SkelCL – A Portable Skeleton Library for High-Level GPU Programming

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*—While CUDA and OpenCL made general-purpose programming for Graphics Processing Units (GPU) popular, using these programming approaches remains complex and error-prone because they lack high-level abstractions. The especially challenging systems with multiple GPU are not addressed at all by these low-level programming models. We propose SkelCL – a library providing so-called algorithmic skeletons that capture recurring patterns of parallel computation and communication, together with an abstract vector data type and constructs for specifying data distribution. We demonstrate that SkelCL greatly simplifies programming GPU systems. We report the competitive performance results of SkelCL using both a simple Mandelbrot set computation and an industrial-strength medical imaging application. Because the library is implemented using OpenCL, it is portable across GPU hardware of different vendors.**

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

I. Introduction

Modern *Graphics Processing Units* (GPUs) provide large numbers of processing units and a high memory bandwidth. To enable *General-Purpose Computation on GPU* (GPGPU), new programming models have been introduced, the two most popular approaches being CUDA and OpenCL [1], [2]. While both programming models are similar, CUDA is a proprietary software developed by NVIDIA, whereas OpenCL is an open industry standard.

Programming GPUs with OpenCL (and CUDA as well) still remains a difficult task because it is a low-level programming model: Data has to be transferred explicitly from the system's main memory (accessible by the CPU) to the GPU's memory and back. Moreover, memory allocation and deallocation also has to be explicitly controlled by the programmer. All this results in a lot of low-level boilerplate code in GPU programs.

In this paper, we present *SkelCL* — a library which introduces an easy-to-use, high-level approach for GPU programming. SkelCL provides an abstract vector data type to perform data exchange between CPU and GPU implicitly. Pre-implemented communication and computation patterns (a.k.a. *algorithmic skeletons*) based on this data type hide boilerplate code from the programmer inside the skeletons' implementations. Because SkelCL is based on OpenCL, it is not bound to a specific hardware and can be executed on any OpenCL-capable device.

The remainder of this paper is organized as follows: A brief introduction to GPU programming using OpenCL is given in Section II. In Section III, we describe the design and implementation of SkelCL, in particular the abstract vector data type and algorithmic skeletons. In Section IV, we present runtime experiments for a simple Mandelbrot set computation, as well as for a real-world application for medical imaging. Section V reviews related work, and Section VI concludes the paper.

II. GPU programming using OpenCL

The OpenCL standard [2] can be used for programming any OpenCL-capable device. These devices embrace most modern GPUs and other accelerators, e. g., the Cell BE, as well as standard multi-core CPUs.

OpenCL distinguishes between a *host* system, usually containing one or several CPUs, and *devices* that are integrated into the host system. An OpenCL device logically consists of one or more *compute units* (CUs) that are divided into one or more *processing elements* (PEs). All computation on the device is performed in the PEs. OpenCL applications run on the host and call *kernel functions* which are executed simultaneously by multiple PEs on one or more devices. A single instance of a kernel function is called a *work-item* and can be identified by its *global ID*. Every work-item executes the same code, but the execution can vary per work-item due to branching according to the global ID. Work-items are organized in *work-groups*. When a kernel function is started, the host code specifies how many work-items are launched and how many work-items form a work-group. All work-items in one work-group are executed on the same CU. Therefore, the size of a work-group can have a significant effect on the runtime performance.

In OpenCL, host and device have separate memories. Thus, functions are provided to transfer data from the host's to the device's memory (*upload*) and back (*download*). Memory areas have to be allocated on the device before data can be accessed by it and deallocated thereafter.

For portability across a wide range of devices, kernel functions are compiled at runtime. The host program passes the kernel's source code as a plain string to the OpenCL driver to create executable binary code. This is different compared to CUDA which provides a special compiler `nvcc` to compile the device code and the host code.

