Target-Specific Refinement of Multigrid Codes

Richard Membarth, Philipp Slusallek (IVCI & DFKI)
Roland Leißa, Marcel Köster, Sebastian Hack (IVCI & UdS)

Intel Visual Computing Institute (IVCI) at Saarland University (UdS)
German Research Center for Artificial Intelligence (DFKI)

WOLFHPC'14, New Orleans

This work is co-funded by BMBF projects Collaborate3D and ECOUSS, as well as EU project Dreamspace.

Serial 2D-Stencil Code

- Iterate over domain, apply fixed stencil

```
for y in range(0, arr.rows) {
  for x in range(0, arr.cols) {
    arr(x, y) =
                          0.25f * in(x, y-1) +
    0.25f * in(x-1, y) + 0.50f * in(x, y  ) + 0.25f * in(x+1, y) +
                          0.25f * in(x, y+1);
  }
}
```

- Domain-specific variants
  - Stencil size, shape
  - Boundary handling
- Target-specific optimizations
  - Blocking
  - Vectorization
  - Accelerator offloading

Stencil Interpreter

- Interpreter considers domain-specific variants

```
fn apply_stencil(x: int, y: int,
                 field: Field, stencil: Stencil,
                 border: fn(int, int, int) -> int
                ) -> float {
  let mut sum = 0.0f;
  let half = stencil.size / 2;

  for ys in range(-half, half+1) {
    for xs in range(-half, half+1) {

        let xx = border(x+xs, 0, field.cols-1);
        let yy = border(y+ys, 0, field.rows-1);
        sum += field(xx, yy) * stencil(xs, ys);

    }
  }

  sum
}
```

Stencil Interpreter

- Interpreter considers domain-specific variants

```
fn apply_stencil(x: int, y: int,
                 field: Field, stencil: Stencil,
                 border: fn(int, int, int) -> int
                ) -> float {
  let mut sum = 0.0f;
  let half = stencil.size / 2;

  for ys in range(-half, half+1) {
    for xs in range(-half, half+1) {
      if stencil(xs, ys) != 0.0f {
        let xx = border(x+xs, 0, field.cols-1);
        let yy = border(y+ys, 0, field.rows-1);
        sum += field(xx, yy) * stencil(xs, ys);
      }
    }
  }

  sum
}
```

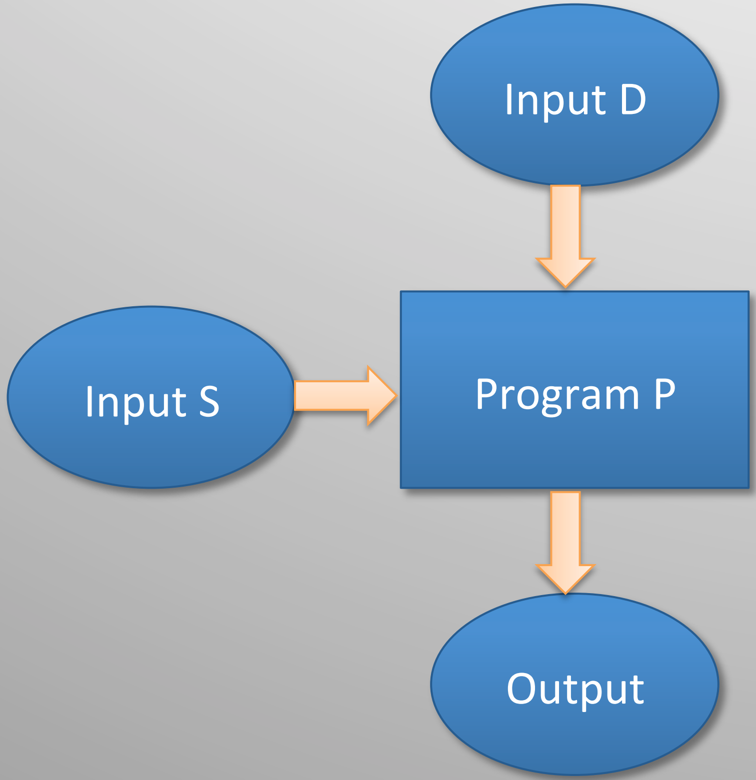
Domain Variants

- Application developer selects domain-specific components
  - Boundary handling
  - Stencil

