Using the SkelCL Library for High-Level GPU Programming of 2D Applications

Michel Steuwer, Sergei Gorlatch, Matthias Buß, and Stefan Breuer

University of Münster, Münster, Germany
{michel.steuwer,gorlatch}@uni-muenster.de

Abstract. Application programming for GPUs (Graphics Processing Units) is complex and error-prone, because the popular approaches — CUDA and OpenCL — are intrinsically low-level and offer no special support for systems consisting of multiple GPUs. The SkelCL library offers pre-implemented recurring computation and communication patterns (skeletons) which greatly simplify programming for single- and multi-GPU systems. In this paper, we focus on applications that work on two-dimensional data. We extend SkelCL by the matrix data type and the MapOverlap skeleton which specifies computations that depend on neighboring elements in a matrix. The abstract data types and a high-level data (re)distribution mechanism of SkelCL shield the programmer from the low-level data transfers between the system's main memory and multiple GPUs. We demonstrate how the extended SkelCL is used to implement real-world image processing applications on two-dimensional data. We show that both from a productivity and a performance point of view it is beneficial to use the high-level abstractions of SkelCL.

1 Introduction

Application programming for GPUs (Graphics Processing Units) is complex and error-prone. The popular programming models for systems with GPUs – CUDA and OpenCL [2,4,5] – require the programmer to explicitly manage GPU's memory, including (de)allocations and data transfers to/from the system's main memory. This leads to lengthy, low-level, complex and thus error-prone code. For emerging systems with multiple GPUs, CUDA and OpenCL additionally require an explicit implementation of data exchange between GPUs and separate management of each GPU. This includes low-level pointer arithmetics and offset calculations, as well as explicit program execution on each GPU. Neither CUDA nor OpenCL offer specific support for such systems, which makes their programming even more complex.

In this paper, we first briefly describe our SkelCL library [9] for high-level single- and multi-GPU computing. It offer pre-implemented recurring computation patterns (*skeletons*) for simplified GPU programming. In addition, the application developer is freed from memory management, which is done implicitly in SkelCL.

The final publication is available at Springer via
http://dx.doi.org/10.1007/978-3-642-36949-0_41.

As a specific contribution of the paper, we add a new two-dimensional data type and an additional skeleton to SkelCL. The *matrix* data type complements the one-dimensional vector data type for working with two-dimensional data, e.g., matrices or images. The new *MapOverlap* skeleton executes a given function on every element of the input data, using also the values of its neighboring elements. Another novel contribution of this paper is the data distribution mechanism for simplifying two-dimensional data processing on systems with multiple GPUs. Finally, we present an application case study using the matrix data type and the MapOverlap skeleton – Sobel edge detection for 2D images – and report experimental results using SkelCL.

The paper is organized as follows. First we briefly introduce the basics of SkelCL in Section 2 using our previous work [9]. The MapOverlap skeleton is then introduces in Section 2.2 and the matrix data type in Section 2.3. In Section 3 we demonstrate how both are used together to implement an application from the area of image processing: the Sobel edge detection. We report experimental results and we evaluate performance and usability of our approach. Section 4 concludes the paper and compares our approach to related work.

2 SkelCL

We have designed SkelCL as a library for high-level programming of multi-core CPU, GPU, and other processing units. SkelCL is built on top of OpenCL and provides a C++ API that shields the programmer from boilerplate code, e. g., for program initialization or recurring tasks such as explicit data transfers between CPU and GPU. Programming is simplified using algorithmic skeletons – generic building blocks that describe commonly used parallel computation and communication patterns. For flexibility, SkelCL can also be used in combination with low-level OpenCL code. The SkelCL library is available as open-source software and can be downloaded from: `http://skelcl.uni-muenster.de`.

2.1 Algorithmic Skeletons

In standard OpenCL parallelism is specified using, special functions (*kernels*) to be executed in a parallel manner on a device. It is programmer's task to specify in the host program how many instances of a kernel are launched. In addition, kernels usually take pointers to device 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 programmer from these low-level programming tasks, the SkelCL library extends OpenCL by means of high-level programming patterns, called *algorithmic skeletons*. Formally, a skeleton is a higher-order function that executes one or more user-defined functions in a pre-defined parallel manner, while hiding the details of parallelism and communication from the user [8].

