This is the author's version of the work. The definitive work was published in *Proceedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC), Munich, Germany, June 25–29, 2012.*

Automatic Optimization of In-Flight Memory Transactions for GPU Accelerators based on a Domain-Specific Language for Medical Imaging

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

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

***Abstract*—An efficient memory bandwidth utilization for GPU accelerators is crucial for memory bound applications. In medical imaging, the performance of many kernels is limited by the available memory bandwidth since only a few operations are performed per pixel. For such kernels only a fraction of the compute power provided by GPU accelerators can be exploited and performance is predetermined by memory bandwidth. As a remedy, this paper investigates the optimal utilization of available memory bandwidth by means of increasing in-flight memory transactions. Instead of doing this manually for different GPU accelerators, the required CUDA and OpenCL code is automatically generated from descriptions in a Domain-Specific Language (DSL) for the considered application domain. Moreover, the DSL is extended to also support global reduction operators. We show that the generated target-specific code improves bandwidth utilization for memory-bound kernels significantly. Moreover, competitive performance compared to the GPU back end of the widely used image processing library OpenCV can be achieved.**

***Keywords*-GPU; CUDA; OpenCL; memory bandwidth utilization; domain-specific language; code generation; medical imaging; global operators; reductions**

I. Introduction

Graphics cards hardware has evolved from being used solely as special hardware for one particular task (visualization) and serves as accelerator for general purpose applications nowadays. There are many reasons for this trend like the high raw processing power provided and the lower power budget required for GFLOPS per watt compared with standard processors. Hence, Graphics Processing Units (GPUs) have been consequently adopted in many application domains such as medical imaging [1].

However, not all applications from medical imaging benefit from the raw processing power of graphics cards. In image preprocessing, many algorithms are applied to an image in a chain, where each algorithms does only little computations. Such algorithms are memory bound and limited by the provided memory bandwidth of GPUs. To increase the performance of such memory bound kernels, this paper presents the automatic optimization of in-flight memory transactions by mapping multiple iteration points to the same thread. The optimizations are performed within the Heterogeneous Image Processing Acceleration (HIPA^{cc}) framework for medical image processing that allows programmers to describe imaging kernels in a DSL and generates low-level device-specific CUDA and OpenCL code from this description.

For the considered application domain, we identified three groups of operators that have to be supported by the DSL for medical imaging: a) point operators, b) local operators, and c) global operators. While we previously presented the description and mapping of point and local operators [2], [3], the notation for global reduction operators is introduced in this paper. Global operators are memory bound and, thus, are first class candidates to investigate the proposed techniques.

The transformations applied by our work are inspired by the work of Volkov [4], showing that high bandwidth utilization can be achieved at low occupancy—which is contrary to the *best practices* provided by hardware manufacturers. For code generation for global operators, we use a similar approach as in [5], where *compilette*-based binary code generation is presented for the Cell B.E., in order to generate device-dependent code at runtime. Also in [6], algorithmic *skeletons* are used for image processing applications targeting distributed memory systems.

The remainder of the paper is organized as follows: Section II introduces the HIPA^{cc} framework and the extension of the framework to support global reduction operators is described in Section III. Section IV discusses memory considerations for exploiting memory bandwidth, while Section V describes the code generation for increasing in-flight memory transactions. In Section VI, the proposed code transformations are evaluated. Finally, the paper is concluded in Section VII.

II. Image Processing Framework

In this paper, we use and extend the HIPA^{cc} framework to design image processing kernels [2], [3]. The framework uses a source-to-source compiler based on Clang [7] in order to generate low-level, optimized CUDA and OpenCL code for execution on GPU accelerators. The framework consists of built-in C++ classes that describe the following four basic components required to express image processing on an abstract level:

- *Image*: Describes data storage for the image pixels. Each pixel can be stored as an integer number, a floating point number, or in another format depending on instantiation of this templated class. The data layout is handled internally using multi-dimensional arrays.
- *Iteration Space*: Describes a rectangular region of interest in the output image, for example the complete image. Each pixel in this region is a point in the iteration space.

- *Kernel*: Describes an algorithm to be applied to each pixel in the *Iteration Space*.
- *Boundary Condition*: Describes the boundary handling mode when an *Image* is accessed out of bounds. Supported are *Undefined*, *Repeat*, *Clamp*, *Mirror*, and *Constant*.
- *Accessor*: Describes which pixels of an *Image* are *seen* within the *Kernel*, taking the *Boundary Condition* into account if specified. Similar to an *Iteration Space*, the *Accessor* defines an *Iteration Space* on an input image.
- *Mask*: Describes and stores the filter mask constants for local operators.

