The Journal of Supercomputing (2020) 76:9976–10008
https://doi.org/10.1007/s11227-020-03234-w

clusterCL: comprehensive support for multi-kernel data-parallel applications in heterogeneous asymmetric clusters

Valon Raca[1] · Eduard Mehofer[1]

Published online: 9 March 2020

Abstract
Heterogeneous cluster systems consisting of CPUs and different kinds of accelerators have become mainstream in HPC. Programming such systems is a difficult task and requires addressing manifold challenges that stem from the intricate composition of such systems and peculiarities of scientific applications. A broad range of obstacles preventing efficient execution have to be considered and dealt with properly. In this paper, we propose a systematic approach and a framework that is capable of providing comprehensive support for running data-parallel applications in heterogeneous asymmetric clusters. Our implementation provides work partitioning and distribution by ensuring workload balance in the cluster while handling of partitioning-induced communication and synchronization in a transparent way. In our experimental section, we choose 11 representative scientific applications from different domains to evaluate our approach. Experimental results show a strong speedup and workload balance for different cluster configurations.

Keywords Heterogeneous computing · Asymmetric clusters · Scientific applications

1 Introduction

Heterogeneous computing with special-purpose devices and accelerators besides CPUs has become state of the art in the field of high performance computing. The key benefits of heterogeneous architectures compared to homogeneous systems—better performance and better energy efficiency—forced a paradigm shift with most of the current top 10 supercomputers of the world [37] using GPUs

✉ Valon Raca
valon.raca@univie.ac.at

Eduard Mehofer
eduard.mehofer@univie.ac.at

[1] Faculty of Computer Science, University of Vienna, Vienna, Austria

or other accelerators. Specialized processors with tremendous peak performance make it even for smaller institutions feasible to run computationally demanding jobs on cost-efficient clusters in-house instead of transferring their data to computing centers. Such mid-size clusters are often not assembled based on identical nodes but have an asymmetric hardware structure either from the very beginning or due to broken hardware. Whereas nowadays such systems are affordable even for smaller groups, programming support for heterogeneous, asymmetric clusters is getting more and more important.

Programming heterogeneous devices has been enabled and simplified by programming models such as CUDA [26], OpenCL [19], OpenACC [28] and OpenMP $\geq$ 4.0 [29]. Except for CUDA, these programming models offer a uniform programming approach to various types of compute devices. Basically, a data-parallel kernel written for a device can be easily ported to any other compute device which supports the very same programming model. As long as single device units are targeted, these programming models offer adequate support. However, distributing work over all compute devices of a heterogeneous cluster requires a multi-paradigm programming approach including MPI [25], OpenMP [29], or pthreads coupled with one of the previously mentioned heterogeneous programming platforms. Further, steering the execution of applications in an asymmetric cluster where different cluster nodes contain different compute devices requires new strategies for work partitioning and work distribution. It is a complex problem to split data-parallel applications into individually sized data portions to fit the devices and to achieve performance and energy efficiency. Distribution of work onto heterogeneous devices taking both performance and energy efficiency into account has been discussed in detail in [32].

In this paper, we propose a generic framework *clusterCL*, which is capable of handling a broad class of regular data-parallel applications and complex cluster configurations. Our framework *clusterCL* focuses on (1) enabling the programmer to pick a partitioning strategy while the framework automatically partitions the work and adapts partitioning sizes, (2) dispatching the work to targeted devices while minimizing the communication to execution time ratio, (3) distributing work taking execution time end energy consumption into account, (4) coordinating the execution in a cluster node while handling device failures and erroneous computations, and (5) steering the computing process for single- or multi-kernel applications while handling synchronizations, dependencies or data restructuring induced by the partitioning process.

The main contributions of this paper are:

- providing programming support for multi-kernel data-parallel applications in heterogeneous clusters
- efficient handling of communication and synchronizations stemming from partitioning of parallel applications with kernel dependencies
- ensuring workload balance between nodes in asymmetric heterogeneous cluster configurations
- minimizing device idle time with prefetching and coordination
- optimizing partitioning to get better performance.

The rest of the paper is organized as follows. Section 2 presents the design of the clusterCL Framework with its features that enable workload distribution in a computer cluster. Section 3 details our approach on workload partitioning and parameter tuning to achieve better speedup, while Sect. 4 describes in detail the comprehensive support of the clusterCL framework for handling various types of data-parallel applications with dependencies. In Sect. 5, we present the results of the experimental evaluation of our approach. We discuss the related work in Sect. 6 and conclude with a summary in Sect. 7.

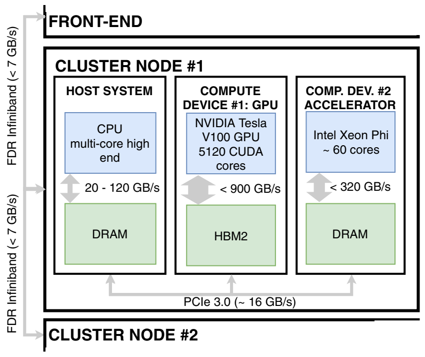
2 clusterCL framework design

In this section, we give an overview of the design of the framework and present the main components. A detailed technical outline of the main components is given in [31]. New features of the framework dealing with workload partitioning decisions and strategies as well as handling of partitioning-induced dependencies for complex HPC applications are discussed in depth in the later sections.

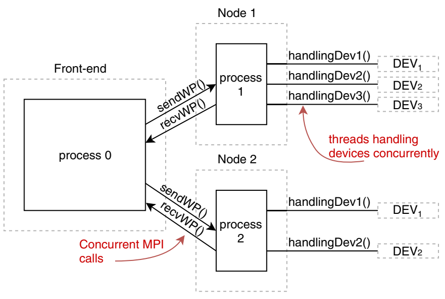
clusterCL is designed to support workload distribution onto heterogeneous asymmetric clusters for data-parallel HPC applications. Applications are typically composed of few kernels which are executed in a specific order over multi-dimensional data arrays.

The targeted system architecture is a distributed memory cluster consisting of front-end and cluster nodes. Cluster nodes host heterogeneous computing devices such as GPUs and other types of accelerators as shown in Fig. 1a. Cluster nodes need not be identical in configuration and may host variable amounts and types of compute devices. Data transfer is handled in two levels with different throughput rates: compute nodes are connected to front-end and each other via Fast Ethernet/InfiniBand, while compute devices are connected via PCIe to CPU and DRAM.

The software architecture of clusterCL as shown in Fig. 1b is based on MPI [25]. MPI handles communication and data transfers between the front-end and cluster



(a) Hardware Architecture



(b) Software Architecture

Fig. 1 Hardware and software architecture

nodes. The master MPI process running on the front-end is multi-threaded. For each compute node, a pair of threads on the front-end and cluster nodes handle bidirectional MPI communications concurrently. One thread feeds the cluster node with data chunks and organizational information (work packages), while the other receives partial results of the computation by the compute devices. The MPI process on cluster nodes runs OpenCL Runtime and steers the computing process on the devices. For each device on a node, a separate thread manages the data transfer, execution, and error-handling on the device.

Workload partitioning and distribution are the main tasks of the framework guided by the programmer. Figure 2 shows the three components of clusterCL, namely *supervisor component* running on the front-end, whereas the *coordination component* and *computation component* run on the cluster nodes.

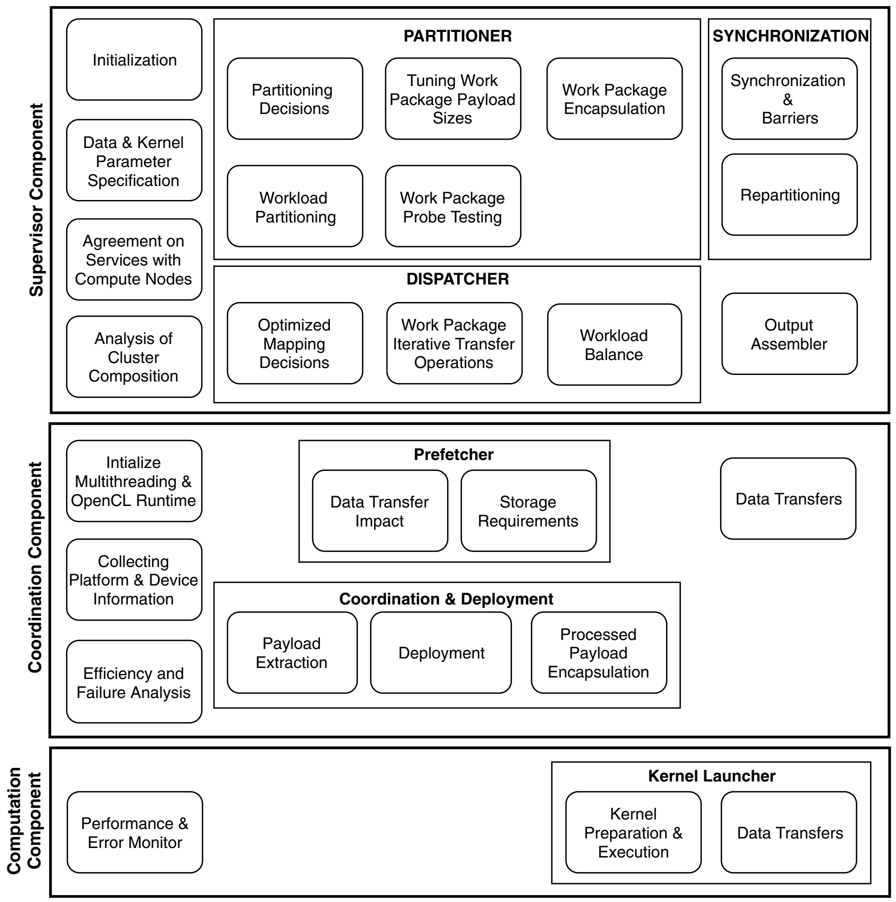

Supervisor Component
Initialization
Data & Kernel Parameter Specification
Agreement on Services with Compute Nodes
Analysis of Cluster Composition
PARTITIONER
Partitioning Decisions
Tuning Work Package Payload Sizes
Work Package Encapsulation
Workload Partitioning
Work Package Probe Testing
SYNCHRONIZATION
Synchronization & Barriers
Repartitioning
DISPATCHER
Optimized Mapping Decisions
Work Package Iterative Transfer Operations
Workload Balance
Output Assembler
Coordination Component
Intialize Multithreading & OpenCL Runtime
Collecting Platform & Device Information
Efficiency and Failure Analysis
Prefetcher
Data Transfer Impact
Storage Requirements
Data Transfers
Coordination & Deployment
Payload Extraction
Deployment
Processed Payload Encapsulation
Computation Component
Performance & Error Monitor
Kernel Launcher
Kernel Preparation & Execution
Data Transfers

Fig. 2 The clusterCL framework: main components and modules

Supervisor component Workload partitioning and distribution is handled by the supervisor component based on the data and kernel parameter specification provided by the programmer. An analysis of the cluster composition is evaluated based on the data provided by the cluster nodes. The *partitioner* handles workload partitioning by analyzing data domain specification and cluster composition. Data chunks created by this process are tested and tuned for different devices and encapsulated into *work packages (WP)* which are distributed to cluster nodes. The partitioning approach and strategies are discussed in more depth in Sect. 3.

