Target-Specific Refinement of Multigrid Codes

Richard Membarth and Philipp Slusallek
German Research Center for Artificial Intelligence
Computer Graphics Lab, Saarland University
Intel Visual Computing Institute
{richard.membarth,philipp.slusallek}@dfki.de

Marcel Köster, Roland Leißa, and Sebastian Hack
Compiler Design Lab, Saarland University
Intel Visual Computing Institute
{koester,leissa,hack}@cs.uni-saarland.de

***Abstract*—This paper applies partial evaluation to stage a stencil code Domain-Specific Language (DSL) onto a functional and imperative programming language. Platform-specific primitives such as scheduling or vectorization, and algorithmic variants such as boundary handling are factored out into a library that make up the elements of that DSL. We show how partial evaluation can eliminate all overhead of this separation of concerns and creates code that resembles hand-crafted versions for a particular target platform. We evaluate our technique by implementing a DSL for the V-cycle multigrid iteration. Our approach generates code for AMD and NVIDIA GPUs (via SPIR and NVVM) as well as for CPUs using AVX/AVX2 alike from the same high-level DSL program. First results show that we achieve a speedup of up to $3\times$ on the CPU by vectorizing multigrid components and a speedup of up to $2\times$ on the GPU by merging the computation of multigrid components.**

***Index Terms*—Multigrid codes, partial evaluation, domain-specific language.**

I. Introduction

Common imperative programming languages (C, Fortran, etc.) provide simple abstractions of fundamental features of a processor: variables to abstract from different storage locations (registers, stack), expression syntax to abstract from linearization, function calls to abstract from calling conventions, etc. These abstractions are essential to write portable and maintainable code productively. It is the job of the compiler to remove the overhead these abstractions cause and map them to efficient machine code.

However, to achieve high performance, the compiler's code optimizations are often not good enough. Many machine properties such as SIMD instructions or the memory hierarchy are not orchestrated well by modern compilers. There are basically two reasons for this: First, it requires intricate knowledge of the target architecture to come up with a transformation strategy. Second, it requires domain knowledge to justify the correctness of the transformation.

Many performance-critical codes are manually "optimized" towards a specific target architecture. This is of course error-prone, results in unportable code, and is a maintenance and debugging nightmare. Therefore, it is common practice to use DSLs or other program generation tools to ease this process.

To make writing DSLs more productive, DSLs are often embedded into a host language using *staging*: The syntax of the host language is overloaded (where possible) to construct the program representation of the DSL program. A DSL compiler written in the host language then compiles the DSL program during the runtime of the host program. While this approach avoids the tedious task of building a front end for the DSL, it has significant drawbacks:

1) The programmer has to resolve the overloading to understand which part of the code belongs to the DSL program or to the host program.
2) Although the host program may build the DSL program successfully, the constructed DSL program is not *guaranteed* to be well-formed and might fail to compile. The programmer does not notice this *at compile time* but only when he runs the host program.
3) The DSL designer has to write a code generator for *every* DSL language he wants to host.

In this paper, we resurrect an old idea that interestingly has not been picked up much lately, although it solves all of the problems just mentioned: The (first) *Futamura projection* [8]. The easiest way to implement a DSL is to write an interpreter for it: For every language element, give a piece of code that implements the semantics of this language element. Essentially, the DSL implementation comes *as a library*. According to the first Futamura projection, partially evaluating the DSL interpreter with an input program *compiles* the DSL program: a process we call *refinement*. The result is basically that the interpreter is inlined into the program to be interpreted. Section II discusses this approach in detail by means of an idealized stencil code example. Note that this corresponds exactly to the output of a compiler that traverses a program representation and emits a piece of code per language element. This approach solves the above-mentioned problems in the following way:

1) There is only one program: The DSL program *is* the host program that contains explicit calls to the interpreter library.
2) Because there is only a single program, the programmer is notified *at compile time* if the program is well-formed or not.
3) The partial evaluator is a reusable component of the (host) language and can be reused for every staged DSL.