Creating applications for multi-GPU systems introduces new challenges, like partitioning the application appropriately and, explicitly implementing data transfer between devices [3]. The host application must coordinate and perform synchronization and data exchange explicitly. The source code for performing such exchanges further increases the amount of boilerplate code.

In the following, we describe how our SkelCL library addresses these problems of GPGPU programming.

III. SkelCL: An OpenCL-based skeleton library

SkelCL is based on data-parallel algorithmic skeletons [4]. A skeleton can be formally viewed as a higher-order function: apart from plain values it also takes one or more user-defined functions as arguments. These functions customize a skeleton to perform a particular calculation in parallel.

SkelCL provides a set of basic skeletons. Two well-known examples are the `Zip` skeleton combining two vectors element-wise, and the `Reduce` skeleton combining all elements of a vector using a binary operation (see Section III-B). Listing 1 shows how a dot product of two vectors is implemented in SkelCL using these two skeletons. Here, the `Zip` skeleton is customized by usual multiplication, and the `Reduce` skeleton is customized by usual addition.

For comparison, an OpenCL-based implementation of a dot product computation provided by NVIDIA requires approximately 68 lines of code (kernel function: 9 lines, host program: 59 lines) [5].

The implementation principles of the SkelCL library are as follows. SkelCL generates OpenCL code (*kernel functions*) from skeletons which is then compiled by OpenCL at runtime. User-defined customizing functions passed to the skeletons are merged with pre-implemented skeleton code during code generation. Since OpenCL is not able to pass function pointers to GPU functions, user-defined functions are passed as strings in SkelCL.

A. Abstract vector class and memory management

SkelCL offers the `Vector` class providing a unified abstraction for a contiguous memory area that is accessible by both, CPU and GPU. Internally, a vector comprises pointers to corresponding areas of main memory (accessible by the CPU) and GPU memory. Upon creation of a vector on the host system, memory is also allocated on the GPU accordingly. Data transfer between these corresponding memory areas is performed implicitly if the CPU accesses the vector that previously has been modified by the GPU or vice versa. Thus, the `Vector` class shields the user from low-level memory operations like allocation (on the GPU) and data transfer between CPU and GPU memory.

```
int main (int argc, char const* argv[]) {
    SkelCL::init(); /* initialize SkelCL */

    /* create skeletons */
    SkelCL::Reduce<float> sum (
       "float sum (float x,float y){return x+y;}");
    SkelCL::Zip<float>    mult(
       "float mult(float x,float y){return x*y;}");

    /* allocate and initialize host arrays */
    float *a_ptr = new float[ARRAY_SIZE];
    float *b_ptr = new float[ARRAY_SIZE];
    fillArray(a_ptr, ARRAY_SIZE);
    fillArray(b_ptr, ARRAY_SIZE);

    /* create input vectors */
    SkelCL::Vector<float> A(a_ptr, ARRAY_SIZE);
    SkelCL::Vector<float> B(b_ptr, ARRAY_SIZE);

    /* execute skeletons */
    SkelCL::Scalar<float> C = sum( mult( A, B ) );

    /* fetch result */
    float c = C.getValue();

    /* clean up */
    delete[] a_ptr;
    delete[] b_ptr;
}
```

Listing 1. SkelCL program computing the dot product of two vectors. Arrays `a_ptr` and `b_ptr` initialize the vectors.

In SkelCL, `Vector` is a class template. It implements a generic container class that is capable of storing data items of any primitive C/C++ data type (e. g., `int`), as well as user-defined data structures (*structs*).

All skeletons accept vectors as their input and output. Before execution, a skeleton's implementation ensures that all input vectors' data is available on the GPU. This might result in implicit data transfers from the main memory to GPU memory. The data of the output vector is not copied back to the main memory but rather resides in the GPU memory. Before every data transfer, the vector implementation checks whether the data transfer is necessary; only then the data is actually transferred. Hence, if an output vector is used as the input to another skeleton, no further data transfer is performed. This *lazy copying* minimizes costly data transfers between host and device.

