

# Simultaneous CPU–GPU Execution of Data Parallel Algorithmic Skeletons

Fabian Wrede<sup>1</sup>  · Steffen Ernsting<sup>1</sup>

Received: 30 September 2016 / Accepted: 23 December 2016 / Published online: 4 January 2017  
© Springer Science+Business Media New York 2017

**Abstract** Parallel programming has become ubiquitous; however, it is still a low-level and error-prone task, especially when accelerators such as GPUs are used. Thus, *algorithmic skeletons* have been proposed to provide well-defined programming patterns in order to assist programmers and shield them from low-level aspects. As the complexity of problems, and consequently the need for computing capacity, grows, we have directed our research toward simultaneous CPU–GPU execution of data parallel skeletons to achieve a performance gain. GPUs are optimized with respect to throughput and designed for massively parallel computations. Nevertheless, we analyze whether the additional utilization of the CPU for data parallel skeletons in the *Muenster Skeleton Library* leads to speedups or causes a reduced performance, because of the smaller computational capacity of CPUs compared to GPUs. We present a C++ implementation based on a static distribution approach. In order to evaluate the implementation, four different benchmarks, including matrix multiplication, N-body simulation, Frobenius norm, and ray tracing, have been conducted. The ratio of CPU and GPU execution has been varied manually to observe the effects of different distributions. The results show that a speedup can be achieved by distributing the execution among CPUs and GPUs. However, both the results and the optimal distribution highly depend on the available hardware and the specific algorithm.

**Keywords** High-level parallel programming · Data parallel algorithmic skeletons · Simultaneous CPU–GPU execution

---

✉ Fabian Wrede  
fabian.wrede@wi.uni-muenster.de  
Steffen Ernsting  
steffen.ernsting@wi.uni-muenster.de

<sup>1</sup> Leonardo-Campus 3, 48149 Münster, Germany

## 1 Introduction

Because of the growing complexity of applications and the growing amount of data, leading to an increased demand for high performance, we can see nowadays that multi-core and many-core processors have become ubiquitous. This includes hardware accelerators such as graphics processing units (GPU),<sup>1</sup> which gained popularity in recent years and promise to deliver high performance in teraflops-scale.

However, programming GPUs is still a challenging task that requires knowledge about low-level concepts such as memory allocation or data transfer between the main and device memory. Additionally, the programmer has to be familiar with the hardware architecture to fully exploit its computing capabilities. These low-level concepts pose a high barrier for developers and make GPU programming a tedious and error-prone task. The task becomes even more complex when considering clusters and multi-GPU systems, which require additional programming models such as MPI and CUDA.

Cole [1,2] proposed algorithmic skeletons as a high-level approach for parallel programming. Algorithmic skeletons provide well-defined, frequently used parallel and distributed programming patterns. Thus, they encourage a structured way of parallel programming and bring the advantages of portability as well as hiding low-level details. The increased level of abstraction allows for providing implementations of skeletons for various architectures.

As programmers demand increasing computation capacities to solve complex problems, we investigate the possibilities of exploiting all available resources for the execution of data parallel skeletons (such as `map`, `zip` or `reduce`) of the *Muenster Skeleton Library* (Muesli) [3]. Yet, data parallel skeletons have been either executed on the CPU, GPU, or several GPUs in the case of a multi-GPU setup. In general, GPUs are considered to have a significantly higher peak performance compared to CPUs, because they are designed for massively parallel workloads and are optimized for high throughput. Nevertheless, we analyze whether the additional utilization of the CPU can lead to notable speedups. We present a static approach with manually set distributions for simultaneously executing data parallel skeletons on CPUs and GPUs, as well as four benchmark applications (matrix multiplication, N-body simulation, Frobenius norm and ray tracing) to determine the changes in performance compared to exclusive GPU execution.

The remainder of this paper is structured as follows. The *Muenster Skeleton Library* and its underlying concepts are briefly introduced in Sect. 2. Section 3 covers different approaches toward simultaneous CPU–GPU execution in the context of data parallel skeletons. The changes to the implementation of data parallel skeletons for simultaneous CPU–GPU execution are presented in Sect. 4. Section 5 evaluates the implementation by discussing the performance of the benchmark applications. Section 6 outlines related work and Sect. 7 concludes the paper and gives a brief outlook to future work.

---

<sup>1</sup> With regard to the CUDA terminology, we also refer to GPUs as *devices*.

## 2 The Muenster Skeleton Library Muesli

The C++ library Muesli provides algorithmic skeletons and distributed data structures for shared and distributed memory parallel programming. It is built on top of MPI [4], OpenMP [5], and CUDA [6,7]. Thus, it provides efficient support for multi- and many-core computer architectures as well as clusters of both. A first implementation of data parallel skeletons with GPU support using CUDA was presented in [3].

Conceptually, we distinguish between data and task parallel skeletons. Data parallel skeletons such as `map`, `zip`, and `fold` are provided as member functions of distributed data structures, including a one-dimensional array, a two-dimensional matrix, and a two-dimensional sparse matrix, of which the last does not support skeletons with accelerator support for now [8]. Communication skeletons such as `permutePartition` assist the programmer in rearranging data that is distributed among several MPI processes. Task parallel skeletons represent process topologies, such as *Farm* [9], *Pipeline* (Pipe), *Divide and Conquer* (D&C) [10] and *Branch and Bound* (B&B) [11]. They can be arbitrarily nested to create a process topology that defines the overall structure of a parallel application. The algorithm-specific behavior of such a process topology is defined by particular user functions that describe the algorithm-specific details.

In Muesli, a user function is either an ordinary C++-function or a functor, i.e., a class that overrides the function call operator. Due to memory access restrictions, GPU-enabled skeletons must be provided with functors as arguments, CPU skeletons can take both, functions and functors, as arguments. As a key feature of Muesli, the well-known concept of *Currying* is used to enable partial applications of user functions [12]. A user function requiring more arguments than provided by a particular skeleton can be partially applied to a given number of arguments, thereby yielding a “curried” function of smaller arity, which can be passed to the desired skeleton. On the functor side, additional arguments are provided as data members of the corresponding functor.

## 3 Simultaneous CPU–GPU Execution in the Context of Data Parallel Skeletons

There are different approaches toward simultaneous CPU–GPU execution. In this section, we outline different approaches as well as their advantages and disadvantages in order to elaborate on the decision to implement a static approach. Considering simultaneous CPU–GPU execution, there are similarities to load balancing and scheduling mechanisms in other contexts such as distributed operating systems. There is typically a distinction between static and dynamic approaches. For example, Zhang et al. [13] compare dynamic and static load-balancing strategies and Feitelson et al. [14] define the *partition size*, i.e., the number of processors a job is executed on, as either fixed, variable, adaptive or dynamic.

