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

Introducing and Implementing the Allpairs Skeleton for Programming Multi-GPU Systems

Michel Steuwer · Malte Friese · Sebastian Albers · Sergei Gorlatch

Received: date / Accepted: date

Abstract Algorithmic skeletons simplify software development: they abstract typical patterns of parallelism and provide their efficient implementations, allowing the application developer to focus on the structure of algorithms, rather than on implementation details. This becomes especially important for modern parallel systems with multiple Graphics Processing Units (GPUs) whose programming is complex and error-prone, because state-of-the-art programming approaches like CUDA and OpenCL lack high-level abstractions.

We define a new algorithmic skeleton for *allpairs computations* which occur in real-world applications, ranging from bioinformatics to physics. We develop the skeleton's generic parallel implementation for multi-GPU Systems in OpenCL. To enable the automatic use of the fast GPU memory, we identify and implement an optimized version of the allpairs skeleton with a customizing function that follows a certain memory access pattern.

We use matrix multiplication as an application study for the allpairs skeleton and its two implementations and demonstrate that the skeleton greatly simplifies programming, saving up to 90% of lines of code as compared to OpenCL. The performance of our optimized implementation is up to 6.8 times higher as compared with the generic implementation and is competitive to the performance of a manually written optimized OpenCL code.

Keywords High-Level Programming Models · Algorithmic Skeletons · GPU Computing · Allpairs Computation · SkelCL

M. Steuwer · M. Friese · S. Albers · S. Gorlatch
Department of Mathematics and Computer Science, University of Muenster, Germany
E-mail: m.steuwer@uni-muenster.de, s.albers@uni-muenster.de, gorlatch@uni-muenster.de

The final publication is available at Springer via
http://dx.doi.org/10.1007/s10766-013-0265-6.

1 Introduction

Algorithmic skeletons enable application developers to program parallel systems at a high level of abstraction [4]. Skeletons abstract recurring patterns of parallel programming and provide efficient parallel implementations for these patterns. The application developer uses skeletons by providing application-specific code (so-called customizing function) which customizes the skeletons' behavior. Skeletons thus allow the application developer to focus on the structure of the algorithms, rather than on their implementation details.

We develop SkelCL [17] – a skeleton library for programming modern many-core architectures, especially systems with Graphics Processing Units (GPUs). Programming these systems using the current low-level approaches like CUDA [14] and OpenCL [13] is challenging: parallelism has to be expressed explicitly by defining and executing parallel *kernels*, and memory management is cumbersome, as data has to be moved manually to and from the GPU memory. By providing skeletons on container data types, SkelCL alleviates programming of systems with GPUs: parallelism is expressed implicitly, using skeletons, and memory management is performed automatically by the SkelCL implementation built on top of OpenCL. The especially tricky programming of multi-GPU systems is greatly simplified by SkelCL's data distribution mechanism which automatically moves data between multiple GPUs.

In this paper, we aim at *allpairs computations* which occur in a variety of applications, ranging from matrix multiplication and pairwise Manhattan distance computations in bioinformatics [3] to N-Body simulations in physics [2]. These applications share a common computational pattern: for two sets of entities, the same computation is performed independently for all pairs in which entities from the first set are combined with entities from the second set. Previous work discussed specific allpairs applications and their parallel implementations on multi-core CPUs [2], the Cell processor [18], and GPUs [3, 16]; it demonstrated that developing well-performing implementations for allpairs is challenging, especially on GPU systems, as it requires exploiting their complex memory hierarchy and assumes a deep knowledge of the target hardware architecture.

The contributions and structure of this paper are as follows. To enable application developers to efficiently use the allpairs computational pattern, we formally define the allpairs skeleton and provide its generic parallel implementation in OpenCL (Section 2). We optimize the use of the GPU memory hierarchy by implementing a specialized version of the allpairs skeleton with a customizing function that follows a certain memory access pattern (Section 3). We show that our implementations can be used for single- and multi-GPU systems (Section 4). We evaluate the runtime performance and the programming effort of the allpairs skeleton within the SkelCL library, using matrix multiplication as application study (Section 5). We discuss related work and conclude in Section 6.

2 The Allpairs Skeleton and its Implementation

We define the allpairs computation pattern for two sets of entities, each entity represented by a vector of length d. Let the cardinality of the first set be n and the cardinality of the second set be m. We model the first set as a $n \times d$ matrix A and the second set as a $m \times d$ matrix B. The allpairs computation yields an output matrix C of size $n \times m$ as follows: $c_{i,j} = A_i \oplus B_j$, where A_i and B_j are row vectors of A and B, correspondingly: $A_i = [A_{i,1}, \cdots, A_{i,d}]$, $B_j = [B_{j,1}, \cdots, B_{j,d}]$, and $\oplus$ is a binary operator defined as vectors.

