Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems

Stefan Breuer
stefan.breuer@uni-muenster.de

Michel Steuwer
michel.steuwer@uni-muenster.de

Sergei Gorlatch
gorlatch@uni-muenster.de

Departement of Mathematics and Computer Science
University of Münster, Germany

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, which makes it a complex, time-consuming, and error-prone task. We describe how stencil computations can be programmed in our SkelCL approach that combines high level of programming abstraction 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 manycore systems with multiple GPUs.

Keywords

Stencils, Manycores, GPU, OpenCL, Skeletons, SkelCL

1. INTRODUCTION

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

However, the current programming approaches for manycore architectures are low level, the most popular examples being OpenCL [1] and CUDA [2]. These approaches require the programmer to explicitly manage 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 leads to lengthy, low-level, complicated and, thus, error-prone code. For multi-GPU systems, programming with CUDA and OpenCL is 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 [8] 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 OpenCL standard 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 are 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 computations.

The paper is organized as follows. 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 in single- (Section 4) and multi-GPU systems (Section 5). We evaluate our approach using two real-world stencil computations 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 computation 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 set of application-dependent values. The computation performed to update the values of each point is called the *stencil operation.* A stencil operation updates the value of a point depending on the values of the neighboring points. The points taken into account for a stencil operation are defined by the *stencil shape.*

Let us consider how stencil computations are implemented on manycore systems with GPUs using the state-of-the-art

```
 1  kernel
 2    void sobel(global const char* in_img,
 3               global char* out_img,
 4               int w, int h) {
 5    int i = get_global_id(0);
 6    int j = get_global_id(1);
 7
 8    if (i < w && j < h) {
 9      char ul = (j-1 > 0 && i-1 > 0)
10          ? in_img[((j-1)*w)+(i-1)] : 0;
11      ...
12      char lr = (j+1 < h && i+1 < w)
13          ? in_img[((j+1)*w)+(i+1)] : 0;
14
15      out_img[j*w+i] =
16          computeSobel(ul, ..., lr);
17    }
18  }
```

Listing 1: Structure of the OpenCL implementation of Sobel edge detection

OpenCL standard. Listing 1 presents the structure of an OpenCL implementation of the Sobel operator on one GPU, a typical stencil computation used in image processing for detecting edges in images. Lines 9–13 show how the direct neighboring elements, e.g., the *upper left* (`ul`) neighbor, are accessed and passed to a function performing the Sobel operation in line 16. Many low-level details have to be considered for a correct implementation, like raw pointer handling, including index computations (e.g., line 10), and explicit out-of-bound accesses handling (e.g., line 9).

The OpenCL version is obviously correct, but 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 GPUs and the communication between them.

3. THE SKELCL SKELETON LIBRARY

We develop SkelCL [8] – a skeleton library for programming systems with Graphics Processing Units (GPUs). By providing skeletons 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 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 original 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* [5]. 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 [5].

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 [8].

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 binary operator $\oplus$, i. e.

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

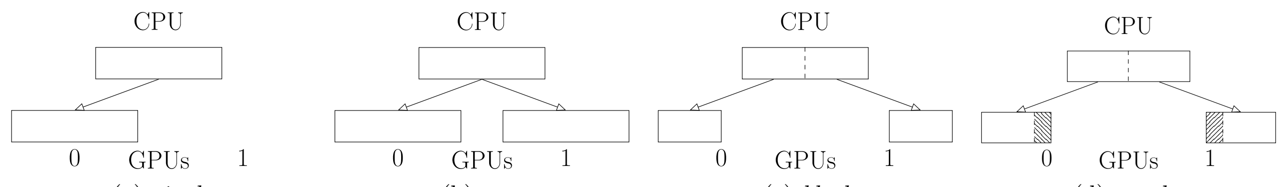
In SkelCL, rather than writing low-level kernels, the application developer customizes suitable skeletons by application-specific functions which work on basic data types and, therefore, they are often much simpler than kernels that work with pointers. Skeletons can be executed on both single- and multi-GPU 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 classes – 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. All computations in SkelCL accept containers as their input and output. Before execution, the SkelCL implementation ensures that all input containers' data is available on all participating GPUs. This may result in implicit (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 implicit data transfers from the GPU which are performed automatically too. Thus, the container classes free the programmer from low-level operations like memory allocation (on GPU) and data transfers between CPU and GPU.

While all data transfers are performed implicitly by SkelCL we understand that advanced application developers want fine grained control over the data transfers between CPU and GPU. For that purpose SkelCL offers a set of APIs developers can use to explicitly initiate and control the data transfer to and from the GPU.



