A High-Performance Image Processing DSL for Heterogeneous Architectures

Kai Selgrad* Alexander Lier* Jan Dörntlein* Oliver Reiche† Marc Stamminger*

*Computer Graphics Group †Hardware/Software Co-Design
Friedrich-Alexander University Erlangen-Nuremberg, Germany

{kai.selgrad, alexander.lier, jan.doerntlein, oliver.reiche, marc.stamminger}@fau.de

ABSTRACT

Over the last decade a number of high performance, domain-specific languages (DSLs) have started to grow and help tackle the problem of ever diversifying hard- and software employed in fields such as HPC (high performance computing), medical imaging, computer vision etc. Most of those approaches rely on frameworks such as LLVM for efficient code generation and, to reach a broader audience, take input in C-like form. In this paper we present a DSL for image processing that is on-par with competing methods, yet its design principles are in strong contrast to previous approaches. Our tool chain is much simpler, easing the burden on implementors and maintainers, while our output, C-family code, is both adaptable and shows high performance. We believe that our methodology provides a faster evaluation of language features and abstractions in the domains above.

CCS Concepts

•**Software and its engineering** → **Domain specific languages; Source code generation;** *Preprocessors; Macro languages;*

Keywords

Domain Specific Langauges, Generative Programming, Common Lisp, Meta Programming

1. INTRODUCTION

Image processing is a wide field with diverse applications ranging from high performance computing (e.g. fluid simulation) and computer vision to medical imaging and post-processing of computer generated imagery (in games and movies). For many of these applications efficient execution is crucial and the tools provided should be accessible to users that are not hardware experts. Therefore, approaches based on domain-specific languages are very popular in image processing (see Section 2), especially when the applications are intended to run on different target platforms, possibly at the same time.

To satisfy the performance demands of these applications, image processing DSLs usually generate output in the form of C or C++, or even lower level representations such as LLVM [10] bytecode. Targeting more closed systems even requires generating code in vendor-specific languages, such as CUDA [13], or the lower level PTX [14]. Additionally, many DSLs are embedded in low level host languages, such as C++, to exploit the already existing parser front-end and AST construction.

In this paper we propose to use a Lisp-based design approach to DSL construction in the high-performance domain and demonstrate our DSL, Chipotle. Chipotle is heavily inspired by HIPA^{cc}[1] [11], a Clang-based, high-performance image processing DSL that targets heterogeneous applications. For instance, Chipotle incorporates efficient execution-patterns for GPU kernels that were found to perform well in HIPA^{cc}.

In contrast to HIPA^{cc}, our DSL builds on C-Mera[2] [17], a lightweight S-Expression to C-style transcompiler embedded in Common Lisp. The benefit of using C-Mera, especially as compared to Clang, is that it carries much smaller overhead and is more easily extensible. This argument indicates that for Chipotle it was a simple task to experiment with different notations, whereas HIPA^{cc} is strictly tied to valid C++ syntax. Providing more concise syntax would require changes to the C++ parser front-end, which is a highly non-trivial task. Using C-Mera's simple internal representation, extracting information from the input code is straightforward, as the input program itself is a (semantically annotated) syntax tree. With these two characteristics the task of expanding upon HIPA^{cc}'s feature-set (e.g. adding heterogeneous scheduling) is very problem-oriented and a correspondingly fast process. It should be noted, however, that Clang is a very powerful tool and provides, amongst others, complete syntactic and semantic analysis and ensures type safety in the input program. While our goal is not to detract from such commodities, we rather propose that a more lightweight and flexible solution benefits research and exploration, and that using a full C++ compiler toolchain might be excessive for the rather limited code generation involved in our target domain. When fully type-checked and deeply implicit information is required, choosing Clang in favour of C-Mera may prove reasonable.

[1] hipacc-lang.org

[2] github.com/kiselgra/c-mera

European Lisp Symposium 2016, Krakow, Poland

The remainder of this paper is structured as follows: In Section 2 we present an overview of the field of image processing DSLs and related approaches. Section 3 gives a short introduction to C-Mera, and our DSL is described in Section 4. In Section 5 we show two real-world examples of practical image processing operators, and compare our method's performance (in terms of executing time, application development time and DSL development effort) to a competing approach in Section 6. We conclude with a description of limitations and future work in Section 7.

The specific contributions of our work are:

— We present a new, high-performance image processing DSL targeting heterogeneous setups, Chipotle, that follows a Lisp-based methodology. The implementation is available and showcases an automatic generation of high-performance CUDA and AVX code from a single algorithm specification.

— We show how this language can be extended to automatically provide a schedule that employs GPUs and CPUs together, based on simple user input.

— We also show how easily these tasks are accomplished using powerful, but lightweight and flexible tools.

— Finally, we provide working examples that go beyond the simple filter setups commonly found in literature.

2. RELATED WORK

The feature set and optimization techniques of the presented Chipotle DSL are primarily inspired by HIPA^{cc} [11]. The HIPA^{cc} framework embodies a DSL for image processing, embedded into C++, and a source-to-source compiler, mainly focusing on point, local, and global operators. HIPA^{cc}'s compiler generates highly optimized code for different target architectures and languages, including CPUs, GPUs, and FPGAs through CUDA, OpenCL, Android's Renderscript, and Vivado HLS. HIPA^{cc} is based on the Clang/LLVM compiler infrastructure and performs AST-level optimizations based on domain and architecture knowledge. Architecture-specific optimizations include image padding for suitable memory alignment, the use of shared and texture memory, and thread-coarsening for GPUs. Furthermore, the automatic vectorization for common instruction sets of CPUs is supported, as well as the generation of a streaming pipeline for FPGAs. Optimizations based on domain knowledge, for instance, include the efficient handling of boundary conditions for local operators.