In this paper, we present a small, staged DSL for the V-cycle, an important multigrid iteration for solving linear systems using the multigrid method [4], [19]. We use a functional and imperative programming language called *Impala* to describe the algorithm on a high level, independent of the target platform. To

```
fn apply_stencil(x: int, y: int,
                 field: Field, stencil: Stencil,
                 border: fn(int, int, int) -> int
                ) -> float {
  let mut tmp = 0.0f;
  for i, j in stencil.each() {
    let xx = border(x+i, 0, field.cols-1);
    let yy = border(y+j, 0, field.rows-1);
    tmp += field(xx, yy) * stencil(i, j);
  }
  tmp
}
```

Listing 1: A generic function that applies a stencil (with respect to boundary handling) to an input field at a given position.

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

let stencil: Stencil = { /* ... */ };
let mut out: Field   = { /* ... */ };

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

Listing 2: Applying a given stencil to a field using clamp for boundary handling.

this end, we derive suitable abstractions for the important and performance-critical operations of the algorithm and factor them out using higher-order functions. They essentially constitute the language elements of our DSL and are implemented by an expert for every target architecture. By partially evaluating the DSL program with an implementation of the language elements for a certain architecture, we obtain a program that looks like it was specifically written for that particular target architecture.

In our experiments, we show that, using this approach, we can map the V-cycle algorithm to different hardware architectures: CPUs and GPUs. In the architecture mapping, we perform important optimizations like vectorization and cache-aware iteration reordering without *tainting* the other parts of the implementation with hardware-dependent, low-level details. In our previous work [13], we have demonstrated that our technique can compete with manually-tuned expert implementations of simple stencil codes while being orders of magnitudes smaller in terms of code size. In this paper, we demonstrate that our approach scales well to more complex stencil codes by considering not only single stencils for optimization, but sequences of operations.

II. Embedding of DSLs

This section describes the two main techniques to embed a DSL in Impala by means of a small stencil code example:

1) Higher-order functions allow to design generic domain-specific APIs that can be instantiated for every target architecture by highly-optimized implementations.
2) Partial evaluation entirely removes the overhead of these generic implementations by specializing the higher-order functions to their arguments.

A. Higher-Order Functions

In stencil algorithms it is often necessary to access neighboring elements which lie outside the actual field. Many frameworks provide hard-coded solutions like setting these virtual elements to zero, mirror the field, or clamp to border values. It is good programming practice to not hard-code boundary handling into the iterator but to separate this concern from other concerns of the implementation. This separation of concerns can be elegantly implemented with higher-order functions: In Listing 1 the `apply_stencil` function expects a `stencil` which it wants to apply on a field `field`. The boundary handling logic is passed as function via `border`. For example, in order to clamp the border to the input field, the programmer defines an appropriate `clamp` function and passes it to `apply_stencil` as shown in Listing 2. Of course, a naïve implementation of the higher-order functions by the compiler inflicts the performance penalty of closure allocation, function call, and so on.

B. Partial Evaluation

In Impala the programmer can annotate a call with `@` to *partially evaluate* that function call at compile time. This will have the effect that the function is specialized to all constant parameters *as far as possible*. By annotating the call to `apply_stencil` with `@`, the compiler generates a specialization where the definition of `clamp` is propagated into the body of `apply_stencil`. In this way, each call to `border` is specialized to a call to `clamp`.

```
out(x, y) = @apply_stencil(x, y, field, stencil,
                           clamp);
```

It is important to note that `@` does not change the semantics of the program. If all `@`s are elided, the program still computes the same result. Equally important, `@` is not an annotation that the compiler can ignore at will. Every `@`-annotated call activates one run of Impala's partial evaluator. Once activated, the partial evaluator keeps partially evaluating as long it makes progress. The rules of Impala's partial evaluator are formally defined (however, this is outside the scope of this paper) and are proven to preserve the termination of the program as long as partial evaluation does not create new `@`-annotated calls. A special `$` annotation can exclude code parts from being partially evaluated, if that is desired (for example some loops with constant trip count as they appear in loop tiling).

