Parallel Processing Letters
© World Scientific Publishing Company

HIGH-LEVEL PROGRAMMING OF STENCIL COMPUTATIONS ON MULTI-GPU SYSTEMS USING THE SKELCL LIBRARY

MICHEL STEUWER, MICHAEL HAIDL,
STEFAN BREUER, *and* SERGEI GORLATCH

*Department of Mathematics and Computer Science,
University of Muenster, Muenster, Germany*

Received April 2014
Revised July 2014

Communicated by Guest Editors

ABSTRACT

The implementation of stencil computations on modern, massively parallel systems with GPUs and other accelerators currently relies on manually-tuned coding using low-level approaches like OpenCL and CUDA. This makes development of stencil applications a complex, time-consuming, and error-prone task. We describe how stencil computations can be programmed in our SkelCL approach that combines high-level programming abstractions with competitive performance on multi-GPU systems. SkelCL extends the OpenCL standard by three high-level features: 1) pre-implemented parallel patterns (a.k.a. skeletons); 2) container data types for vectors and matrices; 3) automatic data (re)distribution mechanism. We introduce two new SkelCL skeletons which specifically target stencil computations – MapOverlap and Stencil – and we describe their use for particular application examples, discuss their efficient parallel implementation, and report experimental results on systems with multiple GPUs. Our evaluation of three real-world applications shows that stencil code written with SkelCL is considerably shorter and offers competitive performance to hand-tuned OpenCL code.

Keywords: Stencils, Manycores, GPU, OpenCL, Skeletons, SkelCL

1. Introduction

Stencil computations play an important role in a number of application domains including time-intensive scientific simulations, image processing and others. Modern manycore architectures comprising Graphics Processing Units (GPUs) and other accelerators provide potentially tremendous computing power for challenging applications including stencil computations.

However, current programming approaches for multi-GPU architectures are low level, the most popular examples being OpenCL [1] and CUDA [2]. Even for one GPU, these approaches require the programmer to explicitly manage the GPU's memory (including memory (de)allocations and data transfers to/from the system's main memory) and explicitly specify parallelism in the computation. This

Electronic version of an article published as Parallel Processing Letters, Volume 24, Issue 03, September 2014, 17 pages. DOI: 10.1142/S0129626414410059 ©World Scientific Publishing Company, Journal URL: http://www.worldscientific.com/worldscinet/ppl

leads to lengthy, low-level, complicated and, thus, error-prone code. For multi-GPU systems, programming with CUDA and OpenCL becomes even more complex, as both approaches require an explicit implementation of data exchange between the GPUs, as well as disjoint management of each GPU, including low-level pointer arithmetics and offset calculations. When implementing stencil computations, additional challenges arise, like handling out-of-bound memory accesses and achieving high performance by making efficient use of the fast but small local GPU memory.

In this paper, we present our SkelCL [16] approach to high-level, manycore programming, and we describe how it simplifies stencil programming and achieves competitive performance on multi-GPU systems. SkelCL extends the standard OpenCL by three high-level mechanisms:

1) computations are easily expressed using pre-implemented parallel patterns (a.k.a. *skeletons*);
2) memory management is simplified using *container data types* for vectors and matrices;
3) data movement in multi-GPU systems is handled automatically by SkelCL's *(re)distribution mechanism.*

For stencil computations, we extend SkelCL with two specialized skeletons: MapOverlap for simple stencil computations, and Stencil for more complex, in particular iterative, stencil applications.

This paper extends of our work presented at the first international workshop on high-performance stencil computations [3] with an real-world application study including experimental results. In Section 2 we introduce stencil computations and their programming on systems with GPUs. Section 3 presents our SkelCL library for high-level GPU programming. In the next two sections we discuss how SkelCL can be used for stencil computations on single- (Section 4) and multi-GPU systems (Section 5). We evaluate our approach using three real-world stencil applications in Section 6, before we compare our approach with related work and conclude in Section 7.

2. Stencils Using OpenCL

A *stencil computation* is a computational pattern on a multi-dimensional grid, where each point of the grid is updated (often iteratively) as a function of its neighboring points. Each point of the grid stores a application-specific values. The particular computation performed to update the values of each point depending on the values of the neighboring points is called the *stencil operation.* The neighboring points taken into account for a stencil operation constitute the so-called *stencil shape.*

Let us consider how stencil computations are implemented on systems with GPUs using the state-of-the-art OpenCL approach. Listing 1 presents the simplified structure of an OpenCL implementation of the Gaussian blur application [14] on one GPU, a typical stencil computation used in image processing for smoothing

```
1  kernel void gauss(global const char* in_img,
2                    global char* out_img, int w, int h) {
3    int i = get_global_id(0); int j = get_global_id(1);
4    if (i < w && j < h) {
5      char ul = (j-1 > 0 && i-1 > 0) ? in_img[((j-1)*w)+(i-1)] : 0;
6      ...
7      char lr = (j+1 < h && i+1 < w) ? in_img[((j+1)*w)+(i+1)] : 0;
8      out_img[j*w+i] = computeGaussianBlur(ul, ..., lr); } }
```

Listing 1. Structure of the OpenCL implementation of the Gaussian blur application.

images. Lines 5–7 show how the direct neighboring elements, e.g., the *upper left* (`ul`) neighbor, are accessed and passed to a function performing the Gaussian blur computation in line 8. Even in such a simple example many low-level details have to be considered by the developer for a correct implementation, like raw pointer handling, including index computations, and explicit out-of-bound accesses handling (e.g., in line 5).

