HIPA^{cc}: A Domain-Specific Language and Compiler for Image Processing

Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert

Abstract—Domain-Specific Languages (DSLs) provide high-level and domain-specific abstractions that allow expressive and concise algorithm descriptions. Since the description in a DSL hides also the properties of the target hardware, DSLs are a promising path to target different parallel and heterogeneous hardware from the same algorithm description. In theory, the DSL description can capture all characteristics of the algorithm that are required to generate highly efficient parallel implementations. However, most frameworks do not make use of this knowledge and the performance cannot reach that of optimized library implementations.

In this article, we present the HIPA^{cc} framework, a DSL and source-to-source compiler for image processing. We show that domain knowledge can be captured in the language and that this knowledge enables us to generate tailored implementations for a given target architecture. Back ends for CUDA, OpenCL, and Renderscript allow us to target discrete Graphics Processing Units (GPUs) as well as mobile, embedded GPUs. Exploiting the captured domain knowledge, we can generate specialized algorithm variants that reach the maximal achievable performance due to the peak memory bandwidth. These implementations outperform state-of-the-art domain-specific languages and libraries significantly.

Index Terms—domain-specific language, image processing, code generation, source-to-source translation, GPU, CUDA, OpenCL, Renderscript.

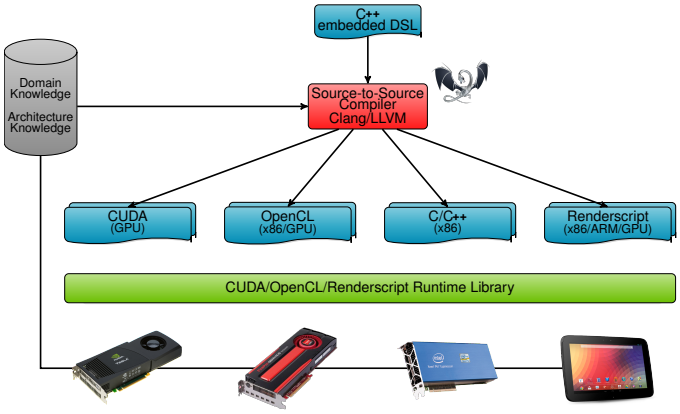


Figure 1: HIPA^{cc} framework: Generation of efficient low-level code based on an algorithm description in a DSL utilizing target hardware and domain knowledge.

1 Introduction

Image processing is one of the basic and ubiquitous algorithm classes. Yet, it is challenging and poses high demands on the environment in which they are embedded in: Mobile devices like phones and tablets strive for efficient implementations to save battery live; driver assistance systems require image processing to be *in time*; and systems in medical imaging have extremely high data volumes that have to be processed fast. All these systems require highly efficient implementations of image processing algorithms on today's more and more (massively) parallel hardware.

This poses challenges to algorithm developers that are typically no machine experts. *What (parallel) language should be used for implementation?* While CUDA might be the number one choice on GPUs from NVIDIA, there is no CUDA support from other vendors. OpenCL is well supported by most hardware manufacturers, but Google removed OpenCL support on Android. *What optimizations should be applied for a given target platform?* Target-specific optimization require profound knowledge of the target hardware. Although languages like CUDA and OpenCL provide functional portability across a range of devices, target-specific tuning is required for good performance [1].

One compelling solution to tackle these challenges are DSLs: Algorithms can be described concisely in a high-level language and code for different target architectures can be generated from this description. All the required knowledge for optimization is captured by DSL constructs and platform-specific optimization strategies can be provided as additional input. This allows to generate highly efficient implementations from a common algorithm description that is not contaminated with target-dependent code for a variety of domains such as image processing [2], [3], PDE solvers [4], or machine learning [5].

In this article, we present the Heterogeneous Image Processing Acceleration (HIPA^{cc}) framework (Figure 1): A DSL for image processing (Section 2) that captures domain-specific characteristics of algorithms and a source-to-source compiler (Section 3) that generates highly optimized and efficient target code. Using source-to-source compilation, we rely on compilers of hardware-vendors for target code generation, but we also benefit from their target-specific optimizations: We *preprocess* the DSL input and emit code for which the compiler of the hardware-vendor can generate efficient code. Our focus is parallel execution on GPUs: CUDA and OpenCL allow effi-

- *R. Membarth is with the German Research Center for Artificial Intelligence, Germany and the Computer Graphics Lab & Intel Visual Computing Institute, Saarland University, Germany.* `richard.membarth@dfki.de`
- *O. Reiche, F. Hannig, and J. Teich are with the Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany.* `{oliver.reiche,hannig,teich}@cs.fau.de`
- *M. Körner is with Siemens Healthcare Sector, Forchheim, Germany.* `mario.koerner@siemens.com`
- *W. Eckert is with Siemens Corporate Technology, Erlangen, Germany.* `wieland.eckert@siemens.com`

cient execution on standalone GPUs [6], [7] and Renderscript allows to target mobile embedded GPUs running Android [8]. Our approach allows compact algorithm descriptions (high productivity), portability between different target platforms, as well as excellent execution speed (performance) compared to state-of-the-art frameworks (Section 4).

The contributions of this article can be summarized as follows:

- We introduce a domain-specific language embedded into C++ for describing image processing algorithms.
- A source-to-source compiler that translates algorithms defined in the DSL into target-specific implementations.
- Target-specific and domain-specific mapping to the deep memory hierarchy as well as to the different types of parallelism found in today's GPUs.
- A range of algorithms and applications implemented in the proposed framework. We show that the DSL description allows a compact and portable representation while providing excellent performance.
- In detail, we evaluate our approach by assessing the performance of generated implementations and the productivity gains by using our DSL. We show that only HIPA^{cc} generates highly-optimized implementations that achieve performance near to the limits as predicted by the Roofline model [9]. Moreover, we show that HIPA^{cc} improves productivity significantly according to Halstead's productivity metrics [10], reducing development times from days to minutes.

2 Language Design

A careful analysis of the application domain is necessary and indispensable before designing a DSL. This includes analysis of the algorithmic type of operations and operators as well as the analysis of basic components characteristic to a domain. Using this information allows to design an *expressive* language that is *compact* and *orthogonal* at the same time.

For image processing algorithms, there exist a wide range of literature. Bankman [11] uses a classification based on *what information* is used to map one image to another. The three basic classes are *pixel operations* updating single pixel values, *local operators* considering also neighboring pixels, and *operations with multiple images* where several images of the same scene are used. Other classifications are based on *why* a method is applied [12]: The classes include *correcting imaging defects* due to imperfect detectors or limitations of the optics, *image enhancement in the spatial domain* to increase the visibility of one aspect, or *processing images in the frequency space* due to computational advantages. We follow the first approach and classify image processing algorithms based on *what information* contribute to the result: *point*, *local*, and *global* operators. However, compared to the classification presented in [13] and [11], it is not important how many images contribute to the operation or whether the result is again an image: Also the computation of a single value (e. g., the sum of pixels) is considered as an image operator.

2.1 Language Components

HIPA^{cc} provides a DSL embedded in C++ and uses C++ classes to describe components of the DSL. Similarly, computations on images are encapsulated in C++ classes, which inherit from base classes provided by the framework.

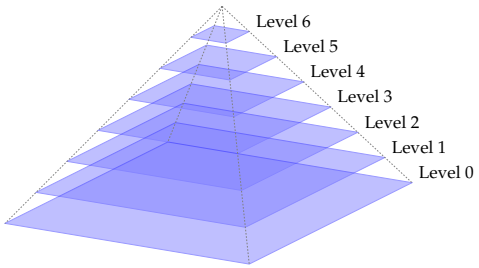


Figure 2: Image pyramid with 7 resolution levels. The most fine-grained image is located at the bottom level and the most coarse-grained image is located at the top level.

2.1.1 Images in the DSL

An *Image* in the DSL describes the data storage for the pixels of a digital image. Each pixel can be stored using the standard data types (such as `int` or `float`) as well as vector types (such as `uchar4`). Images in HIPA^{cc} are two-dimensional at the moment, but also 3D-images (volumes) might be supported in the future:

```
Image<pixel_t>(size_t width, size_t height, pixel_t *img)
```

The data layout of the image is not exposed to the programmer and can be different depending on the target platform.

Images are often processed at different resolutions so that image details can be best detected and processed. A common multiresoultion data representation in image processing are image pyramids [14]: Operating on an image pyramid includes creating different fine- and coarse-grained images each of a certain resolution, which we denote as levels as shown exemplarily for 7 levels in Figure 2.

A *Pyramid* in the DSL is defined by providing the image at the most fine-grained level and the number of levels for the image pyramid:

```
Pyramid<pixel_t>(Image<pixel_t> &img, size_t depth)
```

Again, the data layout of the image pyramid is hidden from the programmer and no care has to be taken for allocating data for the different image pyramid levels.