Halide [15] is also a DSL for image processing, based on the Clang/LLVM infrastructure, similar to HIPA^{cc}. Instead of imperatively describing image filters, a functional programming paradigm is applied, which enables additional sophisticated features, such as kernel fusion. Halide is capable of generating code for various target architectures, namely CUDA, OpenCL, PNaCL, as well as C++. Yet, this is not an entirely automatic process. The developer needs to specify a schedule that defines how the algorithm should be mapped onto the target architecture in order to obtain efficient code. Halide's schedule has to be manually specified and requires the developer to have a certain degree of architecture and domain knowledge. However, as the schedule is evaluated dynamically by the compiler, it can be altered or even entirely replaced at run time.

DeVito et al. [4] present Orion (and its source language, Terra), a stencil DSL for processing images, which is mainly inspired by Halide and makes use of mathematical operators that are implicitly evaluated on the whole image. This results in a very dense image processing pipeline representation. Although the DSL does not support the generation of code for different architectures, the vectorization module from its host language Terra can be used to map the stencil operations to vector instructions. Additionally, Orion adopted the ability to define a schedule for a filter pipeline from Halide, which led to very efficient results.

PolyMage [12] is a DSL for image processing where the image processing pipeline is represented as a directed, acyclic graph. Similar to our representation, this graph implicitly contains the data relationships between different operators and allows extracting this information to implement parallel scheduling for certain computations.

Patus [3] is a DSL and code generation framework for parallel stencil computations based on a C-like syntax. It follows a heterogeneous approach and is capable of generating code for CPU and GPU execution. Its auto-tuner is fed with a user-defined strategy to produce optimized code.

Native Common Lisp image processing libraries that target high-performance, such as Opticl[3], are, in contrast to the aforementioned methods, not DSLs themselves, but might be applicable as a basis for Lisp-only image processing DSL approaches. Targeting heterogeneous architectures might, however, prove problematic in such a setup.

A more general approach is proposed with frameworks for DSLs that can be used to create entirely new languages. Well-known representatives of this class are Delite [2], Asp [7], Terra [4] and AnyDSL [8]. Here, the framework performs generic, parallel and domain-specific optimizations for a new DSL without the necessity of starting development from scratch. Thereby, the effort to create DSLs can be drastically reduced. In a broader sense, C-Mera can also be attributed to this class of frameworks.

3. BRIEF REVIEW OF C-MERA

C-Mera is a simple transcompiler embedded in Common Lisp. It allows writing programs in an S-Expression syntax that is transformed to C-style code. This means that very simple extensions for languages with similar syntax are provided on top of the core C support. For example, the C-Mera distribution provides modules for C++, CUDA, GLSL and OpenCL. The main goal of providing an S-Expression syntax is to write the compiler such that it evaluates this syntax to construct a syntax tree when the input program is read, thereby allowing interoperability with the Common Lisp-system, most importantly by providing support for Lisp-style macros. To keep this part short we refer to the original C-Mera paper [17] for a more detailed description of the system and its implementation.

With the use of macros the input program no longer represents a plain syntax tree, but a semantically annotated tree that is transformed according to the implementation of the semantic nodes (macros). The utility of such a system ranges from simple, ad-hoc abstractions and programmer-centric simplifications [17] to providing otherwise hard to achieve programming paradigms for C-like languages [16] and even to fully fledged domain-specific languages.

The following example, taken from our domain, shows the definition of a simple image filter:

[3] `github.com/slyrus/opticl`

(a) Input image (b) Laplace filtered (c) Blur & Laplace (d) Blur, Laplace & blur

Figure 1: Input image and results from using edge detection via a Laplace operator.

```
(function filter ((float *data) (float *mask) (int filter-w)
                  (int filter-h) (int w) (int h)) -> void
  (for ((int y 0) (< y h) ++y)
    (for ((int x 0) (< x w) ++x)
      (decl ((float accum 0.0f))
        (for ((int dy 0) (< dy filter-h) ++dy)
          (for ((int dx 0) (< dx filter-w) ++dx)
            (set accum
                 (+ accum
                    (* (aref mask (+ (* filter-w dy) dx))
                       (aref data
                             (+ (* (+ y (- (/ filter-h 2)) dy)
                                   w)
                                x (- (/ filter-w 2)) dx)))))))
        (set (aref data (+ (* y filter-w) x))
             accum)))))
```

This describes a C function that takes an array as its input (e.g. a grayscale image) and applies the a dimensional filter mask. Filtering proceeds by iterating over the input image. The weighted average is computed for each pixel using the provided filter mask (centered at the current pixel). Using C-Mera it is easy to reduce the code for this algorithm to a simplified description, as presented in the following listing.

```
(defilter filter (data mask (w h) (filter-w filter-h))
  (loop2d (x y w h)
    (decl ((float accum 0.0f))
      (loop2d (dx dy filter-w filter-h)
        (set accum (* (mask dx dy)
                      (cell (+ x (- (/ filter-w 2)) dx)
                            (+ y (- (/ filter-h 2)) dy)))))
      (set (cell x y) accum))))
```

