Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library

Michel Steuwer, Philipp Kegel, and Sergei Gorlatch

Department of Mathematics and Computer Science
University of Münster, Münster, Germany
Email: {michel.steuwer,philipp.kegel,gorlatch}@uni-muenster.de

***Abstract*—Application programming for GPUs (Graphics Processing Units) is complex and error-prone, because the popular approaches — CUDA and OpenCL — are intrinsically low-level and offer no special support for systems consisting of multiple GPUs. The SkelCL library presented in this paper is built on top of the OpenCL standard and offers pre-implemented recurring computation and communication patterns (skeletons) which greatly simplify programming for multi-GPU systems. The library also provides an abstract vector data type and a high-level data (re)distribution mechanism to shield the programmer from the low-level data transfers between the system's main memory and multiple GPUs. In this paper, we focus on the specific support in SkelCL for systems with multiple GPUs and use a real-world application study from the area of medical imaging to demonstrate the reduced programming effort and competitive performance of SkelCL as compared to OpenCL and CUDA. Besides, we illustrate how SkelCL adapts to large-scale, distributed heterogeneous systems in order to simplify their programming.**

***Keywords*-GPU Computing, GPU Programming, Multi-GPU Systems, SkelCL, OpenCL, Algorithmic Skeletons**

I. Introduction

The two popular programming approaches for systems with *Graphics Processing Units* (GPUs) — CUDA and OpenCL [1]–[3] — work at a low level of abstraction. They require the programmer to explicitly manage the GPU's memory, including data allocations and transfers to/from the system's main memory. This leads to long, complex and, therefore, error-prone code. The emerging multi-GPU systems are particularly challenging for the application developer: they additionally require explicit data exchanges between individual GPUs, including low-level pointer arithmetics and offset calculations, as well as an adaption of algorithms for execution on multiple GPUs.

In this paper, we introduce SkelCL — a library for high-level GPU programming. SkelCL comprises two major abstraction mechanisms: a set of algorithmic skeletons [4] and an abstract vector data type. Skeletons offer several frequently used pre-implemented patterns of parallel communication and computation to the application developer [5]. The vector data type enables implicit data transfers between the system's main memory and GPUs. Using these abstractions, SkelCL frees the programmer from writing low-level and error-prone boilerplate code when programming multi-GPU systems, and still provides all useful features of the OpenCL standard.

The focus of this paper, as compared to our introductory article [6], is the set of specific features and mechanisms of SkelCL for programming multi-GPU systems on a high level of abstraction. We describe how SkelCL implements skeletons on multi-GPU systems and show how SkelCL's concept of *data distribution* frees the user from low-level memory management, without sacrificing performance in real-world applications that use multiple GPUs.

The paper is organized as follows. Section II introduces SkelCL's key features, while Section III focuses on programming multi-GPU systems using SkelCL. In Section IV, we present a real-world application study from the area of medical imaging and experimental results. Section VI compares our approach to related work.

II. Overview of SkelCL

We build our SkelCL approach on top of the OpenCL standard [1], because OpenCL is hardware- and vendor-independent. Thereby, we achieve that multi-core CPUs, GPUs and other accelerators — called *devices* in OpenCL — can be employed using a uniform programming model. An OpenCL program is executed on a *host* system which is connected to one or several OpenCL devices. In the next two subsections, we describe particular problems of application programming using OpenCL and demonstrate how SkelCL addresses these problems.

A. Algorithmic Skeletons

In OpenCL, special functions (*kernels*) are executed in parallel on a device. Kernels are compiled at runtime to allow for portability across different devices. The programmer must specify in the host program how many instances of a kernel are executed. Kernels usually take pointers to GPU memory as input and contain program code for reading/writing data items. Pointers have to be used carefully, because no boundary checks are performed by OpenCL.

To shield the programmer from these low-level programming issues, SkelCL extends OpenCL by means of *algorithmic skeletons*. A skeleton is, formally, a higher-order function that executes one or more user-defined functions

©2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/IPDPSW.2012.229

in a pre-defined parallel manner, hiding the details of parallelism and communication from the user [5]. The current version of SkelCL provides four skeletons: *map*, *zip*, *reduce*, and *scan*. We describe these skeletons semi-formally, with $[x_1, \ldots, x_n]$ denoting a vector of size n:

- The map skeleton applies a unary function (f) to each element of an input vector ($[x_1, \ldots, x_n]$), i. e. $map(f)([x_1, \ldots, x_n]) = [f(x_1), \ldots, f(x_n)]$.
- The zip skeleton operates on two input vectors ($[x_1, \ldots, x_n], [y_1, \ldots, y_n]$), applying an associative binary operator ($\oplus$) to all pairs of elements, i. e. $zip(\oplus)([x_1, \ldots, x_n], [y_1, \ldots, y_n]) =$ $[(x_1 \oplus y_1), \ldots, (x_n \oplus y_n)]$.
- The reduce skeleton computes a scalar value from an input vector using a binary operator ($\oplus$), i. e. $reduce(\oplus)([x_1, \ldots, x_n]) = x_1 \oplus x_2 \oplus \cdots \oplus x_n$. For the result to be correct the operator has to be associative but may be non-commutative.
- The scan skeleton computes a vector of prefix-sums by applying an associative binary operator ($\oplus$), i. e. $scan(\oplus)([x_1, \ldots, x_n]) =$ $[x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n]$.