2.1.2 Accessing Images

We differentiate between reads and writes to an image in the DSL. The *Iteration Space* defines the pixels of an image that are computed during an operation. An iteration space is bound to an image and can be restricted by defining a rectangular Region of Interest (ROI):

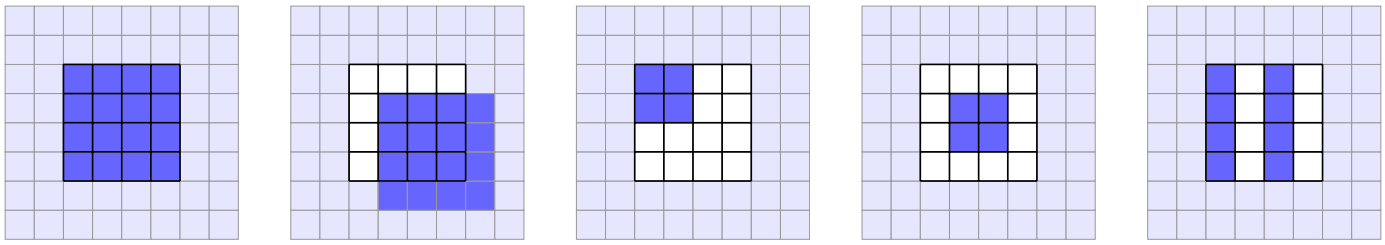
```
IterationSpace<pixel_t>(Image<pixel_t> &img,
    size_t width, size_t height,
    int offset_x, int offset_y)
```

That is, the image bound to the iteration space will be written during the operation. The parameters for the ROI are optional and can be omitted. The image can be also retrieved from an image pyramid, providing the desired level within the image pyramid:

```
Pyramid<pixel_t> &pyr(size_t level)
```

The pixels from (different) images that contribute to the computation are defined by accessors. An *Accessor* is like the iteration space bound to an image and can be restricted to a ROI:

```
Accessor<pixel_t>(Image<pixel_t> &img,
    size_t width, size_t height,
    int offset_x, int offset_y)
```



(a) Image and boundary. (b) Image offset. (c) Image crop. (d) Image crop with offset. (e) Image stride.

Figure 3: The *Accessor* defines different views on an image for image operators: (a) shows the whole image and the surrounding boundary (light blue), while in (b) an offset to the image is used. In (c) only a subregion of an image (crop) is considered and (d) shows the combination of offset and crop. (e) shows an access pattern where only every second pixel in the x-dimension is considered.

Since the computed and contributing images may be of different size, the pixel value returned when accessing a pixel is interpolated using either *nearest neighbor*, *linear filtering*, *cubic filtering*, or *Lanczos* resampling. The interpolation mode is provided as argument to the constructor of an accessor. Figure 3 visualizes the pixels that are returned by the *view* defined by different accessors: The image is extended by a virtual boundary (light-blue); dark-blue pixels are read and white pixels of the image are not considered by the accessor. Note that the stride shown in Figure 3 (e) is implicitly defined by using an input image twice as wide as the output image and using nearest neighbor as interpolation mode.

These are all DSL constructs required to describe point operators. Next, we will introduce support for boundary handling and filter masks, which are required for local operators.

2.1.3 Boundary Handling

As indicated in Figure 3, each image is extended by a virtual boundary so that images can be accessed out-of-bounds. This is, for example, the case for sliding window operators where neighboring pixels contribute to the new pixel value. For such operators the accessor should take care for boundary handling. For this purpose, the *Boundary Condition* defines the size of the region around each pixel where boundary handling is required as well as the desired boundary handling mode:

```
BoundaryCondition<pixel_t>(Image<pixel_t> &img,
    size_t size_x, size_t size_y,
    boundary_mode mode, pixel_t val)
```

The framework supports *repeat*, *clamp*, *mirror*, *constant*, and *undefined* as boundary handling modes as visualized in Figure 4. The accessor can be also defined by providing a boundary condition instance instead of an image. Note that using an undefined boundary handling mode might be desirable in case the accessor is only defined for a smaller ROI.

2.1.4 Sliding Windows

Sliding window, local operators that iterate over neighboring pixels are common in image processing. Therefore, HIPA^{cc} provides two constructs in the DSL for sliding windows: A *Domain*, which defines the iteration space of a sliding window and a *Mask*, which is a more specific version of a *Domain*, providing also filter coefficients for that window.

```
Domain(size_t size_x, size_t size_y)
Domain(uchar &domain[size_y][size_x])
Domain(Mask<pixel_t> &mask)
Mask<pixel_t>(size_t size_x, size_t size_y)
Mask<pixel_t>(pixel_t &data[size_y][size_x])
```

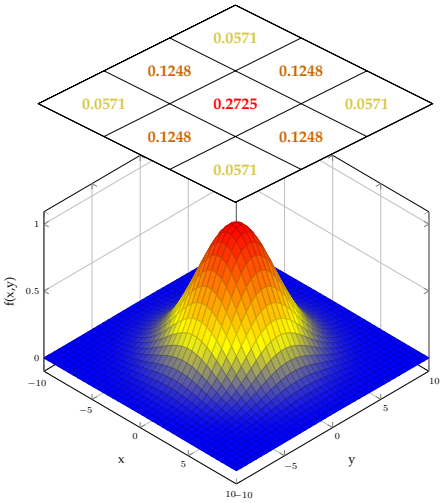


Figure 5: Two-dimensional Gaussian function. The normalized constants for a 3×3 filter mask are shown above.

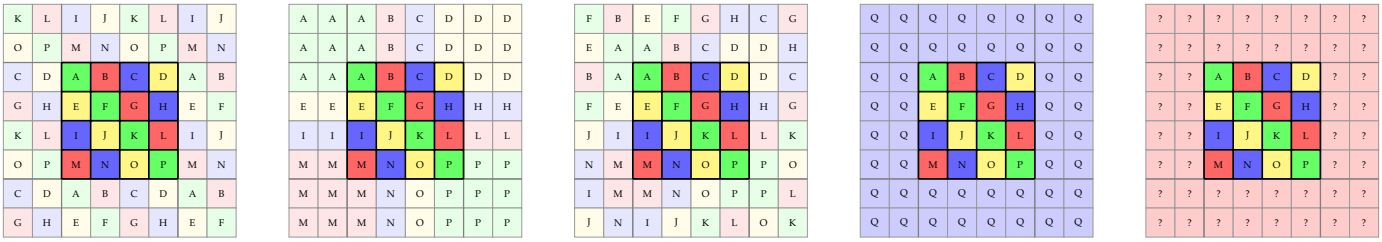
For example, Figure 5 shows the two-dimensional Gaussian function and the normalized constants for a corresponding 3×3 filter mask.

Example: Gaussian Blur Filter

Having all DSL components defined, we consider the Gaussian blur filter as an example application to show how DSL components interact. Listing 1 defines first a *Mask* `mask` for the Gaussian blur filter, reads then the input picture (`image`) from disk and assigns it to the *Image* `in`. To read from the `in` *Image*, a *Boundary Condition* is defined for the size of the *Mask* `mask` and *clamp* is used as boundary handling mode. The *Iteration Space* `iter` is defined on the result *Image* `out`. Using these DSL components, an instance of the `LinearFilter` operator, which will be introduced in the next section, is created and executed.

2.2 Defining Operators

Operators are defined similarly to operators in Threading Building Blocks (TBB) [15], where programmers implement the `operator()` and the `join()` methods of custom C++ classes. The equivalent to the `operator()` method is the `kernel()` method in HIPA^{cc}: The method specifies the computation for each pixel in the *Iteration Space*. Hence, there is no



(a) Repeat. (b) Clamp. (c) Mirror. (d) Constant. (e) Undefined.

Figure 4: Boundary handling modes for image processing. By default, the behavior is undefined when the image is accessed out of bounds (e). The framework allows to specify different boundary handling modes like repeating the image (a), clamping to the last valid pixel (b), mirroring the image at the image border (c), and returning a constant value when accessed out of bounds (d).

```
// filter mask for Gaussian blur filter
const float filter_mask[3][3] = {
  { 0.057118f, 0.124758f, 0.057118f },
  { 0.124758f, 0.272496f, 0.124758f },
  { 0.057118f, 0.124758f, 0.057118f }
};
Mask<float> mask(filter_mask);

// input image
size_t width, height;
uchar *image = read_image(&width, &height, "input.pgm");
Image<uchar> in(width, height, image);

// reading from in with clamping as boundary condition
BoundaryCondition<uchar> cond(in, mask, Boundary::CLAMP);
Accessor<uchar> acc(cond);

// output image
Image<uchar> out(width, height);
IterationSpace<uchar> iter(out);

// instantiate and launch the Gaussian blur filter
LinearFilter Gaussian(iter, acc, mask, 3);
Gaussian.execute();
```

Listing 1: Instantiation of an operator for the Gaussian blur filter in the DSL.

```
class LinearFilter : public Kernel<uchar> {
  private:
    Accessor<uchar> &input;
    Mask<float> &mask;
    size_t size;

  public:
    LinearFilter(IterationSpace<uchar> &iter, Accessor<
        uchar> &input, Mask<float> &mask, size_t size) :
      Kernel(iter), input(input), mask(mask), size(size)
    { add_accessor(&input); }

    void kernel() {
      float sum = 0;
      int range = size/2;

      for (int yf = -range; yf <= range; ++yf)
        for (int xf = -range; xf <= range; ++xf)
          sum += mask(xf, yf) * input(xf, yf);

      output() = (uchar) sum;
    }
};
```

Listing 2: Kernel description for the Gaussian blur filter.

difference between point and local operators in HIPA^{cc}. Both are realized by the `kernel()` method. Instead of providing a generic `join()` method, HIPA^{cc} allows currently only to define global reductions using the `reduce()` method.

