Preprint of an article published in
Parallel Processing Letters, Vol. 24, No. 3 (2014) 1441003 (16 pages)
DOI: 10.1142/S0129626414410035

http://www.worldscientific.com/worldscinet/ppl

CODE REFINEMENT OF STENCIL CODES

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

Received April 2014
Revised July 2014
Communicated by Guest Editors
Published 30 September 2014

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 avoid 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 article 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 which offers explicit control over code refinement. This is used to specialize stencils for particular scenarios. Our results show that our specialized programs achieve competitive performance compared to hand-tuned CUDA programs while maintaining a convenient coding experience.

Keywords: Stencil Codes; Partial Evaluation; Domain-Specific Language

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 often too hard.

Therefore, programmers often optimize their code manually, use meta programming techniques to automate 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 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, for instance, is dynamically typed: C++ type checks the program *after* template instantiation.

Many code transformations that are relevant for high-performance codes and stencil codes in particular 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 runtime imposes a significant performance overhead. Partially evaluating the program at compile time can specialize the program in a 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 article we investigate the implementation and the evaluation of several stencil codes via DSL embedding in our research language *Impala* [1] (Section 2)—a dialect of Rust[a]. Impala supports imperative as well as functional idioms and extends Rust by a partial evaluator that the programmer can control via annotations (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 which support meta

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

programming. Finally, Impala provides Graphics Processing Unit (GPU) code generation capabilities for arbitrary parts of the input program. We show that partially evaluated stencils written in Impala reach a competitive performance to hand-tuned CUDA code on different stencils and different platforms (Section 5).

2. Stencil Codes in Impala

In this section we present our approach for the realization of *stencil codes* in Impala. Consider the following C program that applies an 1D stencil to an array:

```
for (int 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 array and applies a fixed stencil to each element in the array. The stencil computation is specialized in the example for a given kernel. However, this coding style has two problems: First, the stencil is hard-coded. 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 array and the computation are tightly coupled which makes it harder to adapt it to different hardware architectures which might require specific transformations in order to use the underlying hardware efficiently. This, in turn, means that the whole computation needs to be rewritten for different platforms.

To tackle this dilemma, Impala supports code specialization and decoupling of algorithms from its iteration logic. Specialization of the following generic stencil function synthesizes code akin to the optimized code shown above:

```
fn apply_stencil(arr: &[float], stencil: [float], i: int) -> float {
  let mut sum = 0.0f; // sum is changed afterwards so declare it mutable
  let offset = stencil.size / 2;

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

  sum // return the sum
}
```

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

The desired decoupling of the algorithm from its iteration logic is realized by higher-order functions. A custom iteration function `field_indices`, applies the kernel body passed as function to all indices of the elements in a passed array. For example, the following code applies `stencil` to all indices of `arr`:

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

field_indices(arr, |i| {
  out(i) = apply_stencil(arr, stencil, i);
});
```

Like in Ruby or Rust bars indicate a lambda function. Alternatively, Impala offers the possibility to call this function with the syntax of a `for` construct which passes the body of the `for` loop as last argument. Thus, the following syntax is semantically equivalent:

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

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

Note that the parameter `i` of the former lambda function now becomes the iteration variable of our `for` construct. This mechanism allows to overload `for` constructs which is particularly attractive for DSL developers. DSL programmers can then call `field_indices` in a convenient way.

In our approach the iteration function can be provided in form of a library. 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 runtime. An improvement can be achieved by partially evaluating the program at compile time. Especially, a platform-specific mapping of stencil codes can be realized with this approach.

3.1. *Partial Evaluation*

Partial evaluation is a concept for evaluating parts of the program at compile time. Compilers perform partial evaluation during transformation phases by applying techniques such as constant propagation, loop unrolling, loop peeling, or inlining. However, this is completely opaque 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 `@expr`. If the annotated code diverges, the compiler will also diverge. However, this is not a problem in practice as the divergence would otherwise occur at runtime. On the other hand, partial evaluation goes far beyond classic compiler optimizations or unroll-pragmas because the compiler really executes the annotated part of the program.

Partially evaluating the input program can lead to larger code compared to the input program which may cause performance problems later on. This can be caused by unrolling of loops, for instance. In such cases the programmer can explicitly prohibit partial evaluation via a special annotation. This ensures that certain parts of the program remain untouched while the surrounding parts are partially evaluated. However, in our experience programmers want to fully specialize a particular function without forbidding specialization of a particular part of that function in most

cases (see also Section 3.3).

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 evaluates 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. The elements of the `arr` field, however, are unknown at compile time. Hence, accesses to `arr` remain in the code, but the indices for the accesses are updated and adjusted due to partial evaluation: The variables `j` and `offset` are replaced by constants in the index computation.