A. Example: Gaussian Filter

In the following, the HIPA^{cc} framework is illustrated using a Gaussian filter, smoothing an image. By doing so, the Gaussian filter reduces image noise and detail. This filter is a local operator that is applied to a neighborhood (σ) of each pixel to produce a smoothed image (see Equation (1)). The filter mask of the Gaussian filter as described in Equation (2) depends only on the size of the considered neighborhood (σ) and is otherwise constant for the image. Therefore, the filter mask is typically precalculated and stored in a lookup table to avoid redundant calculations for each image pixel.

$$I_{Out}(x,y) = \sum_{ox=-\sigma}^{+\sigma} \sum_{oy=-\sigma}^{+\sigma} G((x,y),(x+ox,y+oy)) * I_{In}(x+ox,y+oy) \quad (1)$$

$$G((x,y),(x',y')) = \frac{1}{2\pi\sigma^2} e^{-\frac{\|(x,y)-(x',y')\|}{2\sigma^2}} \quad (2)$$

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

```
1  class GaussianFilter : public Kernel<float> {
2    private:
3      Accessor<float> &Input;
4      Mask<float> &cMask;
5      const int size_x, size_y;
6
7    public:
8      GaussianFilter(IterationSpace<float> &IS, Accessor<
           float> &Input, Mask<float> &cMask, const int
           size_x, const int size_y) :
9        Kernel(IS), Input(Input), cMask(cMask), size_x(
             size_x), size_y(size_y)
10     { addAccessor(&Input); }
11
12     void kernel() {
13       const int ax = size_x >> 1;
14       const int ay = size_y >> 1;
15       float sum = 0.0f;
16
17       for (int yf = -ay; yf<=ay; yf++) {
18         for (int xf = -ax; xf<=ax; xf++) {
19           sum += cMask(xf, yf)*Input(xf, yf);
20         }
21       }
22       output() = sum;
23     }
24 };
```

Listing 1: Kernel description of the Gaussian filter.

In Listing 2, the input and output *Image* objects `IN` and `OUT` are defined as two-dimensional $W \times H$ grayscale images, having pixels represented as floating-point numbers (lines 10–11). The *Image* object `IN` is initialized with the `host_in` pointer to a plain C array (line 14). The Gaussian filter *Mask* object `GMask` is defined (line 17) and is initialized (line 18) for the filter size. Because of accessing neighboring pixels in the Gaussian filter, border handling is required. In line 21, a *Boundary Condition* object specifying mirroring as boundary mode for the filter size is defined. The region of interest `IsOut` contains the whole image (line 24) and the *Accessor* `AccIn` is defined on the input image taking the boundary condition into account (line 27). The kernel is initialized with the iteration space, accessor, and filter mask objects as well as filter size parameters *size_x* and *size_y* (line 30), and executed by a call to the `execute()` method (line 33). To retrieve the output image, the `host_out` pointer is assigned the *Image* object `OUT`, invoking the `getData()` operator (line 36).

```
1  const int width=1024, height=1024, size_x=3, size_y=3;
2
3  // pointers to raw image data
4  float *host_in = ...;
5  float *host_out = ...;
6  // pointer to Gaussian filter mask
7  float *filter_mask = ...;
8
9  // input and output images
10 Image<float> IN(width, height);
11 Image<float> OUT(width, height);
12
13 // initialize input image
14 IN = host_in; // operator=
15
16 // filter Mask for Gaussian filter
17 Mask<float> GMask(size_x, size_y);
18 GMask = filter_mask;
19
20 // Boundary handling mode for out of bounds accesses
21 BoundaryCondition<float> BcInMirror(IN, GMask,
       BOUNDARY_MIRROR);
22
23 // define region of interest
24 IterationSpace<float> IsOut(OUT);
25
26 // Accessor used to access image pixels with the
       defined boundary handling mode
27 Accessor<float> AccIn(BcInMirror);
28
29 // define kernel
30 GaussianFilter GF(IS, AccIn, GMask, size_x, size_y);
31
32 // execute kernel
33 GF.execute();
34
35 // retrieve output image
36 host_out = OUT.getData();
```

Listing 2: Example code using the Gaussian filter.

III. Global Reduction Operators

Global operators are widely used in image processing. In particular for medical imaging, global reduction operators are required to determine signal to noise ratio using autocorrelation, to normalize images, or to determine the similarity

of different images or of a series of images. Therefore, either the minimum or maximum pixel value (normalization) or the mean pixel value (signal-to-noise, similarity) of the whole image or a subregion of an image is required. Such operators are called *reduction* operators, *reducing* all data elements to a single value. Within image processing categorization, such operators are *global* operators, since they consume the pixels of the whole image as opposed to *local* operators consuming only the pixels within a neighborhood and *point* operators consuming only a single pixel. A reduction operator has the following properties:

Definition 1 (Global reduction operator (see [8])). *The reduce operation takes:*
1) a binary associative operator $\oplus$
2) with identity I,
3) and an ordered set $[A_0, A_1, \ldots, A_{n-1}]$ *of n elements.*
4) It returns the value $A_0 \oplus A_1 \oplus \ldots \oplus A_{n-1}$.

