This is the author version (final manuscript). The final publication, which appeared in
International Journal of Parallel Programming **44**(3):506–530, June 2016 (online: March 2015),
 DOI 10.1007/s10766-015-0357-6,
is available at Springer via `http://dx.doi.org/10.1007/s10766-015-0357-6`

Smart containers and skeleton programming for GPU-based systems

Usman Dastgeer · Christoph Kessler

Received: 11 August 2014 / Accepted: 7 March 2015

Abstract In this paper, we discuss the role, design and implementation of *smart containers* in the SkePU skeleton library for GPU-based systems. These containers provide an interface similar to C++ STL containers but internally perform runtime optimization of data transfers and runtime memory management for their operand data on the different memory units. We discuss how these containers can help in achieving asynchronous execution for skeleton calls while providing implicit synchronization capabilities in a data consistent manner. Furthermore, we discuss the limitations of the original, already optimizing memory management mechanism implemented in SkePU containers, and propose and implement a new mechanism that provides stronger data consistency and improves performance by reducing communication and memory allocations. With several applications, we show that our new mechanism can achieve significantly (up to 33.4 times) better performance than the initial mechanism for page-locked memory on a multi-GPU based system.

Keywords SkePU · Smart containers · Skeleton programming · Memory management · Runtime optimizations · GPU-based systems

1 Introduction

Skeleton programming [4] for GPU-based systems is increasingly becoming popular for mapping common computational patterns. Several skeleton libraries are especially written (from scratch) targeting GPU-based systems including SkePU [10, 6], SkelCL [24] and Marrow [20]. Moreover, many existing skeleton libraries, initially written for execution on MPI-clusters and/or multicore CPUs have been ported for GPU execution, such as FastFlow [12] and Muesli [11]. These libraries differ in their

U. Dastgeer · C. Kessler
PELAB, Dept. of Computer and Information Science
Linköping University
E-mail: <firstname>.<lastname>@liu.se

approach and feature offering but they all aim to provide performance comparable to hand-written code while requiring much less programming effort.

Providing high-level abstraction with good execution performance in a library requires special design consideration. The question comes down to what is exposed to the programmer and what is handled implicitly by the skeleton library. For example, the Marrow library exposes concurrency to the application program by executing skeleton calls asynchronously; it returns a handle which can be used to synchronize execution when needed. This allows Marrow to effectively overlap computation and communication from different skeleton computations. SkelCL makes data distribution explicit so that the application programmer can choose how to map a computation to the underlying computing platform.

Another important aspect in GPU computation is managing communication between CPU (main) memory and GPU (device) memory over PCIe interconnect. In Muesli, FastFlow, SkePU and SkelCL, skeleton calls can execute on a single or multicore CPU as well as on a GPU. Considering that CPUs and GPUs have separate physical memory, execution on a certain compute device may require transferring data back and forth to its associated memory if data is not already available in that memory. For example, in the following code,

```
// 1D arrays: v0, v1
skel_call(v0, v1); // 'v0' read, 'v1' read and written
```

if `skel_call` is executed on a GPU and vectors `v0` and `v1` are not already available in that GPU memory, they need to be transferred. One idea could be to assume that operands always reside in CPU main memory and any GPU execution would require transferring input data to the GPU memory and output data back to main memory. Although simple, transferring data back and forth each time could be sub-optimal in presence of multiple executions on GPU. Libraries such as SkePU implement a lazy memory copying mechanism where copies of operand data in GPU memory are tracked and modified data is copied back to CPU memory only when needed.

In this paper, we discuss the role, design and implementation of containers in the SkePU skeleton library that can be used to wrap operand data of skeleton calls. To the best of our knowledge, SkePU containers have the most advanced implementation available in any skeleton library for GPU-based systems considering memory management and synchronization capabilities. We discuss how SkePU containers provide memory management for operand data while providing a high-level interface (similar to C++ STL containers) to the application programmer. These SkePU containers encapsulate internal book-keeping information about the run-time state of the data, e.g. in which memory units, and where there, valid copies of the container's elements can be found, and all element access is mediated through the containers by suitable operator overloading. Beyond element lookup, the containers provide the following services: memory management, data dependence tracking and synchronization, and communication optimization. We refer to containers with such extended services performing automatic optimizations as *smart containers*. Following are the major contributions of our work:

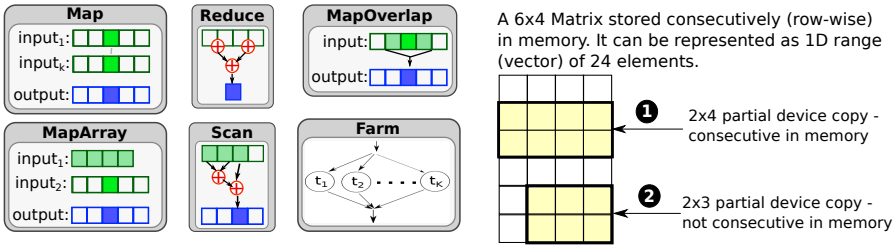

Fig. 1 Left: a simple illustration of SkePU skeletons, here for 1D operands. Right: partial copies of a matrix and their mapping to memory locations in the original matrix.

- We describe how smart containers can help skeleton libraries in providing high-level data abstraction while optimizing memory management and enabling asynchronous skeleton executions at runtime.
- We discuss the initial memory management implemented in SkePU containers [10] and its data consistency and performance limitations. We propose and implement a more efficient and robust memory management mechanism in SkePU containers, released open-source in SkePU v1.1 (May 2014)[1].
- We do performance evaluation with several applications to show benefits of smart containers as well as improvements of our new management mechanism inside SkePU container over the mechanism available in SkePU v1.0 from August 2012.

This paper is structured as follows: Section 2 introduces SkePU and describes how SkePU containers can optimize data transfers and can provide synchronization in presence of asynchronous skeleton calls. In Section 3, we describe our work on re-designing the memory management mechanism in SkePU containers. Evaluation is presented in Section 4 followed by related work in Section 5. Section 6 concludes and presents future work possibilities.

2 Smart containers in SkePU

SkePU [5] is a C++ template library for portable programming of GPU-based systems that provides a simple and unified programming interface for specifying data-parallel and task-parallel computations with the help of pre-defined skeletons including `map`, `reduce`, `maparray`, `mapoverlap`, `scan` and `farm` (see Figure 1(left)). All non-scalar operands of SkePU skeleton calls are passed in smart containers. For a more detailed description of SkePU, we refer to [5].

There are vector and matrix container classes in SkePU for representing 1D and 2D data respectively. Containers in SkePU are made generic in the element type using C++ templates and have interfaces similar to the C++ STL containers. For example, the SkePU vector container implements all operations supported by the STL vector container such as resize, erase, iterators etc., but it also implements several other functions related to memory management on different device memories such as updateHost, updateDeviceCopy and invalidateDeviceCopy. A SkePU container object

[1] SkePU is available at http://www.ida.liu.se/~chrke/skepu

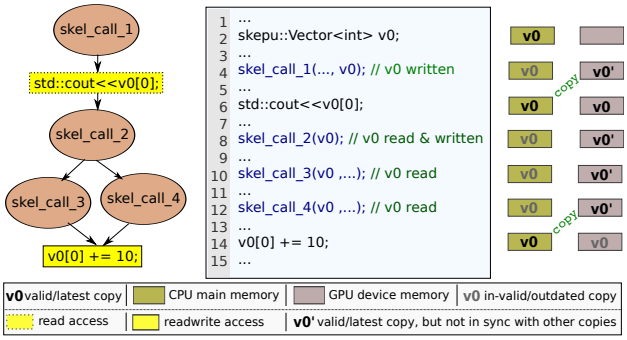

Fig. 2 Example for usage of smart containers with skeleton calls and other user code. The middle part shows a code scenario with four skeleton calls and one vector operand on a system containing 1 CPU and 1 CUDA GPU. The effect of each statement on the state of the data is also shown right for each statement, assuming the four skeleton calls are executed on the GPU. The left part highlights parallelism between different skeleton calls for asynchronous executions based on data dependencies.

internally keeps track of different (partial) copies of its data elements residing on different memory units with their content state (e.g., valid or invalid copy). By operator overloading (e.g. operator []), we ensure that data accesses on CPU side are handled in a consistent manner and necessary data transfers are made implicitly when required. Moreover, the read and write accesses to container data are distinguished by implementing proxy classes for element data in C++ [1]. This allows to differentiate between different types of accesses made to a container object, as shown below for a SkePU vector object v0:

```
skepu::Vector<int> v0(...);
v0[0] = 11; // write access to v0[0]
int a = v0[0]; // read access to v0[0]
v0[0] += 11; // readwrite access to v0[0]
```

In the following, we describe how usage of containers in a SkePU program yields performance benefits while providing a high level of abstraction.