Four basic skeletons are provided by SkelCL: *Map*, *Zip*, *Reduce*, and *Scan*. We describe these skeletons semi-formally, with v_{in}, v_{inl}, v_{inr}, and v_{out} denoting vectors of size n, with $0 \leq i < n$:

```
int main (int argc, char const* argv[]) {
  SkelCL::init();                   /* initialize SkelCL */
  Reduce<float> sum (               /* create skeletons */
    "float func(float x,float y) { return x+y; }" );
  Zip<float> mult (
    "float func(float x, float y) { return x*y; }" );
                                    /* create input vectors */
  Vector<float> A(SIZE); fillVector(A);
  Vector<float> B(SIZE); fillVector(B);
                                    /* execute skeletons */
  Vector<float> C = sum( mult( A, B ) );
  cout << "Result: " << C.front(); /* print result */ }
```

Listing 1.1: SkelCL program computing the dot product of two vectors.

- The *Map* skeleton applies a unary function f to each element of an input vector v_{in}, i. e. $v_{out}[i] = f(v_{in}[i])$.
- The *Zip* skeleton operates on two input vectors v_{inl} and v_{inr}, applying a binary operator $\oplus$ to all pairs of elements, i. e. $v_{out}[i] = v_{inl}[i] \oplus v_{inr}[i]$.
- The *Reduce* skeleton computes a scalar value r from a vector using a binary operator $\oplus$, i. e. $r = v[0] \oplus v[1] \oplus \ldots \oplus v[n-1]$.
- The *Scan* skeleton (a. k. a. prefix-sum) yields an output vector with each element obtained by applying a binary operator $\oplus$ to the elements of the input vector up to the current element's index, i. e. $v_{out}[i] = \bigoplus_{j=0}^{i-1} v_{in}[j]$.

Rather than writing low-level kernels, in SkelCL the programmer customizes skeletons by providing user-defined (usually less complex) functions.

Listing 1.1 shows how a dot product of two vectors is implemented in SkelCL using two skeletons: the *Zip* skeleton is customized by usual multiplication, and the *Reduce* skeleton is customized by usual addition. This program comprises 8 lines of code (omitting comments and empty lines). For comparison, an OpenCL-based implementation of a dot product provided in the NVIDIA SDK [1] requires 68 lines of code (kernel function: 9 lines, host program: 59 lines). Besides, additional code would be necessary for a multi-device implementation, including statements for data transfer between multiple devices and for splitting input data and merging output data on the host.

In SkelCL, skeletons can be executed on single- and multi-device systems. In case of a multi-device system, the calculation specified by a skeleton is performed automatically on all devices available to the system. The SkelCL program in Listing 1.1 can thus be executed on a multi-device system without any change.

2.2 The MapOverlap Skeleton

Many applications dealing with two-dimensional data perform calculations for every data element taking neighboring data elements into account. For example, image processing algorithms, like the gaussian blur, calculate a new value for every pixel of an input image using the previous value of the pixel and its surrounding values.

To facilitate the development of such applications, we extend SkelCL by an additional skeleton in combination with a new matrix data type, which is presented in Section 2.3. This skeleton can be used with either vector or matrix data type. We explain the details of the new skeleton for the matrix data type.

- The *MapOverlap* skeleton takes two parameters: a function f and an integer value d. It applies f to each element of an input matrix m_{in} while taking the neighboring elements within the range $[-d, +d]$ in each dimension into account, i. e.

$$m_{out}[i,j] = f\begin{pmatrix} m_{in}[i-d,j-d] \dots m_{in}[i-d,j] \dots m_{in}[i-d,j+d] \\ \vdots & \vdots & \vdots \\ m_{in}[i,j-d] \quad \dots \quad m_{in}[i,j] \quad \dots \quad m_{in}[i,j+d] \\ \vdots & \vdots & \vdots \\ m_{in}[i+d,j-d] \dots m_{in}[i+d,j] \dots m_{in}[i+d,j+d] \end{pmatrix}$$

In the actual source code, the application developer provides the function f which receives a pointer to the element in the middle, $m_{in}[i,j]$. Listing 1.2 shows a simple example of computing the sum of all direct neighboring values using the MapOverlap skeleton. To access the elements of the input matrix m_{in}, function `get` is used, as provided by SkelCL. All indices are specified relative to the middle element $m_{in}[i,j]$, therefore, for accessing this element the function call `get(m_in, 0, 0)` is used.

