Towards High-Level Programming for Systems with Many Cores

Sergei Gorlatch and Michel Steuwer

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

Abstract. Application development for modern high-performance systems with many cores, i.e., comprising multiple Graphics Processing Units (GPUs) and multi-core CPUs, currently exploits low-level programming approaches like CUDA and OpenCL, which leads to complex, lengthy and error-prone programs. In this paper, we advocate a high-level programming approach for such systems, which relies on the following two main principles: a) the model is based on the current OpenCL standard, such that programs remain portable across various many-core systems, independently of the vendor, and all low-level code optimizations can be applied; b) the model extends OpenCL with three high-level features which simplify many-core programming and are automatically translated by the system into OpenCL code. The high-level features of our programming model are as follows: 1) memory management is simplified and automated using parallel container data types (vectors and matrices); 2) a data (re)distribution mechanism supports data partitioning and generates automatic data movements between multiple GPUs; 3) computations are precisely and concisely expressed using parallel algorithmic patterns (skeletons). The well-defined skeletons allow for semantics-preserving transformations of SkelCL programs which can be applied in the process of program development, as well as in the compilation and optimization phase. We demonstrate how our programming model and its implementation are used to express several parallel applications, and we report first experimental results on evaluating our approach in terms of program size and target performance.

1 Introduction

Modern computer systems become increasingly many-core as they comprise, in addition to multi-core CPUs, also *Graphics Processing Units* (GPUs), Intel Xeon Phi Coprocessors, FPGA, etc. with hundreds and thousands of cores.

The application programming for many-core systems is currently quite complex and error-prone. As the most prominent example, GPUs are programmed using explicit, low-level programming approaches CUDA [17] and OpenCL [13]. Even on a system with one GPU, the programmer is required to explicitly manage GPU's memory, including memory (de)allocations and data transfers, and also to explicitly describe parallelism in the application.

The final publication is available at Springer via
http://dx.doi.org/10.1007/978-3-662-46823-4_10.

For multi-GPU systems, CUDA and OpenCL make programs even more complex, as codes must explicitly implement data exchanges between the GPUs, as well as disjoint management of individual GPU's memories, with low-level pointer arithmetics and offset calculations.

In this paper, we address these main challenges of the contemporary many-core programming, and we present the *SkelCL (Skeleton Computing Language)* – our high-level approach to program many-core systems with multiple GPUs.

The SkelCL programming model extends the standard OpenCL approach with the following high-level mechanisms:

1) *parallel container data types*: data containers (e.g., vectors and matrices) that are automatically managed on GPUs' memories in the system;
2) *data (re)distributions*: a mechanism for specifying suitable data distributions among the GPUs in the application program and automatic runtime data re-distribution when necessary;
3) *parallel skeletons*: pre-implemented high-level patterns of parallel computation and communication, customizable to express application-specific parallelism and combinable to larger application codes.

The high-level, formally defined programming model of SkelCL allows for semantics-preserving transformations of programs for many-cores. Transformations can be used in the process of high-level program development and in optimizing the implementation of SkelCL programs.

The remainder of the paper is structured as follows. In Section 2 we describe our high-level programming model and we illustrate its use for several example applications in Section 3. Section 4 discusses using transformations for optimizing skeleton programs, and Section 5 presents our current SkelCL library implementation. Section 6 reports experimental evaluation of our approach regarding program size and performance. We compare to related work and conclude in Section 7.

2 SkelCL: Programming Model and Library

In this section, we first explain our main design principles for a high-level programming model. We present the high-level features of the SkelCL model and illustrate them using well-known use cases of parallel algorithms.

2.1 SkeCL as Extension of OpenCL

We develop our SkelCL [21] programming model as an extension of the standard OpenCL programming model [13], which is currently the most popular approach to programming heterogeneous systems with various accelerators, independently of the vendor. At the same time, SkelCL aims at overcoming the problematic aspects of OpenCL which make its use complicated and error-prone for the application developer.

In developing SkelCL, we follow two major principles:

First, we take the existing OpenCL standard as the basis for our approach. SkelCL inherits all advantageous properties of OpenCL, including its portability across different heterogeneous parallel systems and low-lev code optimization possibilities. Moreover, this allows the application developers to remain in the familiar programming environment, develop portable programs for various many-core systems of different vendors, and apply the proven best practices of OpenCL program development and optimization.