Figure 2 depicts how usage of smart containers can help in optimizing communication across multiple skeleton calls as well as enable asynchronous skeleton call executions allowing exploitation of parallelism across multiple skeleton calls. The figure shows a simple scenario with four skeleton calls and one vector operand on a system containing 1 CPU and 1 CUDA GPU. When the vector container v0 is created, the payload data is placed in the main memory (*master copy*). Subsequently, depending on the skeleton calls using that data along their respective data access pattern (read, readwrite or write), other (partial) copies of operand data may get created in different memory units. In this case, we have CUDA device memory which is a separate physical memory. Assuming that all skeleton calls are actually executed on the GPU, the figure also shows the effect of each statement execution on the vector data state, i.e., creation/update/invalidation of data copies. As we can see, a SkePU smart container not only keeps track of data copies on different memory units but also helps in reducing the data communication between different memory units by delaying the communication until it becomes necessary. In this case, only 2 copy operations of data are made in the shown program execution instead of 7 copy operations which are required if one considers each skeleton call independently, as done in, e.g., Kicherer et al. [17, 18].

The first skeleton call (line 4) only writes the data (*v*0) and hence no copy is made. Instead, just a memory allocation is made in the device memory where data is written

by the skeleton call. After the completion of the skeleton call (line 4), the master copy in the main memory is marked outdated, which means that, in future, any data access to this copy would first require an update of this copy with the contents of the latest copy. The next statement (line 6) is actually a read data access from main memory. As the master copy was earlier marked outdated, a copy from device memory to main memory is implicitly invoked before the actual data access takes place. This is managed by the container in a transparent and consistent manner without requiring any user intervention. The copy in the device memory remains valid as the master copy is only read. Next, we have a skeleton call (line 8) that both reads and modifies $v0$. As we assume execution of all skeleton calls on the GPU in this scenario, the up-to-date copy already present in the device memory is read and modified. The master copy again becomes outdated. Afterwards, we have two skeleton calls (line 10 and 12) that both only read the data. Executing these operations on the GPU means that no copy operation is required before or after the skeleton call. Finally the statement in line 14 modifies the data in main memory so data is copied back (implicitly) from the device memory to the main memory before the actual operation takes place. Afterwards, the copy in the device memory is marked outdated.

In previous work [7] we have implemented support for the StarPU runtime system in the SkePU skeleton library, which allows execution of SkePU skeleton calls as asynchronous runtime tasks. This means that, when used with the StarPU runtime system, all four skeleton calls are executed asynchronously as tasks. The runtime system can infer data dependencies between different submitted tasks (i.e., skeleton calls in our case) based on their operands, and can run independent tasks concurrently. In the application program, the execution looks no different to the synchronous execution as data consistency is ensured by the smart containers. Blocking is implicitly established for a data access from the application program to data that is still in use with asynchronous skeleton invocations made earlier (with respect to program control flow) than the current data access.

3 Memory management in smart containers

In this section, we discuss the memory management that is implemented inside the SkePU containers. We discuss the initial memory management mechanism and its limitations with respect to data consistency and performance issues. Afterwards, we discuss the new, improved mechanism that we have developed as part of this work.

The idea of having multiple (also, partial) copies for a single container object is common in both (memory management) mechanisms described later. However, in the initial mechanism, copies of an object are represented as a 1D (one-dimensional) range for both SkePU vector and matrix containers. Although the contents of a SkePU matrix is stored consecutively (row-wise) and can be represented as a 1D range, partial (device) copies of a matrix object may not always map to consecutive 1D (sub-)ranges in the matrix. This is shown in Figure 1(right) with two partial device copies representing two sub-matrices. The first partial device copy maps to elements that are stored consecutively in the original matrix and thus can be represented as 1D range. Partial copies that do not map to consecutive elements (such as device copy 2 in Fig-

ure 1(right)) are not possible in the initial mechanism. In the new mechanism, we have solved this problem by defining a separate mapping for matrix operands which allows partial copies that do not map to consecutive elements in the original matrix. In the following, we use the 1D range mapping for explaining the working of both mechanisms. The 1D mapping is used in the initial mechanism for both vector and matrix containers, and for vector containers only in the new mechanism. Later, we describe how the new mechanism generalizes this mapping to represent partial copies with non-consecutive elements for matrix containers.

For explanation, suppose that we have a SkePU container object O that could be a vector or matrix object. One main-copy of object O, called *mainCopy(O)*, is created in the main memory when the object is allocated (i.e., when a constructor for object O is called in C++). This is the only copy in main memory created for that container object and all accesses in CPU code (including C++/OpenMP skeleton implementations and other program accesses) are directed to this copy. It covers the complete contents of object O (i.e., *mainCopy(O)* has elements with *index range* $[0,N)$ (first inclusive)[2] if the size of object O is N where $N \geq 1$) and remains allocated as long as the object is alive. Whenever a container object is resized, i.e., N is changed, all other copies on device memories are deleted and the size of the main-copy is adjusted accordingly.

The communication bandwidth from device to host (DTH) and from host to device (HTD) (roughly) doubles if the main-copy of a container object in CPU memory is allocated as page-locked (pinned) memory. As page-locked memory is always stored in physical CPU memory (i.e., RAM) and cannot be swapped out to disk, it can be transferred via DMA without requiring any intermediate buffers [21]. Both SkePU containers support page-locked memory allocation for their payload data, controlled via a simple flag.

During the program execution, other (partial or full) copies of an object may get created, used and later destroyed in different device memories depending on the object usage with skeleton calls in a program. Let K ($K \geq 0$) denote the number of device memory copies for object O at any given time, which are denoted by $dCopy_i(O)$ where $1 \leq i \leq K$. A device copy $dCopy_i(O)$ of the object O is identified by $\langle devID_i, offset_i, n_i \rangle$ where $devID_i$ is an identifier of the device on which the $dCopy_i(O)$ is present, $offset_i$ is the index of the first element of $dCopy_i(O)$ with respect to the index range of *mainCopy(O)*[3], and n_i is the number of elements stored in the copy; i.e., $dCopy_i(O)$ covers O elements with the index range $[offset_i, (offset_i + n_i))$ on device $devID_i$. Furthermore, each device copy $dCopy_i(O)$ has a *modified* flag which is set when contents in that copy is modified (e.g., by a GPU skeleton implementation). The modified contents of this copy is eventually copied back to the main-copy (and possibly other device copies) as described later.

Partial copies of a container object can exist in device memories for two reasons. First, an application may have skeleton operations on parts of a container object. Both vector and matrix containers support iterators [1] which allow us to call an operation on a subset of the elements as illustrated below:

[2] We use the standard set notation for representing a set of consecutive indices as index range. In set notation, $[a,b)$ is the half-open interval $\{x | a \leq x < b$ and x is a positive integer$\}$ [2].

[3] The *offset* would be 5 if $dCopy_i(O)$ starts from the 6th element of *mainCopy(O)*, i.e., the indexing is zero-based as per C++ style.