Based on these findings, in the following we distinguish between static, dynamic, and hybrid approaches. In our context of data parallel skeleton execution, *static* means that there is a fixed ratio regarding the distribution of CPU and GPU execution. In

contrast, *dynamic* means that the work is dynamically distributed among CPU and GPU at runtime. *Hybrid* approaches make use of both, e.g., start with a fixed ratio that is adjusted dynamically during the execution.

Static approaches work in such a way that a fixed ratio regarding CPU–GPU execution is specified, which is used to distribute the work. For example, in this implementation a value, determining the distribution between CPU and GPU, is set for each data structure. Thus, the distribution remains the same for each skeleton call.

The advantage is that this causes no significant overhead. There is no need for dedicated work management to balance the utilization of CPU and GPUs. Hence, this approach has a good potential to lead to a minimum execution time of a given program, if the optimal distribution is known and the execution environment remains stable.

These conditions are difficult to meet, and therefore, disadvantages come with this static nature. In order to find an optimal distribution, prior knowledge about the system and the program is required. For instance, cost models can be used to find a good distribution [15]. However, in case of an inferior distribution, the CPU or GPU might idle, while the other component is still working. Consequently, the full potential cannot be utilized.

Dynamic approaches distribute the work at runtime. Consequently, there is a varying amount of work done by the CPU and GPUs. This overcomes the aforementioned shortcomings of the static approach. There is no need to have prior knowledge in order to find an optimal distribution. This makes it more convenient to use for programmers, as the work is automatically assigned to the components. Moreover, there should be less idle time, because as soon as either CPU or a GPU has finished, a new task is assigned.

However, this approach introduces disadvantages. An additional management effort is required to distribute the work. For example, an implementation could use a queue that stores work packages and the CPU and GPUs fetch a new package as soon as they have finished their current one. This would require a locking mechanism for the queue access and incur multiple copying steps from the main memory to the GPU memory, both consuming additional time. Hence, the performance gain through simultaneous CPU–GPU execution could be outweighed by the management overhead.

Hybrid approaches combine aspects of both static and dynamic ones. One instance of a hybrid approach is *work stealing*. At first, a static distribution is used so that there is no overhead during the execution. As soon as one component has finished its tasks, there is a transition into a dynamic approach, e.g., tasks are stolen from another component's queue. Hence, there is a better resource utilization, but also overhead for management tasks such as locking mechanisms.

Consequently, hybrid approaches comprise advantages and disadvantages of static and dynamic approaches. They should be able to come close to an optimal distribution, without requiring the user to specify it manually. Additionally, there should be less overhead during the program execution compared to dynamic approaches, e.g., in the case of work stealing, the dynamic distribution is utilized later during the execution. In the best case, the optimal distribution is initially used and thus, the approach behaves like a static one. However, in order to achieve this behavior and to determine a good distribution, knowledge about the system as well as the program is required.

Our goal is to examine the effects of additionally utilizing the CPU for execution of data parallel skeletons and the effects of different distributions among CPU and GPUs. Consequently, for the implementation, we have focused on a static approach since it serves our purpose best, which is to analyze whether a simultaneous CPU–GPU is in general beneficial in terms of execution time and how notable the speedups are. We focus on benchmarking different distributions, while trying to keep the effects of additional factors, such as management overhead introduced by dynamic approaches, as low as possible. For example, Grewe and O’Boyle [16] show that static approaches can outperform dynamic approaches on heterogeneous multi-core platforms due to the lower overhead. Once the potential of simultaneous CPU–GPU execution has been exhibited, we can combine it with a cost model to predict the static distribution or the initial distribution in a hybrid approach. We leave this to future work.

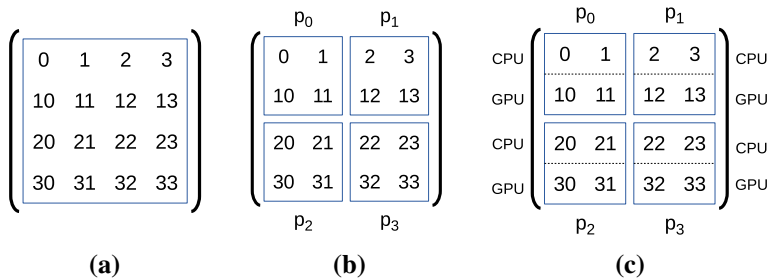
## 4 Implementation Details

In this paper, we focus on the most important changes regarding the implementation of simultaneous CPU–GPU execution of data parallel skeletons in Muesli. A more detailed overview about the original C++ implementation of data parallel skeletons can be found in [3]. The revised version featuring multi-GPU execution of data parallel skeletons is presented in [17].

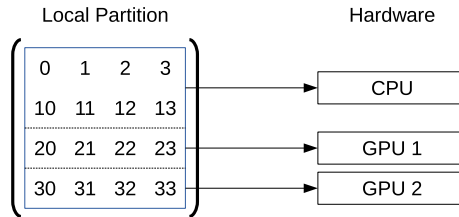
### 4.1 Calculation Ratio

Muesli offers the distributed array and the distributed matrix as distributed data structures that provide data parallel skeletons with GPU support. These data structures are distributed among several processes.

Figure 1 shows the different views on a distributed matrix. In the given example, a  $4 \times 4$  matrix is considered (Fig. 1a), which is distributed among four processes (Fig. 1b). Each local partition is equally distributed between one CPU and one GPU for the execution of data parallel skeletons (Fig. 1c). When designing the functions for data parallel skeletons, the programmer can focus on the virtual global view, i.e., pretending that the data is not distributed. All elements of a matrix are automatically processed in parallel on the available nodes when the skeleton is executed.



**Fig. 1** Distributed matrix. **a** Global view, **b** local view and **c** CPU–GPU distribution



**Fig. 2** Distribution of local partition on hardware resources

The constructor of a distributed array or matrix takes the *calculation ratio* as an additional argument, which determines how the execution of data parallel skeletons is distributed among CPU and GPUs. This satisfies our requirements since we only distribute data between two resource types: CPU and GPU. The distribution is done per row: if a partition consists of 1024 rows and the ratio is 45%,  $\lfloor 1024 \times 0.45 \rfloor = 460$  rows are handled by the CPU and the remaining  $1024 - 460 = 564$  rows by the available GPUs.