Second, our model extends OpenCL gradually: the program developer can either design the program from the initial algorithm at a high level of abstraction while some low-level parts are expressed in OpenCL, or the developer can decide to start from an existing OpenCL program and to replace some parts of the program in a step-by-step manner by corresponding high-level constructs. In both cases, the main benefit of using SkelCL is a simplified software development, which results in a shorter, better structured high-level code and, therefore, the overall maintainability is greatly improved.

SkelCL is designed to be fully compatible with OpenCL: arbitrary parts of a SkelCL code can be written or rewritten in OpenCL, without influencing program's correctness. While the main OpenCL program is executed sequentially on the CPU – called the *host* – time-intensive computations are offloaded to parallel processors – called *devices.* In this paper, we focus on systems comprising multiple GPUs as accelerators, therefore, we use the terms CPU and GPU, rather than more general OpenCL terms host and device.

2.2 Parallel Container Data Types

The first aspect of traditional OpenCL programming which complicates application development is that the programmer is required to explicitly manage GPU's memory (including memory (de)allocations, and data transfers to/from the system's main memory). In our high-level programming model, we aim at making collections of data (containers) automatically accessible to all GPUs in the target system and at providing an easy-to-use interface for the application developer. SkelCL provides the application developer with 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, and the *matrix* provides a convenient access to a two-dimensional memory area.

In a SkelCL program, a vector object is created and filled with data as in the following example (matrices are created and filled analogously):

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

The main advantage of the parallel container data types in SkelCL as compared with the corresponding data types in OpenCL is that the necessary data transfers between the memories of the CPU and GPUs are performed by the system implicitly, as explained further in the implementation section.

2.3 Data (Re-)Distributions

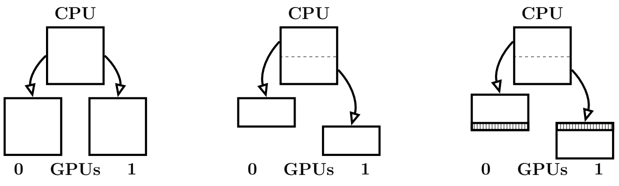
To achieve scalability of applications on systems comprising multiple GPUs, it is crucial to decide how the application's data are distributed across all available GPUs. Applications often require different distributions for their computational steps. Distributing and re-distributing data between GPUs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

SkelCL offers the programmer a *distribution* mechanism that describes how a particular container is distributed among the available GPUs. The programmer can abstract from explicitly managing memory ranges which are spread or shared among multiple GPUs: the programmer can work with a distributed container as a self-contained entity.

SkelCL currently offers four kinds of distribution: *single*, *copy*, *block*, and *overlap*. Fig. 1 shows how a matrix can be distributed on a system with two GPUs. the *single* distribution (omitted in the figure) means that matrix whole data is stored on a single GPU (the first GPU if not specified otherwise). The *copy* distribution in Fig. 1 copies matrix data to each available GPU. By the *block* distribution, each GPU stores a contiguous, disjoint chunk of the matrix. The *overlap* distribution splits the matrix into one chunk for each GPU; in addition, each chunk contains a number of continuous rows from the neighboring chunks. Figure 1c illustrates the overlap distribution: GPU 0 receives the top chunk ranging from the top row to the middle, while GPU 1 receives the second chunk ranging from the middle row to the bottom.

2.4 Patterns of Parallelism (Skeletons)

While the concrete operations performed in an application are (of course) application-specific, the general structure of parallelization often follows common parallel patterns that are reused in different applications. For example, operations can be performed for every entry of an input vector, which is the well-known *map* pattern of data-parallel programming, or two vectors are combined element-wise into an output vector, which is again the common *zip* pattern of parallelism.



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

Fig. 1. Distributions of a matrix in SkelCL (without `single`).

SkelCL extends OpenCL by introducing such high-level programming patterns, called *algorithmic skeletons* [10]. Formally, a skeleton is a higher-order function that executes one or more user-defined (so-called *customizing*) functions in a predefined parallel manner, while hiding the details of parallelism and communication from the user. We show here for brevity the definitions of some basic skeletons on a vector data type. We do this semi-formally, with v, vl and vr denoting vectors with elements v_i, vl_i and vr_i where $0 < i \leq n$, correspondingly:

- The *map skeleton* applies a unary customizing function f to each element of an input vector v, i. e.:
$$\mathit{map}\ f\ [v_1, v_2, \ldots, v_n] = [f(v_1), f(v_2), \ldots, f(v_n)]$$
- The *zip skeleton* operates on two vectors vl and vr, applying a binary customizing operator $\oplus$ pairwise:
$$\begin{aligned}\mathit{zip}\ (\oplus)\ [vl_1, vl_2, \ldots, vl_n]\ [vr_1, vr_2, \ldots, vr_n] =& \\ [vl_1 \oplus vr_1, vl_2 \oplus vr_2, \ldots, vl_n \oplus vr_n]&\end{aligned}$$
- The *reduce skeleton* computes a scalar value from a vector using a binary associative operator $\oplus$, i. e.:
$$\mathit{red}\ (\oplus)\ [v_1, v_2, \ldots, v_n] = v_1 \oplus v_2 \oplus \cdots \oplus v_n$$

These basic skeletons can be composed to express more complex algorithms. For example, the dot product of two vectors a and b of length d is defined as:

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

which can be easily expressed using our basic skeletons zip and reduce, customized by multiplication and addition, correspondingly:

$$\mathit{dotProduct}(a, b) = \mathit{red}\ (+)\ (\ \mathit{zip}\ (\cdot)\ a\ b\ ) \tag{2}$$

As an example of a non-basic skeleton, we present here the *allpairs skeleton*. All-pairs computations occur in a variety of applications, ranging from pairwise Manhattan distance computations in bioinformatics [6] to N-Body simulations in physics [4]. These applications follow a common computation scheme: for two sets of entities, the same computation is performed for all pairs of entities from the first set combined with entities from the second set. We represent entities as d-dimensional vectors, and sets of entities as corresponding matrices. The allpairs skeleton with a customizing binary operation $\oplus$ on vectors is defined as follows:

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

with entries $c_{i,j}$ computed as follows: $c_{i,j} = [a_{i,1}\ \cdots\ a_{i,d}] \oplus [b_{j,1}\ \cdots\ b_{j,d}]$.

Let us consider a first simple example application which can be expressed by customizing the allpairs skeleton with a particular function $\oplus$. The Manhattan distance (or L_1 distance) is defined for two vectors, v and w, of equal length d:

$$ManDist(v, w) = \sum_{k=1}^{d} |v_k - w_k| \qquad (3)$$

In [6], the so-called Pairwise Manhattan Distance (*PMD*) is studied as a fundamental operation in hierarchical clustering for data analysis. *PMD* is obtained by computing the Manhattan distance for every pair of rows of a given matrix. This computation for arbitrary matrix A can be expressed using the allpairs skeleton customized with the Manhattan distance defined in (3):

$$PMD(A) = allpairs(ManDist)\,(A, A) \qquad (4)$$

3 Programming in SkelCL

In original OpenCL, computations are expressed as *kernels* which are executed in a parallel manner on a GPU: the application developer must explicitly specify how many instances of a kernel are launched. In addition, kernels usually take pointers to GPU memory as input and contain program code for reading/writing single data items from/to it. These pointers have to be used carefully, because no boundary checks are performed by OpenCL.

The programming model of SkelCL differs from OpenCL: rather than writing low-level kernels, the application developer customizes suitable skeletons by providing application-specific functions which are often much simpler than kernels as they specify an operation on basic data items rather than containers. Skeletons are created as objects by providing customizing functions which, for technical reasons, must not be recursive and may only contain OpenCL C (not C++) code.

3.1 Example: Dot Product of Vectors

Equation (2) expresses the dot product of two vectors as a composition of two skeletons, zip and reduce. In SkelCL, a zip skeleton object customized by multiplication is created and then used as follows:

```
Zip<float> mult("float func(float x,float y){ return x*y;}");
        resultVector = mult( leftVector, rightVector );
```

The necessary reduce skeleton customized by addition is created similarly as an object and then called as follows:

```
Reduce<float> sum("float func(float x,float y){ return x+y;}");
                result = sum( inputVector );
```

These definitions lead directly to the SkelCL code for dot product shown in Listing 1.1 (8 lines of code plus comments). The OpenCL-based implementation of dot product provided by NVIDIA [17] requires 68 lines (kernel function: 9 lines, host program: 59 lines), i.e., it is significantly longer than our SkelCL code.

```
skelcl::init(); /* initialize SkelCL */
/* create skeleton objects: */
Zip<float>    mult(
  "float mult(float x,float y) {return x*y;}");
Reduce<float> sum (
  "float func(float x,float y) {return x+y;}");
/* create input vectors and fill with data: */
Vector<float> A(SIZE); fillVector(A);
Vector<float> B(SIZE); fillVector(B);
/* execute skeleton objects: */
Vector<float> C = sum( mult(A,B) );
```

