Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 749 – 758

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

High-Level Programming for Medical Imaging on Multi-GPU Systems using the SkelCL Library

Michel Steuwer*, Sergei Gorlatch

Department of Mathematics and Computer Science, University of Muenster, Einsteinstrasse 62, 48149 Muenster, Germany

Abstract

Application development for modern high-performance systems with Graphics Processing Units (GPUs) 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 model 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 patterns (skeletons)*; 2) memory management is simplified using *parallel container data types*; 3) an automatic *data (re)distribution* mechanism allows for scalability when using multi-GPU systems.

We use a real-world example from the field of medical imaging to motivate the design of our programming model and we show how application development using SkelCL is simplified without sacrificing performance: we were able to reduce the code size in our imaging example application by 50% while introducing only a moderate runtime overhead of less than 5%.

Keywords: SkelCL, Multi-GPU Computing, Algorithmic Skeletons, LM OSEM Algorithm, Image Reconstruction

1. Introduction

Modern high-performance computer systems increasingly employ *Graphics Processing Units* (GPUs) and other accelerators. The state-of-the-art application development for systems with GPUs is cumbersome and error-prone, because GPUs are programmed using relatively low-level models like CUDA [1] or OpenCL [2]. These programming 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 to explicitly specify parallelism in the computation. This leads to lengthy, low-level, complex 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 GPUs, as well as separate management of each GPU, including low-level pointer arithmetics and offset calculations.

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

*Corresponding author. Tel.: +49 251 8332744; fax: +49 251 8332742.
E-mail address: michel.steuwer@uni-muenster.de.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.239

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

The paper describes how the SkelCL model is used for programming a sample real-world application from the area of medical imaging, and how the model is implemented as the SkelCL programming library, using C++. Our focus is on programming methodology; therefore, we motivate our work using one typical imaging application and then study it in great detail throughout the paper.

The remainder of the paper is organized as follows. In Section 2, we introduce the application example used throughout the paper – the LM OSEM algorithm for medical image reconstruction. The application is used to identify requirements on a high-level programming model. Section 3 introduces the SkelCL programming model and its C++ implementation in the SkelCL library. In Section 4, we present experimental results for the LM OSEM algorithm using SkelCL, before we compare our contributions with related work and conclude in Section 5.

2. Iterative PET Image Reconstruction and the LM OSEM Algorithm

Our running application example in this paper is the LM OSEM algorithm [3, 4] for image reconstruction used in Positron Emission Tomography (PET). In PET, a radioactive substance is injected into a human or animal body, which is then placed inside a PET scanner that contains several arrays of detectors. As the particles of the applied substance decay, positrons are emitted (hence the name PET) and annihilate with nearby electrons, such that two photons are emitted in the opposite directions (see Fig. 1). These "decay events" are registered by two opposite detectors of the scanner which records these events in a list. Data collected by the PET scanner are then processed by a reconstruction algorithm to obtain a resulting image.

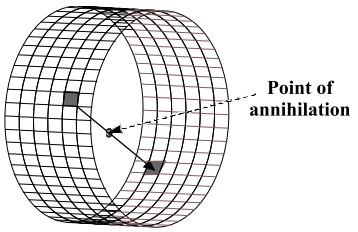

Fig. 1. Two detectors register an event in a PET-scanner

2.1. The LM OSEM Algorithm

List-Mode Ordered Subset Expectation Maximization [3] (called LM OSEM in the sequel) is a block-iterative algorithm for 3D image reconstruction. LM OSEM takes a set of events from a PET scanner and splits them into s equally sized subsets. Then, for each subset $S_l, l \in 0, \ldots, s-1$, the following computation is performed:

$$f_{l+1} = f_l c_l; \quad c_l = \frac{1}{A_N^T \mathbf{1}} \sum_{i \in S_l} (A_i)^T \frac{1}{A_i f_l}. \tag{1}$$

Here $f \in \mathbb{R}^n$ is a 3D image in vector form with dimensions $n = (X \times Y \times Z)$, A – the so called system matrix, element a_{ik} of row A_i is the length of intersection of the line between the two detectors of event i with voxel k of the reconstruction image, computed with Siddon's algorithm [5]. $\frac{1}{A_N^T \mathbf{1}}$ is the so-called normalization vector; since it can be precomputed, we will omit it in the following. The multiplication $f_l c_l$ is performed element-wise. Each subset's computation takes its predecessor's output image as input and produces a new, more precise image.