The OpenCL version in Listing 1 is not efficient: the fast local GPU memory is not used and the control flow diverges heavily between different work items, which is disadvantageous on current GPU architectures. However, the corresponding optimizations require a deep knowledge of the GPU's architecture and must be programmed and tuned manually and are, therefore, a complicated task for application developers. If the program is to be used on a multi-GPU system then the application developer has to additionally implement and optimize the explicit data distribution across multiple GPUs and the communication between them.

3. The SkelCL Skeleton Library

We develop SkelCL [16] – a skeleton library for computing systems with Graphics Processing Units (GPUs). By providing skeletons (constructs implementing common patterns of parallel programming) on container data types, SkelCL alleviates programming of systems with GPUs: parallelism is expressed implicitly using skeletons, and memory management is performed automatically by the SkelCL implementation which is built on top of OpenCL. The especially tricky programming of multi-GPU systems is greatly simplified by SkelCL's data distribution mechanism which automatically moves data between multiple GPUs.

Algorithmic Skeletons In OpenCL, computations are expressed as *kernels*, e.g., as in Listing 1, which are executed in a parallel manner on a GPU; the application developer must explicitly specify how many instances of a kernel are launched. In addition, kernels usually take pointers to GPU memory as input and contain program code for reading/writing single data items from/to it. These pointers have to be used carefully, because no boundary checks are performed by OpenCL.

To shield the application developer from these low-level programming issues, SkelCL extends OpenCL by introducing high-level programming patterns, called *(algorithmic) skeletons*. Formally, a skeleton is a higher-order function that executes one or more user-defined (so-called *customizing*) functions in a pre-defined parallel manner, while hiding the details of parallelism and communication from the user [9].

The current version of SkelCL provides four basic skeletons (*Map*, *Reduce*, *Zip*, and *Scan*) and three more advanced skeletons (*Allpairs*, *MapOverlap*, and *Stencil*). Due to lack of space, we only describe the first two basic skeletons here; the other basic skeletons are described in detail in [16]. The stencil-oriented skeletons MapOverlap and Stencil are described in detail in Section 4.

The Map skeleton applies a unary function f to each element of an input vector $[v_1, v_2, \ldots, v_n]$, i.e.:

$$\mathit{map}\ f\ [v_1, v_2, \ldots, v_n] = [f(v_1), f(v_2), \ldots, f(v_n)]$$

The Reduce skeleton computes a scalar value from a vector using an (associative and commutative) binary operator $\oplus$, i. e.

$$\mathit{red}\ \oplus\ [v_1, v_2, \ldots, v_n] = v_1 \oplus v_2 \oplus \ldots \oplus v_n$$

The programmer customizes suitable skeletons by application-specific functions which work on basic data types and, therefore, they are often much simpler than kernels that work with pointers. Skeletons can be executed on both single- and multi-GPU systems; on a multi-GPU system, the calculation specified by a skeleton is performed automatically on all GPUs of the system.

Container Data Types SkelCL offers two container data types – vector and matrix – which are transparently accessible by both the CPU and the GPUs. The *vector* abstracts a one-dimensional contiguous memory area while the *matrix* provides an interface to a two-dimensional memory area.

The advantage of the container data types in SkelCL as compared with OpenCL is that data transfers between the memories of the CPU and GPUs are performed implicitly. Before execution, the SkelCL implementation ensures that all input containers' data is available on all participating GPUs. This may result in automatic data transfers from the CPU memory to GPU memory, which in OpenCL would require explicit programming. Similarly, before any data is accessed on the CPU, the implementation of SkelCL ensures that this data on the CPU is up-to-date. This may result in data transfers from the GPU which are performed automatically too. Thus, the container classes free the programmer from low-level programming of memory allocation (on GPU) and data transfers between CPU and GPU.

While all data transfers are performed implicitly by SkelCL, advanced application developers may sometimes desire an explicit control over the data transfers between CPU and GPU. For this purpose SkelCL offers a set of APIs to explicitly initiate and control the data transfers to and from the GPUs.

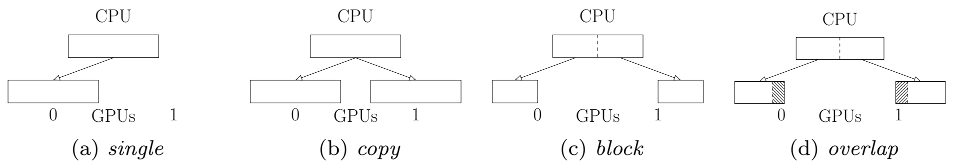

Fig. 1. Distributions of a vector in SkelCL.

(Re)Distribution Mechanism For multi-GPU systems, SkelCL's parallel container data types (vector and matrix) abstract from the separate memory areas on multiple GPUs, i. e., container's data is accessible by all GPUs. To simplify the partitioning of a container on multiple GPUs, SkelCL offers the concept of *distributions* that specify how containers are distributed among the GPUs. It allows the application developer to abstract from explicitly managing memory ranges which are shared or partitioned across multiple GPUs.