	CPU (main memory)	GPU (device memory)
Copies	One *main-copy*	Zero or more *device-copies (dCopies)* A device-copy (*dCopy*) identified by <dev*ID, offset, n*>

Read access	Copy all modified device copies back to *main-copy* and clear their *modified* flags
	Read *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		Copy all modified copies back to *main-copy* and clear their *modified* flags
		Read [offset, offset+n) from *main-copy* into *dCopy*
		end if
		Read the *dCopy*

Write access	Copy all modified device copies back to *main-copy* and clear their *modified* flags
	Remove all device copies
	Write *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		endif
		Set *modified* flag for the *dCopy*
		Write to the *dCopy*

Read/ Write access	Copy all modified device copies back to *main-copy* and clear their *modified* flags
	Remove all device copies
	Read/Write *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		Copy all modified copies back to *main-copy* and clear their *modified* flags
		Read [offset, offset+n) from *main-copy* into *dCopy*
		end if
		Set *modified* flag for the *dCopy*
		Read/Write the *dCopy*

Fig. 3 The initial memory management mechanism for a single container object.

```
skepu::Vector<int> v0(10); // create vector (v0) of 10 elements
...
skel(v0); // readwrite all 10 elements of v0
skel(v0.begin()+5, vo.end()); // readwrite last 5 elements of v0
skel(v0.begin()+3, v0.end()-3);// readwrite elements [3-7) of v0
```

If we execute the three skeleton calls (listed in the above code) on a single GPU, three device copies of `v0`, with index ranges $[0,10)$, $[5,10)$ and $[3,7)$ respectively, would get created[4]. Secondly, partial copies are created if container objects are passed as arguments to a skeleton call that is executed on multiple GPU devices in parallel. Dividing the computation work of data-parallel skeletons across different GPU devices divides the container objects, passed as operands, into chunks.

SkePU containers are primarily designed for passing operand data to skeleton calls and are not designed to be thread-safe. This means that concurrent memory requests to overlapping elements in a container object are not handled in the memory management mechanisms. We will discuss this in more detail in Section 3.4.

The concept of having one main-copy and potentially many overlapping device copies is the same in both the initial and the new memory management mechanism. However, the main difference comes with how read and write accesses to these copies are handled, as described below.

[4] As a device copy ($dCopy_i$) is identified by a tuple $\langle devID_i, offset_i, n_i \rangle$, two or more device copies of an object O with different starting elements (*offset*) and/or number of elements (n) are considered different copies even on the same device.

3.1 Initial mechanism

Figure 3 summarizes the initial memory management mechanism in SkePU. Before any element of *mainCopy(O)* is read in main memory, all modified device copies (i.e., device copies with *modified* flag set) of the object are copied back to *mainCopy(O)*. Whenever a modified device copy is copied back to the main-copy, its modified-flag is cleared. When writing to *mainCopy(O)*, all device copies of the object are removed (deallocated), i.e., only the main-copy remains. Obviously, one can think about a more fine-grained control where only a chunk of data is updated instead of everything; we have implemented this as improvement to the existing mechanism (more on this in the next section).

Copies on device memories are created, if not present already, when a container is used as operand with a skeleton call that is executed on a GPU. GPU read and write accesses[5] for an object are identified by $\langle devID_{acc}, offset_{acc}, n_{acc} \rangle$ and handled as follows. At a GPU read access $\langle devID_{acc}, offset_{acc}, n_{acc} \rangle$ to object O, a lookup is made to find an existing copy ($dCopy_j(O)$) with same index range on the given device. If an existing copy is found, it is used; this leads to consistency problems if the contents of that copy is not the most recent one, as we will discuss later. If no existing copy is found, multiple operations are carried out: (1) space for a new device copy is allocated, (2) contents from all modified device copies is copied back to *mainCopy(O)* and (3) required contents from *mainCopy(O)* is then copied to the newly created copy. Similar operations are carried out for a GPU write access to object O except that no data is copied (in any way) when a new device copy is allocated. Moreover, the *modified* flag for the copy written (no matter if newly created or an existing one was found) is set.

When creating a new device copy, the *modified* flag is cleared by default. It is set for a device copy whenever contents in that copy is changed. Note that we track content modifications for a complete device copy of a given size. This is because the elements in a device copy exactly map to the elements that are requested for a GPU read/write access.

3.2 Limitations of the initial mechanism

There exist both performance and data consistency issues with this mechanism. When a device copy of a container is modified, other (possibly overlapping) device copies of the same container on the same and other devices are neither marked invalid nor are they removed. A future access to those device copies would access stale data and thus result in data consistency issues. This could happen in both single and multi-GPU based systems. In the following, we discuss two example scenarios where overlapping copies of data in the same or different device memories result in data consistency and performance issues[6].

[5] The access mode is implicitly given by the operand's position in the skeleton call.

[6] There exist several other possible scenarios where data consistency and performance issues can arise especially when considering multiple GPUs.

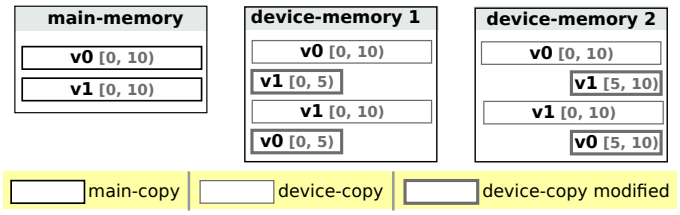

Fig. 4 Main and device memory copies of two vector objects after execution of the first loop iteration in Listing 1 on a 2-GPU system.

Scenario one: In the first scenario, we consider three skeleton calls, all executed on a single GPU while operating on different (overlapping) parts of a container object.

```
skepu::Vector<int> v0(10);
map(v0, ...); // read all 10 elements of v0
map(..., v0.begin()+5); // write last 5 elements of v0
map(v0, ...); // read all 10 elements of v0
```

For the first skeleton call, space for a device copy of vector `v0` is allocated for 10 elements on GPU and data is copied from the main-copy to this new device copy. Later, space for another device copy is allocated for the second skeleton call, mapping to the second half of the `v0` vector object. As the copy is written, no copying of contents takes place; rather, the *modified* flag for this newly written device copy is set. At this time, two device copies of `v0` exist, one covering full contents of `v0` and one covering the second half of `v0` but with newly written contents. Now, for the third skeleton call executing on the same GPU, an existing device copy (created for the first skeleton call) is found and used. However, the contents of this device copy is stale for the second half which was overwritten in the device copy created in the second skeleton call. Correct working of the above source code would require `v0.flush()` between the second and third skeleton call, which discards all devices copies and is highly inefficient fix in this situation.

Scenario two: When executing the computational loop (shown in Listing 1) on a single GPU-based system, we do not need any explicit synchronization as there exist no partial device-copies of data, i.e., only one device copy for each entire vector object is created in the GPU device memory. However, when executing on two GPUs, Fig-

```
1  skepu::Vector<...> v0(10);
2  skepu::Vector<...> v1(10);
3  ...
4  loop(...)
5  {
6    maparray1(v0, v1, v1); // read v0 - readwrite v1
7    ...
8    maparray2(v1, v0, v0); // read v1 - readwrite v0
9  }
10 ...
```

Listing 1 Pseudo-code with two MapArray skeleton calls in a loop.

ure 4 shows the memory state of both device memories as well as of main memory after execution of the first loop iteration. Initially, the main-copies of `v0` and `v1` are created in main memory (Line 1 and 2 respectively in Listing 1). When executing the first skeleton call on two GPUs, the work is divided by partitioning the second input and the output vector (`v1`) between two device memories; the first input vector (`v0`) is replicated on both GPUs for reading purpose, as per the MapArray skeleton semantics[7]. Similar things happen for the second skeleton call except that `v1` is replicated and `v0` is partitioned across the two device memories.

The consistency problem arises with the second loop iteration as stale (full device) copies of `v0` and `v1` are read in the first and the second skeleton call respectively. This is because the updated data from partial device copies is never copied to these full device copies of the same vectors. The obvious fix to this scenario would be to `flush` both vectors after each loop iteration, but this is too inefficient and an overkill as it copies all modified contents back to memory as well as removes all device memory copies. Having it inside a loop means that this communication overhead would be incurred for each loop iteration. Moreover, the problem is too specific as it happens only when executing the skeleton calls on multiple GPUs and `flush`ing vectors becomes unnecessary in other execution scenarios.