The structure of a sequential LM OSEM implementation is shown in Listing 1. The outermost loop iterates over the subsets. The first inner loop (step 1) iterates over subset's events to compute c_l, which requires three substeps: row A_i is computed from the current event using Siddon's algorithm; the local error for row A_i is computed and, finally, added to c_l. The second inner loop (step 2) iterates over all elements of f_l and c_l to compute f_{l+1}.

```
for (int l = 0; l < subsets; l++) {
  /* read subset */

  /* step 1: compute error image c_l */
  for (int i = 0; i < subset_size; i++) {
    /* compute A_i             */
    /* compute local error     */
    /* add local error to c_l */  }

  /* step 2: update image estimate f */
  for (int k = 0 ; k < image_size; k++) {
    if (c_l[k] > 0.0) { f[k] = f[k] * c_l[k]; } }  }
```

Listing 1. Sequential code for LM OSEM comprises one outer loop with two nested inner loops.

2.2. Parallelization of LM OSEM in OpenCL

LM OSEM is a rather time-consuming algorithm that needs parallelization: a typical 3D image reconstruction processing $6 \cdot 10^7$ input events for a $150 \times 150 \times 280$ PET image takes more than two hours on a modern PC.

Although the iterations of the outer loop in Listing 1 are inherently sequential, we can parallelize the two calculation steps within one iteration as shown in Fig. 2 for a system comprising one CPU and two GPUs. Note that these steps require different data distribution patterns:

Step 1: Subset's events are copied from the CPU to all GPUs (*upload*) to compute the summation part of c_l concurrently. This step requires that the complete image estimate f_l is available to all GPUs.
Step 2: For computing the next image estimate f_{l+1} in parallel, the current image estimate f_l and the error image c_l computed in step 1 have to be distributed in disjoint parts (blocks) among all GPUs.

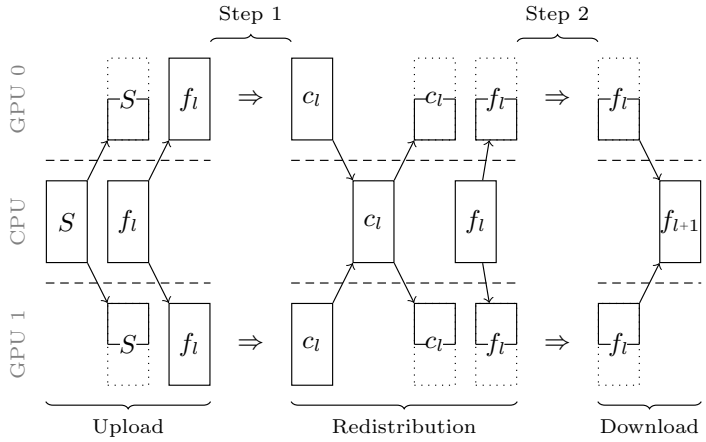

Fig. 2. Parallelization schema of the LM OSEM algorithm.

Thus, the parallelization schema in Fig. 2 requires a data *redistribution* phase between the two computation steps. During step 1, each GPU computes a partial sum of c_l. After step 1, all partial results are summed up and redistributed disjointly to all GPUs. Note that for step 1, each GPU requires a full copy of the image estimate, while in step 2 all GPUs update disjoint parts of it. After step 2, the disjoint parts of the image estimate are copied from all GPUs back to the CPU (*download*).

In the following, we describe how the parallelization schema phases in Fig. 2 are implemented using OpenCL.