B. Skeletons in SkelCL

Skeletons are higher-order functions because they take so-called *customizing functions* as parametes. In SkelCL, skeletons expect the customizing function to be a plain string containing the function's source code. This is merged with the skeleton's own source code to generate source code for an OpenCL kernel. After compilation, the kernel function is ready for execution. Compiling the source code every time from source is a time-consuming task, taking up to several hundreds of milliseconds. For a small kernel, this can be a huge overhead. Therefore, SkelCL saves already compiled kernels on disk. They can be loaded later if the same kernel is used again. For our applications (presented in Section IV), we observed that loading kernels from disk is at least five times faster than building them from source.

SkelCL currently provides four basic skeletons: `Map`, `Zip`, `Reduce`, and `Scan`. Each skeleton consumes vectors as input and produces vectors as output. A skeleton is called by using the name of the skeleton as a function and passing the appropriate number of arguments to it. This behavior is implemented using the C++ operator overloading feature.

The `Map` skeleton takes a unary function and applies it to all elements of a given input vector. The result is a vector of the same length as the input vector. For a given function f and vector $v = [x_0, \ldots, x_{n-1}]$ of length n, the mathematical definition of `Map` is given by

$$\mathit{map}\, f\, [x_0, \ldots, x_{n-1}] = [f(x_0), \ldots, f(x_{n-1})] \tag{1}$$

The customizing function f takes each element of the input vector and returns the corresponding element of the output vector. The output vector can be computed in parallel, because the calculations are independent of each other.

The `Zip` skeleton takes a customizing binary operator and applies it to corresponding elements of two equally sized input vectors; the result is a vector of the same size. For a given operator $\oplus$ and two vectors $v_0 = [x_0, \ldots, x_{n-1}]$ and $v_1 = [y_0, \ldots, y_{n-1}]$ of length n, `Zip` is defined as

$$\begin{aligned} \mathit{zip} \oplus [x_0, \ldots, x_{n-1}], [y_0, \ldots, y_{n-1}] \\ = [x_0 \oplus y_0, \ldots, x_{n-1} \oplus y_{n-1}] \end{aligned} \tag{2}$$

Thus, it is a generalized dyadic form of `Map`. By chaining `Zip` skeletons, variadic forms of `Map` can be implemented. `Zip` is parallelizable in the same manner as `Map`.

The `Reduce` skeleton uses a binary operator to combine all elements of the input vector and returns a scalar result. For a given operator $\oplus$ and a vector $v = [x_0, \ldots, x_{n-1}]$ of length n, `Reduce` is defined as

$$\mathit{reduce} \oplus [x_0, \ldots, x_{n-1}] = x_0 \oplus \ldots \oplus x_{n-1} \tag{3}$$

The parallelization approach for `Reduce` depends on whether the binary operator is commutative and/or associative. SkelCL requires the operator to be associative, such that it can be applied to arbitrarily sized subranges of the input vector in parallel. The final result is obtained by recursively combining the intermediate results for the subranges. To improve the performance, SkelCL saves the intermediate results in the device's fast local memory.

The `Scan` skeleton applies a customizing binary operator $\oplus$ to the elements of an input vector. It returns a vector of the same size as the input vector, such that element i is calculated by combining elements from 0 to $i - 1$ of the input vector using $\oplus$. For a given binary operator $\oplus$ and a vector $v = [x_0, \ldots, x_{n-1}]$ of length n `Scan` is defined as

$$\begin{aligned} \mathit{scan} \oplus [x_0, \ldots, x_{n-1}] \\ = [id, x_0, x_0 \oplus x_1, \ldots, x_0 \oplus \ldots \oplus x_{n-2}] \end{aligned} \tag{4}$$

```
Map<float> mult_num("float f(float input, float
    number) { return input * number }");

Arguments arguments;
arguments.push(5);

mult_num(input, arguments);
```

Listing 2. Passing additional arguments to a `Map` skeleton.

where id is the identity element of the operator $\oplus$.