The aforementioned skeletons have been selected, because they are useful for a broad range of applications. Moreover, the computation patterns of these skeletons match the data-parallel SIMD (Singe Instruction, Multiple Data) execution model used by GPUs. In SkelCL, rather than writing low-level kernels, the programmer customizes a suitable skeleton with a user-defined function which takes a single data item as input and returns a single result instead of working with pointers (like traditional OpenCL kernels do). Input vectors are provided as arguments on skeleton execution.

To customize a skeleton, the application developer passes the source code of the user-defined function as a plain string to the skeleton. SkelCL merges the user-defined function's source code with pre-implemented skeleton-specific program code, thus creating a valid OpenCL kernel automatically. The created kernel is then compiled by the underlying OpenCL implementation before execution. Thanks to this procedure SkelCL can operate on top of every standard compliant OpenCL implementation and does not require a customized compiler.

In real-world applications (see, e. g., Section IV), user-defined functions often work not only on a skeleton's input vector, but may also take additional inputs. With only a fixed number of input arguments, traditional skeletons would not be applicable for the implementation of such applications. The novelty of SkelCL skeletons is that they can accept additional arguments which are passed to the skeleton's user-defined function. Since SkelCL's skeletons rely on pre-defined OpenCL code, they are by default not compatible with additional arguments. Therefore, the SkelCL implementation at runtime adapts this code to the function it uses, such that the skeleton passes its additional arguments to the user-defined function.

```
/* create skeleton Y <- a * X + Y */
Zip<float> saxpy (
    "float func(float x, float y, float a)\
        { return a*x+y; }" );

/* create input vectors */
Vector<float> X(SIZE); fillVector(X);
Vector<float> Y(SIZE); fillVector(Y);
float a = fillScalar();

Y = saxpy( X, Y, a );       /* execute skeleton */

print(Y.begin(), Y.end()); /* print results */
```

Listing 1. The BLAS *saxpy* computation using a zip skeleton with additional arguments

Listing 1 shows an example implementation of the *single-precision real-alpha x plus y* (SAXPY) computation – a commonly used BLAS routine – in SkelCL. SAXPY is a combination of scalar multiplication of a with vector X followed by vector addition with Y. In the example, the computation is implemented by a zip skeleton: vectors X and Y are passed as input, while factor a is passed to the user-defined function as an additional argument. The additional argument is simply appended to the argument list when the skeleton is executed. Note that all input values of the user-defined function are scalar values rather than vectors or pointers. Besides scalar values, like shown in the example, vectors can also be passed as additional arguments to a skeleton. This feature is implemented in SkelCL using variadic templates from the new C++ standard [7], [8].

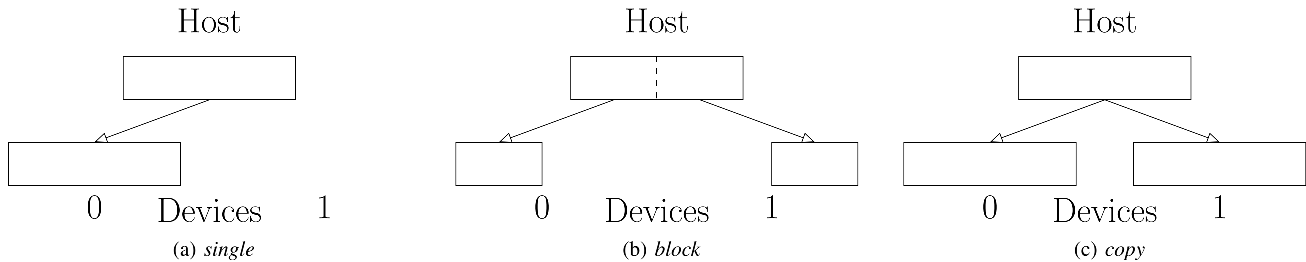
B. Data type `Vector`

OpenCL considers the system's main memory, called host memory, as separated from the device memory. Hence, the programmer has to explicitly manage data transfers between these distinct memory spaces. OpenCL offers functions for allocating memory on devices and transferring data from the host to the device (*upload*) or vice versa (*download*).

