediction of the collision probability of a traffic situation termed as "Criticality"
- Generation of traffic situations based on vehicle sensor information
- A model-based prediction of possible vehicle and pedestrian trajectories
- Detection of possible collisions between traffic participants
- Approximation of conditional probabilities of the possible collisions
- Implementation on multi-core processor
- Output of the algorithm: safe trajectories

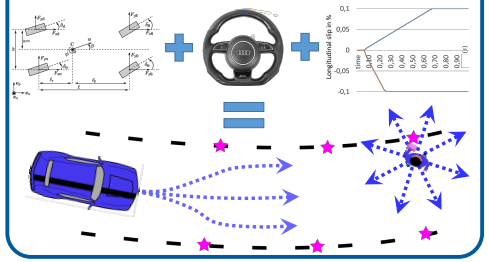
2. Traffic Situation Data Processing

- Sensors deliver information from the traffic situation:
  - Road infrastructure (lanes)
  - State of own (EGO-) vehicle
  - State of collision objects (CO)
    - Other vehicles
    - Pedestrians



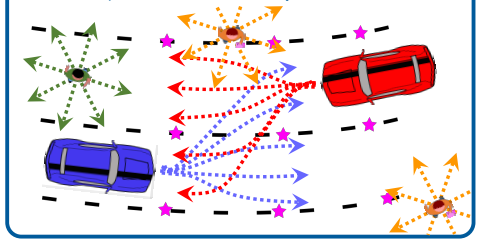
3. Trajectory Generation

- Generation of thousands of realistic trajectories for all vehicles by combining
  - Motion models
  - Controllers for lateral dynamics
  - Acceleration profiles
- Pedestrians are not bound to the road infrastructure



4. Collision Recognition

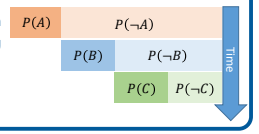
- Comparison between millions of pose combinations (EGO+CO)
- Detection of possible collisions between objects



5. Risk Assessment

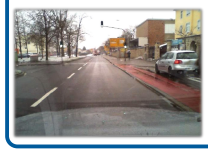
- Considering two objects: joint probability of two objects driving trajectories that lead to a collision
- Extension to more objects: taking into account the order in which the collisions can occur

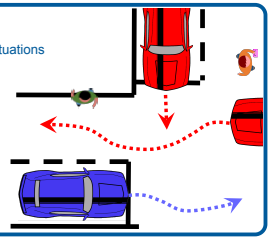
$$P(A)$$
$$P(B \cap \neg A) \approx P(\neg A) \cdot P(B)$$
$$P(C \cap \neg B \cap \neg A) \approx P(\neg A) \cdot P(\neg B) \cdot P(C)$$



6. Evaluation

- 20 generic, designed traffic situations
- 547 real-life traffic situations





7. Algorithm Mapping to Embedded Systems

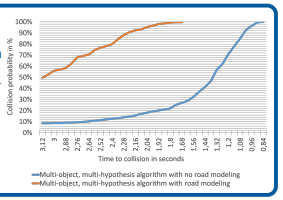
- Possibility of parallel execution of thousands of threads
- Exploit hardware specific features: module mapping to the GPU, extensive use of intrinsics (sine, cosine, tangent), no data-transfers between CPU and GPU
- AnyDSL used to avoid tying the implementation to a specific platform
- Thrift used to retrieve input data from the vehicle and to return the algorithm outputs

Runtime performance on a Drive PX 2

# of CO	# of pose combinations	CPU ARM Cortex A57	Pascal GPU GP106 - 1280 cores	Tegra X2 GPU GP10B - 256 cores
3	25.93 million	1800 ms	10 ms	15 ms
	18.00 million	1600 ms	8 ms	11 ms
10	86.44 million	11600 ms	21 ms	49 ms
	60.01 million	9600 ms	14 ms	35 ms

8. Results

- Smooth, gradual changes in the predicted criticality
- Enough anticipation time for activating passive safety systems (cross-traffic) and active safety systems (longitudinal traffic)

