This is the author's version of the work. The definitive work was published in *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China, May 21–25, 2012.*

Generating Device-specific GPU code for Local Operators in Medical Imaging

Richard Membarth, Frank Hannig, and Jürgen Teich
Department of Computer Science,
University of Erlangen-Nuremberg, Germany.
{richard.membarth,hannig,teich}@cs.fau.de

Mario Körner and Wieland Eckert
Siemens Healthcare Sector, H IM AX,
Forchheim, Germany.
{mario.koerner,wieland.eckert}@siemens.com

***Abstract*—To cope with the complexity of programming GPU accelerators for medical imaging computations, we developed a framework to describe image processing kernels in a domain-specific language, which is embedded into C++. The description uses decoupled access/execute metadata, which allow the programmer to specify both execution constraints and memory access patterns of kernels. A source-to-source compiler translates this high-level description into low-level CUDA and OpenCL code with automatic support for boundary handling and filter masks. Taking the annotated metadata and the characteristics of the parallel GPU execution model into account, two-layered parallel implementations—utilizing SPMD and MPMD parallelism—are generated. An abstract hardware model of graphics card architectures allows to model GPUs of multiple vendors like AMD and NVIDIA, and to generate device-specific code for multiple targets. It is shown that the generated code is faster than manual implementations and those relying on hardware support for boundary handling. Implementations from RapidMind, a commercial framework for GPU programming, are outperformed and similar results achieved compared to the GPU backend of the widely used image processing library OpenCV.**

***Keywords*-GPU; CUDA; OpenCL; domain-specific language; code generation; medical imaging; local operators**

I. Introduction

Computer systems are increasingly heterogeneous, as many important computational tasks, such as multimedia processing, can be *accelerated* by special-purpose processors that outperform general-purpose processors by one or two orders of magnitude, importantly, in terms of energy efficiency as well as in terms of execution speed.

Until recently, every accelerator vendor provided their own application programming interface (API), typically based on the C language. For example, NVIDIA's API called CUDA targets systems accelerated with Graphics Processing Units (GPUs). In CUDA, the programmer dispatches compute-intensive data-parallel functions (*kernels*) to the GPU, and manages the interaction between the CPU and the GPU via API calls. Ryoo et al. [1] highlight the complexity of CUDA programming, in particular, the need for exploring thoroughly the space of possible implementations and configuration options. OpenCL, a new industry-backed standard API that inherits many traits from CUDA, aims to provide software portability across heterogeneous systems: correct OpenCL programs will run on any standard-compliant implementation. OpenCL per se, however, does not address the problem of *performance portability*; that is, OpenCL code optimized for one accelerator device may perform dismally on another, since performance may significantly depend on low-level details, such as data layout and iteration space mapping [2], [3].

CUDA and OpenCL are low-level programming languages since the user has to take care of parallelizing the algorithm as well as the communication between the host and the GPU accelerator. In general, low-level programming increases the cost of software development and maintenance: whilst low-level languages can be robustly compiled into efficient machine code, they effectively lack support for creating portable and composable software.

High-level languages with domain-specific features are more attractive to domain experts, who do not necessarily wish to become target system experts. To compete with low-level languages for programming accelerated systems, however, domain-specific languages should have an acceptable performance penalty.

We present the Heterogeneous Image Processing Acceleration (HIPAcc) framework for medical image processing in the domain of angiography that allows programmers to concentrate on developing algorithms and applications, rather than on mapping them to the target hardware. Specifically, as design entry to our framework, we defined a domain-specific language to describe image processing kernels on an abstract level. We identified three groups of operators that have to be supported by the domain-specific language for medical imaging: a) point operators, b) local operators, and c) global operators. Point operators are applied to the pixels of the image and solely the pixel the point operator is applied to contributes to the operation (e. g., adding a constant to each pixel in the image). Local operators are similar to point operators, but also neighboring pixels contribute to the operation (e. g., replacing the current pixel value with the average value of its neighbors). In contrast, global operators (e. g., reduction operators) produce one output for the operator applied to all pixels of the image (e. g., compute the sum of all pixels). While the basic framework supporting point operations was presented earlier (see [4]), we focus on the support of local operators and the efficient mapping to target hardware in this paper.

Our framework is implemented as a library of C++ classes and will be summarized in Section II. The DSL of our framework is extended to describe local operators (Section III). Our Clang-based compiler translates this description into efficient low-level CUDA and OpenCL code as well as corresponding code to talk to the GPU accelerator (Section IV). The transformations applied to the target code and the run-time configuration are device-specific and take an abstract architecture model

of the target graphics card hardware into account (Section V). This results in efficient code that is evaluated against highly optimized GPU frameworks from RapidMind and OpenCV, as well as hand-written implementations (Section VI). In Section VII, we compare our work against other approaches. We outline future extensions to our framework in Section VIII for code generation optimizations. Finally, we conclude our paper in Section IX.

II. Image Processing Framework

In this paper, we use and extend our previous framework for describing image processing kernels [4]. The proposed framework uses a source-to-source compiler based on Clang [5] in order to generate low-level, optimized CUDA and OpenCL code for execution on GPU accelerators. The framework consists of built-in C++ classes that describe the following four basic components required to express image processing on an abstract level:

- *Image*: Describes data storage for the image pixels. Each pixel can be stored as an integer number, a floating point number, or in another format such as RGB, depending on instantiation of this templated class. The data layout is handled internally using multi-dimensional arrays.
- *Iteration Space*: Describes a rectangular region of interest in the output image, for example the complete image. Each pixel in this region is a point in the iteration space.
- *Kernel*: Describes an algorithm to be applied to each pixel in the *Iteration Space*.
- *Accessor*: Describes which pixels of an *Image* are *seen* within the *Kernel*. Similar to an *Iteration Space*, the Accessor defines an *Iteration Space* on an input image.

These components are an instance of decoupled access/execute metadata [3]: the *Iteration Space* specification provides ordering and partitioning constraints (execute metadata); the *Kernel* specification provides a pattern of accesses to uniform memory (access metadata). Currently, the access/execute metadata is mostly implicit: we assume that the iteration space is independent in all dimensions and has a 1:1 mapping to work-items (threads), and that the memory access pattern is obvious from the kernel code.

A. Example: Bilateral Filter

In the following, we illustrate our image processing framework using a bilateral filter that smoothes an image while preserving edges within the image [6]. The bilateral filter applies a local operator to each pixel that consists of two components to determine the weighting of pixels: a) the *closeness* function that considers the distance to the center pixel, and b) the *similarity* function that takes the difference between pixel values into account. Gaussian functions of the Euclidean distance are used for the closeness and similarity weights as seen in Equation (1) and (2), respectively.

$$closeness((x,y),(x',y')) = e^{-\frac{1}{2}(\frac{\|(x,y)-(x',y')\|}{\sigma_d})^2} \tag{1}$$

$$similarity(p,p') = e^{-\frac{1}{2}(\frac{\|p-p'\|}{\sigma_r})^2} \tag{2}$$

The bilateral filter replaces each pixel by a weighted average of geometrically nearby and photometrically similar pixel values. Only pixels within the neighborhood of the relevant pixel are used. The neighborhood and consequently also the convolution size is determined by the geometric spread σ_d. The parameter σ_r (photometric spread) in the similarity function determines the amount of combination. Thereby, difference of pixel values below σ_r are considered more similar than differences above σ_r.

