Towards a Performance-portable Description of Geometric Multigrid Algorithms using a Domain-specific Language

Richard Membarth*

German Research Center for Artificial Intelligence, Germany.
Computer Graphics Lab & Intel Visual Computing Institute,
Saarland University, Germany.

Oliver Reiche, Christian Schmitt,
Frank Hannig, Jürgen Teich

Hardware/Software Co-Design,
Department of Computer Science,
University of Erlangen-Nuremberg, Germany.

Markus Stürmer, Harald Köstler

System Simulation,
Department of Computer Science,
University of Erlangen-Nuremberg, Germany.

Abstract

High Performance Computing (HPC) systems are nowadays more and more heterogeneous. Different processor types can be found on a single node including accelerators such as Graphics Processing Units (GPUs). To cope with the challenge of programming such complex systems, this work presents a domain-specific approach to automatically generate code tailored to different processor types. Low-level CUDA and OpenCL code is generated from a high-level description of an algorithm specified in a Domain-Specific Lan-

*Corresponding author

Email addresses: `richard.membarth@dfki.de` (Richard Membarth), `oliver.reiche@cs.fau.de` (Oliver Reiche), `christian.schmitt@cs.fau.de` (Christian Schmitt), `hannig@cs.fau.de` (Frank Hannig), `teich@cs.fau.de` (Jürgen Teich), `markus.stuermer@cs.fau.de` (Markus Stürmer), `harald.koestler@cs.fau.de` (Harald Köstler)

Author manuscript: Journal of Parallel and Distributed Computing (JPDC), 74(12), pp. 3191–3201, 2014.
DOI: 10.1016/j.jpdc.2014.08.008 *December 1, 2014*

guage (DSL) instead of writing hand-tuned code for GPU accelerators. The DSL is part of the Heterogeneous Image Processing Acceleration (HIPA^{cc}) framework and was extended in this work to handle grid hierarchies in order to model different cycle types. Language constructs are introduced to process and represent data at different resolutions. This allows to describe image processing algorithms that work on image pyramids as well as multigrid methods in the stencil domain. By decoupling the algorithm from its schedule, the proposed approach allows to generate efficient stencil code implementations. Our results show that similar performance compared to hand-tuned codes can be achieved.

Keywords: multigrid, multiresolution, image pyramid, domain-specific language, stencil codes, code generation, GPU, CUDA, OpenCL

1. Introduction

Mapping algorithms in an efficient way to the target hardware poses challenges for algorithm designers. This is in particular true for heterogeneous systems hosting accelerators like graphics cards. While algorithm developers have profound knowledge of the application domain, they often lack detailed insight into the underlying hardware of accelerators in order to exploit the provided processing power. To tackle this problem, OpenCL[1], a new industry-backed standard Application Programming Interface (API) that inherits many traits from CUDA, was introduced in order 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 [1].

In this paper, a different approach is taken by decoupling algorithms from their schedule in a DSL. This allows to map algorithms efficiently to a target platform. The DSL is part of the HIPA^{cc} framework [2] that provides also a source-to-source compiler to translate not only the high-level algorithm description into low-level CUDA/OpenCL code, but also to apply transformations and optimizations on the code. To describe algorithms that

[1]`http://www.khronos.org/opencl`

work on multiple resolutions of the same data, image pyramids are used in image processing [3] and multigrids for stencil computations [4, 5]. We present in this work a concise syntax for creating such multiresolution data structures and for processing the data on different resolutions. Kernels described in HIPA^{cc} remain unchanged, only a schedule for iterating over the grid has to be specified.

We consider image pyramid construction and High Dynamic Range (HDR) compression of 2D images as example applications. HDR compression can be done efficiently in the gradient space. For it, the image has to be transformed to gradient space and back. While the forward transformation to gradient space is fast by using simple finite differences, the backward transformation requires the solution of a Partial Differential Equation (PDE).

Multigrid methods are one of the most efficient numerical methods to solve large, sparse linear systems arising for example when discretizing elliptic PDEs. Elliptic PDEs are used to model various physical or technical effects in many application fields. One of the most popular elliptic PDEs is the Poisson equation in order to model diffusion processes. In this work we consider a simple multigrid solver in 2D that employs stencil codes for the Poisson equation and apply it to image processing. In previous work we developed a suitable parallel multigrid algorithm and provided a hand-tuned implementation for this task [6]. Our first description of this multigrid solver in HIPA^{cc} had several drawbacks: images had to be provided for each level and the (mostly identical) computation had to be described repeatedly for each level [7]. Here, for concise modeling of multigrid algorithms, we introduce a suitable representation for multigrid and multiresolution data sets in the DSL as well as a concise syntax for describing the operations on each multigrid level and between different multigrid levels.

The focus of this work is on the concise description of algorithms that operate on different resolutions of the same data:

- We present language constructs in our DSL that allow to describe image pyramids. Data for pyramids is managed by the framework and only the data for the finest level has to be provided. Furthermore, we allow to specify how the pyramid is traversed: this includes typical traversals for construction of pyramids in image processing as well as the V-cycle and W-cycle for multigrid stencil computations.

- We evaluate the implementation of image pyramid construction and of a multigrid application using our image pyramid representation. We show that the proposed representation improves productivity significantly. Furthermore, we show that the description in HIPA^{cc} provides portability across different architectures and allows to achieve competitive performance compared to Halide [8] as well as hand-tuned implementations.

The paper first introduces related work on DSLs and frameworks for stencil codes. Then, an overview of the HDR compression application and the used multigrid solver is given. The HIPA^{cc} framework and its extensions used to model multigrid algorithms in its DSL is introduced thereafter. The paper concludes with an evaluation of the domain-specific approach for stencil codes including productivity, portability, and performance aspects.

2. Related Work

In the past, several approaches captured and used knowledge about the domain of stencil codes and their applications in the form of domain-specific languages. The idea is to provide abstractions within the language that are tailored to the domain of stencil-code engineering.

Liszt [9] and Pochoir [10] are stencil compilers for code written in a simple domain-specific language. Pochoir compiles to C++ with Cilk++ extensions and fits the optimized stencil code into a generic, divide-and-conquer template. Pochoir pays particular attention to cache obliviousness on multi-core workstations. However, both languages (Liszt and Pochoir) provide only limited support for the characteristics of the hardware platform.

The hypre library [11] is a collection of high-performance preconditioners and solvers for large sparse linear systems of equations on massively parallel machines. It offers, for example, a stencil-based interface for computations on structured or block-structured grids and also incorporates different multigrid solvers. DUNE [12] is a modular and generic C++ library for the solution of partial differential equations on different kinds of grids. It supports structured or block-structured grids and a variety of algebraic solvers including multigrid is provided as external modules. Both libraries, hypre and DUNE, are flexible and can easily be adapted for stencil applications, but there are neither a domain-specific syntax nor proper editing and debugging facilities, and the stencil code has to be optimized by setting configuration options by hand. There is no specialized syntax for the definition of multigrid solvers. Setting parameters such as the number of pre- and postsmoothing steps and the cycle type is done via functions. Custom cycle types or user-defined restriction and interpolation operators are not supported. All three frameworks provide no native options for execution on GPUs.