To further clarify the distribution mechanism, an additional example is provided in Fig. 2. The (sub)matrix on the left represents a local partition. We assume that the calculation ratio is 70%. Therefore, the number of lines for which the skeleton execution is performed by the CPU is  $\lfloor 4 \times 0.7 \rfloor = 2$ . For the remaining two rows, the skeleton execution is distributed among all available GPUs within the node. In this scenario, there are two GPUs available. Consequently, each GPU executes the skeleton for one row of the local partition.

Muesli targets HPC cluster environments and therefore we assume that each node is equipped with the same hardware. Moreover, as it is typically the case, we assume that each node is equipped with one or more identical GPUs. Consequently, the GPU workload can be equally distributed among all available GPUs, since each GPU provides the same computation capacity.

### 4.2 Execution Plans

The implementation is based on *execution plans*. In Muesli, execution plans have originally been introduced to manage the execution of data parallel skeletons on multi-GPU systems [3]. They include information about the partition size, indices and pointers to the location in main and device memory.

We have extended the scope to also manage the distribution between the CPU and GPUs. As soon as a distributed data structure is instantiated, an execution plan is created for the CPU and for each GPU. Thus, in a system with one CPU and four GPUs, five execution plans are created. Execution plans serve two different purposes: (1) to structure the data such that the skeleton can be executed simultaneously on the CPU and GPUs, and (2) to provide necessary information for the execution of skeletons, such as partition size or pointers to host and device memory. This information is for example required to launch CUDA kernels and to transfer data.

Listing 1 shows the initialization of the execution plans, which is taken care of by Muesli. In lines 4–8, the plans are created and it is checked, whether there are any GPUs

```

1  template <typename T>
2  void Matrix<T>::init()
3  {
4      plans = new ExecutionPlan<T> [num_gpus + 1];
5
6      if(num_gpus == 0){
7          calculation_ratio = 1;
8      }
9      // calculate number of rows for CPU and GPUs
10     // nLocal denotes the number of rows of local partition
11     auto rows_on_host = static_cast<int>(std::floor(nLocal
12         * calculation_ratio));
13     auto rows_on_devices = nLocal - rows_on_host;
14     // set numbers of rows and columns for CPU plan
15     plans[index_cpu_plan].nLocal = rows_remaining_on_host;
16     plans[index_cpu_plan].mLocal = mLocal;
17     // set numbers of rows and columns for GPU plans
18     for (int i = 0; i < num_gpus; i++){
19         plans[i].nLocal = rows_on_devices / num_gpus;
20         plans[i].mLocal = mLocal;
21     }
22     // distribute remaining rows
23     for (int i = 0; i < rows_on_devices % num_gpus; i++){
24         plans[i].nLocal++;
25     }
26     // set further values for execution plans
27     int gpuBase = 0; int rowBase = 0;
28
29     for (int i = 0; i < num_gpus + 1; i++) {
30         plans[i].size = plans[i].nLocal * plans[i].mLocal;
31         plans[i].bytes = plans[i].size * sizeof(T);
32         plans[i].first = gpuBase + firstIndex;
33         // ... further values
34     }
35 }

```

**Listing 1** Initialization of Execution Plans

available. If there are none, skeletons are entirely executed by the CPU. In lines 11–24, the number of rows for each execution plan is determined. First, the number of rows handled by the CPU is calculated and stored in the CPU execution plan. Second, the remaining number of rows is distributed among the GPUs within the two for-loops. Lines 26–33 show the calculation of the remaining necessary information, such as the size, required memory space and indices.

### 4.3 Data Parallel Skeletons

As we have mentioned in Sect. 4.1, data parallel skeletons are provided as member functions of distributed data structures. Conceptually, each skeleton consists of two parts. The first part handles the GPU execution and the second part the CPU execution. The GPU execution consists of asynchronous kernel launches for each GPU. As the

kernels are launched asynchronously using streams, it is possible to overlap the CPU calculations and the execution of the CUDA kernels without explicit threading. The CPU part is parallelized by using OpenMP [5].

```

1 void Matrix<T>::mapInPlace(MapFuncutor& f) {
2     upload();
3
4     // map GPU
5     for (int i = 0; i < Muesli::num_gpus; i++) {
6         f.init(plans[i].nLocal, plans[i].mLocal,
7             plans[i].firstRow, plans[i].firstCol);
8
9         // ... calculate dimGrid etc.
10        detail::mapKernel<<<dimGrid, dimBlock, 0,
11            Muesli::streams[i]>>>(
12            plans[i].d_Data, plans[i].d_Data, plans[i].size,
13            f);
14    }
15
16    // map CPU
17    f.init(plans[index_cpu_plan].nLocal,
18        plans[index_cpu_plan].mLocal,
19        plans[index_cpu_plan].firstRow,
20        plans[index_cpu_plan].firstCol);
21
22    #pragma omp parallel for
23    for (int i = 0; i < plans[index_cpu_plan].nLocal; i++)
24    {
25        for (int j = 0; j < plans[index_cpu_plan].mLocal;
26            j++) {
27            setLocal(/* index calculation */,
28                /* index calculation */,
29                f(localPartition[ /* index calculation */
30                    ]));
31        }
32    }
33
34    msl::syncStreams();
35 }

```

**Listing 2** Implementation of the mapInPlace skeleton

The implementation of the `mapInPlace` skeleton, which is provided by the Muesli library, is shown in Listing 2. The `upload` method implements a lazy copying mechanism for transferring data from main to device memory and vice versa, i.e., data is only copied if required. Lines 5–11 show the GPU part. First, the functor is initialized and second, the kernel is launched. This is done for each GPU. After the asynchronous kernel launches, the CPU part of the skeleton is executed as shown in lines 14–23. First, the functor is again initialized with the correct values (line 14) and second, the local partition is updated with the new values (lines 16–23). The function



`syncStreams` synchronizes the kernel launches with the host thread and therefore ensures that the kernels are finished before returning from the `mapInPlace` skeleton.

The other skeletons provided by Muesli behave in a similar way. One point to further comment on are the final steps of the fold skeleton. After each resource has folded the data assigned to it, each process computes its local result by combining these results with a local fold step, which is performed on the CPU. Finally, all local fold results are collected by one process, which computes the final global result. This last fold step is as well performed on the CPU.

## 5 Experimental Results

To observe the effects on performance, we have conducted four benchmarks. All benchmarks have been executed on a cluster with two Intel Xeon E5-2680 v3 CPUs (total of 24 cores) and two Nvidia K80 boards (total of four GPUs) per node.

As research hypothesis, we have assumed that an increasing usage of the CPU at first leads to a speedup compared to a GPU-only configuration, because additional resources can be utilized and data transfer costs can be reduced, i.e., there is less data to be copied from main to device memory and vice versa. However, with an increasing CPU usage there should be a turning point, where the execution time starts to increase, since the computational capacity of the GPUs cannot be fully exploited anymore.

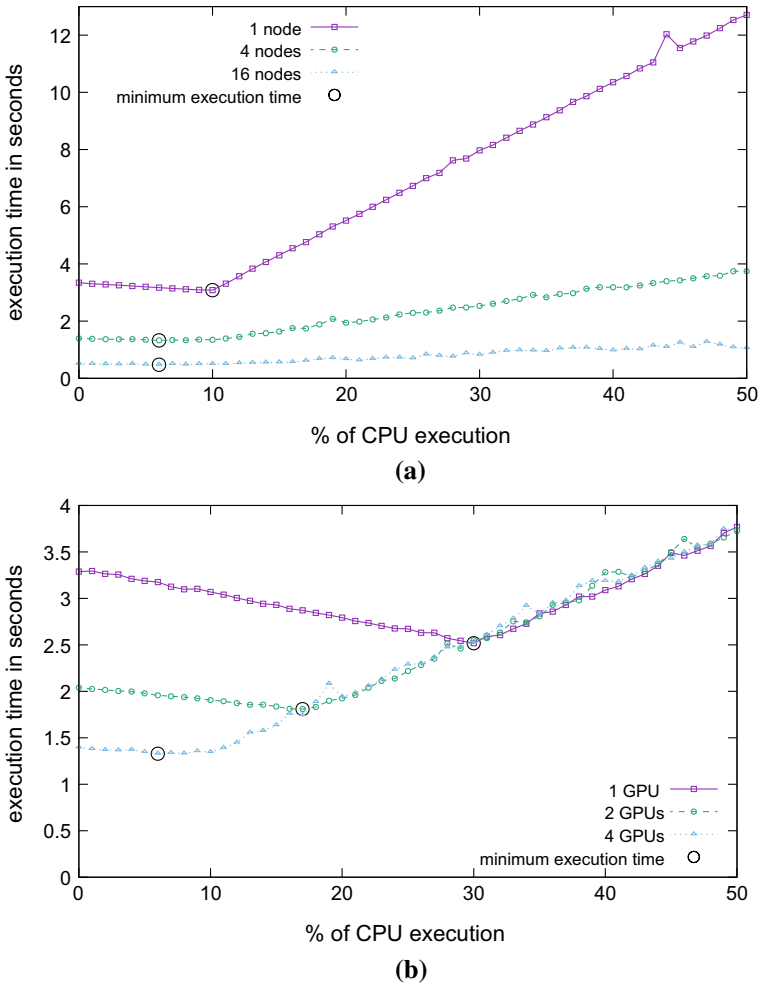
This is due to the architecture of GPUs. Today's GPUs consist of many, yet simple, cores and are optimized for throughput, in contrast to latency-oriented CPUs, which have fewer cores that are more complex, e.g., they offer sophisticated out-of-order execution. Consequently, GPUs are utilized best, if a large number of threads can be launched to efficiently execute a given task. Therefore, an increasing CPU execution should not leave enough work to the GPU so that its potential cannot be fully exploited, leading to an overall increased execution time.

In the following, we approach each benchmark from two perspectives. First, we keep the number of GPUs constant (with four GPUs in each node) and vary the number of nodes. We have used a configuration with 1, 4, and 16 nodes. Second, we keep the number of nodes constant (configuration with four nodes) and vary the number of GPUs in each node (one, two, and four GPUs). The presented execution times measure the entire application's execution time. Besides the execution of the skeletons, this also includes the creation of CUDA streams, instantiation of data structures as well as data transfers from host to device memory and vice versa.

### 5.1 Matrix Multiplication

The matrix multiplication benchmark uses the Cannon algorithm [18]. A detailed description of the implementation can be found in [17]. It uses a dot product calculation as user function and the `mapIndexInPlace` skeleton. Two  $8192 \times 8192$  matrices with single precision values were used for the benchmark. The results are presented in Fig. 3.

Initially, the execution time decreases when the CPU is used to perform a part of the skeleton execution and increases after a turning point, at which the minimum execution



**Fig. 3** Results matrix multiplication benchmark. **a** 1, 4, and 16 nodes with 4 GPUs per node, **b** 4 nodes with 1, 2, and 4 GPUs per node

time is reached. In the case of four GPUs per node as well as one and four nodes, the minimum execution time is reached at 10% CPU execution and with 16 nodes at 6% (Fig. 3a).

Considering the results for four nodes with one, two, and four GPUs each, the results are similar (Fig. 3b). First, the execution time decreases and after a turning point it increases again, because the full potential of the GPUs cannot be utilized anymore and the execution on the CPU consumes more time. Eventually, the results converge, as the number of GPUs becomes less significant, when the CPU takes over a larger part of the execution. With 100% CPU execution, the number of GPUs becomes completely irrelevant and the execution time is the same for all three configurations with four nodes and one, two, and four GPUs in each node.

The highest speedup can be observed when there are few GPUs in the system. i.e., for four nodes with one GPU each, the execution time decreases from 3.29 s (only GPU) to 2.51 s (30% CPU execution), leading to a speedup of 23.7%. For four GPUs the speedup is only 4.8% (0.07 s), as the four GPUs have a higher computational power in relation to the CPU and thus, there is not as much potential for the CPU to increase the overall performance as in a system with a single GPU.

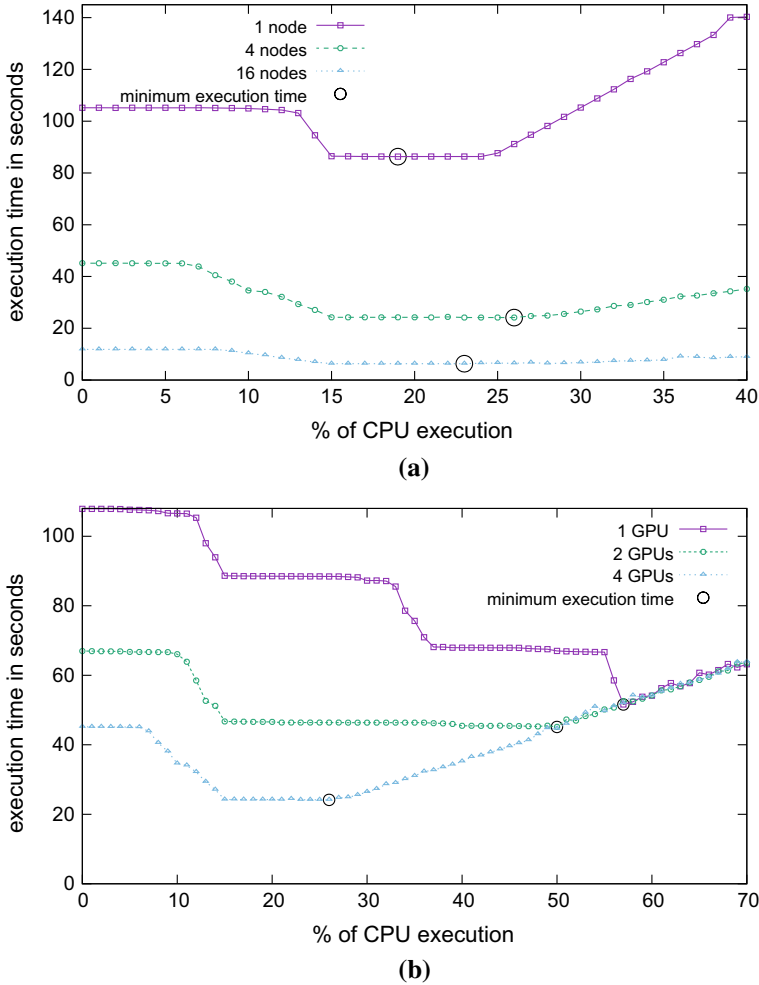
## 5.2 N-Body Simulation

The N-body simulation simulates a dynamical system of particles and their influence on each other over a series of time steps. We used a distributed array with single precision values representing the particles and the `map` skeleton for the calculation. The user function iterates over the array to determine the influence of the other particles on the current one. The benchmark was conducted with 500,000 particles over five time steps. The results are shown in Fig. 4.

The benchmark results are to some extent similar to the results of the matrix multiplication benchmark. With increasing CPU usage, the execution time first decreases and then increases again. However, the improvements occur in steeper steps. Further analysis revealed that the steps can be ascribed to the varying number of thread blocks for the kernel launches and their distribution among the GPUs' multiprocessors. This is known as the *tail effect* [19].

In the following, we consider the configuration with four nodes and one GPU per node. With a calculation ratio of 12%, the execution time is 105.32 s. As stated above, the implementation uses a distributed array as the underlying data structure. Since the benchmark has been executed on four nodes,  $500,000/4 = 125,000$  particles have been processed on each node. The calculation ratio is 12% and hence, the GPU has processed  $125,000 - \lfloor 0.12 \times 125,000 \rfloor = 110,000$  particles. When launching a CUDA kernel, Muesli sets the number of threads per block per default to 512. The number of thread blocks is calculated as  $\lfloor (arraySize + threadsPerBlock - 1) / threadsPerBlock \rfloor$ . In the given case, this is  $\lfloor (110,000 + 512 - 1) / 512 \rfloor = 215$  thread blocks. The HPC cluster, we have executed the benchmark on, is equipped with Nvidia K80 boards. Each GPU has 13 multiprocessors and on each multiprocessor there can be four active thread blocks at the same time, due to the maximum number of 2048 active threads. Thus, seven multiprocessors have processed 17 thread blocks, while six multiprocessors have processed 16 thread blocks ( $215/13 = 16.53$  and  $215 \bmod 13 = 7$ ). Since four thread blocks can be active simultaneously, the processors processing 17 thread blocks have to launch five *waves*, because of the *tail* of one block. With a calculation ratio of 15%, the execution time drops to 88.64s. Since the number of thread blocks decreases to 208, each multiprocessor has to process 16 thread blocks, which is possible within four waves and this leads to the observed speedup.

Taking the results for one node with four GPUs (Fig. 4a), the execution time remains almost constant for 0–13% CPU execution (about 105 s) and then drops to around 86 s for 15–24% CPU execution, before the execution time starts to increase linearly. In contrast to the matrix multiplication benchmark, the speedup for the configuration



**Fig. 4** Results N-body simulation benchmark. **a** 1, 4, and 16 nodes with 4 GPUs per node, **b** 4 nodes with 1, 2, and 4 GPUs per node

with four nodes and four GPUs per node is higher (46.5%) than for the configuration with four nodes and two GPUs per node (32.6%).

### 5.3 Frobenius Norm

We included the calculation of the Frobenius norm as a benchmark that has very simple user functions and makes use of the `fold` skeleton. Thus, it is possible to observe different effects compared to the other benchmarks, which have rather complex user functions. The Frobenius norm is a matrix norm, which is defined as  $\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2}$ . The user functions are shown in Listing 3. The

Square map functor (lines 1–9) takes one argument and returns its square value, while the Sum fold functor (lines 9–15) takes two arguments and returns their sum. The actual user functions are implemented in terms of the function call operator (lines 4–6 and lines 12–14).

```

1  class Square: public MapFunctor<double, double> {
2      public:
3          MSL_USERFUNC //expands to: __host__ __device__
4          double operator()(double x) const {
5              return x * x;
6          }
7  };
8
9  class Sum: public FoldFunctor<double, double> {
10     public:
11         MSL_USERFUNC //expands to: __host__ __device__
12         double operator()(double x, double y) const {
13             return x + y;
14         }
15 };

```

**Listing 3** Frobenius Norm User Functions

```

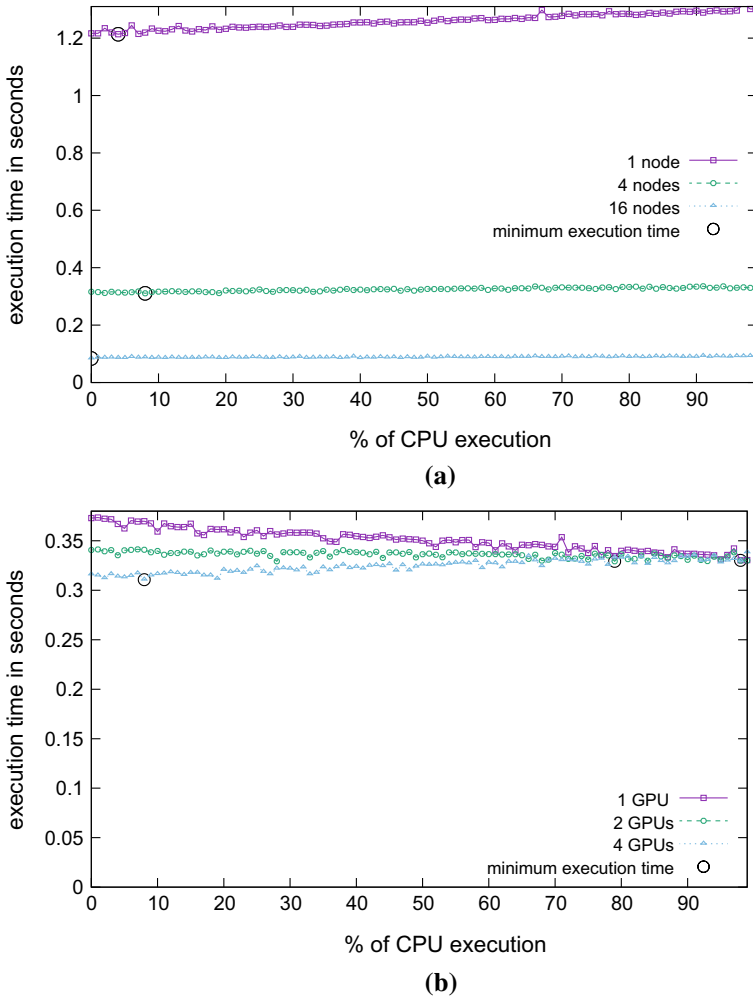
1  double frobenius_norm(int dimension, bool output, double
2      calc_ratio) {
3      // create distributed matrix, with random values
4      // (&init is function pointer)
5      Matrix<double> dm(dimension, dimension, 1,
6          num_total_procs, &init, Distribution::DIST,
7          calc_ratio);
8      // instantiate functors
9      Square sq; Sum sum;
10     // call mapInPlace skeleton with square user function
11     dm.mapInPlace(sq);
12     // call fold skeleton with sum user function
13     double f_norm = dm.fold(sum, 1);
14
15     return std::sqrt(f_norm);
16 }

```

**Listing 4** Frobenius Norm Algorithm

In order to calculate the Frobenius norm, first, all values of a distributed matrix (line 3) are squared using the `mapInPlace` skeleton with the `Square` functor (line 7) and second, all values are folded using the `fold` skeleton with the `Sum` functor (line 9). As a last step, the square root is calculated to get the final result (line 11). For this benchmark, a  $16,384 \times 16,384$  matrix was used. The implementation of the algorithm, as described above, is shown in Listing 4.

The results indeed differ from the previously presented results to some extent. In the case of 16 nodes and four GPUs per node (Fig. 5a), no performance improvement can be achieved by using the CPU. Already 1% of CPU leads to worse results, even though the increase of the execution time is not significant.



**Fig. 5** Results Frobenius norm benchmark. **a** 1, 4, and 16 nodes with 4 GPUs per node, **b** 4 nodes with 1, 2, and 4 GPUs per node

In the case of four nodes with a varying number of GPUs (Fig. 5b), we can observe a slightly different result. With one and two GPUs, the best results can be achieved by heavily making use of the CPU. In the case of one GPU, the minimal execution time is at 98% CPU execution with a speedup of 11% and in the case of two GPUs, it is at 79% CPU execution with a speedup of 3.4%.

As pointed out earlier, the calculation of the Frobenius norm uses very simple user functions. Consequently, utilizing GPUs to calculate the Frobenius norm means that expensive data transfers are required to perform tasks, which are not computation-intensive. By increasing the CPU usage, these data transfers can be avoided and therefore, speedups can be achieved for configurations with one or two GPUs per node. In the case of four GPUs per node, the utilization of the CPU leaves more com-

putational capacity of the GPUs unutilized and therefore, the additional CPU execution is less beneficial.

In general, the results are more unstable than in the other benchmarks. This is due to the low execution times, which are in a range from 0.08 to 1.2 s. Therefore, the results are subject to more significant variations.

## 5.4 Ray Tracing

Ray tracing is an algorithm to render images. Coming from an imaginary camera, rays are extended through an image plane into a scene. A scene consists of various objects with different surfaces, which reflect the rays in different ways, and light sources. By tracing the rays' paths and intersections with the objects in the scene, it is possible to calculate how the scene looks like from the given perspective. In this benchmark, we created  $1024 \times 1024$  pixel images. Each picture was rendered using ten different light sources and 100 spheres, which were randomly placed within the scene.

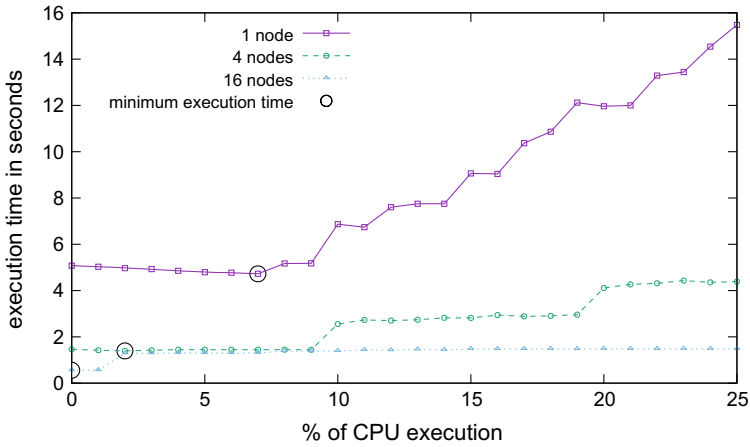
This benchmark confirms the results of the previous ones. In the case of one and four nodes with four GPUs each (Fig. 6a), best results can be achieved with 7% (4.73 s) and 2% CPU execution (1.4 s). Similarly to the matrix multiplication and N-body simulation benchmarks, the execution time decreases at first and increases with a higher utilization of the CPU. The results for 16 nodes correspond to the findings of the Frobenius norm benchmark. Here, already 1% CPU execution leads to an increased execution time.

The results for four nodes with one, two or four GPUs each, also match the other findings (Fig. 6b). First, there is a decrease in the execution time and an increase later on. In the case of one GPU, the CPU part must be larger to achieve the minimum execution time, compared to the configurations with two and four GPUs, because in this scenario, it is possible to benefit most from the additional computational capacity. At 20% CPU execution, the results converge, since the CPU becomes the limiting factor for the execution.

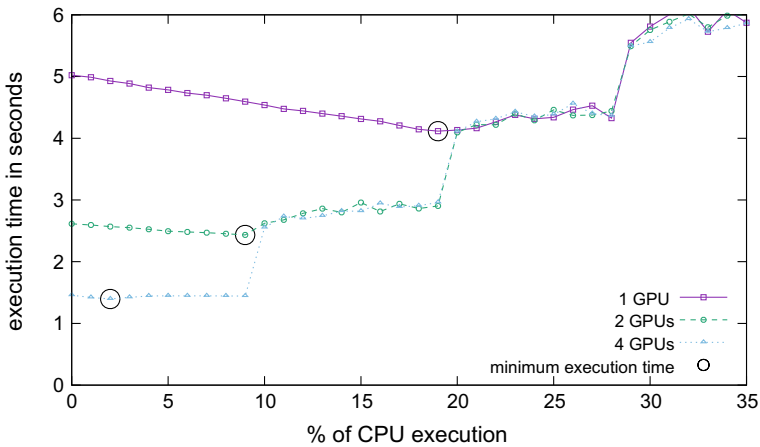
Similar to the N-body simulation benchmark, steps occur in the results. This time, these steps arise from the number of CPU cores and the number of rows processed by the CPU. The following example demonstrates the behavior for a configuration with four nodes. Muesli launches one thread per core and there have been 24 cores per node on the cluster that was used for the benchmark. Since the benchmark has been executed with a  $1024 \times 1024$  matrix, there are 256 rows per node. Consequently, for a calculation ratio of 9%, 23 rows were processed by the CPU and for a calculation ratio of 10%, it has been 25 rows. Thus, with 25 rows it is no longer possible to process all rows within one wave of 24 threads, but an additional wave with only one utilized core has to be launched. This explains the steps in the execution times.

## 5.5 Evaluation

All in all, it becomes clear that the simultaneous CPU–GPU execution can be beneficial regarding the execution time in certain situations. As we have expected in our research hypothesis, it is possible to observe patterns in the results of the four benchmarks. We



(a)



(b)

**Fig. 6** Results ray tracing benchmark. **a** 1, 4, and 16 nodes with 4 GPUs per node, **b** 4 nodes with 1, 2, and 4 GPUs per node

identified two factors that determine the possible performance gain of simultaneous CPU–GPU execution, compared to a GPU-only configuration:

*Hardware Configuration* The speedup achieved by simultaneous CPU–GPU execution of skeletons was highest for configurations with fewer nodes and fewer GPUs per node. In all four benchmarks, the optimal percentage of CPU execution decreased, when the number of GPUs per node increased.

As we have already mentioned above, this can be explained by the difference of the available CPU’s and GPU’s computational capacities. In a setup with a powerful CPU and only one GPU, it is beneficial to utilize the capacity of the CPU. However, if there are more GPUs in the system, this is not desirable anymore, as too much capacity of the GPUs is lost, when the CPU takes over the execution.



**Table 1** Comparison of execution times (4 nodes and 1 GPU per node)

Benchmark	Execution times			CPU-/GPU- only ratio	Optimal calculation ratio (%)	Speedup (%)
	CPU-only (s)	GPU-only (s)	Minimum (s)			
Matrix mult.	6.7248	3.2893	2.5188	2.04	30	23.4
N-body	87.7260	107.8758	51.5904	0.81	57	52.2
Frobenius	0.3349	0.3730	0.3304	0.90	98	11.4
Ray tracing	16.4775	5.0220	4.1160	3.28	19	18.0

*Implementation of Algorithm and User Functions* The benchmark results show that certain implementations are better suited to be executed on GPUs than others. Table 1 summarizes the benchmark results for a configuration with four nodes and one GPU per node. By analyzing the execution times for CPU-only and GPU-only configurations (column 5, CPU-only/GPU-only ratio), it becomes clear that our implementation of the matrix multiplication and ray tracing benchmark are more GPU-friendly than the implementations of the N-body simulation and Frobenius norm calculation.

The concrete implementation has an impact on the optimal CPU execution ratio and the possible speedup. The optimal distribution for the N-body simulation was 57% CPU execution (speedup of 52.2% compared to the GPU-only configuration), whereas in the case of the ray tracing benchmark the optimal distribution was 19% (speedup of 18%). Consequently, the capability to reduce the execution time by using simultaneous CPU–GPU execution is higher for implementations that are less suited to be executed on GPUs.

The findings of the Frobenius benchmark seem to slightly contradict this conclusion. Even though, the optimal distribution for the above mentioned configuration is 98%, which supports our conclusion, only a speedup of 11.4% could be achieved. Moreover, in the case of a configuration with 16 nodes and four GPUs per node, it is best not to make use of the CPU for skeleton execution at all. However in the case of the Frobenius benchmark, the low speedup can be ascribed to the fact that the actual skeleton execution makes up only for a very small portion of the overall execution time, compared to other operations such as the creation of CUDA streams as well as data transfers from device to host memory and vice versa. Additionally, in the case of four GPUs per node, the effects of the hardware configuration as described above outweigh the effects of the implementation.

## 6 Related Work

Research regarding hybrid CPU–GPU execution has been conducted in the field of algorithmic skeletons as well as in other domains. In the following, we will outline related work in the mentioned order and relate them to the presented approach.

SkePU is a skeleton library that focuses on multicore CPUs and multi-GPU systems. It is available in two distributions: as a stand-alone version and as an integrated ver-

sion using the StarPU runtime system as backend. The StarPU runtime system offers dynamic scheduling and memory management support for heterogeneous multicore systems [20]. By mapping SkePU skeleton calls onto StarPU tasks, it is possible to achieve hybrid CPU–GPU execution and load balancing [21]. In contrast to SkePU with StarPU as backend, we implemented the simultaneous CPU–GPU execution as a part of the Muesli library.

The skeleton framework Marrow provides algorithmic skeletons for multi-GPU systems. It does not support CPU execution of skeletons, but it provides a runtime system, which incorporates various modules such as Scheduler, Auto-Tuner, and Task Launcher. These modules allow for scheduling tasks among multiple GPUs, based on their relative performance [22]. Thus, in contrast to the presented approach, the load balancing mechanism does not distribute work between CPU and GPUs, but among different GPUs with different computational capacities. Marrow also relies on a static approach to distribute work among GPUs. To determine the distribution, Marrow uses a benchmark to evaluate the relative performance of the available GPUs and distributes the work accordingly. This is one instance for a cost model to determine the distribution, as mentioned above, which could be implemented in future work.

SkeTo is a framework supporting multi-core clusters and GPUs by utilizing OpenMP and CUDA. Similar to the previous version of Muesli, it is possible to compile two separate programs, one for CPUs and one for GPUs, from the same code base. However, there is no support for simultaneous CPU–GPU execution [23]. The two decision made independently from each other, to provide two executables for CPUs and GPUs out of the same code base, and SkePU’s design decision to use StarPU as an runtime backend show that it poses an additional challenge to incorporate efficient utilization of hybrid architectures within an algorithmic skeleton library.

Additional skeleton libraries that we would like to mention are FastFlow [24] and SkelCl [25]. Both frameworks focus on many-core and multi-GPU systems. FastFlow has been extended to provide a OpenCL-based back-end for hybrid CPU–GPU architectures [26]. During the program execution, each OpenCL kernel is allocated to the best appropriate device. Per default, this means that GPUs are selected first and if there is none available, the kernel is allocated to the CPU.

Besides algorithmic skeletons there are other approaches toward the utilization of heterogeneous architectures. For instance, Lee, Ro, and Gaudiot [27] present the CHC framework, which allows for executing CUDA kernels simultaneous on CPU as well as GPU, and Chen, Huo, and Agrawal [28] show how to accelerate MapReduce applications on heterogeneous CPU–GPU architectures. The results of both papers show that the CPU utilization can reduce the execution times of the benchmark applications in many cases. This motivates the decision to include a load balancing mechanism for heterogeneous architectures in an algorithmic skeleton library.

## 7 Conclusion

We have presented an approach for simultaneous CPU–GPU execution of data parallel skeletons. By using so-called execution plans it was possible to distribute the execution of skeletons among the CPU and an arbitrary number of GPUs. Four different benchmarks were presented in order to determine the effects on the execution time.

The benchmarks revealed that it is possible to achieve a notable speedup in certain situations. The achieved speedup depends on the used hardware and the complexity of the user function.

Since the presented approach has proven to lead to speedups for parallel applications making use of Muesli’s data parallel skeletons, future work will be directed toward further approaches. This will include dynamic approaches in order to examine the effects of additional management overhead. Another research direction would be a further analysis of methods for finding optimal CPU–GPU ratios, such as test runs and cost models.

## References

1. Cole, M.: *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge (1991)
2. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.* **30**(3), 389–406 (2004)
3. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *Int. J. High Perform. Comput. Netw.* **7**(2), 129–138 (2012)
4. The Message Passing Interface (MPI) standard. <http://www.mcs.anl.gov/research/projects/mpi/>. Accessed Apr 2016
5. The OpenMP API Specification for Parallel Programming. <http://openmp.org>. Accessed Apr 2016
6. Nvidia Corporation: CUDA Website. <http://www.nvidia.de/object/cuda-parallel-computing-de.html>. Accessed Apr 2016
7. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. *Queue* **6**(2), 40–53 (2008)
8. Ciechanowicz, P.: Algorithmic skeletons for general sparse matrices on multi-core processors. In: *Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pp. 188–197 (2008)
9. Ernsting, S., Kuchen, H.: A scalable farm skeleton for hybrid parallel and distributed programming. *Int. J. Parallel Prog.* **42**(6), 968–987 (2014)
10. Poldner, M., Kuchen, H.: Skeletons for divide and conquer algorithms. In: *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*. ACTA Press (2008)
11. Poldner, M., Kuchen, H.: Algorithmic skeletons for branch and bound. In: *Software and Data Technologies: First International Conference, ICSoft 2006, Setúbal, Portugal, 11–14 September 2006, Revised Selected Papers*, pp. 204–219. Springer, Berlin (2008)
12. Kuchen, H., Striegnitz, J.: Higher-order functions and partial applications for a C++ skeleton library. In: *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande, JGI '02*, pp. 122–130. ACM, New York (2002)
13. Zhang, Y., Kameda, H., Hung, S.L.: Comparison of dynamic and static load-balancing strategies in heterogeneous distributed systems. *IEE Proc. Comput. Digit. Tech.* **144**(2), 100–106 (1997)
14. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K.C., Wong, P.: *Theory and practice in parallel job scheduling*. In: *Proceedings of the Job Scheduling Strategies for Parallel Processing, IPPS '97*. Springer, London (1997)

15. Ferguson, D.F., Nikolaou, C., Sairamesh, J., Yemini, Y.: Economic models for allocating resources in computer systems. In Clearwater, S.H. (ed.) *Market-based Control: A Paradigm for Distributed Resource Allocation*, pp 156–183 (1996)
16. Grewe, D., O’Boyle, M.F.P.: A static task partitioning approach for heterogeneous systems using OpenCL. In: *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software, CC’11/ETAPS’11*, pp. 286–305. Springer, Berlin (2011)
17. Ernsting, S., Kuchen, H.: Data parallel algorithmic skeletons with accelerator support. *Int. J. Parallel Program.* (2016). doi:[10.1007/s10766-016-0416-7](https://doi.org/10.1007/s10766-016-0416-7)
18. Cannon, L.E.: *A Cellular Computer to Implement the Kalman Filter Algorithm*. Ph.D. thesis, Montana State University, Bozeman, MT, AAI7010025 (1969)
19. Demout, J.: CUDA Pro Tip: minimize the tail effect. <https://devblogs.nvidia.com/paralleforall/cuda-pro-tip-minimize-the-tail-effect/>. Accessed Sep 2016
20. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. Pract. Exp.* **23**(2), 187–198 (2011)
21. Dastgeer, U., Kessler, C., Thibault, S.: Flexible runtime support for efficient skeleton programming on heterogeneous GPU-based systems. In: *Proceedings of the International Conference on Parallel Computing, ParCo ’11* (2011)
22. Alexandre, F., Marques, R., Paulino, H.: On the support of task-parallel algorithmic skeletons for multi-GPU computing. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC ’14*, pp. 880–885. ACM, New York (2014)
23. Sato, S., Iwasaki, H.: A skeletal parallel framework with fusion optimizer for GPGPU programming. In: Hu, Z. (ed.) *Proceedings of the 7th Asian Symposium. APLAS 2009, Seoul, Korea, December 14–16, 2009. Lecture Notes in Computer Science*, vol. 5904, pp. 79–94. Springer, Berlin (2009)
24. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: FastFlow: High-level and efficient streaming on multi-core. In: Pillana, S., Xhafa, F. (eds.) *Programming Multi-core and Many-Core Computing Systems, Parallel and Distributed Computing*, chapt. 13. Wiley-Blackwell (**in press**)
25. Steuwer, M., Gorlatch, S.: SkelCL: enhancing OpenCL for high-level programming of multi-GPU systems. In: Malyshkin, V. (ed.) *Proceedings of the 12th International Conference on Parallel Computing Technologies. PaCT ’13*, pp. 258–272. Springer, Berlin (2013)
26. Goli, M., Gonzalez-Velez, H.: Heterogeneous algorithmic skeletons for fast flow with seamless coordination over hybrid architectures. In: *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP ’13*, pp. 148–156. IEEE Computer Society, Washington (2013)
27. Lee, C., Ro, W.W., Gaudiot, J.-L.: Boosting CUDA applications with CPU–GPU hybrid computing. *Int. J. Parallel Prog.* **42**(2), 384–404 (2014)
28. Chen, L., Huo, X., Agrawal, G.: Accelerating MapReduce on a coupled CPU–GPU architecture. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC ’12*, pp. 25:1–25:11. IEEE Computer Society Press, Los Alamitos (2012)