Upload. The simplified OpenCL implementation of the upload phase is shown in Listing 2. Uploading of the event vector S is performed in lines 3–6, while lines 9–12 show the upload of the image estimate f_l. In OpenCL, we have to manage each GPU explicitly, therefore, for each GPU, we create a set of buffers (`s_gpu` and `f_gpu`) and we use a loop (line 1) to repeat all memory operations for each GPU. For performance reasons, we use asynchronous copy operations, specified using the `CL_FALSE` flag (line 3 and 9): this allows data transfers to multiple GPUs to overlap. We perform different operations with S (distribute among all GPUs) and f_l (copy to each GPU), therefore, there are differences when specifying the amount of bytes to copy (lines 4 and 10) and the offsets in the CPU memory (lines 5 and 11). Altogether eleven such memory operations – each with different amounts and offsets – appear in the OpenCL source code.

```
1  for (int gpu = 0; gpu < gpu_count; gpu++) {
2    // upload S
3    clEnqueueWriteBuffer( command_queue[gpu], s_gpu[gpu], CL_FALSE, 0,
4                          sizeof(float) * size_of_s / gpu_count,
5                          (void*)&s_cpu[ gpu * size_of_s / gpu_count ],
6                          0, NULL, NULL );
7
8    // upload f_l
9    clEnqueueWriteBuffer( command_queue[gpu], f_gpu[gpu], CL_FALSE, 0,
10                         sizeof(float) * size_of_f,
11                         (void*)&f_cpu[0],
12                         0, NULL, NULL ); }
```

Listing 2. Implementation of the upload phase in OpenCL (ommitting error checks for brevity)

Step 1. The implementation of step 1 performs the operations shown in Listing 1: first computing A_i, then the local error for A_i and finally adding the local error to c_l. Because of GPU memory restrictions, the OpenCL implementation is not straightforward, such that, for the sake of brevity, we will not discuss it in more detail here.

Redistribution. Listing 3 shows an OpenCL pseudocode for the redistribution phase. To download the data from all GPUs, we use the `clEnqueueReadBuffer` function and perform the operations asynchronously, but this time, we have to wait for the operations to finish. For such synchronization, OpenCL uses *events*, associated with an operation (line 3) for waiting for the operation to finish (line 4). After all downloads have finished, we combine the different values of c_l to a new value of c_l on the CPU (line 7), and upload the blocks of c_l to the GPUs. Even if we only copied data between GPUs, without processing them on the CPU, we still would have to download them to the CPU because direct GPU-to-GPU transfers are currently not possible in OpenCL.

```
1  // download all c_l values from the GPUs to the CPU
2  cl_event events[gpu_count];
3  for (int gpu = 0; gpu < gpu_count; gpu++) { clEnqueueReadBuffer( ..., &events[gpu] ); }
4  clWaitForEvents(gpu_count, events);
5
6  // combine data on CPU
7  combine( ... );
8
9  // upload block of the new c_l version to each GPU
10 for (int gpu = 0; gpu < gpu_count; gpu++) { clEnqueueWriteBuffer( ... ); }
```

Listing 3. OpenCL pseudocode for the redistribution phase

Step 2. Listing 4 shows the implementation of step 2. In OpenCL, computations are specified as *kernels* which are created from the source code specifying the computation. The computation in step 2 is, therefore, described as a string in lines 3 – 5. The operations used here are the same as in the sequential code in Listing 1.

For executing the computations of step 2, we have to perform the following steps for each GPU:

- create an OpenCL kernel from the source code (requires 50 lines of code in OpenCL);
- compile the kernel specifically for the GPU (requires 13 lines of code in OpenCL);
- specify kernel arguments one-by-one using the `clSetKernelArg` function (line 12 – 17);

- specify execution environment, i. e., how many instances of the kernel to start (line 20 – 21);
- launch the kernel (line 23 – 24).

```
1  // step 2 (in Fig. 2)
2  source_code_step_2 =
3  "__kernel void step2(__global float* f, __global float* c_l, int offset, int size) { \
4    int id = get_global_id(0) + offset;                                                 \
5    if (id < size && c_l[id] > 0.0) { f[id] = f[id] * c_l[id]; } }";
6
7  for (int gpu = 0; gpu < gpu_count; gpu++) {
8   // create kernel (50 lines of code)
9   // compile kernel (13 lines of code)
10
11  // specifying kernel arguments:
12   clSetKernelArg(kernel_step2[gpu], 0, sizeof(cl_mem), (void*)&f_buffer[gpu]);
13   clSetKernelArg(kernel_step2[gpu], 1, sizeof(cl_mem), (void*)&c_l_buffer[gpu]);
14   int offset = gpu * (size_of_f / gpu_count);
15   clSetKernelArg(kernel_step2[gpu], 2, sizeof(int), (void*)&offset);
16   int size   = MIN( (gpu + 1) * (size_of_f / gpu_count), size_of_f );
17   clSetKernelArg(kernel_step2[gpu], 3, sizeof(int), (void*)&size);
18
19  // specify execution environment
20   int local_work_size[1]  = { 32 };
21   int global_work_size[1] = { roundUp(32, size_of_f / gpu_count) };
22  // launch the kernel
23   clEnqueueNDRangeKernel(command_queue[gpu], kernel_step2[gpu],
24                          1, NULL, &global_work_size, &local_work_size, 0, NULL, NULL); }
```

Listing 4. Implementation of step 2 in OpenCL (omitting error checks for brevity)

Download. The implementation of the download phase is similar to the upload phase (Listing 2).

2.3. Requirements to a High-Level Programming Model

The described implementation of the LM OSEM algorithm reveals the main problems and difficulties that the application developer has to overcome when using OpenCL. Our analysis show that to simplify programming for a system with multiple GPUs, the following high-level abstraction are desirable:

Parallel container data types. Compute intensive applications, like the LM OSEM algorithm, typically operate on a (possibly big) set of data items. The LM OSEM algorithm operates on lists (of events) and three-dimensional images (the reconstruction image f_l and the error image c_l). As shown in Listing 2, managing memory is 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 processors in a system and it should provide an easy-to-use interface for the application developer.

Recurring patterns of parallelism. While the concrete operations performed in the LM OSEM algorithm are (of course) application-specific, the general structure of parallelization resembles parallel patterns that are commonly used in many applications. In step 1, for computing the error image c_l, the same sequence of operations is performed for every event from the input subset, which is the well-known *map* pattern of data-parallel programming [6]. In step 2, two images (the current image estimate f_l and the error image c_l) are combined element-wise into the output image (f_{l+1}), see line 5 of Listing 4, which is again the common *zip* 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 manually in much detail.

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. As shown in Fig. 2, the LM OSEM algorithm requires two different distributions for its two computational steps. Distributing and re-distributing data in OpenCL is cumbersome because data transfers have to be managed manually and performed via the CPU, as shown in Listing 3. 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.

3. The SkelCL Programming Model and the SkelCL Library

We develop our SkelCL [7] programming model as an extension of the standard OpenCL programming model [2], which adds to OpenCL the three features that we identified as desirable in Section 2.3:

- parallel container data types for unified memory management between CPU and GPUs;
- algorithmic skeletons for expressing parallel computation patterns on GPUs;
- data distribution and re-distribution mechanisms for programming multi-GPU systems.

SkelCL inherits all properties of OpenCL, including its support for heterogeneous parallel systems. 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 terms host and device.

3.1. Parallel 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. Upon creation of a container 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 like this:

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

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 frefree the programmer from low-level memory operations like allocation (on GPU) and data transfers between CPU and GPU, such as those shown in Listing 2.