PATUS (Parallel Auto-Tuned Stencils) [13, 14] is a code generation and auto-tuning framework for stencil computations on shared-memory architectures. The algorithms and stencils are provided by the user and *strategies* can be specified that define parallelization and optimization.

The parallel Optimized Sparse Kernel Interface (pOSKI) [15] is a collection of algorithms for operations involving sparse matrices on uniprocessor and multi-core machines. It includes auto-tuning at installation- and run-time and is suitable for stencil computations yielding special sparse matrices.

Physis[16] provides a DSL for stencil computations based on C with support for GPU accelerators. They hide communication cost by overlapping boundary exchange with stencil computation.

[17] introduces a small DSL for Jacobi-like iterative methods. Efficient code is generated for GPU accelerators by using overlapping tiles for multiple iterations.

Ypnos [18] and Paraiso [19] provide a functional DSL embedded in Haskell for structured grid computations with support for GPU accelerators. Similarly, Halide [8] uses a functional representation to describe image processing algorithms and stencil codes. The programmer then specifies a schedule in Halide for a pipeline of computations separately. This gives the programmer the flexibility to reuse the same algorithm description for different target architectures by specifying target-specific schedules.

In [20] a graphical DSL based on UML activity diagrams is proposed to model multigrid algorithms for applications in variational imaging.

While these frameworks allow to model stencils and to generate efficient code, the stencils are limited to a single level in most cases. Halide [8] and the UML activity diagrams [20] are the only of the aforementioned approaches that allow users to define custom multigrid algorithms to the best of our knowledge. In [20], the data-flow between kernels is modeled and multiple levels can be modeled through a cycle. The implementation of each UML component is provided by the user and, hence, arbitrary computation can be expressed. Halide, in contrast, allows to define arbitrary control flow. For multigrid applications, a loop iterating over the different levels can be used and the results of each level can be stored to an array.

An alternative approach to code optimization and generation in stencil-code engineering is the use of the polyhedron model [21]. While a DSL captures the required knowledge for optimizations and transformations in its syntax, the polyhedron model relies on code analysis to extract this information. There exist many polyhedron-based approaches for stencil codes and accelerators such as GPUs are well supported (e .g., [22, 23]). However, the analysis fails often for irregular or non-affine patterns.

3. Application

In order to do imaging in the gradient domain as described, for example, in [24] we first compute the gradient $\nabla I = \begin{pmatrix} I_x \\ I_y \end{pmatrix}$ of an image $I : \Omega \mapsto \mathbb{R}$ defined in the rectangular image domain $\Omega \subset \mathbb{R}^2$.

After that we manipulate the gradient by an attenuating function Φ

$$\Phi(\nabla I) = \frac{\alpha}{\|\nabla I\|} \left(\frac{\|\nabla I\|}{\alpha} \right)^{\beta} \tag{1}$$

where the first parameter α determines which gradient magnitudes are left unchanged, and the second parameter $\beta < 1$ is the attenuating factor of the larger gradients.

The HDR compressed image $u : \Omega \mapsto \mathbb{R}$ is then reconstructed by minimizing the energy functional

$$\int_{\Omega} \|\nabla u - \Phi(\nabla I)\|^2 d\Omega. \tag{2}$$

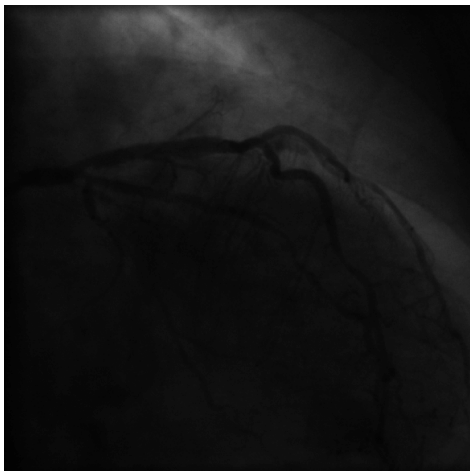
A minimizer has to satisfy the Euler-Lagrange equation

$$\nabla^2 u = \operatorname{div} \Phi(\nabla I). \tag{3}$$

Setting $f = \operatorname{div} \Phi(\nabla I)$, we have to solve the PDE

$$\begin{aligned} \Delta u &= f \quad \text{in } \Omega && \text{(4a)} \\ u &= 0 \quad \text{on } \partial\Omega && \text{(4b)} \end{aligned}$$

where we assume Dirichlet boundary conditions. An example for HDR compression is shown in Figure 1.


(a) Input.

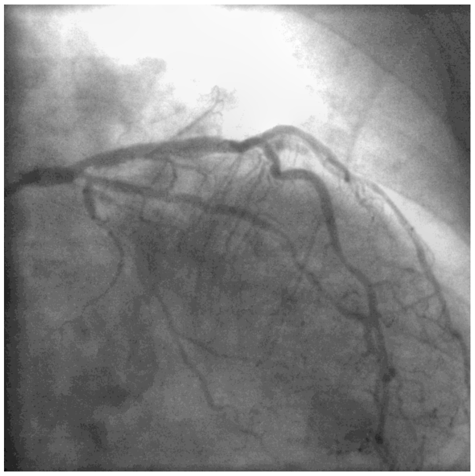

(b) Output.

Figure 1: HDR Compression of a 2D X-ray image of size 960×960: (a) shows the original image, while (b) shows the resulting image after HDR compression.

4. Methods

4.1. Algorithm

For discretization of the image we use finite differences on a node-based grid Ω^h with mesh size h and number of grid points N. The discrete image reads $I^h : \Omega^h \mapsto G^h$. $G^h \subset \mathbb{R}$ denotes the gray value range and is typically 16-bit for medical images. The image derivatives are computed by simple forward and backward finite differences.

Equation (4a) and (4b) are also discretized by finite differences, which leads to a linear system

$$A^h u^h = f^h \, , \quad \sum_{j \in \Omega^h} a_{ij}^h u_j^h = f_i^h \, , i \in \Omega^h \tag{5}$$

with system matrix $A^h \in \mathbb{R}^{N \times N}$, unknown vector $u^h \in \mathbb{R}^N$ and right hand side (RHS) vector $f^h \in \mathbb{R}^N$ on a discrete grid Ω^h. In stencil notation the discretized Laplacian Δ^h reads on a uniform grid

$$\Delta^h = \frac{1}{h^2} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} . \tag{6}$$

Algorithm 1: Recursive V-cycle: $u_h^{(k+1)} = V_h\left(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2\right)$.

```
1  if coarsest level then
2      solve $A^h u^h = f^h$ exactly or by many smoothing iterations
3  else
4      $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}\left(u_h^{(k)}, A^h, f^h\right)$        {pre-smoothing}
5      $r^h = f^h - A^h \bar{u}_h^{(k)}$                                      {compute residual}
6      $r^H = R r^h$                                                          {restrict residual}
7      $e^H = V_H\left(0, A^H, r^H, \nu_1, \nu_2\right)$                      {recursion}
8      $e^h = P e^H$                                                          {interpolate error}
9      $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$                            {coarse grid correction}
10     $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}\left(\tilde{u}_h^{(k)}, A^h, f^h\right)$  {post-smoothing}
11 end
```

The linear system is solved by a standard multigrid method [25, 26, 27, 28]. One multigrid iteration, here the so-called *V-cycle*, is summarized in Algorithm 1.