For each pixel in the output image, the filter kernel is applied to a neighborhood of pixels in the input image. When executed on a GPU, the required operations to produce one pixel in the output image are specified in a program. Such a program is also called *kernel*, executed on the GPU in parallel, and applied to each pixel of the image independently. The pseudo code for the bilateral filter executed on GPUs is shown in Algorithm 1. Here, the kernel itself is described in lines 3–13, defining the bilateral filter for one pixel, taking the pixels in its neighborhood into account. This code is executed sequentially as can be seen in the doubly nested innermost loops. This is the code that is typically written by a programmer in either CUDA or OpenCL. The parallel execution on the graphics hardware is expressed by the parallel *foreach* loops in line 1 and 2. While the first parallel for loop describes the data-parallel execution within groups of processors on the GPU (think of it as a Single Instruction Multiple Data (SIMD) unit), the second parallel for loop depicts the parallel execution on all available groups of processors. Note that this is a two-layered approach exploiting two different types of parallelism: the Single Program Multiple Data (SPMD) parallel execution model within groups of processors, and the Multiple Program Multiple Data (MPMD) parallel execution between multiple groups of processors. Although the graphics hardware supports the execution of different programs on different SIMD units, programs written in CUDA and OpenCL contain the same code for all SIMD units.

Algorithm 1: Bilateral filter implementation on the graphics card.

```
1  foreach thread block b in grid g do in parallel
2      foreach thread t in thread block b do in parallel
3          x ← get_index_x(b,t);
4          y ← get_index_y(b,t);
5          p,k ← 0;
6          for yf = −2∗sigma_d to +2∗sigma_d do
7              for xf = −2∗sigma_d to +2∗sigma_d do
8                  c ← closeness((x,y),(x+xf,y+yf));
9                  s ← similarity(input[x,y],
                       input[x+xf,y+yf]);
10                 k ← k+c∗s;
11                 p ← p+c∗s∗input[x+xf,y+yf];
12             end
13         end
14         output[x,y] ← p/k;
15     end
16 end
```

To express this filter in our framework, the programmer

derives a class from the built-in *Kernel* class and implements the virtual *kernel* function, as shown in Listing 1. To access the pixels of an input image, the parenthesis operator `()` is used, taking the column (`dx`) and row (`dy`) offsets as optional parameters. The output image as specified by the *Iteration Space* is accessed using the `output()` method provided by the built-in *Kernel* class. The user instantiates the class with input image accessors, one iteration space, and other parameters that are member variables of the class.

```
1  class BilateralFilter : public Kernel<float> {
2    private:
3      Accessor<float> &Input;
4      int sigma_d, sigma_r;
5
6    public:
7      BilateralFilter(IterationSpace<float> &IS,
           Accessor<float> &Input, int sigma_d, int
           sigma_r) :
8          Kernel(IS), Input(Input), sigma_d(sigma_d)
               , sigma_r(sigma_r)
9      { addAccessor(&Input); }
10
11   void kernel() {
12     float c_r = 1.0f/(2.0f*sigma_r*sigma_r);
13     float c_d = 1.0f/(2.0f*sigma_d*sigma_d);
14     float d = 0.0f, p = 0.0f, s = 0.0f;
15
16     for (int yf = -2*sigma_d; yf<=2*sigma_d; yf++)
           {
17       for (int xf = -2*sigma_d; xf<=2*sigma_d; xf
             ++) {
18         float diff = Input(xf, yf) - Input();
19
20         s = exp(-c_r * diff*diff);
21         c = exp(-c_d * xf*xf) * exp(-c_d * yf*yf);
22         d += s*c;
23         p += s*c * Input(xf, yf);
24       }
25     }
26     output() = p/d;
27   }
28 };
```

Listing 1: Kernel description of the bilateral filter.

In Listing 2, the input and output *Image* objects `IN` and `OUT` are defined as two-dimensional $W \times H$ grayscale images, having pixels represented as floating-point numbers (lines 8–9). The *Image* object `IN` is initialized with the `host_in` pointer to a plain C array with raw image data, which invokes the = operator of the *Image* class (line 12). The region of interest `IsOut` contains the whole image (line 15). Also the *Accessor* `AccIn` on the input image is defined on the whole image (line 18). The kernel is initialized with the iteration space object, accessor objects and parameters *sigma_d* and *sigma_r* for the bilateral filter (line 21), and executed by a call to the *execute()* method (line 24). To retrieve the output image, the `host_out` pointer to a plain C data array is assigned the *Image* object `OUT`, which invokes the `getData()` operator (line 27).

From this code, the source-to-source compiler can finally create CUDA and OpenCL code for execution on GPU accelerators automatically. In [4], we have presented an initial version of our framework, that is, how to generate efficient code for point operators by support for effective device-specific optimization such as global memory padding for memory coalescing and optimal memory bandwidth utilization. In the remaining of this paper, we describe the extension of the framework of how to utilize access/execute metadata for efficient code generation for local operators in medical imaging and the optimal mapping of this code to different GPU accelerator architectures from AMD and NVIDIA.

```
1  const int width = 1024, height = 1024, sigma_d =
       3, sigma_r = 5;
2
3  // pointers to raw image data
4  float *host_in = ...;
5  float *host_out = ...;
6
7  // input and output images
8  Image<float> IN(width, height);
9  Image<float> OUT(width, height);
10
11 // initialize input image
12 IN = host_in; // operator=
13
14 // define region of interest
15 IterationSpace<float> IsOut(OUT);
16
17 // accessor used to access image pixels
18 Accessor<float> AccIn(IN);
19
20 // define kernel
21 BilateralFilter BF(IS, AccIn, sigma_d, sigma_r);
22
23 // execute kernel
24 BF.execute();
25
26 // retrieve output image
27 host_out = OUT.getData();
```

Listing 2: Example code that initializes and executes the bilateral filter.

III. Local Operators

In image processing and in particular in medical imaging, local operators are widely used. These operators can be applied independently to all pixels of the image and depend only on the pixel itself and the pixels in its neighborhood. The neighborhood read by local operators in medical imaging is typically symmetrically arranged around the pixel to be calculated with the same diameter in each direction. Furthermore, local operators describe the convolution of an image by a *filter mask* (e. g., Gaussian function) with the following properties: a) the filter mask is centered at the pixel it is applied to $[0,0]$ and b) bounded to the neighborhood $[-m,+m] \times [-n,+n]$. The latter implies a window size $(2m+1) \times (2n+1)$ of local operators to be uneven (e. g., 3×3, 5×5, 9×3). Since the filter mask used for convolution is typically constant, the values for the filter mask can be stored to a lookup table. Figure 1 shows the filter masks used for a window size of 3×3: the pixels in the neighborhood (*f*) are convolved with the closeness (*c*) and similarity (*s*) filter masks. While the filter mask for the closeness component is constant and depends only on the distance to the center pixel, the filter mask for the similarity component depends on the pixel values and has to be calculated for each pixel separately.

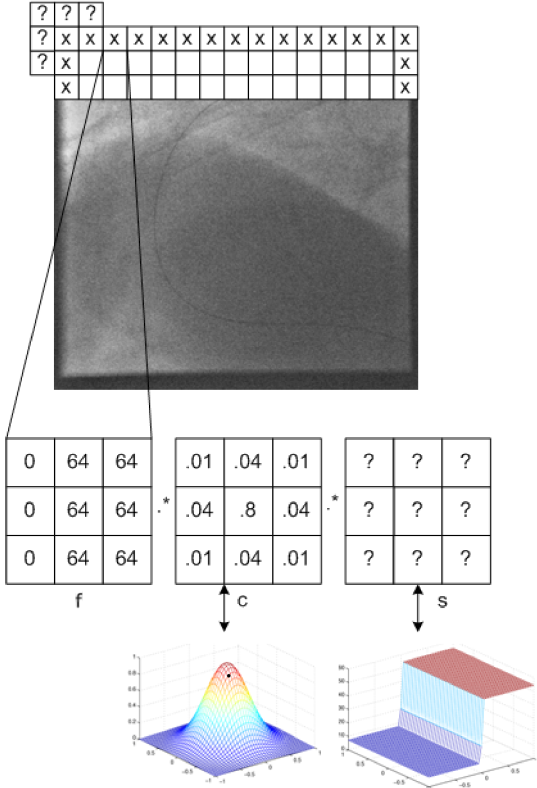

Figure 1: Bilateral filter: Convolution of the image (*f*) with the closeness (*c*) and similarity (*s*) filter masks.