A strategy on work distribution can be used to guide the execution process through the *dispatcher*. Optimized mapping decisions taking into account performance, energy efficiency or a trade-off between the two are used for efficient execution. This problem is discussed in detail in our paper [32].

The *synchronization* module typically deals with intra-kernel and inter-kernel dependencies. Intra-kernel dependencies occur as a result of the partitioning process where kernels might require access to data which are not readily available in the device memory, while inter-kernel dependencies occur in-between kernel launching operations in the course of multi-kernel execution sequence. Section 4 tackles in detail the problem of dependencies and synchronizations for HPC applications and the impact of partitioning strategies in the execution process.

Coordination component Work packages are received by the coordination component in the cluster nodes and dispatched to a compute device via the computation component which handles data transfers to the device and steers the execution. Performance monitoring and profiling of the compute devices is done primarily by the computation component, while the efficiency analysis is run by both the coordination and supervisor components.

The impact of the data transfer time on the overall execution time can be significant, especially for data-intensive applications. Considering that our model targets cluster systems, the process of preparing work packages from the process of execution in the devices is separated by two communication links: transfer from front-end to nodes and transfer from node to the device memory. The former is much slower than the latter and the main contributor to overall performance degradation. We address this challenge with a *prefetcher* to minimize the effect of data transfer over Fast Ethernet/Infiniband. The basic idea is to override the dynamic on-demand fetching of WPs from the front-end by storing additional WPs on the node to keep devices—ideally—always WP-busy. The prefetcher fetches additional WPs while devices on the nodes are executing. However, the number of stockpiled WPs has to be carefully calculated, since storing a larger number of WPs might result in workload imbalance between cluster nodes. The prefetcher calculates several factors such as the trade-off between device execution time and data transfer time, number of nodes used in the computing, speed of all links and the potential latency of front-end arising from serving several nodes.

Computation component Its basic function is to steer the kernel launching procedure and to collect the results of the computation process for each submitted work package. It is the job of the supervisor component to ensure load balance between cluster nodes by adjusting the scheduling strategy. In case of device failures, clusterCL removes the broken device from its dispatcher queue and reroutes work packages toward other devices including the unprocessed work packages at the time of failure. We define a processing failure whenever a device reports an error or becomes unresponsive. Our failure handling does not replace a checkpoint-rollback-recovery strategy, but our framework helps to reduce the number of expensive recovery actions [8, 9, 40].

3 Workload partitioning

Partitioning of work among all CPUs in homogeneous systems has been at the center of High Performance Computing since the beginning. Large cluster and supercomputers hosting thousands of CPUs are used to speed up the execution process by partitioning applications into chunks and assigning them to CPUs while aggregating the results received from CPUs. Many benchmarks that are used to test the performance of cluster systems and supercomputers follow the principle of work partitioning to utilize all system CPUs. The same principle applies to heterogeneous systems.

3.1 Partitioning approach

Workload partitioning in homogeneous systems is simpler than in heterogeneous systems. Having all of compute devices (CPUs) with the same characteristics, workload partitioning is reduced to partitioning the application in same-size chunks which are assigned to all CPUs. This automatically ensures workload balance among CPUs of a compute node and further among compute nodes themselves. Partitioning is more difficult for heterogeneous systems. In Fig. 3, we show the speedup of execution for different heterogeneous devices and a CPU compared to a baseline, which is the slowest device in the system. All devices are executing the same application with the same problem size.

From the figure, we see the huge disparity in execution time between devices. The heterogeneity and diversity between devices can influence the workload partitioning

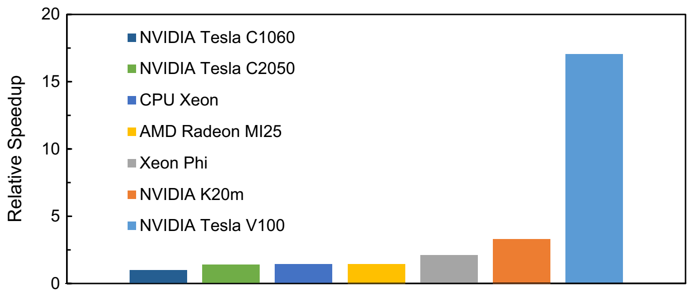


Fig. 3 Single device execution for correlation matrix routine [23] in different heterogeneous devices and CPU

greatly with significant consequences in the overall execution process. Many factors contribute to this. First, the devices differ in performance, memory organization, micro-architecture and other characteristics making the workload balance between the devices very challenging. Second, we target asymmetric clusters, consisting of different amounts and types of devices, making it difficult to achieve workload balance between the cluster nodes. Finally, the application's computational behavior influences both the partitioning strategies and execution time. For complex HPC parallel applications, partitioning induces communication between devices and nodes, which generates a lot of overhead in correctly and precisely mapping data accesses in a distributed memory architecture. Hence, a partitioning approach that addresses these issues is necessary.

In the domain of data-parallel applications workload partitioning and data partitioning usually point to the same thing. The general definition for data-parallel applications assumes that the same instructions apply to each data point of the application's dataset. Therefore, in most cases, workload partitioning reflects data domain partitioning. An exception is imbalanced applications [34], which due to application characteristics have different workloads per data point.

Most of the studies for single-node settings prefer custom-size partitioning of data domain [22, 24, 34]. In these papers, different strategies such as profiling techniques and machine learning are used to adapt the data chunks to device capabilities better, although for a very small number of devices, usually a CPU and a GPU. In [6], Beaumont et al. propose a method to support workload partitioning for a smaller number of heterogeneous devices; however, in another paper [7] they show that the problem of workload partitioning for heterogeneous platforms with many devices is NP-complete.

In general, data-parallel applications for heterogeneous systems can be partitioned using a *custom-size data chunks* approach or *same-size data chunks* approach. The first approach called *custom-size partitioning* partitions the data domain in variable data chunks sizes such that all devices get one data chunk to process. The amount of work assigned to a device has to be tuned or predicted to reflect the computing power of the device. This can be approximated based on the device's hardware characteristics, profiling information, or tuning. However, considering that the computing process in the devices is also influenced by the application structure, tuning workload partitioning to ensure workload balance can become quite complex. The second approach called *same-size partitioning* is similar to the strategy used in homogeneous systems where the work is partitioned in same-size data chunks. Since the partitioning process is the first step toward the execution of data-parallel applications in a multi-device cluster, it influences heavily the upcoming processes. We consider that in a cluster setting with a large number of devices, workload balance between cluster nodes should take precedence over the synchronized completion between devices. Therefore, a partitioning approach which shifts part of the problem of workload balance from the partitioning to distribution decisions might be useful.

We base our work on our partitioning approach, which breaks down data-parallel applications, respectively their datasets into medium-grained same-size-fits-all data chunks. Contrary to the custom-size partitioning, our approach partitions the application such that the number of data chunks created by this process is always larger and in most cases orders of magnitude larger than the number of the devices in the system. Considering the big amount of data chunks to be distributed in a heterogeneous asymmetric cluster, workload balance becomes a problem of distribution strategies which have to ensure that faster devices get more data chunks than slower ones, balancing the work between devices and further between compute nodes of the cluster.

Our workload partitioning approach produces same-size medium-grain data chunks, which are encapsulated into *work packages (WP)*. We use this term to refer to this type of data chunks through the rest of this paper. Work packages are composed of a header and payload resembling the TCP/IP networking packets. The payload may retain the n-dimensional structure of data akin to the original data domain.

In Fig. 4 is shown a work package created by the workload partitioning process. The header includes information and instructions for the work package, which is essential for routing and failure handling. Each work package is given an ID which is used for logging and tracing by the runtime. Partitioning information stores data about the number of original dimensions, size of the payload and offset of the payload which indicates the displacement from the origin of the original data domain. This is usually required by the OpenCL kernel where the computation is based on the spatial coordinates. Routing instructions are used to set the target node/device by the scheduler and is updated by the runtime for rerouting purposes when a device failure occurs. Work package status is an enum updated whenever a work package is conveyed from one stage to another, while iteration count keeps the number of iterations completed for iterative applications and applications with dependencies. Finally, device profiling information is stored by the computing component after each successful work package computation and conveyed to the scheduler (dispatcher).

In Table 1, we have shown some of the advantages of using the work package approach for programming heterogeneous clusters reflecting our experience. The main advantage of this work package approach is that achieving workload balance between cluster nodes is not entirely related to the partitioning sizes but also to the work distribution strategy which is more manageable. Partitioning is very complex

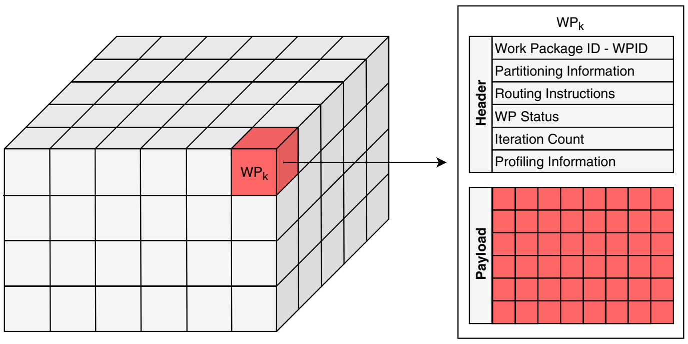


Fig. 4 Workload partitioning: work package

Table 1 Complexity of handling different tasks with work package and custom-size approaches

Task	Work package	Custom-size data chunks
Partitioning and workload balance	Medium	Very complex
Device failures	Simple	Complex
Poor performance	Simple	Complex
Optimizations for performance and energy efficiency	Complex	Extremely complex

with the custom-size approach since it needs tuning of many hardware, software and application-centric characteristics to achieve workload balance between devices.

