SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems

Michel Steuwer and Sergei Gorlatch

University of Muenster, Germany
michel.steuwer@wwu.de and gorlatch@wwu.de

Abstract. Application development for modern high-performance systems with Graphics Processing Units (GPUs) currently relies on low-level programming approaches like CUDA and OpenCL, which leads to complex, lengthy and error-prone programs.
In this paper, we present SkelCL – a high-level programming approach for systems with multiple GPUs and its implementation as a library on top of OpenCL. SkelCL provides three main enhancements to the OpenCL standard: 1) computations are conveniently expressed using parallel algorithmic patterns (*skeletons*); 2) memory management is simplified using parallel *container data types* (vectors and matrices); 3) an automatic *data (re)distribution mechanism* allows for implicit data movements between GPUs and ensures scalability when using multiple GPUs. We demonstrate how SkelCL is used to implement parallel applications on one- and two-dimensional data. We report experimental results to evaluate our approach in terms of programming effort and performance.

1 Introduction

Modern high-performance computer systems become increasingly heterogeneous as they comprise in addition to multi-core processors, also *Graphics Processing Units* (GPUs), Cell processors, FPGA, and other accelerating devices, usually called *accelerators*. The state-of-the-art application programming for systems with GPUs is cumbersome and error-prone, because GPUs are programmed using explicit, low-level programming approaches like CUDA [11] or OpenCL [13]. 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.

In this paper, we describe the *SkelCL* (Skeleton Computing Language) – our high-level programming approach for parallel systems with multiple GPUs. The SkelCL programming model is based on the OpenCL standard and enhances it with three high-level mechanisms:

The final publication is available at Springer via
http://dx.doi.org/10.1007/978-3-642-39958-9_24.

1) *parallel container data types*: collections of data (in particular, vectors and matrices) that are managed automatically on all GPUs in the system;
2) *data (re)distributions*: an automatic mechanism for specifying in the application program suitable data distributions and re-distributions among the GPUs of the target system:
3) *parallel skeletons*: pre-implemented high-level patterns of parallel computation and communication which can be customized to express application-specific parallelism, and combined to a large high-level code.

The structure of the paper is as follows. In Section 2 we formulate the requirements to a high-level programming approach for GPU systems, following from the analysis of compute-intensive applications. Section 3 describes in detail our SkelCL approach. In Section 4 we report experimental evaluation of our approach regarding both programming effort and performance. We compare to related work and conclude in Section in Section 5.

2 Requirements to a High-Level Programming Model

To simplify programming for a system with multiple GPUs, the following high-level abstraction are desirable:

Parallel container data types Compute-intensive applications typically operate on a (possibly big) set of data items. Managing memory hierarchy of multi-GPU systems explicitly is complex and error-prone because low-level details, like offset calculations, have to be programmed manually. A high-level programming model should be able to make collections of data automatically accessible to all GPUs in the target system and it should provide an easy-to-use interface for the application developer.

Distribution and redistribution mechanisms To achieve scalability of applications on systems comprising multiple GPUs, it is crucial to decide how the application's data are distributed across all available GPUs. Applications often require different distributions for their computational steps. Distributing and re-distributing data between GPUs in OpenCL is cumbersome because data transfers have to be managed manually and performed via the CPU. Therefore, it is important for a high-level programming model to allow both for describing the data distribution and for changing the distribution at runtime, such that the system takes care of the necessary data movements.

Recurring patterns of parallelism While the concrete operations performed in an application are (of course) application-specific, the general structure of parallelization often follows some common parallel patterns that are reused in different applications. For example, operations can be performed for every entry of an input vector, which is a well-known pattern of data-parallel programming, or two vectors are combined element-wise into an output vector, which is again a common pattern of parallelism. It would be, therefore, desirable to express the high-level structure of an application using pre-defined common patterns, rather than describing the parallelism explicitly in much detail.