The application developer must ensure that only elements in the range specified by the second argument d of the MapOverlap skeleton, are accessed. In Listing 1.2, range is specified as $d = 1$, therefore, only direct neighboring elements are accessed. To enforce this property, boundary checks are performed at runtime by the `get` function. In future work, we plan to avoid boundary checks at runtime by statically proving that all memory accesses are in bounds, as it is the case in the shown example.

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 1.2, the first option is chosen and 0.0 is provided as neutral value.

Listing 1.3 shows how the same simple calculation can be performed in standard OpenCL. While the amount of lines of code increases by a factor of 2, the complexity of each single line also increases, as follows. Besides a pointer to the output memory, the width of the matrix has to be provided as parameter. The correct index has to be calculated for every memory access using an offset and the width of the matrix. Therefore, knowledge about how the two-dimensional matrix is stored in one-dimensional memory is required. In addition, manual boundary checks have to be performed to avoid faulty memory accesses.

```
MapOverlap<float(float)> m("float func(float* m_in){
     float sum = 0.0f;
     for (int i = -1; i < 1; ++i)
       for (int j = -1; j < 1; ++i)
         sum += get(m_in, i, j);
     return sum;
   }", 1, SCL_NEUTRAL, 0.0f);
```

Listing 1.2: MapOverlap skeleton computing the sum of all direct neightbors for every element in a matrix

```
__kernel void sum_up(__global float* m_in,
                     __global float* m_out,
                     int width, int height) {
  int i_off = get_global_id(0); int j_off = get_global_id(1);
  float sum = 0.0f;
  for (int i = i_off - 1; i < i_off + 1; ++i)
    for (int j = j_off - 1; j < j_off + 1; ++j) {
      // perform boundary checks
      if ( i < 0 || i > width || j < 0 || j > height )
        continue;
      sum += m_in[ j * width + i ];     }
  m_out[ j_off * width + i_off ] = sum; }
```

Listing 1.3: An OpenCL kernel performing the same calculation as the MapOverlap skeleton shown in Listing 1.2.

SkelCL avoids all these low-level details. Neither additional parameter, nor index calculations or manual boundary checks are necessary. In SkelCL, the application developer only provides the source code implementing the steps required by the algorithm.

2.3 Abstract data types and memory management

The extended SkelCL offers two *containers*: the vector and the matrix. While the vector offers support for one-dimensional data, the new matrix data type handles two-dimensional data.

Vector data type The *vector* class provides an abstraction for a contiguous memory area that is accessible by both, the host and the device. The SkelCL vector replicates the interface of the vector from the Standard Template Library (STL), i. e., it can be used as a replacement of the standard vector. Upon creation of a vector on the host system, memory is allocated on the device accordingly.

The vector class shields the user from low-level memory operations like allocation (on the device) and data transfer between host and device memory, as follows. Data transfers between the memory areas on the host and device are performed implicitly: before any data is accessed on the host, SkelCL ensures that the data on the host is up-to-date. This may lead to implicit data transfers from the device which are performed automatically. All skeletons can accept vectors as their input and output. Before execution, a skeleton's implementation ensures that all input vectors' data is available on all participating devices. This may result in implicit (automatic) data transfers from the host memory to device memory. The data of the output vector is not copied back to the host memory but rather resides in the device memory. Hence, if an output vector is used as the input to another skeleton, no further data transfer is performed. This *lazy copying* in SkelCL minimizes costly data transfers between the host and device.