Definition 1 Let A be a $n \times d$ matrix, B be a $m \times d$ matrix, and C be a $n \times m$ matrix, with their elements $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$ respectively. The algorithmic skeleton *allpairs* with customizing binary function $\oplus$ is defined as follows:

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

where elements $c_{i,j}$ of the $n \times m$ matrix C are calculated as follows:

$$c_{i,j} = [a_{i,1} \ \cdots \ a_{i,d}] \oplus [b_{j,1} \ \cdots \ b_{j,d}]$$

Figure 1(a) illustrates this definition: the element $c_{2,3}$ of matrix C marked as ③ is computed by combining the second row of A marked as ① with the third row of B marked as ② using the binary operator $\oplus$. Figure 1(b) shows the same computation with the transposed matrix B.

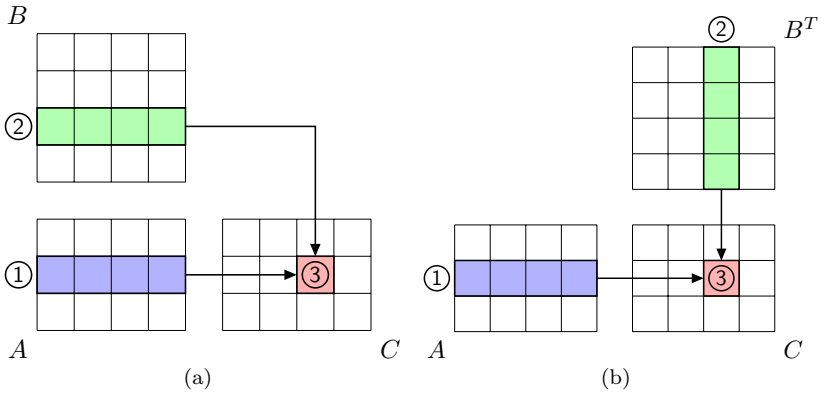

Fig. 1 The allpairs computation. Left: element $c_{2,3}$ (③) is computed by combining the second row of A (①) with the third row of B (②) using the binary operator $\oplus$. Right: the transposed matrix B is used.

Let us consider two example applications which can be expressed by customizing the allpairs skeleton with a particular function $\oplus$.

Example 1: The Manhattan distance (or L_1 distance) is a measure of distance which is used in many applications. In general, it is defined for two vectors, v and w, of equal length d, as follows:

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

In [3], 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 (1):

$$PMD(A) = allpairs(ManDist)\,(A, A) \tag{2}$$

The $n \times n$ matrix computed by the customized skeleton contains the Manhattan distance for every pair of rows of the input $n \times d$ matrix A.

Example 2: 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 a $n \times m$ matrix $C = A \times B$ whose element $c_{i,j}$ is computed as the dot product of the ith row of A with the jth column of B. The dot product of two vectors a and b of length d is computed as:

$$dotProduct(a, b) = \sum_{k=1}^{d} a_k \cdot b_k \tag{3}$$

The matrix multiplication can be expressed using the allpairs skeleton as:

$$A \times B = allpairs(dotProduct)\left(A, B^T\right) \tag{4}$$

where B^T is the transpose of matrix B. We will use the matrix multiplication as our running example for the allpairs skeleton throughout the paper.

We develop the allpairs skeleton within the skeleton library SkelCL [17], which is built on top of OpenCL and targets modern parallel systems with multiple GPUs. Currently, five other skeletons are available in SkelCL: *map*, *zip*, *reduce*, *scan*, and *mapOverlap*. Skeletons operate on container data types (in particular vectors and matrices) which alleviate the memory management of GPUs: data is copied automatically to and from GPUs, instead of manually performing data transfers as required in OpenCL. For programming multi-GPU systems, SkelCL offers the application programmer a data distribution mechanism to specify how the data of a container is distributed among the GPUs in the system. The container's data can either be assigned to a single

```
1  skelcl::init();
2  Allpairs<float(float, float)> mm(
3   "float func(float_matrix_t a, float_matrix_t b) {\
4    float c = 0.0f;\
5    for (int i = 0; i < width(a); ++i) {\
6      c += getElementFromRow(a, i) * getElementFromCol(b, 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 Matrix multiplication in SkelCL using the *allpairs* skeleton.

GPU, be copied to all GPUs, or be partitioned in equal blocks across the GPUs, possibly with an overlap. If the data distribution is changed in the program, the necessary data movements are done automatically by the system [17].

Listing 1 shows the SkelCL program for computing matrix multiplication using the *allpairs* skeleton; the code follows directly from the mathematical formulation (4). 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 types of matrices involved in the computation (`float(float, 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 (line 3 – 7) – is passed to the constructor. Data types for matrices (`float_matrix_t` in line 3) are defined by the SkelCL implementation and used as arguments of helper functions for accessing elements from both matrices (line 6). The transpose of matrix B required by the definition (4) 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.

In our SkelCL library, skeletons are implemented by translating them into executable OpenCL code. Listing 2 shows the OpenCL kernel which is combined with the given customizing function by the implementation of the allpairs skeleton. The customizing function (named *`func`* in Listing 1) is renamed to match the name used in the function call in the implementation (`USER_FUNC` in Listing 2). In addition, the types used in the predefined OpenCL kernel (`TYPE_LEFT`, `TYPE_RIGHT`, and `TYPE_OUT` in Listing 2) are adjusted to match the actual types of the elements used in the computation (in this case, all three types are `float`). These modifications ensure that a valid OpenCL program performing the allpairs calculation is constructed. This generated OpenCL program is executed once for every element of the output matrix C. In lines 6 – 7, the implementation prepares variables (`Am` and `Bm`) of a predefined data type (`float_matrix_t`) which encapsulate the matrices A and B and passes them to the customizing function which is called in line 9.