SkelCL hides low-level memory management and data transfers by means of its abstract *vector* data type. A vector defines a contiguous memory range where data is accessible by both CPU and GPU, such that the application developer does not have to program data transfers between host memory and a GPU.

Internally, a vector holds pointers to ranges of host memory and GPU memory. These memory ranges are kept in a consistent state automatically: when a vector is accessed by the CPU, its data is downloaded from the GPU to the host memory implicitly. Likewise, the data is uploaded to GPU memory before it can be accessed by this GPU.

Since GPU-based applications often perform consecutive computations on data that is accessed only by the GPU, SkelCL tries to optimize data transfers between the host



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

Figure 1. Distributions of a vector in SkelCL.

memory and GPU by performing them *lazily*: they are deferred as long as possible or avoided completely if possible. For example, when a map skeleton's output vector is passed as an input vector to a reduce skeleton, the vector's data resides on the GPU and no data transfer is performed. Therefore, in this case SkelCL can avoid a costly data transfer entirely.

III. Programming Multi-GPU Systems

Additional challenges arise when a system comprises multiple GPUs. In particular, communication and coordination between multiple GPUs and the host have to be implemented. Many low-level details, like pointer arithmetics and offset calculations, are necessary when using OpenCL or CUDA for this purpose. In this section, we demonstrate how SkelCL helps the developer to program multi-GPU systems at a high level of abstraction.

A. Data distribution

SkelCL's vector data type abstracts from memory ranges on multiple GPUs, such that the vector's data is accessible by each GPU. However, each GPU may access different parts of a vector or may even not access it at all. For example, when implementing work-sharing on multiple GPUs, the GPUs will access disjoint parts of input data, such that copying only a part of the vector to a GPU would be more efficient than copying the whole data to each GPU.

For specifying partitionings of vectors in multi-GPU systems, the concept of *distribution* is introduced in SkelCL. A distribution describes how the vector's data is distributed among the available GPUs. It allows the programmer to abstract from the challenges of managing memory ranges which are shared or partitioned across multiple devices: the programmer can think of a distributed vector as of a self-contained entity.

Figure 1 shows three distributions which are currently implemented in SkelCL and offered to the programmer: *single*, *block*, and *copy*. If set to *single* distribution (Figure 1a), vector's whole data is stored on a single GPU (the first GPU if not specified otherwise). With *block* distribution (Figure 1b), each GPU stores a contiguous, disjoint part of the vector. The *copy* distribution (Figure 1c) copies vector's entire data to each available device. A newly created vector can adopt any of these distributions.

The vector distribution can be changed at runtime either explicitly by the programmer or implicitly by the system. A change of distribution implies data exchanges between multiple GPUs and the host, which are performed implicitly by SkelCL. These implicit data exchanges are also performed lazily, i. e. only if really necessary, as described in Section II. Implementing such data transfers in OpenCL manually is a cumbersome task: data has to be downloaded to the host before it can be uploaded to other devices, including the corresponding length and offset calculations; this results in a lot of low-level code which is completely hidden when using SkelCL.

A special situation arises when the distribution is changed from the *copy* distribution, where each GPU holds its own full copy of the data. In such a case, each GPU may hold a different version of the vector as data modifications are only performed locally on the GPU. In order to maintain SkelCL's concept of a self-contained vector, these different versions must be combined using a user-specified function when the distribution is changed. If no function is specified, the copy of the first device is taken as the new version of the vector; the copies of the other devices are discarded.

B. Skeletons for Multiple GPUs

The skeletons of SkelCL possess specific features for working in multi-GPU systems. They take into account the distribution of their input vectors: each GPU that holds a part or a complete copy of a vector is involved in the execution of the skeleton. Therefore, all GPUs automatically cooperate in the skeleton execution if its input vector is *block*-distributed, whereas the skeleton is executed on one GPU if the vector distribution is *single*. If a skeleton's input vector is *copy*-distributed, then all GPUs execute the same skeleton on their own copies.