This is achieved by a set of simple macrolets:

```
(defmacro defilter
    (name (data mask (w h) (filter-w filter-h)) &body body)
 `(function ,name
            ((float* ,data) (float* ,mask)
            ,@(loop for x in (list w h filter-w filter-h)
                    collect `(int ,x))) -> void
     (macrolet
       ((loop2d ((x y w h) &body body)
                `(for ((int ,y 0) (< ,y ,h) (+= ,y 1))
                   (for ((int ,x 0) (< ,x ,w) (+= x 1))
                     ,@body)))
        (mask (x y) `(aref ,',mask (+ (* ,',filter-w ,y) ,x)))
        (cell (x y) `(aref ,',data (+ (* ,',w ,y) ,x))))
       ,@body)))
```

Naturally, further shorthands and simplifications can be incorporated into this kind of macro. Section 4 shows the language that evolved from these considerations and Section 5 shows to more advanced examples.

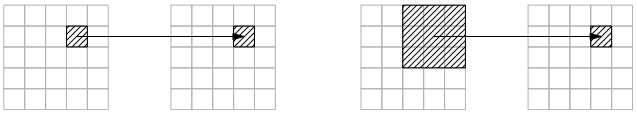

Figure 2: Illustration of point operators (left) and local filters (right).

```
(filter-graph laplacian
  (edge load-base (:output base)
    (load-image :file "test.jpg"))
  (edge laplacian (:input base :output lapla :arch cuda)
    (deflocal
       :mask ((-1 -1 -1)
              (-1  8 -1)
              (-1 -1 -1))
       :aggregator +
       :operator *
       :finally (set accum.x (- 255 (fabs accum.x))
                     accum.y (- 255 (fabs accum.y))
                     accum.z (- 255 (fabs accum.z)))))
  (edge store-ublapla (:input lapla)
    (store-image :file "out.png")))
```

Figure 3: A simple, but complete, Chipotle-program that filters an input image using a Laplace operator.

4. THE CHIPOTLE-DSL

In this section we describe Chipotle, our domain-specific language for image processing. Chipotle provides a very concise notation, is easily extensible and expands into high-performance code for both GPUs (using CUDA) and CPUs (using SSE and AVX). It furthermore provides heterogeneous scheduling based on simple tagging.

As already mentioned, Chipotle is heavily inspired by HIPA^{cc}. HIPA^{cc} is based on the Clang/LLVM infrastructure, which introduces two major drawbacks. First of all the DSL syntax is restricted to its host language C++ and causes the DSL to be rather verbose. Secondly, extending the DSL with new language constructs is a very time consuming process. HIPA^{cc} uses the Clang-AST to substitute certain nodes with domain-specific variants. This AST, however, is generated after a complete semantic analysis of the input C++ code and is thus extremely detailed. Filtering the relevant nodes to extend and adapt the AST for a particular domain is a cumbersome task. Therefore, Chipotle was designed to counter these complications by providing a very concise and declarative style that aims to be easily extensible.

4.1 Notation

As a running example, we will consider the definition of a simple Laplace operator (see Figure 1 (b)). In image processing, the Laplace operator is a simple filter that can be used for edge detection. The code listed in Figure 3 shows how this operator can be expressed using Chipotle. The principal component of a Chipotle-program is the *filter graph*. The contents of a filter graph are nodes that represent (intermediate) images and *edges* that specify transformations on those images. In the code given in Figure 3 there are three image operations: loading an existing image from disk,

```
(edge box (:input base :output filtered :arch cuda)
  (deflocal
    :extent (5 5)
    :grayscale t
    :accum-name val
    :codelet (set val (+ val (local-ref base rel-x rel-y)))
    :finally (set val (/ val 25))))
```

Figure 4: A box-filter stage illustrating the use of `codelet`. The same operation can be implemented by using a filter mask of all ones.

filtering it with a local operator and finally storing it back to disk. The images themselves are implicit in the connection information given with the edges (e.g. `laplacian` takes input from `base` and stores its result in `lapla`).

The largest part of the `laplacian` graph is the description of the local operator. A local operator (see Figure 2), L, is a function on an image, I, mapping a neighborhood, N, of a given pixel of to an single output pixel value [1]:

$$L(x, y) = \oplus_{(i,j)\in N} f_I(x, y, i, j)$$

The most common case of this is discrete convolution using a convolution matrix M (as `mask` in the example above):

$$L(x, y) = \sum_{i,j=0}^{i,j<n} M_{ij} I_{x+i-\lfloor n/2 \rfloor, y+j-\lfloor n/2 \rfloor}$$

Inspired by HIPA^{cc}'s implementation we provide an implicit looping mechanism where only the operator and aggregator, $\otimes$ and $\oplus$, respectively, must be specified and we assume

$$f_I(x, y, i, j) = M_{ij} \otimes I_{x+i-\lfloor n/2 \rfloor, y+j-\lfloor n/2 \rfloor}.$$

The result of the local filter can be further adapted via the `:finally` clause. In the example above we use this clause to store the absolute value of the result. It is also possible to specify an arbitrary function for f by providing a `:codelet`, as with the simple box filter shown in Figure 4. There, the relative positions during the iteration are available in `rel-x` and `rel-y` (naturally, the names of such generator-defined variables, as found throughout this section, can be specified explicitly, too). For a more elaborate example see Section 5.1.

