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

ROLAND LEISSA, KLAAS BOESCHE, and SEBASTIAN HACK, Saarland University, Germany
ARSÈNE PÉRARD-GAYOT, RICHARD MEMBARTH, and PHILIPP SLUSALLEK, Saarland University & DFKI, Germany
ANDRÉ MÜLLER and BERTIL SCHMIDT, Johannes Gutenberg University, Germany

This paper advocates programming high-performance code using partial evaluation. We present a clean-slate programming system with a simple, annotation-based, online partial evaluator that operates on a CPS-style intermediate representation. Our system exposes code generation for accelerators (vectorization/parallelization for CPUs and GPUs) via compiler-known higher-order functions that can be subjected to partial evaluation. This way, generic implementations can be instantiated with target-specific code at compile time.

In our experimental evaluation we present three extensive case studies from image processing, ray tracing, and genome sequence alignment. We demonstrate that using partial evaluation, we obtain high-performance implementations for CPUs and GPUs from *one* language and *one* code base in a *generic way*. The performance of our codes is mostly within 10%, often closer to the performance of multi man-year, industry-grade, manually-optimized expert codes that are considered to be among the top contenders in their fields.

CCS Concepts: • **Software and its engineering → Compilers**; *Parallel programming languages*; • **Hardware → Emerging languages and compilers**; • **Computing methodologies** → Image processing; Ray tracing; • **Applied computing** → Bioinformatics;

Additional Key Words and Phrases: partial evaluation, high-performance, parallelization, vectorization, GPU computing, library design

ACM Reference Format:
Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 119 (November 2018), 30 pages. https://doi.org/10.1145/3276489

1 INTRODUCTION

Writing performance-critical software productively is still a challenging task because performance usually conflicts genericity. Genericity makes programmers productive as it allows them to separate their software into components that can be exchanged and reused independently from each other. To achieve performance however, it is mandatory to instantiate the code with algorithmic variants and parameters that stem from the application domain, and tailor the code towards the target architecture. This requires pervasive changes to the code that destroy genericity.

Authors' addresses: Roland Leißa; Klaas Boesche; Sebastian Hack, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, 66123, {leissa,boesche,hack}@cs.uni-saarland.de; Arsène Pérard-Gayot; Richard Membarth; Philipp Slusallek, Saarland University & DFKI, Saarland Informatics Campus, Saarbrücken, Germany, 66123, perard@cg.uni-saarland.de, richard.membarth@dfki.de, philipp.slusallek@dfki.de; André Müller; Bertil Schmidt, Johannes Gutenberg University, Mainz, Germany, 55128, muellan@uni-mainz.de, bertil.schmidt@uni-mainz.de.

2475-1421/2018/11-ART119
https://doi.org/10.1145/3276489

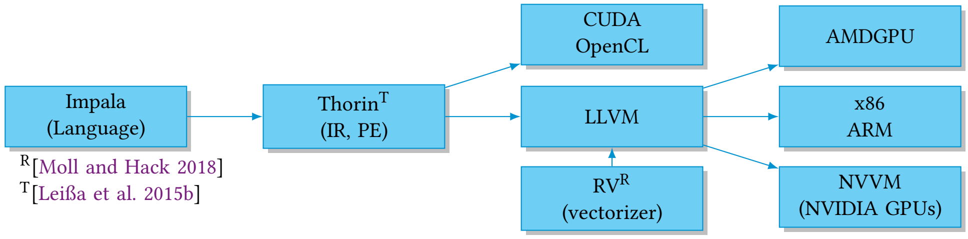

Fig. 1. The AnyDSL System

This instantiation process can in many cases be expressed as a partial evaluation (PE) of the appropriate primitives. PE partitions the input of the program and its execution into two *stages*. In the first stage, the program is *partially* evaluated on the *static* inputs (algorithmic variants, parameter values, target-machine dependent code) to produce the *residuum*. In the second stage, the residuum is *fully* evaluated on the actual, so-called *dynamic* inputs to produce the actual outputs.

PE has been extensively researched over the last decades (see Section 3 for a detailed discussion of related work) and has successfully demonstrated its effectiveness. However, there is hardly a programming system available that provides a built-in partial evaluator and is able to produce high-performance code for modern architectures (including vectorization and GPU mapping). To quote Augustsson [2010]: *O, partial evaluator, where art thou?* In practice, PE is usually "emulated" using metaprogramming techniques (including deep embeddings of domain-specific languages (DSLs)) that have drawbacks over true PE that we discuss in more detail in Section 3.2.

It is hard to pinpoint why PE is not as widely adopted as its success in research suggests, but among the reasons may be the following: Retrofitting an existing implementation of a mature language like C++ with PE is overly complex and unpredictable. For example, the clang C++ compiler has more than 250 different kinds of abstract syntax tree (AST) nodes that would have to be handled by a partial evaluator. Performing PE further down on an intermediate representation (IR) like LLVM [Lattner and Adve 2004] might be too unpredictable in the sense that it is not transparent to the programmer if a feature of the source language is represented in the IR in such a way that it can actually be partially evaluated. Some authors [e.g. Asai 2016] also claim that PE is hard to use because it is not easy for the programmer to understand and control. However, it is not clear if this relates to the difficulties we just discussed or to the inherent difficulties of metaprogramming in general.

In this paper, we counter these problems by presenting a clean-slate design of a programming system called AnyDSL[1] that is built for writing high-performance code using PE. Figure 1 overviews AnyDSL's components and its compilation flow. AnyDSL extends the continuation-passing style (CPS)-based IR Thorin [Leißa et al. 2015b] with a simple *online* partial evaluator. Thorin's design is minimalistic in the sense that its only constructs are continuations and so-called primops (primitive operations, such as arithmetic operations, loads and stores, etc.). This reduces the variety of different constructs the partial evaluator has to handle to a minimum. In fact, our partial evaluator reduces to specializing continuations and performing standard optimizations (like constant folding, algebraic simplifications, etc.). Unlike previous work [Leißa et al. 2015a] on Thorin, our evaluator is entirely driven by implicit and explicit annotations (in the style of Schism filters [Consel 1988]). This makes PE predictable for the programmer because it eliminates the influence of the imprecision of static analyses that guide other evaluators (see Section 3.1). AnyDSL's front end Impala essentially is syntactic sugar for Thorin, for example, providing direct-style functions in addition to continuations.

[1] The name AnyDSL alludes to the first Futamura projection.

Therefore, the “abstraction gap” between Impala and Thorin is rather narrow and the effects of PE are more predictable for the programmer.

1.1 Contributions

In summary, this paper makes the following contributions:

- We present the design of the programming system AnyDSL that is built around a simple *online* partial evaluator for the CPS IR Thorin. Because higher-order functions play a central role in achieving genericity, our partial evaluator provides a default policy to specialize higher-order functions with respect to their higher-order arguments. To provide more fine-grained control over PE, our partial evaluator provides annotations in the form of filters (see Section 4).
- Section 5 shows how specific code generation techniques such as vectorization or mapping to GPUs are exposed as compiler-known higher-order functions. These functions can be treated just as ordinary functions by the programmer as long as their higher-order arguments are constant *after PE* (see Section 2 for an example). This is more flexible (and more elegant as argued in Section 3.4) than pragma-based techniques like OpenMP or similar techniques that rely on higher-order functions such as SYCL [Keryell et al. 2015].
- Section 6 demonstrates that this simple design is sufficient to implement competitive high-performance libraries in a generic way and discusses three non-trivial high-performance case studies from different domains: *image processing*, *ray tracing*, and *genome sequence alignment*. We show that all three applications:
  - can be mapped to different target platforms (CPUs with vectorization and parallelization as well as GPUs) by instantiating a target-independent, shared, generic implementation base with target-specific code. Section 2 showcases this using a simplified version of the image processing code.
  - mostly perform within ~10% (with a ~20% outlier) and sometimes even outperform codes that have been hand-optimized by experts over several years and are considered to be among the top contenders in their respective field: Halide for image processing, Embree (Intel) and OptiX (NVIDIA) for ray tracing, as well as SeqAn, Parasail and NVBIO for sequence alignment.
  - require only a little amount or even no PE annotations by the programmer. In fact, the sequence alignment code has been written by a programmer that did not have a PE background.

2 OVERVIEW

In this section, we give an overview of the writing style encouraged by our system for high-performance code. Figure 2 shows the implementation of a simple image processing stencil code (Section 6.1 discusses a more sophisticated implementation). The similarity to the introductory example[2] of the DSL Halide [Ragan-Kelley et al. 2013] is intentional. The difference is that by using PE, we can implement similar functionality[3] *as a library* without having to write a DSL compiler.

The code defines a 2D blur filter as a sequence of the 1D blur functions `blur_x` and `blur_y` (`|x|e` means $\lambda x.e$). Below its definition, three schedules for `blur_y` are constructed: `seq`, a simple sequential one, `opt`, an optimized one that applies tiling, vectorization, and parallelization, as well as `gpu`, a simple blocked schedule for GPUs. These schedules are just higher-order functions which are passed to `compute` that executes a given loop schedule for a given operator `op` (`blur_y` in the example). The **`for`**-expression is syntactic sugar for capturing a closure for the loop body and

[2] see http://halide-lang.org/

[3] We do not claim to be feature-compatible with Halide.

```
// user code
let blur_x = |x, y| (img.get(x-1, y) + img.get(x, y) + img.get(x+1, y)) / 3;
let blur_y = |x, y| ( blur_x(x, y-1) +  blur_x(x, y) +  blur_x(x, y+1)) / 3;

let seq = combine_xy(range, range);
let opt = tile(512, 32, vec(8), par(16));
let gpu = tile_cuda(32, 4);

compute(out_img_seq, seq, blur_y);
compute(out_img_opt, opt, blur_y);
compute(out_img_gpu, gpu, blur_y);

// implementation
type BinOp  = fn(i32, i32) -> i32;
type Loop1D = fn(i32, i32, fn(i32) -> ()) -> ();
type Loop2D = fn(i32, i32, i32, i32, fn(i32, i32) -> ()) -> ();

fn compute(out: Img, loop: Loop2D, op: BinOp) -> BinOp {
    for x, y in loop(0, 0, img.width, img.height) {
        out.set(x, y, op(x, y))
    }
    |x, y| out.get(x, y)
}

fn combine_xy(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |xa, ya, xb, yb, f|
        loop_y(ya, yb, |y|
            loop_x(xa, xb, |x| f(x, y)))
}

fn tile(xs: i32, ys: i32, loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |xa, ya, xb, yb, f|
        loop_y(0, (yb-ya)/ys, |ly|
            range(ly*ys+ya, (ly+1)*ys+ya, |ry|
                range(0, (xb-xa)/xs, |rx|
                    loop_x(rx*xs+xa, (rx+1)*xs+xa, |lx| f(lx, ry)))))
}

fn tile_cuda(xs: i32, ys: i32) -> Loop2D {
    |xa, ya, xb, yb, f| {
        let (grid, block)  = ((xb - xa, yb - ya, 1), (xs, ys, 1));
        cuda(grid, block, || f(cuda_gid_x(), cuda_gid_y()))
    }
}

fn @vec(vec_length:  i32) -> Loop1D { |a, b, f| vectorize(vec_length, a, b, f) }
fn @par(num_threads: i32) -> Loop1D { |a, b, f| parallel(num_threads, a, b, f) }
```

Fig. 2. Blur in Impala mapped to different hardware

passing it to the **in**-expression (see Section 4.3). We call functions that are invokable like this *generators*. The function combine_xy takes generators for the x- and y-direction and combines them to a 2D generator; tile applies the generator for the y-direction, sets up a tiling loop, and invokes the generator for the x-direction as innermost loop; tile_cuda defines the blocking for data-parallel GPU execution using CUDA [Nickolls et al. 2008].