```
1 __kernel void allpairs(const __global TYPE_LEFT*  A,
2                         const __global TYPE_RIGHT* B,
3                               __global TYPE_OUT*   C,
4                               int n, int d, int m) {
5   int col = get_global_id(0); int row = get_global_id(1);
6   float_matrix_t Am; Am.data = A; Am.width = d; Am.row = row;
7   float_matrix_t Bm; Bm.data = B; Bm.width = m; Bm.col = col;
8   if (row < n && col < m)
9     C[row * m + col] = USER_FUNC(Am, Bm); }
```

Listing 2 Generic OpenCL kernel used in the implementation of the allpairs skeleton.

To achieve high performance, skeleton implementations must efficiently exploit the complex memory hierarchy of multi-GPU architectures. There are two main types of memory in OpenCL: *global* and *local memory*. The global memory is typically large but slow; the local memory is small but fast and has similar performance as caches in traditional systems, but has to be programmed manually. On modern GPUs, accesses to the global memory are very expensive, taking up to 800 processor cycles, as compared to only few cycles required to access the local memory [14].

The generic implementation of the allpairs skeleton in Listing 2 makes no assumption about the order in which the customizing function (`USER_FUNC`) accesses the elements of its two input vectors. In this general case, we cannot assume that the two vectors fit entirely into the restricted GPU local memory. Therefore, we have to use only the global memory in the generic implementation. To improve our implementation of the allpairs skeleton, we restrict the memory access pattern of the customizing function in the next section.

3 The Specialized Allpairs Skeleton

In this section, we first analyze the memory access pattern of the matrix multiplication and then observe that this pattern can also be found in some other allpairs computations. We, therefore, define a specialized version of the allpairs skeleton, which is suitable for applications having this pattern, and show how it can be implemented more efficiently than the generic skeleton.

3.1 The memory access pattern of the matrix multiplication

Figure 2 shows the memory access pattern of the matrix multiplication for 4×4 matrices. To compute the element Ⓡ of the result matrix C, the first row of matrix A and the first column of matrix B are needed. In the skeleton-based code, these two vectors are used by the customizing function (which is the dot product) for pairwise computations: the two elements marked as ① are multiplied and the intermediate result is stored; then, the next elements (marked as ②) are multiplied and the result is added to the intermediate result, and so forth. Let us estimate the number of global memory accesses for

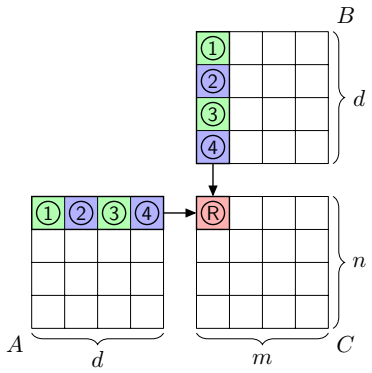

Fig. 2 Memory access pattern of the matrix multiplication $A \times B = C$.

computing an element of the matrix multiplication in the generic implementation (Listing 2). One global memory read access for every element of both input vectors is performed, and a single global memory write access is required to write the result into the output matrix. Therefore, $n \cdot m \cdot (d + d + 1)$ global memory accesses are performed in total, where n and m are the height and width of matrix C and d is the width of A and the height of B.

Obviously, the customizing function of the pairwise Manhattan distance (Example 1 in Section 2) follows the same memory access pattern as matrix multiplication. To find a common representation for a customizing function with this pairwise access pattern, we describe it as a combination of two well-known algorithmic skeletons: *zip* and *reduce*.

The *zip* skeleton combines two input vectors by applying its customizing function $(\odot)$ pairwise, producing the result vector:

$$zip\ (\odot)\ [a_1\ \cdots\ a_n]\ [b_1\ \cdots\ b_n]\ =\ [a_1 \odot b_1\ \cdots\ a_n \odot b_n]$$

The *reduce* skeleton transforms an input vector into a scalar value by repeatedly applying its binary associative customizing operator $(\oplus)$:

$$reduce\ (\oplus)\ [a_1\ \cdots\ a_n]\ =\ a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

It is possible to sequentially compose these two customized skeletons. For two functions $f : X \to Y$ and $g : Y \to Z$, the *sequential composition* denoted by $g \circ f : X \to Z$ means that f is applied first and then g is applied to the return value of f as input: $(g \circ f)(x) = g(f(x))$. Our customized skeletons are functions with types that allow their composition as follows:

$$\begin{aligned}&(reduce\ (\oplus) \circ zip\ (\odot))\,[a_1\ \cdots\ a_n]\ [b_1\ \cdots\ b_n] = \\ &\quad reduce\ (\oplus)\,(zip\ (\odot)\,[a_1\ \cdots\ a_n]\ [b_1\ \cdots\ b_n]) = (a_1 \odot b_1) \oplus \cdots \oplus (a_n \odot b_n)\end{aligned}$$

This composition of the two customized skeletons yields a function which takes two input vectors and produces a single scalar value:

$$zipReduce\ (\oplus, \odot)\ a\ b = (reduce\ (\oplus) \circ zip\ (\odot))\ a\ b \tag{5}$$

Following the definition of *zipReduce*, we can express the customizing function of the Manhattan distance as follows. We use the binary operator $a \ominus b = |a - b|$ as customizing function for zip, and addition as customizing function for the reduce skeleton:

$$\begin{aligned} ManDist(a,b) = \sum_{i=1}^{n} |a_i - b_i| &= (a_1 \ominus b_1) + \cdots + (a_n \ominus b_n) \\ &= zipReduce(+, \ominus)\ [a_1\ \cdots\ a_n]\ [b_1\ \cdots\ b_n] \end{aligned}$$

Similarly, we can express the dot product (which is the customizing function of matrix multiplication) as a zip-reduce composition, by using multiplication for customizing zip and addition for customizing the reduce skeleton:

$$\begin{aligned} dotProduct(a,b) = \sum_{i=1}^{n} a_i \cdot b_i &= (a_1 \cdot b_1) + \cdots + (a_n \cdot b_n) \\ &= zipReduce(+, \cdot)\ [a_1\ \cdots\ a_n]\ [b_1\ \cdots\ b_n] \end{aligned}$$

We can now specialize the generic Definition 1 by employing the sequential composition of the customized reduce and zip skeletons for customizing the allpairs skeleton. From here on, we refer to this specialization as the allpairs skeleton *customized with zip-reduce*.

While not every allpairs computation can be expressed using the specialization, many real-world problems can. In addition to the matrix multiplication and the pairwise Manhattan distance examples are the pairwise computation of the Pearson correlation coefficient [3] and estimation of Mutual Informations [5]. The composition of zip and reduce is well known in the functional programming world. Google's popular MapReduce programming model has been inspired by a similar composition of the *map* and reduce skeletons; see [12] for the relation of MapReduce to functional programming.

Listing 3 shows how the matrix multiplication can be programmed in SkelCL using the allpairs skeleton customized with zip-reduce. In line 1, the SkelCL library is initialized. 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. The implementation of the allpairs