Partial evaluation also helps to specialize `apply_stencil` to the stencil itself. If the stencil is known at compile time (which is often the case), the programmer certainly wants to unroll the inner loop and replace the accesses to `stencil` with concrete values. The partial evaluator transitively runs into the `apply_stencil` function, unrolls the `stencil.each()` function and propagates the stencil constants. Furthermore, as calls to `border` have already been replaced with calls

to `clamp`, the partial evaluator will specialize both calls to `clamp` in the inner loop.

The example from Listing 1 can be optimized even further: It is also possible to create specialized variants for different region of the field with tailored boundary handling for each region [13]. For example, the vast main area of the field does not need any boundary handling at all. This saves unneeded boundary handling checks.

The function `apply_stencil` is essentially an "interpreter" that applies a stencil to a field at a given position. The aspects of boundary handling, application of the stencil, and the stencil itself are cleanly separated. Partial evaluation eliminates the overhead of this separation of concerns and produces high-performance code that looks like if all aspects had been hand-coded into a single piece of code.

III. Code Refinement

Listing 1 is a straightforward CPU implementation. In this section we show how to map implementations to different acceleration devices.

A. Code Generation for Accelerators

Impala offers built-in, compiler-known, higher-order functions to trigger code generation for GPUs as well as vectorization for CPUs with SIMD instruction sets. For example, the following function behaves like a loop ranging from `0` to `size` while invoking `body` in each iteration:

```
fn vectorize(size: int, length: int,
             body: fn() -> ()
            ) -> ();
```

However, the body is actually vectorized with SIMD width `length` [11]. Likewise, the following function triggers code generation for NVVM[1], an Intermediate Representation (IR) for NVIDIA GPUs, using the given blocking for the given grid:

```
fn nvvm(grid:  (int, int, int),
        block: (int, int, int),
        body:  fn() -> ()
       ) -> ();
```

Impala lambda-lifts the body [9] and resolves all dependencies to surrounding data. Any data needed from the outside is automatically copied to the GPU. Output data is pushed back to the CPU.

In a similar fashion, Impala offers built-in functions to generate OpenCL and CUDA source code as well as SPIR[2], an IR for OpenCL. In contrast to pragma-based solutions like OpenACC or OpenMP, Impala's built-in functions are proper functions which integrate seamlessly into Impala's type system and expect Impala values. In particular, we can wrap a call to such a function within another function. This is not possible with pragmas.

[1] http://docs.nvidia.com/cuda/nvvm-ir-spec/index.html
[2] www.khronos.org/registry/spir/specs/spir_spec-1.2.pdf

B. Supporting Accelerators in the DSL

Reconsider Listing 2. We use the function

```
fn iterate(field: Field,
           body: fn(int, int) -> ()
          ) -> ();
```

to abstract over the exact iteration behavior of the field. Impala's `for`-construct is just syntactic sugar for calling a higher-order function while passing the body as lambda function.

```
iterate(field, |x, y| /* body */);
```

This allows DSL developers to overload `for`-constructs and to provide a target-specific iteration strategy, that may use `vectorize` or `nvvm`.

In order to vectorize the stencil computation with vector length 8 for AVX or AVX2, we use the aforementioned `vectorize` function to implement `iterate`:

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

Note that we have not touched `apply_stencil` itself. We can also implement other versions of `iterate` which map to other accelerators, for example, by using the `nvvm` function. Each `iterate` can act as drop-in replacement for another one.

Again, this cleanly separates the overall algorithm from the concrete implementations of its building blocks. By partial evaluation, all those parts are fused into a piece of code that looks like if it had been hand-coded explicitly for the particular architecture.

IV. Evaluation

In this section, we discuss the implementation of a DSL for the V-cycle and show first results on different target architectures. As target hardware, we consider an Intel Core i7-3770K, an NVIDIA GeForce GTX 680, and an AMD Radeon R9 290X.

A. A DSL for the V-Cycle

The basic idea of the multigrid method is to smooth the error (e. g., using an iterative method like Jacobi or Gauss-Seidel) on different resolutions of the same data. The *V-cycle* describes one possible multigrid iteration as summarized in Algorithm 1.

The restrict and interpolate methods are used to transform data between different resolutions of the multigrid. On each level, the error is smoothed (smoother) and the error is estimated (residual). This process is recursive and starts at the finest resolution.