3.2. *Platform-Specific Mapping*

Impala provides built-in functions to trigger code generation for different target types. For example, consider the implementation of the aforementioned `field_indices` function for CUDA-capable GPUs:

```
fn field_indices(arr: &[float], body: fn(int) -> ()) -> () {
  let dim = (arr.size, 1, 1);    // setup dimension
  let block = (128, 1, 1);       // default blocking in this case
  nvvm(dim, block, || {
    let index = nvvm_read_tid_x() + nvvm_read_ntid_x() * nvvm_read_ctaid_x();
    body(index);
  });
}
```

The compiler-known `nvvm` function (Section 5) executes the passed program on a CUDA grid. Its dimensions are set in `dim`. A more sophisticated approach could also take the dimensions of the stencil into account, for instance. Note that the function passed to `nvvm` is allowed to use further target-specific functionality, like resolving the current thread index or the number of threads per group. Memory allocations on the target device are automatically managed by the generated code of the Impala compiler. Required data transfer from the host device to the GPU (and vice versa) is also performed automatically. However, developers can explicitly control memory mapping to target-specific memory regions like texturing memory by the provided functions `mmap` and `munmap`. The following variant of the `field_indices` function maps the input array `arr` to texturing memory.

```
fn field_indices(arr: &[float], body: fn(int, &[float]) -> ()) -> () {
  let dim = (arr.size, 1, 1);   // setup dimension
  let block = (128, 1, 1);      // default blocking in this case
  let texMapped = mmap<Tex>(arr);
  nvvm(dim, block, || {
    let index = nvvm_read_tid_x() + nvvm_read_ntid_x() * nvvm_read_ctaid_x();
    body(index, texMapped);
  });
  munmap(texMapped);
}
```

By introducing this concept, we are able to conveniently support unified-memory architectures, as well as architectures which require explicit memory transfer.

So far we have discussed three different implementations of `field_indices`: a straightforward CPU implementation, a CUDA-capable implementation and a CUDA-capable implementation supporting texture memory. A call to `field_indices` as introduced in Section 2 does not require any tweaking. Each variant acts as drop-in replacement of another one.

3.3. *Boundary handling*

In the code from Section 2, we ignore 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 { lower } else { idx }
}

fn bh_clamp_upper(idx: int, upper: int) -> int {
  if idx >= upper { upper - 1 } else { 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 specialize the code in a way that checks are only executed at the left and right border, there will be no 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) }         // adjust for left region
  if region == 1 { idx = bh_upper(idx, arr.size) } // adjust for right region
  arr(idx)                                          // return element
}
```

In order to specialize the internals of `apply_stencil_bh` function to the different regions, the iteration over the different values of `region` needs to be annotated. The call to `access` will automatically be specialized to the known values of `region` since we enforce specialization over the `region` variable. In addition, we also want the stencil computation to be specialized, and thus, also annotate the `stencil` iteration over the different indices:

```
fn apply_stencil_bh(arr: &[float], stencil: [float],
                    bh_lower: fn(int, int) -> int,
                    bh_upper: fn(int, int) -> int
                   ) -> () {
  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;
    }
  }
}
```

From a different point of view, the `apply_stencil_bh` function is an interpreter for stencils which realizes the domain-specific convolution. This interpreter can be specialized to a particular stencil via 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 [2, 3].

Now consider 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 right field region
```

Further specialization of the left field region eliminates the loop iteration and specializes the boundary-handling calls to `bh_clamp_lower`. This in turn evaluates the if-conditions of the boundary checks and the following code emerges:

```
// 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 right field region
```

In this way, the platform-specific mapping of the whole computation is transparent to the user of our stencil DSL. A machine expert can provide a platform-specific implementation of the stencil-computation functionality and the iteration logic over the elements. Moreover, such an implementation could take the dimensions of the stencil into account and can, for instance, schedule the iterations over the boundary-handling regions to the CPU and the center part to the GPU asynchronously.

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 [4] from the field of image processing. This filter smoothes images while preserving the sharp edges of an image. The computation of

```
fn main() -> () {
  let width   = 1024;
  let height  = 1024;
  let sigma_d = 3;
  let sigma_r = 5.0f;
  let     arr = ~[width*height: float]; // allocate a float array
  let mut out = ~[width*height: float]; // with width*height many elements

  let mask = @precompute(/*...*/);

  for i 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 + (xf, yf)) - arr(i);
        let s = exp(-c_r * diff*diff) *
                mask(xf + sigma_d)(yf + sigma_d);
        k += s;
        p += s * arr(i + (xf, yf));
      }
    }

    out(i) = p/k;
  }
}
```

Listing 1: Bilateral filter description in Impala.

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 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 runtime, 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 discuss the Impala compiler and show performance numbers on two different GPUs from NVIDIA of compiled Impala programs. Our benchmarks include generated code with boundary handling and special support for texturing

```
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);
  }
  sum
}

fn main() -> () {
  let mut arr = ~[width*height: float];
  let mut out = ~[width*height: float];
  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.

memory while previously published results [1] did not consider boundary handling and had no support for textures.

5.1. *The Impala Compiler*

Impala provides back ends for CPUs and GPUs. It uses a higher-order intermediate representation (IR) to perform code transformations. All transformations and code refinements described in Section 3 are applied at this level. Finally, Impala converts its IR to LLVM [5]. For NVIDIA GPUs, the compiler annotates the resulting LLVM IR to conform to the NVVM IR specification[b]. The CUDA compiler SDK[c] includes a tool to compile NVVM IR to PTX assembly—Nvidia's abstract execution format. Alternatively, the Impala compiler can annotate the LLVM IR to conform to Standard Portable Intermediate Representation (SPIR)[d]. SPIR is supported in OpenCL 1.2 via extensions and supports a wide variety of processors including GPU accelerators from other vendors. Furthermore, the Impala compiler utilizes Whole-Function Vectorization [6] to automatically vectorize code for the CPU.

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

5.2. *Performance*

In our experiments, we compare auto-generated variants from an Impala implementation of a Jacobi and a bilateral filter kernel to hand-tuned CUDA implementations. All programs ran on two GPU architectures from NVIDIA: The GTX 580 and GTX 680 GPUs. Performance numbers show the median execution time in milliseconds for seven runs.

Table 1: 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 %

5.2.1. *Jacobi Iteration*

In this experiment we measure the execution time of a single iteration of Listing 2. 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 1.

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 2. The table shows results for four different variants of the Jacobi kernel:

- a CUDA implementation where the stencil is hard-coded ("hand-specialized") instead of looping over a stencil array,
- a hand-tuned version of that with device-specific optimizations (custom kernel tiling, unrolling of the iteration space, and usage of texturing hardware),
- an Impala implementation where a generic stencil loop has been automatically specialized to the Jacobi stencil, and
- two variants where loop unrolling is enforced via partial evaluation and texture memory is used for the GTX 680 (see Section 3.2); additionally, boundary handling has been automatically specialized
  - using different kernels for each image region and boundary handling variant,
  - using one single big kernel that contains all boundary handling variants.

Impala's automatically specialized stencil code is as fast as the corresponding hand-specialized implementation in CUDA. In the case of boundary handling, we launch one kernel after another for each of the nine field regions. In the case of the GTX 580 the total running time is composed as follows:

$$0.33 = \underbrace{0.21}_{\text{main region}} + \underbrace{0.01 + 0.01 + 0.01 + 0.03 + 0.03 + 0.01 + 0.01 + 0.01}_{\text{boundary handling}}$$

In the case of the GTX 680 the total running time is calculated as follows:

$$0.37 = \underbrace{0.21}_{\text{main region}} + \underbrace{0.01 + 0.02 + 0.02 + 0.03 + 0.03 + 0.02 + 0.02 + 0.01}_{\text{boundary handling}}$$

Four of the regions are merely the corners of the field. Kernels in these regions only execute a few instructions and memory operations. Thus, we conclude that launching a kernel poses a overhead of roughly 0.01 ms – 0.02 ms. Instead of launching different kernels for boundary handling, we can also generate a single big kernel that includes the different specialized code variants for boundary handling (cf. [7]). Using a single kernel removes the launch overhead and the performance is almost the same as for the hand-tuned CUDA implementation on the GTX 680.

Table 2: 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)	0.35	0.37
CUDA (hand-tuned)	0.22	0.23
Impala (SS)	0.40	0.36
Impala (SS + BH)[†]	0.33	0.37
Impala (SS + BH)[‡]	0.26	0.24

[†] These numbers are the sum of the execution time for the main region as well as for all eight border regions. See Section 5.2.1.
[‡] Using one big kernel containing all specialized boundary handling variants (cf. [7]).

5.2.2. *Bilateral Filter*

For the bilateral filter we further break down our measurements and show the execution times when reading the input data from global memory (GMEM) as well as texture memory (TEX). We compare Impala's generated bilateral filter implementations with manual implementations in CUDA. The manual implementation computes the similarity component using the exponential function and the closeness component is precomputed and read from a lookup table stored to constant memory. The constants of the closeness component are propagated into the program

instructions in case the stencil computation is unrolled in Impala. Otherwise, the lookup table will be stored in the program code since Impala does currently not support constant memory.

Overall, we obtained mixed results as summed up in Table 3 and Table 4. In one case of specialized boundary handling, the overhead of launching eight additional kernels is included in the numbers as discussed in the previous section. In the other case, we use one big kernel which contains all boundary variants.

As previously mentioned, the manual version benefits from constant memory for the realization of the closeness function, whereas our specialized implementation propagates the constants during partial evaluation. For bigger kernels we observe extreme register pressure when generating unrolled code via NVVM: Registers are spilled to local memory, leading to longer execution times. The NVVM kernels for a 5×5 filter window require already the maximum of 63 registers, while the corresponding hand-tuned CUDA implementations a require less than 40 registers per thread, even when loop unrolling is enforced.

Table 3: Execution times in *ms* for the bilateral filter kernel on the GTX 580 and GTX 680 for a field of size 1024×1024 and a filter window size of 5×5 ($\sigma_d = 1$).

	GTX 580		GTX 680	
	GMEM	TEX	GMEM	TEX
CUDA (hand-tuned)	1.05	0.95	1.03	0.66
Impala (SS)	1.60	1.05	0.95	0.70
Impala (SS + BH)[†]	1.89	1.13	0.92	0.85
Impala (SS + BH)[‡]	1.49	1.07	0.85	0.74

[†] These numbers are the sum of the execution time for the main region as well as for all eight border regions. See Section 5.2.2.

[‡] Using one big kernel containing all specialized boundary handling variants (cf. [7]).

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 [8, 9], auto tuners [10], and DSLs [11, 12, 13] were developed.

We follow the direction of DSLs in our work, but we give the programmer additional opportunities to control the optimization process. Existing DSL approaches such as Liszt [11] and Pochoir [12] 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 [13]: Target-specific scheduling strategies can be defined by the programmer. Still it is not possible to trigger code refinement explicitly. Explicit code refinement

Table 4: Execution times in *ms* for the bilateral filter kernel on the GTX 580 and GTX 680 for a field of size 2048×2048 and a filter window size of 5×5 ($\sigma_d = 1$).

	GTX 580		GTX 680	
	GMEM	TEX	GMEM	TEX
CUDA (hand-tuned)	4.15	3.74	4.06	2.43
Impala (SS)	6.52	4.14	3.75	2.73
Impala (SS + BH)†	6.51	4.21	3.80	2.83
Impala (SS + BH)‡	5.33	4.20	3.10	2.77

† These numbers are the sum of the execution time for the main region as well as for all eight border regions. See Section 5.2.2.

‡ Using one big kernel containing all specialized boundary handling variants (cf. [7]).

can be achieved through staging like in Terra [14] 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 [16] 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 article, we present a DSL 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 outline how to use our concept for domain-specific mapping of stencil codes. As an application of domain-specific mapping, we show that we are able to generate specialized code variants by partial evaluation for different field regions.

Our research compiler is able to generate target code for execution on GPU accelerators as well as CPUs. The results show that GPU code we generate achieves competitive performance compared to manual implementations of different stencil codes. Our next steps include the enhancement of the integrated memory-mapping techniques in order to offer additional control over constant and shared memory.

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.

References

[1] Marcel Köster, Roland Leißa, Sebastian Hack, Richard Membarth, and Philipp Slusallek. Platform-Specific Optimization and Mapping of Stencil Codes through Refinement. In Armin Größlinger and Harald Köstler, editors, *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, pages 1–6, Vienna, Austria, January 2014.

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

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

[4] Carlo Tomasi and Roberto Manduchi. Bilateral Filtering for Gray and Color Images. In *Proceedings of the Sixth International Conference on Computer Vision (ICCV)*, pages 839–846. IEEE, January 1998.

[5] Chris Lattner and Vikram 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, March 2004.

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

[7] Richard Membarth, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Generating Device-specific GPU Code for Local Operators in Medical Imaging. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 569–581. IEEE, May 2012.

[8] Allison Baker, Robert Falgout, Tzanio Kolev, and Ulrike Meier Yang. Scaling Hypre's Multigrid Solvers to 100,000 Cores. *High-Performance Scientific Computing*, pages 261–279, 2012.

[9] Peter Bastian, Markus Blatt, Andreas Dedner, Christian Engwer, Robert Klöfkorn, Mario Ohlberger, and Oliver Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. *Computing*, 82(2):103–119, July 2008.

[10] Matthias Christen, Olaf Schenk, and Helmar 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.

[11] Zachary 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. 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, November 2011.

[12] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles 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.

[13] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo 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.

[14] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan 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.

[15] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus 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, October 2013.

[16] Tiark Rompf and Martin 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, October 2010.