In accordance with Bankman [1] and HIPA^{cc} we also provide a simpler form, the point operator (see Figure 2). Instead of mapping a region of the input image to an output pixel, point operators map input pixels to output pixels without considering the pixel's neighborhood. The following fragment is part of the well-known Harris corner detector [5] that determines whether a pixel is part of a corner in the input. The computation depends on the two input images `xd` and `yd`, which hold the gradients of the original input in x and y direction.

```
(edge hcd (:input (xd yd) :output out :arch cuda)
  (defpoint (:grayscale t)
    (decl ((float xx (* xd xd))
           (float xy (* xd yd))
           (float yy (* yd yd))
           (float M (abs (- (- (* xx yy) (* xy xy))
                            (* 0.04 (+ xx yy) (+ xx yy)))))
           (float res 0))
      (if (< threshold M) (set res 255))
      (set out res))))
```

Since, for point operators, the neighborhood should not be available the names of the input images are mapped to reference the current pixel-location in the respective images.

```
(filter-graph laplacian
  (edge load-base (:output base)
    (load-image :file "test.jpg"))
  (edge gauss (:input base :output blurred :arch cuda)
    (deflocal
      :mask #.(sample-filter #'gaussian 5 :sigma 1.5)
      :aggregator +
      :operator *))
  (edge laplacian (:input blurred :output lapla :arch cuda)
    (deflocal
      :mask ((-1 -1 -1)
             (-1  8 -1)
             (-1 -1 -1))
      :aggregator +
      :operator *
      :finally (set accum.x (- 255 (fabs accum.x))
                    accum.y (- 255 (fabs accum.y))
                    accum.z (- 255 (fabs accum.z)))))
  (edge to-grayscale (:input lapla :output gray :arch cuda)
    (defpoint ()
      (decl ((float out (+ (* 0.2126 (lapla 0))
                           (* 0.7152 (lapla 1))
                           (* 0.0722 (lapla 2)))))
        (set (gray 0) out
             (gray 1) out
             (gray 2) out))))
  (edge gauss2 (:input gray :output output :arch cuda)
    (deflocal
      :mask #.(sample-filter #'gaussian 7 :sigma 4)
      :grayscale t
      :aggregator +
      :operator *))
  (edge store-out (:input output)
    (store-image :file "out.png")))
```

Figure 5: The complete filter graph that transforms the image shown in Figure 1 (a) to that of (d).

4.2 Filter Graphs

As visible in the short graph given in Figure 3, the body of a filter graph consists of the edges that describe how images are transformed. Note that the body of the graph is evaluated and thus can also hold arbitrary forms, for example a set of user- or subdomain-specific macrolets (see below).

In our current implementation the roots of the graph (load operations) are found and used as a seed for topological sorting. Inconsistent graphs (e.g. containing unavailable input nodes) are rejected. Input and output is meant to be executed on the host-CPU, however this is by no means a systematic restriction and, in a future version, we plan on being able to connect Chipotle to hardware-rendered input images or interactive display using OpenGL.

Expanding on the example of using a Laplace operator at the outset of Section 4.1 the filtered version of Figure 1 (a), shown in (b), exhibits strong noise. This is due to the fact that the detection operator picks up very fine detail. Therefore it is common to pre-process images with a low-pass filter to remove small-scale detail prior to detection operators. Figure 1 shows further versions where the image (c) has been filtered with a Gaussian kernel before edge detection (and converted to grayscale) and (d) with an additional low-pass filter applied after edge detection to obtain a smoother image. Figure 5 shows the complete filter graph for this process. Note how the weights for the Gaussian kernels are computed beforehand (line 6 and 29). Figure 6 lists the same algorithm, but with a few convenience macros provided externally. For an example with proper macros in a filter graph see Section 5.

```
(filter-graph laplacian
  (edge load-base (:output base)
    (load-image :file "test.jpg"))
  (simple-filter base blurred
    (:arch cuda :mask #.(sample-filter #'gauss 5 :sigma 1.5)))
  (simple-abs-filter blurred lapla
    (:arch cuda :mask ((-1 -1 -1)
                       (-1  8 -1)
                       (-1 -1 -1))))
  (grayscale-conversion
      lapla gray :arch cuda)
  (simple-filter gray output
      (:arch cuda :mask #.(sample-filter #'gauss 7 :sigma 4))
      :grayscale t)
   (edge store-out (:input output)
      (store-image :file "out.png")))
```

Figure 6: The same graph as shown in Figure 5, however with a few convenience macros.

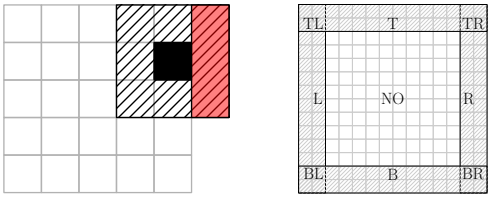

Figure 7: Boundary checks are required to ensure that only valid pixel locations are queried (left). To remove irrelevant bounds checks (e.g. for the inner part of the image, labelled NO) the image is partitioned (right).