The binary associative operator may be the minimum, maximum, or sum of pixels with the corresponding identity (e. g., 0 for summation and integer data type). The ordered set of data elements is the *Image* in the DSL or a subregion as specified by an *Accessor*. The calculation of the returned value can be specified by a calculation rule for two elements. To reflect this in the DSL, a built-in *Global Reduction* class is provided: derived classes have to implement the virtual *reduce* method, taking two pixels as parameters and describing the reduction of these two pixels. The *Global Reduction* base class requires an *Image* or *Accessor* and the identity element as parameters, which have to be specified in the constructor of a global reduction by the programmer. Listing 3 shows an example of how to define a global reduction operator in the DSL, which can calculate the sum of pixels within an image or image region. Two constructors are provided taking an *Image* and *Accessor* as parameter, respectively.

```
1  template<typename data_t>
2  struct SumReduction : public GlobalReduction<data_t> {
3    public:
4      using GlobalReduction<data_t>::reduce;
5
6      SumReduction(Image<data_t> &img, data_t identity) :
7        GlobalReduction<data_t>(img, identity)
8      {}
9      SumReduction(Accessor<data_t> &img, data_t identity
           ) :
10       GlobalReduction<data_t>(img, identity)
11     {}
12
13     data_t reduce(data_t left, data_t right) {
14       return left + right;
15     }
16 };
```

Listing 3: Declaration of a global reduction: the reduce method defines the summation of two elements. The identity element is passed to the base class in the constructor.

The described global operator of Listing 3 can be instantiated using different data types as seen in Listing 4: in line 8, a sum reduction is defined for the *Image* `IN1` (storing integer data types), while the sum reduction defined in line 11 calculates the sum of the inner third of the *Image* `IN2` (storing floating point data types) as specified by the *Accessor* `AccIn`. The call to the *reduce()* method (line 14 and 15) applies the global reduction and returns the final reduced element. Thereby, the *reduce()* method provided by the DSL built-in class *Global Reduction* applies the user-provided calculation rule to all pixels of the *Image* and *Accessor*, respectively.

```
1  // input and output images
2  Image<int> IN1(width, height);
3  Image<float> IN2(width, height);
4  // accessor used to access inner third of image
5  Accessor<float> AccIn(IN2, width/3, height/3, width/3,
       height/3);
6
7  // define global reduction on Image
8  SumReduction<int> redSumImg(IN, 0);
9
10 // define global reduction on Accessor
11 SumReduction<float> redSumAcc(AccIn, 0.0f);
12
13 // execute reduction and retrieve result
14 int sum_img = redSumImg.reduce();
15 float sum_acc = redSumAcc.reduce();
```

Listing 4: Example code showing the usage of global reductions on Images and Accessors.

IV. GPU Memory Bandwidth Utilization

Although GPUs provide an immense compute power, it is hard to exploit this potential in many application domains due to an imbalance between memory bandwidth and peak performance. Providing a memory bandwidth of 144 GB/s, the NVIDIA Tesla C2050 can deliver 36 G (10^9) 32-bit data elements. These data elements result in 36 GFLOPS for operations with one operand from global memory, or in 18 GFLOPS for operations with two operands from global memory. The Multiply-Add (MAD) as well as the Fused Multiply-Add (FMA) operations used to characterize the peak performance require three operands from global memory, resulting in only 12 GFLOPS of the theoretically possible 1288 GFLOPS[1]. This is less than 1% of the peak performance. To achieve higher performance, data reuse is essential and the memory hierarchy has to be used wisely. Table I shows how many operations per data fetch are required to exploit both data bandwidth and compute performance at the same time. As reference for peak performance calculations, normal single-precision operations are considered instead of MAD/FMA operations. Hardware vendors use the MAD/FMA operations to calculate peak performance since these operation can be *counted* as two operations. Still, the number of operations per data fetch is enormous for global memory, ranging from 57–62 to 25–29 operations per data fetch for GPUs from AMD and NVIDIA, respectively. This gets even worse taking into account that less than 80 % of the theoretical peak memory bandwidth can be achieved in real systems. Using caches improves the situation a lot, but still about 8 and 4 operations have to be performed on the fastest L1 cache and local memory for AMD and NVIDIA GPUs. Only registers can provide the required bandwidth to feed the compute units steadily with data.

Global memory on GPUs has a latency of 400–600 cycles and whenever an instruction has to wait for data from global

[1]According to NVIDIA, counting one FMA as two operations and four operations per Special Function Unit (SFU) and cycle.

Table I: Arithmetic operations required per data fetch in order to exploit both peak performance and memory bandwidth for the Evergreen, Northern Island, Tesla, and Fermi GPU architectures.