As multigrid components we choose an ω-Jacobi smoother, linear interpolation, and its transpose as restriction.

Note that we know from our previous work [6] that the optimal smoother with respect to convergence rate for our problem is a red-black Gauss-Seidel smoother, but currently it cannot be modeled efficiently within the DSL.

4.2. The HIPA^{cc} Framework

In order to generate efficient low-level code for our multigrid solver on GPUs we use the HIPA^{cc} framework. Its syntax has been extended to support upsampling/downsampling of images with different interpolation modes (nearest neighbor, bilinear, cubic, or Lanczos resampling) in order to coarsen (restrict) and refine (interpolate) a grid (the image). For multiresolution data sets, image pyramids are introduced that represent images at different resolutions with corresponding syntax to describe the data flow between levels of the pyramid.

4.2.1. Domain-Specific Language (DSL)

The DSL is based on C++ classes that allow to describe kernels operating on a 2D domain (image): these include a) an *Image*, representing the data storage for image pixels (e. g., represented as integer number or floating point number), b) an *Iteration Space*, defining a rectangular region of interest in the output *Image*. Each pixel within this region is generated by c) a *Kernel*. To read from an *Image*, d) an *Accessor* is required, defining how and which pixels are *seen* within the *Kernel*. Similar to an *Iteration Space*, the *Accessor* defines the pixels that are read by the *Kernel*; e) a *Mask* can store the constants of a stencil that can be read in the *Kernel*, and f) a *Domain* represents the corresponding iteration points for applying the stencil (corresponds to non-zero elements of the *Mask*). Since *Kernels* read neighboring pixels, out-of-bounds memory accesses occur and a *Boundary Condition* is required to define how to handle them: this includes to return a *constant* value, to update the index such that either the index is *clamped* at the border or the image is virtually *repeated* or *mirrored* at the border. Also not to care for out-of-bounds memory accesses is a valid option—resulting in an *undefined* behavior.

To describe a V-cycle in the presented DSL, a kernel for each component (gradient, smoothing, residual, restrict, interpolate, coarse grid) is required. Since all components are similar, we show only the implementation of the Jacobi smoother. Therefore, we define the stencil for the smoother and assign it to a *Mask* object of the DSL and define also the corresponding *Domain* for the stencil as seen in Listing 1 (lines 2–8).

As needed for the Jacobi smoother, we define an image for the right hand side (`RHS`) and an *Accessor* to it (lines 11–12). The same is done for the temporary image (`TMP`). The solution image (`SOL`) is written and thus uses an iteration space (lines 15–16). For the smoother we apply Dirichlet boundary condition (*constant* with a value of 0) (lines 21–22). With these instances of DSL classes, an instance of the Jacobi kernel (described later) can be created and executed (lines 25–26).

```
 1 // stencil for Jacobi
 2 const float stencil_jacobi[3][3] = {
 3   { 0.00f, 0.25f, 0.00f },
 4   { 0.25f, 0.00f, 0.25f },
 5   { 0.00f, 0.25f, 0.00f }
 6 };
 7 Mask<float> mask(stencil_jacobi);
 8 Domain dom(mask);
 9
10 // image for RHS
11 Image<float> RHS(width, height);
12 Accessor<float> acc_rhs(RHS);
13
14 // image for SOL
```

```
Image<float> SOL(width, height);
IterationSpace<float> iter_sol(SOL);

// image for TMP
Image<float> TMP(width, height);
// reading from TMP with Dirichlet boundary condition
BoundaryCondition<float> bound_tmp(TMP, mask, BOUNDARY_CONSTANT, 0);
Accessor<float> acc_tmp(bound_tmp);

// instantiate and launch the JacobiKernel operator
JacobiKernel JacobiSol(iter_sol, acc_rhs, acc_tmp, mask, dom);
JacobiSol.execute();
```

Listing 1: Instantiate operator for the Jacobi smoother.

Kernels in the framework are implemented as classes that derive from the framework-provided `Kernel` base class, featuring a constructor, (private) class members, and a `kernel()` method describing the stencil operation. Here, the stencil operation is described using the `reduce()` method taking a) the domain of the stencil, b) the aggregation mode, and c) the computation instructions for a single stencil component with the corresponding image pixel described as a C++ lambda-function. The result can be stored using the `output()` method. An alternative syntax is shown in Listing 3, where the stencil is manually computed accessing neighboring pixels using relative offsets. While the second syntax seems to be more concise for this example, it gets bloated for larger stencils. Using the first notation has also the benefit that the same kernel description can be used for different stencils. Note that the presented DSL is based on standard C++ and 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.

```
class JacobiKernel : public Kernel<float> {
  private:
    Accessor<float> &RHS, &Sol;
    Mask<float> &mask;
    Domain &dom;

  public:
    JacobiKernel(IterationSpace<float> &iter, Accessor<float> &RHS,
        Accessor<float> &Sol, Mask<float> &mask, Domain &dom) :
        Kernel(iter),
        RHS(RHS),
        Sol(Sol),
        mask(mask),
        dom(dom)
    {
        addAccessor(&RHS);
        addAccessor(&Sol);
    }

    void kernel() {
      output() = Sol() + 0.25f*RHS() + reduce(dom, SUM, [&] () -> float {
        return mask(dom) * Sol(dom);
      });
    }
};
```

Listing 2: Kernel description of the Jacobi kernel.

```
    void kernel() {
      output() = Sol() + 0.25f*(RHS() + Sol(-1, 0) + Sol(1, 0) + Sol(0,
          -1) + Sol(0, 1));
    }
```

Listing 3: Kernel description of the Jacobi kernel using relative memory accesses.

For the restrict and interpolate kernels, an *Accessor* is defined that provides (bi)linear interpolation when accessing pixels: no interpolation has to be described by the programmer, instead only a different *Accessor* has to be used, in this case `AccessorLF` in combination with a boundary condition such as *clamp*.

4.2.2. Code Generation

Based on the latest Clang compiler framework[2], version 3.4, our source-to-source compiler uses the Clang front end for C/C++ to parse the input file and to generate an Abstract Syntax Tree (AST) representation of the source code. Our back end uses this AST representation to apply transformations and to generate host API calls and target-specific device code in CUDA or OpenCL.

Host Code: The parsed AST is transformed using Clang's *Rewriter* to replace the textual representation of individual AST nodes by corresponding API calls to the runtime library provided by HIPA^{cc}. For example, API calls are added to allocate memory on the GPU accelerator, transfer data to/from the accelerator, or to start a kernel on the accelerator. This way, any occurrence of compiler-known classes and instantiations are either removed or replaced. Kernel code is generated (described in the following) and written to separate files. These files get included (CUDA) or loaded (OpenCL) such that the rewritten input source file can be compiled using target compilers (nvcc/g++).

