Noname manuscript No.
(will be inserted by the editor)

SkelCL: A High-Level Extension of OpenCL for Multi-GPU Systems

Michel Steuwer · Sergei Gorlatch

Received: date / Accepted: date

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. We present SkelCL – a high-level programming approach for systems with multiple GPUs and its implementation as a library on top of OpenCL. SkelCL makes three main enhancements to the OpenCL standard: 1) memory management is simplified using parallel *container data types* (vectors and matrices); 2) an automatic *data (re)distribution mechanism* allows for implicit data movements between GPUs and ensures scalability when using multiple GPUs; 3) computations are conveniently expressed using parallel algorithmic patterns (*skeletons*). We demonstrate how SkelCL is used to implement parallel applications and we report experimental evaluation of our approach in terms of programming effort and performance.

Keywords Parallel programming · GPU programming · OpenCL · Algorithmic skeletons · SkelCL · Many-cores

1 Introduction

Modern high-performance computer systems become increasingly heterogeneous as they comprise in addition to multi-core processors (CPUs), also *Graphics Processing Units* (GPUs) and other so-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 [8]. These approaches require the programmer to explicitly manage GPU's memory (including memory

University of Muenster, Germany
E-mail: michel.steuwer@wwu.de and E-mail: gorlatch@uni-muenster.de

The final publication is available at Springer via
http://dx.doi.org/10.1007/s11227-014-1213-y.

(de)allocations, and data transfers to/from the system's main memory), and explicitly specify parallelism in the computation. This leads to lengthy, complicated and, thus, error-prone code. For multi-GPU systems, programming with CUDA and OpenCL is even more complex as an explicit implementation of data exchange between the GPUs and disjoint management of each GPU's memory are required, including low-level pointer and offset calculations.

SkelCL (Skeleton Computing Language) is our high-level programming approach for parallel systems with multiple GPUs which is based on the OpenCL standard and enhances it with three high-level mechanisms:

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*: a mechanism for specifying in the application program suitable data distributions and re-distributions among the GPUs, which are then automatically enforced at runtime;
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.

2 SkelCL: Programming Model and Library

We develop SkelCL [14] as an extension of the standard OpenCL programming model [8] which covers multi-core CPUs, GPUs, and other accelerators. SkelCL is fully compatible with OpenCL: arbitrary parts of a SkelCL code can be written in OpenCL, without influencing program's correctness. The main program is executed sequentially on the CPU – called the *host* – and computations are offloaded to GPUs – called *devices*.

2.1 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 and which abstract one- and two-dimensional contiguous memory areas, correspondingly. When a container is created on the CPU, memory is allocated on the GPUs automatically; when a container on the CPU is deleted, its memory on the GPUs is freed automatically. In a SkelCL program, a vector object can be created and filled 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 computing, 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 accessing data on the CPU, 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 memory operations.

2.2 Data Distribution on Multiple GPUs

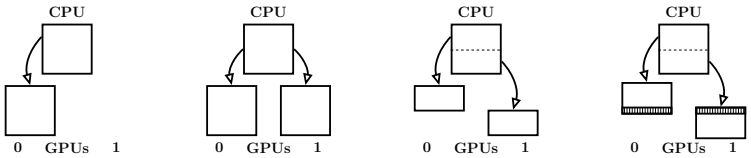
In applications working on containers, GPUs often access disjoint parts of input data. To simplify the specification of containers' partitionings across multiple GPUs, SkelCL implements the *distribution* mechanism that describes how a container is distributed among the available GPUs. It allows the programmer to abstract from managing memory ranges across multiple GPUs: the programmer views a distributed container as a self-contained entity.

Four kinds of distribution are currently available in SkelCL: *single*, *copy*, *block*, and *overlap*; see Fig. 1 for illustration. With *single* distribution, matrix's whole data is stored on a single GPU (the first GPU if not specified otherwise); *copy* copies matrix's entire data to each available GPU; *block* partitions matrix in disjoint chunks across GPUsdistribution, each GPU stores a contiguous, disjoint chunk of the matrix; *overlap* stores on each GPU the chunk together with one or several rows of the neighboring chunk.

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.

2.3 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



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

Fig. 1 Distributions of a matrix in SkelCL on a system with two GPUs.

take pointers to GPU memory as input and contain program code for reading/writing single data items from/to memory.