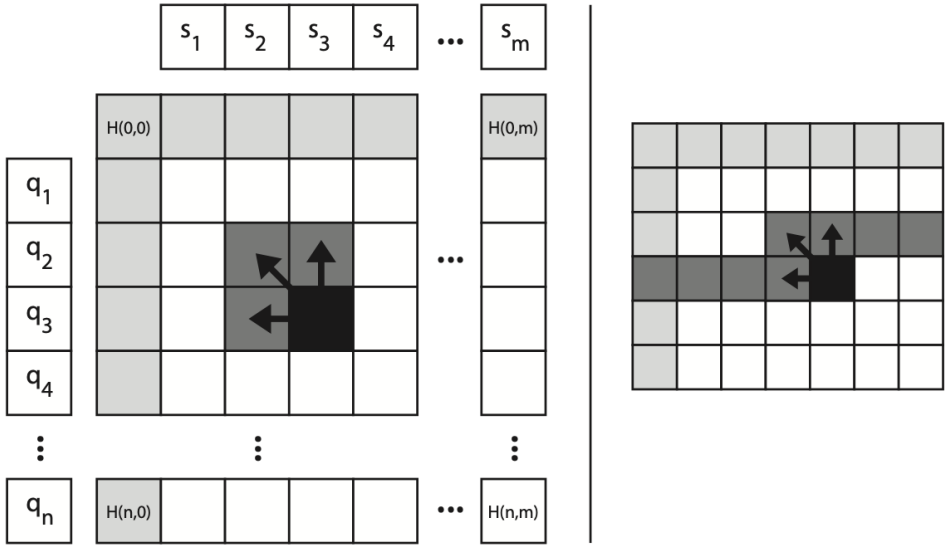


9. Conclusion

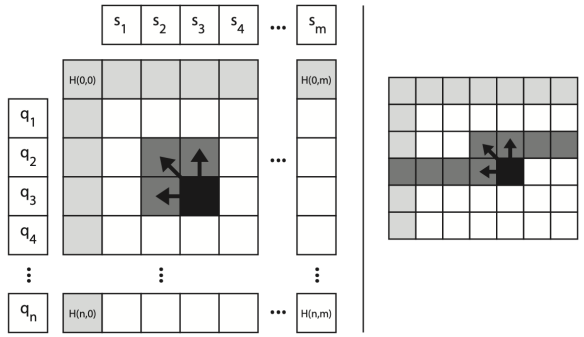
- Prediction of safe trajectories
- A fully model-based, multi-modal, parallelizable algorithm is presented.
- The criticality of the current traffic situation is predicted
- Further inclusion of road infrastructure elements is possible
- Implementation on vehicle-compatible hardware proves prototypically possibility of use in series-production vehicles

AnySeq: A High Performance Sequence Alignment Library based on Partial Evaluation

- **Alignment library**
  - Split in common & architecture part
  - Algorithmic variations by function compositing
  - Mappings for multithreading, vectorization, GPUs, and FPGAs
- **Same speed as state-of-the-art**
  - SeqAn (CPU)
  - NVBio (GPU)



AnySeq/GPU: A Novel Approach for Faster Sequence Alignment on GPUs

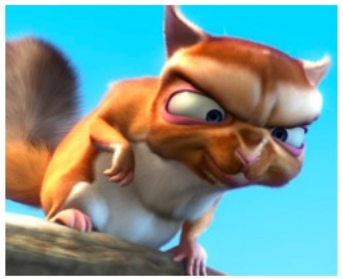


- **Minimized memory accesses**
  - Warp shuffles, half-precision arithmetic
  - Dynamic programming partitioning scheme
- **Fastest available GPU implementations**
  - Over 80% of the available peak performance
  - Median speedup of 19.2x