3.2. Patterns of Parallelism (Algorithmic Skeletons)

In original OpenCL, computations are expressed as *kernels* which are executed in a parallel manner on a GPU. As shown in Listing 4, the application developer must specify in the OpenCL program 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 free the application developer from these low-level programming issues, SkelCL extends OpenCL by means of high-level programming patterns, called *algorithmic skeletons* [6]. Formally, a skeleton is a higher-order function that is parameterized by one or more application-specific *customizing functions* and is executable in a predefined parallel manner, 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*. Due to lack of space, we only describe the first three skeletons for vectors, with v, vl and vr denoting vectors with elements v_i, vl_i and vr_i where $0 < i \leq n$. For a more extensive and formal discussion of skeletons see [6].

- The map skeleton applies a unary 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; ");
```

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

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

The map skeleton is defined accordingly for matrices.

- The zip skeleton operates on two vectors *vl* and *vr*, applying a binary operator $\oplus$ pairwise:
$$zip\ (\oplus)\ [vl_1, vl_2, \ldots, vl_n]\ [vr_1, vr_2, \ldots, vr_n] = [vl_1 \oplus vr_1, vl_2 \oplus vr_2, \ldots, vl_n \oplus vr_n]$$
In SkelCL, a zip skeleton object for adding two vectors is created like this:

```
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 zip skeleton is defined accordingly for matrices.

- The reduce skeleton computes a scalar value from a vector using an associative binary 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 and called as follows:

```
Reduce<float(float)> sum_up( "float func(float x, float y)return x+y; ");

result = sum_up( inputVector );
```

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, 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.

3.3. Data Distributions

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 managing memory ranges which are shared or partitioned across multiple GPUs.

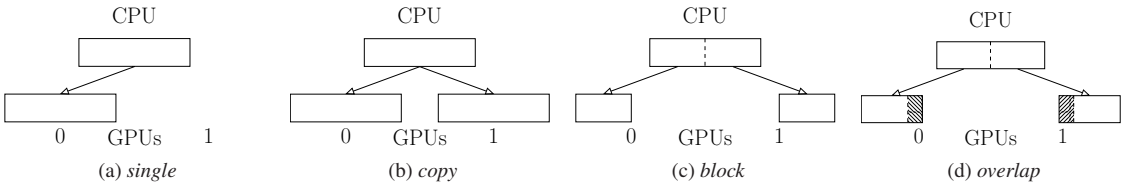

Fig. 3. Distributions of a vector in SkelCL.

Four kinds of distribution are currently available to the application developer in SkelCL: *single*, *copy*, *block*, and *overlap* (see Fig. 3 for illustration on a system with two GPUs). If set to the *single* distribution (Fig. 3a), container's whole data is stored on a single GPU (the first GPU if not specified otherwise). The *copy* distribution (Fig. 3b) copies container's entire data to each available GPU. With the *block* distribution (Fig. 3c), each GPU stores a contiguous, disjoint block of the container. The *overlap* distribution (Fig. 3d) is used for the mapOverlap skeleton: 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. As shown in Listing 3, implementing such data transfers in the 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.

3.4. The SkelCL Library

The SkelCL Library is our current implementation of the SkelCL programming model. It provides to the user a C++ API that implements the features of the SkelCL programming model, and thus liberates the application developer from writing low-level boilerplate code. In addition, the library provides some commonly used utility functions, e. g., for program initialization. For flexibility, SkelCL skeletons can accept additional arguments if the customizing function works not only on a skeleton's input containers, but needs access to additional data [7]; containers passed as additional arguments to a skeleton are automatically transfered to the GPUs. SkelCL can also be used in combination with existing OpenCL codes, as SkelCL is designed as an extension of OpenCL, rather than a replacement for it.

In OpenCL, kernels are compiled at runtime of the host program in order to be executable on different GPUs. Therefore, in the SkelCL library implementation, the customizing functions are provided as strings to their skeletons. SkelCL implementation merges the customizing function with the pre-implemented skeleton-specific program code to build a valid OpenCL kernel automatically. The generated kernel fetches one or more data items from its input containers (vectors or matrices), passes them to the customizing function, and yields the function's result, e. g., by writing it to the output container. Rather than working with pointers to GPU memory (like kernels do), customizing functions in SkelCL take a single data item as input and return a single result. The SkelCL implementation of the vector container resembles the interface of the vector from the C++ Standard Template Library (STL), i. e., it can be used as a replacement for the standard vector. Internally, the containers manage pointers to the corresponding areas of the main memory (accessible by the CPU) and GPU memory. For possible optimizations of the kernel's source code, we rely on the optimization capabilities of the OpenCL compiler.

In some situations, our SkelCL implementation can optimize data transfers: e. g., after executing a skeleton, the output data remains in the GPU memory; this has the advantage that if the output container is used as the input to another skeleton, no data transfer has to be performed. Such *lazy copying* implemented in SkelCL minimizes costly data transfers between the CPU and GPUs.

4. Implementing the LM OSEM Algorithm using SkelCL

We assess the programming effort and runtime performance of our approach by comparing the SkelCL implementation of the LM OSEM algorithm against its OpenCL implementation.

Programming effort. The parallel SkelCL code in Listing 5 fully retains the original sequential structure of Listing 1, which makes it well structured and easily understandable. The sequential loops are replaced by skeleton calls (line 18 and 24) which take the code from the corresponding loop's body, and only 5 lines of code are added for data distribution (lines 13 – 15 and 20 – 21). The computation of the error image is expressed using a map skeleton (lines 1 – 4). Due to memory restrictions, it is not possible to apply the map skeleton to the whole vector `s`: map is rather applied to a vector of size 512 (line 9), such that for every index a block of elements is processed using the loop in line 2. Since detecting memory restrictions and applying blocking automatically is non-trivial, SkelCL currently relies on the user to resolve such situations. The event vector, the image estimate, and the error image are passed as additional arguments to the skeleton. The zip skeleton is used for updating the error image (line 5 – 6).

The SkelCL-based implementation of the LM OSEM is considerably shorter than the OpenCL code: with 232 lines of code (customizing functions: 200 lines, host program: 32 lines) as compared to 436 lines of code (kernel functions: 193 lines, host program: 243 lines) in the OpenCL-based implementation; we save almost 50% of the lines of code. When using SkelCL, we do not have to implement the tedious and lengthy initialization of OpenCL. Expressing the computations as skeletons liberates us from dealing with pointers in the kernel and repeatedly performing the same sequence of steps for each computation. By using container data types, we avoid additional programming effort to implement data transfers between CPU and GPU or between multiple GPUs, and we obtain a multi-GPU-ready implementation of LM OSEM for free.

```
1  Map<void(int)> computeC_l("void func(int index, event* s, float* f, float* c_l) {   \
2    for (int i = index; i < subset_size; i+=512) {                                     \
3      // compute A_i // compute local error // add local error to c_l                 \
4    } }");
5  Zip<float(float, float)> updateF("float func(float f_i, float cl_i) {               \
6        if (cl_i > 0.0) return f_i * cl_i; else return f_i; }");
7
8  Vector<float> f   = readStartImage();
9  Vector<int> index = createIndexVector(512);
10 for (i = 0; i < num_iterations; ++i) {
11   Vector<event> s = read_subset(); // read subset s from file
12   Vector<float> c_l(image_size);
13   s.setDistribution(block);
14   f.setDistribution(copy);
15   c_l.setDistribution(copy, add);
16
17   /* step 1. compute error image c_l */
18   computeC_l(index, s, f, c_l);
19
20   f.setDistribution(block);
21   c_l.setDistribution(block);
22
23   /* step 2. update image estimate f */
24   f = updateF(f, c_l); }
```

Listing 5. SkelCL code of the LM OSEM algorithm

Runtime performance. We tested the performance of our implementations using an NVIDIA Tesla S1070 system comprising 4 Tesla GPUs. Each GPU consists of 240 streaming processors.