Four kinds of distributions are currently available to the application developer in SkelCL: *single*, *copy*, *block*, and *overlap* (see Fig. 1 for illustration on a system with two GPUs). If set to the *single* distribution (Fig. 1a), container's whole data is stored on a single GPU (the first GPU if not specified otherwise). The *copy* distribution (Fig. 1b) copies container's entire data to each available GPU. With the *block* distribution (Fig. 1c), each GPU stores a contiguous, disjoint block of the container. The *overlap* distribution (Fig. 1d) is used for the MapOverlap and Stencil skeletons: it stores on both GPUs a common block of data from the border between the GPUs. For the matrix data type the block and overlap distributions are currently limited for pragmatic reasons to partitioning along the rows dimension.

By default a regular partitioning based on the number of GPUs available is chosen for the block and overlap distribution. In heterogeneous systems combining different types of GPUs or GPUs integrated with CPUs, this partitioning is not optimal and leads to imbalance. SkelCL allows the user to specify the size of each block separately, therefore, advanced users can tune their applications for a particular system.

The application developer can set the distribution of containers explicitly or, otherwise, every skeleton selects a default distribution for its input and output containers. The distribution of a container can be changed at runtime: this implies data exchanges between multiple GPUs and the CPU, which are performed by SkelCL implicitly. Implementing such data transfers in standard OpenCL is a cumbersome task: data has to be downloaded to the CPU before it can be uploaded to other GPUs, including the corresponding length and offset calculations; this results in a lot of low-level code which is completely hidden when using SkelCL.

The consistency of user-specified distributions with skeletons, expectations is checked and enforced at runtime: e. g., if a block distributed container is passed as input to a Stencil skeleton, which expects its input to be overlap distributed, the distribution is changed at runtime by the skeletons implementation.

4. New Skeletons for Stencils

While the particular stencil operations vary for different applications, the overall structure of stencil computations stays the same. Therefore, stencil computations can be implemented as a skeleton which can be customized by the application developer with a stencil operation and stencil shape. To simplify the development of stencil applications, we introduce two specialized skeletons in SkelCL: *MapOverlap* and *Stencil*. While MapOverlap supports simple stencil computations, the Stencil skeleton provides support for more complex stencil computations with more complex stencil shapes and (possibly) iterative execution.

The MapOverlap Skeleton Listing 2 shows the implementation of the Gaussian blur using the *MapOverlap* skeleton. The MapOverlap skeleton applies a given function `func` (defined in lines 2–6) to each element of an input matrix `in_img` while taking the neighboring elements within the range $[-d, +d]$ in each dimension into account. Here, d is the second parameter and the last parameter defines how the skeleton handles out-of-bound memory accesses (line 7). The `get` helper function is used to easily access the neighboring elements. The indexes are specified relative to the current element, e. g. to access the element on the left the function call `get(in_img, -1, 0)` is used.

```
1  MapOverlap<char(char)> gauss(
2   "char func(char_matrix_t in_img) {
3      char ul = get(in_img, -1, -1);
4      ...
5      char lr = get(in_img, +1, +1);
6      return computeGaussianBlur(ul,..., lr);}",
7   1, BorderHandling::NEUTRAL(0));
8  Matrix<char> input = loadImage();
9  output = gauss(input);
```

Listing 2. Implementation of Gaussian blur using the MapOverlap skeleton

Special handling is necessary when accessing elements out of the boundaries of the matrix, e.g., when the item in the top-left corner of the matrix accesses elements above and left of it. The MapOverlap skeleton can be configured to handle such out-of-bound memory accesses in two possible ways: 1) a specified neutral value is returned; 2) the nearest valid value inside the matrix is returned. In Listing 2, the first option is chosen and 0 is provided as neutral value.

The Stencil Skeleton Listing 3 shows the implementation of an iterative stencil application simulating heat transfer. This application simulates heat spreading from one location and flowing throughout a two-dimensional simulation space.

The application developer specifies the function describing the computation (line 2–6), as well as the extents of the stencil shape (line 7) and the out-of-bound han-

```
1  Stencil<char(char)> heatSim(
2   "char func(char_matrix_t in) {
3     char lt = get(in, -1, -1);
4     char lm = get(in, -1,  0);
5     char lb = get(in, -1, +1);
6     return computeHeat(lt,lm,lb);}",
7   StencilShape(1, 0, 1, 1),
8   BorderHandling::NEUTRAL(255));
9  Matrix<char> simSpace = init();
10 output = heatSim(100, simSpace);
```

Listing 3. Implementation of heat simulation using the Stencil skeleton

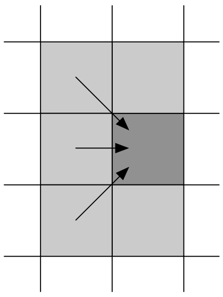

Fig. 2. Stencil shape for heat transfer simulation

dling (line 8). The stencil shape's extents are specified using four values for each of the directions: up, right, down, and left. In the example in Listing 3, the heat flows from left to right, therefore, no accesses to the elements to the right are necessary and the stencil space's extents are specified accordingly (note the 0 in line 7 representing the extent to the right). Figure 2 illustrates this situation: the dark gray element is updated by using the values from the left. The specified stencil shape's extent is highlighted in light gray. In our current implementation, the user has to explicitly specify the stencil shape's extents, which is necessary for performing the out-of-bound handling on the GPU. In future work, we plan to automatically infer the stencil shape from the customizing function using source code analysis in order to avoid inconsistencies and free the user from specifying this information explicitly. To iterate the heat transfer simulation for one hundred steps, we specify the number of iterations to perform when executing the skeleton (line 10). In the future, we plan to allow a user-specified function to check a condition and stop the iterations.