Remarks: Clearly, the initial memory management mechanism implemented inside the SkePU containers has memory consistency issues with execution on single and multi-GPU systems. Although it is possible to ensure program correctness, the fixes are normally an overkill and can hurt performance by increasing the communication volume. Furthermore, whenever contents of a container object in the main-copy is modified, all device copies are removed (i.e. device memory allocations are freed), as in the following example:

```
// v0 with multiple device copies...
v0[5] = 3; // removes all device copies
map(v0); // allocate device copy again if executed on a GPU...
```

Any further usage of the container object for GPU execution would require memory re-allocation on that GPU device memory. Marking device copies invalid instead of removing them could be a better idea considering that the device memory allocation is a blocking operation and can prevent asynchronous GPU execution [21]. Multiple device accesses with different offset and/or size information result in different device copies even in the same device memory. This could potentially lead to a situation where the device memory is exhausted and no space in device memory remains available for future allocations. The initial mechanism does not handle this situation explicitly. Last but not the least, contents of a device memory is always copied from the main-copy that exists in main memory. One might think about scenarios where (part of) the contents can be copied from other existing copies of the same object in the same device memory. Next, we present our work on improving the memory management mechanism implemented in SkePU containers to tackle these data consistency and performance issues in an implicit and efficient manner.

[7] *MapArray* (v_1, v_2, r) is a data-parallel skeleton where each element of the result vector, $r[i]$, is calculated as a function of the corresponding element of one of the input vectors, $v_2[i]$ and any number of elements from the other input vector v_1.

3.3 New mechanism

We replaced the initial mechanism in order to provide a more robust and efficient memory model for multi-GPU skeleton execution. The basic concept of having one main-copy and possibly multiple device memory copies (identified by $\langle devID, offset, n \rangle$[8]) of each object is the same. However, we introduce a more robust mechanism for updating the state of the different copies when reading and writing contents in a copy. A *valid* flag for each copy (including the main-copy) is added which specifies whether the contents of a given copy is valid for reading purpose or not. If the *valid* flag of a copy is not set[9], a read operation on that copy would first require copying updated data from either other device memories with valid contents or from main memory, as described later. Furthermore, in order to control the potential problem of running out of device memory space by creating too many device copies, each device copy has a *lastUsed* attribute which tracks the time when that device copy was last accessed (read and/or written). When not enough space is available for allocating a new device copy, one or more least recently used (LRU) device copies can be deallocated to make space for a new device copy.

Figure 5 summarizes the new memory management mechanism. In the new mechanism, we identify a read and write access on CPU also by an index range. By default, the index range for an object covers all of its elements (i.e., $[0,N)$ for an object of size N); however, it could be a smaller range, e.g., when a skeleton call executes on CPU with subset of elements such as:

```
skepu::Vector<int> v0(10);
map(v0.begin()+5, v0.end()); // index range [5, 10)
```

or when a specific element of object is accessed/modified on CPU:

```
v0[3] = ...; // index range [3, 4)
```

In the new mechanism, most operations are carried out considering whether the index ranges marked by two object copies (or accesses) overlap. The overlap between two index ranges, $[a,b)$ and $[c,d)$, can be determined by taking the intersection ($\cap$) of both ranges, using the standard set notation [2], i.e.:

$$\text{overlap if } [a,b) \cap [c,d) \neq \emptyset\text{; no overlap otherwise.}$$

Read and write accesses to the main-copy of an object with a given index range are handled in the following manner. If the *valid* flag of the main-copy is set, any read operation to it can proceed without any further processing. Otherwise, before an index range $[P,Q)$ of *mainCopy(O)* is read in main memory, all (if any) modified *overlapping* device copies of the object O are copied back to *mainCopy(O)*. The *valid*

[8] For simplicity of presentation, we consider here the vector case. Device copies of matrix objects are identified with a different tuple in the new mechanism, as discussed later.

[9] This test of the *valid* flag is the main source of overhead added by the coherence mechanism at main memory accesses by non-skeleton (CPU) code to the container. SkePU also provides customized variants of the container access operations that skip this test, which can be used to eliminate the overhead where the main copy of the container is statically known to be valid, e.g. in several subsequent accesses. The containers also offer an explicit main memory update (flush) operation `v.updateHost()` that can be used together with the customized access variants.

	CPU (main-memory)	GPU (device memory)
Copies	One *main-copy*	Zero or more device-copies(*dCopies*) A device-copy (*dCopy*) identified by <devID, offset, n>

Read access [P - Q)	**if** *main-copy* NOT valid **then**
	Copy all modified device copies overlapping with range [P, Q) back to *main-copy* and clear their *modified* flags.
	Set *valid* flag of *main-copy* if no other modified device copy exist.
	endif
	Read *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		endif
		if *dCopy* is not valid **then**
		Copy data from other valid copies
		Set *valid* flag for the *dCopy*
		end if
		Update *lastUsed* flag of the dCopy
		Read the *dCopy*

Write access [P - Q)	**if** *main-copy* NOT valid **then**
	Copy all modified device copies overlapping but not subset of range [P, Q) back to *main-copy* and clear their *modified* flags
	Set *valid* flag of *main-copy* if no other modified device copy exist.
	endif
	Clear *valid* flag for all valid device copies overlapping with range [P, Q)
	Write to *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		endif
		Clear *valid* flag for all overlapping copies
		Set *modified* flag for the *dCopy*
		Update *lastUsed* flag of the dCopy
		Write the *dCopy*
		Set (if not already) *valid* flag for the *dCopy*

Read/ Write access	**if** *main-copy* NOT valid **then**
	Copy all modified device copies overlapping with range [P, Q) back to *main-copy* and clear their *modified* flags
	Set *valid* flag of *main-copy* if no other modified device copy exist.
	endif
	Clear *valid* flag for all device copies overlapping with range [P, Q)
	Read/Write *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		endif
		if *dCopy* is not valid **then**
		Copy data from other valid copies
		Set *valid* flag for the *dCopy*
		end if
		Clear *valid* flag for all overlapping copies
		Set *modified* flag for the *dCopy*
		Update *lastUsed* flag of the dCopy
		Read/Write the *dCopy*

Fig. 5 The working of the new memory management mechanism for a single container object.

flag of the main-copy is set if all modified copies have been written back. Notice that only contents of the modified device copies that overlap with the index range of the read access are copied back to the main-copy. This means that the main-copy can still be marked invalid (after the copying) if there exist some modified device copies that do not overlap with the index range required by the current read access and are thus not copied back. In any case, the read access can continue knowing that at least the contents required for the access is up to date in the main-copy. Similarly, before a write access of index range $[P,Q)$ to *mainCopy(O)*, all modified *overlapping* device copies of the object O, whose index range is not a subset of the index range requested in the write access, are copied back. After copying contents appropriately, all *overlapping* device copies of the object are marked invalid.

Note that clearing the *valid* flag for a device copy (i.e., marking it invalid) also clears its *modified* flag if set. When writing contents in the main-copy of an object, the device copies of the object are marked invalid (i.e., their *valid* flags are cleared) but space is not deallocated. Device copies are only removed and space is deallocated when an object is either destroyed or when it is resized or when no space is available for a new device copy allocation (least recently used device copies of an object are

removed in that case). This could yield significant savings in memory (de-) allocation overhead as we will discuss later.

When a GPU read access to object O occurs, a lookup is made to find an existing copy ($dCopy_j(O)$) with same index range on the given device. If an existing copy is found and its contents is valid, it is used; otherwise, if the contents of the found copy is not valid, it is copied from other copies in a similar manner as for a newly created device copy (described below). If no existing copy is found, space for a new device copy is allocated; afterwards, contents from other copies are copied to the newly created device copy in the following manner.

In the new mechanism, contents to a device copy can be copied from other device copies of the object (present in the same or other device memories) as well as from the main-copy. However, when copying from other device memory copies, it might happen that those copies only have partial overlap with what is required. For example, if we need contents for a device copy in index range $[0,10)$ and we have two valid device copies of the same object with index range $[0,7)$ and $[5,10)$ respectively, then we need two content copies of index subranges, e.g., $[0,7)$ and $[7,10)$, to copy all contents. Theoretically, we could require from one copy operation (in the best case) up to potentially as many as the size of the index range (n_{acc}) to which we need to copy. However, in practice, the index subranges that occur in skeleton programs are very few (normally up to the total number of GPU devices) as we will see later.

The pseudocode of the algorithm for copying data to a device copy ($dCopy_{dst}$) is outlined in Algorithm 1. The algorithm for copying data to a device copy builds a *copy plan* for copying contents from multiple sources in the following precedence order:

1. Copy contents from overlapping valid copies in the same device memory. This is the fastest copy option as it happens within the same device memory.
2. Copy contents from the main-copy if it is valid. No further copy should be needed as the main-copy has the full contents of an object available.
3. If GPU devices support direct data transfer between their memories (e.g., using GPUDirect [23]), copy contents from valid copies in other device memories to the current copy using this feature.
4. Copy contents from modified device copies that exist in other device memories back to the main-copy and then copy required contents from the main-copy.

The *getValidDevCopy()* method in Algorithm 1 finds a valid device copy of an object, on a given device, that has an index range overlapping with the specified index range. The *getModifiedDevCopy()* method does the same except that it looks for a modified device copy. When copying contents from another device copy ($dCopy_{src}$), the *getOverlapRange()* and *getRemainingRanges()* methods find out what index range can be copied from a given device copy ($dCopy_{src}$) and what remaining index ranges (if any) need to be copied from other sources[10]. For example, when copying the $[0,10)$ index range to a device copy ($dCopy_{dst}$), we might find a $dCopy_{src}$ with contents $[3,17)$; *getOverlapRange()* would return the overlap between the two index ranges, i.e. $[3,10)$ in this case. As the overlap is not complete, copying elements

[10] There could be at most two new index ranges returned by *getRemainingRanges()* considering that each object copy stores consecutive elements, marked by an index range.

Algorithm 1: UpdateADeviceCopy

```
Input: dCopy_dst, device copy to be updated
       O, object to which dCopy_dst belongs
1  begin
2      Init rangesToCheck and copyInfPerRange
3      range_dst ← [offset_dst, (offset_dst + n_dst))
4      rangesToCheck.add(range_dst)
5      while not rangesToCheck.empty() do
6          range ← rangesToCheck.pop()
7          dCopy_src ← getValidDevCopy(O, range, devID_dst)
8          if dCopy_src then
9              subrange ← getOverlapRange(dCopy_src, range)
10             copyInfPerRange.push(dCopy_src, dCopy_dst, subrange)
11             if range ≠ subrange then
12                 rem ← getRemainingRanges(range, subrange)
13                 rangesToCheck.push(rem)
14             go back to start of while loop
15         if mainCopy(O).valid then
16             copyInfPerRange.push(mainCopy(O), dCopy_dst, range)
17             go back to start of while loop
18         foreach dev ∈ AllDevices do
19             if dev ≠ devID_dst then
20                 dCopy_src ← getValidDevCopy(O, range, dev)
21                 if PeerCopyEnabled(devID_dst, dev) ∧ dCopy_src then
22                     subrange ← getOverlapRange(dCopy_src, range)
23                     copyInfPerRange.push(dCopy_src, dCopy_dst, subrange)
24                     if range ≠ subrange then
25                         rem ← getRemainingRanges(range, subrange)
26                         rangesToCheck.push(rem)
27                     go back to start of while loop
28         foreach dev ∈ AllDevices do
29             if dev ≠ devID_dst then
30                 dCopy_src ← getModifiedDevCopy(O, range, dev)
31                 if dCopy_src then
32                     subrange ← getOverlapRange(dCopy_src, range)
33                     range_src ← [offset_src, (offset_dst + n_src))
34                     copyInfPerRange.push(dCopy_src, mainCopy(O), range_src)
35                     copyInfPerRange.push(mainCopy(O), dCopy_dst, subrange)
36                     if range ≠ subrange then
37                         rem ← getRemainingRanges(range, subrange)
38                         rangesToCheck.push(rem)
39                     dCopy_src.modified ← false
40                     set mainCopy(O).valid flag if no modified copy exists
41                     go back to start of while loop
42         // copy from main-copy (last option)
43         copyInfPerRange.push(mainCopy(O), dCopy_dst, range)
44     foreach copyInf ∈ copyInfPerRange do
45         // do the actual copying using e.g., cudaMemcpy etc.
46     dCopy_dst.valid ← true
```

$[3,10)$ from $dCopy_{src}$ would yield one remaining index range ($[0,3)$) to copy further. In a nutshell, considering that the contents of a copy might need to be copied from more than one source, the copy plan includes the ⟨source, destination, index range⟩ information of all the necessary copy operations to copy the required contents. These copies are carried out by the system considering possible asynchronous execution semantics offered by modern GPUs; the *valid* flag of the device copy (the one being written to) is set in the end.

When a GPU write access, identified by $\langle devID_{acc}, offset_{acc}, n_{acc} \rangle$ for an object O occurs, it is handled in the following manner. First, a lookup is made to find an existing copy with same index range on the given device. If not found, a new device copy is allocated for the given index range on the given device. Before the actual writing of contents in the current device copy can take place, all its overlapping valid (main-copy and all other device) copies are marked invalid. The *valid* flag for the main-copy and all other device copies that are overlapping with the currently written device copy ($dCopy_{acc}$) is cleared. When marking other device copies invalid, the framework checks for copies that have modified contents and whose index range is not a proper subset of the index range currently written. For example, if we write to index range $[0,10)$ and another device copy with modified contents has index range $[5,15)$, partial contents of the latter copy not overwritten by the current write operation (i.e., $[10,15)$) is copied back to the main-copy before its *modified* and *valid* flags are cleared. The pseudocode for handling a write access to a device copy is shown in Algorithm 2.

Algorithm 2: UpdateOthersWhenWritingADeviceCopy

```
   Input: dCopy_w, device copy that is written
          O, object to which dCopy_w belongs
1  range_w ← [offset_w, (offset_w + n_w))
2  begin
3      if mainCopy(O).valid then
4          mainCopy(O).valid ← false
5      foreach dev ∈ AllDevices do
6          dCopies ← getValidDevCopies(O, range_w, dev)
7          foreach dCopy_i ∈ dCopies do
8              if dCopy_i ∧ dCopy_i ≠ dCopy_w then
9                  if dCopy_i.modified then
10                     range_i ← [offset_i, (offset_i + n_i))
11                     subrange ← getOverlapRange(range_i, range_w)
12                     if range_i ≠ subrange then
13                         rem ← getRemainingRanges(range_i, subrange)
14                         Copy rem from dCopy_i to mainCopy(O)
15                     dCopy_i.modified ← false
16                 dCopy_i.valid ← false
```

3.3.1 Matrix operands

As already discussed, SkePU matrix objects in the new mechanism are treated differently to enable partial device copies with contents not stored consecutively in the original matrix (see Figure 1(right)). This is done by identifying device copies of a $R \times C$ matrix by a tuple $\langle devID_i, offset_i, r_i, c_i \rangle$ where:

- $devID_i$ is an identifier of the device on which the device copy is present,
- $offset_i$ is the 1D index of the first element of the device copy with respect to the index range of *mainCopy(O)*,
- r_i is the number of rows stored in the device copy, i.e., $1 \leq r_i \leq R$,
- c_i is the number of columns stored in the device copy, i.e., $1 \leq c_i \leq C$.

This allows us to represent a device copy mapping to any submatrix in the original matrix, including the second device copy in Figure 1(right). Although a matrix device copy may represent multiple (disjoint) index ranges, it is still considered a single device copy in the working of the new mechanism. The process of marking copies valid, modified etc. remains the same as for vector objects. Contents in the device copy are internally stored consecutively no matter if it represents multiple disjoint index ranges. However, the difference comes in the implementation, e.g., when copying data to/from other copies, a copy operation may be needed for each index range. Also, the overlap between two device copies is checked for each index range that they contain and "no overlap" is defined when there exists no overlap with any index range covered by two device copies.

This submatrix approach for two-dimensional containers could also be generalized to higher dimensions. For practical reasons, SkePU currently only implements containers for one and two dimensions, though.

3.3.2 Remarks

The new mechanism is more powerful when it comes to managing read and write accesses to multiple overlapping copies of an object in different device memories. The consistency problems described with the initial mechanism, as depicted in the two scenarios earlier, are taken care of automatically in the new mechanism. This is achieved by tracking the state of each device copy and by proper handling of other overlapping copies when contents in one device copy is updated. From the programmer's perspective, the new mechanism does not require any new information and works transparently behind the generic container's interface. Besides addressing data consistency issues, the new mechanism improves performance first by not deallocating device copies whenever contents is modified in the main-copy. Moreover, copying data to a device copy from existing object copies in the same device memory can reduce the communication overhead associated with data transfers between main memory and GPU device memory. Partial device copies of a matrix object are identified with a different tuple that allows partial device copies representing any submatrix of the original matrix. Last but not the least, support for direct data transfers between two device memories can yield significant savings as only one copy operation is needed rather than two copy operations in the original mechanism (first from one device copy to main-copy and then from main-copy to another device copy).

The granularity of control is more flexible in the new mechanism. Generally, the granularity results from the operands being passed to/from the skeleton invocations, i.e., from the application itself. It can be at the level of entire vectors and matrices, or major or minor subarrays. Theoretically, the control could happen at individual element level if there exist device copies consisting of one element. However, we did not encounter such scenarios in the example applications that we implemented (see Section 4) where the total number of device copies was very small even if many invocations were made.

The framework, when managing communication between different copies, considers their index ranges and optimizes data transfers by transferring only required contents. Several optimizations are made in the new mechanism, e.g., a list of modified device copies per device memory is maintained for fast lookup.

The new mechanism is very similar to the MSI (Modified, Shared, Invalid) cache coherence protocol [9]. A copy can be in valid (shared in MSI), modified or invalid state. The main difference comes from the concept of (partially) overlapping copies where state of different copies is updated considering their potential overlap, allowing disjoint copies to be modified independently. For example, in our case, when a device copy is modified, there exists only one valid copy for the index range covered by that modified device copy. However, there can exist another modified device copy for the same object if the index ranges of both modified device copies do not overlap with each other.

In both initial and new mechanism, we match a device memory access $\langle devID_{acc}, offset_{acc}, n_{acc} \rangle$ to an *exact* copy. For example, if we have a device memory access request $\langle 0, 5, 5 \rangle$ (i.e., index range $[5, 10)$ on the device with ID 0) for a vector object, we will check for an exact copy of that vector object with index range $[5, 10)$ in that GPU device memory. If we do not find an existing copy, we will create a new copy for this index range. A possible optimization could be to look for an overlapping copy that totally includes the index range requested, i.e., covers an index range that is a superset of the requested range. However, this might complicate the handling of modifications and invalidations to device copies as partial contents in a device copy then might be in different states (valid, invalid, modified). Splitting a device copy into multiple *logical* sub-copies corresponding to different access requests might be an interesting idea and can be investigated in future.

3.4 SkePU program execution model

SkePU containers are primarily designed to pass operands to skeletons calls in a SkePU program. A typical SkePU program consists of sequential CPU code with skeleton calls nested in the program control flow. The sequential CPU part manages input/output operations and coordinates execution of different skeleton calls in the program. The actual computations are marked by skeleton calls in the program.

Considering the above sequential-style program execution scenario, the SkePU containers are not designed to be thread-safe, i.e., to be used in a concurrent environment outside the skeleton calls. For example, the behavior is undefined if overlapping contents in a vector object $v0$ are read and written simultaneously in multiple threads.

Considering our target usage scenarios, making containers thread-safe could hurt the performance. The main container operations are performed inside the skeleton implementations, which are designed to ensure proper behavior when using multiple threads. For example, the OpenMP implementations of the skeletons in the SkePU library ensure data consistency for the given skeleton semantics.

Outside the skeleton implementations, the application programmer needs to either use containers in the sequential source code or provide adequate synchronization when overlapping container data is read and written in a multi-threaded context.

Normally, skeleton calls in a SkePU program are blocking, i.e., control returns from a skeleton call when it is completed and operand data is safe for subsequent program accesses. However, SkePU skeleton calls can be non-blocking when using the StarPU runtime system, i.e., the control returns to the calling thread before the actual call is complete. As shown in Listing 2, the containers ensure proper synchronization for program accesses in this case by ensuring that skeleton calls operating on that data complete before those accesses can proceed. Details of the SkePU smart containers' interoperating with the StarPU runtime system can be found in Dastgeer [6, Ch. 4.4].

Although the containers are mainly designed for passing operands to SkePU skeleton calls, they can be used with other computations executing on either CPU or GPU. The containers' API used in the SkePU skeleton framework is generic and can be used in other contexts. This could be useful e.g. in programs where not all computations can be modeled with the existing set of skeletons present in SkePU today.

4 Evaluation

In this section we evaluate the effect of the memory access optimizations implemented inside the SkePU containers with the help of several applications/kernels implemented using SkePU skeletons. The evaluation is carried out on a GPU-based system with 2 Intel Xeon E5520 CPUs and 2 NVIDIA C2050 GPUs; C/C++ and CUDA code is compiled using GCC (v4.8.1) and NVIDIA C (nvcc v0.2.1221 with CUDA 5.0) compilers respectively. We carry out two kinds of evaluation: First, we check the benefits of the memory management implemented in SkePU by comparing

```
1 skepu::Vector<...> v0, v1;
2 ...
3 map(v0); // asynchronous Call 1
4 map(v1); // asynchronous Call 2, can overlap with call1
5 ...
6 v1[10] = 11; // blocks until Call 2 finished
7 ...
8 std::cout<< v0[i]; // blocks until Call 1 finished
```

Listing 2 Task-parallel execution of a SkePU program with the StarPU runtime system. Call 2 is not data dependent on Call 1 and thus the calls can overlap if resources are available.

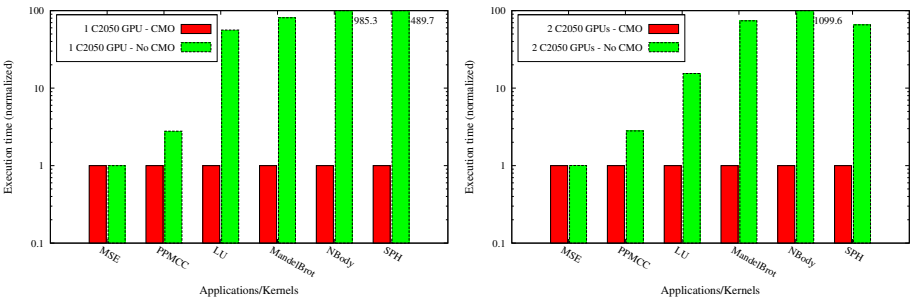

Fig. 6 Normalized execution times of different SkePU applications/kernels with one or more skeleton calls where each skeleton call is executed on 1 (left) or 2 C2050 GPUs. We compare execution where SkePU communication and memory optimizations (CMO) are enabled to where operand data of skeleton calls is transferred back and forth for each skeleton call (No CMO). The execution time is normalized with CMO execution as baseline. MSE is a single-call program (thus nothing to optimize), showing that the overhead of smart containers is negligible.

with executions where operand data to skeleton calls is transferred back and forth for each call. In the second evaluation, we compare our new memory management mechanism with the initial memory management mechanism found in the initial version of SkePU.

4.1 Effect of communication optimizations

We evaluate the benefits of our memory management inside the SkePU containers for four applications (NBody, LU factorization, Smooth Particle Hydrodynamics, Mandelbrot) and two image processing kernels (Pearson Product-Moment Correlation Coefficient and Mean Squared Error). We do the evaluation by comparing the execution time of these programs with the new memory management mechanism to their execution time where data is not kept on a device memory after a skeleton call execution, i.e., operand data is transferred for each skeleton call back and forth. The benefits of memory optimizations come with GPU execution where communication between different device copies and main-copy of an object can be optimized.

Figure 6 shows execution of several applications/kernels on two GPU setups (1 C2050 GPU, 2 C2050 GPUs where the work in each skeleton call is equally divided across the 2 GPUs that are of the same type). The execution time is averaged over multiple executions, with different problem sizes (both small and large), for each application/kernel. In both single and multi-GPU executions, we can see substantial benefit for memory optimizations for applications containing multiple skeleton calls. In some cases, speedup is more than 100 times which shows the importance of applying communication optimizations in GPU based systems across multiple skeleton calls. For applications such as NBody, passing operand data back and forth for each time step makes the communication really expensive even for a small simulation (50 timesteps for these experiments). The MSE kernel contains just one skeleton call and thus does no show any benefit as transferring data (back and forth) at least once is

required in any case. However, the MSE execution shows that the overhead of our memory management is less than 1%[11].

4.2 New versus initial mechanism

Earlier, we have seen the benefits of optimizing communication for operand data across multiple skeleton calls in the program. Now, we analyze the improvements of our new memory management mechanism in comparison to the initial mechanism for several applications. In the new mechanism, main improvements come from device copies of an object that (partially) overlap with each other. One possible way to create these partial copies is having one or more skeleton calls in the program that operate on partial elements of a container object, as shown in Scenario One (Section 3.2). However, none of the applications that have so far been ported to SkePU has such behavior. The other possibility that results in partial copies in device memory comes with multi-GPU execution where work in each (or some) skeleton call(s) is partitioned across different GPUs. This was shown earlier (Section 3.2) with two calls to the MapArray skeleton, and this is found in three of the applications (LU factorization, NBody and SPH) that we have ported to SkePU.

For demonstration, we consider execution of these three applications on the above mentioned GPU-based system. For each application, we do executions over multiple problem sizes with three configurations of memory management, as listed below:

1. *Initial CMO*: Execution with the initial SkePU memory management mechanism.
2. *New CMO*: Execution with our new memory management mechanism. Device-to-device memory transfers between the two peer GPU device memories (using GPUDirect [23]) are still disabled in this configuration. This means that communication between copies residing in the two device memories will happen indirectly (via the main memory where data is first transferred from one device memory to main memory and then from there to the other device memory).
3. *New CMO (peer)*: Execution with our new memory management mechanism. Device-to-device memory transfers between the two peer GPU device memories (using GPUDirect [23]) is enabled in this configuration.

The new mechanism transparently finds out (via the CUDA API) whether the peer device memory transfers can be enabled or not between different GPU device memories and can optimize communication patterns between such devices. Even if the peer device memory transfers are not available, the new mechanism still improves over the initial mechanism by making execution both consistent (without requiring any `flush` operation[12]) as well as reducing communication to main memory.

Table 1 describes the communication volume and device memory allocation size (in megabytes) for the three applications. The values are accumulated over multiple executions of each application with different problem sizes. For each application,

[11] In absolute terms, the overhead is less than 1 microsecond. It is measured by comparing execution of several kernels with different problem sizes.

[12] The `flush` operation is both expensive, as it deallocates all device copies of the object, and exposes data consistency issues to the application programmer.

Table 1 Communication volume and device allocation size (in megabytes) for the three applications, accumulated over different problem sizes. CMO=communication and memory allocation optimization by smart containers. HTD/DTD/DTH=host-to-device/device-to-device/device-to-host communication.

LU Factorization						
CMO Type	HTD	DTD	DTD (peer)	DTH	Total	Alloc
Initial CMO	119.8	0	0	60.1	179.9	359.1
New CMO	60.1	59.4	0	59.7	179.2	4.4
New CMO (peer)	0.6	59.4	59.4	0.3	119.7	4.4
NBody						
CMO Type	HTD	DTD	DTD (peer)	DTH	Total	Alloc
Initial CMO	3121.9	0	0	1135.2	4257.1	6243.9
New CMO	1040.6	1040.6	0	1087.9	3169.1	567.6
New CMO (peer)	0	1040.6	1040.6	47.3	2128.5	567.6
SPH						
CMO Type	HTD	DTD	DTD (peer)	DTH	Total	Alloc
Initial CMO	101.2	0	0	42.7	143.9	203.2
New CMO	33.7	33.7	0	34.1	101.5	4.0
New CMO (peer)	0	33.7	33.7	0.3	67.7	4.0

the table lists the size for different types of communication, namely, device to host (DTH), device to device (DTD), device to device between two peer device memories (DTD peer) and host to device (HTD) for each of the three configurations described above. For each application, the table also lists the total communication size (penultimate column) for each configuration as well as the accumulated size of device memory allocations (last column) made for different container objects used as operands with the skeleton calls.

Just by looking at the total communication size (second-last column) for each configuration, we can clearly observe the improvements of our new mechanism over the initial one even when the peer device memory transfers are not enabled. The savings are more vital than what one might expect from looking at the total communication size as, in the new mechanism, we do more DTD memory transfers rather than HTD each time. In the initial mechanism, there was no DTD data transfer as can be seen in the table. On modern GPUs (e.g., C2050s that we have used for these experiments), the achieved bandwidth for DTD communication is more than 25 times better than the achieved bandwidth for HTD communication, as found by our experiments on a C2050 GPU.

Enabling the peer device memory transfers, the total communication size reduces even further considering that data can be directly transferred between the two device memories without involving the main-copy that resides in the main memory. This obviously reduces the overall communication size by 50%. Furthermore, the achieved bandwidth for peer DTD device transfers is 1.5 to 3 times faster than for HTD. The ratio varies considering whether the HTD transfers are done for page-locked memory or for data with normal (default) allocations; communication with the former is faster than the latter.

Besides the differences in communication size and speed, there comes a substantial saving in the total number of device memory allocations made for the device copies of container objects. The differences in accumulated device memory allocation

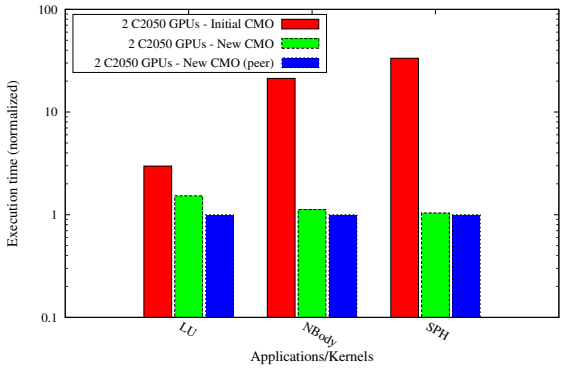

Fig. 7 Normalized execution time of the different memory management configurations for three SkePU applications on a 2-GPU system, where the main-copy of each object is allocated as page-locked/pinned memory. The execution time is normalized considering *New CMO (peer)* as baseline. The average speedup over the three applications due to using the new smart containers instead of the initial ones (Initial CMO) is 19.2; the average speedup due to using peer over non-peer communication with the new containers is 1.23.

size are `82`, `11` and `50` times for the LU, NBody and SPH applications respectively. This is because in the initial mechanism device memory allocations are removed for each write access to the main-copy as well as correct execution requires `flushing` contents between different skeleton calls. The savings in memory allocations are substantial and can have a major performance impact considering that these device allocations (and corresponding deallocations) are made during the actual skeleton program execution.

Figure 7 shows the execution times for the three configurations for each application, here with page-locked memory allocations. The savings are substantial; a speedup of up to 33.4 is observed in the best case (SPH), and on average over three iterative applications we achieved a 19.2 times reduction in execution time when comparing execution with the initial mechanism to the new mechanism for page-locked memory allocations.

The new mechanism reduces the total number of data transfers, localizes communication when possible (e.g., DTD instead of HTD) as well as cuts the total number of device memory allocations by 82 times in the best case. Besides making the application execution faster, these savings can have a huge impact on power consumption considering that memory operations consume 1 to 2 orders of magnitude more power than floating-point operations [16, 22]. From the programming perspective, a skeleton program can now be executed on different GPU-based systems without any data consistency issues.

5 Related work

SkelCL [24] is an OpenCL-based skeleton library that supports several data-parallel skeletons on vector and matrix container operands. The SkelCL vector and matrix containers provide memory management capabilities like the initial SkePU containers. However, unlike SkePU, the data distribution for containers is exposed to the programmer in SkelCL. Furthermore, SkelCL allows only one copy per device for a container object.

The Muesli skeleton library, originally designed for MPI/OpenMP execution [3], has been recently ported for GPU execution [11]. Memory management implemented inside Muesli containers is inspired by the initial memory management mechanism implemented in SkePU containers [10]. Moreover, it currently has a limited set of data-parallel skeletons which makes it difficult to port applications such as N-body simulation and Conjugate Gradient solver.

Marrow [20] is a skeleton programming framework for systems containing a single GPU using OpenCL. It provides data (map) and task parallel (stream, pipeline) skeletons that can be composed to model complex computations. It supports certain data type objects for storing buffer data but their exact implementation is not publicly known. Also, Marrow focuses on GPU execution only (i.e., no execution on multicore CPUs) and exposes concurrency and synchronization issues to the programmer.

NVIDIA Thrust [15] is a C++-template library that provides algorithms (reduction, sorting etc.) with an interface similar to C++ standard template library (STL). For vector data, it has the notion of `host_vector` and `device_vector` modeling data on host and device memory respectively; data can be transferred between two vector types using a simple vector assignment (e.g., `v0 = v1`).

For the functional data-parallel programming language SAC [13] there is a CUDA back-end with support for dynamically scheduled hybrid CPU-GPU and multi-GPU computing [8]. This implementation contains a form of smart containers that follows the MSI protocol. However, it only allows for 1D partitioning along the outermost array dimension and uses fixed-size data blocks as the data units of communication and invalidation in MSI, which currently are unit-sized blocks along the topmost dimension (i.e., individual vector elements or matrix rows), not larger index subspaces as in our case, thus leading to high overhead for large accesses where many blocks need to be copied or invalidated. Moreover, direct GPU-GPU communication is not supported in that work.

For a more detailed discussion of SkePU related work, we refer to [5] and [6].

On very recent and forthcoming GPUs that support a global address space atop a NUMA memory system (such as CUDA Unified Memory [14, 19] on recent Nvidia GPUs), the cost of data communication will not disappear. While a hardware-provided global address space might be more convenient in scenarios with irregular memory access patterns, skeleton computations are much more predictable and thus better suited for software-level optimizations of communication and memory allocation. From a performance portability perspective, we consider it actually better to use location-aware smart containers to control and avoid unnecessary data movement in software than to hide this cost-affecting property in the hardware's programming API. A software solution such as smart containers is also applicable to non-CUDA or older GPUs or to non-GPU accelerators, and thus benefits portability.

6 Conclusion and Future work

We have seen how smart containers can help in optimizing communication and reducing data transfers across multiple skeleton calls while exposing a high-level STL like interface to the application programmer. We also discussed data consistency and

performance issues associated with the initial SkePU memory management mechanism. A new memory management mechanism is designed and developed which gives consistent program execution with better performance (i.e., less communication and data (de-)allocations). Evaluation with several applications/kernels shows benefits for memory optimizations for GPU-based systems. The evaluation also shows that the new mechanism achieves a significant (up to 33.4 times) reduction in execution time when comparing execution with the initial mechanism for page-locked memory.

In future work, smart containers can be extended to internally adapt not only the storage location but also the storage format. This might include transposing a matrix as well as conversion between different storage formats for a sparse matrix.

Acknowledgements This research was partly funded by EU FP7 project EXCESS (www.excess-project.eu) and by SeRC (www.e-science.se) project OpCoReS.

References

1. Alexandrescu, A.: Modern C++ design. Addison-Wesley Professional; 1st edition (2001)
2. Aufmann, R., Barker, V., Lockwood, J.: Intermediate Algebra with Applications, Multimedia Edition. Cengage Learning (2008). URL `http://books.google.se/books?id=QYfJAxqwDE8C`
3. Ciechanowicz, P., Poldner, M., Kuchen, H.: The Münster skeleton library Muesli - a comprehensive overview (2009). ERCIS Working Paper No. 7
4. Cole, M.I.: Algorithmic skeletons: Structured management of parallel computation. MIT Press (1989)
5. Dastgeer, U.: Skeleton programming for heterogeneous GPU-based systems. Licentiate thesis. Thesis No 1504. Department of Computer and Information Science, Linköping University (2011). URL `http://liu.diva-portal.org/smash/record.jsf?pid=diva2:437140`
6. Dastgeer, U.: Performance-aware component composition for GPU-based systems. Ph.D. thesis, Linköping University (2014). URL `http://www.diva-portal.org/smash/record.jsf?pid=diva2:712422`
7. Dastgeer, U., Kessler, C., Thibault, S.: Flexible runtime support for efficient skeleton programming. In: Advances in Parallel Computing, vol. 22, pp. 159–166. IOS Press (2012). Proc. ParCo conference, Ghent, Belgium, Sep. 2011
8. Diogo, M., Grelck, C.: Towards Heterogeneous Computing without Heterogeneous Programming. In: H.W. Loidl, R. Pena (eds.): 13th Int. Symposium on Trends in Functional Programming (TFP 2012), St. Andrews, UK, Lecture Notes in Computer Science 7829, pp. 279–294, Springer, 2013
9. Dubois, M., Annavaram, M., Stenström, P.: Parallel Computer Organization and Design. Cambridge University Press (2012)
10. Enmyren, J., Kessler, C.: SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, USA, Sep. 2010. ACM.
11. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. International Journal of High Performance Computing and Networking **7**, 129–138 (2012)
12. Goli, M., Gonzalez-Velez, H.: Heterogeneous algorithmic skeletons for FastFlow with seamless coordination over hybrid architectures. In: 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 148–156 (2013)
13. Grelck, C., Scholz, S.: SAC–A functional array language for efficient multi-threaded execution. International Journal of Parallel Programming **34**(4), 383-427 (2006)
14. Harris, M.: CUDA Unfied Memory in CUDA 6. Nvidia, `http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6` (2013).
15. Hoberock, J., Bell, N.: Thrust: C++ template library for CUDA (2011). `http://code.google.com/p/thrust/`
16. Keckler, S.W., Dally, W.J., Khailany, B., Garland, M., Glasco, D.: GPUs and the future of parallel computing. *IEEE Micro* **31**(5), 7–17 (2011). DOI 10.1109/MM.2011.89

17. Kicherer, M., Buchty, R., Karl, W.: Cost-aware function migration in heterogeneous systems. In: 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC '11, pp. 137–145. ACM, New York, NY, USA (2011)
18. Kicherer, M., Nowak, F., Buchty, R., Karl, W.: Seamlessly portable applications: Managing the diversity of modern heterogeneous systems. ACM Trans. Archit. Code Optim. **8**(4), 42:1–42:20 (2012)
19. Landaverde, R., Zhang, T., Coskun, A., Herbordt, M.: An investigation of Unified Memory access performance in CUDA. In: IEEE High Performance Extreme Computing Conference, Waltham, USA (2014)
20. Marques, R., Paulino, H., Alexandre, F., Medeiros, P.D.: Algorithmic skeleton framework for the orchestration of GPU computations. In: Euro-Par 2013 Parallel Processing, *Lecture Notes in Computer Science*, vol. 8097, pp. 874–885. Springer Berlin Heidelberg (2013)
21. NVIDIA Corporation: NVIDIA CUDA C Programming Guide (2013). `http://docs.nvidia.com/cuda/cuda-c-programming-guide`
22. Park, J.: Memory optimizations of embedded applications for energy efficiency. Ph.D. thesis, Dept. of Electrical Engineering. University of Stanford (2011)
23. Shainer, G., *et al.*: The development of Mellanox/NVIDIA GPUDirect over InfiniBand - A new model for GPU to GPU communications. *Computer Science - Research and Development* **26**(3-4) (2011)
24. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - A portable skeleton library for high-level GPU programming. In: 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments, HIPS '11 (2011)