2.2.1 Local and Point Operators

The `kernel()` method is embedded into a user-defined class that derives from the `Kernel` base class provided by the framework. Listing 2 shows such a class implementing linear filters that retrieve the precomputed constants from the *Mask* `mask`. The provided implementation iterates manually over the mask elements, accessing neighboring pixels using relative offsets. The result is stored using the `output()` method.

For describing sliding window operators, the framework provides built-in functions that allow a more concise description: The `convolve()` method taking a) the filter mask, b) the aggregation mode, and c) the computation instructions for a single filter mask component with the corresponding image pixel described as a C++ lambda function:

```
void kernel() {
  output() = (uchar)
      convolve(mask, Reduce::SUM, [&] () -> float {
        return mask() * input(mask);
      });
}
```

HIPA^{cc} currently supports *min*, *max*, *sum*, and *prod* as aggregation mode.

For more complex algorithms that do not follow the above scheme, the `iterate` function can be used. Consider the bilateral filter that combines a closeness component `c` and a similarity component `s` over a common sliding window [16]. Using `iterate()`, we can iterate over the sliding window defined by the *Domain* `dom` and compute both components as well as the normalization factor:

```
void kernel() {
  float d = 0, p = 0;

  iterate(dom, [&] () -> void {
    float diff = in(dom) - in();
    float c = mask(dom);
    float s = expf(-c_r * diff*diff);
    d += c*s;
    p += c*s * in(dom);
  });

  output() = p/d;
}
```

The aggregation is explicit in this case and the closeness component is read from the *Mask* `mask`. The similarity could be also read from a *Mask*, but is computed on-the-fly in this case.

2.2.2 Global Reduction Operators

For global reduction operators, the `reduce()` method can be implemented by the programmer. We use the definition provided by Blelloch [17] for the reduce operation, but do not require the identity: The operator defined by the `reduce()` method is applied to all elements defined by the *Iteration Space* of the user-defined *Kernel* class. For instance, the minimum of all pixels can be described as follows:

```
int reduce(int left, int right) {
    return min(left, right);
}
```

The result of the reduction can be retrieved using the `reduced_data()` method of the *Kernel* class.

2.3 Image Pyramid Construction

During traversal of the pyramid, either a more fine-grained or a more coarse-grained representation is created by upscaling or downscaling the image. The actual operators are then applied to the images at each level. To construct the images at different pyramid levels, the DSL provides the `traverse()` function:

```
void traverse(std::vector<Pyramid>,
              const std::function<void()>)
```

It takes a vector of image pyramids as argument and a C++ lambda function that describes the traversal of the pyramids. Pyramids are registered in the `traverse()` function and images of each pyramid can be requested using a relative index. That is, querying a pyramid at index '0' will return an image handle for the current traversal level. The index '1' refers to the image on the next (more coarse-grained) level, whereas index '−1' refers to the image of the previous (more fine-grained) level, respectively.

The traversal of the pyramid in the C++ lambda function is triggered by a *pseudo-recursive* call to `traverse()`: Listing 3 shows a simple example for operating on pyramid data structures using the presented traversal functions. As needed for the traversal, two image pyramids are created (lines 3–4) with a depth of 7. Both are bound to a `traverse` function call (line 6), together with the lambda function describing the actual recursion body (lines 7–26). The body contains the set-up and execution of three kernels: `downscale` (lines 8–11), `compute` (lines 14–17), and `upscale` (lines 22–25). The recursive call (line 19) occurs right after the compute kernel but could also be placed before without changing the schedule.

2.4 C++ Integration

Since the presented DSL is realized through C++ classes, programs can be compiled with any C++ compiler such that incremental porting of applications is possible. However, compiled with the source-to-source compiler provided by HIPA^{cc}, target code for GPU accelerators is generated as discussed in the next section. The programmer can combine the DSL with regular C++ as shown in the following example, using OpenCV for I/O and HIPA^{cc} for computing:

```
// load image in OpenCV
Mat frame = imread("lena.pgm", CV_LOAD_IMAGE_GRAYSCALE);

// use image in hipacc
Image<uchar> img(frame.cols, frame.rows);
img = frame.data;
...

// use OpenCV to display image
frame.data = img.data();
imshow("Result Image", frame);
```

```
1  Image<uchar> in;
2  Image<uchar> out;
3  Pyramid<uchar> pin(in, 7);
4  Pyramid<uchar> pout(out, 7);
5
6  traverse({ &pin, &pout }, [&] {
7    if (!pin.is_top_level()) {
8      Accessor<uchar> acc_in(pin(-1), Interpolate::LF);
9      IterationSpace<uchar> iter_in(pin(0));
10     Downscale ds(iter_in, acc_in);
11     ds.execute();
12   }
13
14   Accessor<uchar> acc_in(pin(0));
15   IterationSpace<uchar> iter_out(pout(0));
16   Compute c(iter_out, acc_in);
17   c.execute();
18
19   traverse();
20
21   if (!pout.is_bottom_level()) {
22     Accessor<uchar> acc_out_cg(pout(1), Interpolate::LF);
23     Accessor<uchar> acc_out(pout(0));
24     Upscale us(iter_out, acc_out, acc_out_cg);
25     us.execute();
26   }
27 });
```

Listing 3: Pyramid traversal: The downscale/upscale kernels create a coarser/finer representation and the compute kernel processes the data at each level.

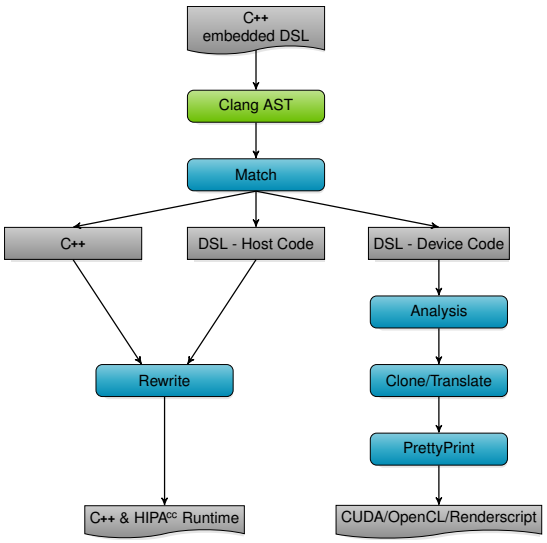


Figure 6: DSL program compilation work flow of HIPA^{cc}.

3 Compiler Framework

Instead of implementing a new front end for C++, we decided to build our source-to-source compiler on top of Clang[1], a C language family front end for LLVM. Clang performs the typical steps of a compiler front end: Syntactic analysis (lexing and parsing), semantic analysis, and the construction of an Intermediate Representation (IR) [18], [19]. Clang uses an Abstract Syntax Tree (AST) as IR that stores further information of the input source program for each AST node.

3.1 Compiler Work Flow

The source-to-source compiler of HIPA^{cc} traverses the AST build by Clang in order to generate target code as shown in

1. http://clang.llvm.org

Figure 6. First, all AST nodes are visited using a preorder depth-first traversal in the **Match** library. AST nodes that corresponds to a) declarations and definitions of DSL classes, b) statements that define objects in the DSL, and c) expressions involving DSL objects are stored and further operations are triggered depending on their type: a) class definitions are removed from the textual source program representation using the **Rewrite** library, but the AST representation is still available to generate target code for the kernel; b) statements defining DSL objects are replaced by corresponding calls to the HIPA^{cc} runtime (e. g., to allocate an image object on the GPU); and c) expressions on DSL objects are also replaced by calls to the HIPA^{cc} runtime (e. g., to copy data to/from the memory allocated previously on the GPU). Depending on the target language, different runtime Application Programming Interface (API) calls are emitted by the **Rewrite** library and the updated textual representation is stored to a file. Only parts of the input source file that make use of the DSL are changed. This helps the user to understand the applied changes and allows him to further modify the generated source files.

The **Analysis** library extracts metadata from DSL classes and image operators. This includes the size of filter masks as well as boundary handling modes, information that is required for target code generation. However, most important is the analysis and categorization of kernels: Memory accesses and arithmetic operations are analyzed for the user-provided `kernel()` method. Therefore, a Control Flow Graph (CFG) of the method is constructed and visited in postorder depth-first traversal in order to collect the following information:

(a) the *number of instructions*. The instructions are classified into memory accesses and arithmetic instructions.
(b) *memory access* properties of images. Each memory access is analyzed and for each image it is stored if it is only read, written, or both read and written. In case of reading and writing an image within the same kernel, a warning is emitted since this may lead to inconsistent memory views.
(c) *memory access pattern* of images and kernels. In addition to the read/write analysis, also the pattern of memory accesses are analyzed. It is differentiated between accesses with stride-x, stride-y, stride-xy, and without any stride. This information is collected for each image, and the aggregated stride information of all images is annotated to the kernel. This allows to categorize kernels as point operators (no stride) and local operators (stride-x, stride-y, stride-xy). Row filters correspond then to stride-x local operators and column filters to stride-y local operators.
(d) *use-def chain*. Each definition of a variable and its uses is stored. This liveness analysis can be used to remove dead code, but is extended to include *divergence* information. This information is essential for vectorization [20].