To shield the application developer from these low-level programming issues, SkelCL extends OpenCL by introducing high-level programming patterns, called *algorithmic skeletons* [6]. Formally, a skeleton is a higher-order function that executes in a pre-defined parallel manner on containers one or more user-defined (so-called *customizing*) functions defined on primitive data, while hiding the details of parallelism and communication from the user [6].

The current version of SkelCL provides six skeletons: *map*, *zip*, *reduce*, *scan*, *mapOverlap* and *allpairs*. We start by illustrating some basic skeletons semi-formally, with v, vl and vr denoting vectors of length n:

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

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

  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 other basic skeletons are created and executed similarly. For brevity we only show the formal definition for two more skeletons:
- The zip skeleton operates on two vectors vl and vr, applying a binary customizing operator $\oplus$ pairwise:

$$zip\ (\oplus)\ [vl_1, \ldots, vl_n]\ [vr_1, \ldots, vr_n] = [vl_1 \oplus vr_1, \ldots, vl_n \oplus vr_n]$$

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

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

In SkelCL, rather than writing low-level kernels, the programmer customizes suitable skeletons by providing application-specific functions; OpenCL kernels are then generated and executed automatically by the library.

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

Listing 1 SkelCL program computing the dot product of two vectors.

Listing 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 addition. For comparison, an OpenCL-based implementation of dot product provided by NVIDIA in [11] requires 68 lines of code (kernel: 9 lines, host program: 59 lines).

2.4 The MapOverlap Skeleton

Many numerical and image processing applications perform calculations for a particular data element (e. g., a pixel) taking neighboring data elements into account. We define in SkelCL a suitable skeleton on 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 d. It applies f to each element of an input matrix m_{in} while taking the neighboring elements within the range $[-d, +d]$ into account:

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

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 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 $m_{in}[i,j]$. In Listing 2, range is specified as $d = 1$, therefore, only direct neighboring elements are accessed. Boundary checks are performed at runtime by the `get` function. When accessing elements out of the boundaries, e.g., 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 this 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, with 0 as neutral value. An equivalent OpenCL implementation requires at least twice as many lines of code, with pointer arithmetic, boundary checks and index calculations for every memory access programmed explicitly.

```
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);
```

Listing 2 MapOverlap skeleton computing the sum of neightbors for every matrix element

3 Application Studies and Experiments

The SkelCL library was used to implemented multiple applications from the fields of medical imaging, image processing and linear algebra. We present here: 1) the calculation of a Mandelbrot fractal, and 2) the Sobel edge detection. Both SkelCL implementations are compared to OpenCL versions, regarding programming effort (Lines of Code - LOC) and runtime performance.

3.1 Application Study: Mandelbrot Set

The Mandelbrot set calculation [10] is a time-consuming task which is often used as a parallel benchmark: it is easily parallelizable as all pixels can be computed simultaneously. SkelCL requires a single line of code for initialization in the host code, whereas OpenCL requires a lengthy creation and initialization of different data structures, about 20 LoC. In OpenCL host code, 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 representing pixels of the Mandelbrot fractal, is passed to the `Map` skeleton upon execution. Specifying the work-group size is mandatory in OpenCL, whereas this is optional in SkelCL. The OpenCL-based implementation has in total 118 LoCs (kernel: 28 lines, host program: 90 lines) and is thus more than twice as long as the SkelCL version with 57 lines (26, 31) (see Figure 2).

We run our implementations on a single Tesla T10 GPU with 240 streaming processor cores to compute a Mandelbrot fractal of size 4096×3072 pixels. In OpenCL two-dimensional work-groups of 16×16 are used; SkelCL uses its default one-dimensional work-group size of 256. We observe that the OpenCL-based implementation is faster by only 4% than the SkelCL version.

3.2 Application Study: Sobel Edge Detection

Listing 3 shows the sequential pseudo-code of the Sobel edge detection [9], with omitted boundary checks for brevity. For computing an output value `out[i][j]`, the input value `img[i][j]` and the direct neighboring elements

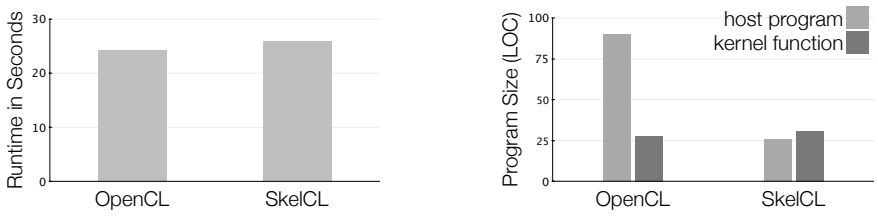


Fig. 2 Runtime and program size of the Mandelbrot application.

are needed. Listing 4 shows the SkelCL implementation using the MapOverlap skeleton and the matrix data type; it follows straightforwardly from the sequential version in Listing 3, the only difference is that for accessing elements the `get` function is used instead of the square bracket notation.