3 SkelCL: Programming Model and Library

We develop our SkelCL [14] programming model as an extension of the standard OpenCL programming model [13], which is an emerging de-facto standard for programming heterogenous systems with various accelerators. SkelCL adds to OpenCL three features that we identified as desirable in Section 2. SkelCL inherits all properties of OpenCL, including its portability across different heterogeneous parallel systems. SkelCL is designed to be fully compatible with OpenCL: arbitrary parts of a SkelCL code can be written or rewritten in OpenCL, without influencing program's correctness. While the main OpenCL program is executed sequentially on the CPU – called the *host* – computations are offloaded to parallel processors – called *devices*. In this paper, we focus on systems comprising multiple GPUs, therefore, we use the terms CPU and GPU, rather than more general OpenCL terms host and device.

3.1 Parallel Container Data Types

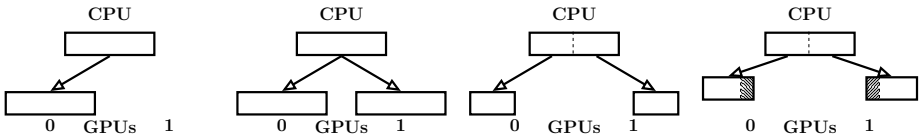
SkelCL offers the application developer two container classes – vector and matrix – which are transparently accessible by both, host and devices, i. e. the CPU and the GPUs. The *vector* abstracts a one-dimensional contiguous memory area while the *matrix* provides an interface to a two-dimensional memory area. When a container is created on the CPU, memory is allocated on the GPUs automatically; when a container on the CPU is deleted, the memory allocated on the GPUs is freed automatically. In a SkelCL program, a vector object can be created and filled with data as in the following example:

```
Vector<int> vec(size);
for (int i = 0; i < vec.size(); ++i){ vec[i] = i; }
```

The main advantage of the container data types in SkelCL as compared with OpenCL is that the necessary data transfers between the CPU and GPUs are performed implicitly. Before performing a computation on container types, the SkelCL system ensures that all input containers' data is available on all participating GPUs. This may result in implicit (automatic) data transfers from the CPU 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 by performing necessary data transfers implicitly and automatically. Thus, the container classes shield the programmer from low-level operations like memory allocation (on GPU) and data transfers between CPU and GPU.

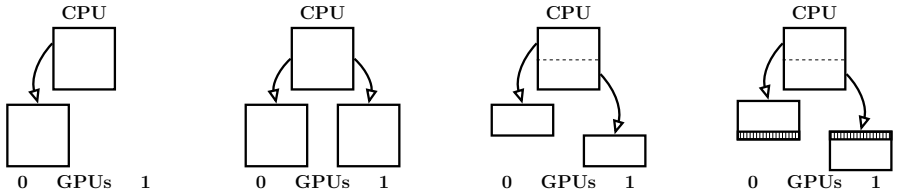
3.2 Data Distribution on Multiple GPUs

In applications working on container data types (vectors, matrices, etc.) GPU's often access disjoint parts of input data, such that copying only a part of the container to a GPU would be more efficient than copying the whole data to each GPU. To simplify the specification of partitionings of containers in programs



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

Fig. 1. Distributions of a vector in SkelCL.



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

Fig. 2. Distributions of a matrix in SkelCL.

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

Four kinds of distribution are currently available in SkelCL: *single*, *copy*, *block*, and *overlap* (see Fig. 1 for distributing a vector on a system with two GPUs). If distribution is set to *single* (Fig. 1a), than vector's whole data is stored on a single GPU (the first GPU if not specified otherwise). The *copy* distribution (Fig. 1b) copies vector's entire data to each available GPU. With the *block* distribution (Fig. 1c), each GPU stores a contiguous, disjoint chunk of the vector. The *overlap* distribution (Fig. 1d) stores on each GPU the chunk like in the block distribution, together with one or several border elements of the neighboring chunk.

The same four distributions are provided also for the matrix data type (Figure 2). In particular the overlap distribution splits the matrix into one chunk for each GPU; in addition, each chunk contains a number of continuous rows from the neighboring chunks. A parameter – the *overlap size* – specifies the number of rows at the borders of a chunk which are copied to the two neighboring GPUs. Figure 2d illustrates the overlap distribution: GPU 0 receives the top chunk ranging from the top row to the middle, while GPU 1 receives the second chunk ranging from the middle row to the bottom. The marked parts are called *overlap region* they are the same on both GPUs.