Fig. 4 compares the runtime of one iteration for our SkelCL and OpenCL implementations using one, two, and four GPUs, correspondingly. While the differences in the programming effort to implement the two versions are significant, the differences in runtime are very small. When running on a single GPU, both implementations take the same time (3.66 seconds) to complete. With two and four GPUs, the OpenCL implementation slightly outperforms the SkelCL implementation, being 1.2% and 4.7% faster. We presume that the increasing overhead is caused by the more complex data distribution performed when using more GPUs. Comparing to the significant reduction in programming effort (50%), the runtime overhead of less than 5% is arguably a moderate one.

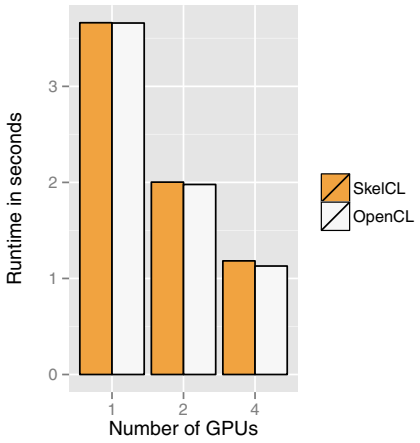

Fig. 4. Average runtime of one iteration of the LM OSEM algorithm using SkelCL and OpenCL.

5. Conclusion and Related Work

This paper presented the SkelCL high-level programming model for multi-GPU systems and its implementation as a library. We focused on programming methodology and, therefore, deliberately restricted ourselves to a single sample real-world application as motivation example and benchmark for experiments. Additional application examples can be found on the SkelCL website `http://skelcl.uni-muenster.de`, including LU decomposition, computation of the Mandelbrot set, matrix multiplication, Jacobi stencil computations, B+ tree traversal, the Mersenne Twister, etc. SkelCL is freely available as open source software.

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. Our SkelCL library significantly reduces the amount of source code necessary to implement the sample imaging application (by 50%) and frees the application developer from low-level memory management and other tedious programming tasks. The performance experiments show that SkelCL introduces a moderate overhead of less than 5% as compared to the arguably more complicated and error-prone OpenCL implementation.

A considerable amount of work exists in the filed of algorithmic skeletons; for an overview we refer to [6]. There are several related approaches to raise the level of program abstraction in GPU programming. While SkelCL can be used for programming multiple OpenCL capable GPUs, the CUDA-based *Thrust* [8] library simplifies programming only for a single NVIDIA GPU. As SkelCL, *SkePU* [9] is a skeleton library targeting multi-GPU systems. In contrast to our work which is based entirely on the portable OpenCL, SkePU is implemented with multiple back-ends which restrict the application developer to the back-ends' smallest common set of functions and, thus, prevents the user from applying optimizations, like using the fast local GPU memory.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments.

References

[1] NVIDIA CUDA C Programming Guide, Version 5.0 (February 2013).
[2] A. Munshi, The OpenCL Specification, Version 1.2.
[3] A. J. Reader, K. Erlandsson, M. A. Flower, R. J. Ott, Fast Accurate Iterative Reconstruction for Low-Statistics Positron Volume Imaging, Physics in Medicine and Biology 43 (4) (1998) 835.
[4] M. Schellmann, S. Gorlatch, D. Meiländer, T. Kösters, K. Schäfers, F. Wübbeling, M. Burger, Parallel Medical Image Reconstruction: From Graphics Processors to Grids, in: Proceedings of the 10th International Conference on Parallel Computing Technologies, PaCT '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 457–473.
[5] R. L. Siddon, Fast Calculation of the Exact Radiological Path for a Three-Dimensional CT Array, Medical Physics 12 (2) (1985) 252–255.
[6] S. Gorlatch, M. Cole, Parallel Skeletons, in: Encyclopedia of Parallel Computing, 2011, pp. 1417–1422.
[7] 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.
[8] J. Hoberock, N. Bell, Thrust: A Parallel Template Library (2009).
[9] 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.