This is the author's version of the work. The definitive work was published in *Proceedings of the 2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), Salt Lake City, Utah, USA, November 16, 2012.*

Towards Domain-specific Computing for Stencil Codes in HPC

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

Harald Köstler
System Simulation,
Department of Computer Science,
University of Erlangen-Nuremberg, Germany.
harald.koestler@cs.fau.de

***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 is generated from a high-level description of a geometric multigrid algorithm written in a Domain-Specific Language (DSL) instead of writing hand-tuned code for GPU accelerators. By decoupling the algorithm from its schedule, the proposed approach allows to generate efficient stencil codes. Our results show that competitive performance compared to hand-tuned codes can be achieved and that more than 25 frames per second for 16.8 Megapixel images are obtained for full High Dynamic Range (HDR) compression of 2D medical data sets.**

***Index Terms*—domain-specific language; stencil codes; code generation; GPU; CUDA; OpenCL**

I. Introduction

Mapping algorithms in an efficient way to the target hardware poses a challenge 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 the algorithms efficiently to a target platform. The DSL is part of the Heterogeneous Image Processing Acceleration (HIPA^{cc}) framework 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. We consider HDR compression of 2D images as example application. 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 is one of the most efficient numerical methods to solve large, sparse linear systems arising for example when discretizing PDEs. PDEs are used to model various physical or technical effects in many application fields. One of the most popular 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.

The paper first introduces related work on DSLs and frameworks for stencil codes. Then, an overview of the application and the used multigrid solver is given. The HIPA^{cc} framework 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.

II. 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 [2] and Pochoir [3] 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 multicore workstations. However, both languages (Liszt and Pochoir) provide only limited support for the characteristics of the hardware platform.

The hypre library [4] is a collection of high-performance preconditioners and solvers for large sparse linear systems of equations on massively parallel machines. It offers, for example,

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

a stencil-based interface for computations on structured or block-structured grids and also incorporates different multigrid solvers. DUNE [5] 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.

PATUS (Parallel Auto-Tuned Stencils) [6], [7] 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 no further domain-specific knowledge engineering is used to generate the hardware-adapted code.

The parallel Optimized Sparse Kernel Interface (pOSKI) [8] is a collection of algorithms for operations involving sparse matrices on uniprocessor and multicore machines. It includes auto-tuning at installation and run time and is suitable for stencil computations yielding special sparse matrices.

III. Application

In order to do imaging in the gradient domain as described, for example, in [9] 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

$$\Delta u = f \quad \text{in } \Omega \tag{4a}$$

$$u = 0 \quad \text{on } \partial\Omega \tag{4b}$$

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

IV. Methods

A. 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.

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   $\bar{u}_h^{(k)} = \mathscr{S}_h^{\nu_1}\left(u_h^{(k)}, A^h, f^h\right)$; `{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   $u_h^{(k+1)} = \mathscr{S}_h^{\nu_2}\left(\tilde{u}_h^{(k)}, A^h, f^h\right)$; `{post-smoothing}`
11 **end**

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

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

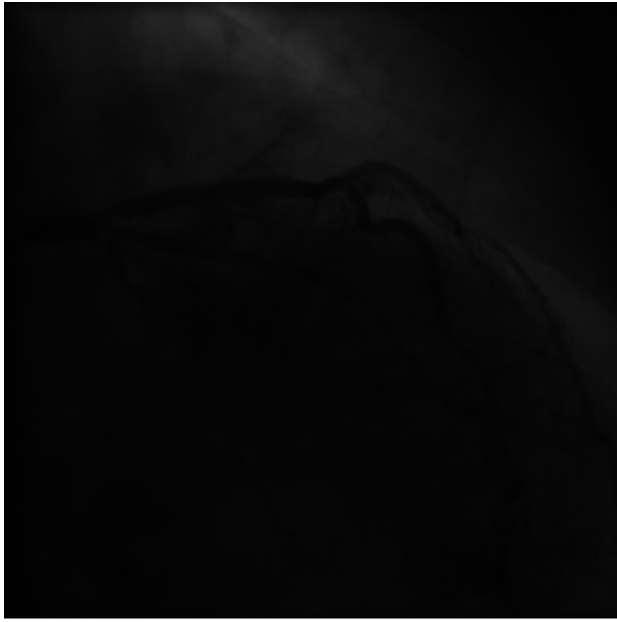
The linear system is solved by a standard multigrid method [10]–[14]. One multigrid iteration, here the so-called *V-cycle*, is summarized in Algorithm 1.