4.3 Checked Memory Access

The local operators shown above are notationally very simple and do not contain explicit boundary checks. These checks are automatically introduced for the iteration over the filter mask such that, for example, when accessing pixels to the right of the target pixel's location Chipotle only inserts checks for the right image border (see Figure 7).

For accessing locations outside the image $I(x, y)$ (i.e. $x \notin \{0, \ldots, w-1\} \vee y \notin \{0, \ldots, h-1\}$) different modes for changing the input coordinates, inspired by OpenGL's texture wrapping functions [19], are provided. The keyword parameter `:wrap` can be used to this end to compute x', y' with `mirror` ($x' = w - x$), `wrap` ($x' = x \mod w$) and `clamp` ($x' = \min(w-1, \max(0, x))$). Furthermore, `border` ($I(x, y) = \text{const}$) provides a constant border color.

In order to efficiently compute a filter over large images, it is also possible to partition the input image into areas requiring different boundary checks [11]. Since local filters usually employ a very small filter mask compared to the image size this ensures that no boundary checks are performed at all for the inner, and largest, part of the image. This combines well with generating border conditions based on the location in the filter mask: a check is only generated if the position in the filter mask is, for example, to the left and the active region after partitioning includes the left image edge. In the setting shown in Figure 7 (left), the local operator requires checked accesses only towards the right. Figure 7 (right) shows the different regions. Inspired by HIPA^{cc}'s implementation we provide a single function that checks for the appropriate boundary-handling scheme to be used and jumps to it. The benefit of this scheme is that

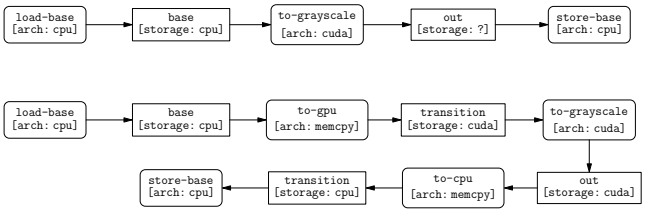

Figure 8: Top: Execution plan for the trivial graph shown in Figure 9 (top). Images are rectangular nodes, operators rounded. Bottom: Plan for the same graph, but with `:arch cuda` for the point operator, including transition edges and images.

it maps well to our target architectures (CUDA, SSE and AVX) as long as it does not introduce divergence, which is easily ensured (see Section 4.4).

4.4 Heterogeneous Image Processing

Chipotle allows transforming its input programs to SSE and AVX with multi-threading as well as to CUDA. The target architecture of an image operation can be specified via the `:arch` parameter (see Figure 3). Note that different target architectures can be mixed freely in the same program.

For SSE and AVX the operations map to multi-threaded, two dimensional iterations over the input image. The specified operation (for local operators including the iteration over the filter mask) is then executed in groups of 4 (SSE) or 8 (AVX) pixels. This is achieved by automatically vectorizing of the provided code. To this end we map the content of declarations (`(decl (...)...)`) to appropriate vectorized types and transform the user-provided arithmetic operations to corresponding vectorized versions. We also sequentialize conditional statements and track the masks of the true and false cases to ensure correct merging. Figure 9 shows an example of these operations for a very simple point operator, converting a color image to grayscale.

For operators to be instantiated for CUDA we generate a kernel function and a host-stub that addresses appropriate parameter forwarding. Transferring image data to the GPU and back to the host memory is implicit in the graph. Host-only operations such as loading and storing images (see Section 4.2 on this limitation) force the initial and final locations of image data. Furthermore, edges that operate on CUDA require that their input and output be present on the GPU. This is resolved by traversing the filter graph in order and introducing transition edges and images where appropriate. The location of images is propagated through the graph and only switches on the previously introduced transition nodes. For operators that do not specify a target architecture it is propagated similarly, to avoid expensive host/device transfers.