The work package approach is superior also when it comes to handling device failures and erroneous computations. Unlike in the custom-size partitioning approach, a cluster node discovering a device failure can reroute the work package to another device without having to deal with device capacity or other hardware limitations, since all work packages fit in all devices of the system. Further, it is also better at handling poor performance of the devices. A non-performing device can be removed from computation and work packages rerouted to other devices. Besides the workload balance and handling failures, the work package approach excels at another very specific aspect. Since all work packages are the same size, we can measure the execution time and energy consumption of a work package for each device type of the system. With the profiling information, we can devise work distribution strategies that optimize for energy efficiency in addition to performance as we have shown in [32]. This is extremely complex with the custom-size approach since the optimization problem becomes very complex and changing the size of the chunk affects both execution time and energy consumption.

3.2 Partitioning strategies

In the previous section, we have dealt with partitioning workload from the perspective of hardware performance numbers by adjusting the chunks to ensure workload balance in asymmetric heterogeneous clusters. Another aspect that is crucial to the performance of data-parallel applications is the application structure. Not all datasets of an application can always be partitioned since data access patterns of an application can affect this process profoundly. Partitioning some datasets might transform applications in such a way that the intensity of data transfers or communications is increased to the point where it dominates execution and makes partitioning infeasible.

The execution process will be influenced depending on *what* we decide to partition. This decision might introduce synchronizations and barriers or communications between devices that are required to keep data coherency during the execution.

Depending on application characteristics, and especially the data accesses of application's kernels, a decision has to be made whether it makes more sense to partition input, output, or both data domains. For instance, if the data access map is scattered over the input data domain for each output data point, perhaps it is more

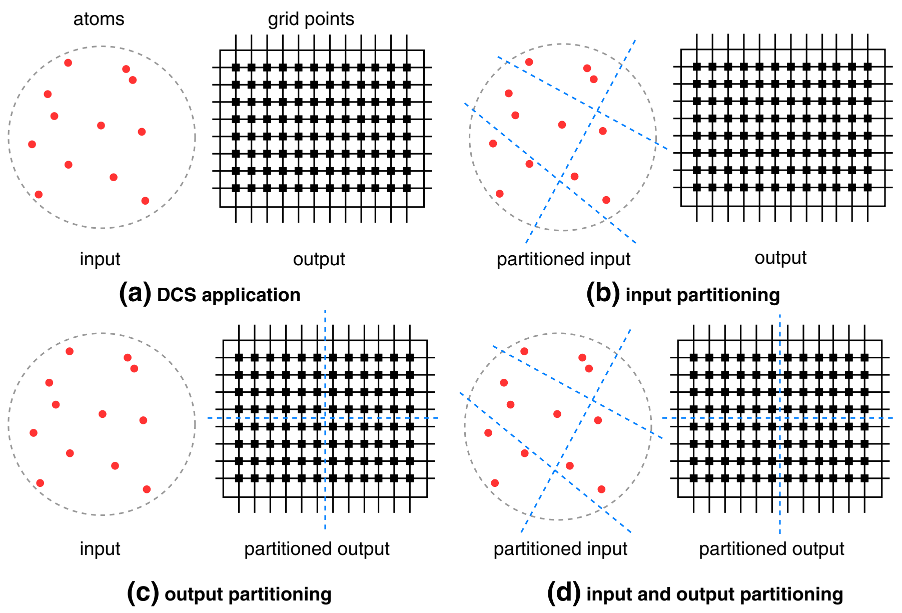


Fig. 5 Partitioning strategies for DCS

sensible to partition the output domain and replicate the input over all devices or vice versa. To illustrate this process, we present in Fig. 5 the partitioning strategies for the Direct Coulomb Summation (DCS).

DCS kernel has two datasets: input data domain which contains data about the atoms and output domain which reflects the spatial positions of the grid points and accumulates the influence of the electrostatic field at that point (see Fig. 5a). There are three possible partitioning strategies, including partitioning only input, only output or both data domains. Which decision would harvest more performance for heterogeneous devices is hard to estimate without analyzing the computational behavior of the application. In DCS, for each grid point, the kernel calculates distances from the grid point to atoms and computes the influence of atom's charges to that grid point. The partitioning strategy to be taken in this case depends on the size of data domains, e.g., the number of atoms or grid points. The complexity of operations that are performed on data also plays an important role. As shown in Fig. 5b, if the workload partitioning is based on input data, the output arrays will be replicated in all devices, while the work packages generated by input partitioning are distributed in different devices. The output arrays resulting from devices are aggregated at the end of the computing process. Conversely, as shown in Fig. 5c, if output array is partitioned, input arrays are replicated to the devices, while output work packages are distributed in devices. Output work packages are finally merged at the end of the computing process. Partitioning of both data domains, in this case, is also possible (see Fig. 5d).

For other applications that consist of many input and output data domains, the partitioning can be partial or complete in relation to input and output. In the case of partial partitioning, only a few or some of the input and/or output arrays are partitioned, whereas complete partitioning translates as either all input and/or output arrays are partitioned. We discuss the partitioning strategies for a broad range of HPC applications in Sect. 5.2.

3.3 Spatial partitioning

HPC applications usually consist of multi-dimensional datasets. The partitioning process retains the multi-dimensionality of the dataset, although it enables partitioning in a smaller number of dimensions than the original data domain. For instance, as shown in Fig. 6 three-dimensional datasets can be partitioned in many ways guided by the programmer and dependent on the application: as smaller cuboids, as two-dimensional slices, two-dimensional blocks partitioning through columns and rows, two-dimensional blocks that include complete rows, two-dimensional blocks that include complete columns or as one-dimensional arrays.

Tuning work package payload sizes An initial work package payload size can be tuned further to increase the performance of execution per work package. The initial probe (sample of work package payload) is tested in a selected number of devices of the cluster, each representing a device type. Profiling information is gathered which along with the execution time and data transfer times also contains power consumption data. Power consumption data are used when the scheduling strategy is a multi-objective optimization problem taking into account both execution time and energy consumption. Decreasing or increasing the size of the payload may result in performance gain in terms of overall execution performance.

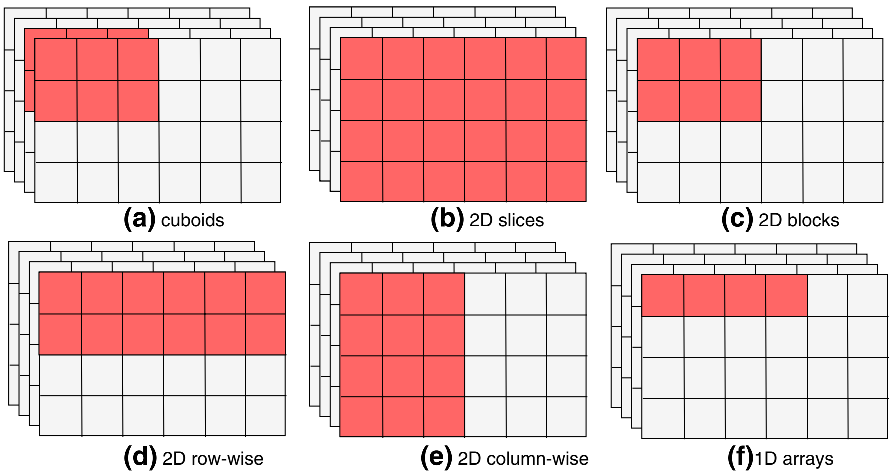


Fig. 6 Partitioning multi-dimensional arrays

Another issue is that different devices may often yield highly different execution times depending on computational behavior of the application and hardware characteristics. The tuner, in this case, aims to get a better balance in execution times between devices by testing a set of finite payload sizes while maintaining the workload balance between cluster nodes with scheduling strategies.

Experimental data with regard to tuning work package payload sizes are presented in more detail in Sect. 5.5. Our observations show that tuning work package payload sizes is highly beneficial, especially for certain types of applications such as imbalanced applications where the workload among work-items varies.

4 Programming support for partitioning-induced dependencies of HPC applications

In [31], we have shown how clusterCL enables execution of single-kernel HPC applications, which generally require no data transfer between the devices—a class of data-parallel applications similar to a bag of tasks applications. In this section, we extend our approach to support work partitioning and distribution for more complex data-parallel applications. We have examined many HPC routines and real-world HPC applications to come up with some general characteristics of applications.

Let us first discuss a general case of multi-kernel execution process using clusterCL. Usually, data-parallel programs consist of a few kernels which are executed in a serial fashion, different from task-parallel programs which frequently have a much more complicated execution sequence. Let us assume we have an application with two kernels $k_1(a, b)$ and $k_2(c, d, e)$ where a, c are input non-partitioned (replicated) arrays for the kernels, d is a partitioned input array $d_1, \dots, d_n$ which is modified by kernel k_2 and b, e are partitioned output arrays $b_1, \dots, b_n; e_1, \dots, e_n$, respectively, for kernels k_1 and k_2. We assume that work has been partitioned into n work packages and that the cluster hosts m devices. The number of work packages is bigger than the number of devices, i.e., $n \gg m$.