Vectors can be passed to skeletons either as main inputs or as additional arguments. For main input vectors, a skeleton-specific default distribution is set automatically by SkelCL, but the programmer can override the defaults, i. e., specify a distribution that fits best for the application. For vectors passed as an additional argument, no meaningful default

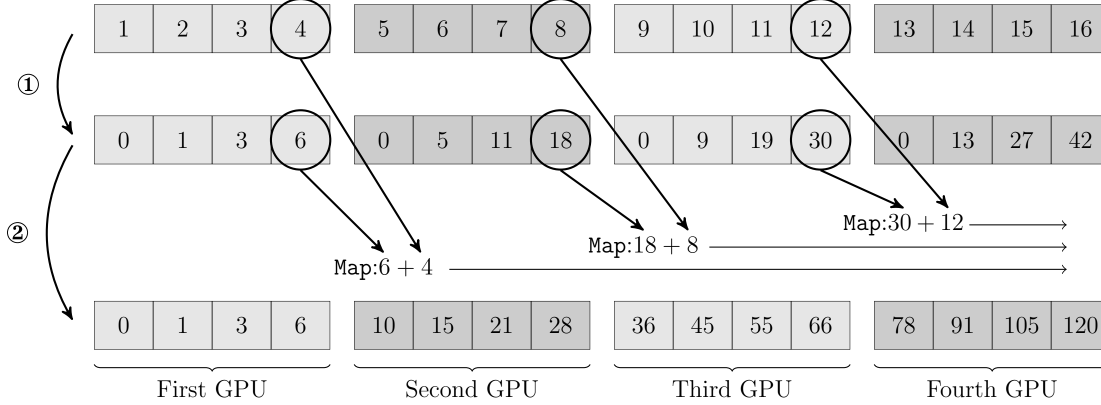


Figure 2. Scan on four GPUs: ① All GPUs scan their parts independently. ② map skeletons are created automatically and executed to produce the result.

distribution can be provided by the system, because the access pattern for the vector is determined by the user-defined function. Therefore, the user has to specify explicitly the distribution for these vectors.

C. Implementation of Skeletons on Multiple GPUs

Map and zip: In a multi-GPU setting, each GPU executes the map's unary function on its part of the input vector. The same holds for the zip skeleton, but it requires both input vectors to have the same distribution, and, in case of the single-distributed vectors, they also have to be stored on the same GPU. If this requirement is not satisfied, SkelCL automatically changes the input vector distribution to block distribution. This distribution is also set by default for input vectors with no distribution specified by the user. Both, the map and zip skeleton, set their output vector distribution to that of their input vectors.

Reduce: The reduce skeleton automatically performs all necessary synchronization and communication between CPU and GPUs, in three steps:

1) Every GPU executes a local reduction for its local part of data;
2) The results of all GPUs are gathered by the CPU;
3) The CPU reduces these intermediate results to compute the final result.

The output vector of the reduce skeleton holds only a single element, therefore, the output vector distribution is set to single.

Scan: An example of the scan skeleton executed on four devices using addition as operation is shown in Figure 2. The input vector $[1, \ldots, 16]$ is distributed using the *block* distribution by default (shown in the top line). After performing the scan algorithm on all devices (second line of the figure), map skeletons are built implicitly using the marked values and executed on all devices except the first one. This produces the final result, as shown in the bottom line.

The SkelCL implementation of the scan skeleton assumes the GPUs to have a fixed order, such that each GPU (except the first one) has a predecessor:

1) Every GPU executes a local scan algorithm for its local part of data;
2) The results of all GPUs are downloaded to the host;
3) For each GPU (except the first one), a map skeleton is implicitly created that combines the result of the GPU's predecessors with all elements of its part using the user-defined operation of the scan skeleton;
4) The newly created map skeletons compute the final results on all GPUs.

The output vector is block-distributed among all GPUs.

IV. Application Study: List-mode OSEM

To demonstrate the advantages of SkelCL as compared to the contemporary GPU programming models, we implemented a real-world application from the field of medical imaging using SkelCL, OpenCL, and CUDA. In this section, we compare these three implementations regarding: 1) programming effort, and 2) runtime performance.

List-Mode Ordered Subset Expectation Maximization (list-mode OSEM) [9], [10] is a time-intensive, production-quality numerical algorithm for *Positron Emission Tomography* (PET): it reconstructs three-dimensional images from huge sets of so-called *events* recorded by a tomograph. Each event recorded represents a *Line Of Response* (LOR) which intersects the scanned volume.

A simplified sequential code of list-mode OSEM is shown in Listing 2. The algorithm splits the events into *subsets* which are processed iteratively: all LORs of subset events and their corresponding intersection *paths* are computed and merged into an *error image*. The error image is merged with the initially empty *reconstruction image*, which is then refined in each iteration. In the following, we refer to the computation of the error image as step 1 (lines 5 to 13), and the update of the reconstruction image as step 2 (lines 15 and 16) of the algorithm.