When targeting CPU vector instructions (SSE or AVX) or parallel GPU code (CUDA) we take care to incorporate the execution width in boundary handling conditions. For example, with AVX the generated code executes an 8×1 image region using a single control flow. If different paths of control flow are to be applied within such a group the code must be sequentialized. Therefore, bounds checks are conservatively clamped to multiples of 8 in the x direction for AVX. For CUDA, where the execution configuration is more flexible, borders are adjusted accordingly.

```
(filter-graph example
  (edge load-base (:output base)
    (load-image :file "test.jpg"))
  (edge gray (:input base :output out :arch sse)
    (defpoint ()
      (decl ((const float r (base 0))
             (const float g (base 1))
             (const float b (base 2))
             (const float luma (+ (* 0.2126 r) (* 0.7152 g)
                                  (* 0.0722 b)))
             (float res))
        (if (> luma .5)
            (set res luma)
            (set res 0))
        (set (out 0) res))))
  (edge store (:input out)
    (store-image :file "out.png")))
```

```
void gray(unsigned char *base, unsigned char *out, unsigned int w, unsigned int h)
{
    unsigned int vecLength = (w * h) - ((w * h) % 4);
    const __m128 xmm_constant_0_5__165    = _mm_set1_ps(5.00000000e-1);
    const __m128 xmm_constant_0_0722__164 = _mm_set1_ps(7.22000000e-2);
    const __m128 xmm_constant_0_7152__163 = _mm_set1_ps(7.15200000e-1);
    const __m128 xmm_constant_0_2126__162 = _mm_set1_ps(2.12600000e-1);
    const __m128 xmm_constant_1_0__161    = _mm_set1_ps(1.00000000e+0);
    for (unsigned int i = 0; i < vecLength; i += 4) {
        //Load: (base 2) to xmm290
        const __m128i xmm291 = _mm_cvtsi32_si128((*((const int*)
                                                &base[(i + (2 * w * h))])));
        const __m128i xmm292 = _mm_unpacklo_epi8(xmm291, _mm_setzero_si128());
        const __m128i xmm293 = _mm_unpacklo_epi16(xmm292, _mm_setzero_si128());
        const __m128 xmm290 = _mm_cvtepi32_ps(xmm293);
        //Load: (base 1) to xmm289
        const __m128i xmm294 = _mm_cvtsi32_si128((*((const int*)
                                                &base[(i + (1 * w * h))])));
        const __m128i xmm295 = _mm_unpacklo_epi8(xmm294, _mm_setzero_si128());
        const __m128i xmm296 = _mm_unpacklo_epi16(xmm295, _mm_setzero_si128());
        const __m128 xmm289 = _mm_cvtepi32_ps(xmm296);
        //Load: (base 0) to xmm288
        const __m128i xmm297 = _mm_cvtsi32_si128((*((const int*)
                                                &base[(i + (0 * w * h))])));
        const __m128i xmm298 = _mm_unpacklo_epi8(xmm297, _mm_setzero_si128());
        const __m128i xmm299 = _mm_unpacklo_epi16(xmm298, _mm_setzero_si128());
        const __m128 xmm288 = _mm_cvtepi32_ps(xmm299);
        const __m128 r = xmm288;
        const __m128 g = xmm289;
        const __m128 b = xmm290;
        const __m128 luma = _mm_add_ps(
                                _mm_add_ps(_mm_mul_ps(xmm_constant_0_2126__162, r),
                                           _mm_mul_ps(xmm_constant_0_7152__163, g)),
                                _mm_mul_ps(xmm_constant_0_0722__164, b));
        __m128 res;
        const __m128 cond395 = _mm_cmpgt_ps(luma, xmm_constant_0_5__165);
        const __m128 mask396 = cond395;
        res = _mm_or_ps(_mm_and_ps(mask396, luma), _mm_andnot_ps(mask396, res));
        const __m128 mask397 = _mm_andnot_ps(cond395,
                                        _mm_set1_ps(xmm_constant_1_0__161));
        res = _mm_or_ps(_mm_and_ps(mask397, 0), _mm_andnot_ps(mask397, res));
        //Store: (out 0)
        const __m128i xmm1129 = _mm_cvtps_epi32(res);
        const __m128i xmm1130 = _mm_packs_epi32(xmm1129, xmm1129);
        const __m128i xmm1131 = _mm_packus_epi16(xmm1130, xmm1130);
        (*((int*)&out[(i + (0 * w * h))])) = _mm_cvtsi128_si32(xmm1131);
    }
    for (unsigned int i = vecLength; i < (w * h); ++i){
        const float r = base[i + (0 * w * h)];
        const float g = base[i + (1 * w * h)];
        const float b = base[i + (2 * w * h)];
        const float luma = (2.12600000e-1 * r) + (7.15200000e-1 * g)
                         + (7.22000000e-2 * b);
        float res;
        if (luma > 5.00000000e-1)
            res = luma;
        else
            res = 0;
        out[i + (0 * w * h)] = res;
    }
}
```

Figure 9: Top: Chipotle input graph for converting a color image to grayscale. Bottom: Generated SSE code for the `gray` edge.

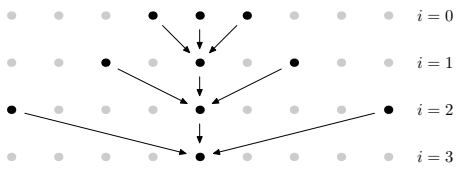

Figure 10: À-Trous filtering: Iterative application of a small filter with increasing gap between samples.

5. EXAMPLES

In this section we provide two examples of using Chipotle and describe further language features and practical considerations that arise.

5.1 Edge-Avoiding À-Trous Filter

Many image processing algorithms (e.g. the night filter following this example) require an image smoothing step that does not filter across edges, that is, a filter that smoothes regions that are similar, but maintains the sharpness of the image. This effect can be achieved by using a bilateral filter [21]. Such filters do not only weight pixel values by their spatial distance (such as the previous local fiters, e.g. the Gauss kernel), but also by difference in value. However, due to non-linearity, these filters are not separable and are thus very expensive to compute for large regions.

A common way to accelerate the computation of this filter is using the À-Trous (with holes) algorithm [18], where a filter with small support is iteratively applied while increasing the image-space gap between the sample locations in each iteration. Figure 10 illustrates this for a one-dimensional filter. Such a filter is easily constructed by filling in the appropriate number of zero-entries in the filter mask and it can then be used as a simple local operator. The large neighborhood introduced by this is easily reduced when checking for zero-coefficients while unrolling the loop over the local filter. Thus, for a Gaussian 3×3 À-Trous filter at iteration 3 the filter mask is 17×17 but the actual number of operations (and bounds checks, if appropriate) is still 9.