The optimized schedule opt uses two specific generators vec and par. These are both constructed from two *compiler-known* generators vectorize and parallel. They do not have an implementation in Impala itself but are recognized by the compiler. The semantics of vectorize and parallel is to execute body in a data-parallel (SPMD) way for the domain defined by the interval from a (inclusive) and b (exclusive). The generator vectorize triggers CPU vector code generation using the region vectorizer (RV) [Moll and Hack 2018]; parallel spawns num_threads C++ threads or an automatically balanced number of threads via Intel® Threading Building Blocks (TBB) [Reinders

2007] if num_threads = 0. Similarly, the GPU schedule gpu utilizes the generator cuda, which triggers GPU code generation in the compiler. It is important to understand that vectorize, parallel, and cuda are no pragmas: They behave like any other function in the program. Most notably, they can be captured in closures and passed around just like any other function.[4]

The code in Figure 2 extensively uses higher-order functions to enable *genericity*: Functions like compute or combine_xy can be reused for different stencils or to build more complex pipelines since compute returns again a BinOp. However, performance relies on PE of these higher-order functions. Otherwise, compute's parameter op remains which ends up having to call a closure for every pixel! The partially evaluated code is competitive to the one Halide generates.

To produce this code, it is crucial that the programmer has a precise understanding and is in full control of PE. Because partially evaluating higher-order functions with respect to their functions is imperative to achieve performance, Impala automatically partially evaluates them based on a comprehensive set of rules discussed in Section 4.3. For this reason, the code in Figure 2 does hardly need *explicit* filters. Merely, vec and par must be forced to specialize such that the values for vec_length and num_threads are propagated because these parameters are of zeroth order.

2.1 Relation to DSL Embedding

Halide is a *deeply* embedded DSL that uses C++ as host language. Halide language elements like tile are in fact C++ methods that *construct* the AST of the embedded Halide program. Then, the Halide compiler inspects this AST and generates high-performance code. In our setting, we do not need a dedicated DSL compiler: The implementation of functions like tile, tile_cuda, combine_xy, etc. *constitute* the code the Halide compiler would emit. For example, compute executes a preconstructed loop-schedule loop for a given operator op.

Put another way, PE of higher-order programs enables a specific kind of *shallow* DSL embedding [Hudak 1998]. Following the *tagless interpreter* approach by Carette et al. [2007], the abstract syntax of the embedded program is not represented as a tree of data but as a tree of closures. Each closure constitutes the *semantics* of the particular language element. Partially evaluating this closure tree corresponds to the first Futamura [1982] projection and leads to the same code a syntax-directed compiler would have emitted. This enables the implementation of a DSL in a library of higher-order functions. However, since our embedding approach is shallow, it is not possible to perform a-posteriori program transformations of the embedded programs as permitted by deep embedding approaches, such as Halide or Scala/LMS (see Section 3.3).

3 BACKGROUND AND RELATED WORK

3.1 Partial Evaluation

A common problem of PE is to decide which parts of the program should be specialized. In particular, a partial evaluator which is too greedy might diverge even if the original program terminates for all valid inputs—a phenomenon called *induced divergence* [Katz and Weise 1992]. The following technique [Cook and Lämmel 2011; Futamura 1999], which we also employ, avoids at least obvious endless recursion: The evaluator keeps track of each specialized call site. If the evaluator now needs to specialize a call again, it inserts a call to the already specialized function instead. However, this technique will not prevent infinite expansion of count(0, d) if d is dynamic [Bondorf and Jørgensen 1993]:

```
fn count(i: i32, n: i32) -> i32 { if i < n { count(i+1, n) } else { i } }
```

[4] With the restriction that after PE, their higher-order arguments have to be constants.

Due to the halting problem this dilemma can only be resolved via heuristics. For this reason, our approach requires user annotations. This puts the programmer in total control over which parts of the program should be specialized under which conditions.

The key idea is that the author of a function knows best *when* and *how* that function is to be partially evaluated. Consider the power function in Figure 3d. The annotation `@(`e`)` delimits PE filters. The filter `?n` states that calls to `pow` shall be specialized to all arguments if `n` is a constant. Now, when specializing a call `pow(x', 4)` the partial evaluator creates a specialization by substituting all occurrences of `x` by `x'` and `n` by `4`. To this end, the evaluator runs a series of optimizations (propagating constants, arithmetic simplifications, folding branches, etc.). The result will be `let r = pow(x', 2); r * r` which will again trigger specialization. The filter ensures that this process terminates because `pow`'s termination only depends on `n`. Filters originate from Schism [Consel 1988, 1993b] and have been used in other partial evaluators as well [Asai 2001].

One typically distinguishes between two flavors of PE: *online* and *offline*. *Online* PE [Cook and Lämmel 2011; Ruf 1993; Shali and Cook 2011] specializes a program on the fly and without prior analysis. *Offline* PE on the other hand, first runs a so-called *binding-time analysis (BTA)* [Birkedal and Welinder 1995; Bulyonkov 1993; Consel 1993a; Gengler and Rytz 1992; Jones 1995; Jones et al. 1989; Jørring and Scherlis 1986; Rytz and Gengler 1992] that identifies which parts of the program can be static and which ones must remain dynamic. Then, the actual specializer runs. Filters can either be used in an online evaluator (as we do) or in an offline one and instantiated during the BTA.

As already discussed in Section 2.1, specializing an interpreter for its input, will result in a residuum that is similar to the code a syntax-directed compiler would have emitted (the first Futamura [1982] projection). If you specialize the partial evaluator for the partial evaluator, you will receive a *compiler generator (cogen)*. This is a program that converts an interpreter to an equivalent compiler (the third Futamura [1982] projection); cogen's output—the *generating extension*—is a program that renders for *any* given static input a specialized program that is only parametric in the dynamic inputs. In order to actually generate cogen, the specializer must be self-applicable—a requirement which is hard to realize in practice [Glück 2012]. However, when doing offline PE, it is not a big deal to directly implement cogen [Birkedal and Welinder 1994]. This approach has the advantage that the output language can be different from its input language while cogen itself may even be written in a third language. Moreover, once the generating extension has been produced, it usually performs much better than running an online evaluator. Nonetheless, the applications presented in Section 6 were all compiled (including partially evaluated) within a few dozens of seconds.

Because of the inherent imprecision of static analyses, approaches that rely on BTA might leave too much of the code dynamic. For example, an online evaluator like ours folds `f(23, x)` to `use(42)`—potentially enabling further PE of `use`:

```
fn f(@s: i32, d: i32) {
    let x = if s == 23 { 42 } else { d };
    use(x)
}
```

A straightforward BTA only knows that `s` is static but not what value and therefore merely infers that `x` is dynamic. Furthermore, when dealing with higher-order programs as we do, a BTA requires a control-flow analysis (CFA) [Midtgaard 2012; Nielson and Nielson 1992; Ruf and Weise 1992; Shivers 1991] which might introduce even more imprecision. In addition, a CFA can be quite costly depending on the desired precision. BTAs for imperative programs [Andersen 1994; Das et al. 1995; Grant et al. 2000] have to deal with control-flow [Hornof and Noyé 2000] and pointers [Andersen 1993]. In addition, BTAs for object-oriented languages have to deal with dynamic dispatch and class hierarchies [Schultz et al. 2003].

```
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

Fig. 3. Specializing the power function to its exponent with metaprogramming and partial evaluation [adapted from Leißa et al. 2015a, Figure 3].

In order to fight induced divergence, a number of techniques have been suggested. Similix [Jørgensen 1998] and Schism [Consel 1993b] will not partially evaluate loops with a dynamic exit condition. Some authors [e.g. Jones and Glenstrup 2002; Jones et al. 1993] suggest applying a termination analysis to conservatively ensure termination of PE. Leißa et al. [2015a] propose to partially evaluate everything until the evaluator hits the first dynamic conditional and resume evaluation at the post-dominator. If further evaluation under this conditional is desired, a user annotation can restart evaluation there. This approach does provably not induce divergence, unless a user annotates a recursive call.

While developing the benchmarks for this paper, one major insight on our previous work was that—especially in larger codes—the effects of its termination heuristics were difficult to assess for the programmer. Ultimately we realized that the fact that it guaranteed termination was less important than the ability of the programmer to precisely anticipate what parts of the program are partially evaluated. This gave rise to the filter-based PE approach described in this paper which gives us more fine-grained control over the specialization process and turned out to scale to much more complex applications.

In addition, PE has been used to optimistically optimize dynamic languages like JavaScript[5], Python, R, or Ruby [Fumero et al. 2017; Leopoldseder et al. 2018; Stadler et al. 2016; Würthinger et al. 2017, 2013]. The implementation of the language consists of an interpreter that is aware of the partial evaluator. This means that the programmer must annotate the interpreter in several ways. For example: fields that are candidates for specialization, functions which should not be entered by the evaluator, or the AST data structure such that the evaluator understands which fields are the children of an AST node. After starting the interpreter for a given program, the runtime system might detect after running several iterations of a "hot" loop that the iteration variable was always an integer. Assumptions like these will be specialized by PE. The resulting code is then JIT-compiled to machine code in order to speed up the execution of such "hot code regions". Runtime checks

[5] see also https://prepack.io/

are inserted to verify that the assumptions made still hold as the compiled version runs. If one of these assumptions is no longer correct, the code must be *deoptimized*. This means that control is transferred back to the interpreter.

3.2 Metaprogramming

There are various metaprogramming techniques varying from preprocessors, templates as in C++ or D, program generators[6] [e.g. Pronk et al. 2013; Rathgeber et al. 2017], JIT-compilation techniques like PACXX [Haidl et al. 2016], quotation-based approaches like MetaML [Taha and Sheard 2000], MetaOCaml [Kiselyov 2014] (see Figure 3a), Terra [DeVito et al. 2013] (see Figure 3b), or quasiquotation in Lisp [Bawden 1999], hygienic macro systems [Kohlbecker et al. 1986] like Racket [Tobin-Hochstadt et al. 2011], to deep DSL embedding (see below). MetaML and MetaOCaml stand out because unlike most other metaprogramming techniques their type systems ensure soundness of the multi-stage execution.

One major difference of PE versus metaprogramming is that in metaprogramming, the residuum shows up as a data structure in the metaprogram. For example, in MetaML the expression <42> has type `<int, 'a>` (`'a` is a free type variable as in ML). While this has the advantage that the metaprogram can potentially analyze, rewrite, and optimize the residuum, it often gets in the way of the programmer. Depending on what technique is being used, the programmer has to deal with different languages in the meta and residual part (e.g. C++ templates vs. C++ core) or syntactically intrusive quotation techniques that impede *polyvariance*, that is, make it hard to seamlessly move and reuse code in the meta and residual part (consider Figure 3a and 3b). PE, however, typically delivers polyvariance because PE does not—except for annotations like the ones presented in this paper—materialize in the syntax (and semantics). If set up properly, partial evaluation followed by full evaluation is equal to full evaluation alone. Hence, ignoring PE annotations seamlessly "moves" the evaluation of a piece of code into the residual stage.

In a sense, metaprogramming means to manually implement the generating extension [Veldhuizen 1998]. In fact, Asai et. al. [Asai 2016; Asai and Kameyama 2016] present a BTA to automatically stage-annotate a MetaOCaml program.

3.3 DSL Embedding

Hudak [1998] was one of the first to *embed* a DSL into a host language in order to inherit much of its infrastructure. Carette et al. [2007] suggested embedding a typed language by ordinary functions instead of object terms. Based on this work, Hofer et al. [2008] have described a *polymorphic* embedding that supports multiple interpretations including an optimizer as yet another interpretation. Rompf and Odersky [2010] extend this work further to lightweight modular staging (LMS) (see Figure 3c). LMS essentially uses Scala's type system to express the binding time and therefore requires the programmer to wrap all values that should appear in the residuum using the type constructor `Rep[T]`. Overloading Scala's language constructs gives the programmer the impression to actually use a DSL [Chafi et al. 2010; DeVito et al. 2011; Sujeeth et al. 2011]. The residuum can be in any desired language or a domain-specific program representation like Delite [Brown et al. 2011]. Scala's type inference minimizes the burden to stage-annotate many types via `Rep[T]` vs. just `T`. This makes the type inference a local BTA [see also Palsberg and Schwartzbach 1994]. With LMS however, we would have to implement all boilerplate code to generate the residuum for any data type not directly supported by LMS like `Rep[BVH]`—a type we use in the ray traversal.

[6] program in language A produces program text of language B

Furthermore, polyvariance can only be achieved by introducing type variables for each desired staging combination [Ofenbeck et al. 2017]:[7]