The implementation of `Scan` provided in SkelCL is a modified version of [6]. It is highly optimized and makes heavy use of local memory, as well as it tries to avoid memory bank conflicts which can occur when multiple work-items access the same memory location. Possible applications of the `Scan` skeleton are stream compaction or a radix sort implementation [6].

C. Passing Additional Arguments to Skeletons

In general, a skeleton's definition dictates how many input arguments can be passed to the skeleton. The `Map` skeleton, for example, specifies that the provided function has to be a unary function, such that only one input vector can be passed to the skeleton. However, not all algorithms easily fit into this strict pattern.

SkelCL allows the user to pass an arbitrary number of arguments to the function called inside of a skeleton: first, the function definition must be changed, such that it expects additional arguments; second, the additional arguments have to be passed to the skeleton upon execution. A simple example is presented in Listing 2. The function definition for the `Map` skeleton in the listing takes two arguments instead of one, which would be common for `Map`. In this example, an additional increment value is passed to the function. Thus, the `Map` skeleton can now be used for adding an arbitrary increment to all elements of an input vector, instead of a fixed increment. The additional argument is packaged into an `Arguments` object that is passed to the skeleton. The implementation ensures passing the argument to all kernel instances called during execution.

Arbitrary types can be passed as arguments; a pointer and the size of the type has to be provided. It is particularly easy to pass vectors as arguments because no information about the size has to be provided. The arguments will be passed to the skeleton in the same order in which they are added to the `Arguments` object. Hence, their order has to resemble the order of parameters in the function definition.

D. Towards multiple GPUs

Using multiple GPUs introduces additional challenges in the programming process. Neither CUDA nor OpenCL offer any particular support to program multiple devices. Basically, every GPU is available to OpenCL as a separate device. To support multi-GPU systems, SkelCL can distribute a `Vector` to all available devices: the `Vector` is either completely copied to every device, or evenly divided into one part per device. When a `Vector` that is split across multiple devices is passed as an input to a skeleton, it is executed in cooperation by multiple devices: each device processes a different part of the input vector's data in parallel. This requires a skeleton to synchronize computations and exchange data between multiple devices.

The user may either rely on a skeleton-specific default distribution, or can control a `Vector`'s distribution explicitly. It is also possible to redistribute a `Vector`, i. e., change its distribution; data exchange between multiple devices is performed automatically by SkelCL. We will provide more detail on multi-GPU programming using SkelCL in our next papers.

IV. Application studies and experiments

We developed implementations of two application case studies using the SkelCL library: 1) the calculation of a Mandelbrot fractal as a simple benchmark application, and, 2) a real-world algorithm for medical image reconstruction. In this section, both implementations are compared to similar implementations in CUDA and OpenCL regarding programming effort and runtime performance.

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

A. Case study: Mandelbrot set

The Mandelbrot [7] set are all complex numbers $c \in \mathbb{C}$ for which the sequence

$$z_{i+1} = z_i^2 + c, i \in \mathbb{N} \tag{5}$$

starting with $z_0 = 0$ does not escape to infinity. If drawn as an image with each pixel representing a complex number, the boundary of the Mandelbrot set forms a fractal. The calculation of such an image is a time-consuming task, because the sequence given by (5) has to be calculated for every pixel. If this sequence does not cross a given threshold for a given number of steps, it is presumed that the sequence will converge. The respective pixel is thus taken as a member of the Mandelbrot set, and it is painted black. Other pixels outside are assigned a color that corresponds to the number of iterations of the sequence given by (5). Computing a Mandelbrot fractal is easily parallelizable, as all pixels of the fractal can be computed simultaneously.

1) Programming effort: We created three similar parallel implementations for computing a Mandelbrot fractal using CUDA, OpenCL, and SkeCL.

CUDA and SkelCL require a single line of code for initialization in the host code, whereas OpenCL requires a lengthy creation and initialization of different data structures which take about 20 lines of code.