[2]http://clang.llvm.org

Device Code: AST nodes of kernels described in the DSL are translated into corresponding nodes for CUDA or OpenCL device code. Some AST nodes are added, for example nodes for global and local index calculations, or nodes to stage image pixels into local memory, to map multiple iterations to one GPU thread (unrolling), or to generate different code variants for the same kernel with tailored boundary handling for different regions of the domain (image). During traversal of the AST, nodes referring to classes derived from built-in DSL classes are replaced by corresponding nodes for CUDA and OpenCL, for example redirecting *Accessor* reads to memory fetches from global memory, texture memory, or local memory—depending on the target device. Similarly, *Mask* accesses get mapped to constant memory or propagated as constants in case the operator is described using the `convolve()` function.

4.2.3. Code Optimization

Optimization is guided by an architecture model for a given GPU architecture and the information captured by the DSL constructs of a program in HIPA^{cc} [2, 29]. The architecture model defines which optimizations are beneficial for a given target architecture and selects the transformations applied during code generation. The information extracted from the DSL program includes the size of a stencil, the shape of a stencil, as well as the memory access pattern within kernels. This information is used to generate tailored variants for each stencil:

- Memory layout and memory alignment/padding for images.
- Mapping to the memory hierarchy: memory accesses can be mapped to registers, shared memory, texture memory, constant memory, and global memory.
- Unrolling of the stencil and propagating of the constants to registers.
- Specialized implementations for boundary handling: up to ten specialized variants for the same kernel optimized for different regions of the image.
- Tiling to improve locality and minimize boundary handling.
- Thread-coarsening [30] to optimize for locality between adjacent pixels.

In addition to the above mentioned optimizations, HIPA^{cc} also extracts resource usage (such as number of registers and shared memory usage) of generated kernels to fine-tune their tiling to achieve good occupancy.

4.3. Efficient Description of Multigrid Algorithms

Describing a multigrid algorithm requires a data storage for holding multiple images of increasing or decreasing resolution for different levels of the grid. Between these images, on the same level, and in between different levels, computational instructions have to be described. Moreover, the data dependencies for these computations have to be provided, which allows to derive a schedule directly from the data flow.

Processing a multigrid algorithm is fundamentally similar to operating on an image pyramid [3], which is a common type of multiresolution data representation in the domain of image processing. Operating on an image pyramid includes creating different fine- and coarse-grained images of different resolution. Furthermore, kernels have to be provided for data exchange between levels (e. g., downscaling and upscaling) and for operating on images within the same level. On each level, the kernel computation instructions are identical, only the input and output images that are bound to kernels differ, depending on the current recursion level. Therefore, the execution of a kernel has to be initiated multiple times, each time with the corresponding images according to the current recursion of the pyramid. Since HIPA^{cc} uses static analysis to generate tailored code variants (e. g., for different boundary condition properties of an *Accessor*), calling the same kernel instance multiple times with different *Accessors* is not supported at the moment. As a consequence, the code for describing the pyramid process contains a great amount of redundancy since each pyramid level has to be described separately. To overcome this drawback, we introduce in this work an expressive and flexible way to describe efficiently image pyramids and operations on them, which can be applied to multigrid algorithms as well.

The HIPA^{cc} language support for this is structured into a) *Pyramids*, a new data structure for storing multiresolution image data, and b) `traverse`, a recursive function for describing data flow between pyramid levels.

4.3.1. Image Pyramids

For creating an image pyramid object, the image of the most fine-grained level and the depth of the pyramid must be provided as arguments. Pyramids are based on templates and the template type specifier of the pyramid and the image argument must match. The pyramid automatically allocates and manages device memory for the more coarse-grained images of all levels defined by the depth of the pyramid. The image dimensions, both x and y, will be consecutively halved per level. While iterating over pyramid levels, each image can be addressed by using an integer value describing a relative index. Querying a pyramid at address '0' will return an image handle for the current recursion level. The address '1' refers to the image on the next (more coarse-grained) level, whereas address '-1' refers to the image of the previous (more fine-grained) level, respectively.

Since the data flow may differ in particular on the most coarse- and fine-grained levels, methods are provided to query the current recursion level. Using these control methods, the execution of the recursion body can diverge depending on the current pyramid level.

```
uint Pyramid<T>::getDepth()
uint Pyramid<T>::getLevel()
bool Pyramid<T>::isTopLevel()
bool Pyramid<T>::isBottomLevel()
```

The `getDepth()` method returns the depth of the pyramid and `getLevel()` returns the current recursion level. This can be used for execution of different kernels per level. The `isTopLevel()` and `isBottomLevel()` methods provide a more convenient way to query common corner cases, when operating on the uppermost and lowermost levels.

4.3.2. Recursive Traversal

The process of operating on an image pyramid can be described using a recursive algorithm. Therefore, the most suitable language construct for operating on pyramids is a recursive traversal function, which is sufficient to describe data dependencies for processing multiresolution data sets while preserving full flexibility. HIPA^{cc} offers two functions to describe pyramid traversals:

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

The first function initiates the recursion by taking a) a vector of pyramids, which are bound to the traversal process, and b) the recursion body described as a C++ lambda-function. Only bound pyramids can be addressed by the traversal lambda-function using a relative index.

The second function defines the recursion within the traversal body. It can take an additional integer argument for defining the number of recursive calls that should be called repeatedly as well as an optional lambda-function for describing the process that should be executed in between these calls.

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

Listing 4: Pyramid traversal: the downscale/upscale kernels create a coarser/finer representation and the compute kernel processes the data at each level.

Listing 4 shows a simple example for operating on pyramid data structures using the recursive traversal functions. As needed for the traversal, two image pyramids are created (lines 3–4) with a depth of 5. 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 `upscale` kernel follows the semantics of an addition assignment operator ($+=$), operating directly on the result of the previous kernel. The recursive call (line 19) occurs right after the compute kernel but could also be placed before without changing the schedule. A graphical representation of the scheduled control flow is illustrated in Figure 2.

Using both functions together creates the impression that a direct recursive call occurs, but instead a language hook is executed that forwards the call to the user-defined lambda-function. This pseudo-recursive approach reduces redundancy (e. g., participating pyramids do not need to be declared

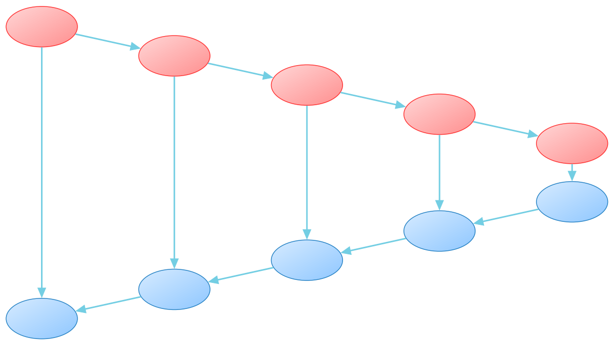


Figure 2: Control flow for an image pyramid of depth 5 with a single recursive call per recursion level. An edge represents the execution of a kernel, the edge direction indicates data flow direction and a node represents the synchronization point (implied by the traverse call) between kernel(s) executions.

again) and can furthermore be used to implicitly fulfill certain control flow requirements, like ending the recursion if the termination condition is met.

4.3.3. Syntactic Sugar

While the introduced language support for multiresolution algorithms is sufficient to describe any algorithm using image pyramids, we want to further emphasize additional support that eases the description of multiresolution algorithms by using concise and distinct expressions. The following section illustrates the expressiveness and flexibility of these techniques.