Figure 7 illustrates the execution process. There are two nodes, with the first node having two devices and the second node having one device. Data transfers

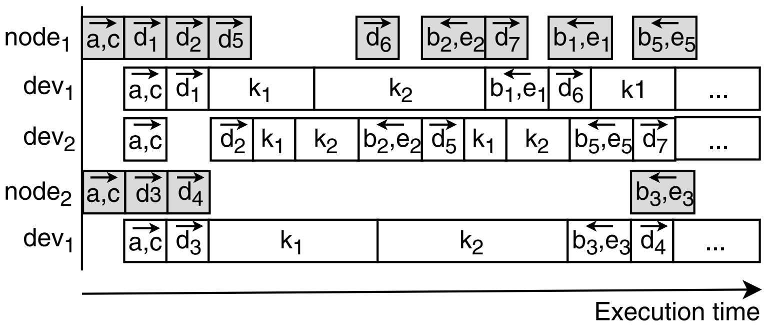


Fig. 7 Multi-kernel execution with clusterCL

are marked with arrows. With → we have marked data transfers to nodes or devices while with ← are marked data transfers from devices to nodes or to front-end. Darker rectangles represent data transfer from front-end to cluster nodes and vice versa, while lighter rectangles containing arrows represent data transfers from cluster node to the device and vice-versa. Execution process of work packages in devices is marked with rectangles that contain no arrows, depicted by k_1 and k_2

As shown in the figure, first we transfer non-partitioned input arrays a, c which are replicated to all devices. This is followed by the partitioned input array d in the form of work packages which are transferred on-demand basis or using a scheduling algorithm. Finally, computed output arrays b, e are transferred to front-end in the form of processed work packages. For the sake of simplicity, the figure shows the inbound and outbound data transfers as serial, although clusterCL implementation is configured to support *MPI_THREAD_MULTIPLE* for cluster-level communication which enables concurrent MPI calls, and out-of-order queuing and execution. This enables overlapping of data transfers from nodes to devices with kernel execution, i.e., hiding communication time. We can observe that device dev_2 is the fastest, while device dev_1 of the second node the slowest. Using the prefetcher, a few work packages are stockpiled on each node, which ensures smooth operation and minimizes device idle time.

In the course of execution of this application, two issues of great importance have to be handled:

- *intra-kernel dependencies*
- *inter-kernel dependencies*

With intra-kernel dependencies, we define data dependencies resulting from kernel data access requirements after partitioning, e.g., kernel k_2 processing work package d_1 in device dev_1 of $node_1$ requires access to data packed in work package d_3 which is processed by dev_1 in $node_2$, or it requires access to data in work package d_8 which hasn't been dispatched yet by the front-end. In both cases, data are not readily available in the dev_1 memory. This computational behavior represents intra-kernel dependencies, where the same kernel running on all devices requires reading or writing to a memory space it does not manage.

With inter-kernel dependencies, we define dependencies which result from data sharing between two different kernels of the same application. In this case, a succeeding kernel may have a data access map that is different from the data access map of the first kernel. For instance, kernel k_2 which is executed immediately after kernel k_1 may require access to some data produced by kernel k_1, but since the kernel k_2 has a different data access map, required data are not readily available in the device memory. This behavior represents inter-kernel dependencies and can be exhibited by multi-kernel applications.

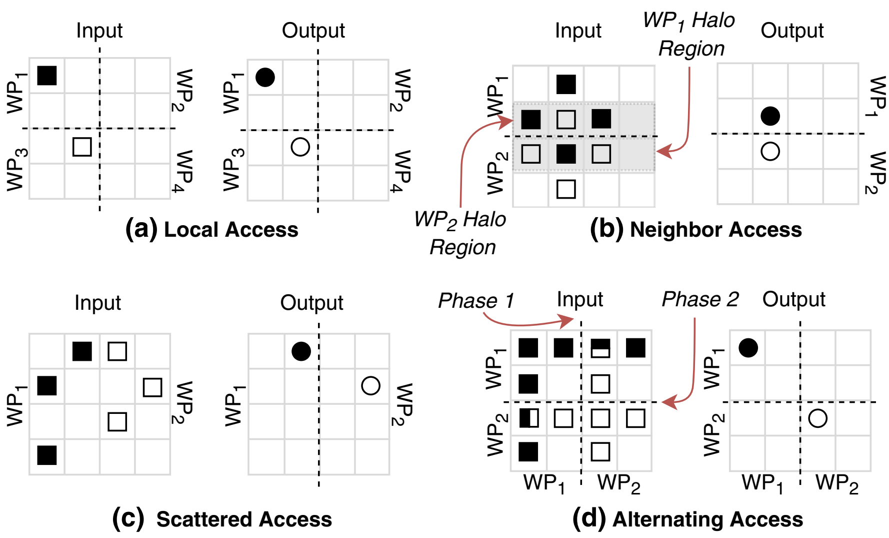


Fig. 8 Data access patterns

4.1 Intra-kernel dependencies

Figure 8 shows various cases of data access patterns that data-parallel kernels may exhibit. With squares we denote input data, while with circles output data generated by the execution process. Items accessed by the same thread have the same color.

Considering our work package approach, communication between devices is not only expensive in terms of performance, but also difficult to manage. Our goal is to analyze data access requirements of HPC kernels after the partitioning process, in order to be able to devise workload and kernel partitioning strategies which would result in no or minimal communication.

Local access Some types of data-parallel applications have a high degree of parallelism. As shown in Fig. 8a, each element of the output array is generated by an element of the input array with the same index. We have used 2D-block partitioning, creating four work packages out of input and output domains, which can be scheduled independently. This case can be extended further such as if the whole row of a two-dimensional array is used to produce an element of the output array. For such applications, workload partitioning is simple and data to be accessed by the kernel code are always packed in the local work package which is available in the device memory.

Neighbor access Some kernel codes require access to neighboring elements after data partitioning. These elements are packed in other work packages. In Fig. 8b, we have shown a case where neighboring elements are used for calculation of each output element. In such cases, there does not exist a partitioning method for which the kernel code could be transformed to access only local work package data. However, as depicted with a gray box in Fig. 8b, the so-called *halo regions* can be used in these cases. For each work package created by the partitioning process, the neighboring elements of the work package which need to be accessed by the kernel code are packed together with the work package and deployed to the devices. All output work package elements can be calculated without data transfers between devices, although, a minor overhead for transferring halo regions is to be expected. In most cases, halo regions account only for a very small fraction of data compared to the amount of data packed in work packages.

Scattered access In some cases, the kernel code has a scattered data access map. The kernel code, as shown in Fig. 8c, requires access to distant elements that cannot be covered by halo regions. A stepwise approach to this problem is required. First, based on data access patterns, we look for a spatial partitioning method that can transform the access requirements. In some cases such as in Fig. 8c, data accesses are scattered and irregular but confined to a region of the array. Black squares are located in the left part, while white squares in the right part. However, if data are scattered in such a way that no partitioning method results in local access, we next analyze potential partitioning strategies, i.e., which of the data domains to be partitioned makes more sense. For instance, if the input array (see Fig. 8) can fit into the memory of the devices, the input array can be replicated to all devices, while the output array is partitioned into work packages. In this case, for each output work package, we will have all input data available in all devices, while the workload has been split through output partitioning. Finally, in rare cases where the previous approaches do not overcome the problem, both input and output domains have to be partitioned regardless of the access patterns. For each output work package, the kernel is executed over all input work packages (instead of one, as in previous cases) to access all data it requires to produce output elements. The kernel code has to be adapted by the programmer to compute and aggregate the partial results. The last approach is used only as a last resort solution, where the input data are too big to fit in the device memory.

Alternating access Some kernels sweep over different regions of the array in phases. For instance, in Fig. 8d the kernel that produces an element of the output array sweeps over all elements of one row, to switch later to sweeping through all the elements of one column. In these cases, splitting the kernel code into two or more kernels that run in sequence (kernel fission) can transform the access requirements to local work package access. In the case shown in the figure, in the first phase work packages are created based on a row-wise partitioning of data. A synchronization barrier is inserted after the execution of the first kernel, resulting in the merging of the work packages in front-end. New work packages are created based on a column-wise partitioning and dispatched to devices where the output work packages are updated.

4.2 Inter-kernel dependencies

Adapting clusterCL to support multi-kernel applications requires implementing features that handle together workload partitioning and the kernel execution schedule.

In this section, we focus on inter-kernel dependencies where kernels of an HPC application executing in a serial sequence share data between each other.

Inter-kernel dependencies can be handled more simply than intra-kernel dependencies. In the context of intra-kernel dependencies, accessing data in other work packages occur during the execution process of the kernel. Data transfers to support kernel access patterns in multi-kernel execution occur at well-defined points between explicit calls for kernel execution and can be managed by the CPU with synchronization mechanisms or work package rerouting.

We refer to Fig. 7 to illustrate inter-kernel dependencies in the context of multi-kernel execution. Let us assume first that kernel k_1 and k_2 do not share data between each other, i.e., no inter-kernel dependencies. Assuming no intra-kernel dependencies exist the execution process is translated into a sequence of kernels each accessing its own local data in the devices where they are being executed. In this case, kernel k_1 needs a to produce $b_i, 0 < i < n$; therefore, a is first transferred, while kernel k_2 needs c and $d_i, 0 < i < n$ to produce $e_i, 0 < i < n$. c is transferred in the beginning, while work packages d_i are dispatched to devices before the beginning of execution for k_2.

In most cases, the succeeding kernel requires the output of the preceding kernel to produce its results. In cases where kernels share data between them, the sequence of the execution is not disrupted as long as the data access maps of the kernels match. For instance, if kernel k_2 uses output array b_i to update values of d_i, such that each element of d_i is updated by the corresponding element of the b_i, the update process is confined to local access and we get $d_1 \leftarrow b_1, \dots, d_n \leftarrow b_n$. Since work package b_i is already in the device memory as a result of the execution of the kernel k_1, k_2 can access them directly to update work packages d_i. In other cases, a synchronization barrier (inter-kernel synchronization) is imminent.

Let us assume that output b from kernel k_1 is used as input for kernel k_2. According to the partitioning strategy, we have used in the case depicted in Fig. 7, c is a non-partitioned input array; therefore, work packages of b cannot replace c for kernel k_2, since work packages of b are spread over all devices, while c is replicated to all devices. If no other partitioning strategy is possible, a synchronization barrier between kernels is required to ensure merging of work packages $b_1, \dots, b_n$ into b at front-end, update and dispatch of c to devices in order to provide data coherence before the beginning of execution of kernel k_2.

Although a synchronization barrier appears like a diminishing factor in terms of performance, using the work package approach, its impact is reduced. In the custom-partitioning approach, the number of data chunks produced by partitioning is the same as the number of devices in the system, while their size is different assuming the workload balance between devices needs to be ensured. Since all the data have to be transferred at once, besides merging and repartitioning synchronization is heavily dependent on communication required to send back data to front-end and from front-end to devices after repartitioning. On the other hand, using the work package partitioning approach, as shown in Fig. 7, processed work packages are immediately sent to the front-end, while in meantime the devices are processing other work packages. An exception are the last work packages before synchronization and the first work packages after synchronization. However, this is only a fraction of the communication time to send all the data at once.