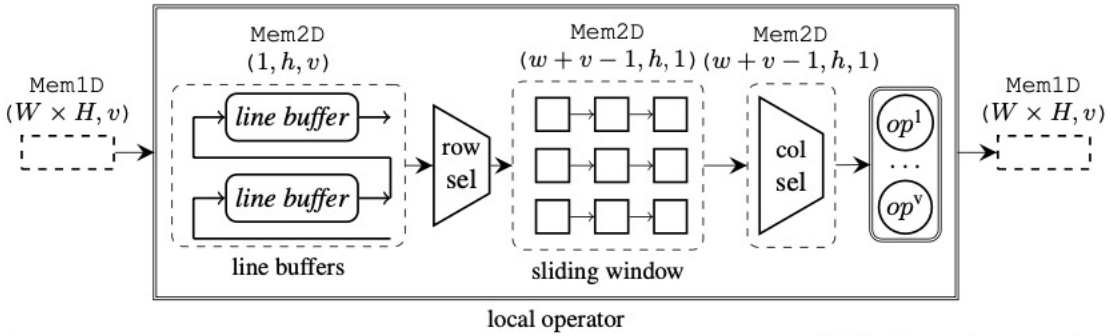
    (GASAL2, ADEPT, NVBIO, AnySeq 1)

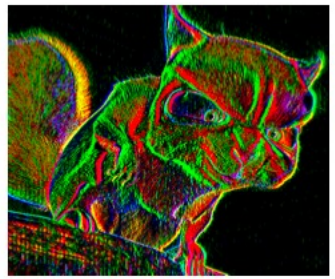
	V100	RTX3090	A100	MI100
FP32 cores	5120	10496	6912	7680
Boost Clock (GHz)	1.91	1.70	1.41	1.50
Cycles Cell Update	7	7	3.5	3.5
TPP in TCUPS	1.40	2.55	2.78	3.29
Achieved TCUPS	1.33	2.09	2.49	2.82
Efficiency	95%	82%	89%	86%

Publications: TCAD'20 (CODES+ISSS'20)



© Blender Foundation (CC BY 3.0)