A DSL for the V-cycle should implement the algorithm described in Algorithm 1, but give the programmer the flexibility to choose custom methods for the multigrid components.

Algorithm 1: Recursive V-cycle: $u_h^{(k+1)} = V_h\left(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2\right)$.

if *coarsest level* **then**
  solve $A^h u^h = f^h$ exactly or by many smoothing iterations
else
  $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}\left(u_h^{(k)}, A^h, f^h\right)$ {pre-smoothing}
  $r^h = f^h - A^h \bar{u}_h^{(k)}$ {compute residual}
  $r^H = R r^h$ {restrict residual}
  $e^H = V_H\left(0, A^H, r^H, \nu_1, \nu_2\right)$ {recursion}
  $e^h = P e^H$ {interpolate error}
  $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ {coarse grid correction}
  $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}\left(\tilde{u}_h^{(k)}, A^h, f^h\right)$ {post-smoothing}
end

Moreover, it should allow to specify additional properties such as the depth of the recursion, the number of smoothing steps, etc. Using Impala, we pass the multigrid components as higher-order functions to the implementation of the V-cycle. By partially evaluating the V-cycle implementation, the user-provided multigrid components are inlined and optimized with respect to the input (stencils, etc.). Listing 3 shows the implementation of a DSL for the V-cycle and its invocation.

A naïve implementation of the V-cycle invokes each multigrid component according to the schedule of Algorithm 1 using the `iterate` function introduced in Section III. By providing specialized `iterate` implementations for CPU and GPU, we map the same algorithm to different target platforms. Likewise, vectorization is triggered by using the `vectorize` function in case we use a mapping for AVX. However, hand-tuned implementations of the V-cycle might merge multiple multigrid components in order to save unnecessary reads/writes to memory. The same optimization can be achieved in Impala by custom `iterate` functions that compute multiple components. As an example, consider the computation of the residual component followed by the restrict component: Instead of computing the residual for the whole field first and then restrict the field produced by the residual, we compute the residual only for two rows and restrict the residual before the next rows are processed. This pipelined processing allows to hold the result of the restrict component in cache on the CPU and allows to merge compute kernels on the GPU when using scratchpad (local or shared) memory. On the GPU, this has the same effect as loop fusion. Listing 4 illustrates this for the CPU. The index passed to the `residual` and `restrict` component refer to the temporary field. The offset to the current row of the other fields are tracked in the `Field` object and are used when accessing field elements. Merging the two components is only valid if the operation of the multigrid components is known: in our example, the restrict component is allowed to access two rows only. Otherwise, a larger temporary array has to be allocated and pre-computed before applying `restrict`.

```
fn vcycle_dsl(input: Field, levels: int,
              vsteps: int, ssteps: int,
              smoother: fn(/* ... */) -> (),
              residual: fn(/* ... */) -> (),
              restrict: fn(/* ... */) -> (),
              interpolate: fn(/* ... */) -> ()
             ) -> Field {
  /* allocate memory for all levels */
  // Sol, RHS, Res, Tmp

  // vcycle implementation
  fn vcycle_dsl_intern(level: int) -> () {
    if level == levels-1 {
      // smooth
      for i in range(0, ssteps) {
        if i>0 { swap(Sol(level), Tmp(level)); }
        for x, y in iterate(Sol(level)) {
          solver(x, y, /* fields */);
        }
      }
    } else {
      // pre-smooth
      for i in range(0, ssteps) {
        if i>0 { swap(Sol(level), Tmp(level)); }
        for x, y in iterate(Sol(level)) {
          solver(x, y, /* fields */);
        }
      }

      for x, y in iterate(Res(level)) {
        residual(x, y, /* fields */);
      }

      for x, y in iterate(RHS(level+1)) {
        restrict(x, y, /* fields */);
      }

      // recursion
      vcycle_dsl_intern(level+1);

      for x, y in iterate(Sol(level)) {
        interpolate(x, y, /* fields */);
      }

      // post-smooth
      for i in range(0, ssteps) {
        if i>0 { swap(Sol(level), Tmp(level)); }
        for x, y in iterate(Sol(level)) {
          solver(x, y, /* fields */);
        }
      }
    }
  }

  // call the vcycle implementation
  for i in range(0, vsteps) {
    vcycle_dsl_intern(0);
  }
}

// specialize call to vcycle_dsl
let result = @vcycle_dsl(input, levels, vsteps,
                         ssteps, jacobi, residual,
                         restrict, interpolate);
```

Listing 3: Implementation of the DSL for the V-cycle.