Sequence of Stencil Operations Many real-world applications perform different stencil operations in a sequence, like the popular *Canny algorithm* [14] which is used for detecting edges in images. For the sake of simplicity we consider a simplified version, which applies the following steps: 1) a noise reduction operation is applied, e. g., a Gaussian blur; 2) an edge detection operator like the Sobel filter is applied; 3) the so-called non-maximum suppression is performed, where all pixels in the image are colored black except pixels being a local maximum; 4) a threshold operation is applied to produce the final result. A more complex version of the algorithm performs the edge tracking by hysteresis as an additional step.

In SkelCL, each single step of the Canny algorithm can be expressed using the Stencil skeleton. The threshold operation performed as the last step, does not need access to neighboring elements, because the user function only checks the value of a single pixel. The implementation of the Stencil skeleton automatically uses the implementation of the simpler (and thus faster) Map skeleton when the user specifies a stencil shape whose extents are 0 in all directions.

```
 1  Stencil<Pixel(Pixel)> gauss(...);
 2  Stencil<Pixel(Pixel)> sobel(...);
 3  Stencil<Pixel(Pixel)> nms(...);
 4  Stencil<Pixel(Pixel)> threshold(...);
 5
 6  StencilSequence<Pixel(Pixel)> canny(
 7    gauss, sobel, nms, threshold);
 8
 9  Matrix<Pixel> input = loadImage();
10  output = canny(1, input);
```

Listing 4. Structure of the Canny algorithm implemented by a sequence of skeletons.

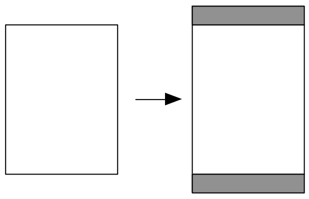

Fig. 3. The MapOverlap skeleton prepares a matrix by copying data on the top and bottom.

To implement the Canny algorithm in SkelCL, the single steps can be combined as shown in Listing 4. The individual steps are defined in lines 1–4 and then combined to a sequence of stencils in lines 6 and 7; this sequence is then executed in line 10.

Implementation In order to achieve high performance, our implementations of both the MapOverlap and the Stencil skeleton use the GPU's fast local memory. Both implementations perform the same basic steps on the GPU: 1) the data is loaded from the global memory into the local memory; 2) the user-defined function is called for every data element by passing a pointer to the element's location in the local memory; 3) the result of the user-defined function is written into the global memory. Although both implementations perform the same basic steps, different strategies are used for loading the data from the global into the local memory.

The MapOverlap skeleton prepares the input matrix on the CPU before uploading it to the GPU: padding elements are added to avoid out-of-bounds memory accesses to the top and bottom of the input matrix, as shown in Figure 3. This slightly enlarges the input matrix, but it reduces branching on the GPU due to avoiding some out-of-bound checks. In SkelCL, a matrix is stored row-wise in memory on the CPU and GPU, therefore, it would be complex and costly to add padding elements on the left and right of the matrix. For handling out-of-bound accesses for these regions, the boundary checks are performed on the GPU.

The Stencil skeleton has to use a different strategy in order to enable the usage of different out-of-bound handling modes and stencil shapes when using several Stencil skeletons in a sequence. As an example, consider two stencils in a sequence where the first defines a stencil shape with a neutral element 0 and the second defines a neutral element 1. This cannot be implemented by padding the input matrix as done by the implementation of the MapOverlap skeleton. Therefore, in the implementation of the Stencil skeleton no padding elements are added on the CPU, but rather all out-of-bounds accesses are handled on the GPU. This slightly increases branching in the code, but enables a more flexible usage of the skeleton.

5. Targeting Multi-GPU Systems

The support of systems with multiple OpenCL devices is one of the key features of SkelCL. By using distributions, SkelCL completely frees the user from error-prone and low-level explicit programming of data (re)distributions on multiple GPUs.

The MapOverlap skeleton uses the overlap distribution with *border regions* in which the elements calculated by a neighboring device are located. When it comes to iteratively executing a skeleton, data has to be transferred among devices between iteration steps, in order to ensure that data is up-to-date for the next iteration step. As the MapOverlap skeleton does not explicitly support iterations, its implementation is not able to exchange data between devices besides a full down- and upload of the matrix. This data exchange has to be performed after each iteration.