Listing 1.1. A SkelCL code for computing the dot product of two vectors

3.2 Example: Matrix Multiplication

Matrix multiplication is a basic linear algebra operation, which is a building block of many scientific applications. An $n \times d$ matrix A is multiplied by a $d \times m$ matrix B, producing an $n \times m$ matrix $C = A \times B$ whose element $C_{i,j}$ is computed as the dot product of the ith row of A with jth column of B. The matrix multiplication can be expressed using the allpairs skeleton introduced in Section 2.4 as follows:

$$A \times B = allpairs(dotProduct)\left(A, B^T\right) \tag{5}$$

where B^T is the transpose of matrix B.

Listing 1.2 shows the SkelCL program for computing matrix multiplication using the *allpairs* skeleton; the code follows directly from the skeleton formulation (5). In the first line, the SkelCL library is initialized. Skeletons are implemented as classes in SkelCL and customized by instantiating a new object, like in line 2. The `Allpairs` class is implemented as a template class specified with the data type of matrices involved in the computation (`float`). This way the implementation can ensure the type correctness by checking the types of the arguments when the skeleton is executed in line 10. The customizing function – specified as a string (lines 3–7) – is passed to the constructor. SkelCL defines custom data types (`float_vector_t` in line 3) for representing vectors in the code of the customizing

```
 1  skelcl::init();
 2  Allpairs<float> mm(
 3   "float func(float_vector_t ar, float_vector bc) {\
 4    float c = 0.0f;\
 5    for (int i = 0; i < length(ar); ++i) {\
 6      c += getElementFromRow(ar,i) * getElementFromCol(bc,i);}\
 7    return c; }");
 8  Matrix<float> A(n, k); fill(A);
 9  Matrix<float> B(k, m); fill(B);
10  Matrix<float> C = mm(A, B);
```

Listing 1.2. Matrix multiplication in SkelCL using the *allpairs* skeleton.

function. Helper functions are used for accessing elements from the row of matrix A and the column of matrix B (line 6). The transpose of matrix B required by the definition (5) is implicitly performed by accessing elements from the columns of B using the helper function `getElementFromCol`. After initializing the two input matrices (line 8 and 9), the calculation is performed in line 10.

4 Transformation Rules for Optimization

Our approach is based on formally defined algorithmic skeletons. This allows for systematically applying semantics-preserving transformations to SkelCL programs with the goal of their optimization. In this section, we briefly illustrate two types of transformation rules: *specialization rules* and *(de)composition rules.*

4.1 Specialization Rule: Optimizing the Allpairs Skeleton

Specialization rules enable optimizations of skeleton implementations using additional, application-specific semantical information. We illustrate specialization for our allpairs skeleton and the matrix multiplication example. If the customizing function f of the allpairs skeleton can be expressed as a sequential composition (denoted with $\circ$) of zip and reduce customized with a binary operator $\odot$ and a binary, associative operator $\oplus$.:

$$f = \mathit{reduce}\ (\oplus)\ \circ \mathit{zip}\ (\odot) \tag{6}$$

then an optimized implementation of the allpairs skeleton for multiple GPUs can be automatically derived as described in detail in [20].

By expressing the customizing function of the allpairs skeleton as a zip-reduce composition, we provide additional semantical information about the memory access pattern of the customizing function to the skeleton implementation, thus allowing for improving the performance. The particular optimization using (6) takes into account the OpenCL programming model that organizes *work-items* (i. e., threads executing a kernel) in *work-groups* which share the same GPU local memory. By loading data needed by multiple work-items of the same work-group into the fast local memory, we can avoid repetitive accesses to the slow global memory. The semantical information of the zip-reduce pattern allows the implementation to load chunks of both involved vectors into the small local memory and reduce them there, before processing the next chunks. That means that the two skeletons zip and reduce are not executed one after the other, but rather the optimized implementation interleaves these two steps. This results in a significant performance gain, as described in Section 6.