Once the metadata has been extracted, the AST of the `kernel()` method is converted for execution on the target hardware. Since Clang does not allow to modify the AST representation—the AST is immutable—the whole AST representation has to be recreated in order to add or modify AST nodes. The **Clone** library provides the facility to duplicate single AST nodes and will issue an error message if C++ operators are used that are not supported on the target hardware such as for `try`, `catch`, and `throw` as well as for instructions not supported by HIPA^{cc} such as `new` and `delete`.

The **Translate** library replaces AST nodes without changing the semantics of the operation they describe. For example, accesses to an image object in the DSL are replaced by accesses to GPU memory. However, this is not limited to simple one to one translations, but also more sophisticated transformations are supported such as loop unrolling or staging of data to scratchpad memory. During AST traversal, we keep track of kernel class member variable uses. Only member variables that are present in the generated code will be added as parameter to the compute kernel. For example, *Domains* need only to be added as parameter in case the iteration space defined by the *Domain* is not unrolled.

Having the AST transformed for the target hardware, the **PrettyPrint** library stores the AST to a file using the syntax of the target language. During pretty printing, function and variable qualifiers are emitted. For example, the `__kernel` qualifier is emitted in OpenCL for kernel entry functions and the `__local` address space qualifier for scratchpad memory. Similarly, CUDA requires attributes for kernel entry functions (`__attribute__ ((global))`) and scratchpad memory (`__attribute__ ((shared))`).

3.2 Kernel Code Generation

The source-to-source compiler transforms the DSL description in target-specific source code optimized with respect to execution speed. Therefore, the metadata extracted from the DSL description as well as the information obtained through code analysis is used.

- *Memory padding*: The data storage for images are allocated by the runtime system such that each image line is aligned to a multiple of the memory transaction size on the target architecture. The generated memory access index computation takes the introduced padding into account. This preserves kernel performance for images of arbitrary size.
- *Memory layout*: Image data can be stored using different memory layouts. This includes linearized one-dimensional memory, two-dimensional memory, as well as hardware-vendor specific memory layouts using space-filling curves. Therefore, the runtime can allocate memory using different memory layouts and the required API calls to read/write to the selected data structure are emitted during code generation. This includes image objects for OpenCL or texture and surface memory for CUDA.
- *Memory hierarchy*: GPU accelerators feature deep memory hierarchies that have to be utilized in order to overcome the long memory latency and improve locality. Depending on the memory access pattern, different approaches are followed in order to improve locality. We distinguish between the following scenarios: a) in case a kernel reads only a single pixel of a given image, locality cannot be improved and no special memory mapping is generated; b) in case a kernel reads multiple surrounding pixels, locality may be improved by using a special memory layout as described above and by staging data to fast scratchpad memory; c) in case a kernel reads the same data for each pixel and that data is of known size, *constant* memory can be utilized to provide a cache that broadcasts the data to multiple threads. Constant memory is predestined for *Masks* (*Domains*) and is used whenever the compiler is not able to resolve all computations involving a *Mask* at compile time. Scratchpad memory and special memory layouts are selected depending on the target platform. Both can be combined as shown in the following for CUDA, reading data from a texture reference bound to linear 1D memory and staging it to shared memory:

```
// texture declaration for input
texture<float, cudaTextureType1D,
                cudaReadModeElementType> tex_in;
// shared memory declaration for input
__shared__ float in_sm[8][33];
// load data to scratchpad
in_sm[threadIdx.y][threadIdx.x] =
    tex1Dfetch(tex_in, gid_x * in_stride + gid_y);
__syncthreads();
// read data from scratchpad
... in_sm[threadIdx.y-1][threadIdx.x+1] ...
```

The example above is generated for a work-group configuration of 32×4 and a filter mask size of 1×5. The allocated scratchpad for the x-dimension are $1 \cdot 32 + 1 = 33$ pixels, adding one element padding to avoid bank conflicts in shared memory. Considering the given work-group configuration in the y-dimension, $4 + (5 - 1) = 8$ lines of scratchpad are required.

- *Constant propagation & loop unrolling*: The local window described by *Domains* and *Masks* can be fully unrolled knowing the filter window size. This removes the looping overhead in the compute kernel and the requirement to pass a *Domain* or *Mask* to the compute kernel as parameter. In case a *Mask* is compile-time constant, the constants of the mask can be propagated in addition to unrolling the iteration space. This can only be done if the access to the *Mask* is also known at compile time.
- *Thread coarsening*: Merging the computation of multiple iteration points into a single thread can improve locality and reuse of data. At the same time, it reduces the launch overhead for lightweight GPU threads. This transformation is equivalent to loop unrolling of the global iteration space, reducing the number of threads running on the GPU. GPU accelerators have strict requirements on memory access patterns in order to utilize bandwidth best: contiguous memory should be accessed by threads executed in lockstep for memory coalescing. For this reason, we apply thread coarsening only to threads of the global iteration space in the y-dimension:

```
gid_x = blockDim.x * blockIdx.x + threadIdx.x;
gid_y = blockDim.y * blockIdx.y * N + threadIdx.y;

// first iteration
{
  // begin kernel code
  ... = in[gid_y * in_stride + gid_x];
  // end kernel code
}

// second iteration
if (gid_y + 1 * blockDim.y < is_height) {
  // begin kernel code
  ... = in[(gid_y+1) * in_stride + gid_x];
  // end kernel code
}

// n-th iteration
if (gid_y + (N-1) * blockDim.y < is_height) {
  // begin kernel code
  ... = in[(gid_y+N-1) * in_stride + gid_x];
  // end kernel code
}
```

The example above unrolls the global iteration space by a factor of `N` with *Image* accesses being updated for the current iteration point. Since we do not know if the image size is a multiple of `N` at compile time, we also need to add guards that ensure that the current iteration point belongs to the global iteration space. Setting `N` to 4 requires now $4 \cdot \texttt{N} + (5 - 1) = 20$ lines of scratchpad in the previous example.

- *Multiple Program, Multiple Data (MPMD) code generation*: To support boundary handling, checks have to be added for reads to an *Image*. Checking boundary conditions for each memory access is expensive and leads to increased execution times in the magnitude of 10 % up to 100 % depending on the boundary handling mode as well as on the target hardware. One solution to this problem is the use of texturing hardware that takes care of boundary handling. However, texturing hardware is not available on all platforms and allows only boundary handling on two-dimensional arrays. This imposes a memory layout that is not optimal for achieving best performance and does not allow to define boundary handling on a ROI (cf. [6], [7]). Therefore, we divide the image in different regions according to the image borders and generate specialized variants for boundary handling for each region as indicated below and visualized in Figure 7:

```
__global__ void kernel(... int bh_t, int bh_fb) {
  if (bh_fb) goto BH_FB;
  if (blockIdx.x < bh_l &&
      blockIdx.y < bh_t) goto BH_TL;
  ... // checks for other variants
  goto BH_NO;

BH_FB:
    // handle all image borders
    <pretty printed AST>
    return;
BH_TL:
    // handle only top left border
    <pretty printed AST>
    return;
... // other variants
BH_NO:
    // no boundary handling
    <pretty printed AST>
}
```

This results in one *big* kernel that includes all specialized variants. This is similar to index set splitting [21], but the code variant is selected at run time depending on the block index. All threads executed on the same compute unit execute always the same code variant, and therefore, in lockstep. Now, only checks are issued at thread blocks processing pixels at image borders and no checks are generated for the inner image region. We choose to generate one big kernel instead of multiple kernels since the overhead of scheduling multiple small kernels for the image borders is higher than the benefit of specialized boundary handling. In total we generate up to ten variants: Up to eight for the image borders, one for the inner region, and one fall-back variant for very small images (e. g., as required for *Pyramids*). The memory access pattern extracted during code analysis defines the code variants required: Only left and right borders have to be handled for row kernels and only top and bottom borders for column kernels.

3.3 Optimization Strategy

HIPA^{cc} follows a model-driven approach for code optimization utilizing the metadata captured by the DSL constructs and information from an architecture model. Tailored implementations are generated using the fused information about the input algorithm and the target architecture. The architecture model describes which optimizations are valid and beneficial for a given target architecture:

- The memory layout and alignment for images.
- Texture memory layout and usage.
- Thresholds for staging data to scratchpad memory.

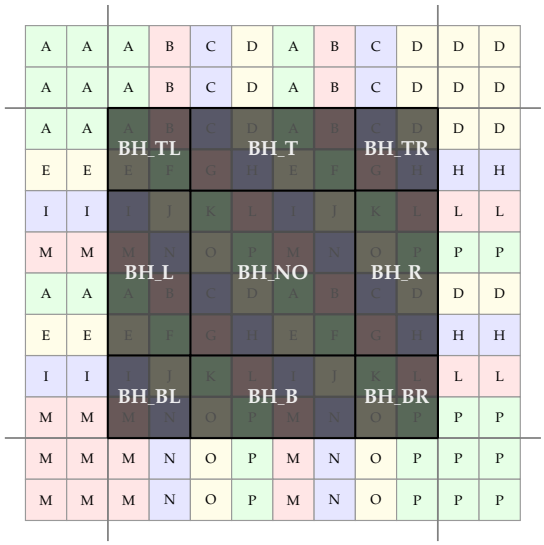


Figure 7: Code variant assignment to image border: Boundary handling for local operators is limited to those regions where out of bounds image accesses occur. Different code variants are generated for the top left (BH_TL), top (BH_T), etc. image borders.