Matrix data type In addition to the vector as a one-dimensional abstract data structure, we introduce in SkelCL a two-dimensional abstract data type, the *matrix*. Developing applications on 2D data for modern parallel architectures is cumbersome, since efficient memory handling is required for achieving good performance. In case of GPUs, exploiting the memory hierarchy by using the fast but small on-chip memory is mandatory for high performance. Simple and obvious solutions taken directly from a textbook will most certainly result in sub-optimal performance.

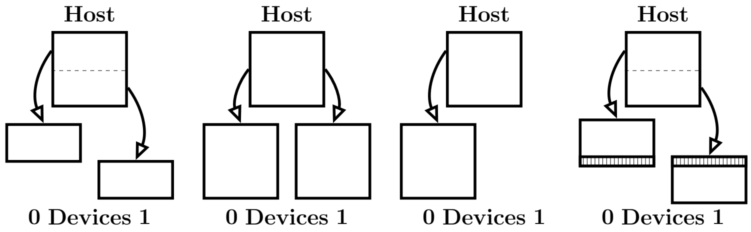
The matrix data type in SkelCL automatically manages all data transfers between the host's memory and the devices' memory. Necessary data transfers are performed implicitly: when a matrix is used by a skeleton, SkelCL copies the matrix's data to the devices, and vice versa. That means that when the data of the matrix is accessed on the host, SkelCL copies the modified data back to the host. Like the vector type, the matrix shields the user from dealing with low-level details like data transfers between different memories. The Map, Zip and MapOverlap skeleton can take matrices as input and output.

2.4 Data distribution on multiple devices

The key feature of SkelCL for multi-device systems is that SkelCL's data types abstract from memory ranges on multiple devices, i. e. the data is accessible by each device. However, each device may access different parts of a container (vector or matrix) or may even not access it at all. For example, when implementing work-sharing on multiple devices, the devices will usually access disjoint parts of input data, such that copying only a part of the container to a device would be more efficient than copying the whole data to each device.

To simplify the specification of partitionings of containers in programs for multi-device systems, SkelCL implements the *distribution* mechanism that describes how a container is distributed among the available devices. It allows the programmer to abstract from managing memory ranges which are shared or spread across multiple devices: the programmer can think of a distributed container as of a self-contained entity.

Four kinds of distribution are currently implemented in SkelCL and offered to the programmer: *block*, *copy*, *single* and *overlap* (see Figure 1). With *block* distribution (Figure 1a), each device stores a contiguous, disjoint part of the container. The *copy* distribution (Figure 1b) copies container's entire data to each available device. In case of *single* distribution (Figure 1c), container's whole data is stored on a single device (the first device by default).



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

Fig. 1: Distributions of a matrix in SkelCL.

Together with the matrix data type, we introduce a new distribution called *overlap* (Figure 1d). The overlap distribution splits the matrix into one chunk for each device, similarly to the block distribution. In addition to the block distribution, the following holds for an overlap-distributed matrix: each chunk consists of a number of continuous rows, and, a parameter – the *overlap size* – specifies the number of rows at the edges of a chunk which are copied to the two neighboring devices. Figure 1d illustrates the overlap distribution: Device 0 receives the top chunk ranging from the top row to the middle, while device 1 receives the second chunk ranging from the middle row to the bottom. The marked regions are the *overlap regions* which are available on both devices.

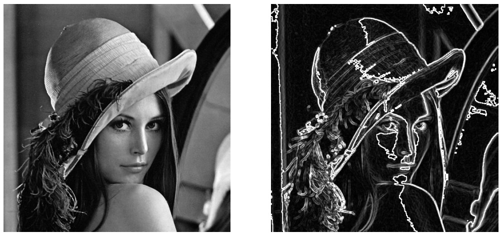
The *overlap* distribution is automatically selected as distribution by the MapOverlap skeleton, to ensure that every device has access to the neighboring elements as needed by the MapOverlap skeleton.

A programmer can set the distribution of containers explicitly, or every skeleton selects a default distribution for its input and output containers otherwise. Container's distribution can be changed at runtime. A change of distribution implies data exchanges between multiple devices and the host, which are performed by SkelCL implicitly and lazily, as described above. Implementing such data transfers in the standard OpenCL is a cumbersome task: data has to be downloaded to the host before it can be uploaded to other devices, including the corresponding length and offset calculations; this results in a lot of low-level code which is completely hidden when using SkelCL.