For the Pairwise Manhattan Distance, we can express the customizing function as a zip-reduce composition, using the binary operator $a \ominus b = |a - b|$ as customizing function for zip, and addition as customizing function for the reduce skeleton:

$$\mathit{ManDist}(a,b) = \sum_{i=1}^{n} |a_i - b_i| = (\mathit{reduce}\ (+)\ \circ \mathit{zip}\ (\ominus))\ [a_1\ \cdots\ a_n]\ [b_1\ \cdots\ b_n]$$

```
1 skelcl::init();
2 Zip<float> mult(
3     "float func(float x, float y) { return x*y; }");
4 Reduce<float> sum_up(
5     "float func(float x, float y) { return x+y; }");
6 Allpairs<float> mm(sum_up, mult);
7 Matrix<float> A(n, d); fill(A);
8 Matrix<float> B(d, m); fill(B);
9 Matrix<float> C = mm(A, B);
```

Listing 1.3. Matrix multiplication in SkelCL using the specialized *allpairs* skeleton.

Similarly, as already demonstrated by (2), dot product (which is the customizing function of allpairs for matrix multiplication) can be expressed as a zip-reduce composition. The corresponding optimized SkelCL code is shown in Listing 1.3. In lines 2 and 3, the zip skeleton is defined using multiplication as customizing function and in lines 4 and 5, the reduce skeleton is customized with addition. These two customized skeletons are passed to the allpairs skeleton on its creation in line 6. This triggers our specialization rule and an optimized implementation is generated. In line 9, the skeleton is executed taking two input matrices and producing the output matrix.

Currently, SkelCL implements such customization of the allpairs skeleton by a combination of the zip and reduce skeleton as a special case. Therefore, the allpairs skeleton in Listing 1.3 accepts a zip and reduce skeleton as customizing functions instead of a string as shown earlier in Listing 1.2. We plan to generalize this in the future and allow arbitrary skeletons to be used as customizing functions of other skeletons – of course when the types match.

4.2 Composition Rules: Optimizing Scan and Reduce

In this section we present examples of composition rules which allow the application programmer to systematically apply transformations to SkelCL programs with the goal of optimization.

Our examples involve the scan skeleton (a. k. a. prefix-sum) which yields an output vector with each element obtained by applying a binary associative operator $\oplus$ to the elements of the input vector up to the current element's index:

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

The scan skeleton has been well studied and used in many parallel applications [5].

- **Scan-Reduce Composition**: This rule allows for a composition of scan followed by reduction to be expressed as a single reduction operating on pairs of values. For arbitrary binary, associative operators $\oplus$ and $\otimes$, such that $\otimes$ distributes over $\oplus$, it holds:

$$\mathit{red}(\oplus) \circ \mathit{scan}(\otimes) \;=\; \pi_1 \,\circ\, \mathit{red}\,(\langle\!\langle \oplus, \otimes \rangle\!\rangle) \,\circ\, \mathit{mappair} \tag{7}$$

where function $pair$, π_1 and operator $\langle\oplus,\otimes\rangle$ are defined as follows:

$$pair\, a \overset{\text{def}}{=} (a, a), \tag{8}$$

$$\pi_1\, (a, b) \overset{\text{def}}{=} a, \tag{9}$$

$$(s_1, r_1)\, \langle\oplus,\otimes\rangle\, (s_2, r_2) \overset{\text{def}}{=} (s_1 \oplus (r_1 \otimes s_2)\, , r_1 \otimes r_2) \tag{10}$$

- **Scan-Scan Composition:** This rule allows to replace two repetitive scan skeletons by a single one.
  For associative operators $\oplus$ and $\otimes$, where $\otimes$ distributes over $\oplus$,

$$\begin{aligned} &scan\,(\oplus) \ \circ \ scan\,(\otimes) \ = \\ &\qquad map\, \pi_1 \ \circ \ scan\,(\langle\!\langle\oplus,\otimes\rangle\!\rangle) \ \circ \ map\, pair \end{aligned} \tag{11}$$

Besides composing skeletons together, it also sometimes pays off to decompose them, for example to split a reduction into multiple steps (so-called decomposition rule). The motivation, proof of correctness and a discussion of the performance benefits of the composition and decomposition rules can be found in [11].

5 Implementation of SkelCL

SkelCL is implemented as a C++ library which generates valid OpenCL code from SkelCL programs. The customizing functions provided by the application developer is combined with skeleton-specific OpenCL code to generate an OpenCL kernel function, which is eventually executed on a GPU. A customized skeleton can be executed on both single- and multi-GPU systems. In case of a multi-GPU system, the calculation specified by a skeleton is performed automatically on all GPUs available in the system.