B. Results

Our first evaluation of the V-cycle DSL presented in Section IV-A shows promising results. Table I lists the execution

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

  for y in $range_step(0, Res.rows, 2) {
    // compute the residual for two rows
    for x in $range(0, Res.cols) {
      @residual(x, 0 /* ... */ Sol, tmp, RHSF);
      @residual(x, 1 /* ... */ Sol, tmp, RHSF);
    }
    // restrict the residual of two rows
    for x in $range(0, RHSC.cols) {
      @restrict(x, 0 /* ... */ tmp, RHSC);
    }
  }
}
```

Listing 4: Merging residual and restrict computation on the CPU.

Table I: Execution times in ms for the components of the V-cycle DSL for a field of size 2048×2048 on an Intel Core i7-3770K (CPU & AVX).

Mapping	smoother	residual	restrict	interpolate
CPU	11.63	11.21	1.61	2.24
CPU:M	11.63	12.18		2.24
AVX	3.98	3.99	2.33	3.39
AVX:M	3.98	4.10		3.39

time in ms on the CPU for the components of the V-cycle on the first level. In addition to a non-vectorized CPU mapping and a vectorized mapping for AVX, we list the execution time when merging the residual and restrict components (CPU:M and AVX:M). It can be seen that we get a speedup of $3\times$ for most components. Only restrict and interpolate are slower due to their memory access pattern. The merged residual and restrict computation for AVX vectorizes only the residual component.

Table II lists the corresponding execution time in ms on the GPU using mapping strategies for NVVM and SPIR. Merging the residual and restrict components is almost twice as fast as computing both components in separate compute kernels on the GPU.

C. Discussion

The proposed DSL can be easily extended to express different multigrid iterations. It is actually sufficient to change the recursion in the V-cycle implementation in order to get the schedule for the W-cycle multigrid iteration.

The performance evaluation has shown that we can map the same high-level description to different target platforms by providing target-specific mappings. Moreover, we merge multiple components as shown exemplarily for the residual and restrict components.

Table II: Execution times in ms for the components of the V-cycle DSL for a field of size 2048×2048 on an NVIDIA GeForce GTX 680 and an AMD Radeon R9 290X.

Mapping	smoother	residual	restrict	interpolate
NVVM	0.51	0.53	0.18	0.35
NVVM:M	0.51	0.34		0.35
SPIR	0.19	0.19	0.08	0.16
SPIR:M	0.19	0.14		0.16

We believe that using our approach, we can get the same performance as target-dependent, hand-optimized implementations for multigrid solvers (e. g., [12]). For single stencil operators, we have already shown that we achieve competitive results compared to hand-tuned implementations on GPU accelerators [13]. Currently, we are working on mappings to merge computations across not only two, but several components in order to get the same performance as hand-tuned implementations.

V. Related Work

Multigrid solvers and in particular stencil codes are one of the most important algorithm classes in HPC and have been a popular research topic for decades. Still, a multitude of solutions exist and emerge to describe stencil codes at a high level and to map them to different target architectures.

Specialized libraries like DUNE [2] or hypre [1] provide a collection of solvers for large linear systems of equations on massively parallel machines. Auto-tuning frameworks like PATUS (Parallel Auto-Tuned Stencils) [10], [5] or the parallel Optimized Sparse Kernel Interface (pOSKI) [3] generate specialized codes for stencil computations on shared-memory architectures. DSLs for stencil codes provide special language constructs to describe stencil computations and to map the computation to parallel target hardware [7], [18], [14], [16].

We use language embedding and DSLs to describe multigrid solvers and stencil codes. However, we give the programmer additional opportunities to influence the optimization and transformation process from within the language.

Other DSL approaches such as Liszt [7], Pochoir [18], and HIPA^{cc} [14] focus on providing a simple and concise syntax to express algorithms. However, they offer no control over the applied optimization strategies. An advancement to this is the explicit specification of schedules in Halide [16]: Target-specific scheduling strategies can be defined by the programmer. Still it is not possible to trigger code refinement explicitly.