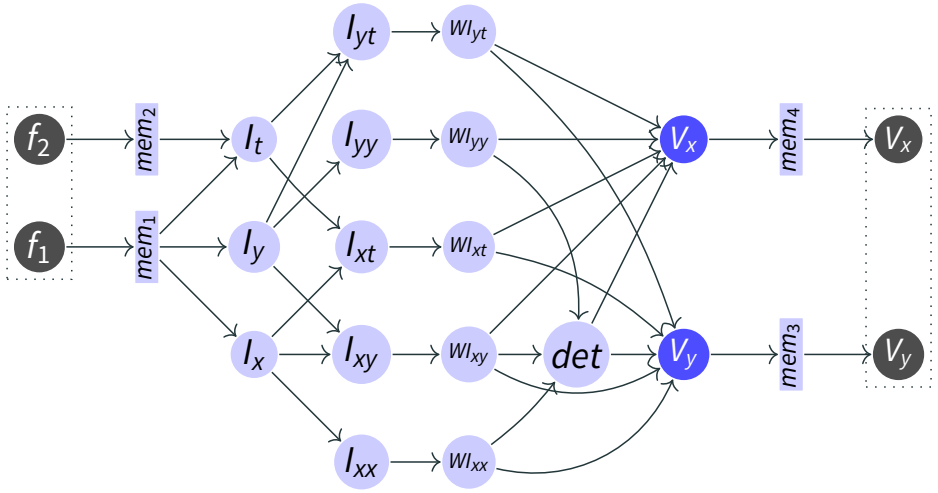


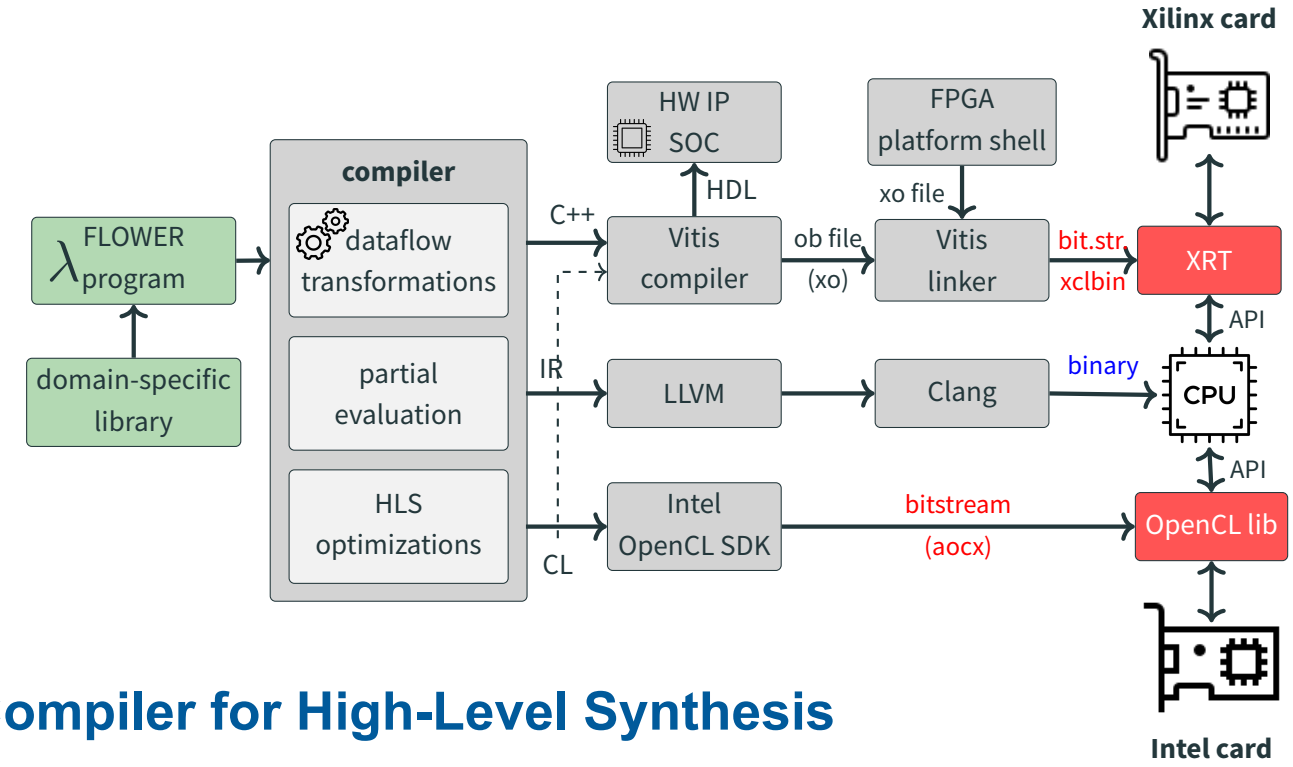
```
let sobel_x = @|img, x, y|
    +1 * img.read(x-1, y-1) + -1 * img.read(x+1, y-1) +
    +2 * img.read(x-1, y  ) + -2 * img.read(x+1, y  ) +
    +1 * img.read(x-1, y+1) + -1 * img.read(x+1, y+1);
```

```
let input = make_img_mem1d("frank.jpg");
let output = make_img_mem1d("frank_sobel.jpg");
let operator = make_local_op(sobel_x);
with generate(vhls) { operator(input, output) }
```

AnyHLS: High-Level Synthesis with Partial Evaluation

- High-level abstractions for FPGA designs
- Support for Xilinx and Intel FPGAs via HLS and OpenCL