A special case of HPC applications that exhibit similar dependencies as multi-kernel applications with inter-kernel dependencies is applications with iterative kernels. They usually consist of one or more kernels that have to be computed iteratively until some condition is true. For these applications, dependencies may occur at the iteration step. Hence, a synchronization barrier at the iteration step is required where merging and repartitioning of data is done.

5 Experimental evaluation

In this section, we present and discuss the results of the evaluation of *clusterCL* framework with 11 applications from different benchmark suites executed on three clusters.

We focus our evaluation on the following aspects of the clusterCL framework:

- The applicability of our approach is examined in Sect. 5.3 where the speedup figures for symmetric clusters are analyzed.
- Workload balance among cluster nodes, including the impact of data transfers in heterogeneous asymmetric configurations, is examined in Sect. 5.4.
- The benefits of partition tuning for WPs are shown in Sect. 5.5.

Table 2 Cluster systems

Cluster	# of nodes	Interconnection	Nodes	Configuration
PHIA	8	InfiniBand QDR (40 Gbps)	1–6	CPU: 2 × Intel Xeon E5-2650
				GPU: 2 × NVIDIA Tesla K20m
				PHI: 2 × Intel Xeon Phi 5110P
			7	CPU: 2 × Intel Xeon E5-2650
				GPU: 4 × NVIDIA Tesla K20m
			8	CPU: 2 × Intel Xeon E5-2650
				PHI: 4 × Intel Xeon Phi 5110P
EXA	4	InfiniBand FDR (56 Gbps)	1	CPU: 4 × Intel Xeon Gold 6138
			2	CPU: 2 × AMD EPYC 7501
			3	CPU: 2 × Intel Xeon Gold 6130
				GPU: 1 × Nvidia Tesla V100
			4	CPU: 2 × Intel Xeon Gold 6130
				GPU: 1 × AMD Rad. Inst. MI25
CORA	7	Ethernet (1 Gbps)	1–7	CPU: Intel Xeon X5550
				GPU: 2 × NVIDIA Tesla C2050
				1 × NVIDIA Tesla C1060

5.1 Hardware setup

In our experiments, we use 3 clusters: PHIA, EXA, and CORA. Specifications of the cluster systems used to evaluate *clusterCL* are summarized in Table 2. All clusters are heterogeneous. Clusters PHIA and EXA have an asymmetric configuration with cluster nodes hosting different types of compute devices, while CORA is symmetric.

PHIA is interconnected with Infiniband QDR (40 Gbps) and consists of 8 nodes. The first six nodes have an identical configuration hosting NVIDIA K20m and Xeon Phi along with the Intel CPUs. Nodes 7 and 8 host K20ms or Xeon Phi only. EXA is interconnected with InfiniBand FDR (QSFP, 56 Gbps) and consists of 4 compute nodes with Intel and AMD CPUs, NVIDIA Tesla V100 and AMD Radeon Instinct MI25 GPUs. CORA is interconnected with Fast Ethernet and consists of 7 nodes that host NVIDIA Tesla C2050 and Tesla C1060 GPUs. The batch system used on PHIA and EXA is SLURM [39], while on CORA we use Sun Grid Engine [13].

We use OpenMPI 4.0.1 in EXA and CORA, and MVAPICH 2.2 in PHIA for inter-node communications. For NVIDIA devices, we use NVIDIA's OpenCL implementation, for AMD devices—rOCM platform, and for Intel devices Intel's implementation of OpenCL.

Cluster configurations We have used different cluster configurations to examine the behavior of the clusterCL framework for our evaluation objectives. As shown in Table 3, PSC1–PSC6 are six symmetric configurations of the PHIA cluster consisting of 1 to 6 nodes, respectively, where we use all compute devices (CPUs, K20m, Xeon Phi) to run experiments.

Other configurations are asymmetric, with PAC1, PAC2, and PAC3 consisting of nodes 6, 7 and 8 of PHIA cluster. In PAC1, we use GPUs and Xeon Phi which are distributed unevenly among PHIA nodes 6, 7 and 8. In PAC2, we use only GPUs, and therefore node 8 is not used since it does not contain any GPU. In PAC3, we use only Xeon Phi, and therefore node 7 is discarded since no Xeon Phi are installed in that node. EAC1 is an asymmetric configuration of the EXA cluster consisting of nodes 3 and 4 with their respective compute devices, NVIDIA Tesla V100 in node

Table 3 Experimental configurations

Configuration	Cluster	Arrangement	Details
PSC1	PHIA	Symmetric	PSC1: Node 1 (all devices)
⋮	⋮	⋮	⋮
PSC6	PHIA	Symmetric	PSC6: nodes 1–6 (all devices)
PAC1	PHIA	Asymmetric	Nodes 6–8 (GPUs and Xeon Phi)
PAC2	PHIA	Asymmetric	Nodes 6–8 (only GPUs)
PAC3	PHIA	Asymmetric	Nodes 6–8 (only Xeon Phi)
EAC1	EXA	Asymmetric	Nodes 3 and 4 (GPUs)
CAC1	CORA	Asymmetric	Node 1 (all GPUs), Node 2 (only Tesla C2050s), Node 3 (only Tesla C1060)

Table 4 List of HPC applications ported to clusterCL

Application	Code	Domain
Direct Coulomb summation	DCS	Scientific simulation
Correlation matrix	COR	Data mining
Covariance matrix	COV	Data mining
Binomial options pricing model	BOP	Finance
Matrix–matrix multiplication	GEMM	Linear algebra
Gram–Schmidt process	GSP	Linear algebra
Rank-K matrix operations	SYRK	Linear algebra
Black–Scholes model	BSM	Finance
Convolution	CONV	Scientific simulation
Biconjugate gradient	BICG	Linear algebra
Scalar, vector and matrix multiplication	GSMV	Linear algebra

3 and AMD Radeon Instinct MI25 in node 4. In CAC1, we have constructed an artificial asymmetric configuration by utilizing all three GPUs (two Tesla C2050 and one Tesla C1060) in the first node, followed by two devices in the second node (only Tesla C2050s) and one device in the third node (only Tesla C1060).

5.2 HPC applications and partitioning strategies

We have analyzed and tested 11 HPC applications from different benchmark and application suites: AMD Accelerated Parallel Processing SDK [2], NVIDIA OpenCL SDK [27], and PolyBench [15]. Benchmark applications are summarized in Table 4. We have adapted these applications and their kernels to allow for multi-device computation.

In the following, we discuss in more detail the partitioning strategies and kernel scheduling techniques for the applications listed in Table 4.

Direct Coloumb summation DCS is used to calculate the electrostatic field for a given n-dimensional grid space based on the charges of the atoms within that grid space. It consists of an input array that stores atom characteristics and an output array that stores data about the energy grid. For this application, either the input or the output domain can be partitioned while the mixed partitioning is impractical. Whichever domain is partitioned, the other must be replicated to all devices.

Correlation matrix and covariance matrix COR and COV compute the correlation, respectively covariance matrix. Since the first three kernels are lightweight, they can be executed by all devices, while the fourth kernel requires output domain partitioning. Output domain is a triangular matrix making column-wise partitioning and work package payload size tuning highly advisable for better performance.

Binomial options pricing model BOP calculates binomial options pricing for European Options. Output domain partitioning is the most beneficial partitioning strategy.

Matrix–matrix multiplication GEMM is a typical BLAS routine. Partial input and output domain partitioning is the most beneficial partitioning strategy since a complete partitioning of both input and output introduces redundant iterations over work packages slowing down the execution.

Gram–Schmidt process GSP is a linear algebra application used for orthonormalization of a set of vectors in an inner product space. A complete partitioning strategy is possible. However, an inter-kernel synchronization barrier is required before the execution of the third kernel. A time-step loop over the multi-kernel sequence is necessary, although it does not affect the synchronization or the data transfer process.

Rank-K matrix operations SYRK is a linear algebra application used for symmetric rank-k update operations. A partitioning strategy similar to GEMM is appropriate.

Black–Scholes model BSM is an implementation of the option pricing model for European Options in financial engineering. A complete partitioning of input and output arrays is appropriate.

Convolution CONV is an application used in the field of neural networks. Our implementation is based on a 2D version of the application. A complete partitioning of input and output arrays is appropriate.

BiConjugate gradient method BICG is a linear algebra application. A complete partitioning of input and output arrays is appropriate.

Scalar, vector, and matrix multiplication GSMV is a linear algebra application. Partial input domain and complete output domain partitioning are possible.

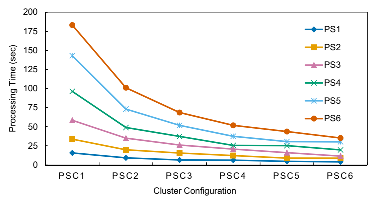
5.2.1 Applications with Different Problem Sizes.

For the first two experiments, we have used six different problem sizes (PS1–PS6), with problem sizes in the lower end (PS1–PS3) that can fit into the smallest device memory and problem sizes in the higher end (PS4–PS6) that cannot be computed without partitioning the data domain between devices.

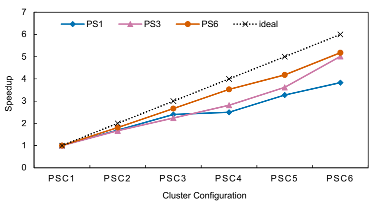
5.3 clusterCL speedup analysis

In this experiment, we compare the performance of clusterCL, the processing times and the speedup with different problem sizes and cluster configurations. Since the application workload is partitioned into many work packages, we use *processing time* to denote the maximal accumulated handling time for all work packages assigned to the devices of the system. Handling time includes writing data from the node to the device, execution time in the device and time required to read the results from the device to the cluster node.

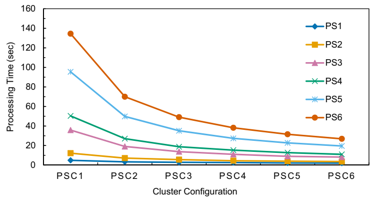
Experiments are carried out in the PHIA cluster for cluster configurations PSC1–PSC6. For the speedup, we use single-node configuration (PSC1) as the baseline to test the flexibility of the framework in work distribution for several compute nodes with identical configuration (PSC2–PSC6, see Table 3). A dotted line is shown in all figures to denote the ideal speedup figures (theoretical peaks for each configuration).