	Evergreen	Northern Island	Tesla	Fermi
	Radeon HD 5870	Radeon HD 6970	Quadro FX 5800	Tesla C2050
GFLOPS†	1088	1352	311	515
Peak (GB/s)	153.6	176	102.4	144
Ops/fetch	56.67	61.45	24.30	28.61

† For single-precision operations (no MAD/FMA, no SFU).

memory, the thread stalls and waits until the data is available. This is in particular a problem since most kernels do only little calculations. Consider for example a simple kernel that adds only a constant to each pixel of an image: index calculation and the arithmetic operation itself take only a few cycles compared with the latency to fetch the pixel from global memory and to store the modified pixel back to global memory. To solve this dilemma, GPUs support massive multithreading: whenever the threads within a group of threads (*warp*) stall, the compute unit schedules the next warp. All warps (up to 48 for the Tesla C2050) are scheduled this way in a round-robin fashion. At that time, the warp that stalled first is ready to continue execution—the data has been fetched from global memory while the other warps have been scheduled. This way, execution of single threads has a high latency, but the throughput of the total workload is high.

Therefore, it is essential to utilize the available memory hierarchy effectively for memory bound kernels. This is in particular the case for the considered image processing kernels from medical imaging. Point operators perform a few operations per data fetched and the performance is limited by the global memory bandwidth. Similarly, local operators perform only little calculations per data fetch, but the pixels fetched for one output pixel can be reused for the local operator of neighboring pixels. To preserve global memory bandwidth, the data fetched for local operators can be cached either manually using local memory or automatically when utilizing a data path traversing a cache such as texture cache or L1/L2 cache. However, for most local operators global bandwidth is still the limiting factor. Global operators face the same challenge.

As a remedy for suboptimal global memory bandwidth utilization due to either scheduling overhead, or insufficient multithreading provided by the underlying hardware, we propose to increase the number of in-flight memory transactions by automatically mapping multiple iteration points of the *Iteration Space* to one thread. By doing so, multiple output image pixels are calculated per thread (pixels per thread, PPT). This reduces the parallelism, but increases the number of possible in-flight memory transactions and helps to hide global memory latency.

V. Code Generation

Based on the latest Clang compiler framework [7], our source-to-source compiler uses the Clang front end for C/C++ to parse the input files 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 Application Programming Interface (API) calls and target-specific device code in CUDA or OpenCL.

A. 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++).

B. Point Operators and Local Operators

For point and local operators, AST nodes of kernels described in the DSL are translated into 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. 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.

For mapping multiple output pixels to one thread, multiple iterations of the iteration space are mapped to the same thread and executed sequentially. This can be seen as a combination of loop unrolling and loop tiling in traditional compiler techniques. In order to do the same on the GPU, three steps are required: a) creating kernel code that contains the code for n iterations (loop unrolling), b) adjusting the indices within the unrolled code, and c) update the iteration space configuration (divide the iteration space by n). To preserve coalescing, iterations are always unrolled in the y-dimension by the framework and, hence, only the y-index has to be updated. Guards for unrolled iterations have to be added to ensure that only points within the iteration space are processed. For example, for a kernel working on an image subregion of 100×100 pixels, mapping 8 pixels to one thread means that $\lceil 100/8 \rceil = \lceil 12.5 \rceil = 13$ thread blocks are used for a configuration of 128×1, resulting in an iteration space of 128×104. The updated configuration of 128×104 is calculated in the host code before launching the kernel using a API call provided by the framework. Listing 5 shows how the resulting code looks like, emitting code for `PPT` iterations, adding necessary guards, and updating

index calculations. Note that no synchronization directives are allowed within guards—otherwise the GPU will hang due to a deadlock. In that case, the guard is terminated before the synchronization instruction is hit, and a new guard is added for the instructions after the barrier.

```
1  gid_x = get_global_id(0);
2  gid_y = get_global_id(1)*PPT;
3
4  // first iteration
5  // begin kernel code
6  ... = Input[gid_y * stride + gid_x];
7  // end kernel code
8
9  // second iteration
10 if (gid_y + 1 * get_local_size(1) < height) {
11   // begin kernel code
12   ... = Input[(gid_y+1) * stride + gid_x];
13   // end kernel code
14 }
15
16 // n-th iteration
17 if (gid_y + (PPT-1) * get_local_size(1) < height) {
18   // begin kernel code
19   ... = Input[(gid_y+PPT-1) * stride + gid_x];
20   // end kernel code
21 }
```

Listing 5: Mapping the calculation of multiple output pixels to one thread.