- The thread coarsening factor.

The launch configuration of a kernel determines the tiling of the global iteration space and the occupancy of hardware resources. HIPA^{cc} extracts resource usage (such as number of registers and shared memory usage) of generated kernels to fine-tune their tiling to achieve good occupancy and improve locality: The compiler calculates for all valid configuration sizes (S) (due to shared memory/register usage) that result in good memory bandwidth utilization (due to coalescing) the occupancy. Then the heuristic of Algorithm 1 selects a configuration $c_{s,t_{xy}}$ from the list of configuration sizes S. For each configuration size s, the shape of the configuration is determined such that the resulting tiling t_{xy} of the image into sub-problems processed on a compute unit adapts to the available resources and inherent locality of the considered kernel.

Depending on the operator kind, a different strategy is used. For point and global operators, a 1D-tiling is chosen since the resulting code requires less instructions (e. g., no guards for the y-dimension, simpler index address calculation) and point and global operators do not benefit from a 2D-tiling (there is no temporal locality). The size of the tiling is selected such that a high occupancy is given. In case multiple configurations result in the same occupancy, the configuration with the lowest number of threads is chosen. This gives higher flexibility to the hardware scheduler since multiple work-groups can be scheduled and executed at the same time on a single compute unit. This results in 1D-configurations like 128×1 or 256×1. Such configurations are typically selected by expert programmers and yield good performance for most kernels. In contrast, a 2D-tiling is chosen for local operators. This has two reasons: To minimize the number of threads that execute code with conditionals for boundary handling and to maximize locality by selecting a 2D-shape. The minimal size for the x-dimension (the warp size) is typically enough to cover boundary handling in the x-dimension with a single work-group (e. g., for a kernel window size of 3×1 up to 65×1 for a warp size of 32). Instead, the y-dimension

Algorithm 1: Heuristic for selecting kernel configuration and tiling depending on resource usage, filter mask sizes, and target graphics card.

Input: Kernel K, list of configuration sizes S for GPU G
Output: Configuration $c_{s,t_{xy}}$ with size s and tiling t_{xy} for kernel K

$S \leftarrow$ configuration sizes of S multiple of warp size of G and within resource limitations of G
$S \leftarrow$ sorted configuration sizes of S with descending occupancy and ascending number of threads
$s \leftarrow$ first configuration size of S
if *local operator* **then**
  $t_{xy} \leftarrow$ tiling of s, prefer y over x
  $threads\_bh \leftarrow$ calculate number of threads for border handling for kernel K with tiling t_{xy}
  $S' \leftarrow$ configuration sizes with occupancy within $10\,\%$ of s
  foreach *configuration size* s' *of* S' **do**
    $t'_{xy} \leftarrow$ tiling of s', prefer y over x
    $threads\_bh' \leftarrow$ calculate number of threads for border handling for kernel K with tiling t'_{xy}
    if $threads\_bh' < threads\_bh$ **then**
      $s \leftarrow s'$
      $t_{xy} \leftarrow t'_{xy}$
      $threads\_bh \leftarrow threads\_bh'$
    end
  end
else
  $t_{xy} \leftarrow$ tiling of s, prefer x over y
end
$c_{s,t_{xy}} \leftarrow s, t_{xy}$

is increased and not only the first configuration size with the highest occupancy is considered. All configuration sizes that are within a given threshold, for instance 10% of the configuration size with the highest occupancy are considered: The configuration is chosen such that the number of threads with boundary handling conditionals is minimized (e. g., a configuration of 32×6 is preferred over 32×4 for a kernel window size of 13×13; however, a configuration of 32×3 would be preferred to the two aforementioned configurations). Once a configuration size and tiling for a kernel have been determined according to the presented heuristic, the source-to-source compiler can emit the kernel configuration for invocations of the kernel.

3.3.1 Configuration Space Exploration

The rules of the architecture model decide what code variant is generated. However, the compiler allows also to set transformations manually using compiler switches. For example, the user can specify that scratchpad memory or a certain kind of texture memory should be turned on or off. Similarly, the amount of padding or the thread coarsening factor can be set by the user. This allows for an easy exploration of features for future architectures.

In addition, the source-to-source compiler can generate code to explore kernel tiling parameters using the `--explore-config` compiler switch. The generated code includes macro-based code snippets for features that depend on tiling (statically allocated scratchpad memory and the work-group size). During execution, the runtime system compiles those variants using Just-In-Time (JIT) compilation. The influence of the kernel configuration (size and shape) on execution time is shown in Figure 8 for the bilateral filter.

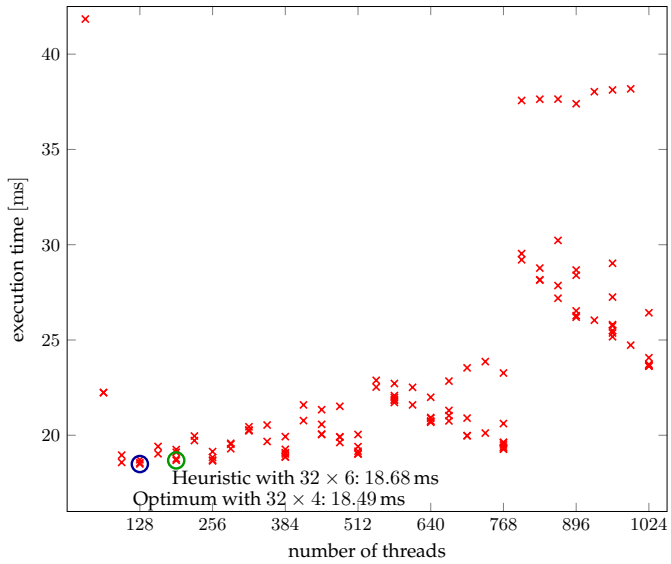


Figure 8: Configuration space exploration for the bilateral filter (filter window size: 13×13) for an image of size 4096×4096 on the Tesla K20.

Only configurations that are a multiple of the warp size are considered. In the example, execution times vary between 18.49 ms for a configuration of 32×4 and 41.84 ms for a configuration of 32×1. The proposed heuristic (Algorithm 1) selected the configuration of 32×6, which results in an execution time of 18.68 ms. The insights from explorations can be used to set an optimized configuration by hand or to fine-tune the heuristic.

4 Results

In this section, we evaluate important aspects of domain-specific languages for image processing. In particular, we will evaluate the performance of the generated implementations, the portability of the description itself, and the gains in productivity using the proposed DSL. In the following, we first investigate a single, representative sliding window algorithm in detail, before we show how bigger applications can be described in the DSL.

4.1 Comparison against RapidMind, Halide, OpenCV, and NPP

We picked the Gaussian blur filter as representative sliding window algorithm since implementations exist in highly-optimized domain-specific libraries such as Open Source Computer Vision (OpenCV) and NVIDIA Performance Primitives (NPP). At the same time, implementations in other domain-specific language such as RapidMind or Halide are available.

RapidMind [22] is a multi-core development platform that allows to describe algorithms as computations on arrays similar to the proposed framework. Actually, the kernel description (embedded into C++) differs only by the used keywords and data types. GPUs from NVIDIA are supported by the CUDA back end.

Halide [2] is a domain-specific language embedded into C++ for image processing and supports GPU accelerators through a CUDA and OpenCL back end. Algorithms are expressed using functional expressions and the target-specific optimization (the schedule) is defined separately by the programmer.

The **OpenCV** library is a widely used library for image processing and provides a CUDA and OpenCL module to target GPU accelerators. Hand-tuned implementations are provided for each module, which are further optimized depending on the target GPU.

The **NPP** library is part of CUDA and provides optimized primitives for imaging and video processing on NVIDIA GPUs. In contrast to OpenCV, NPP supports only processing of 8-bit gray-scale images for the considered primitives. There is also no boundary handling support in NPP. The valid image region shrinks after each kernel invocation.

4.1.1 Results

Table 1 and Table 2 show the execution times of the automatically generated implementations for the Gaussian blur filter on an image of 4096×4096 pixels on a Tesla K20 and Radeon R9 290X. The optimizations applied by the HIPA^{cc} framework include constant propagation and unrolling of the convolve method, using texture memory when reading from global memory (CUDA), staging the data to scratchpad memory as well as unrolling of the global iteration space. The configuration is determined by the HIPA^{cc} framework and six specialized variants for boundary handling are generated. The tables list also a naïve implementation generated by HIPA^{cc} where only global and constant memory are used with a fixed configuration of 128×1.