(a) *single* (b) *copy* (c) *block* (d) *overlap*

Figure 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 each GPU. To simplify the partitioning of a container on multiple GPUs, SkelCL supports the concept of *distribution* that specifies how a container is 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.

The application developer can set the distribution of containers explicitly or every skeleton selects a default distribution for its input and output containers otherwise. The distribution of a container can be changed at runtime: this implies data exchanges between multiple GPUs and the CPU, which are performed by the SkelCL implementation 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.

4. NEW SKELETONS FOR STENCILS

The idea of our approach is that while the stencil operation varies for different applications, the overall structure of stencil computations stays the same. Therefore, stencil computations can be implemented as a skeleton which is customized by the application developer with a particular stencil operation and particular 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.

Listing 2 shows the implementation of the Sobel edge detection 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

```
1  MapOverlap<char(char)> sobel(
2   "char func(const char* in_img) {
3       char ul = get(in_img, -1, -1);
4       ...
5       char lr = get(in_img, +1, +1);
6       return computeSobel(ul,..., lr);}",
7   1, Padding::NEUTRAL, 0);
8
9  output = sobel(input);
```

Listing 2: Implementation of Sobel edge detection using the MapOverlap skeleton

```
Stencil<char(char)> heatSim(
 "char func(const char* in) {
     char lt = get(in, -1, -1);
     char lm = get(in, -1,  0);
     char lb = get(in, -1, +1);
     return computeHeat(lt, lm, lb); }",
 StencilShape(1, 0, 1, 1),
 Padding::NEUTRAL, 255);

output = heatSim(100, input);
```

Listing 3: Implementation of heat simulation using the Stencil skeleton

neighboring elements within the range $[-d, +d]$ in each dimension into account. Here, d is the second parameter (line 7) and two additional parameters define how the skeleton handles out-of-bound memory accesses (line 8). A helper function (`get`) 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, -1, 0)` is used.

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.

Simple stencil computations with a regular stencil shape can easily be expressed using the MapOverlap skeleton. For more complex stencil computations, e. g. iterative stencils, we introduce the more advanced *Stencil* skeleton.

The MapOverlap 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.

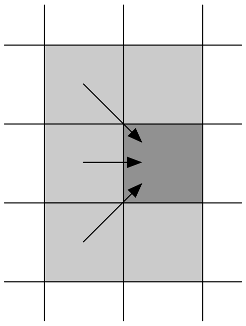


Figure 2: Stencil shape for heat transfer simulation

The application developer specifies the function (line 2–6) describing the computation and, therefore, the stencil shape, as well as the stencil shape's extents (line 7) and the out-of-bound handling (line 8). The stencil shape's extents are specified using four values for each of the directions: up, right, down, left. In the example in Listing 3, the heat flows from left to right, therefore, no accesses to 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's extents from the customizing function using source code analysis in order to free the user from specifying this information explicitly.

Many stencil applications apply a stencil multiple times for a fixed number of iterations, or until a certain condition is met. For example, 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 the user to specify a custom function which checks a condition to stop the iterations.

The MapOverlap skeleton can be configured to handle out-of-bounds accesses by returning the nearest valid value of the input matrix. Another distinction can be made regarding iterations and sequences of stencil operations: using elements of the **initial**, user-provided input matrix or using elements of the **current** step's input matrix, which already was updated during earlier stencil operations. The Stencil skeleton can be configured to handle out-of-bounds accesses in both ways, thus offering three possible ways, including the neutral value. For each of them, there is an own kernel function, loading appropriate elements into local memory.

The Stencil Skeleton.

Sequence of Stencil Operations.

Many real-world applications perform different stencil operations in a sequence. Let us consider the popular *Canny algorithm* which is used for detecting edges in images. For the sake of simplicity we consider a simplified version, which applies the following stencil operations in a sequence: first, a noise reduction operation is applied, e.g., a Gaussian filter; second, an edge detection operator like the Sobel filter

```
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  output = canny(1, input);
```

Listing 4: Structure of the Canny algorithm as implemented with a sequence of skeletons.

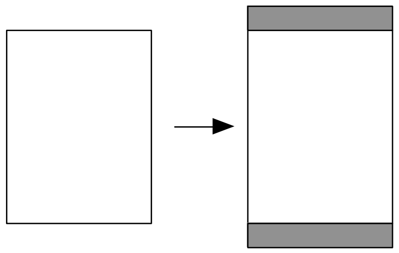


Figure 3: The MapOverlap skeleton prepares a matrix by copying data on the top and bottom.

is applied; third, the so-called non-maximum suppression is performed, where all pixels in the image are colored black except pixels being a local maximum; finally, 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. This results in detecting some weaker edges, but even without this additional step the algorithm usually achieves good results.