```
1 skelcl::init();
2 Zip<float(float, float)> mult
3     ("float func(float x, float y) { return x*y; }");
4 Reduce<float(float, float)> sum_up
5     ("float func(float x, float y) { return x+y; }");
6 Allpairs<float(float, 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 3 Matrix multiplication in SkelCL using the specialized *allpairs* skeleton.

skeleton then uses the two customizing functions of zip and reduce to generate the OpenCL kernel performing the allpairs computation. In line 9, the skeleton is executed taking two input matrices and producing the output matrix. Note that we create objects of the same `Allpairs` class when using the generic allpairs implementation (Listing 2 line 2) and the specialized implementation (Listing 3 line 6). Depending on which of the overloaded constructors is used, either the generic or the specialized implementation is created.

3.2 Implementation of the specialized allpairs skeleton

By expressing the customizing function of the allpairs skeleton as a zip-reduce composition, we provide additional semantic information about the memory access pattern of the customizing function to the skeleton implementation, thus allowing for improving the performance. Our idea of optimization is based on 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 local memory, we can avoid repetitive accesses to the global memory.

For the allpairs skeleton with the zip-reduce customizing function, we can adopt the implementation schema for GPUs [16], as shown in Figure 3. We allocate two arrays in the local memory, one of size $r \times k$ ($r = 2$, $k = 3$ in Figure 3) for elements of A and one of size $k \times c$ ($c = 3$ in Figure 3) for elements of B. A work-group consisting of $c \times r$ work-items computes s blocks ($s = 2$ in Figure 3) of the result matrix C. In Figure 3, the two blocks marked as ⑦ and ⑧ are computed by the same work-group as follows. In the first iteration, the elements of blocks ① and ② are loaded into the local memory and combined following the zip-reduce pattern. The obtained intermediate result is stored in block ⑦. Then, the elements of block ③ are loaded and combined with the elements from ② which still reside in the local memory. The intermediate result is stored in block ⑧. In the second iteration, the algorithm continues in the same manner with blocks ④, ⑤, and ⑥, but this time, the elements of the

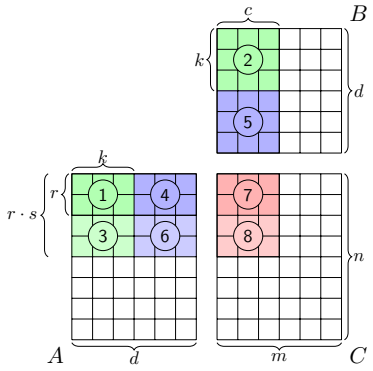

Fig. 3 Implementation schema of the specialized allpairs skeleton.

blocks are also combined with the intermediate results of the first iteration, which are stored in blocks ⑦ and ⑧. The advantage of computing multiple blocks by the same work-group is that we keep the elements of B in the local memory when computing the intermediate results, i. e., we do not reload block ② twice for the computation of blocks ⑦ and ⑧.