The results show that our generated implementations are on both GPUs faster than low-level hand-written implementations (OpenCV and NPP) as well as the implementations generated by RapidMind and Halide. The RapidMind implementation crashed when *Repeat* is used as boundary handling mode and is otherwise slower by a factor of three. For RapidMind, we recompute the filter mask, which is faster than using precomputed values. For Halide, we explored different optimizations (schedules). It turned out that a separated schedule yields the fastest implementation on both GPUs. Halide provides also a *RDom* object that can be used the same way as a *Domain* in HIPA^{cc}. Describing the convolution using *RDom* results in higher execution times (11.62 ms vs. 10.95 ms) for the K20 and is significantly faster on the R9 290X (0.82 ms vs. 1.10 ms). Since both Halide and RapidMind use a JIT compiler, NVIDIA's *nvprof* profiler is used to measure kernel timings.

The OpenCV implementation has a competitive performance across different boundary handling modes and is faster compared to the naïve implementation. OpenCV is even faster compared to NPP, although NPP does not implement boundary handling and requires only to transfer one fourth of the data (using `uchar` instead of `float` for the intermediate result).

Table 3 shows the execution times on the Intel Xeon Phi 7120P with similar results as for the AMD and NVIDIA cards: Our generated code is significantly faster compared to the implementation in OpenCV ($3\times$) and more than 10 % faster compared to Halide.

For the ARM Mali T-604 running Android, we generate specialized Renderscript implementations. Table 4 shows that we get a speedup of roughly 40 % compared to a naïve implementation in Renderscript. While there is no other framework targeting Renderscript at the moment, we have shown previously that our generated Renderscript implementations outperforms a) highly optimized implementations

Table 1: Execution times in ms for the **Gaussian blur filter** on a **Tesla K20** using the **CUDA** back end for an image of 4096×4096 pixels and a filter window size of 5×5.

	Undef.	Clamp	Repeat	Mirror	Const.
naïve	crash	3.04	3.14	3.15	3.19
RapidMind	5.40	6.00	crash	n/a	5.97
Halide	n/a	4.17	n/a	n/a	n/a
OpenCV	n/a	2.12	2.15	2.22	2.01
NPP†	2.40	n/a	n/a	n/a	n/a
HIPA^{cc}	1.32	1.38	1.40	1.38	1.39

† NPP's implementation works on 8-bit data only; no support for boundary handling; output image dimensions are reduced by the filter window size.

Table 2: Execution times in ms for the **Gaussian blur filter** on a **Radeon R9 290X** using the **OpenCL** back end for an image of 4096×4096 pixels and a filter window size of 5×5.

	Undef.	Clamp	Repeat	Mirror	Const.
naïve	1.19	1.18	1.19	1.18	1.20
Halide†	n/a	0.82	n/a	n/a	n/a
OpenCV	n/a	0.89	1.43	0.90	0.91
HIPA^{cc}	0.67	0.64	0.68	0.67	0.66

† Using RDom for the convolution, normal schedule requires 1.10 ms.

from the hardware-vendor of the Mali T-604 when executed on the embedded GPU; as well as b) the hand-tuned and vectorized CPU implementations provided in OpenCV when executed on the ARM CPU (cf. [8]).

4.1.2 Performance Model

Since it is known that stencil codes are usually bandwidth limited, we use a simple performance model to estimate the performance of the generated code. For a separated Gaussian blur filter, we load pixels of type `uchar` and store `float` values in the first component. The second component loads `float` values and stores pixels back as `uchar`. If we assume that all neighboring memory accesses for the convolution are in cache, we have to transfer $4 \cdot 2 + 1 \cdot 2 = 10$ bytes per pixel. On the Tesla K20 with achievable memory bandwidth of $b = 140$ GB/s and for a problem size $N = 4096 \times 4096$ we thus estimate $\frac{N \cdot 10}{b} \cdot 1000 \approx 1.20$ ms for the Gaussian blur. This corresponds to the maximal performance the Gaussian blur filter can achieve due to the peak *stream* memory bandwidth in the Roofline model [9]. We use OpenCL's `clEnqueueCopyBuffer()` and Renderscript's `copy1DRangeFrom()` to measure this achievable *memcpy* bandwidth, which we consider as an upper bound.

Table 3: Execution times in ms for the **Gaussian blur filter** on a **Xeon Phi 7120P** using the **OpenCL** back end for an image of 4096×4096 pixels and a filter window size of 5×5.

	Undef.	Clamp	Repeat	Mirror	Const.
naïve	crash	6.42	5.96	6.34	6.39
Halide	n/a	3.94	n/a	n/a	n/a
OpenCV	n/a	10.96	14.06	11.13	9.75
HIPA^{cc}	crash	3.57	3.78	3.70	3.87

Table 4: Execution times in ms for the **Gaussian blur filter** on a **Mali T-604** using the **Renderscript** back end for an image of 4096×4096 pixels and a filter window size of 5×5.

	Undef.	Clamp	Repeat	Mirror	Const.
naïve	crash	342.87	348.34	338.58	348.36
HIPA^{cc}	crash	213.38	214.28	215.71	219.65

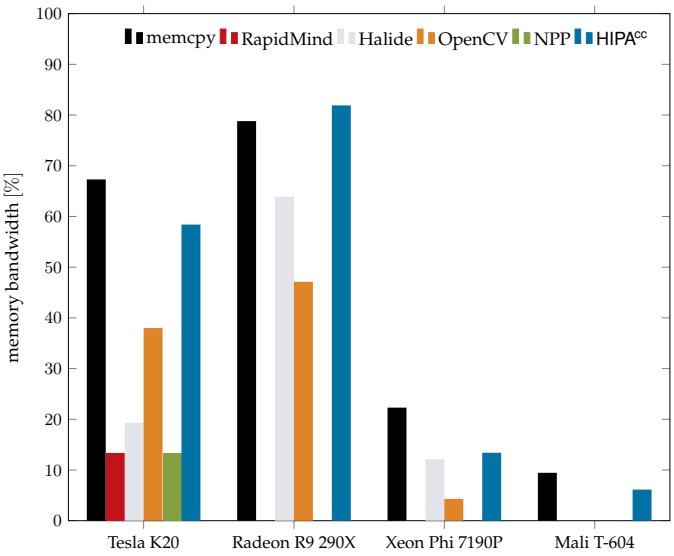


Figure 9: Memory bandwidth of the Gaussian blur filter implementations in different frameworks. The *memcpy* bandwidth is given as reference.

Figure 9 shows the achieved memory performance of the Gaussian blur filter implementations in the different frameworks[2]. It can be seen that the memory bandwidth utilization is highest when using HIPA^{cc}. For NVIDIA hardware, we achieve almost the *memcpy* bandwidth and exceed it even on the AMD card. Efficient implementations for the current Xeon Phi coprocessor require prefetching which is not exposed in OpenCL. This results in a bandwidth utilization that is far below the theoretical peak memory bandwidth. Similarly, the bandwidth utilization we see for the ARM Mali is far below the theoretical memory bandwidth. Here, we consider the dual port memory bandwidth as peak, but the system might only use a singe port of the memory. However, all other implementations we investigated suffer from the same low bandwidth utilization.

4.1.3 Programming Effort

We consider the productivity metrics introduced by Halstead [10] in order to quantify the programming effort using HIPA^{cc}. Halstead's metrics are based on two code features: *operators* and *operands*. Operands denote the variables and constants of a program on which operators act. If we count the number of distinct operators η_1 and operands η_2 as well

2. Note that the NPP implementation transfers only 4 bytes per pixel and the OpenCL implementation in OpenCV only 8 bytes per pixel.

Table 5: Halstead's productivity measures for the Gaussian blur filter in HIPA^{cc}, OpenCV (CUDA), and the generated CUDA implementation. Shown are the number of unique/total operators (η_1/N_1) and operands (η_2/N_2) as well as the resulting volume V and effort E for the implementation.

		η_1	η_2	N_1	N_2	V	E	Lines of Code (LoC)
HIPA^{cc}	col	15	3	23	5	261.70	3082.87	10
	row	16	5	26	7			
OpenCV	col	42	49	357	230	7447.45	713047.41	185
	row	40	47	345	218			
generated	col	32	78	2951	1509	74208.40	24701839.63	1190
	row	32	104	3935	2268			

as the total numbers of operators N_1 and operands N_2 within an algorithm implementation, we can compute the volume V, difficulty D, and effort E for the given implementation:

$$\text{Volume} \qquad V = (N_1 + N_2) \times \log_2(\eta_1 + \eta_2)$$

$$\text{Difficulty} \qquad D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$$

$$\text{Effort} \qquad E = D \times V$$

We use and extend the implementation from [23] to recognize also the CUDA language in order to compute Halstead's productivity metrics. Table 5 shows the productivity metrics as well as the LoC for the Gaussian blur filter description in HIPA^{cc}, the corresponding CUDA implementation in OpenCV, as well as for the generated CUDA implementation, specialized for the Tesla K20. We describe the row and column component in HIPA^{cc} separately in order to round the second component the same way as in OpenCV in order to get consistent results. Apart from rounding, the code of both components is identical and a single description would be sufficient to generate the same specialized implementations. In contrast, two separate and completely different implementations are used in OpenCV. It can be seen that the description in HIPA^{cc} has by far the smallest volume and effort. The volume of the generated code is $10\times$ the volume of the hand-tuned CUDA implementation in OpenCV, but has an $\sim 35\times$ higher effort to implement.

We can estimate the time T required for the implementation if we divide the effort E by the *Stroud number* S:

$$\text{Time} \qquad T = \frac{E}{S} \text{ seconds}$$

Using 18 for S for fluent, concentrating programmers[3] [25] we estimate the time T as follows: 3 minutes for the description in HIPA^{cc}, 11 hours for the CUDA implementation in OpenCV, and 15.8 days for the generated CUDA implementation. Note that this takes into account neither that parallel programming increases programming complexity nor the fact that the generated CUDA implementations compute 4 pixels per thread ($4\times$ the LoC).

4.2 Applications

In addition to the Gaussian blur filter, we consider typical image processing algorithms for feature detection: a Lapacian edge detector, a multiresolution filter, a Harris corner detector [26], and the computation of optical flow using the census transform [27]. Those algorithms form a realistic scenario and reflect possible cost-sensitive implementations in commercial products, such as augmented reality or driver assistance systems. All of them are based on local and point operators or a combination of both but differ greatly in implementation detail.

Figure 10 shows the number of operators and operator invocations for these algorithms. It shows also the LoC required to describe them in HIPA^{cc} as well as the performance of the generated implementations. In the following, we summarize which operators are required for each algorithm and highlight features of the DSL to realize them:

- The Laplacian operator is a sliding window operator similar to the Gaussian blur filter. However, the filter mask of size 5×5 cannot be separated and is realized as a single operator.
- The multiresolution filter creates first a pyramid representation of the input image of depth 6. Then, it applies the bilateral filter [16] on each image of the pyramid. Finally, it reconstructs the image at the most fine-grained level. For the image pyramid traversal, we use the `traverse()` function working on pyramids for the input image, output image, and an image for intermediate results.
- The Harris corner detector embodies a combination of point and local operators that form a complex image pipeline. In total, twelve operator invocations are required to detect the edges in the input image.
- The optical flow includes the computation of a signature for each pixel of the smoothed input images. The signatures of two successive images are used to compute the optical flow: They are compared within a sliding window of size 15×15 using the `iterate()` function over a *Domain* that excludes the center.

5 Related Work

Abstractions are a compelling way to hide low-level details. High-level programming languages provide abstractions from storage locations or calling conventions of a processor by variables and function calls. However, these abstractions are specific to the target processor and offer no means to exploit optimizations specific to a given application domain or parallelism within algorithm descriptions. As a consequence, tools and languages have been developed to extract or capture the required knowledge. For example, the polyhedron model [28] relies on code analysis to extract information required for parallelization. The need for abstraction of domain properties resulted in several languages and frameworks for a variety of domains including image processing. A prominent example for DSLs is Delite [29], [30]: A common compiler and runtime infrastructure for building new performance-oriented DSLs. It provides facilities for defining and embedding a

3. Stroud [24] suggests $5 \leq S \leq 20$ discriminations per second.

1 Operator & 1 Invocation	
HIPA^{cc}:	$8 + 6$ LoC
Radeon R9 290X:	408.33 FPS
Tesla K20:	272.79 FPS
Xeon Phi 7120P:	16.84 FPS
Mali T-604:	2.88 FPS

(a) Laplace.

4 Operators & 21 Invocations	
HIPA^{cc}:	$29 + 17$ LoC
Radeon R9 290X:	362.85 FPS
Tesla K20:	159.49 FPS
Xeon Phi 7120P:	compiler segfault
Mali T-604:	1.77 FPS

(b) Multiresolution Filter.

6 Operators & 12 Invocations	
HIPA^{cc}:	$47 + 26$ LoC
Radeon R9 290X:	232.99 FPS
Tesla K20:	137.58 FPS
Xeon Phi 7120P:	77.30 FPS
Mali T-604:	1.26 FPS

(c) Harris Corner.

3 Operators & 5 Invocations	
HIPA^{cc}:	$26 + 28$ LoC
Radeon R9 290X:	163.59 FPS
Tesla K20:	60.66 FPS
Xeon Phi 7120P:	65.46 FPS
Mali T-604:	0.44 FPS

(d) Optical Flow.

Figure 10: Algorithms in HIPA^{cc}: edge detection (a), multiresolution processing (b), corner detection (c), as well as optical flow (d). Listed are the number of operators required for the algorithms as well as the total number of operator invocations. The (X + Y) LoC denote the DSL parametrization (X) and the algorithm implementation (Y) in DSL code. All benchmarks were performed using OpenCL on 4K UltraHD (3840×2160) images.

DSL in Scala. A DSL can make use of parallel building blocks provided by the Delite infrastructure, which can be in turn mapped efficiently to parallel execution patterns. Programs written in a DSL are broken down into these building blocks which are then combined and optimized for parallel execution. A number of DSLs have been implemented on top of Delite such as Liszt [4] for mesh-based PDE solvers or OptiML [5] for machine learning, but none for image processing. This work follows a different approach: Instead of breaking computations down into their building blocks, in order to combine them as required, this work employs domain-specific optimizations using domain knowledge for code generation.

Other domain-specific or domain-agnostic approaches focus on image processing just as HIPA^{cc}: These include RapidMind [22], Halide [2], KernelGenius [3], and the work of Howes *et al.* [31] and Cornwall *et al.* [32].

The work most close to the work at hand is the RapidMind multi-core development platform [22] targeting standard multi-core processors as well as accelerators like the Cell Broadband Engine (Cell B. E.) and GPUs. The RapidMind technology is based on Sh [33], a high-level metaprogramming language for graphics cards. RapidMind provides its own data types that can be arranged in multi-dimensional arrays. Boundary handling properties are defined on accessors and neighboring elements can be accessed using the *shift()* method on input data. Since there are no details on code generation for boundary handling publicly available for RapidMind, the approach followed in this work can only be compared quantitatively with the one of RapidMind. However, our evaluation indicates that no domain-specific knowledge is used for code generation. In 2009, Intel acquired RapidMind and incorporated the RapidMind technology into Intel Array Building Blocks (ArBB) [34]. The focus of Intel's ArBB is on vector parallel programming and for that reason, image processing features of RapidMind like generic boundary handling support were not adopted. Also the back ends for accelerators were dropped.

Cornwall *et al.* [32] introduced a domain-specific framework for visual effects that generates target code for CUDA. Their framework is based on *indexed metadata* in the form of C++ classes. Similar optimizations are available for code generation (staging data to local memory, realignment of threads for coalescing, 1:N mapping, and automatic work-group size and shape selection). However, only data containers for images are provided. The work at hand introduces notations for other domain-specific traits such as filter masks and provides additional domain-specific optimizations such as specialized code variant generation for boundary handling.

Howes *et al.* [31] propose a framework for decoupled access/execute (Æcute) specification, capturing both execution constraints and memory access patterns of a computational kernel. Their domain-agnostic framework is also based on C++ classes with tailored implementations for the Cell B. E. and standard multi-core processors. In [35], the authors highlight possible annotations for GPU accelerators using CUDA as target language. They provide hand-tuned implementations showing the benefit that can be achieved if a source-to-source compiler makes use of this knowledge. HIPA^{cc} is capable to employ not only the proposed optimizations, but also further transformations. We have shown that we can achieve the same performance compared to their hand-tuned code [36].

Halide [2], a DSL for image processing, follows a promising path by using functional programming to express kernels in a compact and concise way. Using functional programming, images have no explicit storage, but are pure functions that define the value of each pixel. The schedule for these functions has to be specified separately and determines how the computation is mapped to the target hardware and if memory needs to be allocated for the computation. In particular, image processing pipelines benefit from this concept. However, no domain-specific optimizations are applied during code generation. Boundary handling is realized as a function that is applied for each image pixel (with the resulting overhead).

KernelGenius [3] provides a DSL for image processing kernels using C-like syntax and generates target OpenCL code for the STHORM embedded many-core architecture of STMicroelectronics. Functions in a *kernel* can be combined

using common predefined templates to create an extended Synchronous Dataflow (SDF) graph. Nodes of this graph are then scheduled within an OpenCL kernel to exploit locality. A concise syntax is provided to express boundary handling modes and convolutions.

6 Conclusion

In this article, we have introduced a DSL for image processing and a source-to-source compiler to generate highly efficient parallel implementations. Both are part of the HIPA^{cc} framework and are available as open-source under http://hipacc-lang.org.

We have shown that a close co-design of language and compiler can exploit domain-specific properties for specialization. Considering target architecture information yields tailored implementations that are in many cases faster than corresponding hand-tuned implementations. The abstractions of the DSL allow to map an algorithm to a quite large spectrum of GPU target architectures and to exploit the available types of parallelism. The algorithm description itself is not contaminated with the mapping and parallelization for a specific architecture or other hardware-dependent optimizations. We plan to exploit this property also for vectorization and have recently shown that it can be used for synthesis of efficient hardware designs [37].

Acknowledgments

This work is supported by the German Research Foundation (DFG), as part of the Research Training Group 1773 "Heterogeneous Image Systems". The Tesla K20 used for this research was donated by the NVIDIA Corporation.

References

[1] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming", *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2011.

[2] 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)*, vol. 31, no. 4, 32:1–32:12, 2012.

[3] T. Lepley, P. Paulin, and E. Flamand, "A novel compilation approach for image processing graphs on a many-core platform with explicitly managed memory", in *Proc. of the 2013 Int. Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, IEEE, 2013, 6:1–6:10.

[4] 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 *Proc. of the 2011 Int. Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, 2011, 9:1–9:12.

[5] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun, "OptiML: An implicitly parallel domain-specific language for machine learning", in *Proc. of the 28th Int. Conference on Machine Learning (ICML)*, ACM, 2011, pp. 609–616.

[6] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert, "Generating device-specific GPU code for local operators in medical imaging", in *Proc. of the 26th IEEE Int. Parallel & Distributed Processing Symposium (IPDPS)*, IEEE, 2012, pp. 569–581.

[7] R. Membarth, "Code generation for GPU accelerators from a domain-specific language for medical imaging", Verlag Dr. Hut, Munich, Germany, Dissertation, Hardware/-Software Co-Design, Department of Computer Science, University of Erlangen-Nuremberg, Germany, 2013.

[8] R. Membarth, O. Reiche, F. Hannig, and J. Teich, "Code generation for embedded heterogeneous architectures on Android", in *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, IEEE, 2014, 86:1–86:6.

[9] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures", *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[10] M. H. Halstead, *Elements of Software Science*, ser. Operating and Programming Systems. Elsevier, 1977.

[11] I. N. Bankman, *Handbook of Medical Image Processing and Analysis*. Academic Press, 2008, vol. 2.

[12] J. C. Russ, *The Image Processing Handbook*. CRC Press, 2006, vol. 5.

[13] R. Klette and P. Zamperoni, *Handbook of Image Processing Operators*. John Wiley & Sons, 1996, vol. 1.

[14] P. Burt and E. Adelson, "The Laplacian pyramid as a compact image code", *IEEE Transactions on Communications*, vol. 31, no. 4, pp. 532–540, 1983.

[15] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

[16] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images", IEEE, 1998, pp. 839–846.

[17] G. E. Blelloch, "Prefix sums and their applications", in *Synthesis of Parallel Algorithms*, J. H. Reif, Ed., Morgan Kaufmann, 1993, ch. 1, pp. 35–60.

[18] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986, vol. 2.

[19] N. Wirth, "Program development by stepwise refinement", *Communications of the ACM*, vol. 14, no. 4, pp. 221–227, 1971.

[20] R. Karrenberg and S. Hack, "Whole-function vectorization", in *Proc. of the 9th Annual IEEE/ACM Int. Symposium on Code Generation and Optimization (CGO)*, IEEE, 2011, pp. 141–150.

[21] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[22] RapidMind, *RapidMind development platform documentation*, RapidMind Inc., 2009.

[23] C. H. González and B. B. Fraguela, "A generic algorithm template for divide-and-conquer in multicore systems", in *Proc. of the 12th Int. Conference on High Performance Computing and Communications (HPCC)*, IEEE, 2010, pp. 79–88.

[24] J. M. Stroud, "The fine structure of psychological time", *Information Theory in Psychology*, 1956.

[25] R. D. Gordon and M. H. Halstead, "An experiment comparing Fortran programming times with the software physics hypothesis", in *Proc. of the National Computer Conference; American Federation of Information Processing Societies (AFIPS)*, ACM, 1976, pp. 935–937.

[26] C. Harris and M. Stephens, "A combined corner and edge detector", in *Proc. of the 4th Alvey Vision Conference*, 1988, pp. 147–151.

[27] F. Stein, "Efficient computation of optical flow using the census transform", in *Pattern Recognition*, ser. Lecture Notes in Computer Science, vol. 3175, Springer, 2004, pp. 79–86.

[28] P. Feautrier and C. Lengauer, "Polyhedron model", in *Encyclopedia of Parallel Computing*, Springer, 2011, pp. 1581–1592.

[29] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun, "Language virtualization for heterogeneous parallel computing", in *Proc. of the ACM Int. Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, ACM, 2010, pp. 835–847.

[30] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to hetero-

geneous parallelism", in *Proc. of the 16th Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM, 2011, pp. 35–46.

[31] L. Howes, A. Lokhmotov, A. Donaldson, and P. H. J. Kelly, "Deriving efficient data movement from decoupled access/execute specifications", in *Proc. of the 4th Int. Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, Springer, 2009, pp. 168–182.

[32] J. L. Cornwall, L. Howes, P. H. J. Kelly, P. Parsonage, and B. Nicoletti, "High-performance SIMT code generation in an active visual effects library", in *Proc. of the 6th ACM Conference on Computing Frontiers (CF)*, ACM, 2009, pp. 175–184.

[33] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, "Shader algebra", *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 787–795, 2004.

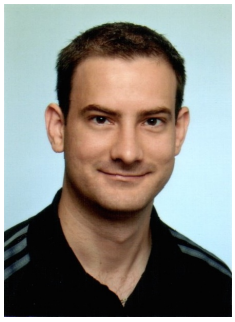
[34] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Du Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language", in *Proc. of the 9th Annual IEEE/ACM Int. Symposium on Code Generation and Optimization (CGO)*, IEEE, 2011, pp. 224–235.

[35] L. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. J. Kelly, "Towards metaprogramming for parallel systems on a chip", in *Proc. of the 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, Springer, 2009, pp. 36–45.

[36] R. Membarth, A. Lokhmotov, and J. Teich, "Generating GPU code from a high-level representation for image processing kernels", in *Proc. of the 5th Workshop on Highly Parallel Processing on a Chip (HPPC)*, Springer, 2011, pp. 270–280.

[37] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich, "Code generation from a domain-specific language for C-based HLS of hardware accelerators", in *Proc. of the Int. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, ACM, 2014, 17:1–17:10.

7 Author Biography



Richard Membarth is a senior researcher at the German Research Center for Artificial Intelligence (DFKI). He holds a diploma degree in Computer Science from the University of Erlangen-Nuremberg and a postgraduate diploma in Computer and Information Sciences from Auckland University of Technologies. In 2013, he received his Ph.D. (Dr.-Ing.) at the University of Erlangen-Nuremberg on the automatic code generation for GPU accelerators from a domain-specific language for medical imaging. After his Ph.D., he joined the Graphics Chair and the Intel Visual Computing Institute at Saarland University as postdoctoral researcher. His research interests include parallel computer architectures and programming models with focus on automatic code generation.

Oliver Reiche holds a bachelor's degree and a master's degree in Computer Science, both from the University of Applied Sciences in Nuremberg. Since 2012, he is a Ph.D. student at the Chair for Hardware/Software Co-Design at the Department of Computer Science at the University Erlangen-Nuremberg and a member of the Research Training Group Heterogeneous Image Systems. His research interests are embedded systems, efficient mapping strategies of image algorithms to heterogeneous architectures and domain-specific languages.

Frank Hannig leads the Architecture and Compiler Design Group in the CS Department at the University of Erlangen-Nuremberg, Germany, since 2004. He received a diploma degree in an interdisciplinary course of study in EE and CS from the University of Paderborn, Germany in 2000 and a Ph.D. in CS from the University of Erlangen-Nuremberg in 2009. His main research interests are the design of massively parallel architectures, ranging from dedicated hardware to multi-core architectures, mapping methodologies for domain-specific computing, and architecture/compiler co-design.

Jürgen Teich received an M.S. degree (Dipl.-Ing.; with honors) from the University of Kaiserslautern, Germany, in 1989 and a Ph.D. (summa cum laude) from Saarland University, Germany, in 1993. In 1994, he joined the DSP design group of Prof. E. A. Lee in the Department of Electrical Engineering and Computer Sciences (EECS), University of California at Berkeley (PostDoc). From 1995 to 1998, he held a position at the Institute of Computer Engineering and Communications Networks Laboratory (TIK), ETH Zurich, Switzerland (Habilitation). From 1998 to 2002, he was a Full Professor in the Electrical Engineering and Information Technology Department, University of Paderborn, Germany. Since 2003, he has been a Full Professor in the Department of Computer Science, University of Erlangen-Nuremberg, Germany, holding a chair in Hardware/Software Co-Design.

Mario Körner works as software architect for the Angiography and Interventional X-Ray Systems business unit at Siemens Healthcare. He completed his degree in Computer Science at the University of Erlangen-Nuremberg in 2007. Since then he has been working in different roles on the development of medical imaging applications for interventional procedures. His main interests are algorithms for image processing, visualization and geometry calibration, software engineering methods, efficient mapping to hardware accelerators as well as usability aspects for making new technologies available to medical users.

Wieland Eckert received a M.S. degree in Computer Science (Dipl.-Inf.) and a Ph.D. (Dr.-Ing.) at the University of Erlangen-Nuremberg in 1991 and 1996, respectively. From 1996 to 1999 he held a Senior Researcher position at AT&T Labs Research in Florham Park, NJ, in the speech recognition and dialog systems laboratory. In 1999 he joined Lucent Technologies in Nuremberg, Germany, designing and developing architectures and software for embedded communication systems. In 2004 he joined Siemens Healthcare as a Senior Software Architect for interventional X-ray systems. Since 2012 he is with the research unit Corporate Technology of Siemens AG. His main research interests are effective software architectures and challenges of efficient implementation on new and powerful hardware designs like multi-core and many-core processors combined with easy to use programming models.