Figure 1 also indicates a problem when local operators are applied to an image: if the 3×3 filter mask is applied to a pixel located at the image border, pixels beyond the image are accessed. Accessing pixels out of bounds leads to erroneous pixel values in the resulting image and may result in program termination. Therefore, boundary handling modes need to be defined that determine the behavior when pixels beyond the image are accessed.

The remainder of this section discusses how properties of local operators are expressed in our framework that relieve developers from low-level programming of GPU accelerators. In the following sections, the benefit of this domain-specific knowledge for efficient low-level code generation for GPU accelerators using source-to-source compilation is shown.

A. Boundary Handling

When an image is accessed out of bounds, there are different possibilities to handle this situation: the image is virtually expanded beyond the border and the pixel value of the expanded image is returned. This can be realized by a) adding additional pixels at each image boundary, and b) adjusting the index of the accessed pixel to a pixel that resides within the image. In our implementation, we follow the second approach, which is discussed in Section IV.

However, we will show that boundary handling is completely transparent to the programmer in our framework. The programmer specifies only the desired boundary handling mode on images and the source-to-source compiler will handle the low-level details. Supported boundary handling modes are listed in Table I and visualized in Figure 2. These include the most important boundary handling modes commonly encountered in image processing[1]. In particular, mirroring is important in medical imaging, for example, when a multiresoultion filter (see [7]) is applied to an image: the image gets upsampled multiple times and at the border occur large unnatural-looking artifacts when the border pixel gets replicated repeatedly. In contrast, using mirroring leads to natural looking images.

Table I: Supported boundary handling modes.

Mode	Returned pixel value for out of bounds:
Undefined	not specified, undefined
Repeat	pixel value of image repeated at the border
Clamp	last valid pixel within image
Mirror	pixel value of image mirrored at the border
Constant	constant value, user defined

Instead of tying the boundary handling mode to a particular image, the framework provides *Accessors* to describe the way an image is accessed as mentioned earlier. That is, no memory is allocated and hold by *Accessors*. In case boundary handling is required, the *Accessor* defines the view on a *BoundaryCondition* object rather than on an *Image*. The *BoundaryCondition* itself specifies the boundary handling mode on an input image and the size of the local operator. Listing 3 shows the collaboration of *BoundaryCondition* and *Accessor* in order to define clamping as boundary handling mode for a local operator with a window size of $(4 * \sigma_d + 1) \times (4 * \sigma_d + 1)$. The resulting *Accessor* can be used in the *BilateralFilter* implementation defined in Listing 1 and instantiated in Listing 2 to add boundary handling support. In case a constant boundary handling is specified, the *BoundaryCondition* requires the constant value as an additional parameter.

```
1 // input image and accessor with boundary handling
2 Image<float> IN(width, height);
3 BoundaryCondition<float> BcIn(IN, 4*sigma_d+1, 4*
      sigma_d+1, BOUNDARY_CLAMP);
4 Accessor<float> AccIn(BcIn);
```

Listing 3: Usage of a *BoundaryCondition* to define the boundary handling mode for a local operator.

The source-to-source compiler creates then the appropriate boundary handling support utilizing the access metadata provided by the framework. No further modifications are required. Tying the boundary handling mode to an *Accessor* instead of an *Image* has the additional benefit that multiple boundary handling modes can be defined on the same image. This allows different algorithms or kernels to access the same image, but to use different boundary handling modes—the

[1]However, if necessary, the framework can be easily extended to support further boundary handling modes.

(a) Undefined.

(b) Repeat.

(c) Clamp.

(d) Mirror.

(e) Constant.

Figure 2: Boundary handling modes for image processing. By default, the behavior is undefined when the image is accessed out of bounds (a). The framework allows to specify different boundary handling modes like repeating the image (b), clamping to the last valid pixel (c), mirroring the image at the image border (d), and returning a constant value when accessed out of bounds (e).

most appropriate one for the current algorithm—without the need to keep separate copies of the image.

B. Filter Masks

The framework provides also support for filter masks for local operators: a *Mask* holds the precalculated values used by the convolution filter function. Since the filter mask is constant for one kernel, this allows the source-to-source compiler to apply optimizations such as constant propagation to avoid redundant calculations. To define a *Mask* in the framework, the filter mask size has to be provided. Afterwards, the precalculated values can be assigned to the *Mask*. Listing 4 shows the usage of a *Mask* for the closeness filter mask.

```
1 // pre-calculated mask coefficients
2 float mask[] = {0.018316, ... };
3
4 // filter mask
5 Mask<float> CMask(4*sigma_d+1, 4*sigma_d+1);
6 CMask = mask;
```

Listing 4: Usage of a *Mask* to store the filter mask for a local operator.

Listing 5 shows the usage of the *Mask* within the kernel of the *BilateralFilter*: using the indices *xf* and *yf*, the precalculated values are retrieved from the closeness filter mask. Also note that the calculation of *c_d* is not necessary anymore.

```
1 void kernel() {
2   float c_r = 1.0f/(2.0f*sigma_r*sigma_r);
3   float d = 0.0f, p = 0.0f, s = 0.0f;
4
5   for (int yf = -2*sigma_d; yf<=2*sigma_d; yf++) {
6     for (int xf = -2*sigma_d; xf<=2*sigma_d; xf++)
          {
7       float diff = Input(xf, yf) - Input();
8
9       s = exp(-c_r * diff*diff);
10      c = CMask(xf, yf);
11      d += s*c;
12      p += s*c * Input(xf, yf);
13    }
14  }
15  output() = p/d;
16 }
```

Listing 5: Using a *Mask* within the *kernel* to retrieve the filter mask coefficients for a local operator.

IV. Source-to-Source Compilation

This section describes the transformations applied by our source-to-source compiler and the steps taken to create CUDA and OpenCL code from a high-level description of local operators and access metadata. Based on the latest Clang compiler framework [5], our source-to-source compiler uses the Clang frontend for C/C++ to parse the input files and to generate an abstract syntax tree (AST) representation of the source code. Our backend uses this AST representation to apply transformations and to generate host and device code in CUDA or OpenCL.

A. Local Operators

Since local operators and convolution functions read typically neighboring pixels to calculate the value of the result pixel, most of the neighboring pixels read for pixel $p_{i,j}$ are also required for pixel $p_{i+1,j}$. This results in redundant fetches of the same pixels and imposes high pressure on the global memory bandwidth. To preserve global memory bandwidth, the region read by a group of threads can be a) staged into fast on-chip memory (*scratchpad memory*) and read from there afterwards, or b) accessed by a memory path that traverses a cache. In the latter case, only the first access to a pixel has the long latency of global memory. Subsequent accesses are served by the cache. In the former case, the data is staged

into scratchpad memory and memory accesses of the kernel go to the fast scratchpad memory. However, synchronization is required before the calculation can begin. This separates data transfer and calculation phases within the kernel. The benefit of massive multithreading provided by the underlying hardware is to hide memory transfers when data transfers and calculations are done at the same time. This benefit is lost when data is staged to scratchpad memory. Hence, staging to scratchpad memory makes only sense in case the benefit of data reuse exceeds the multithreading benefit. For local operators with small window sizes, this is rarely the case. Nonetheless, our source-to-source compiler supports both options.

Texturing memory: All graphics cards from AMD and NVIDIA support cached data access, either using texturing hardware or by default (on newer Fermi GPUs from NVIDIA). In CUDA, texturing hardware can be utilized reading from a texture reference that is bound to global memory. In OpenCL, the texturing hardware is used when data is read from an image object. Therefore, accesses to *Image(x, y)* objects have to be mapped to the corresponding *tex1Dfetch()* and *read_imagef()* functions in CUDA and OpenCL, respectively. However, this is only valid if data is read. When data is written to an *Image* object, normal global memory array pointers are used in CUDA and the *write_imagef()* function in OpenCL. That is, prior to the mapping of *Image* accesses to the low-level equivalents, we perform a read/write analysis of the *kernel* method. Therefore, a control-flow graph (CFG) of the instructions in the *kernel* method is created and traversed afterwards. Access information is stored for each *Image* and *Accessor* object and used to select the appropriate texturing function call. This results in a mapping of read/write-accesses as shown in Listing 6. The mapping is done using a recursive AST-visitor. Whenever an *Image* or *Accessor* node is visited, the above described transformations are applied.

```
1  // Read-access to Accessor
2  IN(xf, yf)
3  // CUDA read with offset
4  tex1Dfetch(_texIN, gid_x+xf + (gid_y+yf)*stride)
5  // OpenCL read with offset
6  read_imagef(imgIN, Sampler, (int2)(gid_x+xf, gid_y
       +yf)).x
7
8  // Write-access to output() method
9  output()
10 // CUDA write without offset
11 OUT[gid_x + gid_y*stride]
12 // OpenCL write without offset
13 write_imagef(imgOUT, (int2)(gid_x, gid_y), (float4
       ) val);
```

Listing 6: Using texturing hardware to read from an *Accessor*.