3 Application study: Sobel edge detection

To evaluate the usability and performance of the MapOverlap skeleton and the matrix data type, we implemented an algorithm commonly used in image processing: The Sobel edge detection is applied to an input image and produces an output image, in which the detected edges in the input image are marked in white and plain areas are shown in black. Figure 2 shows the famous Lena image [7] and the output of Sobel edge detection applied to it.

Listing 1.4 shows the algorithm of the Sobel edge detection in pseudo-code. To keep this version simple, necessary boundary checks are omitted. In this



(a) Original image (b) Image after Sobel edge detection

Fig. 2: The famous Lena image often used as an example in image processing.

```
for (i = 0; i < width; ++i)
  for (j = 0; j < height; ++j)
    h = -1*img[i-1][j-1] +1*img[i+1][j-1]
        -2*img[i-1][j  ] +2*img[i+1][j  ]
        -1*img[i-1][j+1] +1*img[i+1][j+1];
    v = ...;
    out_img[i][j] = sqrt(h*h + v*v);
```

Listing 1.4: Sequential implementation of the Sobel edge detection.

sequential version, for computing one output value `out_img[i][j]` the input value `img[i][j]` and the direct neighboring elements are needed. Therefore, the MapOverlap skeleton is a perfect fit for implementing the Sobel edge detection.

Listing 1.5 shows the SkelCL implementation using the MapOverlap skeleton and the matrix type. The implementation is straightforward and very similar to the sequential version in Listing 1.4. The only notable difference is that for accessing elements the `get` function is used instead of the square bracket notation.

Listing 1.6 shows a part of the standard OpenCL implementation for Sobel edge detection. The actual computation is performed inside the `computeSobel` function, which is omitted in the listing, since it is quite similar to the sequential version describing the actual Sobel edge detection algorithm. The listing shows that extra low-level code is necessary to deal with technical details, like boundary

```
// skeleton customized with Sobel edge detection algorithm
MapOverlap<char(char)> m( "char func(const char* img) {
  short h = -1*get(img,-1,-1) +1*get(img,+1,-1)
            -2*get(img,-1, 0) +2*get(img,+1, 0)
            -1*get(img,-1,+1) +1*get(img,+1,+1);
  short v = ...;
  return sqrt(h*h + v*v); }", 1, SCL_NEUTRAL, 0);
Matrix<char> out_img = m(img); // execution of the skeleton
```

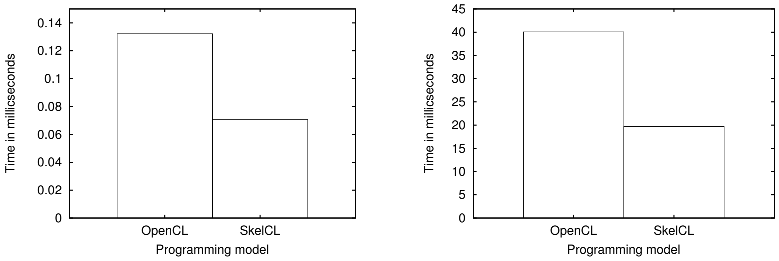
Listing 1.5: SkelCL implementation of the Sobel edge detection.

```
__kernel void sobel_kernel( __global const char* img,
                            __global       char* out_img,
                                     int w, int h ) {
 size_t i = get_global_id(0);   size_t j = get_global_id(1);

 if(i < w && j < h) {
  // perform boundary checks
  char ul = (j-1 > 0 && i-1 > 0) ? img[((j-1)*w)+(i-1)] : 0;
  char um = (j-1 > 0           ) ? img[((j-1)*w)+(i+0)] : 0;
  char ur = (j-1 > 0 && i+1 < w) ? img[((j-1)*w)+(i+1)] : 0;
  // ... 5 more
  char lr = (j+1 < h && i+1 < w) ? img[((j+1)*w)+(i+1)] : 0;

  out_img[j * w + i] = computeSobel(ul, um, ur, ..., lr); } }
```

Listing 1.6: Additional boundary checks and index calculations, necessary in the standard OpenCL implementation.