Repetitive Recursive Calls. One typical task applied to image pyramids is repeating recursive calls on each recursion level. The number of recursive calls can either vary, depending on the depth of the current level, or can be constant in the most common case. Both situations can be expressed using an optional argument to the `traverse()` function without the necessity to apply fundamental changes to the existing DSL code.

Consider the simple recursion in Listing 4 with the corresponding schedule as shown in Figure 2, which represents the schedule of the V-cycle multigrid. Using two repetitive recursive calls instead of one results in a schedule corresponding to a W-cycle as shown in Figure 3. To achieve this, it is sufficient to provide an additional parameter to the `traverse()` function of Listing 4 (line 19) as indicated in Listing 5 (line 3): the first argument specifies that two recursive calls should occur per level and the second argument contains the lambda-function that will be executed in between both calls.

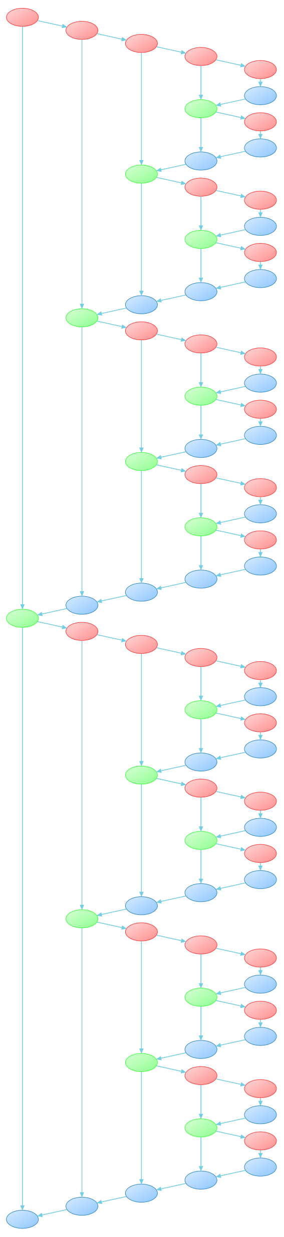


Figure 3: Control flow for an image pyramid of depth 5 with two recursive calls per recursion level. An edge represents the execution of a kernel, the edge direction indicates data flow direction and a node represents the synchronization point (implied by the traverse call) between kernel(s) executions.

```
traverse({ &pin, &pout }, [&] {
  /* ... downscale ... */
  traverse(2, [&] { pin.swap(pout); });
  /* ... upscale ... */
});
```

Listing 5: Repetitive recursive calls by defining optional arguments.

Because the traversal function body, for linking the pyramid images, is constant during the execution, the kernel executed by the second recursive call will always operate on the input image provided by the kernel of the previous level. Defining such a schedule will overwrite any results of the first recursive call. Therefore, to ensure the correctness of the data flow, it is necessary to swap the underlying data structures of the input and output pyramids before each consecutive call on the same level. This must be done explicitly by using the `swap()` method and cannot be done implicitly by the recursive call, because if multiple pyramids are bound to the traversal function it is not clear which of them must be swapped or whether swapping is necessary at all. The `swap()` method can only be applied to two pyramids, both of which need to be constructed with the same template type specifier. Further properties of this method are reflexivity (swapping a pyramid with itself does not have any effect) and symmetry (switching the caller object and argument will result in the same swapping operation).

Describing Local Peaks. To further demonstrate the flexibility of the language support for image pyramids in HIPA^{cc}, we want to show the adaptability of executions on the *local peak*. The term *local peak* denotes the DSL code executed between refinement and coarsement kernel calls, which can be described using the optional lambda-function argument of the recursive traverse function call. The body of the lambda-function can contain DSL code without any additional limitations, and therefore the schedule on *local peaks* can be arbitrarily expanded. Listing 6 illustrates the description of an additional compute kernel call in between two repetitive recursive calls. The transformation of the schedule resulting from such a modification is visualized in Figure 4, which demonstrates the gain of flexibility by providing the possibility to insert custom DSL code blocks on *local peaks*.

Nested Traversals. We build up a stack, which stores pyramid traversal calls whenever a new call is initiated before the predecessing call is finished.

```
traverse({ &pin, &pout }, [&] {
  /* ... downscale ... */
  traverse(2, [&] {
    pin.swap(pout);
    Accessor<float> acc_in(pin(0));
    IterationSpace<float> iter_out(pout(0));
    Compute c(iter_out, acc_in);
    c.execute();
  });
  /* ... upscale ... */
});
```

Listing 6: Describing *local peak* execution using an expanded lamda-function body between two repetitive recursive calls.

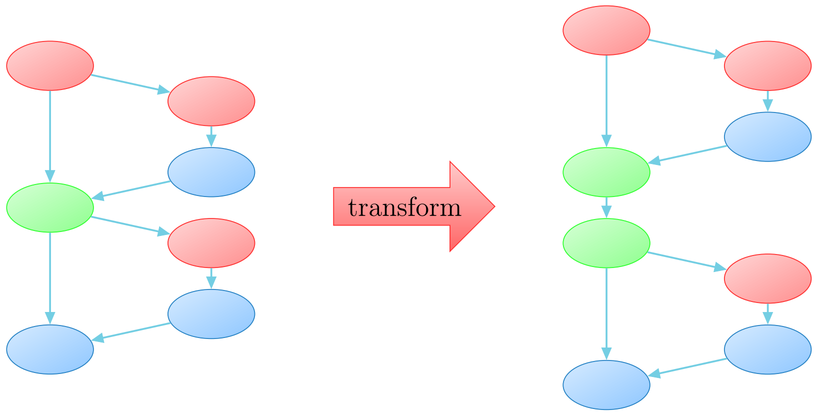


Figure 4: Transformation of the control flow on *local peaks* (green) by using a lambda-function describing an additional compute kernel execution between two repetitive recursive calls.

Therefore, such traversal calls can be nested within another traversal function. That means it is possible to process the traversal of another pyramid at any level of the currently ongoing traversal. Pyramids bound to the outer traversal functions can also be accessed by the inner ones and can be used for computations. In such scenarios, however, every pyramid P is bound to exactly one traversal t at any step during the traversal process, formally defined by

$$\forall p\ \exists! t\ \ \mathrm{Bind}(t, p).$$

Hereby, the use of a relative index, which defines the address of an image of a specific pyramid with respect to a certain level, is then only related to exactly one particular traversal call and its current level. Even though

Table 1: Comparison of lines of DSL code with and without pyramids for initial image allocation (Init), different recursion levels (L) and additional control structures (control) depending on recursion level i and recursion depth n.

	Manual (in LoC)	Pyramid (in LoC)
Init(n):	$8n + 2$	$8 + 2$
L($i = 1$):	$34 + 5$	$32 + 5$
L(i):	34	0
L($i = n$):	15	4
control:	0	5

nested traversals are only useful for very special image algorithms such as the Local Laplacian Filter [31], it further emphasizes the expressiveness of our approach.