The Stencil skeleton supports iterative execution and exchanges only the elements from the border region, rather than performing a full down- and upload of the matrix. Using the Stencil skeleton in multiple iteration steps can be performed before exchanging data by enlarging the number of elements in the border region. The user can specify the number of iterations between *device synchronizations*, where all border regions are updated with elements from the corresponding inner border regions of the neighboring device. Data is exchanged by default after each iteration, however, there may be cases in which a different number of iterations between device synchronizations may result in better performance (see Section 6).

Figure 4 shows how the device synchronization is performed. Only elements from the inner border regions are downloaded and stored as `std::vector`s in a `std::vector`. Within the outer vector, the inner vectors are swapped pair-wise on the host, so that the inner border regions can be uploaded in order to replace the out-of-date border regions.

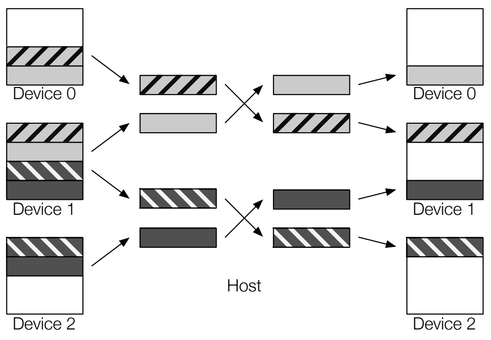

Fig. 4. Device synchronization for three devices. Equally patterned and colored chunks represent the border regions and their matching inner border region. After the download of the appropriate inner border regions, they are swapped pair-wise on the host. Then the inner border regions are uploaded in order to replace the out-of-date border regions.

6. Evaluation

For evaluating the MapOverlap and Stencil skeleton implementations, we study three real-world stencil applications:

1) the Gaussian blur, a popular noise reduction technique in image processing,
2) the Canny algorithm for detecting edges in images, and
3) the Finite-Difference-Time-Domain (FDTD) Method [18] for random lasing simulations from the field of optical physics.

These three applications have different characteristics. The Gaussian blur applies a single stencil computation, possibly iteratively, for reducing the noise in images. The Canny edge detection algorithm consists of a sequence of stencil operations which are applied once to obtain the final result. The FDTD application performs a large number of iterations, where in each iteration three stencil operations are performed.

We compare the performance of our MapOverlap and Stencil skeletons using an input image of size 4096×3072. The measurements run on a Tesla S1070 computing system with 4 GPUs, each providing 4 GB of memory and 240 compute units per GPU. Altogether 200 runs were performed for each configuration and the average was calculated; to reduce measuring inaccuracy, the best and worst 5% measurements were not considered.

Gaussian Blur using a single GPU Figure 5 shows the total runtime of the Gaussian blur using: 1) a naïve OpenCL implementation using global memory (see Listing 1), 2) an optimized OpenCL version using local memory, 3) the MapOverlap (see Listing 2), and 4) the Stencil skeleton based implementation for different sizes of stencil shape, correspondingly. We observe that on larger stencil shape sizes, MapOverlap and Stencil outperform the naïve OpenCL implementation by over 60%. The optimized OpenCL version is 5% faster than MapOverlap and 10% faster than Stencil for small stencil shapes and only 3—5% faster for larger stencil shapes.

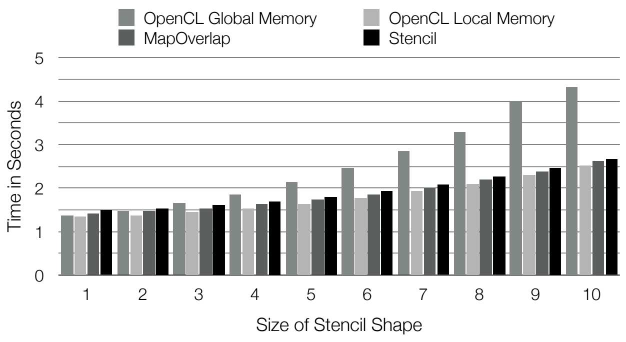

Fig. 5. Single GPU runtime of the Gaussian blur using a naïve OpenCL implementation, an optimized OpenCL version and SkelCL's MapOverlap and Stencil skeletons.

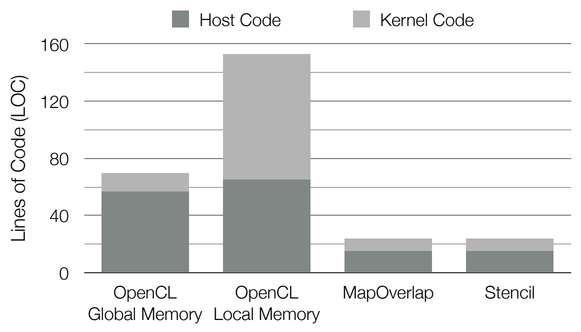

Fig. 6. Lines of code (LOCs) of the Gaussian blur using an OpenCL version with global memory, an optimized OpenCL version with local memory and SkelCL's MapOverlap and Stencil skeletons.

The implementation based on the Stencil skeleton is slower than the implementation using the MapOverlap skeleton for small stencil shapes. However, this disadvantage becomes negligible for stencil shapes larger than 5. The runtime difference is due to the increased branching in the kernel function of the Stencil skeleton's implementation when copying data into the local memory.