The application developer can set the distribution of containers (vectors and matrices) explicitly, otherwise every skeleton selects a default distribution for its input and output containers. Container's distribution 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 the standard OpenCL is a cumbersome task: data has to be downloaded to the CPU before it is uploaded to the GPUs, including the corresponding length and offset calculations; this results in a lot of low-level code which becomes completely hidden when using SkelCL.

3.3 Basic Patterns of Parallelism (Skeletons)

In original OpenCL, computations are expressed as *kernels* 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* [15]. 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 [15].

The current version of SkelCL provides six skeletons: *map*, *zip*, *reduce*, *scan*, *mapOverlap* and *allpairs*. We define first the four basic skeletons. We do this semi-formally, with c, cl and cr denoting vectors with elements c_i, cl_i and cr_i where $0 < i \leq n$:

- The map skeleton applies a unary customizing function f to each element of an input vector c, i. e.

  $$map\ f\ [c_1, c_2, \ldots, c_n] = [f(c_1), f(c_2), \ldots, f(c_n)]$$

  In a SkelCL program, a map skeleton is created as an object for a unary function f, e. g. negation, like this:

  ```
  Map<float(float)> neg("float func(float x){ return -x;}");
  ```

  This map object can then be called as a function with a vector as argument:

  ```
  resultVector = neg( inputVector );
  ```

- The zip skeleton operates on two vectors cl and cr, applying a binary customizing operator $\oplus$ pairwise:

  $$zip\ (\oplus)\ [cl_1, cl_2, \ldots, cl_n]\ [cr_1, cr_2, \ldots, cr_n] = [cl_1 \oplus cr_1, cl_2 \oplus cr_2, \ldots, cl_n \oplus cr_n]$$

  In SkelCL, a zip skeleton object for adding two vectors is created like as:

```
Zip<float(float, float)> add("float func(float x,float y){return x+y;}");
```

and can then be called as a function with a pair of vectors as arguments:

```
resultVector = add( leftVector, rightVector );
```

– The reduce skeleton computes a scalar value from a vector using a binary associative operator $\oplus$, i. e.

$$red\ (\oplus)\ [v_1, v_2, \ldots, v_n] = v_1 \oplus v_2 \oplus \cdots \oplus v_n$$

For example, to sum up all elements of a vector, the reduce skeleton is created with addition as customizing function, and called as follows:

```
Reduce<float(float)> sumUp("float func(float x,float y){ return x+y;}");

result = sumUp( inputVector );
```

– The scan skeleton (a. k. a. prefix-sum) yields an output vector with each element obtained by applying a binary associative operator $\oplus$ to the elements of the input vector up to the current element's index, i. e.

$$scan\ (\oplus)\ [v_1, v_2, \ldots, v_n] = [v_1, v_1 \oplus v_2, \ldots, v_1 \oplus v_2 \oplus \cdots \oplus v_n]$$

The prefix sums customized with addition is specified and called in SkelCL as follows:

```
Scan<float(float)> prefixSum("float func(float x,float y){return x+y;}");

result = prefixSum( inputVector );
```

In SkelCL, rather than writing low-level kernels, the application developer customizes suitable skeletons by providing application-specific functions which are often much simpler than kernels as they specify an operation on basic data items rather than containers. Skeletons can be executed on both single- and multi-GPU systems. In case of a multi-GPU system, the calculation specified by a skeleton is performed automatically on all GPUs available in the system.

```
int main (int argc, char const* argv[]) {
  SkelCL::init(); /* initialize SkelCL */
/* create skeletons */
  SkelCL::Reduce<float> sum ("float sum (float x,float y)\
      {return x+y;}");
  SkelCL::Zip<float>    mult("float mult(float x,float y)\
      {return x*y;}");
/* create input vectors */
  SkelCL::Vector<float> A(SIZE);
  SkelCL::Vector<float> B(SIZE);
/* fill vectors with data */
  fillVector(A.begin(), A.end());
  fillVector(B.begin(), B.end());
/* execute skeleton */
  SkelCL::Scalar<float> C = sum( mult( A, B ) );
/* fetch result */
  float c = C.getValue();
```

```
}
```

Listing 1.1. SkelCL program computing the dot product of two vectors. Arrays `a_ptr` and `b_ptr` initialize the vectors.