4.3.4. Describing V-Cycles using Image Pyramids

To describe a simple V-cycle, it is only necessary to specify a single recursion step of the image pyramid, which results in a straight-forward implementation of Algorithm 1. Table 1 compares the Lines of Code (LoC) for the manual and pyramid approach, based on initial image allocation and the number of declared kernels on each level, including the required definitions for *Iteration Spaces* and *Accessors*. It can be seen that the LoC using the manual approach scales linearly with the number of levels n, while the use of pyramids results in a constant LoC.

The number of pre- and postsmoothing steps can be varied per level or depending on other conditions. Since `traverse()` accepts a parameter describing the number of recursive calls, switching to a W-cycle is a straight-forward task. As the number of recursive calls does not have to be constant on each recursion level, custom cycle types are easy to describe.

$$\mathrm{LoC}(n) = \mathrm{Init}(n) + \sum_{i=1}^{n} \mathrm{L}(i) + \mathrm{control} \tag{7}$$

5. Evaluation and Discussion

In this section, we evaluate the construction of image pyramids and the presented HDR compression application using the proposed language extension.

Table 2: Pyramid construction in Halide and HIPAcc for an image of 4096×4096 pixels using 11 levels.

	kernel invocations	Tesla K20	Tesla C2050	Quadro FX 5800
Halide†	3	5.10 ms	12.74 ms	n/a
Halide‡	1	1.24 ms	3.45 ms	n/a
HIPAcc	33	1.16 ms	2.42 ms	12.99 ms

† Using the same schedule as the HIPAcc implementation.
‡ Using the optimized schedule.

The implementations are compared with a corresponding implementation in Halide and a hand-tuned OpenCL implementation. We evaluate the performance of the generated code on three GPU architectures from NVIDIA, using the Tesla K20, the Tesla C2050, and the Quadro FX 5800 GPUs.

5.1. Gaussian Laplacian Pyramid Construction

To evaluate the performance for constructing multigrid representations, we implement the multiresolution representation of [32] and compare the implementation against the corresponding implementation in Halide. The multiresolution representation of [32] creates a pyramid of Gaussian and Laplacian images[3].

A user-defined kernel in HIPAcc will be translated to a corresponding compute kernel and result in a kernel invocation for each level of the grid. Each kernel invocation reads its input data and stores its output back to memory. In contrast, computations in Halide are pure functions that define the value of pixels. Whether a computation in Halide is translated to a compute kernel depends on the schedule specified by the programmer.

Considering an input image of 4096×4096 pixels, we will get images of 11 different resolutions. To create the pyramid representation and reconstruct the original image requires 33 kernel invocations in HIPAcc: during construction of the pyramid, one kernel is required for computing the Gaussian image and one for the Laplacian image; the reconstruction requires a single kernel only. Table 2 shows the total execution time and number of kernel invocations for

[3]The local Laplacian filter implementation in [8] uses a similar multiresolution representation and is available online.

Halide and HIPA^{cc}. We use NVIDIA's profiler *nvprof* to measure the kernel execution time and report the minimum of 10 runs. Since Halide relies on LLVM's back end to generate code for NVIDIA GPUs, they do not support code generation for the Quadro FX 5800[4]. We list two schedules in Halide: the first schedule tells Halide to precompute the entire required region of each computation before the data is consumed by subsequent computations (using the *root* schedule). This corresponds to the schedule imposed by HIPA^{cc} and results in 3 kernel invocations (each kernel computes all levels of the pyramid). The second schedule optimizes for locality and generates just a single kernel for pyramid construction and reconstruction of the original image. It can be seen that the schedule using a single kernel is much faster compared to the schedule using multiple kernels in Halide. The total execution time in HIPA^{cc} is even faster compared to the optimized schedule in Halide although no kernels are fused in HIPA^{cc}.

Note that the optimized Halide implementation stores neither the Gaussian pyramid nor the Laplacian pyramid. If this is desired, the execution time would increase correspondingly.

To describe the algorithm in HIPA^{cc}, exactly 26 LoC are necessary regardless of the pyramid's depth n. Considering Equation (7), this sum consists of the following components:

$$\mathrm{LoC}(n) = 7 \text{ (Init)} + 15 \text{ } (\textstyle\sum \mathrm{L}) + 4 \text{ (control)}$$

The description of the execution on each pyramid level is identical. Therefore, only one single level with 15 LoC needs to be taken into account for the second summand. Including the LoC for describing the compute kernels, the complete code base for this algorithm written in HIPA^{cc} results in 31 LoC—the same size is necessary to express the algorithm in Halide[5].

The Halide implementation builds up an array of function objects for each level. As the computational steps on each level are the same, functions can be defined either by iterating over pyramid levels using *for*-loops or by utilizing a recursive function. For more complex structures like W-cycles, pyramids are much more difficult to construct using the iterative approach. Therefore, it appears more natural to describe image pyramids in a recursive manner. Halide however, does not distinguish between computational steps and data

[4]LLVM's PTX back end supports only GPUs with compute capability greater than 2.0
[5]without considering the LoC for providing an efficient schedule

Table 3: Theoretical **peak** and the achievable (**memcpy**) memory bandwidth in GB/s for Quadro FX 5800, Tesla C2050, and Tesla K20 GPUs.

	Tesla K20	Tesla C2050	Quadro FX 5800
Peak	208 GB/s	144 GB/s	102 GB/s
Memcpy	$140^{\ddagger}$ GB/s	$102^{\dagger}$ GB/s	71.4 GB/s
Percentage	$67.3^{\ddagger}$ %	$70.8^{\dagger}$ %	69.7 %

† Error-Correcting Code (ECC) memory turned on, 118 GB/s with ECC turned off (82.1 % of peak bandwidth).
‡ ECC memory turned on, 159 GB/s with ECC turned off (76.4 % of peak bandwidth).

storage. Both are represented by functions. Since data storage is bound to functions, the computational steps within functions cannot be reused. Thus, many temporary functions need to be created to achieve a W-cycle with Halide.

5.2. HDR Compression

Since it is known that stencil codes are usually bandwidth limited we list the theoretical peak and the achievable memory bandwidth of all GPUs in Table 3. In order to estimate the quality of the generated code we consider one smoothing step on the finest level as example. Here, we have to load two values from main memory (from RHS and SOL) and to store one value (TMP), if we assume that all neighboring memory accesses for the stencil are in cache. This means for single precision accuracy we have to transfer $4 \cdot 3 = 12$ bytes per pixel. On the Tesla C2050 with achievable memory bandwidth of $b = 102$ GB/s and for problem size $N = 2048 \times 2048$ we thus estimate $\frac{N \cdot 12}{b} \cdot 1000 \approx 0.5$ ms for the smoother. This matches quite well to the measured runtime of 0.53 ms (and 0.59 ms) for the hand-tuned (and generated) OpenCL implementation of the smoother as seen in Table 4. The hand-tuned OpenCL implementation of the multigrid solver [6] using a red-black Gauss-Seidel (RBGS) smoother includes a splitting of red and black points, kernel fusion (e. g., for residual computation and restriction) and spatial blocking techniques to reduce the amount of required memory transfers. These are the main reasons why it is faster than the automatically generated code. Since the manual OpenCL implementation merges multiple kernels (e. g., smooth, residual, and restrict) into a single kernel, only one runtime is listed in Table 4 for kernels merged in the hand-tuned implementation.