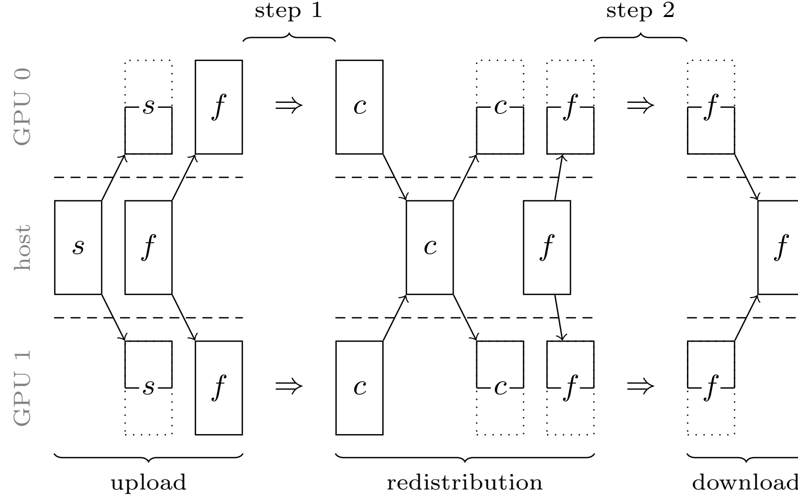


Figure 3. Data distribution changes and computations during a single subset iteration of list-mode OSEM using two GPUs.

```
1  for (l = 0; l < num_subsets; ++l) {
2    /* read subset from file */
3    events = read_events();
4    /* compute error image c (step 1) */
5    for (i = 0; i < num_events; ++i) {
6      /* compute path of LOR */
7      path = compute_path(events[i]);
8      /* compute error */
9      for (fp = 0, m = 0; m<path_len; ++m)
10       fp += f[path[m].coord] * path[m].len;
11       /* add path to error image */
12       for (m = 0; m<path_len; ++m)
13         c[path[m].coord] += path[m].len / fp;
14   }
15   /* update reconstruction image f (step 2) */
16   for (j = 0; j < image_size; ++j)
17     if (c[j] > 0.0) f[j] *= c[j];
18 }
```

Listing 2. Simplified sequential code of list-mode OSEM.

A. Parallelization strategy

For parallelization, we consider two possible decomposition strategies for the OSEM algorithm as initially suggested in [11]: Projection Space Decomposition (PSD) and Image Space Decomposition (ISD).

In PSD, the subsets are split into sub-subsets that are processed simultaneously while all processing units access a common reconstruction and error image. Using this approach, we are able to parallelize step 1 of the algorithm, but step 2 is performed by a single processing unit. On a multi-GPU system, we have to copy the reconstruction image to all GPUs before each iteration, and we have to merge all GPUs' error images computed in step 1 before proceeding with step 2. While both steps are easy to implement, step 2 does not efficiently use the available processing units.

In ISD, the reconstruction image is partitioned, such that each processing unit processes the whole subset with respect to a single part of the reconstruction image. Thus we are able to parallelize both steps of list-mode OSEM, but each processing unit still accesses the whole reconstruction image in order to compute the error value for each path before merging it with the error image. On a multi-GPU system, the whole subset has to be copied to each GPU in step 1. ISD requires large amounts of memory (up to several GB in practically relevant cases) to save all paths computed in step 1. Summarizing, it is hard to implement step 1 on the GPU, while step 2 can be parallelized easily.

Therefore, we decided to use a hybrid strategy for implementing list-mode OSEM on a multi-GPU system: Step 1 is parallelized using the PSD approach, while we use ISD for step 2. This results in the sequence of five phases shown both in Figure 3 and Listing 3:

1) *Upload:* the subset (s) is divided into sub-subsets, one sub-subset per GPU. One sub-subset and the reconstruction image (f) are uploaded to each GPU.
2) *Step 1:* each GPU computes a local error image (c) for its sub-subset using a map skeleton with additional arguments.
3) *Redistribution:* the local error images that are distributed on all GPUs are downloaded and combined into a single error image on the host by performing element-wise addition. Afterwards, the combined error image and reconstruction image are partitioned, in order to switch the parallelization strategy from PSD to ISD. The corresponding parts of both images are distributed to the GPUs again.
4) *Step 2:* each GPU updates its part of the reconstruction image using a zip skeleton
5) *Download:* finally, all parts of the reconstruction image are downloaded from the GPUs to the host and merged into a single reconstruction image.

The SkelCL program in Listing 3 reflects the described

```
for (l = 0; l < num_subsets; l++) {
  /* 1. Upload: distribute events to devices*/
  Vector<Event> events(read_events());
  events.setDistribution(Distribution::block);
  f.setDistribution(Distribution::copy);
  c.setDistribution(Distribution::copy(add));
  /* 2. Step 1: compute error image
        (map skeleton) */
  mapComputeC(index,events,events.sizes(),f,c);
  c.dataOnDevicesModified();
  /* 3. Redistribution: distribute
        reconstruction image to devices; reduce
        (element-wise add) all error images and
        distribute result to devices */
  f.setDistribution(Distribution::block);
  c.setDistribution(Distribution::block);
  /* 4. Step 2: update reconstruction image
        (zip skeleton) */
  zipUpdate(f, c, f);
  /* 5. Download: merge reconstruction image
        (is performed implicitly) */ }
```

Listing 3. Parallel implementation of list-mode OSEM in SkelCL.

five phases in a concise, high-level manner, as shown by the corresponding comments. The subset, the error image, and the reconstruction image are declared as SkelCL vectors which enables an easy and automatic data transfer between GPUs. As data transfers are performed implicitly by SkelCL, the upload phase (1.) is implemented by simply setting vector distributions (lines 4–6), while the download phase (5.) is omitted entirely.