The image object access functions in OpenCL take and return always vector elements with size of four, although only one of the four components is required for the example above. Therefore, the *CL_R* channel order is used, which maps only one of the four components to memory and populates the remaining three channels with zeros. The corresponding extraction from and packing to vector elements is added by the framework as well. Once the accesses to *Image* objects are mapped, the low-level code can be emitted. When this is done, also the corresponding CUDA texture reference and OpenCL *sampler* definitions are created. The OpenCL kernel-function parameters for images are emitted with the corresponding *read_only* and *write_only* attributes obtained from the read/write analysis. Texture references are not added as kernel-function parameters in CUDA since they are static and globally visible in CUDA.

Scratchpad memory: Current graphics cards provide fast on-chip scratchpad memory, also called *shared memory* in CUDA and *local memory* in OpenCL, that is shared between all threads of a SIMD unit. Adding the *__shared__* (CUDA) and *__local* (OpenCL) keywords to memory declarations within a kernel allows to use this memory. Since the memory is shared between all threads mapped to one SIMD unit (number of threads » SIMD width), synchronization between threads mapped to a SIMD unit is required so that all threads have a consistent view of the scratchpad memory. This is done by the *__synchthreads()* (CUDA) and *barrier()* (OpenCL) function. Only after all threads have reached this synchronization point, execution continues. Using scratchpad memory includes two phases: first, the data is staged from the GPU memory into scratchpad memory and second, data accesses are redirected to the scratchpad. In Listing 7, the size of the scratchpad memory depends on `BSY`/`BSX`, the size of the 2D image subregion mapped to the SIMD unit as well as on `SY`/`SX`, the image region accessed beyond the subregion within the local operator. A constant of 1 is added to `BSX` so that different banks of the scratchpad memory are accessed for row-based filters to avoid bank conflicts. The data from global memory is staged into scratchpad memory in multiple steps, depending on the size of additional pixels required by the kernel. When data is read from the scratchpad memory, the thread identifiers of the threads mapped to one SIMD unit `threadIdx` and *get_local_id()* are used for the CUDA and OpenCL code, respectively.

```
1  // Phase 1: stage data to scratchpad memory
2  // CUDA
3  __shared__ float _smemIN[SY + BSY][SX + BSX + 1];
4  _smemIN[threadIdx.y][threadIdx.x] = IN[..];
5  if (...) _smemIN[threadIdx.y + ..][threadIdx.x +
       ..] = IN[..];
6  ..
7  __syncthreads();
8  // OpenCL
9  __local float _smemIN[SY + BSY][SX + BSX + 1];
10 _smemIN[get_local_id(1)][get_local_id(0)] = IN[..]
       ;
11 if (...) _smemIN[get_local_id(1) + ..][
       get_local_id(0) + ..] = IN[..];
12 ..
13 barrier(CLK_LOCAL_MEM_FENCE);
14
15 // Phase 2: map accesses to scratchpad memory
16 IN(xf, yf)
17 // CUDA read with offset
18 _smemIN[threadIdx.y + yf][threadIdx.x + xf]
19 // OpenCL read with offset
20 _smemIN[get_local_id(1)+ yf, get_local_id(0) + xf]
```

Listing 7: Staging pixels to scratchpad memory before accessing image pixels.

B. Boundary Handling

Boundary handling for image processing kernels in CUDA/OpenCL is typically done with the help of texturing hardware. When a texture reference/sampler is defined, an address mode can be specified. This mode defines the behavior when textures are accessed out of bounds and can be set to clamp or repeat accesses. For all other modes, boundary handling code has to be added by the programmer. That is, conditional statements have to be added to check if the indices address pixels within the image. However, this goes along with large overheads: since the same code is executed by all threads (i. e., for each pixel), the conditional statements have to be evaluated for each pixel, although it is only required at the image border.

The access metadata obtained from the *Accessor* and *Boundary Condition* definition introduced in Section III specifies the range of pixels for a local operator that require boundary handling code. This access metadata can be combined with the two-layered parallel execution model described in Algorithm 1 to reduce the overhead imposed by boundary handling code: special boundary handling mode is added for each border—resulting in nine different kernel implementations as depicted in Figure 3. At the top-left image border, only the conditionals are added that check if the indices are less than zero, at the top image border only the conditionals to check if the `y`-index is less than zero, and so on. For the largest part of the image, no conditionals have to be added at all. Launching the nine specialized implementations sequentially for the different regions of the image would decrease the total performance due to the launch overhead of the kernels and the relatively small amount of work per kernel at the image borders. Instead, the source-to-source compiler creates one big kernel that hosts all nine implementations, but executes only the required one depending on the currently processed image region.

```
1  __global__ void kernel(...) {
2  if (blockIdx.x < 1 && blockIdx.y < 6) goto TL_BH;
3  if (...) goto TR_BH;
4  if (...) goto T_BH;
5  ...
6  goto NO_BH;
7
8  TL_BH:
9    //kernel implementation with border handling for
         top left border
10   ...
11   return;
12 ...
13 NO_BH:
14   //kernel implementation with no border handling
15   ...
16 }
```

Listing 8: Using specialized border handling implementations for local operators in CUDA.

The current region is identified by the `blockIdx.x` and `blockIdx.y` built-in variables in CUDA and by the `get_group_id()` functions in OpenCL. Whether boundary handling is required for that regions depends on the size of the block processed by one SIMD unit (`blockDim.x`/`blockDim.y` and `get_local_size()`,

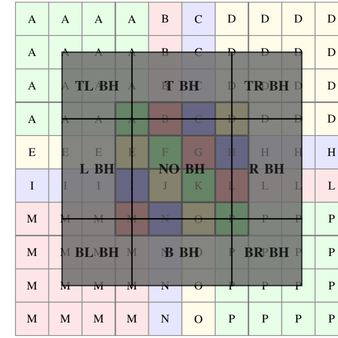

Figure 3: Kernel assignment to image border: border handling for local operators is limited to those regions where the image is accessed out of bounds.

respectively) as well as on the `width`/`height` of the image, the resulting number of blocks per image (`gridDim.x`/`gridDim.y` and `get_num_groups()`, respectively), as well as on the size of the filter mask as specified by the *Accessors* and *Boundary Condition*. In case multiple *Accessors* are used within one kernel, the largest window size specified is taken. This results in low-level source code as seen in Listing 8 for a local operator reading 13×13 pixels and a block size of 128×1. The conditional statements are also added during traversal of the AST using a visitor.