Figure 6 shows the program sizes in lines of code (LOC) for the four implementations. The application developer needs 57 lines of OpenCL host code and 13 LOCs for performing a Gaussian blur with global memory. When using local memory, more arguments are passed to the kernel, increasing the host-LOCs to 65. The kernel function copies elements necessary for the calculation of a work-group into the local memory, which requires 88 LOCs including explicit out-of-bounds handling and complex index calculations. The implementations using the MapOverlap and Stencil skeletons are similar and both require only 15 LOCs host code and 9 LOCs kernel code to perform a Gaussian blur. The source code of the two SkelCL implementations remains the same when using multi-GPU systems. This is an important advantage of SkelCL over the OpenCL implementations of the Gaussian blur which are single-GPU only. Additional LOCs would be required to enable multi-GPU support in these versions.

Gaussian Blur using multiple GPUs Figure 7 shows the speedup achieved on the Gaussian blur using the Stencil Skeleton on up to four GPU devices using different sizes of the stencil shape. Using multiple GPUs for this application is only beneficial for stencil shapes larger than 4. For an increasing size of the stencil shape and, therefore, the complexity of the computation, the overhead of managing multiple devices is hidden. This leads to a maximum speedup when using a stencil shape of size 20 of 1.90 for two devices, 2.66 for three devices, and 3.34 for four devices. The Gaussian blur is usually used with small stencil shapes but similar applications from the field of computer vision can make use of large stencil shapes, e. g., feature extraction and object tracking [14, 17].

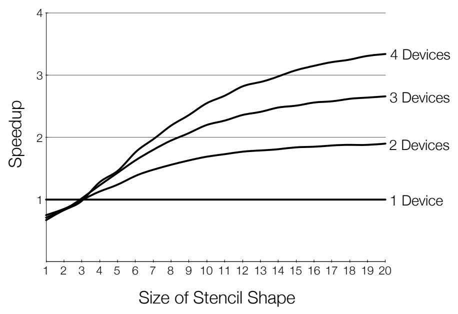

Fig. 7. Speedup of the Gaussian Blur application on up to four GPUs.

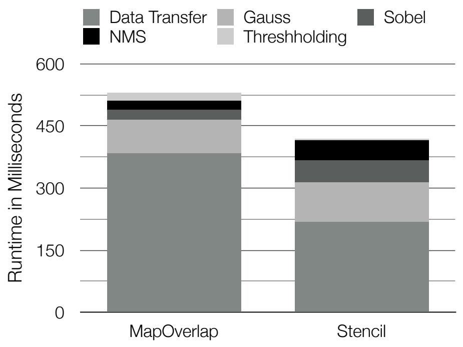

Fig. 8. Single GPU runtime of the Canny algorithm implemented with the MapOverlap and Stencil skeletons.

Canny edge detection Figure 8 shows the absolute runtime of the Canny algorithm (Listing 4) on a single GPU. As the MapOverlap skeleton adds padding elements to the matrix, the matrix has to be downloaded, resized and uploaded again to the GPU at each step of the sequence. This additional work leads to an increased time for data transfers as compared to the Stencil skeleton. The computations without the data transfer are 2.1 and 2.2 times faster when using the MapOverlap skeleton as compared to using the Stencil skeleton. This performance difference is mainly due to the different strategies used to load elements form the global into the local memory. Overall, when performing sequences of stencil operations, the Stencil skeleton avoids unnecessary copy operations and therefore leads to a better performance. When performing the Canny algorithm, Stencil outperforms MapOverlap by 21%.

Finite-Difference-Time-Domain (FDTD) Method for Random Lasing Simulations As our third application study we use a simulation from the field of optical physics, where the propagation of light through a medium is simulated.

In the simulation two fields, the electric field $\vec{E}$ and the magnetic field $\vec{H}$, are iteratively updated using stencils computations. The Maxwell's equations are the basic equations describing electrodynamic processes in nature and are used here to describe the light propagating through a non-magnetic (*dielectric*) medium.

$$\vec{\nabla}\vec{E}\,(\vec{r},t) = 0, \tag{1}$$

$$\vec{\nabla}\vec{H}\,(\vec{r},t) = 0, \tag{2}$$

$$\frac{\partial\vec{H}\,(\vec{r},t)}{\partial t} = -\frac{1}{\mu_0}\vec{\nabla}\times\vec{E}\,(\vec{r},t)\,, \tag{3}$$

$$\frac{\partial\vec{D}\,(\vec{r},t)}{\partial t} = \frac{1}{\epsilon_0}\vec{\nabla}\times\vec{H}\,(\vec{r},t)\,, \tag{4}$$

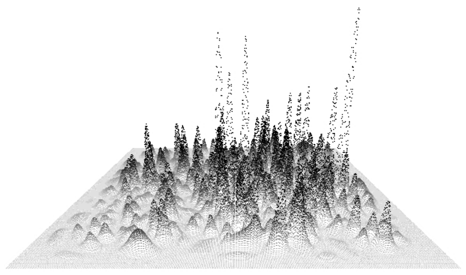
Fig. 9. The image shows a 3D representation of the intensity for the 2D electric field as computed by the SkelCL FDTD implementation after 60 000 iterations.