Listing 1.1 shows how a dot product of two vectors is implemented in SkelCL using two of the basic skeletons. Here, the `Zip` skeleton is customized by multiplication, and the `Reduce` skeleton is customized by usual addition. For comparison, an OpenCL-based implementation of the dot product computation provided by NVIDIA requires approximately 68 lines of code (kernel function: 9 lines, host program: 59 lines) [2].

3.4 The MapOverlap Skeleton

Many numerical and image processing applications dealing with two-dimensional data perform calculations for a particular data element (e. g., a pixel) taking neighboring data elements into account. To facilitate the development of such applications, we define in SkelCL a skeleton that can be used with both vector and matrix data type; we explain the details for the matrix data type.

– The *MapOverlap* skeleton takes two parameters: a unary 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.

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

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 provided by SkelCL. All indices are specified relative to the middle element $m_{in}[i,j]$; therefore, for accessing this element the function

```
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.

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: 1) Besides a pointer to the output memory, the width of the matrix has to be provided as parameter; 2) the correct index has to be calculated for every memory access using an offset and the width of the matrix, i. e. knowledge about how the two-dimensional matrix is stored in one-dimensional memory is required. 3) In addition, manual boundary checks have to be performed to avoid faulty memory accesses.

SkelCL avoids all these low-level details. Neither additional parameter, nor index calculations or manual boundary checks are necessary.

3.5 The Allpairs Skeleton

All-pairs computations occur in a variety of applications, ranging from pairwise Manhattan distance computations used in bioinformatics [12] to N-Body simulations used in physics [3]. All these applications follow a common computational

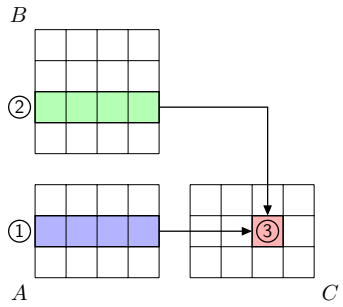


Fig. 3. The allpairs computation: Element $c_{2,3}$ (③) is computed by combining the second row of A (①) with the third row of B (②) using the binary operator $\oplus$

scheme: for two sets of entities, the same computation is performed independently for all pairs of entities from the first set combined with entities from the second set. An entity is usually described by a d-dimensional vector.

We define the all-pairs computation scheme for two sets of n and m entities, each entity represented by a d-dimensional vector. We represent the sets as an $n \times d$ matrix A and an $m \times d$ matrix B. The all-pairs computation yields an output matrix C of size $n \times m$ as follows: $C_{i,j} = A_i \oplus B_j$, where A_i and B_j are rows of A and B, correspondingly: $A_i = [A_{i,1}, \cdots, A_{i,d}]$, $B_j = [B_{j,1}, \cdots, B_{j,d}]$, and $\oplus$ is a binary function applied to every pair of rows from A and B.

Figure 3 illustrates this definition: the element marked as ③ of matrix C is computed by combining the second row of A marked as ① with the third row of B marked as ② using the binary operator $\oplus$.

For formally defining the all-pairs skeleton, let d, m and n be positive numbers. Let A be a $n \times d$ matrix, B be a $m \times d$ matrix and C be a $n \times m$ matrix with their entries $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$ respectively. Let $\oplus$ be a binary function on vectors. The algorithmic skeleton *allpairs* is defined as follows:

$$allpairs(\oplus)\left(\begin{bmatrix} a_{1,1} & \cdots & a_{1,d} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,d} \end{bmatrix}, \begin{bmatrix} b_{1,1} & \cdots & b_{1,d} \\ \vdots & & \vdots \\ b_{m,1} & \cdots & b_{m,d} \end{bmatrix}\right) \stackrel{\text{def}}{=} \begin{bmatrix} c_{1,1} & \cdots & c_{1,m} \\ \vdots & & \vdots \\ c_{n,1} & \cdots & c_{n,m} \end{bmatrix}$$

with entries $c_{i,j}$ of the computed $n \times m$ matrix C defined as:

$$c_{i,j} = [a_{i,1} \ \cdots \ a_{i,d}] \oplus [b_{j,1} \ \cdots \ b_{j,d}]$$