C. Filter Masks

Since the same CUDA/OpenCL kernel is executed for all pixels and, hence, the same element of the filter mask is retrieved for all threads within one thread block, it is sufficient to load the coefficient only once for a thread block. For such access patterns, the *constant memory* is optimized. Constant memory is cached and broadcasts data to all threads if all threads access the same element. The source-to-source compiler stores the filter coefficients of *Mask* objects to constant memory in CUDA and OpenCL. Constant memory can be declared either globally and initialized statically or used as dynamically allocated memory. Our framework supports both statically and dynamically initialized memory. Static constant memory can be only utilized if the mask coefficients are constants and can be precomputed at compile time. In this case, the source-to-source compiler can declare a statically initialized constant memory array using the `__device__` `__constant__` qualifiers in CUDA and the `__constant` qualifier in OpenCL. For statically initialized constant memory, no kernel-function parameter is added to the kernel function. In case the *Mask* object uses coefficients that are not constant at compile time, dynamically initialized constant memory is used. In CUDA, the constant memory is only declared and gets initialized at run-time using the `cudaMemcpyToSymbol()` function. Dynamically initialized constant memory is handled like normal global memory in OpenCL and is stored to a buffer object, only the *__constant* qualifier is added to the kernel-function parameter. During AST-traversal, accesses to *Mask* objects are translated

to accesses to the constant memory arrays created for the *Mask* coefficients.

V. Device-specific Mapping

In this section, the selection of target-specific optimizations is described. The decision what optimization should be applied for what target (hardware and backend) is based on our own micro-benchmarks for typical kernel candidates from the medical domain as well as on other micro-benchmarks available online for CUDA [8] and OpenCL [9].

A. Function Mapping

Since the GPU cannot execute arbitrary functions, the source-to-source compiler has to check every function call in the *kernel* method. The basic functions supported by CUDA and OpenCL are roughly the same, but differ in an important point: while CUDA preserves the suffix of mathematical functions that denotes the data type the function operates on, OpenCL removes these suffixes and overloads the mathematical functions for different data types. That is, suffixes are not allowed in OpenCL code. Therefore, our source-to-source compiler supports the mapping of function calls to the corresponding functions on different backends. For example, the `expf()` function gets mapped to `exp()` when code is generated for OpenCL. The mappings are defined in a file and by default all supported mathematical functions supported by CUDA and OpenCL are listed therein. In case a function is not supported, our compiler emits an error message to the user. This technique can be also used to map mathematical functions to faster intrinsic functions on graphics cards, for example, the `expf()` function can be mapped to the hardware accelerated `__expf()` function in CUDA. Although this feature is supported by our source-to-source compiler, we do not use it within this paper to select faster, hardware accelerated intrinsic functions instead of the standard functions.

B. Optimization Selection

The knowledge we get from our micro-benchmarks as well as those from external ones are stored in a database that is utilized by the source-to-source compiler to decide what optimization should be applied for which a) target hardware and b) backend. This includes the amount of padding required for optimal memory bandwidth utilization, whether texture memory is beneficial, or whether constant memory should be initialized statically or dynamically. When the source-to-source compiler is invoked, a compiler flag is used to indicate if CUDA or OpenCL low-level code should be generated. The target graphics card hardware is specified the same way. Currently, the compiler database contains information about all available CUDA-capable graphics cards as specified by the *compute capability* and AMD GPUs of the Radeon HD 6900 and HD 5800 series (VLIW4 and VLIW5 architecture).

C. Automatic Configuration Selection

One common problem when writing CUDA and OpenCL programs is to select an appropriate kernel configuration, which defines the number of threads that are mapped to one SIMD unit. There are many possible configurations and moving on to another device, a different configuration will perform better. Even worse, some configurations for one device will not run on a second device at all due to different hardware capabilities or configuration limitations imposed by the hardware vendor. For example, on graphics cards from AMD, the maximal number of threads that can be mapped to one SIMD unit is 256, while this limit is either 512, 768, or 1024 on graphics cards from NVIDIA. Even more important are the hardware limitations in terms of available registers and shared/local memory per SIMD unit. Selecting a configuration that allocates more resources than available results in a kernel launch error at run-time.

Therefore, our source-to-source compiler passes the generated CUDA and OpenCL code to the *nvcc* compiler and a tool invoking the OpenCL run-time, respectively. These generate machine-specific assembly code and provide the resource usage information of kernels. This information is combined with a hardware model of the target GPU, describing a) the SIMD width, b) the maximal thread configuration as specified by the programmer, c) the maximal threads that can be mapped to a SIMD unit, and d) the maximal available registers and shared memory as well as their allocation strategy. This allows the compiler not only to select configurations that are valid, but also those that increase the utilization of available resources (also called *occupancy*) in order to hide instruction and global memory latency. To do so, our source-to-source compiler calculates for all configurations that result in good memory bandwidth utilization (i. e., which are a multiple of the SIMD width to get coalesced memory accesses) their occupancy and remembers them.

From these configuration-occupancy pairs, the source-to-source compiler selects a configuration, determines a 2D-tiling for it, and sets it as kernel configuration based on a heuristic as described in Algorithm 2: first, all invalid configurations are removed, and the remaining configurations are sorted by descending occupancy. If no border handling code was generated, we use the configuration with the highest occupancy and give precedence to the x-component for tiling, that is, we get 1D-configurations like 128×1 or 256×1. In case multiple configurations have the same occupancy, the one with the lowest number of threads is chosen. Such configurations are typically selected by expert programmers and yield good performance for most kernels. In case border handling code was generated by our source-to-source compiler, the tiling strategy is different. Here, the heuristic tries to minimize the number of threads that execute code with conditionals for boundary handling. Since the boundary handling is typically symmetric (e. g., 3×3 or 5×5), the minimal size for the x-configuration of the SIMD width is in most cases sufficient and the y-configuration is preferred instead. When more than one configuration is available with high occupancy, we select the one that minimizes the number of threads with boundary handling conditionals (e. g., we prefer a configuration of 32×6 over 32×4 for a window size of 13×13, a configuration of 32×3, however, would be preferred to the two aforementioned

configurations.).

Once an optimal configuration 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. In case of border handling, the final kernel code is generated after the kernel configuration and tiling are determined. This is necessary, since the constants in Listing 8 that determine what code to execute by which block relies on the tiling of the kernel configuration. The initial kernel code that is used to determine the resource usage uses default constants.

Algorithm 2: Heuristic for selecting optimal kernel configuration and tiling depending on resource usage, border handling size, and target graphics card.

Input: Kernel K, set of configurations C for GPU G
Output: Optimal configuration c_{opt} and tiling t_{opt} for kernel K

```
1  C ← configurations of C multiple of SIMD width of G;
2  C ← configurations of C within resource limitations of G;
3  C ← sorted configurations of C with descending occupancy and
   ascending number of threads;
4  if border handling then
5      c_opt ← first configuration of C;
6      t_opt ← tiling of c_opt, prefer y over x;
7      threads_bh_opt ← calculate number of threads for border
                        handling for tiling t_opt for kernel K;
8      C' ← configurations with highest occupancy;
9      foreach configuration c of C' do
10         t ← tiling of c, prefer y over x;
11         threads_bh ← calculate number of threads for border
                        handling for tiling t for kernel K;
12         if threads_bh < threads_bh_opt then
13             c_opt ← c;
14             t_opt ← t;
15             threads_bh_opt ← threads_bh;
16         end
17     end
18 else
19     c_opt ← first configuration of C;
20     t_opt ← tiling of c_opt, prefer x over y;
21 end
```

D. Configuration Exploration

To evaluate our heuristic that selects kernel configurations and tiling, our source-to-source compiler can generate code that explores all possible configurations for a given kernel. This allows to assess the heuristic easily. Therefore, the source-to-source compiler replaces the kernel preparation (setting arguments and configuration, binding textures, etc.) and invocation by two nested loops iterating over all valid configurations for the current kernel. In this mode, the constants that determine what code is executed by which block are replaced by macros. The macros are set at run-time when the kernel source code is compiled just-in-time. Since CUDA does not support just-in-time compilation, our run-time library provides a wrapper to *nvcc* to allow this.

VI. Evaluation and Results

In this section, the source-to-source compiler and the device-specific code generation is evaluated and compared against other approaches. Of particular interest is the support for boundary handling and the efficient implementation in different frameworks as well as the support for filter masks.

A. Boundary Handling, Filter Masks

First, we compare the code that is generated from our description with manual implementations and then, with those from RapidMind [10] and OpenCV [11].