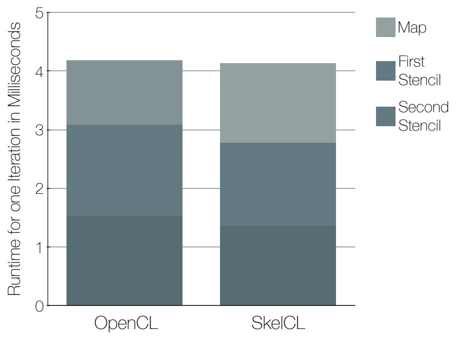

Fig. 10. Runtime for one iteration of the FDTD application.

Eq. (1)-(4) show the Maxwell's equations consisting of four coupled partial differential equations (PDEs). To couple the polarisation of a medium $\vec{P}$ to the electric field, Eq. (5) is introduced:

$$\vec{E}\left(\vec{r},t\right) = \frac{\vec{D}\left(\vec{r},t\right) - \vec{P}\left(\vec{r},t,\vec{N}\right)}{\epsilon_0 \epsilon_r\left(\vec{r}\right)} \tag{5}$$

Here $\vec{N}$ is the induced energy distribution in the medium using the model proposed in [11]. The parameters μ_0, ϵ_0 and ϵ_r describe the permeability and permittivity of free space and the relative permittivity of the dielectric medium.

To solve this set of coupled PDEs, the Finite-Difference-Time-Domain Method (short FDTD) [19] can be used. Here we use a form of FDTD where the electric and magnet field are discretized within a n-dimensional regular grid. $\vec{E}$ and $\vec{H}$ are shifted against each other by a half grid-cell. This allows the calculation of the new values by computing finite differences between two values of the grid. Using the FDTD method, we implemented a simulation of the effect of random lasing on a nano-meter scale [5] for our evaluation.

Figure 9 shows a visualization of the electric field (and the field intensity) after about $1\,ps$ of simulation time equal to 60 000 iterations. The shown field distribution can be found also in [4, 15, 18].

We implemented a two-dimensional version using SkelCL as well as a manually tuned OpenCL implementation. To solve the PDEs (3) and (4), two separated three-point stencil computations are performed and one map computation for the gain-model is necessary. Eq. (1) and (2) are implicitly solved by the FDTD method [19]. Listing 5 shows the SkelCL code of the application: in every iteration first the energy distribution is updated (line 11) using a map skeleton (defined in line 1); then the first stencil (defined in line 2) updates the electric field $\vec{E}$ by combining a single element of $\vec{E}$ with three elements of the magnetic field $\vec{H}$ (line 12); and finally the

```
1   Map<float4(float4)>     updateEnergyDist(...);
2   Stencil<float4(float4)> updateEField(...);
3   Stencil<float4(float4)> updateHField(
4     "float4 func(float4_matrix_t E, float4_matrix_t H) { ... }");
5
6   Matrix<float4> N; // energy distribution in the medium
7   Matrix<float4> E; // E (electric) field
8   Matrix<float4> H; // H (magnetic) field
9
10  for (...) { // for each iteration
11    updateEnergyDist(out(N), N, out(E));
12    updateEField(out(E), H, E);
13    updateHField(out(H), E, H); }
```

Listing 5. Source code of the FDTD application in SkelCL

second stencil (defined in line 3) updates $\vec{H}$ by combining a single element of $\vec{H}$ with three elements of $\vec{E}$ (line 13).

Please note that the two stencil computations require both fields ($\vec{E}$ and $\vec{H}$) as input. To implement this, we use the *additional argument* feature of SkelCL (see [16] for details) which allows the additional field to be passed to skeletons on execution (see line 12 and 13). The additional arguments are passed unchanged to the customizing function of the skeleton, therefore, the function customizing the stencil in line 4 now accepts $\vec{H}$ as a second parameter. This feature greatly increases the flexibility of applications written in SkelCL.

In the evaluation we used a 2048×2048 sized matrix with a spatial resolution of 100 cells per μm. This matrix corresponds to a square-shaped medium with the edge length of $20.1\,\mu m$. The medium size is actually smaller than the matrix size because of the border handling. To provide a physically correct simulation, the borders of the magnet field must be treated specially. The Stencil skeleton provides sufficient functionality to allow for such border handling in the computation code.

We compared our SkelCL based implementation to a handwritten, fine-tuned OpenCL implementation which is based on [12]. The OpenCL version is specifically designed for modern Nvidia GPUs. In particular, it exploits the L1 and L2 caches of the Nvidia Fermi and Kepler architecture and does not explicitly make use of the local memory. We performed the experiments on a system with a modern Nvidia K20c Kepler GPU with 5GB memory and 2496 compute cores. Figure 10 shows the median times of a simulation time of $1\,ps$ equal to 60 000 iterations. The SkelCL version slightly outperforms the OpenCL version by 2%. The two stencil skeletons achieve ∼10% faster runtimes than the corresponding OpenCL kernels but the map skeleton is ∼20% slower, because it reads and writes all elements exactly once, while the customized OpenCL kernel does not write back all elements. For this application it seems beneficial to make direct usage of the local memory as our implementation of the Stencil skeleton does, instead of relying on the caches of the hardware, as the OpenCL implementation does.