The host code differs significantly between all implementations. In CUDA, the kernel is called like an ordinary function. A proprietary syntax is used to specify the size of work-groups executing the kernel. With OpenCL, several API functions are called to load and build the kernel, pass arguments to it and to launch it using a specified work-group size. In SkelCL, the kernel passed to a newly created instance of a `Map` skeleton (see Section III-B). A `Vector` of complex numbers, each of which is represented by a pixel of the Mandelbrot fractal, is passed to the `Map` skeleton upon execution. Specifying the work-group size is mandatory in CUDA and OpenCL, whereas this is optional in SkelCL. However, it is sometimes reasonable to also hand-optimize the work-group size in SkelCL, since it can have a considerable impact on performance.

Program size: The OpenCL-based implementation has 118 lines of code (kernel: 28 lines, host program: 90 lines) and is thus more than twice as long as the CUDA and SkelCL versions with 49 lines (28, 21) and 57 lines (26, 31), respectively (see Figure 1). The length of the CUDA- and SkelCL-based implementations differs by only a few lines.

Kernel size: The kernel function is similar in all implementations: it takes a pixel's position (i.e., a complex number) as input, performs the iterative calculation for this pixel, and returns the pixel's color. However, while the input positions have to be given explicitly when using the `Map` skeleton in SkelCL, no positions are passed to the kernel in the CUDA- and OpenCL-based implementations.

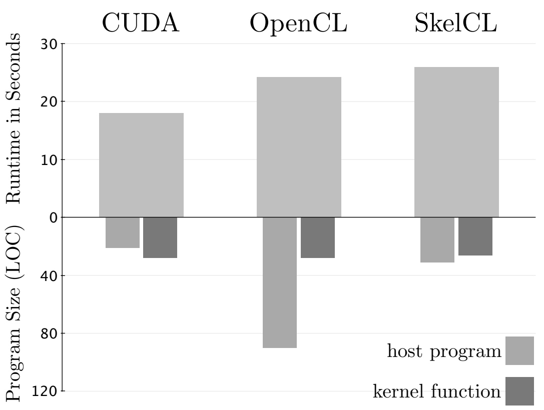

Figure 1. Runtime and program size of the Mandelbrot application.

2) Performance experiments: We tested our implementations on a single GPU of our test system to compute a Mandelbrot fractal of size 4096×3072 pixels. In CUDA and OpenCL, work-groups of 16×16 are used; SkelCL uses its default work-group size of 256. The results are shown in Figure 1. As compared to the runtime of the SkelCL-based implementation (26 seconds), the implementation based on OpenCL (25 seconds) and CUDA (18 seconds) are faster by 4% or 31%, respectively. Since SkelCL is built on top of OpenCL, the performance difference of SkelCL and OpenCL can be regarded as the overhead introduced by SkelCL. Previous work [8] also reported that CUDA was usually faster than OpenCL, which also explains the higher performance of the implementation based on CUDA. The Mandelbrot application demonstrates that SkelCL introduces a tolerable overhead of less than 5% as compared to OpenCL. A clear benefit of this overhead is the reduced programming effort required by the SkelCL programm.

B. List-mode OSEM

List-Mode Ordered Subset Expectation Maximization (list-mode OSEM) is a time-intensive, production-quality algorithm from a real-world application in medical image reconstruction. It is used to reconstruct three-dimensional images from huge sets of so-called *events* recorded in *positron emission tomography* (PET). Each event represents a *line of response* (LOR) which intersects the scanned volume. A simplified sequential implementation of list-mode OSEM is shown in Listing 3. The algorithm splits the events into *subsets* which are processed iteratively: All LORs of a subset's events and their corresponding intersection *paths* are computed and merged into an *error image* which is merged with the reconstruction image, thus refining a *reconstruction image* in each iteration.