```
def sin_poly[R[_]: IRep](l: R[Double]): R[Double]
```

LMS is similar to Halide or Intel® Array Building Blocks (ArBB) [Newburn et al. 2011]. The DSL embedding is achieved via a *deep embedding* (see Section 2.1). As with metaprogramming, deep embeddings have the advantage that the programmer can manipulate the embedded program. But deep embeddings also have their downsides [see Jovanovic et al. 2014, §1].

Shallowly embedded DSLs like HIPA^{cc} [Membarth et al. 2016] or SYCL, on the other hand, do not have the cited downsides but usually rely on a modified compiler of the host language to achieve performance. In this setting, the modified compiler recognizes domain-specific patterns and manipulates the program representation. An alternative is to use PE in order to remove the overhead of the shallow embedding [Hudak 1998; Leißa et al. 2015a]. This way, the DSL implementer does not have to modify a compiler. For this reason, our work follows this line of research.

3.4 Accelerator Support

Traditionally, programmers have to use dedicated programming environments like CUDA or OpenCL [Stone et al. 2010] to write kernels for accelerators. As a result, it is difficult to share code between the kernel language and the language of the host program. Furthermore, using two (or more) different languages decreases programmer productivity as each language has its own syntax and semantics.

There exist several technologies to tackle this issue. OpenACC[8] provides pragma-based annotations similar to OpenMP[9]. These pragmas allow the programmer to mark a code region for execution on the GPU. SYCL and PACXX [Haidl and Gorlatch 2014; Haidl et al. 2017] on the other, are shallowly embedded DSLs for C++. Similar to Impala's built-in higher-order functions, SYCL and PACXX provide higher-order functions whose lambda expression arguments are offloaded onto an accelerator.

All these approaches have in common that it is difficult to abstract from this offloading. Pragmas are not first-class citizens of the language and cannot be passed to functions. Both SYCL and PACXX require that the function is a lambda expression (a function literal); the programmer cannot pass an arbitrary closure. However, AnyDSL only requires that the passed function is a literal *after* PE. This will be the case due to the implicit filter annotations described in Section 4.3. So it is fine to abstract from these built-ins as demonstrated in Section 2.

3.4.1 Algorithmic Skeletons. *Algorithmic skeletons* or *parallel patterns* represent algorithmic concepts such as *recursive divide & conquer*, *map*, or *reduce* that appear in many algorithms and can be implemented in a parallel way independent of the concrete algorithm [Cole 1988]. There exist many frameworks that embrace parallel patterns like FastFlow [Aldinucci et al. 2011], Thrust [Bell and Hoberock 2011], or SkelCL [Steuwer et al. 2011]. Some compiler IRs like Delite (see above) or Lift [Hagedorn et al. 2018; Steuwer et al. 2017] make extensive use of parallel patterns to simplify code analyses and transformations. Primitives like `parallel` or `vectorize` that AnyDSL provides, can be seen as hardware-specific skeletons that expose certain features of the hardware to the programmer. More high-level skeletons such as a pipeline or divide & conquer can be implemented as a library using AnyDSL. Furthermore, by "carving out" the hardware-dependent part via higher-order functions, the programmer can map the same code to different hardware. Along these lines,

[7] In this context polyvariance is also called *stage polymorhphism*. This programming technique even allows several levels of interpreters to be collapsed like a meta-circular Lisp interpreter [Amin and Rompf 2018].
[8] see https://www.openacc.org/
[9] see https://www.openmp.org/

the function `tile` in Section 2 can be considered as a parallel skeleton that provides tiled iteration over a 2D regular grid.

4 PARTIAL EVALUATION FILTERS

4.1 Basics

As we will see in Section 4.4 it is crucial to not always specialize all arguments if the callee is recursive. In addition, only specializing performance-critical arguments and retaining other parameters helps the programmer to prevent size explosion of the specialized code. Consequently, each parameter $\mathtt{p}_i$ of a function can be annotated with an expression $\mathtt{e}_i$ which controls if the corresponding parameter should be specialized to its associated argument at a given call site:

```
fn f(@(e_1) p_1: T_1, ..., @(e_n) p_n: T_n) -> R { B }
```

An expression $\mathtt{e}_i$ is an ordinary expression that may reference all parameters. Eliding the annotation `@(`$\mathtt{e}_i$`)` means `@(false)`. Additionally, the expression $?e$ yields **`true`** if e is a constant whereas $\$e$ yields e but is never considered constant by $?\$e$. The partial evaluator evaluates at each call site `f(`$\mathtt{a}_1$`, ..., `$\mathtt{a}_n$`)` each argument's corresponding expression $\mathtt{e}_i$ to $\mathtt{s}_i$ by substituting each $\mathtt{p}_i$ with $\mathtt{a}_i$. In doing so, the evaluator folds constants and applies various arithmetic simplifications via standard term rewriting like $a + 0 \equiv a$ or $a == a \equiv$ **`true`**. If any $\mathtt{s}_i$ yields **`true`**[10], the partial evaluator will create a specialization `f'` of `f` where each $\mathtt{p}_i$ is specialized to $\mathtt{a}_i$ if $\mathtt{s}_i$ is **`true`**. Again, the evaluator will fold constants, apply arithmetic simplifications, and also fold branches during specialization. Afterwards, the evaluator will rewrite the call such that it calls `f'` while eliding all $\mathtt{a}_i$ which have been specialized into `f'`. For example,

```
fn f(@(?a) a: i32, @(?a) b: i32, @(b<5) c: i32) -> i32 { a+b+c }
let (x, y) = /* a dynamic pair */;
f(3, x, 5) + f(x, 3, x+y)
```

yields

```
let (x, y) = /* a dynamic pair */;
fn f3x_(@(b<5) c: i32) -> i32 { 3+x+c }
fn f__xy(@(?a) a: i32, @(?a) b: i32) -> i32 { a+b+(x+y) }
f3x_(5) + f__xy(x, 3)
```

In contrast to C++ templates and most other metaprogramming techniques, specialization is performed in a *polyvariant* way. This means that each call site decides on its own which parameters are specialized and which ones remain dynamic. Moreover, note that specialization isn't limited to constant expressions.

Often, functions that are only used and called once appear in the residuum, for example `f3x_` or `f__xy`, if no other part of the program asked for them. However, the partial evaluator will completely *inline* such calls (see Section 4.6). This means that all remaining arguments will be specialized into that function while the call will be substituted with the resulting body.

As a shortcut, the programmer can put a condition in front of the signature. This condition is combined via logical OR with every e_i. Therefore, the following two functions are equivalent:

```
fn @(e) f(@(e_1) p_1: T_1, ..., @(e_n) p_n: T_n) -> R { B }
fn f(@(e_1 | e) p_1: T_1, ..., @(e_n | e) p_n: T_n) -> R { B }
```

Furthermore, the programmer can simply write `@` as syntactic sugar for `@(true)`. Hence, the following function will aways be specialized to its return continuation (see Section 4.2):

```
fn @get_42() -> i32 { 42 }
```

Finally, the programmer may force inlining of a *call* and ignore any filters of the callee: `@@f(args)`.

```
fn f(a: i32, b: i32) -> () {
    for i in range(a, b) {
        if i == 23 {
            continue()
        } else if i == 42 {
            break()
        } else if i == 97 {
            return()
        }
        g(continue);
        print(i)
    }
}
```

(a) Direct-style Impala program

```
fn f(a: i32, b: i32, return: fn() -> !) -> ! {
    fn body(i: i32, continue: fn() -> !) -> ! {
        if i == 23 {
            continue()
        } else if i == 42 {
            break()
        } else if i == 97 {
            return()
        }
        g(continue, G)
        fn G() -> ! { print(i, P) }
        fn P() -> ! { continue() }
    }
    fn break() -> ! { return() }
    range(a, b, body, break)
}
```

(b) Desugared Impala program in CPS

Fig. 4. Direct style and CPS in Impala

4.2 Continuations

To allow library designers to provide non-trivial control-flow patterns, Impala embraces *CPS*. In fact, Impala represents all control flow (including functions) as continuations; *direct-style* functions and function calls are only syntactic sugar for their CPS counterparts. Figure 4 showcases Impala's desugared CPS representation of a **for**-expression that uses unstructured control flow. Note that **break**, **continue**, and **return** are not actually keywords. Impala will usually name the return parameter of a function "**return**" if the programmer uses direct style. However, when using a **for**-expression, Impala names the return parameter of the passed continuation "**continue**". This is more appropriate for the semantic effect when calling this continuation. Additionally, **continue** will not shadow the implicitly declared **return** parameter of the contained function. Similarly, **break** denotes the return continuation that is passed to `range` when using the **for**-expression. This makes **break**, **continue**, and **return** first-class citizens, which can be captured, invoked, or even passed to other functions. This allows the library designer to build her own sophisticated generators while still supporting (potentially multi-leveled) **break** and **continue**. However, application programmers will hardly need explicit continuations beyond familiar **break**, **continue**, and **return** calls.

4.3 Automatic Annotations

In the case studies (Section 2 and Section 6) higher-order functions are a powerful tool to parameterize for algorithmic variants. Yet, one does not want to pay for costly closure allocation at runtime. Instead of manually annotating all higher-order parameters with `@`, Impala automatically annotates every higher-order parameter unless

(1) the parameter is of first-order,
(2) its continuation does not have other higher-order parameters (so it is a second-order continuation), and
(3) its continuation does not have free variables (other than more continuations without free variables).

This strategy transforms the program to *control-flow form* [Leißa et al. 2015b]: All residual continuations will either be global functions (i.e. continuations with a return continuation), or basic blocks (i.e. first-order continuations) that are nested inside them. This means that the resulting residuum will not need costly closures at runtime. See the discussion of `range` in the next section for an example.

[10] Note that s_i may also be **false** or some symbolic expression like $x < 5$.