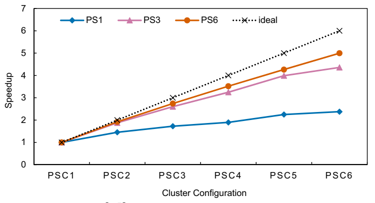
(a) BOP: Processing Time



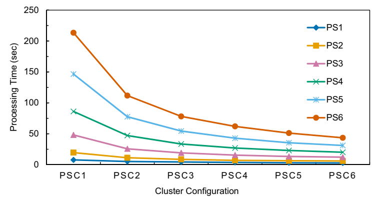
(b) BOP: Speedup



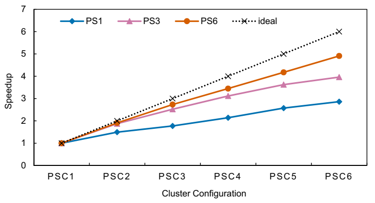
(c) GEMM: Processing Time



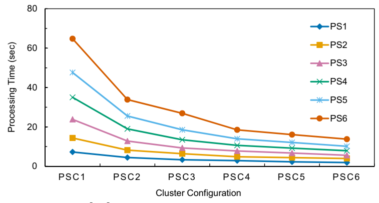
(d) GEMM: Speedup



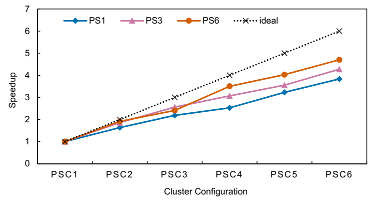
(e) COV: Processing Time



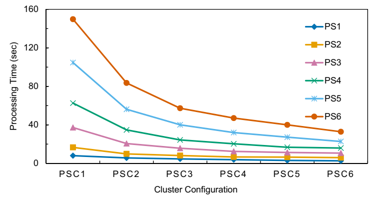
(f) COV: Speedup



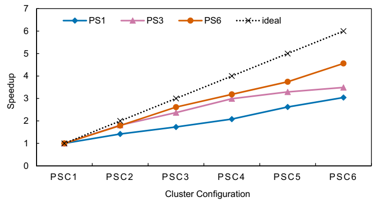
(g) DCS: Processing Time



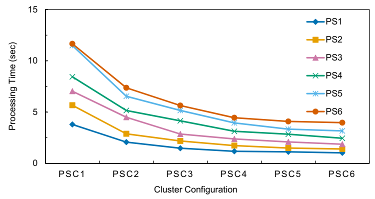
(h) DCS: Speedup



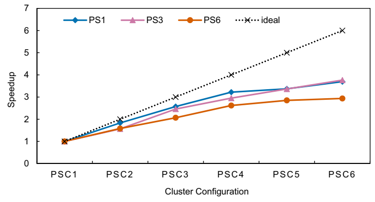
(i) COR: Processing Time



(j) COR: Speedup



(k) BICG: Processing Time



(l) BICG: Speedup

Fig. 9 clusterCL: processing time and speedup for HPC applications in symmetric heterogeneous clusters with different cluster configurations

In Fig. 9a, we show the processing times for BOP. Different problem sizes from the smallest size PS1 to the biggest PS6 are plotted using different curves, from the smallest problem size processing times (the bottommost curve) to the biggest problem size (the uppermost curve). Processing times (vertical axis) are given for each of the cluster configurations (PSC1–PSC6) (horizontal axis). Figures on the right such as Fig. 9b show the relative speedup compared to baseline for problem sizes (PS1, PS3, and PS6)—we have omitted problem sizes PS2, PS4, PS5 for the sake of readability. For most of the applications shown in Fig. 9, i.e., Fig. 9a–j, the speedup numbers approach the ideal speedup, especially with bigger problem sizes which show the strongest performance gain. BOP speeds up to 5.17x over the baseline version, GEMM, COV, DCS, and COR up to 4.99x, 4.91x, 4.7x, and 4.55x, respectively, in a PSC6 configuration for the biggest problem size PS6. The gap between the ideal speedup and real speedup figures is mainly due to inter-node data transfers. While for single-node execution there are no WP transfers to other nodes, in other configurations transfer of the WPs cannot be hidden by the prefetcher until the execution begins in the devices. Upon the start of the computing process in the devices, the prefetcher transfers other work packages while the devices are busy executing, effectively minimizing the packet switching time. We observe from the speedup figures that this gap is bigger for small problem sizes and smaller for bigger problem sizes. As the problem size increases, so does the number of work packages created by the partitioning process and the overall execution time. Since we have the same configurations and the same number of devices, data transfer time for the first work packages which cannot be hidden by the prefetcher stays the same. Therefore, the impact of the time required to transfer first work packages in overall execution time is reduced proportionally with the increase in the number of work packages. It is worth noting that in all of these experiments we can observe that the speedup curves for bigger problem sizes are projected in such a way that strong performance is expected to be sustained also for system configurations with more identical nodes and bigger problem sizes.

In Fig. 9d, j for GEMM and COR for problem size PS3, it appears like speedup figures diminish as the scale of the system gets bigger. Our observations show that this is due to scheduling. For this experiment, we are using a work-stealing strategy where the device which finishes first gets the next work package. Big differences in execution times among compute devices (e.g., CPU executes a WP seven times slower than the GPU for GEMM) result in workload imbalance due to the small number of work packages. In both cases, the CPU is getting one of the last WPs and while other devices complete execution, the whole process is delayed waiting for the CPU to finish processing. This can be seen in the figures which show the speedups for GEMM for configuration PSC6, and for COR for configurations PSC5 and PSC6. This problem can be solved by applying a static scheduling algorithm such as the one proposed in our paper that deals with scheduling problems in heterogeneous clusters [32].

In Fig. 9k, l, we present processing time and speedup for BICG over the baseline. As shown in the figure, the speedup over the baseline is around 3x. These figures also resemble the results for GSMV, BSM, CONV and SYRK applications which we omitted for readability reasons. In Fig. 9l, we see that for small problem sizes the speedup is better than for bigger problem sizes, contrary to the other applications shown in Fig. 9. These applications fall into the category of data-intensive applications with low computing intensity. A big ratio of data transfer time over execution time hampers the distribution process and inhibits the prefetcher to supply enough work packages in time. In these cases, using work package payload size tuner in conjunction with increasing thread granularity can improve performance.

5.4 Workload balance, prefetcher efficiency and impact of synchronizations

In this section, we examine the efficacy of our approach in guaranteeing workload balance between cluster nodes for asymmetric configurations of heterogeneous clusters. Additionally, we show the work share between different types of devices and

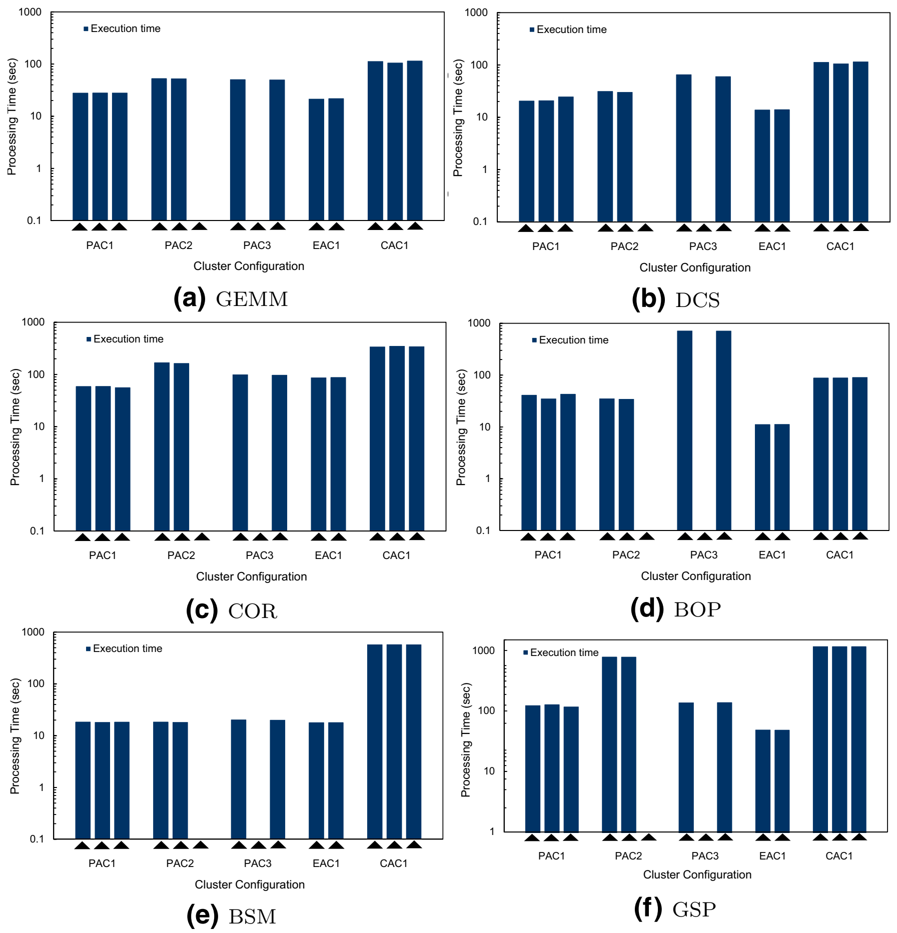


Fig. 10 clusterCL: workload balance between cluster nodes in various asymmetric cluster configurations

Table 5 GEMM: work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Avg. proc. time per WP (s)	
	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi
PAC1	51.12%	48.88%	99.90%	99.70%	0.70	0.59
PAC2	100.00%	–	99.94%	–	0.70	–
PAC3	–	100.00%	–	99.70%	–	0.59
	Tesla V100	AMD MI25	Tesla V100	AMD MI25	Tesla V100	AMD MI25
EAC1	77.46%	22.54%	100.00%	100.00%	0.06	0.21
	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060
CAC1	83.04%	16.96%	93.90%	96.89%	0.55	1.45