Skeletons operate on container data types (in particular vectors and matrices) which alleviate the memory management of GPUs. The SkelCL implementation namely ensures that data is copied automatically to and from GPUs, instead of manually performing data transfers as required in OpenCL. Before performing a computation on container types, the SkelCL system ensures that all input containers' data is available on all participating GPUs. This may result in implicit (automatic) data transfers from the CPU to GPU memory, which in OpenCL would require explicit programming. Similarly, before any data is accessed on the CPU, the implementation of SkelCL ensures that this data on the CPU is up-to-date by performing necessary data transfers implicitly and automatically.

For multi-GPU systems, the application developer can use the distributions directives of SkelCL introduced in Section 2.3 to specify how data is distributed across the GPUs in the system. If no distribution is set explicitly then every skeleton implementation selects a suitable default distribution for its input and output containers. Containers' distributions can be changed at runtime: this implies data exchanges between multiple GPUs and the CPU, which are performed by the SkelCL implementation implicitly. Implementing such data transfers in the standard OpenCL is a cumbersome task: data has to be downloaded to the

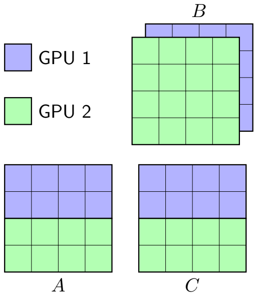


Fig. 2. Data distributions used for a system with two GPUs: matrices A and C are block distributed, matrix B is copy distributed.

CPU before it is uploaded to the GPUs, including the corresponding length and offset calculations; this results in a lot of low-level code which becomes completely hidden when using SkelCL.

For example, two SkelCL distributions are used in our multi-GPU implementation of the allpairs skeleton, as shown in Figure 2: Matrix B is *copy* distributed, i. e., it is copied entirely to all GPUs in the system. Matrix A and C are *block* distributed, i. e., they are row-divided into as many equally-sized blocks as GPUs are available; each block is copied to its corresponding GPU. Following these distributions, each GPU computes one block of the result matrix C. In the example with two GPUs shown in Figure 2, the first two rows of C are computed by GPU 1 and the last two rows by GPU 2. The allpairs skeleton uses these distributions by default; therefore, no changes to the already discussed SkelCL codes for matrix multiplication are necessary for using multiple GPUs.

The object-oriented design of SkelCL allows the developers to extend it easily: e.g., in order to add a new skeleton to SkelCL, a new class with the skeleton's implementation has to be provided, while all existing classes and concepts /data containers and data distributions) can be freely reused.

6 Experimental Evaluation

We use matrix multiplication as an example to evaluate our SkelCL implementations regarding programming effort and performance. We compare the following six implementations of the matrix multiplication:

1. the OpenCL implementation from [14] without optimizations,
2. the optimized OpenCL implementation from [14] using GPU local memory,
3. the optimized BLAS implementation by AMD [2] written in OpenCL,
4. the optimized BLAS implementation by NVIDIA [16] written in CUDA,
5. the SkelCL implementation using the generic allpairs skeleton (Listing 1.2),
6. the SkelCL implementation optimized using specialization (Listing 1.3).

6.1 Programming effort

As the simplest criterion for estimating the programming effort, we use the program size in lines of code (LoC). Figure 3 shows the number of LoCs required for each of the six implementations. We did not count those LoCs which are not relevant for parallelization and are similar in all six implementations, like initializing the input matrices with data and checking the result for correctness. For every implementation, we distinguish between CPU code and GPU code. For the OpenCL implementations, the GPU code is the kernel definition; the CPU code includes the initialization of OpenCL, memory allocations, explicit data transfer operations, and management of the execution of the kernel. For the BLAS implementations, the CPU code contains the initialization of the corresponding BLAS library, memory allocations, as well as a library call for performing the matrix multiplication; no definition of GPU code is necessary, as the GPU code is defined inside the library function calls. For the generic allpairs skeleton (Listing 1.2), we count lines 1–2 and 8–10 as the CPU code, and the definition of the customizing function in lines 3–7 as the GPU code. For the allpairs skeleton customized with zip-reduce (Listing 1.3), lines 3 and 5 are the GPU code, while all other lines constitute the CPU code.

As expected, both skeleton-based implementations are clearly the shortest due to using high-level constructs, with 10 and 9 LoCs, correspondingly. The next shortest implementation is the cuBLAS implementation with 65 LoCs – 7 times longer than the SkelCL implementations. The other implementations require even 9 times more LoCs than the SkelCL implementation. Besides their length, the other implementations require the application developer to perform many low-level, error-prone tasks, like dealing with pointers or offset calculations. Furthermore, the skeleton-based implementations are more general, as they can be used for arbitrary allpairs computations, while the OpenCL and CUDA implementations perform matrix multiplication only.