In a parallel implementation of list-mode OSEM, the loops for calculating the error image (c) and for updating the reconstruction image (f) can be parallelized.

```
for (l = 0; l < num_subsets; l++) {
    /* read subset from file */
    events = read_events();
    /* compute error image c */
    for (i = 0; i < num_events; i++) {
        /* compute path of LOR */
        path = compute_path(events[i]);
        /* compute error */
        for (fp = 0, m = 0; m<path_len; m++)
            fp += f[path[m].coord] * path[m].len;
        /* add path to error image */
        for (m = 0; m<path_len; m++)
            c[path[m].coord] += path[m].len / fp;
    }
    /* update reconstruction image f */
    for (j = 0; j < image_size; j++)
        if (c[j] > 0.0) f[j] *= c[j];    }
```

Listing 3. Sequential implementation of list-mode OSEM.

```
for (l = 0; l < num_subsets; l++) {
    // read events from file
    Vector<Event> events(read_events());

    // distribute events to devices
    events.setDistribution(
        Distribution::block);
    // copy reconstruction (f) and error image (c)
         to all devices
    f.setDistribution(Distribution::copy);
    c.setDistribution(Distribution::copy);

    // prepare arguments of error image
       computation
    SkelCL::Arguments arguments;
    arguments.push(events);
    arguments.push(events.size());
    arguments.push(paths); // memory for paths
    arguments.push(f);
    arguments.push(c);

    // compute error image (map skeleton)
    compute_c(index, arguments);

    // signal modification of error image
    c.dataOnDevicesModified();
    // distribute reconstruction image to  all
       devices
    f.setDistribution(Distribution::block);
    // reduce (element-wise add) all copies of
       error image; re-distribute to all devices
       after reduction
    c.setDistribution(Distribution::block, add);

    // update reconstruction image (zip skeleton)
    update(f, c, f);    }
```

Listing 4. Implementation of list-mode OSEM in SkelCL.

1) Programming effort: We develop implementations of list-mode OSEM using OpenCL and SkelCL; a CUDA-based implementation using multiple GPUs has already been implemented [3]. Both, CUDA and OpenCL, require us to add a considerable amount of boilerplate code for running a kernel on multiple GPUs, in particular for uploading and downloading data to and from the GPUs. In CUDA, we have to create one CPU thread for each device to be managed. This introduces the additional challenge of multi-threaded programming, including the need of thread synchronization.

With SkelCL, we use the `Vector` class and the `Map` and `Zip` skeletons to implement list-mode OSEM (see Listing 4). The events of a subset, as well as the error image and the reconstruction image are stored in a SkelCL `Vector`. Thus, we can easily distribute subsets across all GPUs and copy both images to all devices. We use a `Map` skeleton to implement the computation of the error image. However, we must not compute too many paths in parallel to avoid excessive memory consumption. Therefore, the input of the `Map` skeleton is not a subset, but rather a vector of 512 indices. These indices refer to disjoint sub-subsets of events, each of which is processed within a single kernel instance on the GPUs. For each event, this kernel performs the same steps as the first inner loop in the sequential implementation. The reconstruction and the error image, as well as the events, are passed to the `Map` skeleton as additional arguments. The skeleton produces no result, but updates the error image by side-effect. Therefore, the error image has to be marked as "modified on the device" after executing the `Map` skeleton.

So far, separate copies of the error image have been used on all GPUs. To obtain the final error image, we have to merge all copies into a single image. Afterwards, the final error image and the reconstruction image are distributed across all GPUs, such that each device processes a part of these images. In SkelCL, the aforementioned data movement is easily achieved by changing the kind of distribution of the vectors that contain this data. A `Zip` skeleton is used to implement the update of the reconstruction image, taking the distributed images as input. The kernel function of this skeleton resembles the body of the second inner loop of the sequential implementation.

The parallelization using SkelCL is quite similar to our CUDA- and OpenCL-based implementations. However, when using SkelCL's vector data type, we avoid additional programming effort to implement data transfer between host and GPU or between multiple GPUs, and we obtain a multi-GPU-ready implementation of list-mode OSEM for free. The SkelCL-based implementation is the shortest with 232 lines of code (kernel function: 200 lines, host program: 32 lines). The CUDA- and OpenCL-based implementations are considerably longer with 329 (199, 130), or even 436 lines of code (193, 243), respectively (see Figure 2).