We performed runtime experiments using one NVIDIA Tesla GPU with 480 processing elements and 4 GByte memory. Figure 3 shows the runtime of two OpenCL versions (from AMD and NVIDIA SDK) vs. the SkelCL version with the MapOverlap skeleton presented in Listing 4. 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 with a size of 512×512 pixel and took the mean values of six runs. The AMD version is clearly slower than 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 from the application developer. The NVIDIA and SkelCL implementations perform similarly well. 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: it comprises only

```
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[i][j] = sqrt(h*h+v*v);
```

Listing 3 Sequential Sobel edge detection. Boundary checks are omited for brevity.

```
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 = m(img);
```

Listing 4 SkelCL implementation of the Sobel edge detection.

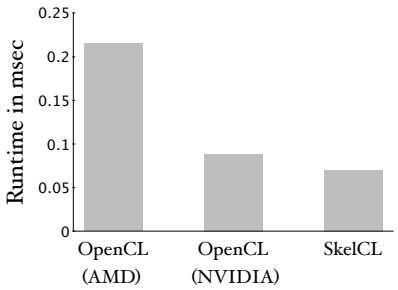


Fig. 3 Performance results for Sobel edge detection

few LOC (Listing 4) while the AMD implementation requires 37 LOC for its kernel, and the NVIDIA implementation requires even 208 LOC. No index calculations or boundary checks are necessary in the SkelCL version whereas they are crucial for a correct implementation in OpenCL.

4 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 for systems with multiple GPUs. We showed that SkelCL greatly simplifies programming of GPU systems without sacrificing performance: the overhead is usually less than 10% as compared to manually-optimized OpenCL codes. The SkelCL library is available as open source software from `http://skelcl.uni-muenster.de`.

Projects like *SkePU* [4] and *Muesli* [5] are skeleton-based approaches similar to SkelCL, but differing in focus and implementation, see [13] for comparison. There exist wrappers for OpenCL or CUDA and libraries for GPU Computing, most popular: *Thrust* [7] and *Bolt* [2]. Compiler-based approaches similar to OpenMP [12] include *OpenACC* [1] and *OmpSs-OpenCL* [3]. These projects aim at reducing boilerplate code in GPU applications, rather than introducing high-level abstractions – like SkelCL does.

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. *OpenACC Application Program Interface*, 2011. Version 1.0.
2. AMD. Bolt – A C++ template library optimized for GPUs, 2013.
3. V. K. Elangovan, R.M. Badia, and E. A. Parra. OmpSs-OpenCL programming model for heterogeneous systems. In H.Kasahara and K.Kimura, editors, *Languages and Compilers for Parallel Computing*, volume 7760 of *LNCS*, pages 96–111. Springer, 2013.
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. Ernsting and H. Kuchen. Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *Int. J. of High Performance Comput. and Networking*, 7(2):129–138, 2012.
6. S. Gorlatch and M. Cole. Parallel skeletons. In *Encyclopedia of Parallel Computing*, pages 1417–1422. 2011.
7. J. Hoberock and N. Bell (NVIDIA). Thrust: A Parallel Template Library, 2013.
8. Khronos OpenCL Working Group. *The OpenCL Specification*, Nov. 2013. Version 2.0.
9. J. Kittler. On the accuracy of the sobel edge detector. *Image and Vision Computing*, 1(1):37 – 42, 1983.

10. 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 Science*, pages 249–259, 1980.
11. NVIDIA. *NVIDIA CUDA SDK code samples*, February 2013. Version 5.0.
12. OpenMP Architecture Board. *OpenMP API*, 2013. Version 4.0.
13. M. Steuwer and S. Gorlatch. SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems. In V.Malyshkin, editor, *Parallel Computing Technologies (PaCT 2013)*, volume 7979 of *LNCS*, pages 258–272. Springer, 2013.
14. M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A portable skeleton library for high-level GPU programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182, 2011.