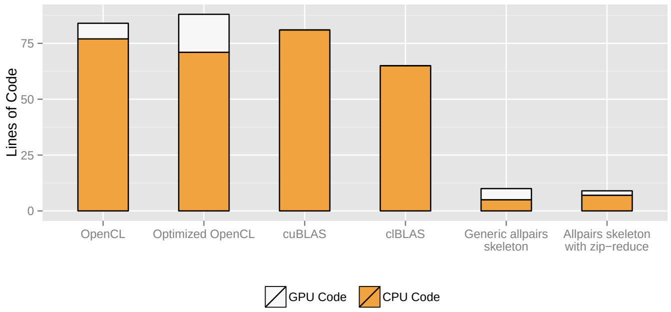


Fig. 3. Programming effort (Lines of Code) of all compared implementations.

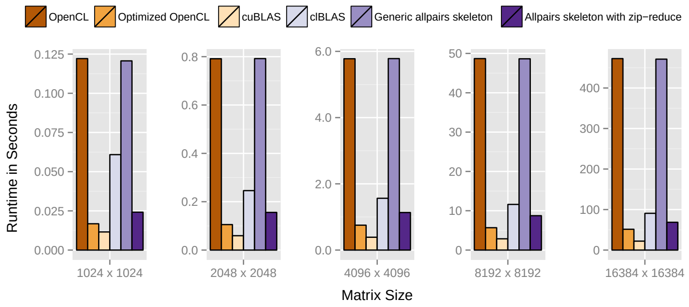


Fig. 4. Runtime of different matrix multiplication implementations on a NVIDIA system for different sizes for the matrices.

6.2 Performance experiments

We performed our performance experiments with the six different implementations of matrix multiplication on a test system using a host PC with a quad-core CPU (Intel Xeon E5520, 2.26 GHz) and 12 GB of memory, connected to a Tesla S1070 computing system equipped with 4 Tesla GPUs. Its dedicated 16 GB of memory (4 GB per GPU) is accessed with up to 408 GB/s (102 GB/s per GPU). Each GPU comprises 240 streaming processor cores running at 1.44 GHz. In all experiments, we include the time of data transfers to/from the GPU, i. e. the measured runtime consists of: 1) uploading the input matrices to the GPU; 2) performing the actual matrix multiplication; 3) downloading the computed result matrix.

Using one GPU. Figure 4 shows the runtime in seconds of all six implementations for different sizes of the matrices (note that for readability reasons, all charts are scaled differently). Clearly, the unoptimized OpenCL- and SkelCL-based implementations are the slowest, because both do not use the fast GPU local memory, in contrast to all other implementations. The SkelCL implementation optimized with specialization rule performs between 5.0 and 6.8 times faster than the implementation using the generic allpairs skeleton, but is 33% slower on 16384 × 16384 matrices than the optimized OpenCL implementation using local memory. However, the latter implementation works only for square matrices and, therefore, omits many conditional statements and boundary checks. Not surprisingly, cuBLAS by NVIDIA is the fastest of all implementations, as it is manually tuned for NVIDIA GPUs using CUDA. The clBLAS implementation by AMD using OpenCL is apparently well optimized for AMD GPUs but performs poorly on other hardware. Our optimized allpairs skeleton implementation outperforms the clBLAS implementation for all matrix sizes tested.

Using multiple GPUs. Figure 5 shows the runtime behavior of both implementations using the allpairs skeleton on up to four GPUs of our multi-GPU system. The other four implementations (OpenCL and CUDA) are not able to handle multiple GPUs and would have to be specially rewritten for such systems. We observe

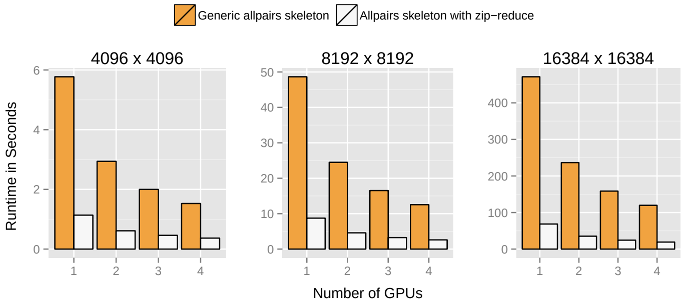


Fig. 5. Runtime of the allpairs based implementations using multiple GPUs.

a good scalability of our skeleton-based implementations, achieving speedups between 3.09 and 3.93 when using four GPUs.