```
fn @(?a & ?b & ?s)unroll_step(a: i32, @b: i32, @s: i32, f: fn(i32)->()) -> () {
    if a < b { @@f(a); unroll_step(a+s, b, s, f) }
}
// desugared variant
fn @(?a & ?b & ?s)unroll_step(a: i32, @b: i32, @s: i32, f: fn(i32)->(), return: fn()->!) -> ! {
    if a < b { @@f(a); unroll_step(a+s, b, s, f, return) } else { return() }
}
fn @unroll    (a: i32, b: i32,         f: fn(i32)->()) -> () { unroll_step( a, b, 1, f) }
fn @range_step(a: i32, b: i32, s: i32, f: fn(i32)->()) -> () { unroll_step($a, b, s, f) }
fn @range     (a: i32, b: i32,         f: fn(i32)->()) -> () {  range_step( a, b, 1, f) }
```

Fig. 5. unroll, unroll_step, range, and range_step in Impala

4.4 Recursion and Static Arguments

When specializing f, all functions nested inside of its body are rewritten as well and all references to these are updated accordingly. Free variables and hence functions not defined within f stay as they are. For instance, in the following code both v and h appear free in f:

```
fn f(@a: i32, b: i32) -> i32 {
  fn g(i: i32) -> i32 { a + i }
  g(a+b+v) + h(23)
}
f(x, y)
```

Thus, PE yields a *different* g nested inside of fx_ and not f:

```
fn fx_(b: i32) -> i32 {
  fn g(i: i32) -> i32 { x + i }
  g(x+b+v) + h(23)
}
fx_(y)
```

But what happens for recursive calls? In general, when specializing f to f' all recursive uses are considered *free* and thus are *not* rewritten. However, under certain conditions the partial evaluator can rewrite a recursive call to call the new f' instead.

Consider the generator unroll in Figure 5 that iterates from a (inclusive) to b (exclusive) and executes f each time. The second variant, which is just a desugared version of the first one, makes the return continuation explicit. Note that the end b and the step s are explicitly annotated, whereas the body f and **return** are implicitly annotated with @ (see Section 4.3). But these parameters are just passed as corresponding arguments to the recursive call—called *static arguments* [de Santos 1995]. In particular, **return** is a static argument because the direct-style version is tail-recursive. If in a recursive call, all specialized arguments are static, the partial evaluator will rewrite this call to recursively call the specialization instead. Thus, the following code unfolds as follows (using r as the return continuation):

```
      range(0, 4, g, r)
-->   range_step(0, 4, 1, g, r)
-->   unroll_step($0, 4, 1, g, r)
```

Since $0 prevents the evaluator from inlining the whole call site according to unroll_step's filter, only b, s, f, and **return** are specialized to 4, 1, g, and r:

```
fn @(?a) unroll_step'(a i32) -> () {
  if a<4 { @@g(a); unroll_step'(a+1) } else { r() }
}
unroll_step'(0)
```

Note that the recursive call within the specialization calls again unroll_step' because all specialized parameters were static arguments in that call:

$$\texttt{unroll\_step(a+1, 4, 1, g, }\textbf{return}\texttt{)} \quad \Rightarrow \quad \texttt{unroll\_step'(a+1)}$$

If g is a constant, the evaluator will also inline g. The resulting residuum is then akin to a counting loop in static single assignment (SSA) form [Kelsey 1995] because `unroll_step'` is of first order and therefore a basic block; the parameter `a` acts as ϕ-function with arguments `0` and `a+1`.

Invoking `unroll(0, 3, g)` on the other hand yields a complete unrolling:

```
      unroll(0, 3, g, r)
-->   unroll_step(0, 3, 1, g, r)
-->   @@g(0); unroll_step(1, 3, 1, g, r)
-->   @@g(0); @@g(1); unroll_step(2, 3, 1, g, r)
-->   @@g(0); @@g(1); @@g(2); unroll_step(3, 3, 1, g, r)
-->   @@g(0); @@g(1); @@g(2); r()
```

4.5 Summary

In contrast to quotation techniques or program generators, the programmer only needs to annotate code that should be specialized at compile time. Higher-order parameters are automatically annotated according to the strategy presented in Section 4.3. Only if the programmer desires to specialize for non-function parameters, explicit annotations are required. In our case studies, we found three typical scenarios:

(1) Small wrapper and helper functions can be forced to specialize with a simple `@` (like `unroll`, `range_step`, and `range`).
(2) For recursive functions, the programmer should inspect the recursive calls: It is always safe to specialize static arguments (see Section 4.4).
(3) Only if an unfolding of the recursion is desired, the programmer needs to determine *which parameters actually contribute to the termination of the algorithm* and specify an appropriate filter (as for power and `unroll_step`).

4.6 Implementation

We implemented this partial evaluator in C++ on top of Thorin. The evaluator mainly consists of two parts: one component instantiates for each call site the given filter and checks which parameters need to be specialized (~200 lines of code (LoC)); the second component is the actual specializer (~250 LoC). It appropriately specializes the called function while performing the static argument transformation (see Section 4.4).

As already mentioned, the partial evaluator will always inline a sole call to a function. This is important, as it ensures that the programmer actually experiences the behavior expected from his filters. Suppose, in the following example, that termination of PE depends among other things on propagating the constant 42 to use:

```
fn @get_42(return: fn(i32) -> !) -> ! { return(42) }
get_42(|res| use(res, k))
```

Without additional inlining the constant gets stuck in the return continuation and is not propagated to use:

```
(|res: i32| use(res))(42)
```

Note that the return continuation will always be unique when CPS-translating a direct-style program and, hence, will be subject to inlining after PE. Only when programming with explicit continuations can the programmer reuse the same return continuation more than once:

```
let k = |x| /*...*/;
if cond { get_42(k) } else { get_42(k) }
```

Inlining these calls would potentially cause code explosion because k can be an arbitrarily large program. If this is the desired behavior, the programmer can provide a proper filter for k.

Unfortunately, specialization may create continuations that later become unreachable. Whether a function needs to be inlined, may thus depend on an unreachable code elimination (UCE). But inlining in turn opens up opportunities for further PE. Due to this interaction, the partial evaluator is hooked into Thorin's optimizer (~50 LoC). First, common optimizations (including UCE) are performed. Then, the partial evaluator runs for a *constant number of steps*. These two steps are run in a fixed-point loop, which terminates as soon as the evaluator has nothing left to do.

In addition, Thorin already applies constant folding, various arithmetic simplifications, a simple common subexpression elimination (CSE), and a simple UCE on the fly during construction of the IR nodes. This vastly decreases the number of iterations in the fixed-point loops.

Previous filter-based partial evaluators (see Section 3.1) distinguish between complete inlining and specialization of parameters. With our automatic inlining strategy, we found this differentiation to be unnecessary.

4.6.1 Mutable State and Compound Data Structures. Before PE, an SSA-construction phase [Braun et al. 2013] translates all mutable variables whose addresses are not taken into functional variables that are non-destructively updated. Unlike common imperative IRs like LLVM, this is also performed for compound data structures: tuples, structs, and arrays. By doing this, the partial evaluator does not have to evaluate side effects in order to deal with these variables.

Destructive updates on heap-allocated data structures or variables whose addresses are taken are ignored by the evaluator. Nevertheless, this strategy was sufficient for the applications presented in Section 6.

5 ACCELERATOR SUPPORT

Section 2 demonstrates how code generation can be triggered for various accelerators via compiler-known higher-order functions. For this, the compiler needs to massage the code to match the expected format of each accelerator so that the code can be executed by the runtime.

5.1 Parallel

For data-parallel execution on the thread level, Thorin provides the parallel function that applies its body B over the range defined by a and b:

```
for i in parallel(num_threads, a, b) { array(i) = f(x); }
parallel(num_threads, a, b, |i: i32| { array(i) = f(x); }); // desugared variant
```

If num_threads = 0, the runtime will map the execution to the parallel, work-stealing scheduling framework TBB:

```
void anydsl_parallel_for(int a, int b, void* args, void (*fun)(void*, int)) {
    tbb::parallel_for(tbb::blocked_range<int>(a, b), [=] (auto& range) {
        for (int i = range.begin(); i < range.end(); ++i) fun(args, i);
    });
}
```

Note, that the signature of fun is different from B's signature. For this reason, Thorin closure-converts B:[11]

```
    parallel(a, b, |i: i32| { array(i) = f(x); });
--> parallel(a, b, |closure: &(), i: i32| {
        let (array, x) = *bitcast[&(&mut[i32], i32)](closure);
        array(i) = f(x);
    });
```

Other functions like f used within B do not need any further consideration. When emitting code for parallel, Thorin emits a call to anydsl_parallel_for instead. Similarly, AnyDSL utilizes C++ threads, if the programmer asked for a specific number of threads.

[11] Actually, this transformation is performed on Thorin. The presented code is pseudo-code.

5.2 Vectorize

AnyDSL uses the LLVM-based vectorizer RV. It performs best on whole functions without free variables. Therefore, Thorin applies lambda lifting [Johnsson 1985] to eliminate them:[11]

```
    vectorize(vec_length, a, b, align, |i: i32| array(i) = f(x));
--> vectorize(vec_length, a, b, align, |i: i32, array: &mut[i32], x: i32| array(i) = f(x));
```

For performance reasons, Thorin tries to inline all functions like f beforehand to get rid of free functions within the passed function *B*. Then, Thorin employs RV to vectorize *B* to B_simd. By doing this, Thorin tells RV that the loop counter i is continuous and a multiple of align. This implies *a* mod align = 0 and *b* mod align = 0. All other parameters (the former free variables) are marked as uniform[12] with an alignment of align. Finally, Thorin replaces the call to vectorize by an appropriate loop:[11]

```
for i in range_step(a, b, vec_length) { B_simd(i, array, x); }
```

The programmer is responsible for ensuring that the alignment assumptions actually hold. These constraints allow aggressive optimizations within RV for maximal performance. All memory allocations in AnyDSL already fulfill these alignment constraints. If the alignment requirements for the loop bounds are uncertain at compile-time, the programmer can dissect a given loop into a prologue, an epilogue, and a vectorized, appropriately aligned loop:

```
fn @vec(vec_length: i32, a: i32, b: i32, align: i32, body: fn(i32) -> ()) -> () {
    if b - a < vec_length * simd_threshold { // don't vectorize small loops
        range(a, b, body);
    } else {
        let (vec_a, vec_b) = (round_up(a, vec_length), round_up(b-vec_length+1, vec_length));
        range(a, vec_a, body);                                           // prologue
        for i in vectorize(vec_length, vec_a, vec_b, align) { @@body(i); }   // aligned loop
        range(vec_b, b, body);                                           // epilogue
    }
}
```

Similarly, the programmer can implement other abstractions to guarantee the proper alignment of other free variables that might occur inside body.

5.3 GPUs

GPU programs cannot have free variables. For this reason, Thorin also applies lambda lifting to eliminate them:[11]

```
    cuda(device, grid, block, |i: i32| array(i) = f(x));
--> cuda(device, grid, block, |i: i32, array: &mut[i32], x: i32| array(i) = f(x));
```

Thorin then creates a new module that contains the lifted version of the passed function as well as *transitively all the functions* used in it. In the example above, Thorin imports f and potentially other functions. By doing this, Thorin guarantees that the new module does not contain any free functions. Afterwards, Thorin translates the new module to a CUDA program:

```
__device__ int f(int x) { /* ... */ }
__global__ void lambda(int* array, int x) { /* ... */ }
```

The call to cuda in the original module is appropriately rewritten to call the following runtime function:

```
void anydsl_launch_kernel(DevId device, const char* file, const char* kernel, const uint* grid,
  const uint* block, void** args, const uint* sizes, const Type* types, uint num);
```

The generated CUDA program is passed to file while the name of the kernel to be launched is specified via kernel; device specifies the target device for multi-GPU systems; grid and block correspond to the original cuda call in Impala; args, sizes, types, and num encode the signature

[12] This means that the value is the same for all SIMD lanes.

of the kernel function. Similarly, AnyDSL supports OpenCL, NVVM[13], and AMDGPU[14] code generation.

5.3.1 Memory Management and Address Spaces. Memory management is done explicitly using runtime-provided functions. This allows a greater flexibility in the design of domain-specific libraries: Domain knowledge can be leveraged to decide when to allocate and deallocate memory. For accelerators, memory can reside in different address spaces: global, shared, private. This can be manually annotated by the user as a part of the pointer type:

```
let gmem: &mut global [i32] = alloc_cuda(/* size */);
let smem: &mut shared [i32] = reserve_shared(/* size */);
```

If the user doesn't specify an address space, a generic address space is inferred.

6 EVALUATION

In this section, we introduce different domain-specific libraries developed using our framework. We show that:

- from the same code base, we can target both CPUs and GPUs,
- we can specialize the data layout and algorithmic variants, and
- we get good performance compared to state-of-the-art, hand-tuned implementations.

Furthermore, we speculate that our implementations tend to be smaller and more concise although we do not present hard scientific evidence to corroborate this claim. The reason is that while our implementations are more than toy examples, they are still prototypes and do not provide all the features of the implementations we compare against.

6.1 Image Processing

Section 2 already discusses how to implement a simple blur filter and map this code to various hardware platforms. For the sake of presentation, that code is simple. It does not contain techniques for boundary handling and caching temporary results in memory, which are important for performance.

To specify such more sophisticated schedules, we use a function similar to Halide's `produce_root` that splices the computation of one loop schedule `loop1` into the computation of another loop schedule `loop2`. The results of the `loop1` schedule are stored to temporary memory. In case the `loop2` schedule needs neighboring pixels, `loop1` will also be scheduled for those pixels so that all required data is available. `produce_root` provides a schedule that can be fed to the `tile` function introduced earlier. This way, we generate a tiled schedule that fuses the computation of two stencils cached in temporary memory.