To illustrate the definition, we show how matrix multiplication can be expressed using the allpairs skeleton.

Example 1: The matrix multiplication is a basic linear algebra operation, which is a building block of many scientific applications. A $n \times d$ matrix A is multiplied by a $d \times m$ matrix B, producing a $n \times m$ matrix $C = A \times B$ whose element $C_{i,j}$

is computed as the dot product of the ith row of A with jth column of B. The dot product of two vectors a and b of length d is computed as:

$$dotProduct(a, b) = \sum_{k=1}^{d} a_k \cdot b_k \tag{1}$$

The matrix multiplication can be expressed using the allpairs skeleton as:

$$A \times B = allpairs(dotProduct)\left(A, B^T\right) \tag{2}$$

where B^T is the transpose of matrix B.

4 Application Studies and Experiments

We consider two application case studies using the SkelCL library: 1) the calculation of a Mandelbrot fractal, and 2) the Sobel edge detection. Both SkelCL implementations are compared to similar implementations in CUDA and OpenCL regarding their programming effort and runtime performance.

For our runtime experiments we use a PC with a quad-core CPU (Intel Xeon E5520, 2.26 GHz) and 12 GB of memory. The system is connected to a Tesla S1070 computing system equipped with 4 Tesla GPUs. Its dedicated 16 GB of memory (4 GB per GPU) is accessed with up to 408 GB/s (102 GB/s per GPU). Each GPU comprises 240 streaming processor cores running at 1.44 GHz.

4.1 Application Study: Mandelbrot Set

The Mandelbrot set calculation [10] is a time-consuming task which is often used as a benchmark. Computing a Mandelbrot fractal is easily parallelizable, as all pixels can be computed simultaneously. As the criteria for programming effort we use the number of Lines of Code (LoC), the results are in Fig. 4.

We created three similar parallel implementations for computing a Mandelbrot fractal using CUDA, OpenCL, and SkeCL.

CUDA and SkelCL require a single line of code for initialization in the host code, whereas OpenCL requires a lengthy creation and initialization of different data structures which take about 20 LoC. The host CPU code differs significantly between all implementations. In CUDA, the kernel is called like an ordinary function. A proprietary syntax is used to specify the size of work-groups executing the kernel. In OpenCL, several API functions are called to load and build the kernel, pass arguments to it and launch it using a specified work-group size. In SkelCL, the kernel is passed to a newly created instance of the `Map` skeleton. A `Vector` of complex numbers, each of which represents a pixel of the Mandelbrot fractal, is passed to the `Map` skeleton upon execution. Specifying the work-group size is mandatory in CUDA and OpenCL, whereas this is optional in SkelCL.

The OpenCL-based implementation has in total 118 lines of code (kernel: 28 lines, host program: 90 lines) and is thus more than twice as long as the CUDA

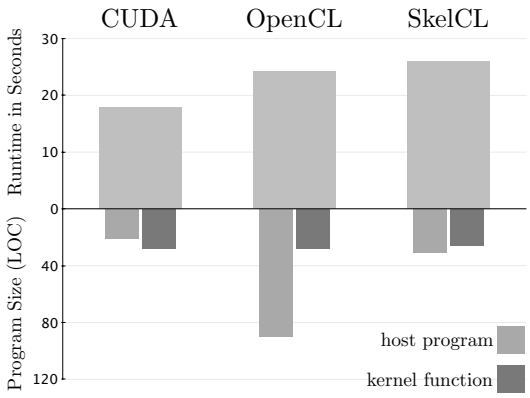


Fig. 4. Runtime and program size of the Mandelbrot application.

and SkelCL versions with 49 lines (28, 21) and 57 lines (26, 31), respectively (see Figure 4).

We tested our implementations on a single GPU of our test system to compute a Mandelbrot fractal of size 4096×3072 pixels. In CUDA and OpenCL, work-groups of 16×16 are used; SkelCL uses its default work-group size of 256.

As compared to the runtime of the SkelCL-based implementation (26 sec), the implementation based on OpenCL (25 sec) and CUDA (18 sec) are faster by 4% or 31%, respectively. Since SkelCL is built on top of OpenCL, the performance difference of SkelCL and OpenCL can be regarded as the overhead introduced by SkelCL. Previous work [9] reported that CUDA was usually faster than OpenCL, which explains the higher performance of the implementation based on CUDA. The Mandelbrot application demonstrates that SkelCL introduces a tolerable overhead of less than 5% as compared to OpenCL.

4.2 Application Study: Sobel Edge Detection

To evaluate the usability and performance of the MapOverlap skeleton on the matrix data type, we implemented the Sobel edge detection that produces an output image in which the detected edges in the input image are marked in white and plain areas are shown in black.