7 Conclusion and Related Work

This paper presents the SkelCL high-level programming model for multi-GPU systems and its implementation as a library. SkelCL is built on top of the OpenCL standard which provides familiar programming environment for application developers, portability across various many-core platforms of different vendors, and proven best practices of OpenCL programming and optimization. Our SkelCL approach significantly raises OpenCL's low level of abstraction: it offers 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 data dustributions can be viewed as instances of *covers* [15, 19] which define a general framework for reasoning about possible distributions of data. Semantic-preserving transformation rules allows for systematically optimizing SkelCL programs. The SkelCL library is available as open source software from `http://skelcl.uni-muenster.de`.

There are other approaches to simplify GPU programming. *SkePU* [8] and *Muesli* [9] are fairly similar to SkelCL, but greatly differ in their focus and implementation, as discussed in [21]. There exist wrappers for OpenCL or CUDA as well as convenient libraries for GPU Computing, most popular of them are *Thrust* [12] and *Bolt* [3]. Compiler-based approaches similar to the popular OpenMP [18] include *OpenACC* [1] and *OmpSs-OpenCL* [7]. While reducing boilerplate in GPU-targeted applications, these approaches do not simplify the programming process by introducing high-level abstractions as done in SkelCL.

Acknowledgments

This work is partially supported by the OFERTIE (FP7) and MONICA projects. We would like to thank the anonymous reviewers for their valuable comments, as well as NVIDIA for their generous hardware donation used in our experiments.

References

1. *OpenACC Application Program Interface*, 2011. Version 1.0.
2. AMD. *AMD APP SDK code samples*, February 2013. Version 2.7.
3. AMD. Bolt – A C++ template library optimized for GPUs, 2013.
4. N. Arora, A. Shringarpure, and R. W. Vuduc. Direct N-body kernels for multicore platforms. In *2012 41st International Conference on Parallel Processing*, pages 379–387, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
5. G. E Blelloch. Prefix sums and their applications. In *Sythesis of parallel algorithms*, pages 35—60. Morgan Kaufmann Publishers Inc., 1990.
6. D.-J. Chang, A.H. Desoky, M. Ouyang, and E.C. Rouchka. Compute pairwise manhattan distance and pearson correlation coefficient of data points with GPU. In *10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pages 501–506, 2009.
7. V. K. Elangovan, R.M. Badia, and E. A. Parra. OmpSs-OpenCL programming model for heterogeneous systems. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, volume 7760 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2013.
8. 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)*, pages 5–14, 2010.
9. S. Ernsting and H. Kuchen. Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *International Journal of High Performance Computing and Networking*, 7(2):129–138, 2012.
10. S. Gorlatch and M. Cole. Parallel skeletons. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1417–1422. Springer, 2011.
11. S. Gorlatch and C. Lengauer. (De)Composition Rules for Parallel Scan and Reduction. In *In Proc. 3rd Int. Working Conf. on Massively Parallel Programming Models (MPPM'97)*, pages 23–32. IEEE Computer Society Press, 1998.
12. J. Hoberock and N. Bell (NVIDIA). Thrust: A Parallel Template Library, 2013.
13. Khronos Group. *The OpenCL Specification*, November 2013. Version 2.0.
14. D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach.* Morgan Kaufman, 2010.
15. T. Nitsche. Skeleton implementations based on generic data distributions. In *2nd Intern. Workshop on Constructive Methods for Parallel Programming*, 2000.
16. NVIDIA. CUBLAS, 2013. http://developer.nvidia.com/cublas.
17. NVIDIA. *NVIDIA CUDA SDK code samples*, February 2013. Version 5.0.
18. OpenMP Architecture Review Board. *OpenMP API*, 2013. Version 4.0.
19. P. Pepper and M. Südholt. Deriving parallel numerical algorithms using data distribution algebras: Wang's algorithm. In *30th Annual Hawaii International Conference on System Sciences (HICSS)*, pages 501–510, 1997.
20. M. Steuwer, M. Friese, S. Albers, and S. Gorlatch. Introducing and implementing the allpairs skeleton for programming multi-GPU systems. *International Journal of Parallel Programming*, 2013.
21. M. Steuwer and S. Gorlatch. SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems. In Malyshkin Victor, editor, *Parallel Computing Technologies - 12th International Conference (PaCT 2013)*, volume 7979 of *Lecture Notes in Computer Science*, pages 258–272. Springer, 2013.