An alternative to these approaches is explicit code refinement. It can be achieved through staging like in Terra [6] and in *Spiral in Scala* [15]. Terra is an extension to Lua. Program parts in Lua can be evaluated and used to build Terra code during the run of the Lua program. However, this technique makes use of two different languages and type safety of the constructed fragments can only be checked before executing the constructed Terra code. Spiral in Scala uses the concept

of lightweight modular staging [17] to annotate types in Scala. Computations which make use of these types, are automatically subject to code refinement.

VI. Conclusion

In this paper, we present an approach for DSL creation via embedding into our host language Impala. It features high-level abstractions through higher-order functions and explicit control over code specialization through partial evaluation. These features allow to realize a novel refinement concept: We are able to abstract from target-machine details and to specialize code for different platforms.

As an example, we realize a DSL for multigrid codes. We are able to generate code for CPUs (including vectorization) and GPUs by only switching the mapping for the desired target architecture. This process does not involve an adaption of the implemented algorithm, since the realization is cleanly separated from the surrounding building blocks. Code refinement can then be applied to these different building blocks in order to glue them together into a single piece of code. The resulting code does not contain any abstractions nor unspecialized code for the target architecture. Starting from the same high-level description, we achieve a speedup of up to $3\times$ on the CPU through vectorizing and a speedup of up to $2\times$ on the GPU through kernel fusion.

VII. Acknowledgments

This work is partly supported by the Federal Ministry of Education and Research (BMBF), as part of the Collaborate3D and ECOUSS projects as well as by the Intel Visual Computing Institute Saarbrücken. It is also co-funded by the European Union (EU), as part of the Dreamspace project.

References

[1] A. Baker, R. Falgout, T. Kolev, and U. M. Yang. Scaling Hypre's Multigrid Solvers to 100,000 Cores. *High-Performance Scientific Computing*, pages 261–279, 2012.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. *Computing*, 82(2):103–119, July 2008.

[3] Berkeley Benchmarking and Optimization (BeBOP) Group, University of California, Berkeley. *pOSKI: Parallel Optimized Sparse Kernel Interface Library*, Apr. 2012.

[4] W. L. Briggs, H. Van Emden, and S. F. McCormick. *A Multigrid Tutorial*, volume 2. Society for Industrial And Applied Mathematics (SIAM), June 2000.

[5] M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 676–687. IEEE, May 2011.

[6] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A Multi-Stage Language for High-Performance Computing. In *Proceedings of the 34th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 105–116. ACM, June 2013.

[7] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 9:1–9:12. ACM, Nov. 2011.

[8] Y. Futamura. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 1999. Reproduction of the 1971 paper.

[9] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Functional Programming Languages and Computer Architecture*, 1985.

[10] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–12. IEEE, Apr. 2010.

[11] R. Karrenberg and S. Hack. Whole-Function Vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 141–150. IEEE, Apr. 2011.

[12] H. Köstler, M. Stürmer, and T. Pohl. Performance Engineering to Achieve Real-time High Dynamic Range Imaging. *Real-Time Image Processing*, pages 1–13, Jan. 2013.

[13] M. Köster, R. Leißa, S. Hack, R. Membarth, and P. Slusallek. Code Refinement of Stencil Codes. *Parallel Processing Letters (PPL)*, 24(3):1–16, Sept. 2014.

[14] R. Membarth, F. Hannig, J. Teich, and H. Köstler. Towards Domain-specific Computing for Stencil Codes in HPC. In *Proceedings of the 2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 1133–1138. IEEE, Nov. 2012.

[15] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *Proceedings of the 12th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 125–134. ACM, Oct. 2013.

[16] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Transactions on Graphics (TOG)*, 31(4):32:1–32:12, July 2012.

[17] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 127–136. ACM, Oct. 2010.

[18] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir Stencil Compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 117–128. ACM, June 2011.

[19] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, Dec. 2000.