Every element loaded from the global memory is used by multiple work-items: e. g., the upper left element of block ① is loaded only once from the global memory, but used three times: in the computation of the upper left, upper middle, and upper right elements of ⑦. In general, every element loaded from A is reused c times, and every element from B is reused $r \cdot s$ times. As the intermediate results are stored in the global memory of matrix C, we perform two additional memory accesses (read/write) for every iteration, i. e., $2 \cdot \frac{d}{k}$ in total. Therefore, instead of $n \cdot m \cdot (d+d+1)$ global memory accesses necessary when not using the local memory, only $n \cdot m \cdot (\frac{d}{r \cdot s} + \frac{d}{c} + 2 \cdot \frac{d}{k})$ global memory accesses are performed. By increasing the parameters s and k, or the number of work-items in a work-group (c and r), more global memory accesses can be saved. However, the work-group size is limited by the GPU hardware. While the parameters can be chosen independently of the matrix sizes, we have to consider the amount of available local memory. [16] discusses how suitable parameters can be found by performing runtime experiments.

4 The Allpairs Skeleton using Multiple GPUs

The allpairs skeleton can be efficiently implemented not only on systems with a single GPU, but on multi-GPU systems as well. The SkelCL library provides four *data distributions* which specify how a container data type (vector or matrix) is distributed among multiple GPUs [17]. We use two of them in our multi-GPU implementation of the allpairs skeleton, as shown in Figure 4: 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

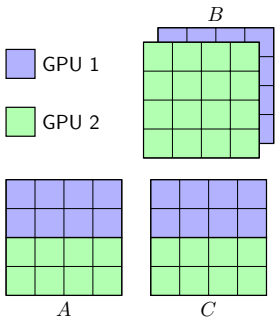

Fig. 4 Data distributions used for a system with two GPUs: matrices A and C are block distributed, matrix B is copy distributed.

corresponding GPU. Following these distributions, each GPU computes one block of the result matrix C. In the example with two GPUs shown in Figure 4, the first two rows of C are computed by GPU 1 and the last two rows by GPU 2. The allpairs skeleton automatically selects these distributions; therefore, no changes to the already discussed implementations of the matrix multiplication are necessary for using multiple GPUs.

5 Experimental Evaluation

We use matrix multiplication as an example to evaluate our allpairs skeleton implementations regarding programming effort and performance.

5.1 Implementations of matrix multiplication

We compare six different implementations of the matrix multiplication:

1. the OpenCL implementation from [11] without optimizations,
2. the optimized OpenCL implementation from [11] using GPU local memory,
3. the optimized BLAS implementation by AMD [1] written in OpenCL,
4. the optimized BLAS implementation by NVIDIA [15] written in CUDA,
5. the implementation using the generic allpairs skeleton,
6. the implementation using the allpairs skeleton customized with zip-reduce.

1. OpenCL implementation The kernel of the first, unoptimized OpenCL implementation from [11] is shown in Listing 4.

```
1 __kernel void mm(__global float* A, __global float* B,
2                   __global float* C, int m, int d, int n) {
3   int row = get_global_id(0); int col = get_global_id(1);
4   float sum = 0.0f;
5   for (int k = 0; k < d; k++)
6     sum += A[row * d + k] * B[k * n + col];
7   C[row * n + col] = sum; }
```

Listing 4 OpenCL kernel of the matrix multiplication without optimizations [11].

2. Optimized OpenCL implementations The kernel of the optimized OpenCL implementation from [11] using local memory is shown in Listing 5. Two fixed-sized arrays of local memory are allocated in lines 4 and 5. Matrix multiplication is carried out in the loop starting in line 9. In each iteration, data is loaded into the local memory (lines 10 and 11) before it is used in the computation in line 14. Note that two synchronization barriers are required (lines 12 and 15) to ensure that the data is fully loaded into the local memory and that the data is not overwritten while other work-items are still using it. Both OpenCL implementations 1. and 2. from [11] are only capable of performing matrix multiplication for square matrices.

```
1  #define TILE_WIDTH 16
2  __kernel void mm(__global float* A, __global float* B,
3                   __global float* C, int m, int d, int n) {
4    __local float Al[TILE_WIDTH][TILE_WIDTH];
5    __local float Bl[TILE_WIDTH][TILE_WIDTH];
6    int   row = get_global_id(0); int   col = get_global_id(1);
7    int l_row = get_local_id(0);  int l_col = get_local_id(1);
8    float sum = 0.0f;
9    for (int m = 0; m < d / TILE_WIDTH; ++m {
10     Al[l_row][l_col] = A[row * d + (m * TILE_WIDTH + l_col)];
11     Bl[l_row][l_col] = B[(m * TILE_WIDTH + l_row) * d + col];
12     barrier(CLK_LOCAL_MEM_FENCE);
13     for (int k = 0; k < TILE_WIDTH; k++)
14       sum += Al[l_row][k] * Bl[k][l_col];
15     barrier(CLK_LOCAL_MEM_FENCE); }
16   C[row * n + col] = sum; }
```