C. Global Operators

In contrast to the code generation for point and local operators, where the AST is translated into corresponding CUDA/OpenCL target code, we use a template-based code generation approach for global operators. Global reductions can be efficiently implemented based on parallel prefix sums (scans) [8] and efficiently mapped to GPUs [9]. In the framework, skeleton codes based on parallel prefix sums are provided that can reduce data sets. The available skeleton codes can be instantiated by providing a) the data type of the reduction set, b) providing the identity (as parameter), c) the calculation rule for reducing two elements, and d) information about the image region to be considered for reduction. The image region information includes image offsets, image stride, and the size of the subregion.

The skeleton codes are provided for both CUDA and OpenCL, and are optimized to exploit the underlying GPU architectures: offsets are adjusted in case the image was padded; idle threads are added for reductions on *Accessors* such that a coalesced global memory access pattern is granted; fast, local on-chip memory is utilized to perform the reduction of the loaded data with a bank-conflict free access pattern to the local memory; and synchronization is avoided when the parallelism within one work-group reaches lockstep granularity. In addition to these optimizations, the reduction code can map multiple input pixels to one thread. That is, a group of 128×1 threads can reduce more than a block of 128×1 pixels, for instance, a block of 128×16 pixels can be reduced by a group of 128×1 threads, increasing in-flight memory transactions. The amount of pixels mapped to one thread is configurable. Furthermore, different available memory object types on the GPU are supported, for example, the image can be stored to a *memory buffer* or *memory image* in OpenCL. The appropriate implementation is automatically selected by the framework.

For the generated skeleton code, the framework provides API calls that perform the reduction. Since a reduction requires typically several steps ($\mathcal{O}(log(n))$) to get the final result, intermediate memory is allocated by the runtime system and used for the reduction. Furthermore, only the first reduction step needs to care about the 2D data layout of images, subsequent reduction steps can use linear memory. Mapping several input pixels to one thread, the global operator can be implemented in two steps: a 2D reduction, followed by the final 1D reduction. Beginning with the Fermi architecture, NVIDIA added the possibility to synchronize globally (using thread fences). When creating reduction code for Fermi, a skeleton supporting thread fences is chosen (CUDA only), such that the reduction can be performed in a single step.

VI. Evaluation and Results

In this section, the effect of mapping multiple pixels to one thread for point, local, and global operators is evaluated. For evaluation, four GPUs are used, representing the two most recent GPU architectures from AMD and NVIDIA. For each sample point, 10 runs have been performed and only the minimum measured timing is considered. Only the execution time on the GPU is considered, that is, the runtime system overhead is not included in the timings. For AMD cards, the APP SDK version 2.6 is used and CUDA 4.1 for NVIDIA cards.

The achievable memory bandwidth on each card is determined using OpenCL API calls to copy data between different locations on the GPU. Table II shows the attained bandwidth for copying a memory junk of 2^{24} 4-byte values, corresponding to 64 MB of memory. It can be seen that the cards can only achieve 70–80 % of peak memory bandwidth. This is the bandwidth we can achieve copying an image of 4096×4096 pixels and serves as reference for further evaluation.

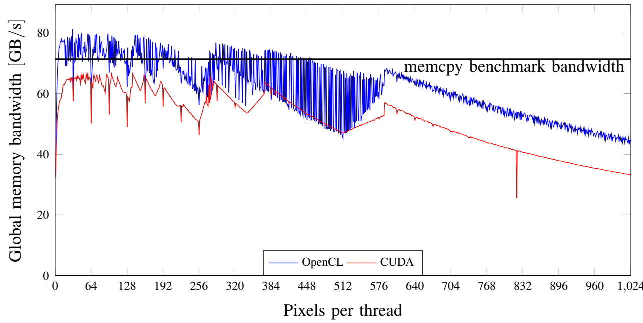
A. Global Operators

To evaluate global memory bandwidth for reductions, the `SumReduction` global operator from Listing 3 is used. The framework was extended to provide an exploration

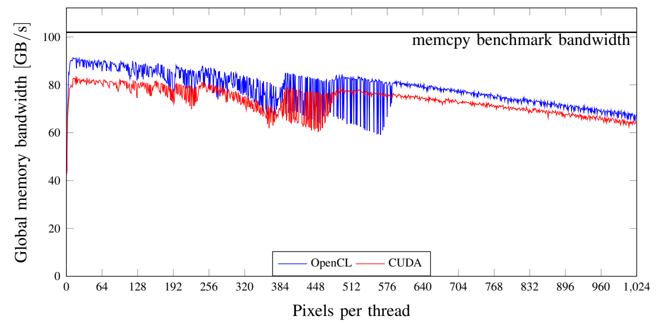
Table II: Memory bandwidth of the Evergreen, Northern Island, Tesla, and Fermi GPU architectures. Shown is the theoretical **peak** and the achievable (**memcpy**) memory bandwidth in GB/s.