In SkelCL, each single step of the Canny algorithm can be expressed using the Stencil skeleton. The last step, threshold operation, does not need access to neighboring elements, as the user threshold function only checks the value of the current pixel. Therefore, this step can be expressed using SkelCL's simpler Map skeleton. The Stencil skeleton's implementation automatically uses the simpler Map skeleton's implementation when the user specifies a stencil shape which extents are 0 in all directions.

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 line 6 and 7. During execution (line 9), the stencil operation are performed in the order which is specified when creating the *StencilSequence* object.

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: first, the data is loaded from the global memory into the local memory; then, the user-defined function is called for every data element by passing a pointer to the element's location in the local memory; finally, the result of the user-defined function is copied back into the global memory.

Although both implementations perform the same basic steps, different strategies are implemented 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 appended; they are used 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. To avoid 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 padding modes and stencil shapes when using several Stencil skeletons in a sequence. As an example, consider two stencil shapes in a sequence where the first shape defines a neutral element 0 and the second defines a neutral element 1. This cannot be implemented using MapOverlap's implementation strategy. Therefore, Stencil does not append padding elements on the CPU, but rather manages all out-of-bounds accesses on the GPU, which slightly increases branching.

5. TARGETING MULTI-GPU SYSTEMS

The implicit and automatic support of systems with multiple OpenCL devices is one of the key features of SkelCL. By using distributions, SkelCL completely liberates 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 for the next iteration step is up-to-date. 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. In addition, data exchange has to be performed after each iteration. We can enlarge the number of elements in the border regions and perform multiple iteration steps on each device before exchanging data. However, this introduces redundant computations, such that a trade-off between data exchange and redundant computations has to be found.

For the Stencil skeleton, the user can specify the number of iterations between *device synchronisations*, where all border regions are updated with elements from the corresponding inner border regions of the neighboring device. The border regions are sized by default in such a way that the specified number of iterations can be performed without leading to incorrect results. However, there may be cases in which a different number of iterations between device synchronizations may result in better performance. Therefore, Stencil offers the user the possibility to specify that number. Please note that the implementation of the Stencil skeleton only exchanges elements from the border region and does not perform a full down- and upload of the matrix, as the MapOverlap skeleton does.

Figure 4 shows the device synchronization. Only the appropriate elements in the inner border region 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.

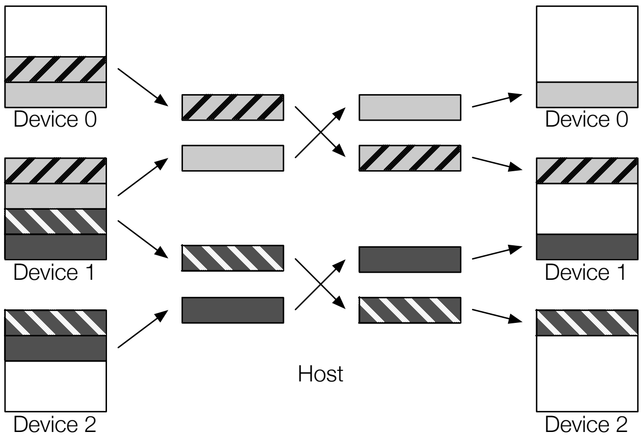


Figure 4: Device synchronization for three devices. Equally patterned and colored chunks represent the border regions and their matching inner border region. After the download of the appropriate inner border regions, they are swapped pair-wise on the host. Then the inner border regions are uploaded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

For the first iteration after a device synchronization, there are as many work-items on the GPU active as there are total elements on the device. As the first and last rows of the border regions become invalid within an iteration, the corresponding work-items become inactive in the following iteration step. This is done by using an offset and by reducing the number of total work-items when launching the OpenCL kernel. The Stencil's four kernel functions (one for each out-of-bounds handling mode and one for the adapted Map skeleton) can be used for both single- and multi-GPU systems.

6. EVALUATION

For evaluating our two skeleton implementations, we study two stencil applications: 1) the Gaussian blur, a popular noise reduction technique in image processing, and 2) the Canny algorithm for detecting edges in images. These two applications have different characteristics. The Gaussian blur applies a single stencil computation, possibly iterated multiple times, 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. For each application, 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, accessing this memory with 102 GB/s, and 240 compute units per GPU running at 1.44 GHz. The GPUs are connected to the host system with a quad-core CPU (Intel E5520, 2.26 GHz) and 12 GB of main memory. 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.