1) Manual Implementation: The basic version of the manual implementations uses straightforward CUDA/OpenCL code. These versions are then subsequently improved to utilize linear texture memory in CUDA (image objects in OpenCL), constant memory to store the filter masks, and combinations of both. There are two versions using texture memory, one using only the texturing hardware to read data (called Tex/Img), and one which uses the texturing hardware for boundary handling (called Tex2D/ImgBH). Both OpenCL versions use image objects to store images. The CUDA version using texturing hardware to read data uses linear memory while the second version using textures for boundary handling relies on 2D arrays, which support boundary handling in two dimensions. Using texturing hardware for boundary handling, the following boundary handling modes are supported in CUDA: a) Clamp and b) Repeat. Using OpenCL the following modes are supported: a) Clamp, b) Repeat, and c) Constant. However, the constants can be only floating point values of either 0.0 or 1.0.

The different implementation variants are summarized in Table II (Tesla C2050, CUDA), III (Tesla C2050, OpenCL), IV (Quadro FX 5800, CUDA), V (Quadro FX 5800, OpenCL), VI (Radeon HD 5870, OpenCL), and VII (Radeon HD 6970, OpenCL). It can be seen in particular that our generated CUDA code outperforms even the code that utilizes texturing hardware for boundary handling. The difference between the boundary handling modes is small, while the performance of the manual implementation varies significantly (up to a factor of two). Note that on the Tesla hardware, the implementations accessing unallocated memory (undefined boundary handling) crash. The benefit of texturing hardware in OpenCL is not present anymore since no linear memory can be used (which is of higher performance compared to arrays for the considered applications). On GPUs from AMD, the different optimizations have varying impact on performance and is not predictable. We attribute this to the current implementations, which are scalar and do not utilize the VLIW4 or VLIW5 hardware architecture. We expect to get a more predictable behavior once vector operations are used (see also Section VIII).

2) RapidMind: RapidMind, a multi-core development platform allows to describe the considered filter in the same way as our framework. Actually, the kernel description differs only by the keywords and data types used in our DSL and in RapidMind [10]. In Table II and IV, the performance of the RapidMind implementation is listed in addition to the manual implementations and the generated code for the graphics cards

Table II: Execution times in *ms* for the bilateral filter on a **Tesla C2050** using the **CUDA** backend with manual border handling implementation for an image of 4096×4096 pixels and a filter window size of 13×13 ($\sigma_d = 3$). Kernel configuration is 128×1 for all kernels.

	Undef.	Clamp	Repeat	Mirror	Const.
Manual	crash	302.27	363.96	321.81	568.46
+Tex	260.03	285.61	362.70	310.61	520.25
+2DTex	272.39	272.40	300.56	n/a	n/a
+Mask	crash	214.51	281.89	225.88	481.76
+Mask+Tex	170.79	192.46	259.26	205.29	425.13
+Mask+2DTex	181.19	181.19	203.13	n/a	n/a
Generated	crash	285.29	298.29	289.22	291.26
+Tex	276.76	265.36	285.57	278.04	268.01
+Mask	crash	181.45	200.66	193.16	197.23
+Mask+Tex	172.60	182.80	180.38	173.59	175.52
RapidMind	430.95	489.94	crash	n/a	539.69
+Tex	456.35	514.63	crash	n/a	518.49

Table III: Execution times in *ms* for the bilateral filter on a **Tesla C2050** using the **OpenCL** backend with manual border handling implementation for an image of 4096×4096 pixels and a filter window size of 13×13 ($\sigma_d = 3$). Kernel configuration is 128×1 for all kernels.

	Undef.	Clamp	Repeat	Mirror	Const.
Manual	449.86	485.60	552.83	504.39	505.11
+Img	465.48	487.80	557.88	501.18	508.28
+ImgBH	452.15	452.39	464.07	n/a	452.24
+Mask	215.23	250.67	331.11	261.05	267.62
+Mask+Img	228.29	251.51	322.61	264.54	288.08
+Mask+ImgBH	214.68	227.74	215.07	n/a	215.07
Generated	453.78	466.49	474.86	455.59	467.05
+Img	463.62	466.61	472.67	468.43	466.62
+Mask	217.95	215.61	222.78	220.27	220.16
+Mask+Img	219.49	219.64	238.81	220.28	232.57

supporting CUDA. It can be seen that our generated code outperforms the one of RapidMind by a factor of two, even without boundary handling. Just-in-time compilation overhead is not included in the timing of RapidMind and was verified using the CUDA profiler. Using Repeat as boundary handling, the program crashes on the Tesla and is by a factor of three slower on the Quadro compared to the variant with undefined boundary handling.

3) OpenCV: The *Open Source Computer Vision* (OpenCV) library [11] is one widely used library for image processing and provides support for GPU accelerators using CUDA. Previously, we have shown that we can generate efficient code for point operators that outperforms the implementations of OpenCV (which did not support boundary handling and are wrapped calls to the NVIDIA Performance Primitives (NPP) library) [4]. Since then, OpenCV added low-level CUDA implementations for row-based and column-based (separable) kernels like Gaussian and Sobel filters. Their implementation stages image data to shared memory and utilizes precalculated

Table IV: Execution times in *ms* for the bilateral filter on a **Quadro FX 5800** using the **CUDA** backend with manual border handling implementation for an image of 4096×4096 pixels and a filter window size of 13×13 ($\sigma_d = 3$). Kernel configuration is 128×1 for all kernels.

	Undef.	Clamp	Repeat	Mirror	Const.
Manual	319.67	349.32	394.96	393.00	779.68
+Tex	310.22	336.46	369.74	378.47	590.18
+2DTex	330.50	330.49	369.06	n/a	n/a
+Mask	224.56	321.55	323.50	321.46	778.48
+Mask+Tex	199.11	237.60	271.45	278.89	497.75
+Mask+2DTex	214.53	215.53	348.92	n/a	n/a
Generated	321.24	331.36	404.81	332.17	436.77
+Tex	312.71	313.74	356.52	316.08	383.19
+Mask	225.58	227.65	281.82	228.18	290.78
+Mask+Tex	200.55	204.45	218.22	204.53	246.96
RapidMind	737.69	862.86	2352.34	n/a	989.55
+Tex	679.52	734.48	2226.33	n/a	805.62

Table V: Execution times in *ms* for the bilateral filter on a **Quadro FX 5800** using the **OpenCL** backend with manual border handling implementation for an image of 4096×4096 pixels and a filter window size of 13×13 ($\sigma_d = 3$). Kernel configuration is 128×1 for all kernels.

	Undef.	Clamp	Repeat	Mirror	Const.
Manual	439.55	504.79	537.04	528.47	770.34
+Img	509.95	529.39	560.77	550.43	732.55
+ImgBH	509.82	509.33	509.38	n/a	509.65
+Mask	355.70	455.69	458.90	452.71	775.83
+Mask+Img	468.94	466.67	467.19	464.62	708.93
+Mask+ImgBH	468.00	470.04	468.80	n/a	470.46
Generated	446.24	449.67	514.89	453.68	460.68
+Img	511.38	512.50	553.23	511.78	654.08
+Mask	354.93	357.77	407.01	357.72	384.30
+Mask+Img	466.26	465.70	522.53	461.56	539.77

Table VI: Execution times in *ms* for the bilateral filter on a **Radeon HD 5870** using the **OpenCL** backend with manual border handling implementation for an image of 4096×4096 pixels and a filter window size of 13×13 ($\sigma_d = 3$). Kernel configuration is 128×1 for all kernels.

	Undef.	Clamp	Repeat	Mirror	Const.
Manual	334.96	408.36	404.83	419.59	440.64
+Img	353.93	385.23	405.81	396.45	484.25
+ImgBH	353.93	353.91	353.96	n/a	353.95
+Mask	311.85	397.40	434.36	408.32	402.59
+Mask+Img	341.23	373.93	400.71	375.48	444.36
+Mask+ImgBH	341.25	341.24	341.24	n/a	341.27
Generated	342.67	354.49	472.20	355.57	351.83
+Img	372.14	376.91	482.28	382.71	446.98
+Mask	326.22	357.96	487.53	359.72	348.77
+Mask+Img	350.56	364.34	481.76	364.39	428.22

Table VII: Execution times in *ms* for the bilateral filter on a **Radeon HD 6970** using the **OpenCL** backend with manual border handling implementation for an image of 4096×4096 pixels and a filter window size of 13×13 ($\sigma_d = 3$). Kernel configuration is 128×1 for all kernels.

	Undef.	Clamp	Repeat	Mirror	Const.
Manual	286.29	337.13	375.11	346.18	381.76
+Img	286.38	319.20	364.59	328.12	435.16
+ImgBH	286.44	286.44	286.43	n/a	286.46
+Mask	265.57	332.41	387.81	340.59	349.37
+Mask+Img	268.26	310.84	349.31	311.42	387.73
+Mask+ImgBH	268.20	268.23	268.20	n/a	268.24
Generated	291.30	309.52	470.90	322.69	321.19
+Img	303.36	298.50	465.30	305.38	438.74
+Mask	289.33	296.20	467.76	332.91	314.05
+Mask+Img	279.66	291.49	474.60	291.58	414.31

masks. In addition, OpenCV maps multiple output pixels to the same thread on the GPU in order to minimize scheduling overheads and maximize data reuse. Here, we investigate two kernels implemented in such a way in OpenCV, namely the Gaussian and Sobel filter, and compare them against our generated code with boundary handling. Table VIII (Tesla C2050) and IX (Quadro FX 5800) show the results for different boundary handling modes for the Gaussian filter (note that the Sobel filter uses the same implementation and has the same performance). For OpenCV, two implementations are added: one with the original implementation, mapping eight output pixels to one thread (PPT=8), and one with a one-to-one mapping (PPT=1). Mapping multiple output pixels to the same thread gives a significant performance boost to the OpenCV implementation, but the performance varies a lot—depending on the boundary handling mode. In contrast, the execution times of the different boundary handling modes is constant for the code generated by our framework. The performance of the generated code is about as fast as the OpenCV implementation using the simple one-to-one mapping. All our implementations use constant memory for the mask coefficients and automatic kernel configuration as determined by our framework. Texture memory has only marginal benefits for these kernels and scratchpad memory slows the kernel down for reasons mentioned earlier.

B. Configuration Exploration

To verify that the heuristic presented in Algorithm 2 for automatic kernel configuration selects a good configuration and tiling, we generate code for the bilateral filter using the CUDA backend on the Tesla C2050 that explores all valid configurations and visualize them in Figure 4 (note that the configuration with 32 threads with an execution time of 425 ms is not shown). Multiple points with the same number of threads denote a different tiling for that configuration. The configuration selected by our framework, 32×6, is in this case also the optimal configuration. This is not always the case, but the configurations selected by our heuristic are typically within 10 % of the best configuration.

Table VIII: Execution times in *ms* for the Gaussian filters from OpenCV on the **Tesla C2050** and our generated implementations using the **CUDA** and **OpenCL** backends for an image of 4096×4096 pixels and different filter window sizes.

Gaussian: 3×3				
	Clamp	Repeat	Mirror	Const.
OpenCV: PPT=8	5.10	6.36	8.09	6.75
OpenCV: PPT=1	9.44	11.85	15.97	12.36
CUDA(Gen)	7.00	7.53	7.21	7.10
CUDA(+Tex)	7.00	7.44	7.17	7.13
CUDA(+Smem)	7.73	8.09	8.02	8.00
OpenCL(Gen)	9.26	9.70	9.40	9.33
OpenCL(+Tex)	13.41	13.62	13.33	13.16
OpenCL(+Lmem)	11.29	11.46	11.12	11.13
Gaussian: 5×5				
	Clamp	Repeat	Mirror	Const.
OpenCV: PPT=8	5.11	6.36	8.10	6.76
OpenCV: PPT=1	9.45	11.88	15.99	12.37
CUDA(Gen)	8.84	9.86	9.47	9.45
CUDA(+Tex)	8.94	9.72	9.35	9.47
CUDA(+Smem)	9.38	9.59	9.44	9.55
OpenCL(Gen)	10.88	11.82	11.13	10.44
OpenCL(+Tex)	14.96	15.87	15.17	15.12
OpenCL(+Lmem)	13.24	13.72	13.35	13.22

Table IX: Execution times in *ms* for the Gaussian filters from OpenCV on the **Quadro FX 5800** and our generated implementations using the **CUDA** and **OpenCL** backends for an image of 4096×4096 pixels and different filter window sizes.

Gaussian: 3×3				
	Clamp	Repeat	Mirror	Const.
OpenCV: PPT=8	4.86	5.82	10.46	6.22
OpenCV: PPT=1	7.63	9.22	20.98	9.79
CUDA(Gen)	8.60	8.63	8.64	8.67
CUDA(+Tex)	8.55	8.58	8.60	8.63
CUDA(+Smem)	11.83	11.83	11.84	11.90
OpenCL(Gen)	13.58	13.47	13.10	13.46
OpenCL(+Img)	15.42	15.47	15.06	15.24
OpenCL(+Lmem)	17.84	17.86	17.91	18.35
Gaussian: 5×5				
	Clamp	Repeat	Mirror	Const.
OpenCV: PPT=8	4.90	5.87	10.45	6.22
OpenCV: PPT=1	7.64	9.22	20.98	9.79
CUDA(Gen)	9.88	9.95	9.95	10.12
CUDA(+Tex)	9.91	9.97	9.98	10.20
CUDA(+Smem)	14.36	14.36	14.37	14.43
OpenCL(Gen)	16.14	16.26	16.18	16.60
OpenCL(+Img)	18.38	18.44	18.33	18.65
OpenCL(+Lmem)	23.61	23.62	23.62	24.13

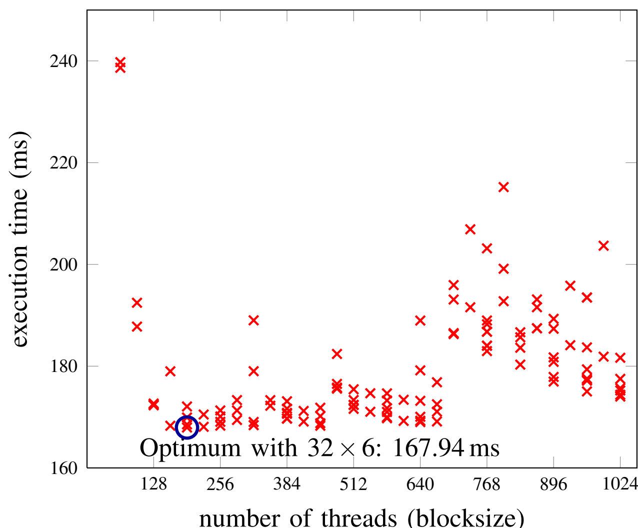

Figure 4: Configuration space exploration for the bilateral filter (filter window size: 13×13) for an image of 4096×4096 on the Tesla C2050. Shown are the execution times for processing the bilateral filter in dependence on the blocksize.

C. Discussion

Although the presented framework focuses on programmability for domain experts in medical imaging, it offers decent performance on GPUs from different manufacturers. The domain experts can express algorithms in a high-level language tailored to their domain. This allows high productivity and the mapping to different target hardware platforms from the same algorithm description. The development time for the manual implementations is in the range of several days (for non-GPU experts even weeks), while the DSL description takes only a couple of minutes.

For example, the source-to-source compiler generates a CUDA kernel with 317 lines of code for the kernel description shown in Listing 5 (16 lines of code). This comes from 9 different kernel implementations for the top, top-right, right, etc. image borders plus index adjustments for boundary handling. In addition, the generated code depends on the filter window size and image size. Writing such code by hand is often error-prone and tedious.

VII. Related Work

The work most close to ours is the RapidMind multi-core development platform [10] targeting standard multi-core processors as well as accelerators like the Cell B.E. and GPUs. The RapidMind technology is based on Sh [12], a high-level metaprogramming language for graphics cards. RapidMind provides its own data types that can be arranged in multi-dimensional arrays. Accessors can be used to define boundary handling properties of the underlying data. A two-dimensional array in RapidMind corresponds to an *Image* object in our framework. In addition to the boundary handling modes supported in RapidMind, we support also mirroring at the image border, a widely used boundary handling mode in medical imaging. Programs that operate on arrays are identified by special keywords in RapidMind, while we use compiler-known C++-classes to express image processing kernels. Within a RapidMind program, neighboring elements can be accessed using the *shift()* method on input data. Since there are no details on code generation for border handling publicly available for RapidMind, we can compare our approach only quantitatively with the one of RapidMind.

In 2009, Intel acquired RapidMind and incorporated the RapidMind technology into Intel ArBB (Array Building Blocks) [13]. Since then, RapidMind is discontinued as is Sh. 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. In ArBB, only a constant value is returned when arrays are accessed out of bounds. To access neighboring elements, the current processed element and the offset is passed to the *neighbor()* function. Using the *position()* function, the position within the n-dimensional iteration space can be retrieved and used to implement more sophisticated boundary handling modes. However, this comes along with large overheads on GPUs that we remedy by exploiting multiple levels of parallelism in our code-generation backend. When merging RapidMind technology with Intel's Ct, the backend for graphics cards was dropped and is not supported anymore.

Beside language based frameworks, there exist library based frameworks like OpenCV [11] and the NVIDIA Performance Primitives (NPP) library. These libraries allow to use predefined kernels. However, to offload new algorithms that are not available, low-level code has to be written.

Other compiler based approaches allow also to offload code to GPU accelerators. The input to such compilers is typically sequential C or basic CUDA as well as annotations describing transformations applied to the code. Examples are HMPP Workbench [14], PGI Accelerator [15], *hi*CUDA [16], and *CUDA-lite* [17], just to name a few. In order to obtain a decent performance using these compiler based approaches, the programmer has to know what compiler transformations can be applied and how to rewrite code to make such transformations possible. Algorithm designers and domain experts, however, have only little knowledge of the underlying hardware and compiler transformations. As a consequence, the full potential of such frameworks is only rarely exploited. In the proposed DSL, however, the required metadata is implicitly given by the DSL syntax and has not to be specified separately.

Our framework is most similar in spirit to Cornwall *et al.*'s work on indexed metadata for visual effects [18], but introduces additional device-specific optimizations such as global memory padding for memory coalescing, support for boundary handling, and the heuristic for automatic kernel configuration selection.

The main contributions of this work include a) a domain-specific description of local operators in medical imaging, b) a new code generation framework that utilizes a two-layered parallelization approach exploiting both SPMD and MPMD parallelism on current graphics cards architectures, and c)

a heuristic for automatic kernel configuration selection and tiling. The presented approach is not limited to algorithms stemming from the medical domain, but can be also utilized for other application domains, in particular the two-layered parallelization approach.

VIII. Outlook

The current compiler optimizations for local operators can be further extended to unroll the loops of convolutions and to propagate the constants of the filter masks. To do so, we defined a syntax using lambda-functions as seen in Listing 9. However, Clang, on which our source-to-source compiler is based, does not yet support lambda-functions. As soon as this support is available, we will also be able to support constant propagation and loop unrolling. For global operators, we look for a similar syntax that allows the programmer to define operations that merge/reduce two pixels.

```
1 void kernel() {
2   output() = convolve(cMask, SUM, [&] () {
3     return cMask()*Input(cMask);
4     });
5 }
```

Listing 9: Using a lambda-function and the framework-provided *convolve* function to express convolution kernels.

Furthermore, we are looking into vectorization for graphics cards from AMD so that the impact of the transformations described in this paper is visible. First manual vectorization shows that the performance improves significantly on graphics cards from AMD.

IX. Conclusions

We presented a domain-specific description for local operators in medical imaging and the efficient mapping to low-level CUDA and OpenCL code. Based on the metadata provided by the programmer, a two-layered parallel code utilizing SPMD and MPMD parallelism is generated. Using this approach, we showed that we can generate code for boundary handling that has constant performance independent from the selected boundary handling mode while the performance of other solutions varies significantly. Filter masks are stored to constant memory to avoid unnecessary recalculations. To determine a good configuration for the generated kernels, we presented a heuristic that takes boundary handling metadata, the resource usage of kernels, as well as hardware capabilities and limitations into account. The resulting kernel configuration and tiling minimizes the number of threads executing code for boundary handling. Also, the generated code by our framework is typically even faster than manual implementations and those relying on hardware support for boundary handling. In an experimental analysis, we outperform even implementations from RapidMind, a commercial framework for multi-core and GPU programming, and get similar results to the GPU backend of the widely used image processing library OpenCV. The presented HIPAcc framework is available as open-source under https://sourceforge.net/projects/hipacc.

Acknowledgments

We thank Anton Lokhmotov and anonymous reviewers for their helpful comments.

References

[1] S. Ryoo, C. Rodrigues, S. Stone, J. Stratton, S. Ueng, S. Baghsorkhi, and W. Hwu, “Program Optimization Carving for GPU Computing,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1389–1401, Oct. 2008.

[2] 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*, 2011.

[3] L. Howes, A. Lokhmotov, A. Donaldson, and P. Kelly, “Towards Metaprogramming for Parallel Systems on a Chip,” in *Proceedings of the 3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*. Springer, Aug. 2009, pp. 36–45.

[4] R. Membarth, A. Lokhmotov, and J. Teich, “Generating GPU Code from a High-level Representation for Image Processing Kernels,” in *Proceedings of the 5th Workshop on Highly Parallel Processing on a Chip (HPPC)*. Springer, Aug. 2011.

[5] Clang, “Clang: A C Language Family Frontend for LLVM,” http://clang.llvm.org, 2007–2012.

[6] C. Tomasi and R. Manduchi, “Bilateral Filtering for Gray and Color Images.” IEEE Computer Society, Jan. 1998, pp. 839–846.

[7] D. Kunz, K. Eck, H. Fillbrandt, and T. Aach, “Nonlinear Multiresolution Gradient Adaptive Filter for Medical Images,” in *Proceedings of SPIE Medical Imaging 2003: Image Processing*, vol. 5032. SPIE, Feb. 2003, pp. 732–742.

[8] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying GPU Microarchitecture through Microbenchmarking,” in *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2010, pp. 235–246.

[9] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer, “Automatic OpenCL Device Characterization: Guiding Optimized Kernel Design,” in *Proceedings of the 17th International European Conference on Parallel and Distributed Computing (Euro-Par)*. Springer, Aug. 2011, pp. 438–452.

[10] RapidMind, *RapidMind Development Platform Documentation*, RapidMind Inc., Jun. 2009.

[11] Willow Garage, “Open Source Computer Vision (OpenCV),” http://opencv.willowgarage.com/wiki, 1999–2012.

[12] 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, Aug. 2004.

[13] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Du Toit, Z. Wang, Z. 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 *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Apr. 2011, pp. 224–235.

[14] R. Dolbeau, S. Bihan, and F. Bodin, “HMPP: A Hybrid Multi-core Parallel Programming Environment,” in *Proceedings of the 1st Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, Oct. 2007.

[15] M. Wolfe, “Implementing the PGI Accelerator Model,” in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*. ACM, Mar. 2010, pp. 43–50.

[16] T. Han and T. Abdelrahman, “*hi*CUDA: High-level GPGPU Programming,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 78–90, Jan. 2011.

[17] S. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu, “CUDA-lite: Reducing GPU Programming Complexity,” *Languages and Compilers for Parallel Computing*, vol. 5335, pp. 1–15, 2008.

[18] J. Cornwall, L. Howes, P. Kelly, P. Parsonage, and B. Nicoletti, “High-Performance SIMT Code Generation in an Active Visual Effects Library,” in *Proceedings of the 6th ACM Conference on Computing Frontiers (CF)*. ACM, May 2009, pp. 175–184.