2) Performance experiments: We tested our implementations of list-mode OSEM using a typical data set of about 10^7 events for $150 \times 150 \times 280$ PET image. The data set is split into 10 equally sized subsets. We measured the average runtime of processing all subsets. Figure 2 shows the runtime of our three implementations of list-mode OSEM using one, two, and four GPUs.

Running on a single GPU, the CUDA-based implementation (3.03 seconds) outperforms the ones based on OpenCL and SkelCL (3.66 seconds each) by about 20%. These relative performane differences also hold for using two GPUs, with only negligible overhead of SkelCL compared to OpenCL. With four GPUs, the CUDA-based implementation again is faster than the SkelCL- (23%) and OpenCL-based (17%) implementations. SkelCL provides a speedup of 3.1, while OpenCL and CUDA provide speedups of 3.24 and 3.15 respectively. On four GPUs the SkelCL code runs 2.56 times faster than the CUDA on one GPU.

The SkelCL-based implementation only introduces a moderate overhead of less than 5% as compared to OpenCL. Since the OpenCL-based implementation requires a lot of low-level boilerplate code (over 100 lines of code only for initialization), the SkelCL-based implementation clearly provides a higher level of programming. Especially, the additional argument feature and the data distributions are crucial for this application as it cannot be implemented efficiently without these two features. In conclusion, this example shows that SkelCL is suitable for implementing a real-world application and provides performance close to a native OpenCL implementation.

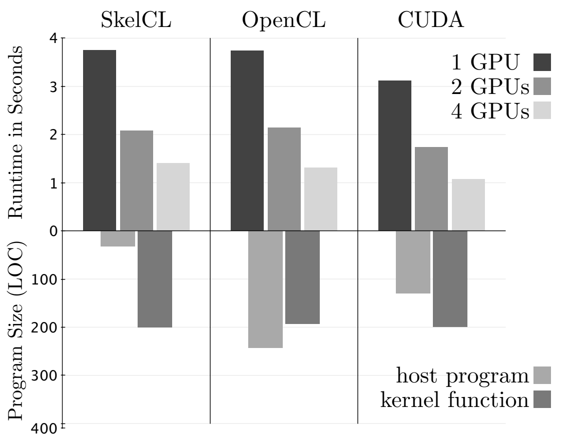

Figure 2. Runtime and program size of parallel list-mode OSEM using CUDA, OpenCL, and SkelCL.

V. Related work

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

SkePU [9] uses container classes and algorithmic skeletons to ease multi-GPU computing. Although SkePU and SkelCL have been developed independently, both projects share some concepts: SkePU provides a vector class similar to SkelCL's `Vector` class, but unlike SkelCL it does not support different kinds of data distribution on multi-GPU systems. SkePU and SkelCL both provide a map and a reduce skeleton. However, SkelCL additionally provides the `Zip` and `Scan` skeleton, while SkePU supports two additional variants of the map skeleton. Unlike SkelCL, which allows for an arbitrary number of arguments, in SkePU the user-defined functions are restricted to a fixed skeleton-specific number of arguments. Currently, SkePU is the only project other than SkelCL that supports data-parallel computations on multi-GPU systems. SkelCL provides a more flexible memory management than SkePU, as data transfers can be expressed by changing data distribution settings. Only this flexibility provides the best performance for our second case study (Section IV-B) and similar applications. Both projects differ significantly in the way how functions are passed to skeletons. While functions are defined as plain strings in SkelCL, SkePU uses a macro language, which brings some serious drawbacks. For example, it is not possible to call mathematical functions like sin or cos inside a function generated by a SkelPU macro, because these functions are either named differently in all three target programming models or might even be missing entirely.