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

B. 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).

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



(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.

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*. 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 as seen in Listing 1 (lines 1–8).

As needed for the Jacobi smoother, we define an image for the right hand side (`RHS`) and an *Accessor* to it (lines 10–12). 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 20–21). With these instances of DSL-classes, an instance of the Jacobi kernel (described later) can be created (line 25) and the kernel can be executed (line 28).

```
1  // filter mask for jacobi
2  const float filter_jacobi[] = {
3    0,       0.25f,  0,
4    0.25f,   0,      0.25f,
5    0,       0.25f,  0
6  };
7  Mask<float> Mjacobi(size_x, size_y);
8  Mjacobi = filter_jacobi;
9
10 // image for RHS
11 Image<float> RHS(width, height);
12 Accessor<float> AccRHS(RHS);
13
14 // image for SOL
15 Image<float> SOL(width, height);
16 IterationSpace<float> IsSOL(SOL);
17
18 // image for TMP
19 Image<float> TMP(width, height);
20 // reading from TMP with Dirichlet boundary condition
21 BoundaryCondition<float> BcTMPConst(TMP, Mjacobi,
       BOUNDARY_CONSTANT, 0);
22 Accessor<float> AccTMPConst(BcTMPConst);
23
24 // kernel declaration
25 JacobiKernel JacobiSol(IsSOL, AccRHS, AccTMPConst,
       Mjacobi);
26
27 // compute jacobi
28 JacobiSol.execute();
```

Listing 1: Example code for 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 `convolve()` method taking a) 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.

```
1  class JacobiKernel : public Kernel<float> {
2    private:
3      Accessor<float> &RHS, &Sol;
4      Mask<float> &cMask;
5
6    public:
7      JacobiKernel(IterationSpace<float> &IS, Accessor<
           float> &RHS, Accessor<float> &Sol, Mask<float> &
           cMask) :
8        Kernel(IS),
9        RHS(RHS),
10       Sol(Sol),
11       cMask(cMask)
12     {
13       addAccessor(&RHS);
14       addAccessor(&Sol);
15     }
16
17     void kernel() {
18       output() = Sol() + 0.25f*RHS() + convolve(cMask,
             SUM, [&] () -> float {
19         return cMask() * Sol(cMask);
20       });
21     }
22 };
```

Listing 2: Kernel description of the Jacobi kernel.

```
1      void kernel() {
2        output() = Sol() + 0.25f*(RHS() + Sol(-1, 0) + Sol
             (1, 0) + Sol(0, -1) + Sol(0, 1));
3      }
```

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*.

2) Code Generation: Based on the latest Clang compiler framework [2], 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.

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

	Quadro FX 5800	Tesla C2050
Peak	102 GB/s	144 GB/s
Memcpy	71.4 GB/s	102† GB/s
Percentage	69.7 %	70.8† %

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

V. Evaluation and Discussion

A. Evaluation

We evaluate the performance of the generated code on two GPU architectures from NVIDIA, using the Quadro FX 5800 and the Tesla C2050 GPUs. Since it is known that stencil codes are usually bandwidth limited we list the theoretical peak and the achievable memory bandwidth of both GPUs in Table I. In order to estimate the quality of the generated code we consider for example one smoothing step on the finest level. 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 cash. 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 Jacobi smoother. This matches quite well to the measured runtime of 0.53 ms for the hand-tuned OpenCL implementation in that case in Table II. This hand-tuned OpenCL implementation of the multigrid solver includes kernel fusion (e. g., for residual computation and restriction) and a kind of wavefront blocking, which reduces the amount of required memory transfers. These are the main reason why it is faster than the automatically generated code.