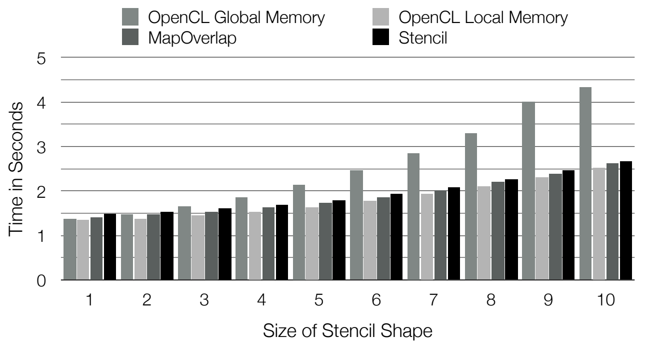


Figure 5: Runtime of the Gaussian blur using a naïve OpenCL implementation with global memory, an OpenCL version using local memory and SkelCL's MapOverlap and Stencil skeletons.

Gaussian Blur with a single iteration.

Figure 5 shows the total runtime of the Gaussian blur using: 1) a naïve OpenCL implementation using global memory, 2) an optimized OpenCL version using local memory, and 3) the MapOverlap, and 4) the Stencil skeletons for different sizes of stencil shape, correspondingly. We observe that on larger stencil shape sizes, MapOverlap and Stencil outperform the naïve OpenCL implementation by 65% and 62%, respectively. The optimized OpenCL version, which copies all necessary elements into local memory prior to calculation, is 5% faster than MapOverlap and 10% faster than Stencil for small stencil shapes. When increasing the stencil shape size, this disadvantage is reduced to 3% for MapOverlap and 5% for Stencil with stencil shape's extent of 10 in each direction.

As expected, the Stencil skeleton's implementation is slower for small stencil shapes than the MapOverlap skeleton's, up to 32% slower for an stencil shape size of 1. However, this disadvantage is reduced to 4.2% for an stencil shape size of 5 and becoming negligible for bigger stencil shape sizes. Due to the increased branching in Stencil's kernel function, one might expect a worse runtime for the Stencil skeleton. As the ratio of copying into local memory decreases in comparison to the number of calculations when enlarging the stencil shape's extents, the Stencil skeleton kernel function's runtime converges to the MapOverlap skeleton's. The Stencil skeleton's disadvantage is also due to its ability to manage multiple stencil shapes and explicitly support the use of iterations. While both features are not used in this use case, they incur some overhead for the Stencil skeleton as compared to the MapOverlap skeleton for simple stencil computations.

Figure 6 shows the program sizes (in lines of code) 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, some more arguments are passed to the kernel, increasing the host-LOCs to 65, while the LOCs for the kernel function, which copies all necessary elements for a work-group's calculation into local memory, requires 88 LOCs with explicit out-of-bounds handling and complex index calculations. MapOverlap and Stencil are similar to use and both require only 15 LOCs host code and 9 LOCs kernel code to perform a Gaussian blur. The support for multi-

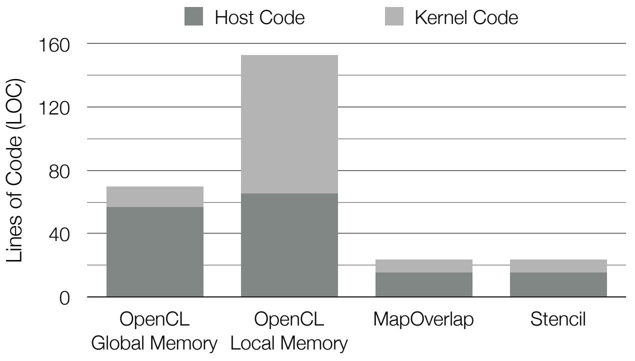


Figure 6: Lines of code (LOCs) of the Gaussian blur using a naïve OpenCL implementation with global memory, an optimized OpenCL version using local memory and SkelCL's MapOverlap and Stencil skeletons.

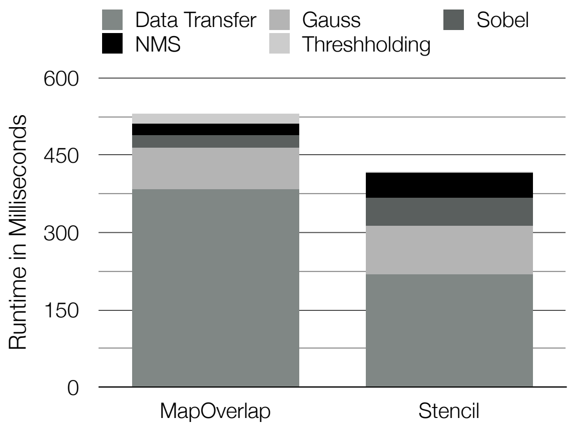


Figure 7: Runtime of the Canny algorithm implemented with the MapOverlap and Stencil skeletons.

GPU systems is implicitly given when using SkelCL's skeletons, such that the kernel remains the same as for one-GPU systems. This is an important advantage of SkelCL over the OpenCL implementations of the Gaussian blur which are single-GPU only, and they require additional LOCs when fitting to multi-GPU environments.

The implementations using MapOverlap and Stencil are only $5 - 10\%$ slower than an optimized OpenCL implementation of the Gaussian blur while being much shorter than the OpenCL version.

Canny edge detection.

Figure 7 shows the absolute runtime of the Canny algorithm (Listing 4). As the MapOverlap skeleton appends padding elements to the matrix, the matrix has to be downloaded, resized and uploaded again to the GPU between each step of the sequence. This additional work to an increased time for data transfers. The Gaussian blur with a stencil shape extent of 2, as well as the Sobel filter and the non-maximum suppression with a stencil shape of 1, are 2.1 to 2.2 times faster when using MapOverlap. However, the threshold operation, which is expressed as the Map skeleton in the Stencil sequence, is 6.8 times faster than MapOverlap's threshold operation. Overall, when performing sequences of stencil operations, the Stencil skeleton reduces the number of copy operations and therefore leads to a better overall performance. When performing the Canny algorithm, Stencil outperforms MapOverlap by 21%.

Gaussian Blur using multiple GPUs.

Figure 8 shows the speedup achieved on the Gaussian blur using `Stencil` on up to four devices. The higher the computational complexity for increasing size of stencil shape, the better the overhead is hidden, leading to a maximum speedup of 1.90 for two devices, 2.66 for three devices, and 3.34 for four devices.

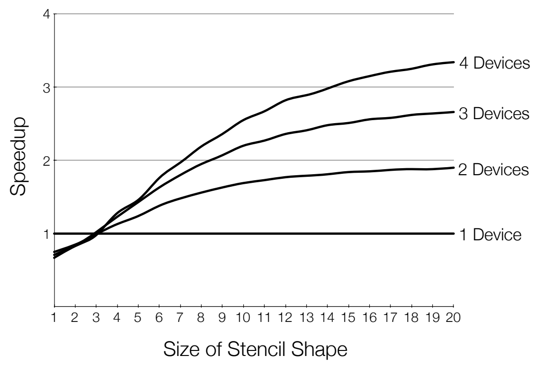


Figure 8: Speedup on up to four GPUs.

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. We demonstrate that when executing a single stencil shape once, the MapOverlap skeleton should be used; in all other cases, the Stencil skeleton is the better choice regarding both user comfort and performance. Both skeletons meet SkelCL's requirements of offering high levels of programming abstraction together with a competitive performance on multiple devices, and yield much shorter and simpler codes than when using OpenCL.

For future work, we want to study the applicability of our approach for stencil applications from different fields, like physical simulations, or solving partial differential equations. To enable more applications to use SkelCL, we want to introduce a three-dimensional data structure and adopt the existing skeletons to it. Furthermore, we are interested in building a performance model for our skeletons to better understanding and predicting the runtime of skeleton based applications on different target architectures.

Several approaches aiming at simplifying GPU programming exist. *SkePU* [4] 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. *Thrust* [6] 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 it is not limited to CUDA-capable GPUs and supports multiple GPUs. Both SkePU and Thrust provide no explicit support for stencil computations.

Several projects focus on stencil computations on GPUs. PATUS [3] is a code generation and tuning framework for stencil computations. It can generate optimized code for multicore processors and a single GPU. PARTANS [7] is a code generation and autotuning framework for stencil computations on multiple GPUs. It automatically distributes and optimizes stencil computations on multiple GPUs, by searching for optimal parameters for a given hardware architecture. These specialized domain-specific approaches can only be applied to stencil computations, whereas SkelCL is a general-purpose approach.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments, as well as NVIDIA for their generous hardware donation used in our experiments.

9. REFERENCES

[1] *The OpenCL Specification*, November 2012. Version 1.2.
[2] *NVIDIA CUDA C Programming Guide*, July 2013. Version 5.5.
[3] 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 IEEE International*, pages 676–687, 2011.
[4] 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)*, 2010.
[5] S. Gorlatch and M. Cole. Parallel skeletons. In *Encyclopedia of Parallel Computing*, pages 1417–1422. 2011.
[6] J. Hoberock and N. Bell. Thrust: A Parallel Template Library, 2009. Version 1.1.
[7] 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, 2013.
[8] 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 Berlin Heidelberg, 2013.