B. Programming effort: SkelCL vs. OpenCL & CUDA

Using the described hybrid parallelization strategy, we developed three parallel implementations of list-mode OSEM using: 1) CUDA, 2) OpenCL, and 3) SkelCL. To study the programming effort, we compare the program sizes in LOC (lines of code). Though LOC is not a universal measure for programming effort, we consider it as a reasonable first approximation.

We observe (see Figure 4a) that while the kernel size in CUDA and OpenCL, or the user-defined function in SkelCL, respectively, are rather similar (about 200 LOC), the lengths of the corresponding host programs differ considerably. Unlike CUDA, OpenCL requires code for selecting the target platform and an OpenCL device and for compiling kernel functions at runtime. For a single GPU, the OpenCL-based implementation has the longest code (206 LOC), i. e. about 2.5 times longer than the CUDA-based host program (88 LOC) and more than 11 times longer than the SkelCL program. Our SkelCL-based implementation has 18 LOC, i. e. its length is only about 20% of the CUDA-based version.

Using multiple GPUs in OpenCL and CUDA requires explicit code for additional data transfers between GPUs. Prior to CUDA 4.0, also multi-threaded code was required to manage several GPUs. This accounts for additional 42 LOC for the CUDA-based implementation and additional 37 LOC for the OpenCL-based one. In SkelCL, only 8 additional LOC are necessary to describe the changes of data distribution. These lines are easily recognizable in the SkelCL program (lines 4–6, 11–12 in Listing 3, plus 3 lines during the initialization) and make this high-level code arguably better understandable and maintainable than the CUDA and OpenCL versions.

C. Performance experiments: SkelCL vs. OpenCL & CUDA

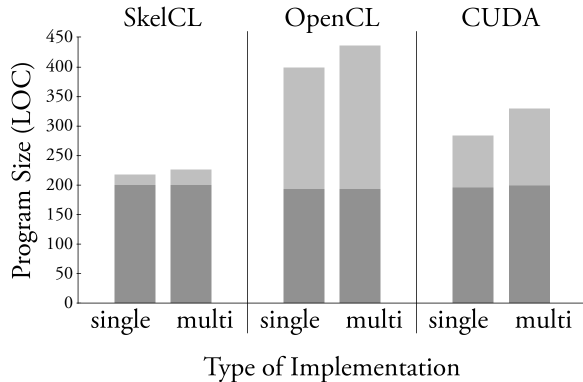
We evaluated the runtimes of our three implementations of list-mode OSEM, by reconstructing an image of $150 \times 150 \times 280$ voxels from a real-world PET data set with about 10^8 events. From this data set, about 10^2 equally sized subsets are created. In our experiments, we measured the average runtime of processing one subset. In a full reconstruction application, all subsets are processed multiple times, thus making list-mode OSEM a time-intensive application that runs several hours on a single-core CPU. Unlike CUDA, both OpenCL and SkelCL compile kernels at runtime. As compilation is only required once, when launching the implementation, but not during the subset iterations, we excluded compilation time from our runtime measurements. We ran our implementations on a system comprising a quad-core CPU (Intel Xeon E5520, 2.26 GHz) and an NVIDIA Tesla S1070 system with 4 Tesla GPUs. Each GPU consists of 240 streaming processors. The CPU has 12 GB of main memory, while each GPU owns 4 GB of dedicated memory.

Figure 4b shows the runtime of our three implementations of list-mode OSEM using up to four GPUs. We observe that CUDA always provides best performance, being about 20% faster than OpenCL and SkelCL on the same number of GPUs. As compared to the OpenCL-based implementation, SkelCL introduces only a moderate overhead of less than 5%. Hence, the runtime overhead of SkelCL as compared to CUDA is mainly caused by the fact that SkelCL is built on top of OpenCL.