Our macro, which provides the code for the edge-avoiding À-Trous filter, is given in Figure 11. It shows how more complicated operators can be made available to various filter graphs (see the next example, for instance). As the bilateral filter is not a simple accumulation we exploit the flexibility of our `deflocal` implementation. After fixing parameter and filter names at the beginning, we set up our accumulators and reference values in line 12. There, `r0` references the red color component at the pixel of the filter's center. We accumulate color to `r`, `g` and `b` and also accumulate the weight, `W`, by which the result must normalized. The provided `:codelet` is then evaluated for the non-zero entries of the filter mask and combines the weights in the domain (i.e. the mask's value) and in the range (i.e. weighted by similarity, e.g. by an exponential term). Finally, we normalize the result and write it back to memory.

5.2 Night Tonemapping

The standard method of transforming images taken under daytime lighting conditions to look as if taken by night is by reducing brightness, blurring similar colors (reducing acuity), and shifting the colors towards more bluish tones [20]. In our implementation of such an algorithm we first blur similar regions in the input image by using the bilateral filter shown in the previous example. After a few iterations of this local operator we compute the actual scotopic image by applying a blue shift. To this end we follow Jensen et al. [6] and first convert the image to the XYZ color space, reduce the brightness, Y, compute the scotopic luminance [9], V, and use it to compute a darkened image that is shifted to a more bluish tone. Finally, we convert the image back to the RBG color space. Figure 12 shows the effect computed by the filter graph given in the lower part of Figure 11.

```
1  (defmacro atrous-step (n &key (pre "atrous") input output arch)
2    (let ((in  (cl:if input input
3                      (cintern (format nil "~a~a" pre (cl:1- n)))))
4          (out (cl:if output output
5                      (cintern (format nil "~a~a" pre n)))))
6      `(edge ,(cintern (format nil "compute-~a~a" pre n))
7           (:input ,in :output ,out :arch ,arch)
8         (deflocal
9           :mask ,(atrous '((0.057118 0.124758 0.057118)
10                           (0.124758 0.272496 0.124758)
11                           (0.057118 0.124758 0.057118)) n)
12          :initially ((float r0 (/ (,in 0 0 0) 255.0f))
13                      (float g0 (/ (,in 0 0 1) 255.0f))
14                      (float b0 (/ (,in 0 0 2) 255.0f))
15                      (float r 0) (float g 0) (float b 0)
16                      (float W 0))
17          :codelet
18              (decl ((float R (/ (,in rx ry 0) 255.0f))
19                     (float G (/ (,in rx ry 1) 255.0f))
20                     (float B (/ (,in rx ry 2) 255.0f))
21                     (float w0 (mask rx ry))
22                     (float rd (- R r0))
23                     (float gd (- G g0))
24                     (float bd (- B b0))
25                     (float w1 (+ (^2 rd) (^2 gd) (^2 bd))))
26                (set w1 (* (fminf 1.0f (expf (- (* w1 1)))) w0))
27                (set W (+ W w1))
28                (set r (+ r (* R w1))
29                     g (+ g (* G w1))
30                     b (+ b (* B w1)))))
31          :finally (set (,out 0) (* 255.0f (/ r W))
32                        (,out 1) (* 255.0f (/ g W))
33                        (,out 2) (* 255.0f (/ b W)))))))
```

```
1  (defmacro nightvision-filter (&key (iterations 3))
2    `(filter-graph blub
3       (edge load-base (:output base) (load-image :file "test.jpg"))
4
5       (atrous-step 0 :input base)
6       (atrous-step 1)
7       ,@(loop for i from 1 to (cl:1- iterations)
8               collect `(atrous-step ,i))
9       (atrous-step ,iterations :output prefiltered)
10
11      (edge scoto (:input prefiltered :output scotopic :arch cuda)
12        (defpoint ()
13          (decl ((float r (prefiltered 0))
14                 (float g (prefiltered 1))
15                 (float b (prefiltered 2))
16                 (float X (to-X r g b)
17                 (float Y (* (to-Y r g b) 0.33f))
18                 (float Z (to-Z r g b))
19                 (float V (scotopic-luminance X Y Z))
20                 (float W (+ X Y Z))
21                 (float s (* Y 0.2))
22                 (float xl (/ X W))
23                 (float yl (/ Y W))
24                 (const float xb 0.25)
25                 (const float yb 0.25))
26            (set xl (+ (* (- 1.0f s) xb) (* s xl))
27                 yl (+ (* (- 1.0f s) yb) (* s yl))
28                 Y  (+ (* V 0.4468f (- 1 s)) (* s Y))
29                 X  (/ (* xl Y) yl)
30                 Z  (- (/ X yl) X Y))
31            (decl ((float rgb_r (to-r X Y Z))
32                   (float rgb_g (to-g X Y Z))
33                   (float rgb_b (to-b X Y Z)))
34              (set (scotopic 0) (fminf 255.0f (fmaxf 0.0f rgb_r)))
35              (set (scotopic 1) (fminf 255.0f (fmaxf 0.0f rgb_g)))
36              (set (scotopic 2) (fminf 255.0f (fmaxf 0.0f rgb_b)))))))
37
38      (edge store-scotopic (:input scotopic)
39        (store-image :file "night.jpg"))))
40
41 (nightvision-filter :iterations 3)
```

Figure 11: Top: Our macro that generates different iterations of the edge-avoiding À-Trous blur filter. Bottom: The filter is used as a pre-process for the night filter.

Figure 12: Input image (left) and night-filtered version (right). Note how not only the tone changed, but also many details are blurred out while edges (such as the roof, columns and the balcony) are still clearly visible.