Table II: Execution times in *ms* for one V-cycle on the Quadro FX 5800 and Tesla C2050 for an image of size 2048×2048 pixels. Shown is the hand-tuned OpenCL as well as the generated CUDA and OpenCL implementations.

<table>
<tr><th></th><th colspan="3">Tesla C2050</th><th colspan="3">Quadro FX 5800</th></tr>
<tr><th></th><th>Manual</th><th>OpenCL</th><th>CUDA</th><th>Manual</th><th>OpenCL</th><th>CUDA</th></tr>
<tr><td>L1: smooth</td><td>0.53</td><td>0.58</td><td>0.79</td><td>1.35</td><td>1.50</td><td>1.01</td></tr>
<tr><td>L1: smooth</td><td></td><td>0.57</td><td>0.79</td><td></td><td>1.48</td><td>0.99</td></tr>
<tr><td>L1: residual</td><td>0.67</td><td>0.57</td><td>0.79</td><td>1.65</td><td>1.62</td><td>0.93</td></tr>
<tr><td>L1: restrict</td><td></td><td>0.28</td><td>0.28</td><td></td><td>0.59</td><td>0.53</td></tr>
<tr><td>L2: smooth</td><td>0.12</td><td>0.16</td><td>0.26</td><td>0.35</td><td>0.44</td><td>0.26</td></tr>
<tr><td>L2: smooth</td><td></td><td>0.16</td><td>0.26</td><td></td><td>0.44</td><td>0.27</td></tr>
<tr><td>L2: residual</td><td>0.19</td><td>0.16</td><td>0.25</td><td>0.44</td><td>0.46</td><td>0.26</td></tr>
<tr><td>L2: restrict</td><td></td><td>0.08</td><td>0.12</td><td></td><td>0.18</td><td>0.16</td></tr>
<tr><td>L3–L6</td><td>0.70</td><td>0.63</td><td>1.85</td><td>1.33</td><td>1.73</td><td>1.34</td></tr>
<tr><td>L2: interpolate</td><td></td><td>0.21</td><td>0.17</td><td></td><td>0.29</td><td>0.18</td></tr>
<tr><td>L2: smooth</td><td>0.15</td><td>0.16</td><td>0.27</td><td>0.34</td><td>0.45</td><td>0.27</td></tr>
<tr><td>L2: smooth</td><td rowspan="2">0.34</td><td>0.16</td><td>0.27</td><td rowspan="2">0.86</td><td>0.44</td><td>0.27</td></tr>
<tr><td>L1: interpolate</td><td>0.83</td><td>0.48</td><td>0.96</td><td>0.61</td></tr>
<tr><td>L1: smooth</td><td>0.53</td><td>0.57</td><td>0.89</td><td>1.35</td><td>1.48</td><td>1.01</td></tr>
<tr><td>L1: smooth</td><td>0.53</td><td>0.57</td><td>0.88</td><td>1.35</td><td>1.49</td><td>1.01</td></tr>
<tr><td>∑V-cycle</td><td>3.90</td><td>5.75</td><td>8.31</td><td>9.02</td><td>13.54</td><td>9.07</td></tr>
</table>

Furthermore, in Table II 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.

B. 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.

a) Productivity: writing code that maps efficient to 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 describe the used algorithm, which is exactly the high-level description, 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 took more than three months after a basic version in OpenCL was available. Similarly, the lines of code 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 1.200 lines of code only for the kernels executed on the GPU accelerators.

b) Portability: the proposed framework provides support for different target platforms. This includes not only support for GPU architectures from AMD and NVIDIA, but also the support of different target languages (CUDA and OpenCL). Depending on the target platform, different code variants are generated (employing target-tailored optimizations, kernel configurations according to available resources, etc.). Further GPU accelerators can be easily supported by providing an architecture model description and new back ends can be added to support new architectures (e. g., Intel MIC). All these implementations are generated from the same high-level description.