(a) Example image: Lena

(b) Example image: World

Fig. 3: Performance results (runtimes)

checks and index calculations. These extra lines are arguably complex and error-prone because they handle low-level details, rather than the application logic.

We performed runtime experiments using a NVIDIA Tesla T10 GPU with 480 processing elements and 4 GByte memory. Figure 3 shows the runtime of the OpenCL version in Listing 1.6 vs. the SkelCL version with the MapOverlap skeleton in Listing 1.5. Only the kernel runtimes are shown, as the data transfer times are equal for both versions. Measurements were taken using the OpenCL profiling API. Besides the Lena image [7] with a size of 512 × 512 pixel, we also used a bigger image by NASA showing the world [6] with a resolution of 15296 × 7648 pixel. The mean values of 6 runs are shown in Figure 3. The Lena image is shown on the left, the NASA world image on the right.

The SkelCL version clearly outperforms the OpenCL implementation. This is due to the fact, that the MapOverlap skeleton uses the fast local memory inside its implementation which is hidden from the application developer.

In addition to the performance advantage, the SkelCL program is also significantly simpler than the cumbersome OpenCL implementation. The OpenCL implementation requires 19 lines in total while the SkelCL program only comprises 4. No index calculations or boundary checks are necessary in the SkelCL version whereas they are crucial for a correct implementation in OpenCL.

The OpenCL program in Listing 1.6 is not an optimized, but rather a straightforward version most programmers who are not OpenCL experts would write. Since SkelCL targets such programmers rather than GPU experts, we take this version for comparison with the SkelCL version. An optimized OpenCL version, e.g., using local memory would probably perform better but would definitely require additional low-level code.

4 Conclusion and Related Work

In this paper we showed how the SkelCL library can be extended for developing applications on two-dimensional data. We used an image processing application as a case study. Using SkelCL, such applications can easily benefit from the performance of GPUs. Application developers do not have to be GPU computing experts to achieve good performance, since SkelCL's skeletons exploit the GPU memory hierarchy transparently for the user. The two-dimensional data type significantly simplifies memory management. The SkelCL library is available as open-source software at `http://skelcl.uni-muenster.de`.

Similar approaches have been proposed recently to simplify GPU programming. *SkePU* [3] is a high-level framework for multi-core CPUs and multi-GPU systems offering skeletons similar to SkelCL. A macros-based mechanism allows for using either OpenMP, CUDA or OpenCL as back-end to execute the skeletons, but also restricts the programmer to the back-ends' smallest common set of functions.A skeleton similar to our new proposed MapOverlap skeleton is available in SkePU, but restricted to one-dimensional data. Recently a two-dimensional data type has been added to SkePU, but it is not yet fully integrated in its current version.

In future work we will extend the set of skeletons offered by SkelCL and we will specifically target heterogeneous systems with multiple GPUs.

References

1. NVIDIA CUDA SDK code samples, Feburary 2010. Version 3.0.
2. *NVIDIA CUDA API Reference Manual*, 2012. Version 4.1.
3. J. Enmyren and C. Kessler. SkePU: A multi-backend skeleton programming library for multi-gpu systems. In *Proc. of HLPP-2010*, 2010.
4. D. B. Kirk and Hwu W. W. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufman, 2010.
5. A. Munshi. *The OpenCL Specification*, 2011. Version 1.2.
6. NASA. Blue marble: Land surface, shallow water, and shaded topography. http://visibleearth.nasa.gov/view.php?id=57752.
7. University of Southern California SIPI Image Database. Girl (lena, or lenna). http://sipi.usc.edu/database/database.php?volume=misc.

8. F. A. Rabhi and S. Gorlatch, editors. *Patterns and skeletons for parallel and distributed computing*. Springer-Verlag, 2003.
9. M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL – A portable skeleton library for high-level GPU programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, pages 1176–1182, 2011.