Listing 5 OpenCL kernel of the optimized matrix multiplication using local memory [11].

3. BLAS implementation by AMD The implementation offered by AMD is called clBLAS, written in OpenCL and is part of their Accelerated Parallel Processing Math Libraries (APPML) [1].

4. BLAS implementation by NVIDIA The cuBLAS [15] is implemented using CUDA and, therefore, can only be used on GPUs built by NVIDIA.

5. Generic allpairs skeleton Listing 1 in Section 2 shows the implementation using the generic allpairs skeleton.

6. Allpairs skeleton customized with zip-reduce Listing 3 in Section 3 shows the implementation using the allpairs skeleton customized with zip-reduce.

5.2 Programming effort

As the simplest criterion for programming effort, we use the program size in lines of code (LoC). Figure 5 shows the number of LoCs required for each of the six implementations. Table 1 presents the detailed numbers. 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, as shown in Listing 4 and Listing 5; 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), we count lines 1 – 2 and 8 – 10 as the CPU code,

Fig. 5 Programming effort (Lines of Code) of all compared implementations.

and the definition of the customizing function in lines 3 – 7 as the GPU code. For the allpairs skeleton customized with zip-reduce (Listing 3), lines 3 and 5 are the GPU code, while all other lines constitute the CPU code.

Both skeleton-based implementations are clearly the shortest, with 10 and 9 LoCs. The next shortest implementation is the cuBLAS implementation with 65 LoCs – 7 times longer than the SkelCL-based implementation. The other three implementations require even 9 times more LoCs than the SkelCL-based 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 other four implementations perform matrix multiplication only.

	Lines of Code	
Implementation	CPU	GPU
OpenCL	77	7
Optimized OpenCL	71	17
cuBLAS	81	–
clBLAS	65	–
Generic allpairs skeleton	5	5
Allpairs skeleton with zip-reduce	7	2

Table 1 Lines of Code of all compared implementations.

5.3 Runtime experiments

We performed our experiments with the six different implementations 1. – 6. of matrix multiplication on two different computer systems with GPUs:

System A: An NVIDIA S1070 equipped with four NVIDIA Tesla GPUs, each with 240 streaming processors and 4 GByte memory.
System B: An AMD Radeon HD 6990 graphics card containing two GPUs, each with 1536 streaming processors and 1 GByte memory.

In all our experiments, we include the time of data transfers to and from the GPU, i. e. the measured runtime consists of: 1) uploading the two input matrices to the GPU; 2) performing the actual matrix multiplication; 3) downloading the computed result matrix.

System A using one GPU. Figure 6 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). For detailed numbers, see Table 2.

Clearly, the naive OpenCL implementation and the implementation using the generic allpairs skeleton are the slowest, because both do not use the fast GPU local memory, in contrast to all other implementations.

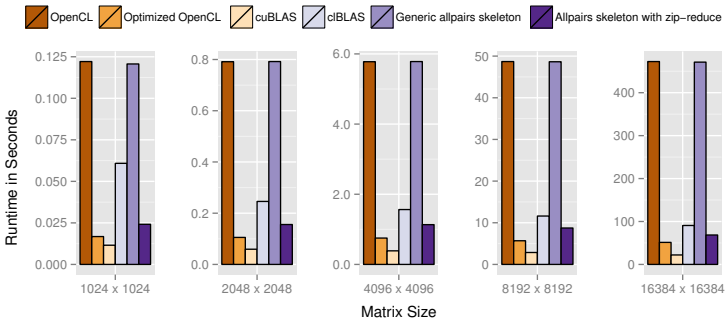

Fig. 6 Runtime of different matrix multiplication implementations on the NVIDIA system for different sizes for the matrices.

Implementation	Runtimes in Seconds				
	1024 ×1024	2048 ×2048	4096 ×4096	8192 ×8192	16384 ×16384
OpenCL	0.122	0.791	5.778	48.682	472.557
Optimized OpenCL	0.017	0.105	0.752	5.683	51.337
cuBLAS	0.012	0.059	0.387	2.863	22.067
clBLAS	0.061	0.246	1.564	11.615	90.705
Generic allpairs skeleton	0.121	0.792	5.782	48.645	471.235
Allpairs skeleton with zip-reduce	0.024	0.156	1.134	8.742	68.544

Table 2 Detailed runtime results for all implementations on the NVIDIA system.