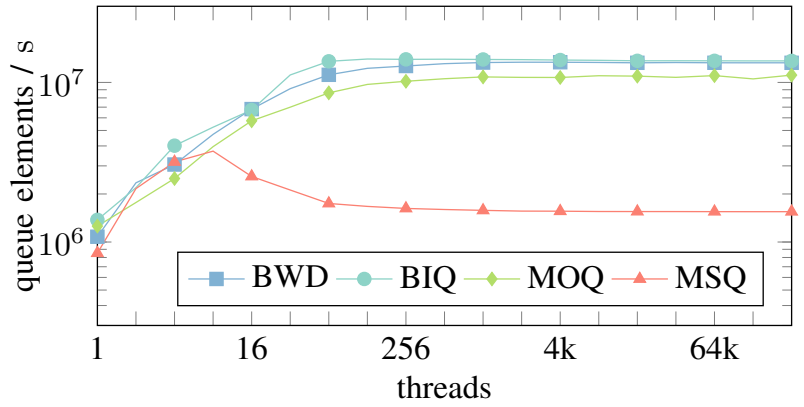
FLOWER: A Comprehensive Dataflow Compiler for High-Level Synthesis

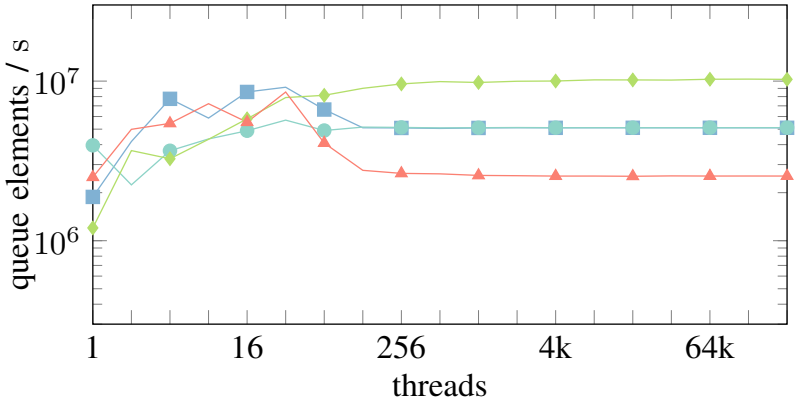
- **Automatic generation of mixed host/FPGA implementations**
- **Dataflow generation for Xilinx and Intel FPGAs via HLS and OpenCL**

Publications: IPDPS'23

accepted

```
type source[T] = fn() -> T;
type sink[T] = fn(T) -> ();

struct ProducerConsumerQueue[T] {
    push: fn(source[T]) -> fn(ctx) -> i32,
    pop:  fn(sink[T]) -> fn(ctx) -> i32
}
```



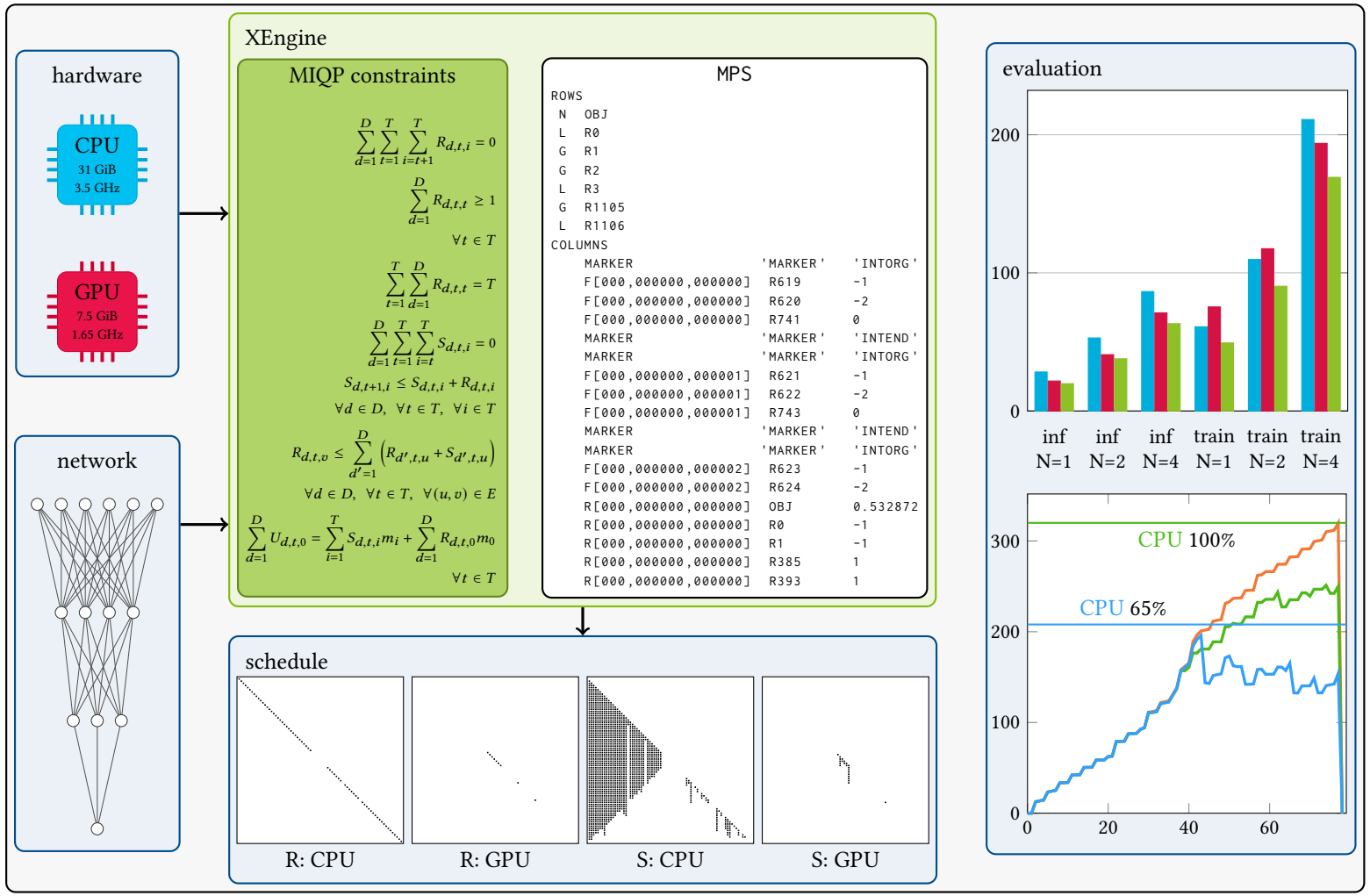


AnyQ: An Evaluation Framework for Massively-Parallel Queue Algorithms

- Queue abstractions for efficient mapping of concurrent queue algorithms
- Queue benchmarking and testing framework for CPUs (x86 / arm) and GPUs (NVIDIA / AMD)

XEngine: Optimal Tensor Rematerialization for Neural Networks in Heterogeneous Environments

GitHub



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

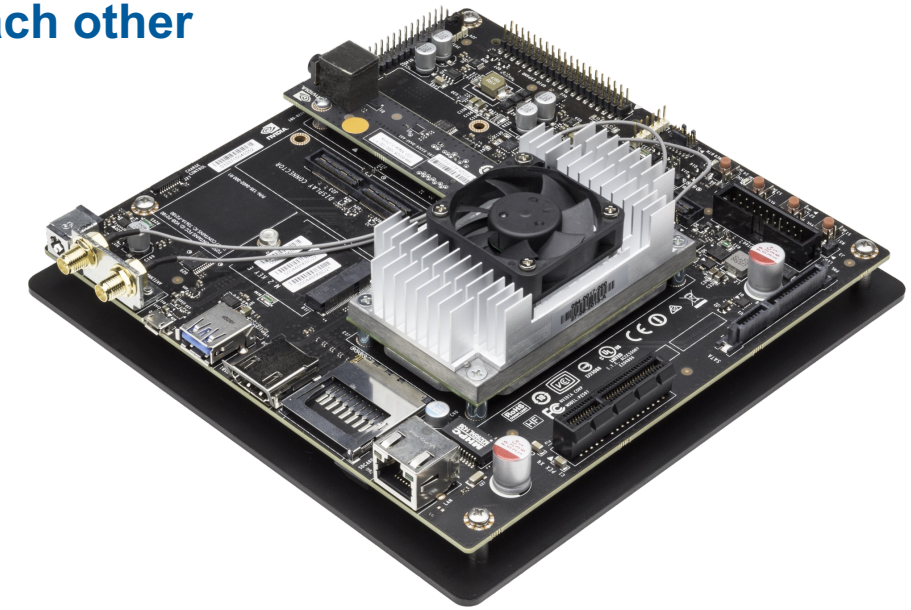
Joint Project: Audi, DFKI, and THI

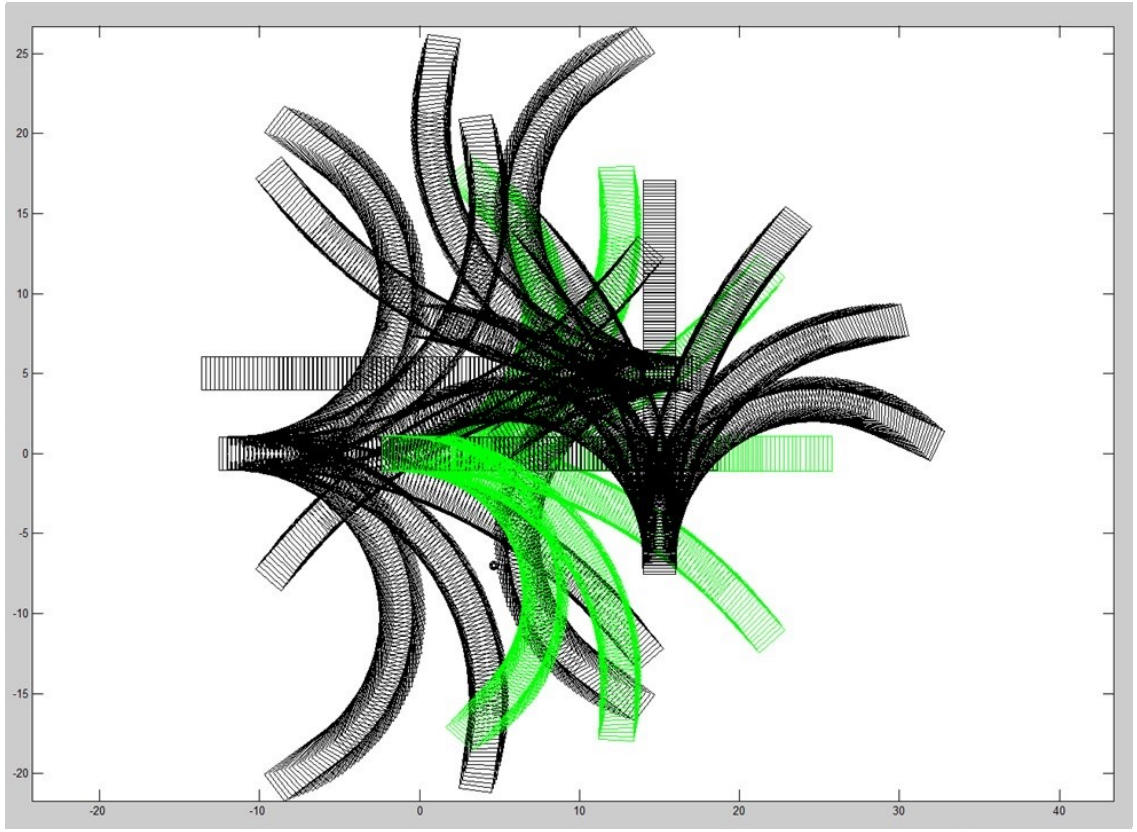
CARISSMA
Automotive Safety Research

Technische Hochschule Ingolstadt

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

Lang	HW	Time
MatLab	Intel Core i5	6 min
AnyDSL	Tegra X1 CPU	4 s
AnyDSL	Tegra X1 GPU	36 ms

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

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