Table 4: Execution times in ms for one V-cycle on the Quadro FX 5800, Tesla C2050, and Tesla K20 for an image of size 2048×2048 pixels. Shown is the hand-tuned OpenCL (with RBGS smoother) as well as the generated CUDA and OpenCL implementations.

	Tesla K20			Tesla C2050			Quadro FX 5800		
	Manual	OpenCL	CUDA	Manual	OpenCL	CUDA	Manual	OpenCL	CUDA
L1: smooth	0.35	0.40	0.40	0.53	0.59	0.77	1.33	1.27	0.93
L1: smooth		0.39	0.39		0.58	0.77		1.30	0.93
L1: residual	0.45	0.40	0.39	0.67	0.57	0.77	1.63	1.25	0.94
L1: restrict		0.16	0.16		0.25	0.27		0.43	0.41
L2: smooth	0.10	0.11	0.12	0.12	0.17	0.20	0.34	0.38	0.26
L2: smooth		0.11	0.12		0.16	0.21		0.36	0.27
L2: residual	0.16	0.11	0.13	0.19	0.16	0.21	0.43	0.37	0.27
L2: restrict		0.05	0.06		0.07	0.09		0.13	0.12
L3–L6	0.75	0.47	0.66	0.63	0.59	1.85	1.28	1.37	1.34
L2: interpolate		0.08	0.09		0.10	0.13		0.24	0.22
L2: smooth	0.12	0.11	0.12	0.15	0.16	0.22	0.34	0.36	0.32
L2: smooth	0.28	0.11	0.12	0.34	0.16	0.22	0.84	0.37	0.32
L1: interpolate		0.27	0.31		0.39	0.41		0.91	0.62
L1: smooth	0.35	0.40	0.39	0.53	0.59	0.78	1.32	1.29	0.99
L1: smooth	0.35	0.39	0.39	0.53	0.59	0.78	1.32	1.27	0.99
$\sum$V-cycle	2.91	3.54	3.91	3.69	5.13	6.93	8.83	11.30	9.70

Table 5: Execution times in ms for the smoother on the Quadro FX 5800, Tesla C2050, and Tesla K20 for an image of size 2048×2048 pixels. Shown is the hand-tuned OpenCL (with RBGS/Jacobi smoother) as well as the generated CUDA and OpenCL implementations (Jacobi smoother).

		Tesla K20	Tesla C2050	Quadro FX 5800
Manual	RBGS	0.35	0.53	1.33
	Jacobi	0.40	0.69	1.74
HIPA^{cc}	OpenCL	0.40	0.59	1.27
	CUDA	0.40	0.77	0.93

Furthermore, in Table 4 the execution times for all other steps of one V-cycle (see Algorithm 1) are found. The V-cycle has six levels and for smoothing (pre and post) two iterations are executed on each level. It can be seen that most of the execution time is spent in the smoother kernel on the finest level and that for very coarse levels the generated CUDA and OpenCL codes become inefficient because GPU kernels working only on a small amount of data achieve lower memory bandwidth.

Since the manual implementation in [6] uses an RBGS smoother, we implemented and auto-tuned also the Jacobi smoother to assess the performance of the generated code. Table 5 lists the runtime for the hand-tuned RBGS and Jacobi smoother as well as of the generated implementations for the Jacobi smoother for the finest level. It can be seen that the RBGS smoother is the fastest, but it can be also seen that the generated OpenCL implementations are faster compared to the hand-tuned OpenCL implementations. The automatically generated code is even faster as the hand-tuned implementation on the Quadro FX 5800. This can be attributed to the fact that the automatically generated code here uses texture memory and the manual implementation does not make use of texture memory.

5.3. Discussion

While the final execution time of an application is for most evaluations the only criteria, we consider here multiple criteria: productivity, portability, and performance. Achieving high performance comes often at the cost of low productivity and the loss of portability (e. g., [1] showed that performance cannot be preserved simply by using OpenCL to target different GPUs accelerators). We believe that using a domain-specific approach, competitive performance can be achieved without making concessions with regard to productivity and portability.

Productivity. Writing code that maps efficiently to a target platform requires profound insights into the target hardware architecture. However, HPC users come often from other disciplines like biology, physics, or medicine. These users do not want to care about low-level programming and architecture details in order to map algorithms to a target platform. Instead, they only want to give a high-level description of the used algorithm, which can be written in HIPA^{cc}. For example, the stencil codes for this paper were written in less than half a day using the HIPA^{cc} framework, while the hand-tuned optimizations done by a domain expert took several weeks after a basic version in OpenCL was available. Similarly, the LoC that have to be written for a kernel in the framework is moderate (about 15 lines for a kernel class, of which 3 lines describe the algorithm). From this description, about 500 lines of CUDA/OpenCL (including ten different code variants for boundary handling) are generated. Note that the hand-tuned and blocked OpenCL implementation consumes almost 1200 LoC only for the kernels executed on the GPU.

Portability. Computationally intensive parts that are accelerated on special hardware such as graphics cards are typically written in a special language such as CUDA or OpenCL. Moving to a different accelerator may require, hence, to rewrite accelerated parts in another language. In contrast, using a high-level description with automatic code generations provides portability and allows to use the same algorithm description to target different accelerators. Even worse, running a program written and optimized for one given target architecture on a different platform comes often along with a performance penalty. That is, such codes are often not (performance) portable. To solve this dilemma, HIPA^{cc} provides support for different target languages and target architectures: CUDA, OpenCL, as well as Renderscript [33] (the parallel programming model of Android[6]). Depending on the target platform (e. g., Tesla K20, Tesla C2050, or Quadro FX 5800), different code variants are generated (employing target-tailored optimizations, kernel configurations according to available resources, etc.). All these implementations are generated from the same high-level description. New GPU accelerators can be easily supported by providing an architecture model description. Only emerging architectures programmed in

[6]http://developer.android.com/guide/topics/renderscript/compute.html

a different language require a new back end.

Performance. Competitive performance can be achieved using domain-specific code generators. In [2, 30], we showed that the code generated by HIPA^{cc} has competitive performance for local operators compared to hand-written CUDA codes in the Open Source Computer Vision (OpenCV) library or in the NVIDIA Performance Primitives (NPP) library. In most cases, our code is even faster. Our first results for stencil codes show also that we achieve similar performance as the hand-tuned implementation. One can expect the same performance in case of code generation, if all of the performance optimization techniques from the hand-tuned code are applied.

5.4. Future Work

While we have shown that optimized code can be generated for single kernels, it is essential to optimize from an application perspective in order to capitalize GPU accelerators best. This is in particular true for a sequence of memory bound kernels, where the fusion of two kernels saves global memory stores/reads and results in significantly faster execution (up to 2×). All required information for kernel fusion is available in the DSL description: execution constraints (iteration space) and memory accesses within a kernel. For a sequence of kernels, the DSL syntax can be extended to describe pipelining (e. g., the kernels within one V-cycle level) such that the same optimizations can be applied as for hand-tuned implementations.

Going a step further, the whole V-cycle can be described even more declarative. This delegates decisions such as the choice of the smoother (e. g., Jacobi vs. Gauss-Seidel) or the multigrid variant (e. g., V-cycle vs. W-cycle) to the framework depending on target platform properties such as memory bandwidth to FLOPS ratio.