	Evergreen	Northern Island	Tesla	Fermi
	Radeon HD 5870	Radeon HD 6970	Quadro FX 5800	Tesla C2050
Peak	154 GB/s	176 GB/s	102 GB/s	144 GB/s
Memcpy	119 GB/s	132 GB/s	71.4 GB/s	$102^{\dagger}$ GB/s
Percentage	77.3 %	75.0 %	69.7 %	$70.8^{\dagger}$ %

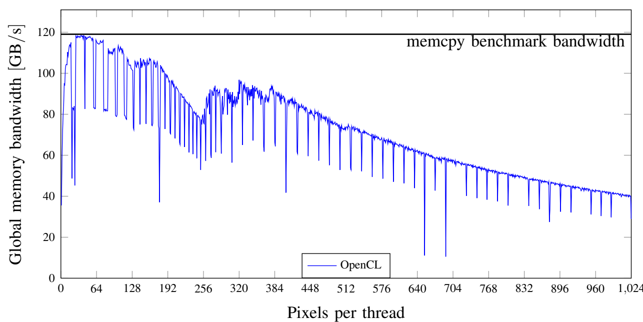
† Error-Correcting Code (ECC) memory turned on, 118 GB/s with ECC turned off (82.1 % of peak bandwidth).



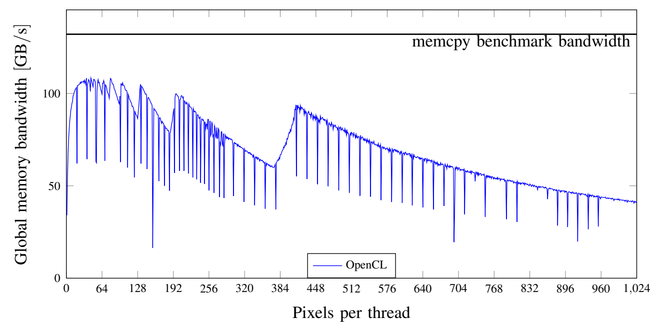
(a) Quadro FX 5800.



(b) Tesla C2050.



(c) Radeon HD 5870.



(d) Radeon HD 6970.

Figure 1: Throughput of the generated CUDA and OpenCL sources in GB/s for the **sum** reduction operator for an image of 4096×4096 pixels. Shown is the achieved bandwidth in dependence on the pixels per thread.

testbed for evaluating the effect of varying pixels per thread (PPT) for reductions. The exploration version of the reduction varies PPT from 1 to the image—or accessor—height, compiles the resulting kernels just-in-time and executes them. The results of this exploration are shown in Figure 1 for an image of 4096×4096 pixels. Only the results for PPT up to 1024 are shown since the bandwidth decreases further with increasing PPT. Without mapping multiple pixels to one thread, only half of the memcopy bandwidth is achieved for cards from NVIDIA and only one third for AMD cards. With increasing PPT, also the achieved bandwidth rises, before a peak for PPT between 16 and 64 is reached, and the achieved bandwidth declines again. Compared to an implementation with a 1:1 mapping between input pixels and threads, the performance can be more than doubled on NVIDIA cards and more than tripled on AMD cards.

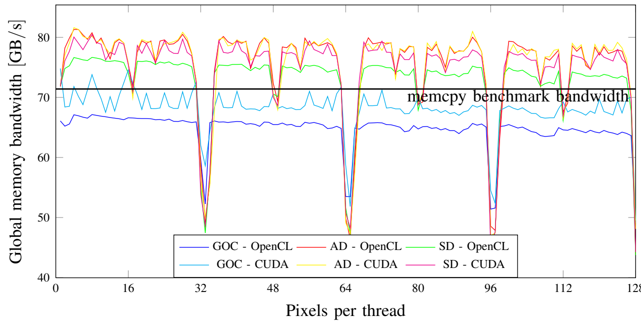
B. Point Operators

As reference for point operators, we consider three kernels typically encountered in medical imaging: a) a simple kernel for global offset correction, adding a constant to each pixel within an image (GOC), b) a kernel calculating pixel-wise the absolute difference of two images (AD), and c) a kernel calculating pixel-wise the square difference of two images (SD). Application of the presented sum reduction kernel on the output images of the AD and SD kernels calculates the quality measures sum of absolute difference (SAD) and sum of square difference (SSD) encountered in medical imaging. While GOC reads only one pixel per output pixel, the AD and SD kernels read two pixels per output pixel.