```
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.

Listing 1.4 shows the algorithm of the Sobel edge detection in pseudo-code, with omitted boundary checks for brevity. In this sequential version, for computing an 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.

```
// 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.

Listing 1.5 shows the SkelCL implementation using the MapOverlap skeleton and the matrix data 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.

```
__kernel void sobel_kernel( __global const uchar* img,
                            __global       uchar* out_img)
 uint i = get_global_id(0);   uint j = get_global_id(1);
 uint w = get_global_size(0); uint h = get_global_size(1);
 // perform boundary checks
 if(i >= 1 && i < (w-1) && j >= 1 && j < (h-1)) {
  char ul = img[((j-1)*w)+(i-1)];
  char um = img[((j-1)*w)+(i+0)];
  char ur = img[((j-1)*w)+(i+1)];
  // ... 5 more
  char lr = img[((j+1)*w)+(i+1)];

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

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

Listing 1.6 shows a part of the OpenCL implementation for Sobel edge detection provided by AMD as an example for their software development kit [1]. The actual computation is performed inside the `computeSobel` function, which is omitted in the listing, since it is quite similar to the sequential version in Listing 1.4. The listing shows that extra low-level code is necessary to deal with

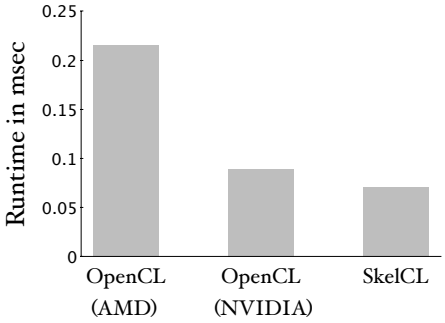


Fig. 5. Performance results for Sobel edge detection

technical details, like boundary checks and index calculations, which are arguably complex and error-prone.

We performed runtime experiments using one NVIDIA Tesla GPU with 480 processing elements and 4 GByte memory. Figure 5 shows the runtime of two OpenCL versions (from AMD and NVIDIA SDK) vs. the SkelCL version with the MapOverlap skeleton presented in Listing 1.5. Only the kernel runtimes are shown, as the data transfer times are equal for all versions. Measurements were taken using the OpenCL profiling API. We used the popular Lena image [18] with a size of 512×512 pixel and took the mean values of six runs. The AMD version is clearly slower then the two other implementations, because it does not use the fast local memory which the NVIDIA implementation and the MapOverlap skeleton of SkelCL do. SkelCL totally hides the memory management details inside its implementation from the application developer. The NVIDIA and SkelCL implementations perform similar. In this particular example, SkelCL even slightly outperforms the implementation by NVIDIA.

In addition to the performance advantage over the AMD and NVIDIA versions, the SkelCL program is also significantly simpler than the cumbersome OpenCL implementation. The SkelCL program only comprises the few lines of code shown in Listing 1.5. The AMD implementations requires 37 lines of code for its kernel implementation and the NVIDIA implementation requires even 208 lines of code. Both versions require additional lines of code for the host program which manages the execution of the OpenCL kernel. No index calculations or boundary checks are necessary in the SkelCL version whereas they are crucial for a correct implementation in OpenCL.

5 Conclusion and Related Work

This paper presents the SkelCL high-level programming model for multi-GPU systems and its implementation as a library. The SkelCL programming model significantly raises the level of abstraction: it combines parallel patterns to express computations, parallel container data types for simplified memory management

and a data (re)distribution mechanism to improve scalability in systems with multiple GPUs. The SkelCL library is available as open source software from `http://skelcl.uni-muenster.de`.

There are a number of other projects aiming at high-level GPU programming.