Table 6 DCS: work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Avg. proc. time per WP (s)	
	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi
PAC1	66.96%	33.04%	98.95%	99.07%	1.59	3.23
PAC2	100.00%	–	99.68%	–	1.59	–
PAC3	–	100.00%	–	99.74%	–	3.23
	Tesla V100	AMD MI25	Tesla V100	AMD MI25	Tesla V100	AMD MI25
EAC1	62.50%	37.50%	99.90%	99.97%	0.05	0.08
	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060
CAC1	89.29%	10.71%	99.63%	99.00%	2.16	9.53

Table 7 COR: work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Avg. proc. time per WP (s)	
	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi
PAC1	49.78%	50.22%	99.81%	99.75%	1.53	1.26
PAC2	100.00%	–	99.93%	–	2.23	–
PAC3	–	100.00%	–	99.90%	–	1.24
	Tesla V100	AMD MI25	Tesla V100	AMD MI25	Tesla V100	AMD MI25
EAC1	68.97%	31.03%	99.83%	100.00%	0.28	0.64
	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060
CAC1	64.51%	35.49%	99.77%	99.85%	4.81	4.39

Table 8 BOP: work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Avg. proc. time per WP (s)	
	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi
PAC1	94.64%	5.36%	91.32%	99.96%	0.46	9.77
PAC2	100.00%	–	99.70%	–	0.46	–
PAC3	–	100.00%	–	99.98%	–	9.75
	Tesla V100	AMD MI25	Tesla V100	AMD MI25	Tesla V100	AMD MI25
EAC1	67.86%	32.14%	99.91%	99.99%	0.03	0.08
	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060
CAC1	81.47%	18.53%	97.97%	97.55%	0.95	2.14

Table 9 BSM: work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Avg. proc. time per WP (sec)	
	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi
PAC1	50.67%	49.33%	17.19%	11.81%	4.E−04	2.E−03
PAC2	100.00%	–	28.07%	–	4.E−04	–
PAC3	–	100.00%	–	32.57%	–	2.E−03
	Tesla V100	AMD MI25	Tesla V100	AMD MI25	Tesla V100	AMD MI25
EAC1	67.75%	32.25%	48.57%	32.89%	7.E−05	8.E−04
	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060
CAC1	56.14%	43.86%	1.69%	2.43%	1.E−03	6.E−03

Table 10 GSP: work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Proc. time (range) per WP (sec)	
	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi	Tesla K20m	Xeon Phi
PAC1	53.06%	46.94%	98.66%	99.2%	1.0E−05 to 27.3	2.0E−05 to 8.57
PAC2	100.00%	–	99.12%	–	1.0E−05 to 31.1	–
PAC3	–	100.00%	–	98.94%	–	2.E−03 to 10.1
	Tesla V100	AMD MI25	Tesla V100	AMD MI25	Tesla V100	AMD MI25
EAC1	81.05%	18.95%	99.3%	99.15%	4.0E−06 to 1.32	2.0E−03 to 11.13
	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060
CAC1	64.71%	35.29%	94.52%	93.56%	3.0E−03 to 48.32	1.0E−02 to 112.34

discuss the efficiency of our prefetcher module in minimizing packet switching time during the computing process. The impact of synchronizations in the execution process is also investigated in this section. For this experiment, we use asymmetric configurations of our three clusters as detailed in Table 3.

Figure 10 shows the workload balance between cluster nodes for different applications such as GEMM, DCS, COR, BOP, BSM, and GSP. Execution time (vertical axis) is set in the logarithmic scale, triangles in the horizontal axis act as placeholders for the nodes of a specific configuration. In Fig. 10 for PAC1 cluster configuration, execution times are shown for each of the three nodes, while execution times are shown only for two nodes for configurations PAC2 and PAC3. In the case of PAC2, no GPUs are installed on node 8, so execution takes place only in nodes 6 and 7. In a similar fashion in PAC3, no Xeon Phi is installed in node 7, and therefore execution takes place only in nodes 6 and 8. For EAC1, we show execution times for nodes 3 and 4 (nodes consisting of heterogeneous devices), and for CAC1 we show execution times for nodes 1, 2 and 3.

Additionally, in Tables 5, 6, 7, 8, 9, and 10 we show the experimental data for the same applications. The first column-group shows the work share between devices of different types for each configuration. The number of each type of devices in a configuration is specified in Table 3. The second column-group shows average WP-busy time per device, which means the time devices spent on processing work packages. The remainder, in this case, is device idle time. The third column-group shows average execution time per work package for a specific device type.

We observe from Fig. 10 that workload balance is achieved despite the heterogeneity and asymmetry of the system. In Fig. 10a, we see the workload balance for GEMM in different cluster configurations. For PAC1, we see that the relative standard deviation from the mean (*%rsd*) of processing times is less than 3%, i.e., all nodes finish processing within ± 1.5 s, while the overall execution time of the application is 32.89 s. For PAC2, PAC3, EAC1, and CAC1 configurations *%rsd* is even lower at 0.58%, 0.98%, 1.33%, and 2.04% respectively. In experiments with other applications (see Fig. 10b–f), we observe that relative standard deviation from the mean in processing times between nodes is in the range of $0.07\% < \%rsd < 1.5\%$, which as in the case of GEMM show a good balance of workload among computing nodes. We also see few exceptions where $1.5\% < \%rsd < 5\%$ which are usually associated with PAC1 configuration such as for DCS, COR, BOP and GSP. According to our experimental data, discrepancies in runtime between nodes in these cases have all come as a result of work-stealing scheduling decisions, similar to the problem we have described in the previous section for GEMM and COR for problem size PS3. This problem can be fixed by using our best performance algorithm for work distribution in [32].

Another interesting observation is the discrepancies in processing times for some types of application in different types of devices. For PAC2 configuration, we use only Tesla K20m GPUs, and for PAC3 we use only Xeon Phi. A more detailed examination of Fig. 10d, f shows that some types of applications such as BOP fit much better for a PAC2 configuration, respectively for a GPU architecture, while we observe the opposite for GSP in PAC3 configuration, which seems to fit better with a Xeon Phi architecture. This is supported by experimental data in Tables 8 and 10 where the processing times per WP are notably different between K20m and Xeon Phi. In a combined approach with GPUs and Xeon Phi such as the PAC1 configuration, these substantial differences in execution time do not degrade overall application performance since a bigger amount of work packages shifts toward faster devices ensuring workload balance and faster completion of execution. We observe this in case of BOP where the GPUs have a bigger share of work packages (see Table 8), while in the case of GSP the share of work packages is more balanced, tilting more toward Xeon Phi (see Table 10).

Next, we investigate the efficiency of the prefetcher module. As shown in Tables 5, 6, 7, 8, 9 and 10 the second column-group—average WP-busy time per device—shows the device utilization during the computing process, i.e., the ratio of the overall execution time during which the devices are not idle. We see that for all of the applications, except for BSM, the utilization of the devices is more than 98% (in most cases more than 99.5%), which indicates that the device idle time, or packet switching time by the prefetcher is less than 2% of the overall computing time for an application. This shows the high efficiency of clusterCL in handling heterogeneous asymmetric clusters by overlapping data transfer with execution and amortization of the impact of data transfers in computing.

On the other hand, for BSM (see Table 9) device idle time is relatively high. BSM is a data-intensive application with very low arithmetic intensity and data transfer times are orders of magnitudes higher than kernel execution times. When a massive amount of data has to be transferred to the nodes and computing intensity of the application is low (e.g., data-intensive applications), the prefetcher is incapable of overlapping data transfers with execution, i.e., hide packet switching time. In this case, the data transfer time for one WP from front-end to the node was 60 times more than the execution of that WP in K20m, respectively 12 times more than the execution in Xeon Phi. However, when the problem size of the application is bigger than the capacity of one device or the collective capacities of all devices in one node, workload partitioning between the nodes is unavoidable for running the application and lower device utilization numbers are the consequence in case of data-intensive applications.

Finally, we examine the impact of synchronizations for applications with dependencies. GSP is a compute-intensive imbalanced application with multiple kernels executed iteratively, where the workload is determined by the iteration loop and spatial coordinates of array elements. In this case in Table 10, we have shown the range of processing times per work package for different devices instead of average processing times since different work packages have different workloads due to application imbalance. The first two kernels of the application have the same data access map on partitioned arrays, while the third kernel has a different access map leading to merging and repartitioning and requiring a synchronization barrier before the beginning of execution of the third kernel. The overall execution times for GSP in PAC1, PAC2 and PAC3 configurations are 130.36s, 811.36s, and 143.97s respectively. The time required for merging and repartitioning by the front-end is 0.45 s ± 0.05 s, including data transfer time which cannot be hidden by the prefetcher. The average accumulated time for these transfers is 0.00144 s ± 0.00002 depending on cluster configuration. Therefore, the share of synchronization in the overall execution time of the application is 0.35%, 0.05%, and 0.31% respectively for PAC1, PAC2, and PAC3 configurations. The impact of synchronization in overall execution time in this case is smaller, mainly due to prefetching and high arithmetic intensity of the application. Experimental data for WP-busy time shown in Table 10 are slightly bigger than these numbers since they also account for packet switching time in addition to synchronization.

5.5 Partitioning tuning

To test our approach for work package tuning as we have presented it in Sect. 3.3, we use different problem sizes (PS1–PS6) for selected HPC applications COR, GEMM, GSP, and CONV. Whereas in the experiments above we have used standard work package sizes (non-tuned), in this experiment we use the tuner to test a set of different work package sizes and show the execution times for work package sizes WPS1–WPS4, where WPS1 is the smallest (usually 1/128–1/448 of the tested problem size for PS1 to PS6) and WPS4 the biggest work package size (usually 1/16–1/56 of the tested problem size for PS1 to PS6). We use the PSC6 symmetric cluster configuration for this experiment.