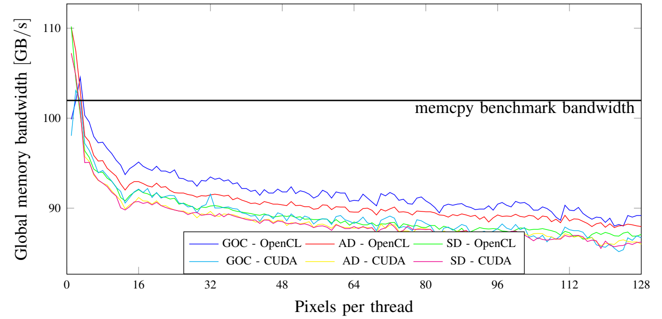
Similar to the evaluation of global operators, the number of pixels per thread is changed as shown in Figure 2. Here, the bandwidth utilization increases depending on the number of pixels read and the card used. Compared with the global reductions, the increase in memory bandwidth is only in the range of 10–20 %. On AMD GPUs, all point operators benefit from the optimization, while on NVIDIA cards, not all point operators benefit. The bandwidth on the Quadro FX 5800 drops when the pixels processed per thread is a multiple of 32. We attribute this to *partition camping*, when all memory requests hit the same memory bank. To avoid such situations, NVIDIA changed the memory assignment to banks in newer architectures.

C. Local Operators

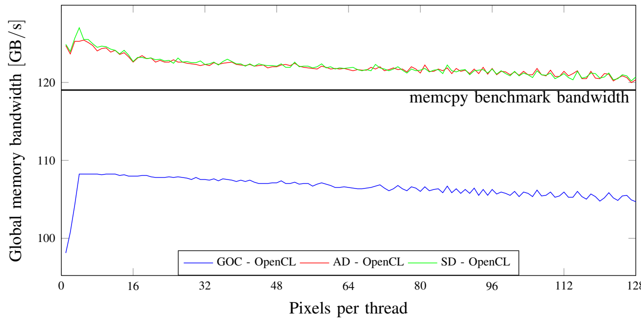
The Gaussian filter introduced in Section II is evaluated as a representative example for local operators. However, an implementation is used where the filter is separated into a row and column filter. The implementation for the Gaussian filter includes boundary handling and is compared against the implementations from the Open Source Computer Vision (OpenCV) library [10], one widely used library for image processing, providing support for GPU accelerators using CUDA. We use the most recent version 2.3.3 of OpenCV for our evaluation. The OpenCV implementation maps 8 input pixels to one thread. As reference we also compile



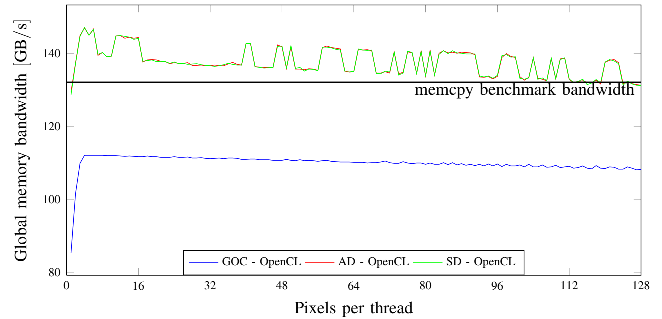
(a) Quadro FX 5800.



(b) Tesla C2050.



(c) Radeon HD 5870.



(d) Radeon HD 6970.

Figure 2: Throughput of the generated CUDA and OpenCL sources in GB/s for the **GOC**, **AD**, and **SD** point operators for an image of 4096×4096 pixels. Shown is the achieved bandwidth in dependence on the pixels per thread.

OpenCV with a 1:1 mapping. Table III, IV, V, and VI show the execution times for different boundary handling modes when multiple pixels are mapped to one thread for an image of 4096×4096 using a filter window size of 3×3. Again, the performance improves for most implementations of up to 30 %. Increasing the filter window size of the Gaussian filter to 5×5, no significant improvement can be observed for the AMD cards while the performance on NVIDIA benefits still from the optimization.

D. Discussion

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

Mapping multiple pixels to one thread is implemented transparently in the framework and can be enabled target-specific and context-specific. That is, how many pixels are mapped to one thread may depend on the target GPU as well as on the kernel characteristics. The latter can be obtained from the analysis of kernel code (number of memory accesses) as well as extracted from DSL metadata

Table III: Execution times in *ms* for the Gaussian filter from OpenCV on the **Tesla C2050** and our generated implementations using the **CUDA** and **OpenCL** back ends for an image of 4096×4096 pixels, mapping multiple output pixels to one thread (PPT).

	Clamp	Repeat	Mirror	Const.
OpenCV				
PPT: 1	10.36	16.07	15.68	11.47
PPT: 8	**4.95**	7.22	7.07	5.37
CUDA				
PPT: 1	7.62	7.64	7.47	7.49
PPT: 2	6.74	6.75	6.68	6.74
PPT: 4	6.29	6.34	6.09	6.09
PPT: 8	6.01	6.10	6.06	6.04
PPT: 16	6.05	6.27	6.05	6.00
PPT: 32	5.98	6.59	5.96	6.01
OpenCL				
PPT: 1	7.02	7.06	6.84	7.09
PPT: 2	6.01	6.04	6.22	5.93
PPT: 4	5.57	5.46	5.59	5.42
PPT: 8	5.26	5.21	5.26	5.18
PPT: 16	5.13	5.12	5.13	5.09
PPT: 32	5.08	**5.10**	**5.09**	**5.05**

Table IV: Execution times in *ms* for the Gaussian filter from OpenCV on the **Quadro FX 5800** and our generated implementations using the **CUDA** and **OpenCL** back ends for an image of 4096×4096 pixels, mapping multiple output pixels to one thread (PPT).