Since the DSL decouples the algorithm from its schedule, code can be generated tailored to target platforms that span multiple nodes that can feature heterogeneous components: the domain can be decomposed such that sub-domains are mapped to and executed by different types of processing elements (CPU, GPU, Intel MIC, etc.). This is possible since no target-specific code is written in the DSL. Already now, the source-to-source compiler generates different code variants for efficient boundary handling that are selected/executed depending on the currently processed image region. This can be extended to not only support different code variants, but also different target languages. The target platform description can be provided separately, for instance, to the compiler in the form of a configuration file.

6. Conclusions

We have shown that automatic code generation is a useful tool to increase productivity for HPC applications. The performance of the generated code is on a par with the performance of state-of-the-art stencil frameworks as well as hand-optimized implementations. By introducing language constructs in the DSL to represent data at different resolution, we have shown that multiresolution algorithms can be described in a concise way. In the next years we want to establish these techniques in order to generate code also for large distributed memory parallel HPC clusters.

The presented domain-specific language and source-to-source compiler have been implemented in the HIPA^{cc} framework, which is available as open-source under http://hipacc-lang.org.

Acknowledgments

This work is partly supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 “Software for Exascale Computing” in project under contracts TE 163/17-1 and RU 422/15-1 as well as 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, J. Dongarra, From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming, Parallel Computing 38 (8) (2011) 391–407.

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

[3] P. Burt, E. Adelson, The Laplacian pyramid as a compact image code, IEEE Transactions on Communications 31 (4) (1983) 532–540.

[4] A. Brandt, Multi-level adaptive solutions to boundary-value problems, Mathematics of Computation 31 (138) (1977) 333–390.

[5] W. Hackbusch, Multi-Grid Methods and Applications, Springer Series in Computational Mathematics, Springer, 1985.

[6] H. Köstler, M. Stürmer, T. Pohl, Performance engineering to achieve real-time high dynamic range imaging, Real-Time Image Processing (2013) 1–13.

[7] R. Membarth, F. Hannig, J. Teich, H. Köstler, Towards Domain-specific Computing for Stencil Codes in HPC, in: Proceedings of the 2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), IEEE, Salt Lake City, UT, USA, 2012, pp. 1133–1138.

[8] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, F. Durand, Decoupling algorithms from schedules for easy optimization of image processing pipelines, ACM Transactions on Graphics (TOG) 31 (4) (2012) 32:1–32:12.

[9] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, P. Hanrahan, Liszt: A domain specific language for building portable mesh-based PDE solvers, in: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), ACM, 2011, pp. 9:1–9:12.

[10] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, C. E. Leiserson, The Pochoir stencil compiler, in: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM, 2011, pp. 117–128.

[11] A. Baker, R. Falgout, T. Kolev, U. M. Yang, Scaling Hypre's multigrid solvers to 100,000 cores, High-Performance Scientific Computing (2012) 261–279.

[12] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander, A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework, Computing 82 (2) (2008) 103–119.

[13] S. Kamil, C. Chan, L. Oliker, J. Shalf, S. Williams, An auto-tuning framework for parallel multicore stencil computations, in: Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS), IEEE, 2010, pp. 1–12.

[14] M. Christen, O. Schenk, H. Burkhart, PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures, in: Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS), IEEE, 2011, pp. 676–687.

[15] Berkeley Benchmarking and Optimization (BeBOP) Group, University of California, Berkeley, pOSKI: Parallel Optimized Sparse Kernel Interface Library (Apr. 2012).

[16] N. Maruyama, T. Nomura, K. Sato, S. Matsuoka, Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers, in: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), IEEE, 2011, pp. 11:1–11:12.

[17] J. Holewinski, L.-N. Pouchet, P. Sadayappan, High-performance code generation for stencil computations on GPU architectures, in: Proceedings of the 26th ACM international conference on Supercomputing, ACM, 2012, pp. 311–320.

[18] D. A. Orchard, M. Bolingbroke, A. Mycroft, Ypnos: Declarative, parallel structured grid programming, in: Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, ACM, 2010, pp. 15–24.

[19] T. Muranushi, Paraiso: An automated tuning framework for explicit solvers of partial differential equations, Computational Science & Discovery 5 (1) (2012) 1–40.

[20] I. Dietrich, R. German, H. Köstler, U. Rüde, Modeling multigrid algorithms for variational imaging, in: Proceedings of the 21st Australian Software Engineering Conference (ASWEC), IEEE, 2010, pp. 224–234.

[21] P. Feautrier, C. Lengauer, Polyhedron model, in: Encyclopedia of Parallel Computing, Springer, 2011, pp. 1581–1592.

[22] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, A compiler framework for optimization of affine loop nests for GPGPUs, in: Proceedings of the 22nd annual international conference on Supercomputing, ACM, 2008, pp. 225–234.

[23] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, R. Lethin, A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction, in: Proceedings of the 3rd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU), ACM, 2010, pp. 51–61.

[24] R. Fattal, D. Lischinski, M. Werman, Gradient domain high dynamic range compression, in: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), ACM, 2002, pp. 249–256.

[25] W. L. Briggs, H. Van Emden, S. F. McCormick, A Multigrid Tutorial, Vol. 2, Society for Industrial And Applied Mathematics (SIAM), 2000.

[26] U. Trottenberg, C. W. Oosterlee, A. Schüller, Multigrid, Academic Press, 2000.

[27] M. Stürmer, J. Treibig, U. Rüde, Optimising a 3d multigrid algorithm for the IA-64 architecture, International Journal of Computational Science and Engineering 4 (1) (2008) 29–35.

[28] H. Köstler, M. Stürmer, U. Rüde, A fast full multigrid solver for applications in image processing, Numerical Linear Algebra with Applications 15 (2–3) (2008) 187–200.

[29] R. Membarth, F. Hannig, J. Teich, M. Körner, W. Eckert, Mastering software variant explosion for GPU accelerators, in: Proceedings of the 10th International Workshop on Algorithms, Models and Tools for

Parallel Computing on Heterogeneous Platforms (HeteroPar), Springer, 2012, pp. 123–132.

[30] R. Membarth, F. Hannig, J. Teich, M. Körner, W. Eckert, Automatic optimization of in-flight memory transactions for GPU accelerators based on a domain-specific language for medical imaging, in: Proceedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC), IEEE, 2012, pp. 211–218.

[31] S. Paris, S. W. Hasinoff, J. Kautz, Local Laplacian filters: Edge-aware image processing with a Laplacian pyramid, ACM Transactions on Graphics (TOG) 30 (4) (2011) 68:1–68:12.

[32] D. Kunz, K. Eck, H. Fillbrandt, T. Aach, Nonlinear multiresolution gradient adaptive filter for medical images, in: Proceedings of SPIE Medical Imaging 2003: Image Processing, Vol. 5032, SPIE, 2003, pp. 732–742.

[33] R. Membarth, O. Reiche, F. Hannig, J. Teich, Code Generation for Embedded Heterogeneous Architectures on Android, in: Proceedings of the Conference on Design, Automation and Test in Europe (DATE), IEEE, 2014, pp. 86:1–86:6.