SkePU [17] 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. Both approaches differ significantly in the way how functions are passed to skeletons. While functions are defined as plain strings in SkelCL, SkePU uses a macro language, which brings some serious drawbacks. For example, it is not possible to call mathematical functions like sin or cos inside a function generated by a SkelPU macro, because these functions are either named differently in all of their three target programming models (CUDA, OpenCL, OpenMP) or might even be missing entirely. The same holds for functions and keywords related to performance tuning, e. g., the use of local memory. SkelCL does not suffer from these drawbacks because it relies on OpenCL and thus can be executed on a variety of GPUs and other accelerators.

CUDPP [4] provides data-parallel algorithm primitives similar to skeletons. These primitives can be configured using only a predefined set of operations, whereas skeletons in SkelCL are true higher-order functions which accept any user-defined function. CUDPP does not simplify data management, because data still has to be exchanged between CPU and GPU explicitly. There is also no support for multi-GPU applications.

Thrust [16] 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.

Unlike SkelCL, *OpenACC* [6], *PGI Acccelerator* [5], and *HMPP* [8] are compiler-based approaches to GPU programming, similar to the popular OpenMP [7]. The programmer uses compiler directives to mark regions of code to be executed on a GPU. A compiler generates executable code for the GPU, based on the used directives. Although source code for low-level details like memory allocation or data exchange is generated by the compiler, these operations still have to be specified explicitly by the programmer using suitable compiler directives. We consider these approaches low-level, as they do not perform data transfer automatically to shield the programmer from low-level details and parallelism is still expressed explicitly.

Acknowledgments

This work is partially supported by the OFERTIE (FP7) and MONICA projects. We would like to thank NVIDIA for their generous hardware donation.

References

1. AMD APP SDK code samples, February 2013, version 2.7. [Online]. Available: `http://developer.amd.com/`
2. NVIDIA CUDA SDK code samples, February 2013, version 5.0. [Online]. Available: `http://developer.nvidia.com/`
3. Arora N, Shringarpure A, Vuduc RW (2009) *Direct N-body Kernels for Multicore Platforms*. In: Proceedings of the 2009 International Conference on Parallel Processing, IEEE Computer Society, Washington, DC, USA, ICPP '09, pp 379–387.
4. S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Graphics Hardware 2007*.
5. T. P. Group, PGI Accelerator Programming Model for Fortran & C, 2010.
6. *OpenACC Application Program Interface*, 2011, version 1.0. [Online]. Available: `http://http://www.openacc.org/`
7. *OpenMP Application Program Interface*, OpenMP Architecture Review Board, 2008, version 3.0. [Online]. Available: `http://www.openmp.org/mp-documents/spec30.pdf`
8. S. Bihan, G. Moulard, R. Dolbeau *et al.*, "Directive-based heterogeneous programming a GPU-accelerated RTM use case," in *Proceedings of the 7th International Conference on Computing, Communications and Control Technologies*, 2009.
9. J. Kong, M. Dimitrov, Y. Yang *et al.*, "Accelerating MATLAB image processing toolbox functions on GPUs," in *GPGPU '10: Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010.
10. B. B. Mandelbrot, "Fractal aspects of the iteration of $z \mapsto \lambda z(1-z)$ for complex λ and z," *Annals of the New York Academy of Sciences*, vol. 357, pp. 249–259, 1980.
11. NVIDIA CUDA API Reference Manual, version 5.0 (February 2013).
12. Chang D, Desoky A, Ouyang M, Rouchka E (2009) Compute Pairwise Manhattan Distance and Pearson Correlation Coefficient of Data Points with GPU. In: Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD '09, pp 501 –506.
13. A. Munshi, The OpenCL Specification, version 1.2.
14. M. Steuwer, P. Kegel, S. Gorlatch, SkelCL – A Portable Skeleton Library for High-Level GPU Programming, in: 2011 IEEE 25th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2011, pp. 1171–1177.
15. S. Gorlatch, M. Cole, Parallel skeletons, in: Encyclopedia of Parallel Computing, 2011, pp. 1417–1422.
16. J. Hoberock, N. Bell, Thrust: A Parallel Template Library (2009).
17. J. Enmyren, 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, 2010, pp. 5–14.
18. University of Southern California SIPI Image Database. Girl (lena, or lenna). http://sipi.usc.edu/database/database.php?volume=misc.