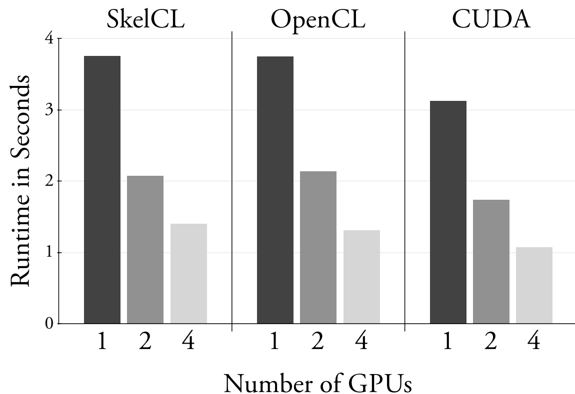
V. Enhancing SkelCL towards exascale systems

SkelCL itself is not limited to stand-alone systems with a single or multiple GPUs but is able to handle an arbitrary number of OpenCL devices. However, it relies on OpenCL which by default gives access to the devices of a stand-alone system only. Hence, to enable SkelCL to use devices from a distributed system comprising multiple stand-alone systems (*nodes*) with GPUs, a distributed OpenCL implementation is required that connects the stand-alone systems and provides access to all their devices.

We develop dOpenCL [12], a distributed implementation of the OpenCL API. When using dOpenCL, all CPUs, GPUs and accelerators of a distributed system become accessible as OpenCL devices. dOpenCL integrates the native OpenCL implementations on the nodes (we view them as *servers*) of the distributed system into a unified OpenCL implementation on a single dedicated node which can be viewed as a *client*. The client runs the OpenCL applications, while all



(a) Lines of code for host (light gray) and GPU (dark gray)



(b) Average runtime of one subset iteration

Figure 4. Program size (a) of parallel list-mode OSEM (single- and multi-GPU versions), and runtime (b) for processing one subset using SkelCL, OpenCL, and CUDA.

computations are executed on the servers' devices. For example, in our laboratory we use dOpenCL to connect our GPU system described in Section IV-C and two other GPU systems, each equipped with 1 multi-core CPU and 2 GPUs (3 servers) to a desktop PC (the client) with no OpenCL capable devices. To an OpenCL application that is executed on the desktop PC, all 8 GPUs and 3 multi-core CPUs of this distributed system appear as if they were local devices. Since dOpenCL is a drop-in replacement for any OpenCL implementation, it can be used together with SkelCL without any modifications.

In combination with dOpenCL, SkelCL allows for programming all devices of a distributed system using a single, unified high-level programming model. While dOpenCL hides the communication details of the distributed system, these systems still pose new challenges for SkelCL: devices are more likely to be heterogeneous, like in our aforementioned laboratory system which comprises several multi-core CPUs and GPUs with different characteristics. To use the heterogeneous devices efficiently, in particular to employ all devices during the complete execution of a skeleton, SkelCL should not assign evenly-sized workload to the devices. Even a simple skeleton like map requires to schedule appropriate workloads to each device. For example, compute-intensive user-defined functions are usually executed faster on GPUs than on CPUs. Hence, GPUs will be assigned larger workloads in this case. A more complex scheduling problem arises in case of the reduce skeleton (see Section III-C). Firstly, CPUs can also be involved to perform a local reduction, though they will not be able to process the same workload as GPUs. Moreover, the local reduction on each GPU should not compute a single value but an intermediate, small result vector. CPUs will be faster to perform the final reduction of these vectors than GPUs which provide poor performance when reducing only few elements. In order to decide when to use CPU rather than the GPU to perform this final reduction, a scheduling mechanism is required.

Currently, SkelCL employs a static scheduling approach based on an enhanced performance prediction approach: While the performance of pure OpenCL-based programs can only be predicted based on the low-level program code and device properties including benchmark results, SkelCL provides additional information about a program based on the known implementation of skeletons and data distributions. In SkelCL, performance prediction based on statistical code analysis and benchmarks is only used for the user-defined functions rather than the whole program code. The results of this performance prediction are completed by analytical performance models for the skeletons; this enables a more accurate prediction and leads to a more efficient scheduling of workloads on heterogeneous devices.

VI. Conclusion and Related Work

In this paper, we presented SkelCL – a high-level multi-GPU programming library. The novel contributions of SkelCL are two-fold: 1) using the vector data type and the built-in mechanism of data distributions, it considerably simplifies memory management in multi-GPU programs; 2) the high-level algorithmic skeletons are used for programming multi-GPU systems, resulting in shorter, better structured programs as compared to OpenCL and CUDA. Moreover, SkelCL extends the flexibility of skeletons with its additional arguments feature, as demonstrated on a real-world, medical imaging application.

Our case study showed that SkelCL significantly reduces programming effort in terms of lines of code, and greatly improves program structure and maintainability, while it causes less than 5% performance decrease as compared to the low-level OpenCL-implementation. We obtained similar results about the programming effort and performance for the Mandelbrot benchmark application [6].