The implementation using the allpairs skeleton customized with zip-reduce 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 can only be used 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 highly tuned specifically for NVIDIA GPUs using CUDA. The clBLAS implementation by AMD using OpenCL performs not as well: presumably, it is optimized for AMD GPUs and performs poorly on other hardware. Our optimized allpairs skeleton implementation outperforms the clBLAS implementation for all matrix sizes tested.

System B using one GPU. Figure 7 shows the runtime in seconds for five of the six implementations for different sizes of the matrices. Detailed numbers can be found in Table 3. We could not use the NVIDIA-specific cuBLAS implementation as it does not work on the AMD GPU.

For bigger matrices, the slowest implementations are, again, the unoptimized OpenCL implementation and the implementation using the generic allpairs skeleton.

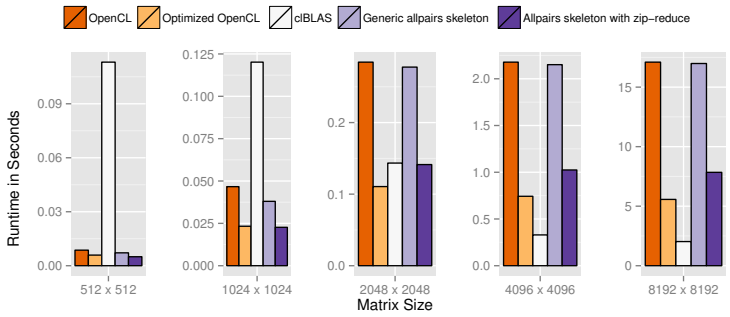

Fig. 7 Runtime of all compared implementations for a matrix multiplication on the AMD system using one GPU and different sizes for the matrices.

Implementation	Runtimes in Seconds				
	512 ×512	1024 ×1024	2048 ×2048	4096 ×4096	8192 ×8192
OpenCL	0.008	0.046	0.284	2.178	17.098
Optimized OpenCL	0.006	0.023	0.111	0.743	5.569
clBLAS	0.113	0.120	0.143	0.329	2.029
Generic allpairs skeleton	0.007	0.038	0.278	2.151	16.983
Allpairs skeleton with zip-reduce	0.005	0.023	0.141	1.025	7.842

Table 3 Detailed runtime results for all implementations on the AMD system.

The optimized OpenCL implementation and the allpairs skeleton customized with zip-reduce perform similarly. For matrices of size 8192×8192, the optimized OpenCL implementation is about 30% faster.

The clBLAS implementation performs very poorly for small matrices, but is clearly the fastest implementation for bigger matrices. Similar to the cuBLAS implementation on the NVIDIA hardware, it is not surprising that the implementation by AMD performs very well on their own hardware.

System A using multiple GPUs. Figure 8 shows the runtime behavior for both allpairs skeleton-based implementations when using up to four GPUs of our multi-GPU system. The other four implementations are not able to handle multiple GPUs and would have to be specially rewritten for such systems. We observe a good scalability of our skeleton-based implementations, achieving speedups between 3.09 and 3.93 when using four GPUs. Detailed numbers can be found in Table 4. For the matrices of size 16384×16384, performance is also provided in GFlops; to compute this value we excluded the data-transfer time (as usually done in related work) for a better comparison.

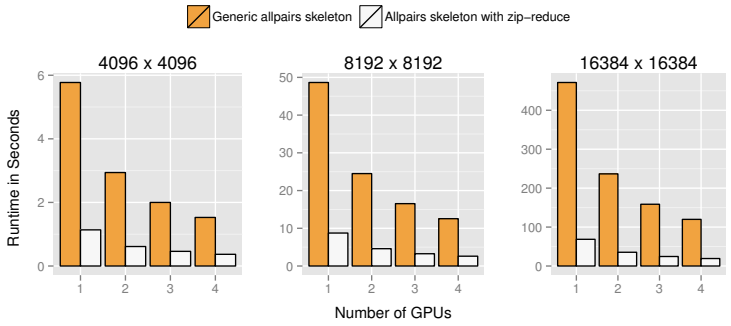

Fig. 8 Runtime of the allpairs based implementations using multiple GPUs.

Implementation	Number of GPUs	Runtimes in Seconds 4096 ×4096	8192 ×8192	16384 ×16384	GFlops 16384 ×16384
Generic allpairs skeleton	1 GPU	5.772	48.645	471.328	18.72
	2 GPUs	2.940	24.495	236.628	37.43
	3 GPUs	2.000	16.532	158.611	56.17
	4 GPUs	1.527	12.540	119.786	74.90
Allpairs skeleton with zip-reduce	1 GPU	1.137	8.740	68.573	130.93
	2 GPUs	0.613	4.588	35.294	262.18
	3 GPUs	0.461	3.254	24.447	392.87
	4 GPUs	0.368	2.602	19.198	523.91

Table 4 Detailed runtime of the allpairs based implementations using multiple GPUs. For the matrices of size 16384×16384 the results are also shown in GFlops.