6. EVALUATION

In the following we give a brief evaluation of implementing filter graphs using our DSL, Chipotle, and compare them to HIPA^{cc}. We focus on the night filter with three iterations of edge-avoiding À-Trous filtering (see Section 5.1) followed by night tonemapping (see Section 5.2). This corresponds to evaluating line 41 in the lower part of Figure 11.

In terms of filtering performance CUDA code generated by HIPA^{cc} takes 14.7 ms to filter the 1754×1280 image shown in Figure 12 (left) on a Nvidia Geforce GTX 680 graphics card. HIPA^{cc}'s SSE2 version takes 400 ms running on a Intel Xeon E5-1620 processor running at 3.50 GHz, using OpenMP for parallelization. Our code generated by Chipotle runs equally fast at 14.9 ms on CUDA, while our SSE2 version lags behind at 901 ms, with the same hardware. Due to the use of domain knowledge and the ability to generate code for special cases (e.g. for border handling) these computation times are hard to achieve with hand-written code [11].

Regarding code size there are two factors we consider: firstly the size of the code written in the DSL, and secondly the size of the DSL's code base itself, which is of particular importance when considering extensions and maintenance of the DSL. The HIPA^{cc}-version of the night-filter shown in Figure 11 (bottom) consists of 264 lines of code. The complete code for use with Chipotle, including the expansion of a simple input mask to an À-Trous filter (`atrous`, see line 9 in Figure 11 (top)) and the edge-avoiding filter, totals 85 lines. This is mainly due to the fact that our notation contains almost no boilerplate code.

The HIPA^{cc} distribution we used to take the above measurements consists of 48941 lines of code. As described earlier this includes a number of additional back-ends that are not available for Chipotle (most notably Renderscript and VivadoHLS for which there is no support in C-Mera). However, even with the additional back-ends the code base appears immense when compared to Chipotle's 978 lines of code (as-is, and including vectorization).

7. CONCLUSION

In this paper we presented our new image processing DSL, Chipotle, and showed how easily it is constructed using an existing, Common Lisp-based tool chain. At only 2% of the code size of competing methods our DSL yields highly optimized code that runs on-par with the state of the art on GPUs using CUDA. Our SSE2 version runs at around 50% the performance of HIPA^{cc}'s vectorized output. It should be

noted that, even in this case, our performance is significantly faster than results from using a compiler's auto-vectorization routines as these cannot rely on domain knowledge. However, we believe that this performance gap is an artefact of our not yet fully matured vectorization routines and that Chipotle will catch up with HIPA^{cc} shortly. We further believe that extensions to our DSL are much simpler as its implementation is very short and uses higher-level programming paradigms, most notably Common Lisp macros and feature-oriented programming [16].

We also showed that the implementation of a filter graph using Chipotle is only around 33% of the code size of the corresponding HIPA^{cc} implementation and that the DSL code itself is still amenable to further abstractions and simplifications using macros. Thus, we are confident that Chipotle can compete with state of the art image processing DSLs while increasing productivity both on the level of the DSL users and implementors.

Acknowledgments

The authors gratefully acknowledge the generous funding by the German Research Foundation (GRK 1773).

8. REFERENCES

[1] I. N. Bankman. *Handbook of Medial Image Processing and Analysis*. Academic Press, Burlington, second edition edition, 2009.

[2] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 89–100, Oct 2011.

[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 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA, 2013. ACM.

[5] C. Harris and M. Stephens. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.

[6] H. W. Jensen, S. Premoze, P. Shirley, W. B. Thompson, J. A. Ferwerda, and M. M. Stark. Night rendering. Technical Report UUCS-00-016, Computer Science Department, University of Utah, Aug. 2000.

[7] S. Kamil, D. Coetzee, and A. Fox. Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2011.

[8] 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.

[9] G. W. Larson, H. Rushmeier, and C. Piatko. A visibility matching tone reproduction operator for high dynamic range scenes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):291–306, Oct. 1997.

[10] 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'04)*, Palo Alto, California, Mar 2004.

[11] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. HIPAcc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):210–224, 2016.

[12] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, 2015.

[13] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, September 2015.

[14] NVIDIA Corporation. *Parallel Thread Execution ISA Version 4.3*, September 2015.

[15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[16] K. Selgrad, A. Lier, F. Köferl, M. Stamminger, and D. Lohmann. Lightweight, generative variant exploration for high-performance graphics applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 141–150, New York, NY, USA, 2015. ACM.

[17] K. Selgrad, A. Lier, M. Wittmann, D. Lohmann, and M. Stamminger. Defmacro for C: Lightweight, ad hoc code generation. In *Proceedings of ELS 2014 7rd European Lisp Symposium*, pages 80–87, 2014.

[18] M. J. Shensa. The discrete wavelet transform: wedding the a trous and mallat algorithms. *IEEE Transactions on Signal Processing*, 40(10):2464–2482, 1992.

[19] D. Shreiner and T. K. O. A. W. Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009.

[20] W. B. Thompson, P. Shirley, and J. A. Ferwerda. A spatial post-processing algorithm for images of night scenes. *J. Graphics, GPU, & Game Tools*, 7(1):1–12, 2002.

[21] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, pages 839–, Washington, DC, USA, 1998. IEEE Computer Society.