```
fn produce_root(loop1: Schedule, loop2: Schedule) -> Schedule {
    |read, xa, ya, xb, yb, write| {
        // allocate memory storing results for loop1 taking tile size and extents into account
        let tmp = /*...*/;
        // step1: schedule loop1 and store to tmp
        for x, y, val in loop1(read,     /*...*/) { write_tmp(x, y, val); }
        // step2: schedule loop2, reading from tmp
        for x, y, val in loop2(read_tmp, /*...*/) { write    (x, y, val); }
    }
}
let blur_x_loop = combine_xy(vec(8), range, blur_x);
let blur_y_loop = combine_xy(vec(8), range, blur_y);
let inner_tile  = produce_root(blur_x_loop, blur_y_loop);
let outer_tile  = par(0);
let schedule    = tile(256, 32, inner_tile, outer_tile);
```

[13] see https://llvm.org/docs/NVPTXUsage.html and https://docs.nvidia.com/cuda/nvvm-ir-spec/
[14] see http://llvm.org/docs/AMDGPUUsage.html

	CPU		GPU	
	Ours	Halide	Ours	Halide
Blur	1.99 (+12%)	1.77	14.22 (+7%)	13.31
Harris Corner	1.14 (+37%)	0.83	8.39 (+44%)	5.83

Fig. 6. Median pixel throughput in **Gpixels/s** (giga pixels per second, higher is better) for the blur filter of Figure 2 on **i32** pixel type and the Harris corner detector for **f32** pixel type, both for an image of 4096 × 4096 pixels. CPU execution on a Skylake i7 6700K and GPU execution on a GeForce GTX 970. The execution time on the GPU for Halide are the average numbers reported by nvprof.

This schedule describes the same strategy as in the original Halide paper [Ragan-Kelley et al. 2012]. However, it does not consider boundary handling and results in out-of-bounds memory accesses when applied to the whole image. A simple technique to handle boundaries is to clamp the indices of every access. However, this leads to many redundant clamp operations. For this reason, we introduce a function `handle_boundaries`. This function generates a schedule that is parametric in its `read` and `write` functions and deals with the given `boundary`. As soon as the partial evaluator specializes the schedule, `iterate_regions` will be unrolled: this will only generate the boundary handling needed in the current region. In particular, no boundary handling will be generated in the vast inner region of the image.

```
fn handle_boundaries(boundary: Boundary, xstep, ystep, loop: Schedule) -> Schedule {
    |read, xa, ya, xb, yb, write| {
        for xl, xu, yl, yu, region in iterate_regions(/*...*/) {
            let boundary_read = boundary(region, read);
            loop(boundary_read, xl, yl, xl, yl, write);
        }
    }
}
handle_boundaries(clamp(img), 256, 32, schedule);
```

Similar to Halide, we provide a function `tile_cuda_at` that generates a fused CUDA schedule for two stencils. This results in a single kernel with the intermediate result stored to shared memory. Passing this schedule to `handle_boundaries` will generate one GPU kernel per image region with specialized boundary handling.

6.1.1 Discussion. To compare our generated code with Halide, we use the same schedule for Halide and our framework. We implement the same schedule as in the original Halide paper and apply boundary handling as described. We compare against a recent Halide version that specializes boundary handling internally as we do. As a more involved example, we implement the Harris corner detector [Harris and Stephens 1988] using the same schedule as provided in the Halide repository. For the Harris corner detector we use in total 7 functions to describe computations, which are wired in 8 statements using calls to the computation functions, and scheduled using 5 statements. Figure 6 compares the execution times of the blur filter and the Harris corner detector compared to Halide. For both applications, the final schedule on the CPU parallelizes the outermost loop using TBB to all available cores and vectorizes the inner loops. We get slightly faster execution times for the blur filter (~10%) and significantly faster execution times for the Harris corner detector (~40%).

The Harris corner detector implementation in Halide takes about the same LoC as our implementation (70 LoC). Our implementation relies on a *library* that provides primitives for scheduling iterations, boundary handling, and helper data structures for images and stencils. These can be reused for other image processing pipelines and correspond to less than 400 LoC.

In contrast, Halide is a full-blown DSL compiler (more than 100 kLoC) that of course provides way more features than our prototype. So, comparing lines of code is not adequate. In Halide, scheduling, boundary handling, etc. is implemented in the code generator. By using PE we leverage the first Futamura projection (as discussed in Section 2) and get around writing a specific DSL compiler. For accelerator mapping, we benefit from the functionality in Thorin that is reusable for other domain-specific libraries.

In comparison to the image processing library we presented in our previous work [Leißa et al. 2015a], the library presented in this section is completely written from scratch. The old library is only capable of applying a single stencil, given by a matrix of coefficients, to the image. The library presented here allows for building more complex image processing pipelines (such as the Harris corner detector) by decoupling the pipeline from the schedule—in a way similar to Halide. For this to be possible, the programmer must also be able to apply more complex functions than coefficient matrices to the image. To achieve this, we abstract from coefficient matrices to operators that are represented by functions. Accordingly, the scheduling functions have to be general enough to take this into account. We also made the scheduling functions more general and modular. For example, `tile` can now take the schedule for the inner and outer loop as higher-order arguments in contrast to our earlier work where these schedules were fixed to `parallel` and `vectorize`.

6.2 Ray Tracing

Physically accurate rendering methods usually rely on ray tracing to perform visibility queries. Ray traversal algorithms take a ray and a 3D scene, and return the closest intersection, if any. These algorithms *traverse* an acceleration data structure, traditionally hierarchical, such as a Kd-tree or a Bounding Volume Hierarchy (BVH). In order to get the best possible performance, kernel developers rewrite the entire algorithm for every possible combination of architecture and traversal variant. With common programming environments such as C++ this is necessary to achieve performance: The programmer has complete control over the generated code, can optimize, fine-tune, and vectorize the kernels manually.

Using AnyDSL, we demonstrate that this is not necessary. We have implemented a set of BVH ray traversal kernels that cover all but one of the variants implemented in state-of-the-art libraries. A BVH traversal kernel depends on several components: The traversal loop, the intersection routines, and the data layouts (in particular, the optimal BVH arity may vary from platform to platform). In Impala, we implemented three different traversal loop schemes (packet, single, and hybrid), a set of intersection routines (ray-box and ray-triangle), and three BVH data layouts (binary, quaternary, and octonary). At compile-time, these components can be combined to form an implementation that is optimized for a specific use case.

On the CPU, vectorization is performed using `vectorize` (see Section 5.2). Previous work [Pérard-Gayot et al. 2017] was using type inference to vectorize kernels, and thus was not able to express complex traversal algorithms. Our kernels additionally use the `parallel` function (see Section 5.1) to distribute the work among CPU cores.

In our setting, there are two different ways to vectorize the traversal: Traversing several rays (a *packet*) at the same time, or traversing a *single* ray with a wide BVH. Packet traversal is the simplest variant because vectorization is triggered from the outside:

```
for i in vectorize(vector_width, 0, num_rays) {
    let ray = load_ray(i);
    /*...*/ // every use of 'ray' will be vectorized
}
```

In the single-ray variant, the intersection routine gets vectorized. The rest of the logic, however, remains scalar:

```
// inside the single-ray traversal loop
let mut hit : [Hit * vector_width];
for i in vectorize(vector_width, 0, bvh.arity) {
    let child = node.child(i);
    hit(i) = intersect_ray_box(ray, child);
}
```

The packet and single-ray traversal loops are combined to form a hybrid variant, which switches between the two loops depending on the SIMD utilization.

In order to improve traversal performance, the nodes of the tree are sorted by their distance from the origin of the ray, so that the leaves containing the closest intersection are found first. There might not be any good choice of traversal order, as child nodes may overlap, or, in the case of the packet traversal, rays might disagree on the order. For this reason, the packet traversal only uses a simple heuristic. Conversely, the single-ray variant uses a sorting network to find a good order. Using PE, we specialize a sorting network for every possible number of intersections:

```
for i in unroll(0, bvh.arity + 1) {
    if i == bvh.arity || num_intersections == i { // generate a sorting network of size 'i'
        sorting_network(i, |p, q| { if stack.dist(p) < stack.dist(q) { stack.swap(p, q) } });
    }
}
```

The sorting network is parametric in its compare & swap function and provides appropriate filters to recursively unfold it:

```
fn @bose_nelson_sort(n: i32, cmp_swp: fn(i32, i32) -> ()) -> () {
    fn @(?n) p_star(i: i32, n: i32) -> () { /*...*/ }
    fn @(?n1 & ?n2) p_bracket(i1: i32, n1: i32, i2: i32, n2: i32) -> () { /*...*/ }
    p_star(0, n)
}
```

Because `i`, the counter of the unrolled loop, is known at compile time, the sorting network can be specialized by the compiler. Additionally, since the sorting networks are provided as higher-order functions, we can select the best-performing network at compile-time, depending on the BVH arity:

```
let sorting_network = match bvh.arity { 4 => bose_nelson_sort, 8 => batcher_sort };
```

The GPU version of the traversal loop uses the `cuda` function (see Section 5.3) to generate accelerator-specific code:

```
let (grid, block) = ((num_rays, 1, 1), (64, 1, 1));
for cuda(grid, block) {
    let id = cuda_gidx();
    if id > num_rays { continue() }
    let ray = load_ray(id);
    // ...
}
```

During execution, the runtime system offloads the kernel to the GPU and runs it transparently.

The user of our ray-tracing library only has to configure a traversal loop in order to generate the variant of his choice:

```
cpu_traverse_single(
    vectorize_avx2(),              // vectorization ISA
    make_ray1_layout(rays, hits), // ray layout type
    make_bvh4(*bvh),              // BVH type
    false,                        // find closest hit
    ray_count,                    // #rays to trace
    1                             // BVH root
);
```

The functions `vectorize_avx2`, `make_ray1_layout`, and `make_bvh4` return a structure containing functions that are used within each traversal loop. For instance, `make_ray1_layout` returns a value of the following type:

	BVH4						BVH8					
	Primary		AO		Diffuse		Primary		AO		Diffuse	
Scene	Ours	Embree	Ours	Embree	Ours	Embree	Ours	Embree	Ours	Embree	Ours	Embree
Sponza	34.73 (-4%)	36.35	76.34 (+8%)	70.66	9.78 (-12%)	11.07	34.84 (-4%)	36.40	76.73 (+13%)	67.81	11.46 (-10%)	12.74
Crown	102.51 (+5%)	97.86	40.28 (-9%)	44.26	19.48 (-12%)	22.20	95.48 (+6%)	89.92	42.12 (-5%)	44.25	21.04 (-9%)	23.16
San-Miguel	22.06 (-4%)	23.04	13.82 (-13%)	15.91	6.46 (-12%)	7.33	18.74 (-2%)	19.13	14.77 (-10%)	16.32	6.98 (-8%)	7.62
Powerplant	49.34 (-3%)	50.63	102.89 (+8%)	95.42	11.86 (-15%)	13.88	43.02 (-4%)	44.82	98.10 (+11%)	88.04	13.29 (-9%)	14.66

(a) CPU: Skylake i7 6700K

	BVH4					
	Primary		AO		Diffuse	
Scene	Vec.	Scalar	Vec.	Scalar	Vec.	Scalar
Sponza	2.75 (+100%)	1.38	5.36 (+101%)	2.66	0.95 (-5%)	1.00
Crown	9.82 (+69%)	5.80	3.65 (+21%)	3.01	1.87 (-2%)	1.91
San-Miguel	2.07 (+94%)	1.07	1.49 (+14%)	1.31	0.72 (-8%)	0.78
Powerplant	4.44 (+71%)	2.59	8.19 (+102%)	4.06	1.09 (-11%)	1.23

(b) CPU: Cortex-A53

	BVH2					
	Primary		AO		Diffuse	
Scene	Ours	Aila et al.	Ours	Aila et al.	Ours	Aila et al.
Sponza	330.41 (-2%)	336.50	884.33 (-2%)	899.62	123.46 (-9%)	135.80
Crown	695.41 (-11%)	778.52	315.09 (-14%)	366.28	133.85 (-19%)	165.71
San-Miguel	181.67 (-5%)	190.78	132.85 (-7%)	142.13	54.06 (-15%)	63.40
Powerplant	465.42 (-12%)	528.21	998.02 (-4%)	1040.93	138.57 (-11%)	155.57

(c) GPU: GeForce GTX Titan X (Maxwell)

Fig. 7. Performance of our traversal kernels on different architectures, in **Mrays/s** (mega rays per second, higher is better). Speed-ups (slow-downs) with respect to the reference are indicated in parentheses. On CPUs (GPUs), we perform 10 (100) warmup iterations and report the average of 50 (500) runs. *Primary* rays start from the camera, *AO* rays compute Ambient Occlusion, and *Diffuse* rays compute purely diffuse reflections.

```
struct RayLayout {
    packet_size: i32,
    input:       fn(i32, i32) -> Ray,
    output:      fn(i32, i32, Hit) -> ()
}
```

We implement different ray layouts by specifying a packet size and providing different functions for the members `input` and `output`.

6.2.1 Discussion. We compared the performance of our kernels with state-of-the-art, manually optimized traversal implementations: Embree [Wald et al. 2014], a CPU-centric library designed by Intel, and on the GPU, the work of Aila et al. [Aila and Laine 2009; Aila et al. 2012], used within the OptiX [Parker et al. 2010] library developed by NVIDIA. On the CPU, profiling indicates that most of the performance difference can be attributed to excessive register spilling, and as a result, increased memory traffic. This difference is likely to even up as the various LLVM backends that are used by Impala mature. We also measured the performance of our packet traversal kernel on an ARM CPU, and use the scalar (non-vectorized) variant of our own kernels as a reference, due to the lack of highly-optimized ray-tracing libraries on this architecture. The results in Figure 7 show that our generated code is mostly within ~10% (with a ~20% outlier) of the two reference libraries, and sometimes performs better. On the ARM architecture, packet tracing can speed the code up to 2×, but slowdowns are to be expected for diffuse rays, due to their incoherent nature.

Overall, we achieve greater portability and flexibility than existing libraries by defining an embedded traversal kernel description language. In fact, Embree is written with vector intrinsics and implements every intersection routine twice: Once for single rays, and once for packets of rays. This situation is clearly not optimal, since intersection routines represent 22% of the total number of lines of code in Embree. Using AnyDSL, intersection routines are only implemented once, and get auto-vectorized by RV. Additionally, OptiX and Embree are two completely incompatible code bases, written in two different languages (CUDA and C++, respectively). Our implementation does not suffer from these issues: From a single code base, we target three different platforms. Maintaining and extending our code is also simpler, since platforms share most of the logic.

6.3 Genome Sequence Alignment

Sequence alignment is a fundamental task in bioinformatics that is mainly used for DNA sequence matching. Optimal sequence alignments can be computed using dynamic programming (DP) algorithms like the widely-used Smith-Waterman algorithm. Because such sequences are usually quite long, there is a broad interest in high-performance sequence-alignment implementations. In practice, different variants of sequence alignment are of interest. Sometimes only the score of the alignment is required, sometimes the optimal alignment itself needs to be reconstructed. This choice and the underlying hardware (CPU vs. GPU) require the usage of different DP algorithms. Finally, the computation of the DP matrix should be parameterizable with different kinds of cost functions.

We use AnyDSL to implement *AnySeq*, a library to map the alignment construction and score computation to CPUs (using multi-threading and vectorization) as well as to GPUs. The algorithm can be parameterized and optimized for different hardware platforms and use cases. This is achieved through function composition, mostly in the form of providing behavior-controlling functions as arguments of higher-order functions.

DP algorithms for sequence alignment usually solve the problem of finding an optimal alignment for two input sequences $S = (s_1 s_2 \ldots s_n)$ and $Q = (q_1 q_2 \ldots q_m)$ by recursively solving three smaller subproblems. For each pair (s_i, q_j) of characters one has to decide if these characters should be aligned or if a gap should be introduced with respect to either one of the input sequences.

In case of optimal local alignments one looks for the best scoring alignment starting at any position (s_i, q_j) and ending at any other position (s_k, q_l) with $i \leq k \leq n, j \leq l \leq m$. Global alignments start at position $(1, 1)$ and end at position (n, m). In case of semi-global alignments, gaps at the beginning and at the end are not penalized.

The optimal alignment score at position (i, j) is given by the recurrence relation:

$$H(i,j) = \max\{H(i-1,j-1) + \sigma(s_i, q_j), E(i,j), F(i,j), v\}$$

where σ is the substitution function. For local alignment, v has to be set to 0, for global alignment it must be set to $-\infty$. In case of a linear gap penalty g we set $E(i,j) = H(i-1,j) - g$ and $F(i,j) = H(i,j-1) - g$. In case of affine gap penalties, where a gap of length k is penalized with $G_o + k \cdot G_e$ we need two additional auxiliary DP matrices E and F:

$$E(i,j) = \max\{E(i-1,j) - G_o, H(i-1,j) - G_e - G_o\}$$
$$F(i,j) = \max\{F(i,j-1) - G_o, H(i,j-1) - G_e - G_o\}$$

Initialization of the first rows and columns of H, E, and F—as well as in what cell(s) to look for the optimal score—depends on whether the alignment shall be global, local, or semi-global. In case of computing an optimal *local alignment*, we look for the best scoring alignment starting at any position (q_i, s_j) and ending at any other position (s_k, q_l) with $i \leq k \leq n, j \leq l \leq m$. The parameter v has to be set to 0 in order to ensure that scores will always be positive. *Global alignments* always start at position $(1, 1)$ and end at position (n, m), meaning that the optimal score can be found in cell $H(n, m)$. The parameter v has to be set to $-\infty$ so that scores are unbounded. In case of *semi-global alignments*, gaps at the beginning and at the end are not penalized, which leads to the same initialization as for local alignments). The optimal score can be found in the last row $(i = n)$ or the last column $(j = m)$ of H.

Note that score-only computations can be performed in linear space and quadratic time with respect to the length of the alignment targets. Actual alignments producing this value can be constructed by tracing back the predecessor information in the DP matrices. In order to avoid quadratic space consumption (which is prohibitive for long DNA strings), the traceback procedure can be implemented in linear space by a divide-and-conquer approach [Hirschberg 1975] that recursively determines optimal midpoints of the DP matrix (at the cost of at most doubling the amount of computed DP cells).

Each cell of H depends on three neighboring cells, which means that relaxing them in parallel can be done along minor diagonals. When relaxing all cells in a submatrix row of H, only the subproblem scores for the row above and one column left of the current cell are needed. If we want to compute submatrices in parallel, the first row and first column of a submatrix must have been computed earlier and the last row and last column must be kept available for the computation of subsequent submatrices (to the right of and below the current one).

6.3.1 Modularization. With the help of higher-order functions, we obtain algorithmic variants. Scoring schemes, memory access, and iteration strategies are all encapsulated in functions that can be interchanged. These functions are often grouped into descriptively named structs to make parametrization more convenient and the resulting code more expressive.

Efficient scalar CPU implementations need radically different iteration and blocking schemes than efficient SIMD-vectorized CPU implementations or efficient GPU code. This in turn leads to different data access patterns and data storage requirements (memory alignment, RAM vs. VideoRAM, GPU shared memory, etc.). A key idea of our design is the usage of data accessor objects for decoupling data access from the actual data storage strategy.

For example, while the resulting data indexing schemes may differ, the underlying relaxation equations can stay the same. It is thus desirable to decouple these two aspects of the algorithm. The basic recurrence relation for one *DP* matrix cell update can be encoded in one function, which in this example is the one used for global alignments:

```
fn relax_global(scoring: Scoring,  // scoring scheme, may access E and F
                prev: PrevScores,  // accessor to previous scores
                next: CharPair     // accessor to sequence letter pair
                ) -> NextStep {    // optimal score and predecessor
    let mut res = NextStep{score: scoring.subst(prev,next), predc: PRED_NO_GAP }; // no gaps
    let sgap = scoring.gap_s(prev,next); // subject gap
    if sgap > res.score {
        res.score = sgap;
        res.predc = PRED_SKIP_S;
    }
    let qgap = scoring.gap_q(prev,next); // query gap
    if qgap > res.score {
        res.score = qgap;
        res.predc = PRED_SKIP_Q;
    }
    res // return maximum over all three choices
}
```

Member functions of type `PrevScores` can be used to access scoring information of the three ancestral subproblems. If no gap should be introduced, the scoring substitution function is used to determine the cost. In case of affine gap penalties the functions `gap_s` and `gap_q` will access the auxiliary matrix elements from E and F, for linear penalties they return a constant. Objects of type `NextStep` store the maximum score for the current cell and what previous subproblem was chosen as ancestor. The ancestor information encoded in `res.predc` is sometimes needed for the innermost level of the traceback since recursion on subsequences is only done if the subsequence sizes exceed a hardware-specific threshold. Whether any of the functions accesses main memory, GPU memory, or simply returns a constant is determined based on the algorithm parametrization at compile time. The relaxation order (which for the recursive traceback is reversed for half of the

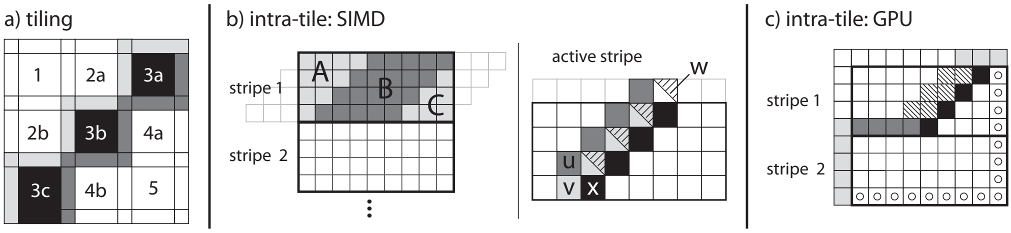

Fig. 8. (a) Tiled relaxation: cells marked light gray need to be communicated to the currently active, black blocks. Dark gray cells need to be stored for the blocks on the next diagonal. Blocks on diagonals can be relaxed in parallel as indicated by the numbers. (b) Left: division of *DP* matrix into stripes shown for the case of $l = 4$ vector lanes. Only part B needs all vector lanes. Right: the black vector x marks the currently active diagonal, u (dark gray), v (light gray) and w (hatched) contain the values of the ancestral subproblems. (c) GPU: Values of the cells in gray and hatched cells are located in shared memory. Light gray indicates values still needed for the current tile. Dark gray indicates values needed for the next stripe (re-use memory previously claimed by initialization cells that are no longer needed). The circles indicate values that need to be written to global memory after the current tile has been completed.

DP matrix) and subproblem sizes are also determined based on algorithmic parameters and the target hardware platform.

Access to the values of previously computed subproblems as well as to the input sequence characters is also abstracted by function calls. This makes it possible to vary intermediate result storage strategies, indexing, and blocking schemes independently from other parts of the algorithm:

```
struct MutableSequence {     // read/write access to a sequence of characters
    len:      fn() -> Index,          at:      fn(Index) -> Char,
    write:    fn(Index,Char) -> (), release: fn() -> ()
}
struct Scores {              // access to alignment scores
    prev:     fn() -> PrevScores,          at:      fn(IndexPair) -> Score,
    update:   fn(IndexPair, Score) -> (), release: fn() -> ()
}
```

Indexing/blocking schemes are defined by generators that can be used in GPU-specific code blocks as well as CPU code.

6.3.2 CPU Parallelization. We compute the values of different DP submatrix blocks concurrently with t CPU threads, relaxing *DP* matrix cells in parallel along diagonals. In the non-vectorized version, cells within a block will be relaxed in row major order whereas submatrix blocks on a diagonal will be relaxed in parallel (see Figure 8a). We have decoupled indexing schemes from most of the other aspects of the alignment algorithms by means of generator functions. Also, if tiles are to be processed serially or in parallel is determined with special generator functions called *schedules*.

A second level of CPU parallelization can be achieved by SIMD vectorization. We partition the *DP* matrix into tiles that are processed in parallel along diagonals as described before. Each tile is computed by one CPU thread and further divided into stripes of height l along the first dimension which corresponds to the input sequence Q. The stripe heights are set to the number of SIMD vector lanes. If the size of the input sequence Q is not a multiple of l, we simply pad it with dummy characters whose substitution score is 0.

Parallelization is done over diagonals within a stripe as shown in Figure 8b. We have to distinguish three cases, labeled A, B, and C in the left part of the figure. In both cases A and C, some vector

		CPU			GPU	
		Ours	SeqAn	Parasail	Ours	NVBIO
Score only	linear	11.9 (-3%)	12.3	n/a	148 (+10%)	135
	affine	10.8 (-8%)	11.8	11.0	133 (-3%)	136
Traceback	linear	8.1 (-9%)	8.9	n/a	112 (+5%)	107
	affine	7.7 (+11%)	n/a	6.9	106 (+4%)	102

Fig. 9. Median runtime performance in **GCUPS** (giga cell updates per second, higher is better) for aligning pairs of six DNA sequences with 4.4 to 50 million characters. CPU execution on two Xeon E5-2683v4 CPUs with 32 threads and GPU execution on a Titan Xp.

lanes are not needed. All three cases are handled separately to avoid bounds checking which would introduce branch divergence.

To avoid the substitution score lookup for each cell of H we use precomputed *profiles* for the sequence Q. For a local stripe at global position (i, j) the profile stores the substitution scores of the characters in range $q_i ... q_{i+l-1}$ with any other character from the alphabet Σ.

At any given time l values of matrices H, E, and F and the substitution scores from the current profile are accessed simultaneously. Figure 8b sketches how the values in H are accessed using four vectors. For each diagonal, the vectors u, v, w contain the values of the ancestral subproblems and the results of the current diagonal will be written to x. In the next step we set $u = w$, $v = x$, $w = x$ and shuffle w one position up. Similar schemes can be used for the other matrices.

6.3.3 GPU Parallelization. In the GPU variant the host starts a GPU kernel for each diagonal of tiles. The kernel uses a one-dimensional grid of thread-blocks where each block computes one matrix tile. Each tile is further divided into stripes which are computed in sequence by a thread-block. Within a stripe, threads compute diagonals in parallel. As in the vectorized CPU code, we divide the function that computes one stripe into three parts in order to avoid branch divergence.

Again, data accessors are used to manage memory buffers that can now point to global or block-local shared GPU memory. In order to enable coalesced memory access we have to use memory access patterns different from the CPU code. Since memory access is abstracted by functions, it is easy to exchange access schemes for the CPU and the GPU code. Device-independent code, like the core relaxation function, remains the same, because it only calls those memory access functions to read and write values.

Before starting a block of threads, the segments of the input sequences that are needed for the current tile are stored in block-local shared memory. The values of the row immediately above the current stripe are needed to access ancestral problems. Also, the values in the bottom-most row of the current stripe will be needed by the next stripe. Since we are relaxing in diagonals along the stripe, we re-use the memory cells with the values of the uppermost row that are no longer needed and store the results of the last row in them. The buffer needed for this does also reside in shared memory. This way, we can compute all stripes within a tile in succession without starting a new kernel. The last row and column of the current tile are always written back to global memory (see Figure 8c).

6.3.4 Discussion. We compare the execution time of our implementation against three state-of-the-art, high-performance sequence alignment libraries: SeqAn [Döring et al. 2008] 2.3.0, Parasail [Daily 2016] 2.0, and NVBIO [Pantaleoni and Subtil 2014] 1.1. These libraries contain manually tuned SIMD or CUDA codes in contrast to the unified codebase of our Impala implementation that utilizes the built-in `vectorize`, `parallel`, and `cuda` generators. Figure 9 reports median execution times per pair alignment.

The performance of our implementation closely matches that of SeqAn, Parasail, and NVBIO. At the same time our code base is smaller and was developed over the course of weeks whereas the other libraries have been fine-tuned over the course of several years.

It is also worth noting that retrofitting libraries like SeqAn with GPU parallelization is notoriously difficult since almost all aspects of such a complex library need to be adapted to support different data layouts and blocking schemes. This requires also the usage of another infrastructure such as *nvcc* to compile CUDA code.

The development of AnySeq was greatly simplified by being able to support different hardware architectures within one unified programming environment. Additional glue and setup code that is usually required to interface CPU code with GPU kernels could be mainly reduced to writing data accessors. Also, to achieve the same level of genericity in C++ one would usually need to write a substantial amount of involved and hard to diagnose template code while we only needed to write regular functions in Impala.

7 CONCLUSIONS

This paper presents a novel system for programming high-performance libraries using PE. We present the design and implementation of a simple online partial evaluator that is implemented on a CPS-based IR. A cornerstone of this system is to expose accelerator code generation by compiler-known higher-order functions. Via PE, target-independent code can be instantiated with target-specific kernels. We demonstrate that the simple design of our system is sufficient to build high-performance applications that compile to CPUs and GPUs from the same code base. Their performance is mostly within 10% to the performance of multi man-year, industry-grade, manually-optimized expert codes that are considered to be among the top contenders in their fields.

ACKNOWLEDGMENTS

This work is supported by the Federal Ministry of Education and Research (BMBF) as part of the Metacca and ProThOS projects as well as by the Intel Visual Computing Institute (IVCI) and Cluster of Excellence on Multimodal Computing and Interaction (MMCI) at Saarland University.

REFERENCES

Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics 2009, New Orleans, Louisiana, USA, August 1-3, 2009*. 145–149. https://doi.org/10.2312/EGGH/HPG09/145-150

Timo Aila, Samuli Laine, and Tero Karras. 2012. *Understanding the Efficiency of Ray Traversal on GPUs - Kepler and Fermi Addendum*. Technical Report NVR-2012-002. NVIDIA Technical Report.

Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. 2011. Accelerating Code on Multi-cores with FastFlow. In *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*. 170–181. https://doi.org/10.1007/978-3-642-23397-5_17

Nada Amin and Tiark Rompf. 2018. Collapsing towers of interpreters. *PACMPL* 2, POPL (2018), 52:1–52:33. https://doi.org/10.1145/3158140

L.O Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph.D. Dissertation. Københavns Universitet. Datalogisk Institut.

Lars Ole Andersen. 1993. Binding-Time Analysis and the Taming of C Pointers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark, June 14-16, 1993*. 47–58. https://doi.org/10.1145/154630.154636

Kenichi Asai. 2001. Integrating Partial Evaluators into Interpreters. In *Semantics, Applications, and Implementation of Program Generation, Second International Workshop, SAIG 2001, Florence, Italy, September 6, 2001, Proceedings*. 126–145. https://doi.org/10.1007/3-540-44806-3_8

Kenichi Asai. 2016. Toward introducing binding-time analysis to MetaOCaml. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 97–102. https://doi.org/10.1145/2847538.2847547

Kenichi Asai and Yukiyoshi Kameyama. 2016. Automatic Staging via Partial Evaluation Techniques. In *7th International Symposium on Symbolic Computation in Software Science, SCSS 2016, Tokyo, Japan, March 28-31, 2016.* 1–13. http://www.easychair.org/publications/paper/262500

Lennart Augustsson. 2010. O, partial evaluator, where art thou?. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18-19, 2010.* 1–2. https://doi.org/10.1145/1706356.1706357

Alan Bawden. 1999. Quasiquotation in Lisp. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1.* 4–12.

N. Bell and J. Hoberock. 2011. Thrust: A productivity-oriented library for CUDA. In *GPU Computing Gems Jade Edition* (1st ed.), Wen-mei W. Hwu (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Lars Birkedal and Morten Welinder. 1994. Hand-Writing Program Generator Generators. In *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94, Madrid, Spain, September 14-16, 1994, Proceedings.* 198–214. https://doi.org/10.1007/3-540-58402-1_15

Lars Birkedal and Morten Welinder. 1995. Binding-Time Analysis for Standard ML. *Lisp and Symbolic Computation* 8, 3 (1995), 191–208.

Anders Bondorf and Jesper Jørgensen. 1993. Efficient Analysis for Realistic Off-Line Partial Evaluation. *J. Funct. Program.* 3, 3 (1993), 315–346. https://doi.org/10.1017/S0956796800000769

Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings.* 102–122. https://doi.org/10.1007/978-3-642-37051-9_6

Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011.* 89–100. https://doi.org/10.1109/PACT.2011.15

Mikhail A. Bulyonkov. 1993. Extracting Polyvariant Binding Time Analysis from Polyvariant Specializer. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark, June 14-16, 1993.* 59–65. https://doi.org/10.1145/154630.154637

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2007. Finally Tagless, Partially Evaluated. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings.* 222–238. https://doi.org/10.1007/978-3-540-76637-7_15

Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. 2010. Language virtualization for heterogeneous parallel computing. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA.* 835–847. https://doi.org/10.1145/1869459.1869527

Murray Cole. 1988. *Algorithmic skeletons : a structured approach to the management of parallel computation.* Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/11997

Charles Consel. 1988. New Insights into Partial Evaluation: the SCHISM Experiment. In *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings.* 236–246. https://doi.org/10.1007/3-540-19027-9_16

Charles Consel. 1993a. Polyvariant Binding-Time Analysis For Applicative Languages. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark, June 14-16, 1993.* 66–77. https://doi.org/10.1145/154630.154638

Charles Consel. 1993b. A Tour of Schism: A Partial Evaluation System For Higher-Order Applicative Languages. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark, June 14-16, 1993.* 145–154. https://doi.org/10.1145/154630.154645

William R. Cook and Ralf Lämmel. 2011. Tutorial on Online Partial Evaluation. In *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011.* 168–180. https://doi.org/10.4204/EPTCS.66.8

Jeff Daily. 2016. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC bioinformatics* 17, 1 (2016), 1.

Manuvir Das, Thomas W. Reps, and Pascal Van Hentenryck. 1995. Semantic Foundations of Binding Time Analysis for Imperative Programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, USA, June 21-23, 1995.* 100–110. https://doi.org/10.1145/215465.215569

André L. M. de Santos. 1995. *Compilation by Transformation in Non-Strict Functional Languages.* Ph.D. Dissertation. University of Glasgow.

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013.* 105–116. https://doi.org/10.1145/2462156.2462166

Zach DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011.* 9:1–9:12. https://doi.org/10.1145/2063384.2063396

Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. 2008. SeqAn An efficient, generic C++ library for sequence analysis. *BMC Bioinformatics* 9, 1 (09 Jan 2008), 11. https://doi.org/10.1186/1471-2105-9-11

Juan José Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017.* 60–73. https://doi.org/10.1145/3050748.3050761

Yoshihiko Futamura. 1982. Parital Computation of Programs. In *RIMS Symposium on Software Science and Engineering, Kyoto, Japan, 1982, Proceedings.* 1–35. https://doi.org/10.1007/3-540-11980-9_13

Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (01 Dec 1999), 381–391. https://doi.org/10.1023/A:1010095604496 Revision of the 1971 paper.

Marc Gengler and M. Rytz. 1992. A Polyvariant Binding Time Analysis Handling Partially Known Values. In *Actes WSA'92 Workshop on Static Analysis (Bordeaux), September 1992, Laboratoire Bordelais de Recherche en Informatique (LaBRI), Proceedings.* 322–330.

Robert Glück. 2012. A self-applicable online partial evaluator for recursive flowchart languages. *Softw., Pract. Exper.* 42, 6 (2012), 649–673. https://doi.org/10.1002/spe.1086

Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. 2000. DyC: an expressive annotation-directed dynamic compiler for C. *Theor. Comput. Sci.* 248, 1-2 (2000), 147–199. https://doi.org/10.1016/S0304-3975(00)00051-7

Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018.* 100–112. https://doi.org/10.1145/3168824

Michael Haidl and Sergei Gorlatch. 2014. PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM 2014, New Orleans, LA, USA, November 17, 2014.* 1–11. https://doi.org/10.1109/LLVM-HPC.2014.9

Michael Haidl, Simon Moll, Lars Klein, Huihui Sun, Sebastian Hack, and Sergei Gorlatch. 2017. PACXXv2 + RV: An LLVM-based Portable High-Performance Programming Model. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2017, Denver, CO, USA, November 13, 2017.* 7:1–7:12. https://doi.org/10.1145/3148173.3148185

Michael Haidl, Michel Steuwer, Tim Humernbrum, and Sergei Gorlatch. 2016. Multi-stage programming for GPUs in C++ using PACXX. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit, GPGPU@PPoPP 2016, Barcelona, Spain, March 12 - 16, 2016.* 32–41. https://doi.org/10.1145/2884045.2884049

Christopher G. Harris and Mike Stephens. 1988. A Combined Corner and Edge Detector. In *Proceedings of the Alvey Vision Conference, AVC 1988, Manchester, UK, September, 1988.* 1–6. https://doi.org/10.5244/C.2.23

Daniel S. Hirschberg. 1975. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Commun. ACM* 18, 6 (1975), 341–343. https://doi.org/10.1145/360825.360861

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of dsls. In *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings.* 137–148. https://doi.org/10.1145/1449913.1449935

Luke Hornof and Jacques Noyé. 2000. Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity. *Theor. Comput. Sci.* 248, 1-2 (2000), 3–27. https://doi.org/10.1016/S0304-3975(00)00048-7

Paul Hudak. 1998. Modular domain specific languages and tools. In *Proceedings of the Fifth International Conference on Software Reuse, ICSR 1998, Victoria, BC, Canada, June 2-5, 1998.* 134–142. https://doi.org/10.1109/ICSR.1998.685738

Thomas Johnsson. 1985. Lambda Lifting: Treansforming Programs to Recursive Equations. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings.* 190–203. https://doi.org/10.1007/3-540-15975-4_37

Neil D. Jones. 1995. Special Address: MIX ten years after. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, USA, June 21-23, 1995.* 24–38. https://doi.org/10.1145/215465.215468

Neil D. Jones and Arne J. Glenstrup. 2002. Program generation, termination, and binding-time analysis. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.* 283. https://doi.org/10.1145/581478.581505

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice Hall.

Neil D. Jones, Peter Sestoft, and Harald Søndergaard. 1989. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation* 2, 1 (1989), 9–50.

Jesper Jørgensen. 1998. SIMILIX: A Self-Applicable Partial Evaluator for Scheme. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998.* 83–107. https://doi.org/10.1007/3-540-47018-2_3

Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986.* 86–96. https://doi.org/10.1145/512644.512652

Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-yang: concealing the deep embedding of DSLs. In *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014.* 73–82. https://doi.org/10.1145/2658761.2658771

Morry Katz and Daniel Weise. 1992. Towards a New Perspective on Partial Evaluation. In *PEPM'92, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Fairmont Hotel, San Francisco, CA, USA, June 19-20, 1992, Proceedings (TR YALEU/DCS/RR-909).* 29–37.

Richard Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995.* 13–23. https://doi.org/10.1145/202529.202532

Ronan Keryell, Ruyman Reyes, and Lee Howes. 2015. Khronos SYCL for OpenCL: a tutorial. In *Proceedings of the 3rd International Workshop on OpenCL, IWOCL 2015, Palo Alto, California, USA, May 12-13, 2015.* 24:1. https://doi.org/10.1145/2791321.2791345

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings.* 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. 1986. Hygienic Macro Expansion. In *LISP and Functional Programming.* 151–161.

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA.* 75–88. https://doi.org/10.1109/CGO.2004.1281665

Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. 2015a. Shallow embedding of DSLs via online partial evaluation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, Pittsburgh, PA, USA, October 26-27, 2015.* 11–20. https://doi.org/10.1145/2814204.2814208

Roland Leißa, Marcel Köster, and Sebastian Hack. 2015b. A graph-based higher-order intermediate representation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015.* 202–212. https://doi.org/10.1109/CGO.2015.7054200

David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018.* 126–137. https://doi.org/10.1145/3168811

Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2016. HIPA^{CC}: A Domain-Specific Language and Compiler for Image Processing. *IEEE Trans. Parallel Distrib. Syst.* 27, 1 (2016), 210–224. https://doi.org/10.1109/TPDS.2015.2394802

Jan Midtgaard. 2012. Control-flow analysis of functional programs. *ACM Comput. Surv.* 44, 3 (2012), 10:1–10:33. https://doi.org/10.1145/2187671.2187672

Simon Moll and Sebastian Hack. 2018. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018.* 543–556. https://doi.org/10.1145/3192366.3192413

Chris J. Newburn, Byoungro So, Zhenying Liu, Michael D. McCool, Anwar M. Ghuloum, Stefanus Du Toit, Zhi-Gang Wang, Zhaohui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. 2011. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011.* 224–235. https://doi.org/10.1109/CGO.2011.5764690

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *ACM Queue* 6, 2 (2008), 40–53. https://doi.org/10.1145/1365490.1365500

Hanne Riis Nielson and Flemming Nielson. 1992. *Semantics with applications - a formal introduction.* Wiley.

Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2017. Staging for generic programming in space and time. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017.* 15–28. https://doi.org/10.1145/3136040.3136060

Jens Palsberg and Michael I. Schwartzbach. 1994. Binding-time Analysis: Abstract Interpretation versus Type Inference. In *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France.* 277–288. https://doi.org/10.1109/ICCL.1994.288372

Jacopo Pantaleoni and Nuno Subtil. 2014. NVBIO. http://nvlabs.github.io/nvbio/. [Online; accessed 06-October-2017].

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David P. Luebke, David K. McAllister, Morgan McGuire, R. Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. *ACM Trans. Graph.* 29, 4 (2010), 66:1–66:13. https://doi.org/10.1145/1833351.1778803

Arsène Pérard-Gayot, Martin Weier, Richard Membarth, Philipp Slusallek, Roland Leißa, and Sebastian Hack. 2017. RaTrace: simple and efficient abstractions for BVH ray traversal algorithms. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017.* 157–168. https://doi.org/10.1145/3136040.3136044

Sander Pronk, Szilárd Páll, Roland Schulz, Per Larsson, Pär Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. 2013. GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics* 29, 7 (2013), 845–854. https://doi.org/10.1093/bioinformatics/btt055

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12. https://doi.org/10.1145/2185520.2185528

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013.* 519–530. https://doi.org/10.1145/2462156.2462176

Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2017. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Softw.* 43, 3 (2017), 24:1–24:27. https://doi.org/10.1145/2998441

James Reinders. 2007. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism.* O'Reilly. http://www.oreilly.com/catalog/9780596514808/index.html

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010.* 127–136. https://doi.org/10.1145/1868294.1868314

Erik Ruf and Daniel Weise. 1992. Improving the Accuracy of Higher-Order Specialization using Control Flow Analysis. In *PEPM'92, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Fairmont Hotel, San Francisco, CA, USA, June 19-20, 1992, Proceedings (TR YALEU/DCS/RR-909).* 67–74.

Erik Steven Ruf. 1993. *Topics in Online Partial Evaluation.* Ph.D. Dissertation. Stanford University, Stanford, CA, USA. UMI Order No. GAX93-26550.

Bernhard Rytz and Marc Gengler. 1992. A Polyvariant Binding Time Analysis. In *PEPM'92, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Fairmont Hotel, San Francisco, CA, USA, June 19-20, 1992, Proceedings (TR YALEU/DCS/RR-909).* 21–28.

Ulrik Pagh Schultz, Julia L. Lawall, and Charles Consel. 2003. Automatic program specialization for Java. *ACM Trans. Program. Lang. Syst.* 25, 4 (2003), 452–499. https://doi.org/10.1145/778559.778561

Amin Shali and William R. Cook. 2011. Hybrid partial evaluation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011.* 375–390. https://doi.org/10.1145/2048066.2048098

Olin Grigsby Shivers. 1991. *Control-flow Analysis of Higher-order Languages of Taming Lambda.* Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA. UMI Order No. GAX91-26964.

Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R language execution via aggressive speculation. In *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016.* 84–95. https://doi.org/10.1145/2989225.2989236

Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska,* *USA, 16-20 May 2011 - Workshop Proceedings*. 1176–1182. https://doi.org/10.1109/IPDPS.2011.269

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. 74–85. http://dl.acm.org/citation.cfm?id=3049841

John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*. 609–616.

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 132–141. https://doi.org/10.1145/1993498.1993514

Todd L. Veldhuizen. 1998. C++ Templates as Partial Evaluation. *CoRR* cs.PL/9810010 (1998), 1–13. http://arxiv.org/abs/cs.PL/9810010

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics* 33, 4, Article 143 (2014), 8 pages. https://doi.org/10.1145/2601097.2601199

Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 662–676. https://doi.org/10.1145/3062341.3062381

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. 187–204. https://doi.org/10.1145/2509578.2509581