6 Conclusion and Related Work

Considerable theoretical as well as practical research has been conducted in the field of algorithmic skeletons since its introduction in the late 1980s. Due to lack of space, we refer to [9] for an overview of skeletal programming and [8] for a recent survey of skeleton libraries for clusters and multi-core CPUs. Our contribution to skeletal programming is the introduction and efficient implementation of a new algorithmic skeleton for performing allpairs computations. As other skeletons, the allpairs skeleton can be used as a basic building block by application developers who do not have to be experts in GPU computing or parallel programming in general.

In previous work, efficient parallel implementations of allpairs computations on modern parallel processors were studied (e. g., multi-core CPUs [2], the Cell processor [18], and GPUs [3]) in the context of specific applications. In contrast to [16], which presents an efficient implementation scheme of allpairs computations for GPUs, we abstract the computation as an algorithmic skeleton and offer its efficient implementation to application developers as part of the SkelCL skeleton library.

The evaluation of the programming effort shows that the allpairs skeleton allows to express many applications considerably shorter and at a higher level of abstraction, as compared to using OpenCL or library implementations like BLAS. The performance comparison shows that by making information about the memory access pattern available to the implementation, we can considerably improve the performance by efficiently using the fast GPU local memory.

Several current approaches address simplifying GPU programming. As SkelCL, also SkePU [6] and Muesli [7] are skeleton libraries targeting multi-GPU systems. In contrast to our work, which is based entirely on the portable OpenCL, Muesli is implemented using NVIDIA's CUDA and SkePU is implemented with multiple back-ends which restrict the application developer to the back-ends' smallest common set of functions. While SkelCL can be used for programming multiple OpenCL-capable GPUs, the CUDA-based Thrust [10] library simplifies programming only for a single NVIDIA GPU.

In future work we will further study the allpairs skeleton, and explore more real-world applications, like n-body simulations, to evaluate the generality and applicability of the allpairs skeleton. Currently the implementation of the specialized allpairs skeleton in SkelCL is customized with two skeletons: one zip and one reduce skeleton. In future work we want to relax this and allow for arbitrary nesting of skeletons in SkelCL.

Acknowledgements

We thank the anonymous reviewers for their valuable comments and NVIDIA for donating hardware.

References

1. AMD (2013) Accelerated Parallel Processing Math Libraries (APPML). URL http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-math-libraries/
2. Arora N, Shringarpure A, Vuduc R (2009) Direct N-body Kernels for Multicore Platforms. In: Proceedings of ICPP '09, IEEE, pp 379–387
3. Chang D, Desoky A, Ouyang M, Rouchka E (2009) Compute Pairwise Manhattan Distance and Pearson Correlation Coefficient of Data Points with GPU. In: Proceedings of SNPD '09., IEEE, pp 501 –506
4. Cole M (2004) Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Computing 30(3):389–406, Elsevier
5. Daub C, Steuer R, Selbig J, Kloska S (2004) Estimating Mutual Information using B-Spline Functions – An Improved Similarity Measure for Analysing Gene Expression Data. BMC Bioinformatics 5(1):118
6. Enmyren J, Kessler C (2010) SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. In: Proceedings 4th Int. Workshop on High-Level Parallel Programming and Applications, pp 5–14, ACM
7. Ernsting S, Kuchen H (2012) Algorithmic Skeletons for Multi-core, Multi-GPU Systems and Clusters. International Journal of High Performance Computing and Networking 7(2):129–138, Inderscience
8. González-Vélez H, Leyton M (2010) A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. Software Practice & Experience 40(12):1135–1160, John Wiley & Sons
9. Gorlatch S, Cole M (2011) Parallel Skeletons. In: Padua D (ed) Encyclopedia of Parallel Computing, Springer, pp 1417–1422
10. Hoberock J, Bell N (2009) Thrust: A Parallel Template Library. URL https://developer.nvidia.com/thrust
11. Kirk DB, Hwu WW (2010) Programming Massively Parallel Processors - A Hands-on Approach. Morgan Kaufman
12. Lämmel R (2007) Google's MapReduce programming model – Revisited. Science of Computer Programming 68(3):208–237, Elsevier
13. Munshi A (2011) The OpenCL Specification. Version 1.2
14. NVIDIA (2012) NVIDIA CUDA C Programming Guide. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide, Ver. 5.0
15. NVIDIA (2013) CUBLAS. URL http://developer.nvidia.com/cublas
16. Sarje A, Aluru S (2013) All-pairs Computations on Many-Core Graphics Processors. Parallel Computing 39(2):79–93, Elsevier
17. Steuwer M, Kegel P, Gorlatch S (2012) Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), IEEE, pp 1858–1865
18. Wirawan A, Schmidt B, Kwoh CK (2009) Pairwise Distance Matrix Computation for Multiple Sequence Alignment on the Cell Broadband Engine. In: Proceedings of ICCS '09, Springer, pp 954–963