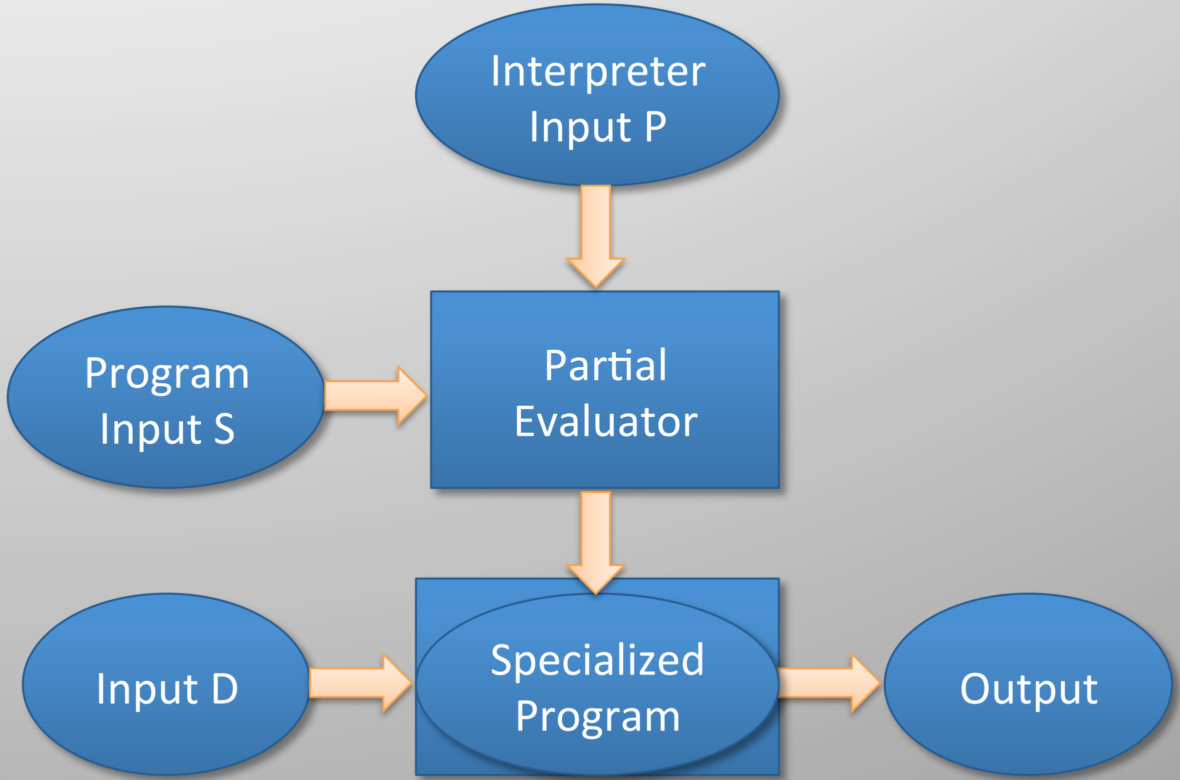
```
fn clamp(idx: int, lower: int, upper: int) -> int {
  min(upper, max(lower, idx))
}

let stencil: Stencil = { data: [[0.00f, 0.25f, 0.00f],
                                [0.25f, 0.50f, 0.25f],
                                [0.00f, 0.25f, 0.00f]],
                         /* ... */ };
let mut out: Field   = { /* ... */ };

for x, y in iterate(out) {
  out(x, y) = apply_stencil(x, y, field, stencil, clamp);
}
```

Stencil Specialization using Partial Evaluation

- Normal program execution



- Execution with program specialization

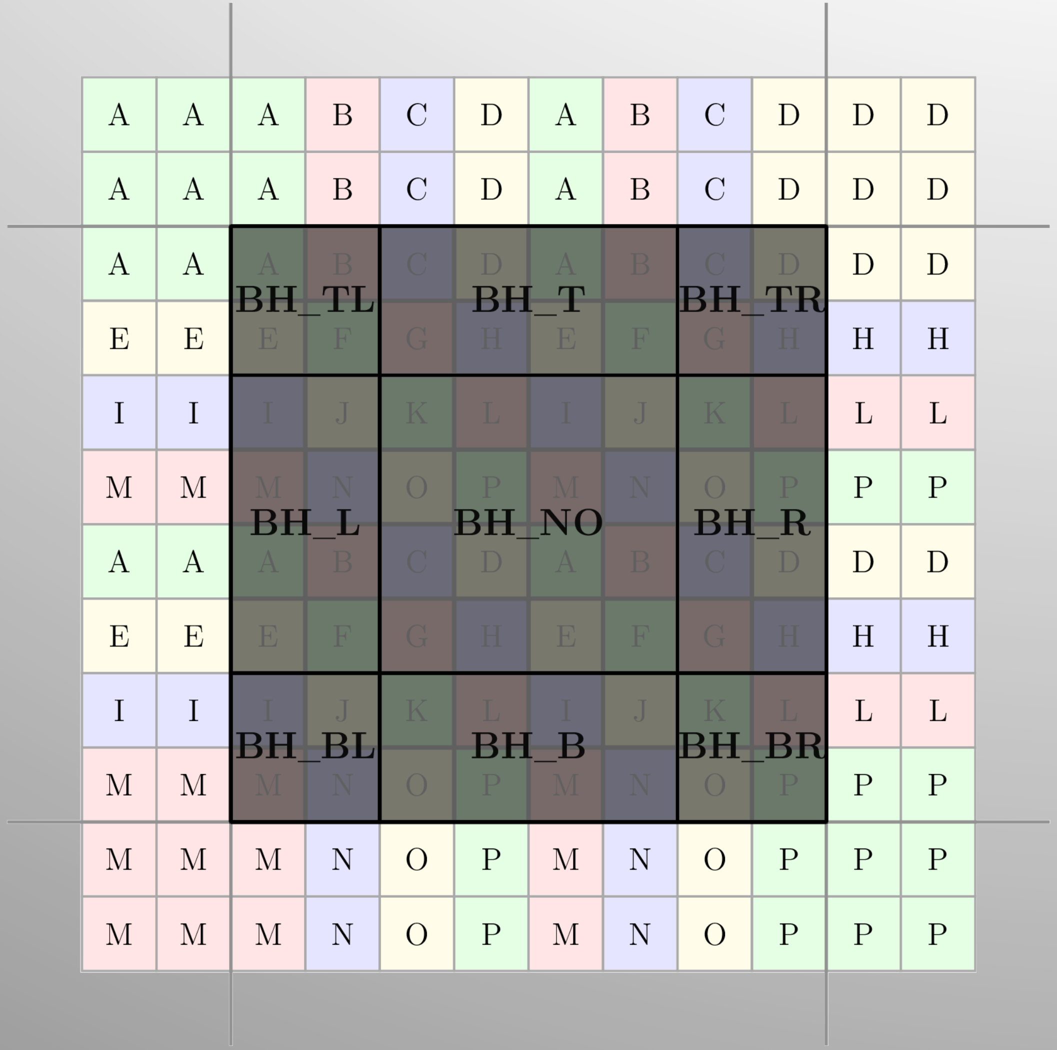


Stencil Specialization through Partial Evaluation

- Partial evaluation is exposed through @
- Preserves program semantics

```
fn clamp(idx: int, lower: int, upper: int) -> int {
  min(upper, max(lower, idx))
}

let stencil: Stencil = { data: [[0.00f, 0.25f, 0.00f],
                                 [0.25f, 0.50f, 0.25f],
                                 [0.00f, 0.25f, 0.00f]],
                         /* ... */ };
let mut out: Field   = { /* ... */ };

for x, y in iterate(out) {
  out(x, y) = @apply_stencil(x, y, field, stencil, clamp);
}
```

Exploiting Boundary Handling

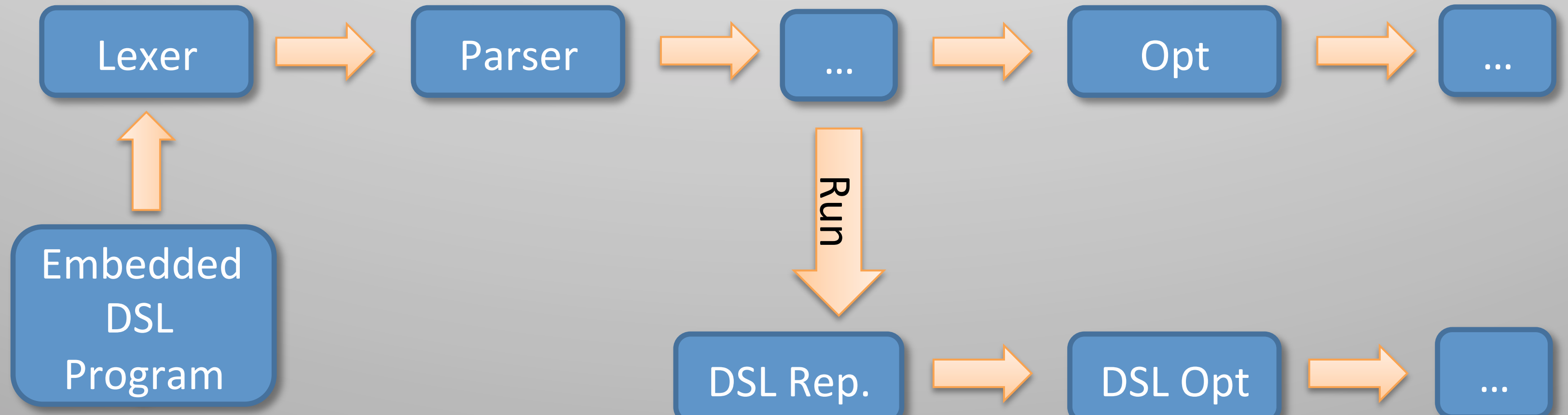


- Boundary handling
  - Evaluated for all points
  - Unnecessary evaluation of conditionals
- Specialized variants for different regions [HiStencils14]
- Automatic generation of variants
  → Partial evaluation

Compiler Work-Flow

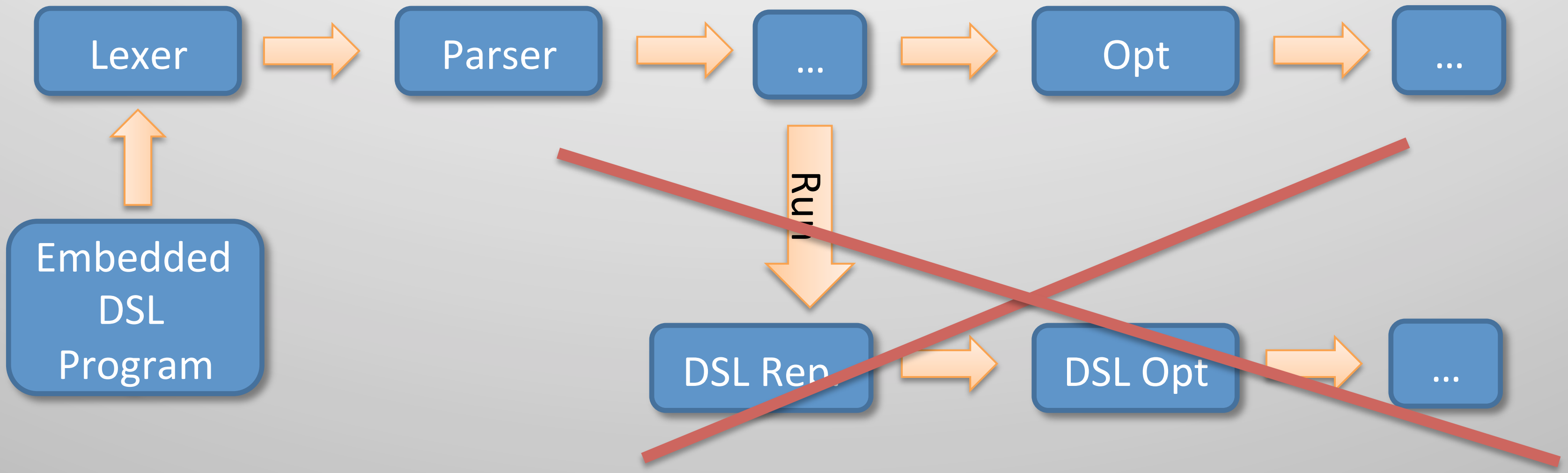
- General-purpose languages

- Domain-specific languages (embedding)



Our Approach

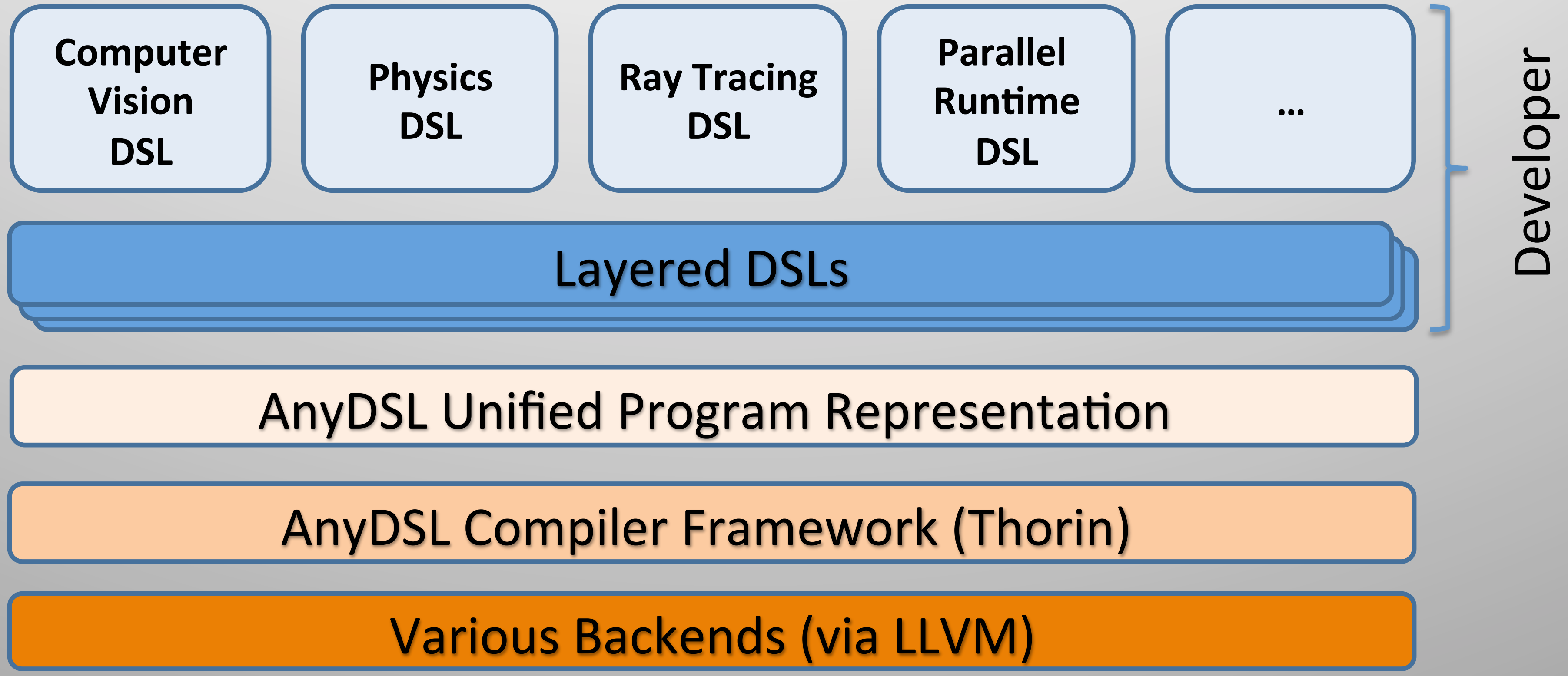
- DSL embedding in own host language



- Partial evaluation
- Triggered code generation
- Typesafe

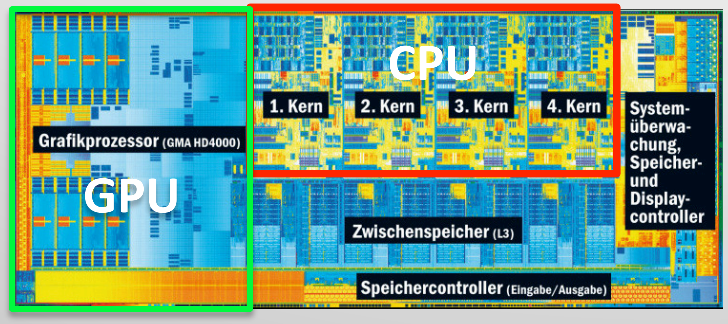
Our Approach

- AnyDSL framework

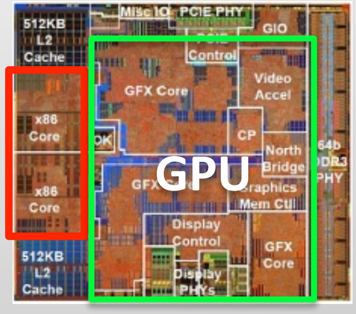


Many-Core Dilemma

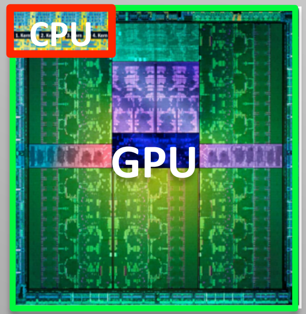
- Many-core HW is everywhere – but programming it is still hard



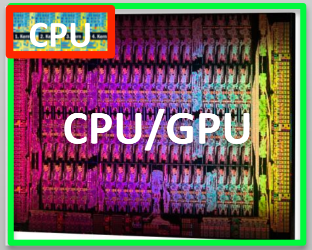
Intel Haswell Architecture (1.4B Transistors)



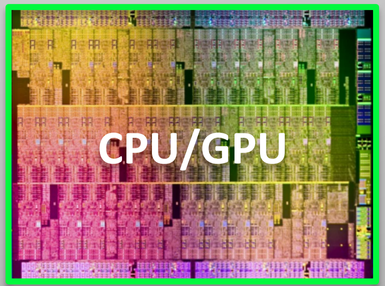
AMD Brazo



Nvidia Kepler
(~7B Transistors)



Intel Knights Ferry
(~5B Transistors)



Intel Knights Landing

Mapping to Target Hardware

- Higher level domain-specific code
  - iterate function iterates over field (provided by machine expert)

```
fn clamp(idx: int, lower: int, upper: int) -> int {
  min(upper, max(lower, idx))
}

let stencil: Stencil = { data: [[0.00f, 0.25f, 0.00f],
                                [0.25f, 0.50f, 0.25f],
                                [0.00f, 0.25f, 0.00f]],
                         /* ... */ };
let mut out: Field   = { /* ... */ };

for x, y in iterate(out) {
  out(x, y) = @apply_stencil(x, y, field, stencil, clamp);
}
```

Mapping to Target Hardware

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

```
fn clamp(idx: int, lower: int, upper: int) -> int {
  min(upper, max(lower, idx))
}

let stencil: Stencil = { data: [[0.00f, 0.25f, 0.00f],
                                [0.25f, 0.50f, 0.25f],
                                [0.00f, 0.25f, 0.00f]],
                         /* ... */ };
let mut out: Field   = { /* ... */ };

iterate(out, |x, y| -> () {
  out(x, y) = @apply_stencil(x, y, field, stencil, clamp);
});
```

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

```
fn iterate(arr: Field, body: fn(int, int) -> ()) -> () {
  for y in range(0, arr.rows) {
    for x in range(0, arr.cols) {
      ...
      body(x, y);
    }
  }
}
```

Mapping to Target Hardware (2)

- Scheduling & mapping provided by machine expert
  - CPU code using vectorization (e.g. AVX)
  - vectorize is provided by the compiler, uses whole-function vectorization

```
fn iterate(arr: Field, body: fn(int, int) -> ()) -> () {
  let vector_length = 8;
  for y in range(0, arr.rows) {
    for vectorize(vector_length, 0, arr.cols) {
      let x = wfv_get_tid();
      ...
      body(x, y);
    }
  }
}
```

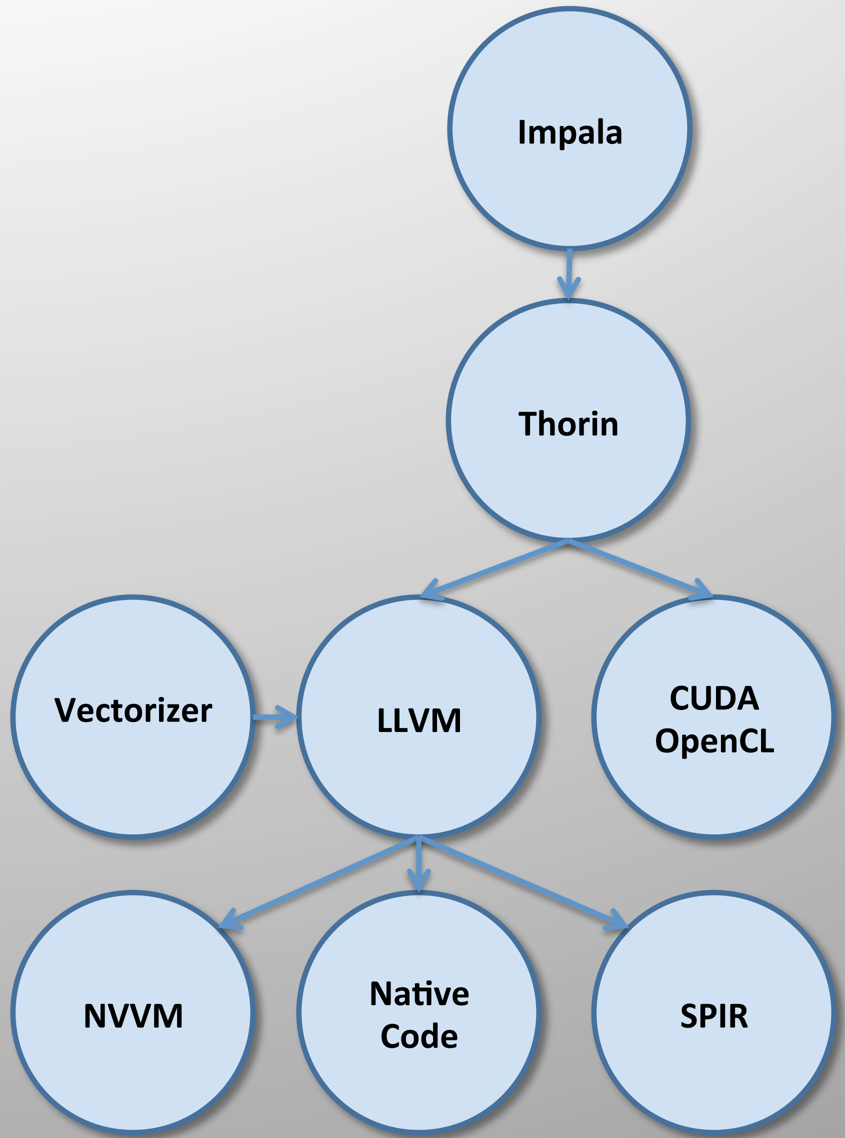
Mapping to Target Hardware (3)

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

```
fn iterate(arr: Field, body: fn(int, int) -> ()) -> () {
  let grid  = (arr.cols, arr.rows, 1);
  let block = (32, 4, 1);

  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);
  });
}
```

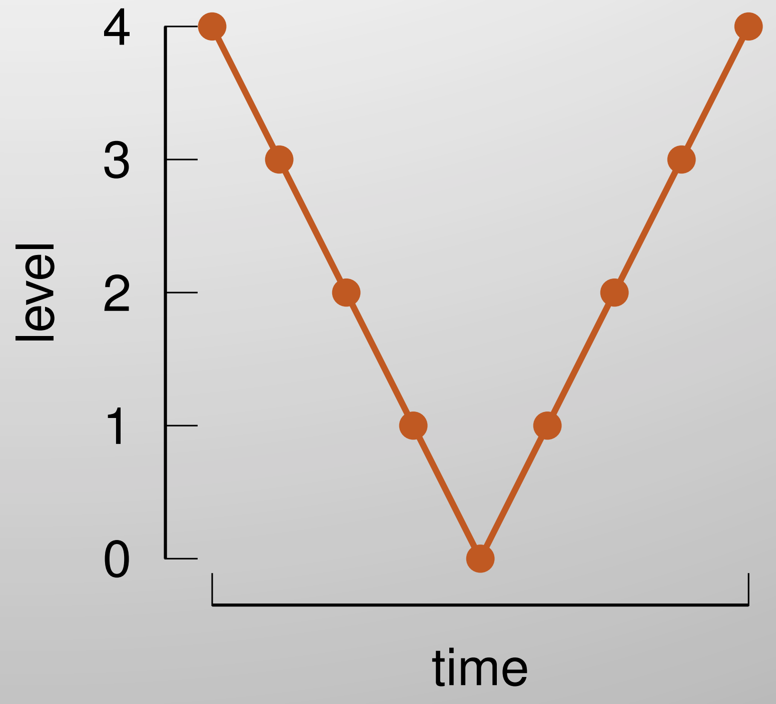
Compiler Framework

- Impala language (Rust dialect)
  - Functional & imperative language
- Thorin compiler [CGO 2015]
  - Higher-order functional IR
    - Special optimization passes
    - No overhead during runtime
- Whole-Function Vectorizer [CGO 2011]
- LLVM
  - Full compiler optimization passes
  - Multi-target code generation
    - SPIR, NVVM
    - CPUs, GPUs, MICs, …



Application: Multigrid Method

1. Pre-smoothing
2. Residual computation
3. Restriction
4. Recursion
5. Interpolation
6. Correction
7. Post-smoothing



- Previous work: Modeled in Hipacc [WOLFHPC'12]

Application: Multigrid Method

1. Pre-smoothing
2. Residual computation
3. Restriction
4. Recursion
5. Interpolation
6. Correction
7. Post-smoothing

```
fn vcycle(in: Field, levels) -> Field {
  // allocate memory for all levels

  /* vcycle implementation */
  fn vcycle_intern(level: int) -> () {
    if level == levels-1 {
      jacobi(/* fields */);
    } else {
      jacobi(/* fields */);
      residual(/* fields */);
      restrict(/* fields */);

      vcycle_intern(level+1); // recursion

      interpolate(/* fields */);
      jacobi(/* fields */);
    }
  }

  vcycle_intern(0);
}

/* call to vcycle */
let result = vcycle(input, levels);
```

A DSL for the V-cycle

- Pass V-cycle components as higher-order functions

```
fn vcycle_dsl(in: Field, levels: int,
              smoother:    fn(/* ... */) -> (),
              residual:    fn(/* ... */) -> (),
              restrict:    fn(/* ... */) -> (),
              interpolate: fn(/* ... */) -> ()
             ) -> Field {
  /* ... */
}

/* call to vcycle_dsl */
let result = @vcycle_dsl(input, 6 /* levels */,
                         jacobi, residual, restrict, interpolate);
```

A DSL for the V-cycle

- Perform scheduling in the DSL

```
fn vcycle_dsl(/* ... */) -> Field {
  fn vcycle_dsl_intern(level: int) -> () {
    if level == levels-1 {
      for x, y in iterate(Sol(level)) {
        solver(x, y, /* fields */);
      }
    } else {
      // call smoother
      // call residual
      // call restrict
      vcycle_dsl_intern(level+1); // recursion
      // call interpolate
      // call smoother
    }
  }

  vcycle_dsl_intern(0);
}
```

Loop Fusion

- Exemplarily shown for residual and restrict
- Create schedule that merges loop bodies

```
fn iterate_rr(Sol: Field, Res: Field, RHSF: Field, RHSC: Field,
              residual: fn(/* ... */) -> (),
              restrict: fn(/* ... */) -> ()) -> () {


  for y in $range(0, Res.rows) {
    for x in range(0, Res.cols) @{ // residual for two rows
      residual(x, y /* ... */ Sol, Res, RHSF);
    }
  }
  for y in $range(0, RHSC.rows) {
    for x in range(0, RHSC.cols) @{ // restrict the residual
      restrict(x, y /* ... */ Res, RHSC);
    }
  }
}
```

Loop Fusion

- Exemplarily shown for residual and restrict
- Create schedule that merges loop bodies

```
fn iterate_rr(Sol: Field, Res: Field, RHSF: Field, RHSC: Field,
              residual: fn(/* ... */) -> (),
              restrict: fn(/* ... */) -> ()) -> () {
  let mut tmp: Field = { /* ... */ }; // temporary array for 2 rows

  for y in $range_step(0, Res.rows, 2) @{
    for yi in range(0, 2) {
      for x in $range(0, Res.cols) @{ // residual for two rows
        residual(x, yi /* ... */ Sol, tmp, RHSF);
      }
    }
    for x in $range(0, RHSC.cols) @{ // restrict the residual
      restrict(x, 0 /* ... */ tmp, RHSC);
    }
  }
}
```

A DSL for the V-cycle

- Same high-level description
  - Intel Core i5-4288U
    - CPU
    - AVX
  - :M mapping merges residual and restrict components

	smoother	residual	restrict	interpolate
CPU	18.62	17.08	6.82	10.24
CPU:M	18.62	17.84		10.24
AVX	16.80	16.69	10.15	16.18
AVX:M	16.80	17.26		16.18

Times in ms for finest level of V-cycle; field of 4096x4096, Jacobi as smoother

A DSL for the V-cycle

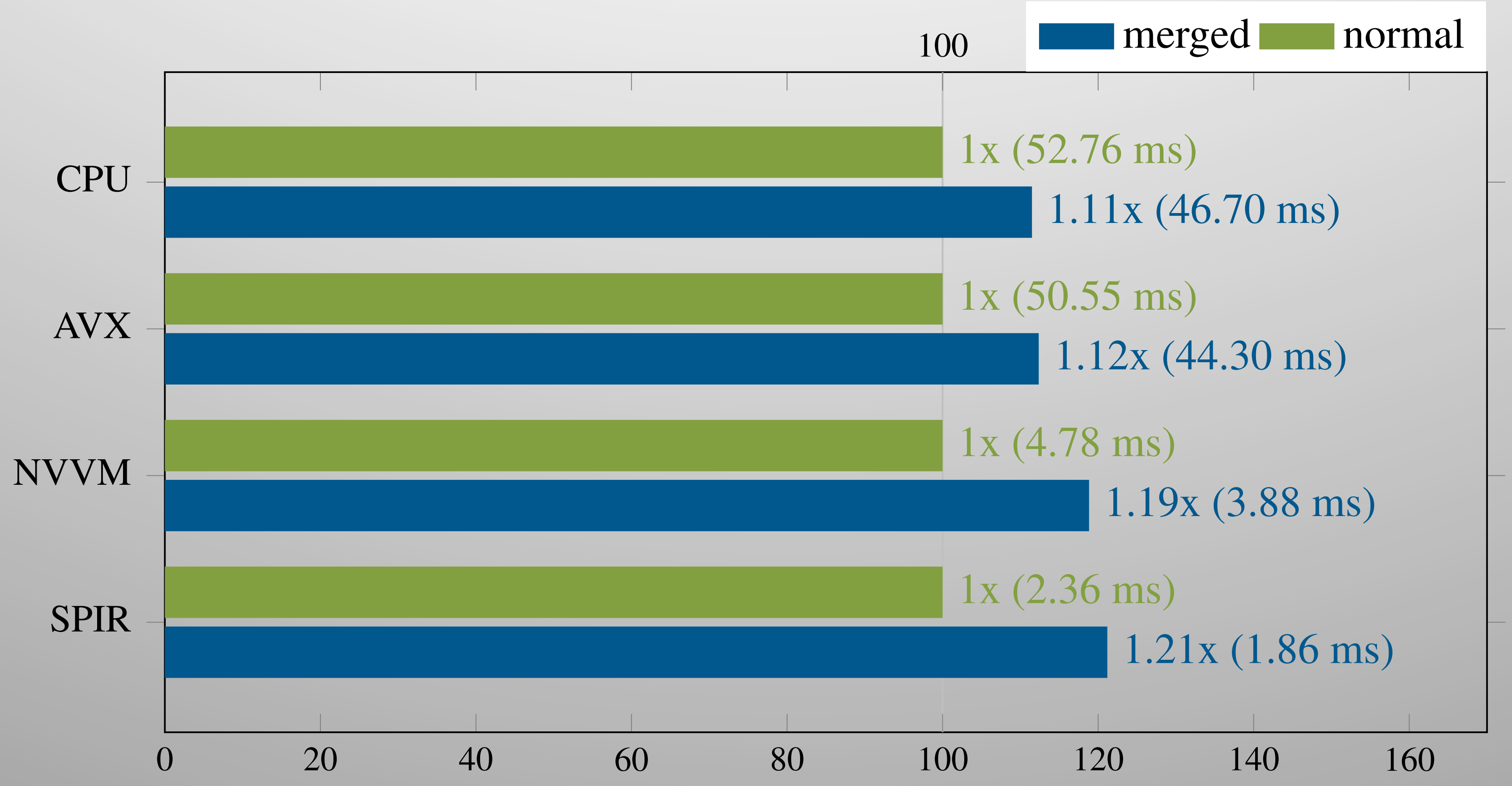
- Same high-level description
  - NVIDIA GeForce GTX 680
    - NVVM
  - AMD Radeon R9 290X
    - SPIR

  - :M mapping merges residual and restrict components

	smoother	residual	restrict	interpolate
NVVM	1.61	1.61	0.55	1.01
NVVM:M	1.61	1.26		1.01
SPIR	0.77	0.77	0.29	0.53
SPIR:M	0.77	0.56		0.53

Times in ms for finest level of V-cycle; field of 4096x4096, Jacobi as smoother

Speedup by Merged Computation



Conclusion

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

Application developer

```
let result = vcycle(jacobi, ...);
```

DSL developer

```
for x, y in @iterate(out) {
  out(x, y) = apply(x, y, field, stencil,
                    bh_lower, bh_upper);
}
```

Machine expert

```
fn iterate(field: Field, ...) -> () {
  let grid = (field.cols, field.rows);
  nvvm(grid, (128, 1, 1), || {
    ...
    body(x, y);
  }
}
```

Thank you for your attention.
Questions?