c) Performance: competitive performance can be achieved using domain-specific code generators. In [15], [16], 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 competitive performance. On the Quadro FX 5800, the automatically generated CUDA code for the whole application has the same performance as the hand-tuned implementation. The execution of a single smoother is even faster compared to the hand-tuned implementation. This can be attributed to the fact that the

manual implementation was optimized for the Tesla C2050 architecture and no texturing memory was considered for that implementation. The automatically generated code selects texturing memory automatically for the Quadro FX 5800 due to improved bandwidth utilization for memory accesses with high locality. In case code generation takes kernels of the whole application into account, we believe that the same performance can be obtained.

C. 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 reduces global memory accesses by a factor of two and result in significantly faster execution (up to $2\times$). The polyhedral model [17] can serve here as a technique for fusing kernels while fulfilling data dependency constraints. All required information for the polyhedral model 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 declarative instead of using an imperative syntax. This delegates low-level details such as memory management for the different grid levels to the DSL framework and even 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) can be delegated 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 accelerator, 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.

VI. Conclusions

We have shown that automatic code generation is a useful tool to increase productivity for HPC applications. In the next years we want to establish these techniques in order to generate code also for large HPC clusters.

The presented domain-specific language and source-to-source compiler has been implemented in the HIPA^{cc} framework, which is available as open-source under https://sourceforge.net/projects/hipacc.

Acknowledgments

We thank Markus Stürmer for providing the hand-tuned manual implementation. This work is 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.

References

[1] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, “From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming,” *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2011.

[2] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, “Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers,” in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, Nov. 2011, pp. 9:1–9:12.

[3] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, “The Pochoir Stencil Compiler,” in *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, Jun. 2011, pp. 117–128.

[4] A. Baker, R. Falgout, T. Kolev, and U. M. Yang, “Scaling Hypre’s Multigrid Solvers to 100,000 Cores,” *High-Performance Scientific Computing*, pp. 261–279, 2012.

[5] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, “A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework,” *Computing*, vol. 82, no. 2, pp. 103–119, Jul. 2008.

[6] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, “An Auto-Tuning Framework for Parallel Multicore Stencil Computations,” in *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, Apr. 2010, pp. 1–12.

[7] M. Christen, O. Schenk, and H. Burkhart, “PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures,” in *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, May 2011, pp. 676–687.

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

[9] R. Fattal, D. Lischinski, and M. Werman, “Gradient Domain High Dynamic Range Compression,” in *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, Jul. 2002, pp. 249–256.

[10] W. L. Briggs, H. Van Emden, and S. F. McCormick, *A Multigrid Tutorial*. Society for Industrial And Applied Mathematics (SIAM), Jun. 2000, vol. 2.

[11] U. Trottenberg, C. W. Oosterlee, and A. Schüller, *Multigrid*. Academic Press, Dec. 2000.

[12] M. Stürmer, J. Treibig, and U. Rüde, “Optimising a 3D Multigrid Algorithm for the IA-64 Architecture,” *International Journal of Computational Science and Engineering*, vol. 4, no. 1, pp. 29–35, Nov. 2008.

[13] H. Köstler, M. Stürmer, and U. Rüde, “A Fast Full Multigrid Solver for Applications in Image Processing,” *Numerical Linear Algebra with Applications*, vol. 15, no. 2–3, pp. 187–200, Apr. 2008.

[14] I. Dietrich, R. German, H. Koestler, and U. Ruede, “Modeling Multigrid Algorithms for Variational Imaging,” in *Proceedings of the 21st Australian Software Engineering Conference (ASWEC)*. IEEE, Apr. 2010, pp. 224–234.

[15] R. Membarth, F. Hannig, J. Teich, M. Körner, and 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, May 2012, pp. 569–581.

[16] R. Membarth, F. Hannig, J. Teich, M. Körner, and 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, Jun. 2012, pp. 211–218.

[17] P. Feautrier and C. Lengauer, “Polyhedron Model,” in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1581–1592.