7. Conclusion and Related Work

In the paper, we describe how stencil computations are programmed in our SkelCL approach that combines high level of programming abstraction with competitive performance on multi-GPU systems. We introduce two SkelCL skeletons for stencil computations – MapOverlap and Stencil – and we discuss their efficient parallel implementation, and report experimental results using three real-world stencil applications. We demonstrate that when executing a single stencil shape once, the MapOverlap skeleton performs best; in all other cases, the Stencil skeleton is the better choice regarding both user comfort and performance. Both skeletons offer a high level of programming abstraction together with a competitive performance on multiple devices, and yield much simpler codes than when using OpenCL.

Several approaches aiming at simplifying GPU programming exist. *SkePU* [8] provides a vector class similar to our `Vector` class, but unlike SkelCL it does not support different kinds of data distribution on multi-GPU systems. SkelCL provides a more flexible memory management than SkePU, as data transfers can be expressed by changing data distribution settings. Similar to SkelCL there exists a MapOverlap skeleton in SkePU, whose implementation adapts to the execution hardware using autotuning techniques [7]. *Thrust* [10] provides two vector types similar to the vector type of the C++ Standard Template Library. While these types refer to vectors stored in CPU or GPU memory, respectively, SkelCL's vector data type provides a unified abstraction for CPU and GPU memory. Thrust also contains data-parallel implementations of higher-order functions, similiar to SkelCL's skeletons. SkelCL adopts several of Thrust's ideas, but we are not limited to CUDA-capable GPUs and we support multiple GPUs. Both SkePU and Thrust provide no explicit support for iterative stencil computations as presented in this paper.

Several projects focus on stencil computations on GPUs. PATUS [6] is a code generation and tuning framework for stencil computations. It can generate optimized code for multicore processors and a single GPU. PARTANS [13] is a code generation and autotuning framework which automatically distributes and optimizes stencil computations on multiple GPUs, by searching for optimal parameters for a given hardware architecture. These specialized approaches can only be applied to stencil computations, whereas SkelCL is a general-purpose approach.

8. Acknowledgments

We would like to thank the reviewers for their detailed and valuable comments, as well as Nvidia Corporation for their generous hardware donation used in our experiments.

References

[1] *The OpenCL Specification*, 2012. Version 1.2.

[2] *NVIDIA CUDA C Programming Guide*, 2013. Version 5.5.

[3] S. Breuer, M. Steuwer, and S. Gorlatch. Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems. In A. Größlinger and H. Köstler, editors, *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, pages 15–21, Vienna, Austria, 2014.

[4] H. Cao, J. Xu, D. Zhang, S. Chang, S. Ho, E. Seelig, X. Liu, and R. Chang. Spatial confinement of laser light in active random media. *Physical review letters*, 84(24):5584–7, 2000.

[5] H. Cao, Y. Zhao, S. Ho, E. Seelig, Q. Wang, and R. Chang. Random Laser Action in Semiconductor Powder. *Physical Review Letters*, 82(11):2278–2281, 1999.

[6] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS 2011)*, pages 676–687, 2011.

[7] U. Dastgeer, J. Enmyren, and C. W. Kessler. Auto-tuning SkePU: A Multi-Backend Skeleton Programming Framework for Multi-GPU Systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 25–32, New York, NY, USA, 2011. ACM.

[8] J. Enmyren and C. Kessler. SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. In *Proceedings 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010)*, pages 5–14, 2010.

[9] S. Gorlatch and M. Cole. Parallel skeletons. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1417–1422. Springer, 2011.

[10] J. Hoberock and N. Bell. Thrust: A Parallel Template Library, 2009. Version 1.1.

[11] X. Jiang and C. Soukoulis. Time dependent theory for random lasers. *Physical review letters*, 85(1):70–3, 2000.

[12] S. Knitter, M. Kues, M. Haidl, and C. Fallnich. Linearly polarized emission from random lasers with anisotropically amplifying media. *Optics Express*, 21(25):31591–31603, 2013.

[13] T. Lutz, C. Fensch, and M. Cole. PARTANS: An autotuning framework for stencil computation on multi-GPU systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):59:1–59:24, 2013.

[14] M. Nixon and A. S. Aguado. *Feature Extraction & Image Processing, Second Edition.* Academic Press, 3rd edition, 2012.

[15] P. Sebbah and C. Vanneste. Random laser in the localized regime. *Physical Review B*, 66(14):1–10, 2002.

[16] M. Steuwer and S. Gorlatch. SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems. In M. Victor, editor, *Parallel Computing Technologies - 12th International Conference (PaCT 2013)*, volume 7979 of *Lecture Notes in Computer Science*, pages 258–272. Springer, 2013.

[17] F. Toledo-Moreo, J. Martinez-Alvarez, and J. Ferrandez-Vicente. Hand-based interface for augmented reality. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 291–292, April 2007.

[18] A. Yamilov, X. Wu, H. Cao, and A. Burin. Absorption-induced confinement of lasing modes in diffusive random media. *Optics Letters*, 30(18):2430–2432, 2005.

[19] K. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Transactions on Antennas and Propagation*, 1966.