The same holds for functions and keywords related to performance tuning, e. g., the use of local memory. SkelCL does not suffer from these drawbacks because it relies on OpenCL and thus can be executed on a variety of devices.

CUDPP [10] is a C++ library based on CUDA. It provides data-parallel algorithm primitives similar to skeletons. These primitives can be configured using only a predefined set of operations, whereas skeletons in SkelCL are true higher-order functions, which accept any user-defined function. CUDPP does not simplify data management, because data still has to be exchanged between CPU and GPU explicitly. There is also no support for multi-GPU applications.

Thrust [11] is an open-source library by NVIDIA. It provides two vector types similar to the vector type of the C++ Standard Template Library. While these types refer to vectors stored in CPU or GPU memory, respectively, SkelCL's vector data type provides a unified abstraction for CPU and GPU memory. Thrust also contains data-parallel implementations of higher-order functions, similiar to SkelCL's skeletons. SkelCL adopts several of Thrust's ideas, but it is not limited to CUDA-capable devices and supports multiple GPUs.

Unlike SkelCL, *PGI Acccelerator* [12] and *HMPP* [13] are compiler-based approaches to GPU programming, similar to the popular OpenMP [14]. The programmer uses compiler directives to mark regions of code to be executed on a GPU. A compiler generates executable code for the GPU, based on the used directives. Although source code for low-level details like memory allocation or data exchange is generated by the compiler, these operations still have to be specified explicitly by the programmer using suitable compiler directives. We consider these approaches low-level, as they do not perform data transfer automatically to shield the programmer from low-level details.

VI. Conclusion

We developed and implemented SkelCL – an OpenCL-based skeleton library for high-level GPU programming, based on an abstract data type and algorithmic skeletons. Currently, it provides a vector data type and four basic skeletons (`Map`, `Zip`, `Reduce`, `Scan`). SkelCL shields the user from the low-level details of GPU programming. Data transfer and synchronization are performed implicitly.

Our application examples show that SkelCL provides competitive performance and scalability on real-world applications as compared with CUDA and OpenCL. While SkelCL adds a minor performance overhead, it significantly reduces the programming effort, since much of the boilerplate code required in CUDA or OpenCL is replaced by shorter and more intuitive high-level constructs.

Acknowledgment

We would like to thank NVIDIA for providing hardware for developing and testing SkelCL.

References

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

[2] A. Munshi, *The OpenCL Specification*, Beaverton, OR, 2010, version 1.1, Document Revision: 33.

[3] M. Schellmann, J. Vörding, and S. Gorlatch, "Systematic parallelization of medical image reconstruction for graphics hardware," in *Euro-Par 2008 Parallel Processing*, ser. LNCS, vol. 5168. Springer, 2008, pp. 811–821.

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

[5] "NVIDIA CUDA SDK code samples," February 2010, version 3.0. [Online]. Available: http://developer.nvidia.com/object/cuda_3_0_downloads.html

[6] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, August 2007, ch. 39, pp. 851–876.

[7] B. B. Mandelbrot, "Fractal aspects of the iteration of $z \mapsto \lambda z(1 - z)$ for complex λ and z," *Annals of the New York Academy of Sciences*, vol. 357, pp. 249–259, December 1980.

[8] J. Kong, M. Dimitrov, Y. Yang *et al.*, "Accelerating MATLAB image processing toolbox functions on GPUs," in *GPGPU '10: Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010.

[9] J. Enmyren and C. Kessler, "SkePU: A multi-backend skeleton programming library for multi-gpu systems." in *Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications*, 2010.

[10] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Graphics Hardware 2007*.

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

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

[13] S. Bihan, G. Moulard, R. Dolbeau *et al.*, "Directive-based heterogeneous programming a gpu-accelerated rtm use case," in *Proceedings of the 7th International Conference on Computing, Communications and Control Technologies*, 2009.

[14] *OpenMP Application Program Interface*, OpenMP Architecture Review Board, 2008, version 3.0. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf