Platform-Specific Optimization and Mapping of Stencil Codes through Refinement

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

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

ABSTRACT

A straightforward implementation of an algorithm in a general-purpose programming language does usually not deliver peak performance: compilers often fail to automatically tune the code for certain hardware peculiarities like memory hierarchy or vector execution units. Manually tuning the code is firstly error-prone as well as time-consuming and secondly taints the code by exposing those peculiarities to the implementation. A popular method to circumvent these problems is to implement the algorithm in a Domain-Specific Language (DSL). A DSL compiler can then automatically tune the code for the target platform.

In this paper we show how to embed a DSL for stencil codes in another language. In contrast to prior approaches we only use a single language for this task. Furthermore, we offer explicit control over code refinement in the language itself which is used to specialize stencils for particular scenarios. Our first results show that our specialized programs achieve competitive performance compared to hand-tuned CUDA programs.

1. INTRODUCTION

Many scientific codes, including stencil codes, require careful tuning to run efficiently on modern computing systems. Specific hardware features like vector execution units require architecture-aware transformations. Moreover, special-case variants of codes often boost the performance. Usually, compilers for general-purpose languages fail to perform the desired transformations automatically for various reasons: First, many transformations are not compatible with the semantics of languages like C++ or Fortran. Second, the hardware models the compiler uses to steer its optimizations, are far too simple. Lastly, the static analysis problems that arise when justifying such transformations are too hard in general.

Therefore, programmers often optimize their code manually, use meta programming techniques to automate the manual tuning or create a compiler for a DSL. A prominent example for meta programming is the C++ template language that is evaluated at compile time and produces a program in the C++ core language. A DSL compiler has the advantage of custom code transformations and code generation. However, implementing a compiler is a cumbersome, time-consuming endeavor. Hence, many programmers embed a DSL in a host language via staging. In DSL staging, a program in a *host language* A is used to construct another program in another language B. A compiler written in A then compiles the B program.

Both approaches have significant limitations concerning the productivity of the programmer: Very often, languages with meta programming capabilities and DSL staging frameworks involve more than one language. One language is usually evaluated at compile time while the other is evaluated during the actual runtime of the program. This requires the programmer to decide which part of the program runs in which stage before he starts implementing. For example, compare the implementation of a simple function, say the factorial, in the C++ template language and the core language. Another significant disadvantage of the two-languages approach is that the type systems of the two languages need to cooperate which is often only rudimentarily supported or not the case at all (C++'s template language is dynamically typed).

Many of the code transformations that are relevant for high-performance codes, and also for stencil codes, can however be expressed by partial evaluation of code written in one single language. Take for example the handling of the boundary condition of a stencil operation, an example we will go through in more detail later. Using a simple conditional the program can test if the stencil overlaps with the border of the field and execute specialized code that handles this situation. However, evaluating this conditional during

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

runtime imposes a significant performance overhead. Partially evaluating the program at compile time can specialize the program in way that a particular case of boundary handling is applied to the corresponding region of the field which eliminates unnecessary checks at runtime.

In this paper we investigate the implementation of several stencil codes via DSL embedding in our research-prototype language called *Impala* (Section 2). Impala is an imperative and functional language that borrows heavily from Rust[1]. It extends Rust by a partial evaluator that the programmer can access through Impala's syntax (Section 3). Partial evaluation in Impala is realized in a way that erasing the partial evaluation operators from the program does not change the semantics of the program. This is often not possible in existing languages supporting meta programming. Finally, Impala provides Graphics Processing Unit (GPU) code generation for a subset of programs. In this paper we show that the partially evaluated stencils written in Impala reach a comparable performance to hand-tuned CUDA code (Section 5).

2. STENCIL CODES IN IMPALA

In this section we present our DSL approach for the realization of *stencil codes* in Impala. Consider an implementation which applies a 1D stencil to a data array in Impala written in C-like imperative style:

```
for (let mut i = 0; i<size; ++i) {
  out[i] = 0.25f * arr[i-1] +
           0.50f * arr[i  ] +
           0.25f * arr[i+1];
}
```

The loop iterates over the data array and applies a fixed stencil to each element in the array. The stencil computation is hard-coded in the example for a given kernel. However, this coding style has two problems: First, the stencil is hard-coded and is not generic which means that the code has to be rewritten for a different stencil. Furthermore, an extension to 2D or 3D makes the code even harder to maintain and to understand. Second, the logic iterating over the data array and the computation are tightly coupled which makes it harder to adapt it to different hardware architectures.

To tackle this dilemma, Impala supports code specialization and decoupling of algorithms from schedules. Specialization allows to generate the same optimized code as shown above from a generic stencil function, like `apply_stencil`:

```
fn apply_stencil(arr: [float], stencil: [float],
                 i: int) -> float {
  let mut sum = 0.0f;
  let offset = stencil.size / 2;

  for j in indices(stencil) {
    sum += arr[i + j - offset] * stencil[j];
  }

  return sum;
}
```

The specialization of this function is triggered at a call site which is shown in Section 3.

The desired decoupling of the algorithm and the concrete schedule can be realized by making use of higher-order functions. A custom iteration function `field_indices`, which takes another function (the kernel body) as an argument, can be used for this task in our scenario. The `field_indices` function applies the given body (in form of a lambda function) to all indices of the elements in a passed array. Similar to Rust, Impala offers the possibility to call this function with the syntax of a `for` construct which passes the body of the for loop as function to the `field_indices` functionality:

```
let stencil = [ ... ];

for i in field_indices(arr) {
  out[i] = apply_stencil(arr, stencil, i);
}
```

In our approach the iteration function can be provided in form of a library. That is, the stencil code remains unchanged while the iteration logic can be exchanged by just linking a different target library or just calling a specific library function. The required hardware-specific and cache-aware implementations can then be written separately.

3. CODE REFINEMENT

In this section we describe our refinement approach of algorithms. One of the main reasons for refinement in our setting is to improve performance at run time. An improvement can be achieved by partially evaluating the program at compile time. Especially, a platform-specific mapping of the stencils can be realized with this approach.

3.1 Partial Evaluation

Partial evaluation is a concept for the simplification of program parts which is typically performed at compile time by specialization of compile-time known values. Compilers perform partial evaluation during transformation phases by applying techniques such as constant propagation, loop unrolling, loop peeling, or inlining. However, this is completely transparent to the programmer. That is, programmers cannot control which parts of a program should be partially evaluated with which values. Furthermore, a compiler will usually only apply a transformation, if the compiler can guarantee the termination of the transformation. For this reason, Impala delegates the termination problem to the programmer. He can explicitly trigger partial evaluation by annotating code with `@`. If the annotated code diverges, the compiler will also diverge. On the other hand, partial evaluation goes beyond classic compiler optimizations or unroll-pragmas because the compiler really executes the annotated part of the program. Moreover, the programmer can explicitly forbid partial evaluation via `#`.

In the following example, we trigger partial evaluation of the *apply_stencil* function introduced in the previous section by annotating the call-site of the function:

```
for i in field_indices(arr) {
  out[i] = @apply_stencil(arr, stencil, i);
}
```

During specialization of *apply_stencil*, the compiler tries to evaluate expressions and constructs that are known to be constant and replaces them by the corresponding results of the evaluation.

In our example, the for loop which iterates over the stencil is unrolled and the constants from the stencil are loaded and inserted into the code for each iteration. In order to apply this specialization, the size of the stencil has to be constant. Such an array is called *definite* in Impala and is immutable:

```
let stencil = [ 0.25f, 0.50f, 0.25f ];
```

[1] http://www.rust-lang.org

The arr field, however, is *indefinite* which means that the values of the array are not known at compile time. Hence, accesses to arr remain in the code, but the indices for the accesses are updated: j and offset are replaced by constants in the index computation. The result of partially evaluating *apply_stencil* is the same like hard-coding the stencil computation, as shown before.

3.2 Platform-specific Mapping

In the previous example, we ignored the fact that the arr field is accessed out of bounds at the left and right border when the stencil is applied. One possibility to handle out of bounds memory accesses is to apply boundary handling whenever the field is accessed: For instance, the index can be *clamped* to the last valid entry at the extremes of the field. Therefore, we use two functions: one for the left border (bh_clamp_lower) and one for the right border (bh_clamp_upper):

```
fn bh_clamp_lower(idx: int, lower: int) -> int {
  if (idx < lower) idx = lower;
  return idx;
}

fn bh_clamp_upper(idx: int, upper: int) -> int {
  if (idx >= upper) idx = upper-1;
  return idx;
}

...
for j in indices(stencil) {
  // clamp the index for arr
  let mut idx = i + j - offset;
  idx = bh_clamp_lower(idx, 0);
  idx = bh_clamp_upper(idx, arr.size);
  sum += arr[idx] * stencil[j];
}
```

These checks ensure that the field is not accessed out of range, but at the same time they are applied for each element of the field whether required or not. Applying the check for each memory access comes at the cost of performance when executed on platforms such as GPU accelerators. If we can specialize the code in a way that checks are only executed at the left and right border, there will be no noticeable performance loss. This could be achieved by manually peeling off stencil.size / 2 iterations of the loop iterating over the field and applying boundary handling only for those iterations. However, doing this results in an implementation that cannot be used for different stencils and different scenarios.

Specializing the apply_stencil implementation to consider boundary handling of different field regions, allows us to write reusable code. Hence, we create a function apply_stencil_bh that applies a stencil to a field. It takes two additional functions for boundary handling as arguments. To specialize on the different field regions, we create a loop that iterates over these regions. Applying boundary handling is delegated to the access function that applies boundary handling for the left border only in case of the left field region and for the right border only in case of the right field region:

```
fn access(arr: [float], region: int, i: int, j: int,
          bh_lower: fn(int, int) -> int,
          bh_upper: fn(int, int) -> int
         ) -> float {
    let mut idx = i + j;
    if (region==0) idx = bh_lower(idx, 0);
    if (region==2) idx = bh_upper(idx, arr.size);
    return arr[idx];
}
```

In order to specialize the apply_stencil_bh function, the range for the region and the access function need to be annotated. In addition, we want the stencil computation to be specialized, and thus, also annotate the stencil iteration:

```
fn apply_stencil_bh(arr: [float], stencil: [float],
         bh_lower: fn(int, int) -> int,
         bh_upper: fn(int, int) -> int
         ) -> float {
  let offset = stencil.size / 2;
  // lower bound of regions
  let L = [0, offset, arr.size - offset];
  // upper bound of regions
  let U = [offset, arr.size - offset, arr.size];

  // iterate over field regions
  for region in @range(0,3)
    // iterate over a single field region
    for i in range(L(region), U(region)) {
      let mut sum = 0;
      for j in @indices(stencil)
        // access function applies boundary handling
              depending on the region
        sum += @access(arr, region, i, j+offset,
             bh_lower, bh_upper) * stencil[j];
      arr[i] = sum;
    }
}
```

The apply_stencil_bh function for a single kernel defines an interpreter for stencils which can be specialized through partial evaluation. The synthesized code then performs the actual computation of a specific stencil while the imposed overhead by the interpreter is completely removed according to the first Futamura projection [6, 7].

Consider now the case of partially evaluating the presented interpreter. This yields three distinct loops that iterate over the corresponding regions of the input field. Each loop now only contains region-specific boundary handling checks. Mapping the stencil computation to the GPU results in three distinct compute kernels that operate on the different field regions.

The following code listing shows an application of the previously introduced apply_stencil_bh function and applies it to a specific stencil and boundary handling methods:

```
let stencil = [ 0.25f, 0.50f, 0.25f ];
@apply_stencil_bh(arr, stencil,
                  bh_clamp_lower,
                  bh_clamp_upper)
```

As previously described, this will result in a specialized version of apply_stencil_bh for this scenario:

```
// iterate over the left field region
for i in range(0, 1) {
  let mut sum = 0;
  sum += arr[bh_clamp_lower(i - 1, 0)] * 0.25f;
  sum += arr[bh_clamp_lower(i,     0)] * 0.50f;
  sum += arr[bh_clamp_lower(i + 1, 0)] * 0.25f;
  arr[i] = sum;
}
... // iterate over the center field region
```

Further specialization of the left field region can be used to eliminate the loop iteration and to specialize the boundary-handling calls to bh_clamp_lower. This would evaluate the if-condition of the boundary checks and the following code would emerge:

```
// iterate over the left field region
let mut sum = 0.0f;
sum += arr[0] * 0.25f;
sum += arr[0] * 0.50f;
sum += arr[1] * 0.25f;
arr[0] = sum;
... // iterate over the center field region
```

While we have shown the refinement approach for 1D examples only, the concept can be applied to the multi-dimensional case by introducing a generic `index` type. This type encapsulates the index handling for an arbitrary number of dimensions. Consequently, further changes in the code can be minimized which may typically be required during an adaption to another number of dimensions.

4. APPLICATIONS

In this section we present two example applications, one from the field of image processing and one from the field of scientific computing. We discuss how specialization triggers important optimizations opportunities for the compiler.

Consider a bilateral filter [14] from the field of image processing. This filter smoothes images while preserving the sharp edges of an image. Algorithm 1 shows the pseudo code for the parallel computation of the bilateral filter.

The computation of the filter mainly consists of two components: closeness and similarity. Closeness depends on the distance between pixels and can be precomputed. Similarity depends on the difference of the pixel values and is evaluated on the fly.

Listing 1 shows an implementation in Impala. The precomputed closeness function is stored in a `mask` array. The two inner loops which iterate over the range of the kernel are annotated, to enforce partial evaluation for a given `sigma_d`. This will propagate the constant mask into the computation and will specialize the index calculation. Another possibility in this context would be a mapping of the `mask` to constant memory, in the case of a GPU.

The use of a Jacobi iteration to solve the heat equation can be specialized similarly (Listing 2). We can use the presented `apply_stencil` function to apply the stencil for Jacobi in each step of the iteration. Partial evaluation of this call site propagates the Jacobi stencil into the function. This causes the calculations that would normally be multiplied with zero at run time, to be evaluated to zero at compile time. Hence, these computations will not be performed during execution of the stencil later on.

5. EVALUATION

In this section we outline our compiler framework and show first results on the CPU and GPU.

5.1 Compiler Framework

Our compiler provides back ends for CPUs and CUDA-capable graphics cards from NVIDIA. Programs written in Impala are parsed into an higher-order intermediate representation (IR) that captures functional properties. All transformations and code refinements described in Section 3 are applied on this level of the intermediate representation.

Stencils are not limited to a particular output field but can perform write accesses to different output targets which simplifies the development of stencil codes. Moreover, we allow random read and write accesses to fields as well as different resolutions of the input data.

When generating code for the GPU, for instance, memory allocations on the target device are fully managed by the generated code of our compiler. Required data transfer from the host device to the GPU (and vice versa) is also performed automatically.

For target code generation, we use LLVM [9]. When our

Algorithm 1: Parallel bilateral filter algorithm.

```
1  foreach pixel pix in image arr do in parallel
2  |  x ← get_index_x(arr, pix)
3  |  y ← get_index_y(arr, pix)
4  |  p, k ← 0
5  |  for yf = −2 · sigma_d to 2 · sigma_d do
6  |  |  for xf = −2 · sigma_d to 2 · sigma_d do
7  |  |  |  c ← closeness((x, y), (x + xf, y + yf))
8  |  |  |  s ← similarity(arr[x, y], arr[x + xf, y + yf])
9  |  |  |  k ← k + c · s
10 |  |  |  p ← p + c · s · arr[x + xf, y + yf]
11 |  |  end
12 |  end
13 |  out[x, y] ← p/k
14 end
```

```
fn main() {
  let width   = 1024;
  let height  = 1024;
  let sigma_d = 3;
  let sigma_r = 5.0f;
  let mut arr = ~array::new(width, height);
  let mut out = ~array::new(width, height);

  let mask = @precompute(...);

  for i: index in field_indices(out) {
    let c_r = 1.0f/(2.0f*sigma_r*sigma_r);
    let mut k = 0.0f;
    let mut p = 0.0f;

    for yf in @range(-2*sigma_d, 2*sigma_d+1) {
      for xf in @range(-2*sigma_d, 2*sigma_d+1) {
        let diff = arr[i + index(xf, yf)] - arr[i];
        let s = exp(-c_r * diff*diff) *
                mask[xf + sigma_d][yf + sigma_d];
        k += s;
        p += s * arr[i + index(xf, yf)];
      }
    }

    out[i] = p/k;
  }
}
```

Listing 1: Bilateral filter description in Impala.

```
fn apply_stencil(arr: [float], stencil: [float],
                 i: index) -> float {
  let mut sum = 0.0f;
  let offset  = stencil.size / 2;
  for j in indices(stencil) {
    sum += arr[i + j - offset] * stencil[j];
  }
  return sum;
}

fn main () {
  let mut arr = ~array::new(width, height);
  let mut out = ~array::new(width, height);
  let a = 0.2f;
  let b = 1.0f - 4.0f * a;
  // stencil for Jacobi
  let stencil = [[0.0f, b, 0.0f],
                 [   b, a,    b],
                 [0.0f, b, 0.0f]];
  while (/* not converged */) {
    for i in field_indices(out) {
      out[i] = @apply_stencil(arr, stencil, i);
    }

    swap(arr, out);
  }
}
```

Listing 2: Jacobi iteration in Impala.

Table 1: Execution times in *ms* for the Jacobi kernel in Impala on an Intel Core i7 3770 for a field of size 2048×2048.

	Scalar	Vectorized (SSE)
Impala (generic)	13.23	4.39
Impala (specialized)	5.41	2.51

IR is converted to LLVM IR, target-specific mapping is also performed: In particular for GPU execution, index variables are mapped to special hardware registers and the compute kernels are annotated as *kernel*. The resulting IR conforms to the NVVM IR specification[2] and can be compiled to PTX assembly using the CUDA compiler SDK[3]. Code for the CPU can be automatically vectorized with the help of our data-parallel vectorizer [8].

Using the same mapping with similar annotations, LLVM IR can be generated that conforms to Standard Portable Intermediate Representation (SPIR)[4]. SPIR is supported in OpenCL 1.2 via extensions and will allow us to support GPU accelerators from other vendors in the future.

5.2 Performance Estimation

We evaluate the performance of the generated code on two GPU architectures from NVIDIA, using the GTX 580 and GTX 680 GPUs and on an Intel Core i7 3770 CPU. In order to estimate the quality of the generated code we consider one iteration of the Jacobi iteration from Listing 2.

CPU Evaluation

Table 1 shows the evaluation of the Jacobi kernel on the CPU on a single core. The generic version contains an inner loop which iterates over the elements of a given stencil array. Each value of the stencil is loaded from memory for each access (even for elements which are zero). The specialized version is the generated one from Impala after partially evaluating the program. It does not contain an inner loop and no further memory accesses for loading of stencil elements are required. With the help of our data-parallel vectorizer we are able to vectorize the main loops of both programs.

The scalar generic version takes about 2.5× longer than the scalar specialized version. However, making use of vectorization reduces the execution times of both versions significantly. For the specialized version, vectorization improves performance by a factor of 2.1.

GPU Evaluation

Since it is known that stencil codes are usually bandwidth limited, we list the theoretical peak and the achievable memory bandwidth of the GPUs in Table 2. For the Jacobi kernel, we have to load one value from main memory (from `arr`) and to store one value (`out`), if we assume that all neighboring memory accesses for the stencil are in cache. This means for single precision accuracy we have to transfer $4 \cdot 2 = 8$ bytes per element. On the GTX 680 with achievable memory bandwidth of $b = 147.6$ GB/s and for a problem size $N = 2048 \times 2048$ we thus estimate $\frac{N \cdot 8}{b} \cdot 1000 \approx 0.23$ ms for the kernel. This matches quite well to the measured runtime of our hand-tuned CUDA implementation (0.23 ms) for the Jacobi kernel as seen in Table 3.

[2] docs.nvidia.com/cuda/nvvm-ir-spec/index.html
[3] developer.nvidia.com/cuda-llvm-compiler
[4] www.khronos.org/registry/cl/specs/spir_spec-1.2-provisional.pdf

Table 2: Theoretical **peak** and the achievable (**memcpy**) memory bandwidth in GB/s for the GTX 580 and GTX 680 GPUs.

	GTX 580	GTX 680
Peak	192.4 GB/s	192.2 GB/s
Memcpy	161.5 GB/s	147.6 GB/s
Percentage	83.9 %	76.8 %

Table 3: Execution times in *ms* for the Jacobi kernel on the GTX 580 and GTX 680 for a field of size 2048×2048.

	GTX 580	GTX 680
CUDA (hand-specialized)	0.33	0.35
CUDA (hand-tuned)	0.26	0.23
Impala (specialized)	0.32	0.35

The table shows results for three different variants of the Jacobi kernel:

- a hand-specialized implementation in CUDA where the stencil is hard-coded,
- a hand-tuned refinement of the first implementation with device-specific optimization (custom kernel tiling, unrolling of the iteration space, and usage of texturing hardware), and
- our generated implementation from Impala after specialization.

It can be seen that our generated version from a generic description is as fast as the corresponding hand-specialized implementation in CUDA. The additional performance gains of the hand-tuned implementations stem from unrolling of the global iteration space and usage of texturing memory. In particular, the use of texturing memory is device-specific and only the GTX 680 benefits from this. While we do not support texturing hardware and unrolling of the global iteration space yet, we believe that we will see the same performance improvement once supporting these features.

6. RELATED WORK

Stencil codes are an important algorithm class and consequently a considerably high effort has been spent in the past on tuning stencil codes to target architectures. To simplify this process, specialized libraries [1, 2], auto tuners [3], and DSLs [5, 13, 11] were developed.

We follow the direction of DSLs in our work, but we give the programmer additional opportunities to control the optimization process. Previous DSL approaches such as Liszt [5] and Pochoir [13] 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 [11]: target-specific scheduling strategies can be defined by the programmer. Still it is not possible to trigger code refinement explicitly. Explicit code refinement can be achieved through staging like in Terra [4] and in *Spiral in Scala* [10]. 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 [12] to annotate types in Scala. Computations which make use of these types, are automatically subject to code refinement. Transformations on these computations, however, are performed implicitly, and thus, the programmer has no further control over the applied transformations.

7. CONCLUSION

In this paper, we have presented a domain-specific language for stencil codes through language embedding in Impala. Unique to our approach is our code refinement concept. By partially evaluating fragments of the input program it is possible to achieve platform-specific optimizations. Compared to traditional compilers, code refinement is triggered explicitly by the programmer through annotations. This allows to achieve traditional optimizations such as constant propagation and loop unrolling. Moreover, we outlined how our concept can be used for domain-specific mapping of stencil codes. As an application of domain-specific mapping, we have shown that we are able to generate specialized code variants for different field regions.

Currently, our research compiler is able to generate target code for execution on GPU accelerators from NVIDIA as well as CPUs. First results show that we can generate GPU code that is as fast as manual implementations for simple stencil codes. Our next steps include evaluation of more complex stencils such as bilateral filtering and 3D stencils. Furthermore, we are currently working on the integration of device-specific optimizations (e. g., support for texturing hardware).

8. ACKNOWLEDGMENTS

This work is partly supported by the Federal Ministry of Education and Research (BMBF), as part of the ECOUSS project as well as by the Intel Visual Computing Institute Saarbrücken.

9. 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] 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.

[4] 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.

[5] 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.

[6] Y. Futamura. Partial evaluation of computation process, revisited. *Higher Order Symbol. Comput.*, 1999.

[7] N. D. Jones. Mix ten years later. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. ACM, 1995.

[8] 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.

[9] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, Mar. 2004.

[10] 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.

[11] 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, July 2012.

[12] 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.

[13] 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.

[14] C. Tomasi and R. Manduchi. Bilateral Filtering for Gray and Color Images. pages 839–846. IEEE, Jan. 1998.