	Clamp	Repeat	Mirror	Const.
OpenCV				
PPT: 1	7.42	9.00	20.76	9.58
PPT: 8	**4.65**	**5.64**	10.24	**6.00**
CUDA				
PPT: 1	8.77	8.79	8.81	8.84
PPT: 2	7.66	7.68	7.69	7.73
PPT: 4	6.91	6.94	6.95	7.01
PPT: 8	6.55	6.59	6.59	6.68
PPT: 16	6.40	6.46	**6.44**	6.54
PPT: 32	6.86	7.05	6.94	7.03
OpenCL				
PPT: 1	12.80	12.81	12.84	12.83
PPT: 2	11.98	11.99	12.01	11.99
PPT: 4	12.00	12.02	12.05	12.03
PPT: 8	12.18	12.19	12.22	12.21
PPT: 16	12.35	12.38	12.42	12.39
PPT: 32	12.50	12.55	12.60	12.57

Table V: Execution times in *ms* for the Gaussian filter on the **Radeon HD 5870** for our generated implementations using the **OpenCL** back end for an image of 4096×4096 pixels, mapping multiple output pixels to one thread (PPT).

	Clamp	Repeat	Mirror	Const.
PPT: 1	6.99	7.72	7.01	7.90
PPT: 2	7.33	8.28	7.34	8.58
PPT: 4	6.78	7.60	6.78	8.07
PPT: 8	6.62	7.46	6.65	**7.95**
PPT: 16	**6.56**	**7.43**	**6.61**	7.98
PPT: 32	6.86	7.68	6.98	8.51

Table VI: Execution times in *ms* for the Gaussian filter on the **Radeon HD 6970** for our generated implementations using the **OpenCL** back end for an image of 4096×4096 pixels, mapping multiple output pixels to one thread (PPT).

	Clamp	Repeat	Mirror	Const.
PPT: 1	6.49	8.23	6.49	8.22
PPT: 2	6.76	8.36	6.74	8.28
PPT: 4	5.89	7.64	5.93	7.71
PPT: 8	5.77	7.42	5.79	**7.55**
PPT: 16	**5.70**	**7.40**	**5.75**	7.61
PPT: 32	6.07	7.77	6.16	8.32

(size of filter mask).

VII. Conclusions

In this paper, we presented the automatic optimization of in-flight memory transactions in order to maximize memory bandwidth utilization of GPU accelerators. Based on the description of medical imaging algorithms in a DSL, it was shown that optimized code can be generated for CUDA and OpenCL increasing the in-flight memory transaction by mapping multiple iteration points to the same thread. This was shown for point, local, and global operators. In addition, a new syntax for global reduction operators was introduced.

It was shown that the generated CUDA and OpenCL codes improve bandwidth utilization for memory-bound kernels significantly and that we get similar results compared to the GPU back end of the widely used image processing library OpenCV. The optimizations and DSL extension have been implemented in the presented HIPAcc framework, which is available as open-source under https://sourceforge.net/projects/hipacc.

References

[1] F. Xu and K. Mueller, “Real-Time 3D Computed Tomographic Reconstruction using Commodity Graphics Hardware,” *Physics in Medicine and Biology*, vol. 52, pp. 3405–3419, 2007.

[2] R. Membarth, A. Lokhmotov, and J. Teich, “Generating GPU Code from a High-level Representation for Image Processing Kernels,” in *Proceedings of the 5th Workshop on Highly Parallel Processing on a Chip (HPPC)*. Springer, Aug. 2011, pp. 270–280.

[3] 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.

[4] V. Volkov, “Better Performance at Lower Occupancy. Presentation at the GPU Technology Conference (GTC),” Sep. 2010.

[5] K. Brifault and H. Charles, “Efficient Data Driven Run-time Code Generation,” in *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR)*. ACM, Oct. 2004, pp. 1–7.

[6] C. Nicolescu and P. Jonker, “A Data and Task Parallel Image Processing Environment,” *Parallel Computing*, vol. 28, no. 7–8, pp. 945–965, Aug. 2002.

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

[8] G. Blelloch, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, Feb. 1993, ch. Prefix Sums and Their Applications, pp. 35–60.

[9] M. Harris, S. Sengupta, and J. Owens, *GPU Gems*. Addison-Wesley, Aug. 2007, ch. Parallel Prefix Sum (Scan) with CUDA, pp. 851–876.

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