SkelCL is a runtime, library-based approach for GPU programming, unlike compiler-based approaches, e. g., *HMPP* [13] and the *PGI Accelerator compilers* [14]. It offers

the user a consistent high-level API while still allowing the programmer to use all features of the underlying OpenCL standard.

There are other library-based approaches for high-level GPU programming.

SkePU [15] is a skeleton-based framework for multi-core CPUs and multi-GPU systems. An architecture-independent macro language is used which, however, makes architecture-specific optimizations impossible, like the use of local memory in OpenCL. SkelCL avoids this drawback by building on top of OpenCL. SkePU provides memory abstractions similar to SkelCL, but does not support different data distributions on multi-GPU systems as in SkelCL: vectors are always distributed to all available devices, with no possibility of data exchanges between devices. Therefore, our list-mode OSEM application which heavily relies on multiple data exchanges between devices cannot be implemented efficiently using SkePU.

Thrust [16] provides an interface similar to the C++ Standard Template Library (STL). For data management, two distinct dynamic containers, a `host_vector` and a `device_vector`, can be used like STL vectors for managing host and device memory respectively. In addition, Thrust offers common parallel algorithms, including searching, sorting, reductions, and transformations. Thrust is based on CUDA, therefore, restricting the user to NVIDIA GPUs.

GPUSs [17] is an implementation of the Star Superscalar model for multi-GPU systems. While SkelCL is focused on data parallelism, GPUSs provides simple task parallelism; annotations are used for data transfers between host and GPU. SkelCL offers a higher level of memory abstraction: communication is specified implicitly by a distribution scheme instead of individual data transfers.

Rabhi and Gorlatch [5] present different approaches of skeletal programming for parallel as well as distributed systems. González-Vélez and Leyton [18] provide an overview of available skeleton frameworks.

Acknowledgement

The authors would like to thank NVIDIA for their hardware donation, as well as Thomas Kösters and Klaus Schäfers (European Institute for Molecular Imaging, WWU Münster) for providing the reconstruction software EMRECON [10] and the quadHIDAC PET data used in our application case study.

References

[1] A. Munshi, *The OpenCL Specification*, 2011, version 1.2.

[2] *NVIDIA CUDA API Reference Manual*, 2012, version 4.1.

[3] D. B. Kirk and H. W. W., *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufman, 2010.

[4] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman and MIT Press, 1989.

[5] F. A. Rabhi and S. Gorlatch, Eds., *Patterns and skeletons for parallel and distributed computing*. Springer-Verlag, 2003.

[6] M. Steuwer, P. Kegel, and S. Gorlatch, "SkelCL – A portable skeleton library for high-level gpu programming," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2011, pp. 1176–1182.

[7] D. Gregor and J. Järvi, "Variadic templates for C++0x," *Journal of Object Technology*, vol. 7, no. 2, pp. 31–51, February 2008.

[8] *ISO/IEC 14882:2011 Information technology – Programming languages – C++*, ISO/IEC JTC1/SC22/WG21 – The C++ Standards Committee, 2011.

[9] A. J. Reader, K. Erlandsson, M. A. Flower, and R. J. Ott, "Fast accurate iterative reconstruction for low-statistics positron volume imaging," *Physics in Medicine and Biology*, vol. 43, no. 4, pp. 823–834, April 1998.

[10] T. Kösters, K. Schäfers, and F. Wübbeling, "EMRECON: An expectation maximization based image reconstruction framework for emission tomography data," in *NSS/MIC Conference Record, IEEE*, 2011. [Online]. Available: http://emrecon.uni-muenster.de

[11] J. P. Jones, W. F. Jones, and F. Kehren, "SPMD cluster-based parallel 3-D OSEM," *IEEE Transactions on Nuclear Science*, vol. 50, no. 5, pp. 1498–1502, 2003.

[12] P. Kegel, M. Steuwer, and S. Gorlatch, "dOpenCL: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems," in *2012 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2012.

[13] R. Dolbeau, F. Bihan, and B. F., "HMPP: A Hybrid Multi-core Parallel Programming Environment," in *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[14] T. P. Group, *PGI Accelerator Programming Model for Fortran & C*, 2010.

[15] 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.

[16] J. Hoberock and N. Bell, "Thrust: A Parallel Template Library," 2009, version 1.1. [Online]. Available: http://www.meganewtons.com

[17] E. Ayguadé, R. M. Badia, F. D. Igual *et al.*, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Euro-Par 2009 Parallel Processing*, ser. Lecture Notes in Computer Science, H. J. Sips, D. H. J. Epema, and H. Lin, Eds., vol. 5704. Springer, 2009, pp. 851–862.

[18] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Software: Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.