Figure 11 shows the execution times for different WP sizes. COR and GSP, as shown in the figure, have the most remarkable performance improvements with speedups as much as 3x for the smallest tested WP size compared to the biggest WP size for PS6. This huge speedup can be attributed mainly to specific application characteristics. Both applications are imbalanced such that different threads have a different amount of work to process. Reducing the work package

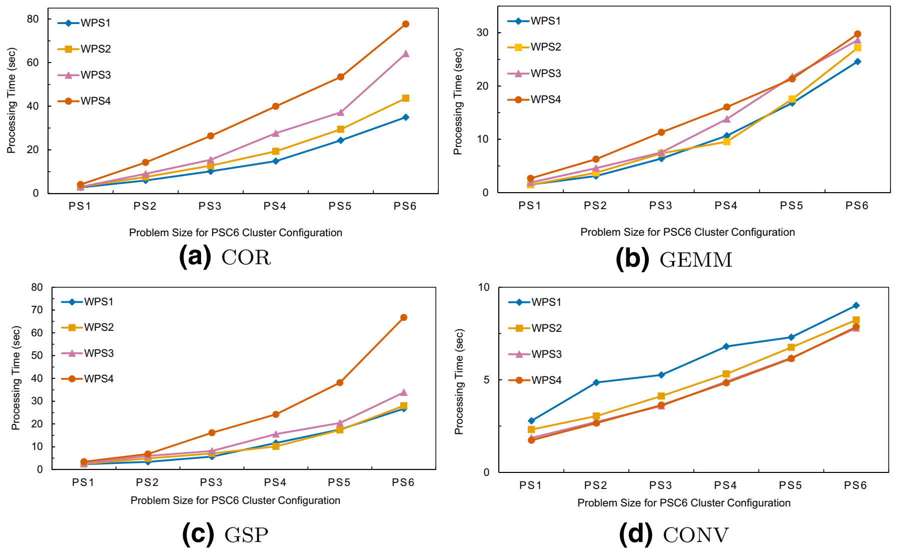


Fig. 11 Effects of the WP Tuner in the overall execution time

size balances better the workload among the work packages, and the work-items have a more uniform distribution of computation load.

In the case of GEMM where the workload between the work-items is balanced, the differences in execution time are much smaller, however very important for large problem size applications as shown in Fig. 11b—switching from the biggest tested WP size to the smallest results in a 21% performance improvement.

Finally, we can observe that for data-intensive applications such as CONV (see Fig. 11d), a bigger WP size is much more beneficial, since it increases the kernel execution times, which effectively enables the prefetcher module to better balance between execution time and data transfer time, respectively decreases the packet switching time. As shown in the figure, choosing the biggest tested WP size over the smallest reduces the overall execution time by almost 15%.

Based on the results from Fig. 11, we can conclude that for all types of applications, compute-intensive, data-intensive and mostly for imbalanced application, use of the WP Tuner is beneficial and improves the overall performance in all cases.

6 Related work

Many studies have shown the benefits of providing programming support for heterogeneous cluster architectures in terms of performance gain and increased programmer productivity [1, 3, 10, 14, 18, 20].

rCUDA by Duato et al. [11] is among the first frameworks that have been developed to support multi-device programming for nVidia GPUs. Similar work based on OpenCL and extending the diversity of device platforms have been proposed by Barak et al. [5]. Alves et al. [1] propose a platform clOpenCL that enables deployment of simple OpenCL-based applications to different devices in a cluster environment. dOpenCL [18] provides a central device manager which manages the assignment of kernels to devices. Similarly, Diop et al. [12] developed DistCL to support distributed execution of OpenCL kernels in a GPU cluster. Aoki et al. [3] have implemented a framework that consists of a runtime that provides an abstraction layer for OpenCL implementations and a bridge program to connect those implementations. SkelCL [36] is a library providing algorithmic skeleton that aims at alleviating programming GPU systems. All of those approaches have proved that enabling distributed execution of OpenCL kernels is important and valuable. They are focused on providing an abstraction layer to the programmer, without dealing in detail with intricacies of the data-parallel applications. Our approach is an essential improvement to these efforts since clusterCL includes advanced features beyond simple OpenCL kernel deployments to the devices in cluster nodes.

Other works have tackled the problem of extending OpenCL platforms to clusters using language extension to support communication via MPI [14, 20] and recently [33]. In [20], Kim et al. provide a framework implementation, SnuCL, which enables programmers to view all cluster devices as if they are in the host node. However, the programmers should be aware of simple MPI calls in order to transfer data between nodes. In a similar fashion, Grasso et al. [14] provide a library, libWater, which handles transparently communications and data transfer between compute nodes, besides enabling distributed kernel execution to the devices. Rasch et al. [33] developed a high-level C++ library which simplifies the development of host code and provides optimizations for data transfer and memory management. Although these works are very similar to our approach of providing an abstraction layer for the programmer, they differ fundamentally from our framework, having that clusterCL require only parameter specification for OpenCL kernels, while the process of partitioning, distribution, and scheduling of kernels and work packages is handled automatically by the framework ensuring workload balance between cluster nodes.

Several works have proposed different methods for custom-size partitioning of workload in heterogeneous systems [16, 17, 22, 24, 34]. In [24], Luk et al. propose a method of adaptive mapping of work to the devices to find the best work distribution. Grewe et al. [16] propose a machine learning approach to determine workload distribution in the presence of GPU contention, while Kofler et al. [17] use an offline training approach based on static and dynamic features of programs to decide on work distribution. Lee et al. [22] propose SKMD framework which enables work partitioning for single-node setting between CPUs and GPUs based on a heuristic method, taking into account data transfer costs and execution time. Shen et al. [34] present an approach based on modeling, profiling and prediction techniques to determine optimal workload partitioning. All of these studies are based on parallel applications which are either single-kernel applications or multi-kernel applications that have no inter-kernel dependencies which would result in communication between devices or synchronization with the host.

Besides workload partitioning for single-kernel applications, solutions for multi-kernel applications have been proposed in [4, 30, 38] and [21]. These studies are mainly focused on dynamic task partitioning and scheduling. StarPU [4] is a runtime system providing task-based scheduling for heterogeneous systems. Similarly, OmpSs [30] is a framework that requires annotating serial applications with directives in order to provide asynchronous execution of tasks in heterogeneous architectures. In [38], Wen et al. propose a task scheduling scheme that can schedule multiple kernels in different devices. In a similar fashion, Lee et al. [21] propose a system for mapping different kernels to different devices while data are migrated based on the order of execution. These works differ from ours since they are based in task-aware (kernel-level) partitioning and scheduling of applications to different devices, while our approach is based in data-aware work distribution where the kernels of data-parallel applications are executed in serial fashion over work packages and inter-kernel dependencies are maintained.

In [35] Shen et al. propose a multi-kernel partitioning approach for heterogeneous single-node systems. Their work is closely related to ours in terms of the partitioning of work for multi-kernel applications and dealing with dependencies between kernels. However, our approach is different in many aspects. Whereas in [35] they use a profiling and prediction technique to determine the best work partitioning for specific devices, we use the work package approach of fine-grain same-size data chunks and improve the overall performance from distribution of work packages to the devices. This, in turn, enables us to perform work package rerouting in case of device failures while with custom-size partitioning this is much harder to achieve and sometimes impossible. Further, our approach extends to clusters and our framework extends the optimization model to minimize the data transfer costs incurred from inter-node communications.

The main difference of our approach and the works cited above is that clusterCL is a framework with many advanced features providing comprehensive support for workload distribution for a broad range of complex data-parallel applications for heterogeneous asymmetric clusters.

7 Conclusion

In this paper, we presented a workload partitioning approach for data-parallel applications and a framework which provides programming support for handling multi-kernel HPC applications with dependencies in heterogeneous asymmetric clusters. We show the advantages of our medium-grained same-size work package approach which can be used to maintain workload balance across cluster nodes and support optimizations for a broad range of work distribution strategies which besides performance takes also energy efficiency into account. Moreover, we discussed dependencies for multi-kernel HPC applications which result from the partitioning process and we have implemented a synchronization mechanism that ensures data coherency along the execution process while minimizing its impact on the runtime using a prefetcher. An experimental section with HPC applications from different benchmark suites shows that our framework supports a broad range of data-parallel applications and achieves workload balance in heterogeneous asymmetric clusters.

Acknowledgements This work was supported in part by the Austrian Science Fund (FWF) Project P 29783 Dynamic Runtime System for Future Parallel Architectures.

References

1. Alves A, Rufino J, Pina A, Santos L (2013) clOpenCL—supporting distributed heterogeneous computing in HPC clusters. In: Euro-Par 2012: Parallel Processing Workshops, LNCS, vol 7640. Springer, Berlin, pp 112–122
2. AMD: AMD accelerated parallel processing SDK. https://developer.amd.com/tools-and-sdks/. Accessed May 2019
3. Aoki R, Oikawa S, Nakamura T, Miki S (2011) Hybrid OpenCL: enhancing OpenCL for distributed processing. In: International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp 149–154
4. Augonnet Cedric, Thibault Samuel, Namyst Raymond, Wacrenier Pierre-Andre (2011) Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr Comput Pract Exp 23(2):187–198
5. Barak A, Ben-Nun T, Levy E, Shiloh A (2010) A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In: 2010 IEEE Conference on Cluster Computing Workshops and Posters, pp 1–7
6. Beaumont O, Becker BA, DeFlumere A, Eyraud-Dubois L, Lambert T, Lastovetsky A (2019) Recent advances in matrix partitioning for parallel computing on heterogeneous platforms. IEEE Trans Parallel Distrib Syst 30(1):218–229
7. Beaumont O, Boudet V, Rastello F, Robert Y (2002) Partitioning a square into rectangles: Np-completeness and approximation algorithms. Algorithmica 34(3):217–239

8. Coti C, Herault T, Lemarinier P, Pilard L, Rezmerita A, Rodriguezb E, Cappello F (2006) Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In: SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p 18
9. Daly JT (2006) A higher order estimate of the optimum checkpoint interval for restart dumps. Fut Gener Comput Syst 22(3):303–312
10. Diop T, Gurfinkel S, Anderson J, Jerger NE (2013) Distcl: a framework for the distributed execution of opencl kernels. In: 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, pp 556–566
11. Duato J, Peña AJ, Silla F, Mayo R, Quintana-Ortí ES (2010) rCUDA: reducing the number of GPU-based accelerators in high performance clusters. In: HPCS, pp 224–231
12. Eskikaya B, Altilar DT (2012) Distributed OpenCL distributing OpenCL platform on network scale. Int J Comput Appl ACCTHPCA(2):25–30
13. Gentzsch V (2001) Sun grid engine: towards creating a compute power grid. In: Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid, pp 35–36
14. Grasso I, Pellegrini S, Cosenza B, Fahringer T (2014) A uniform approach for programming distributed heterogeneous computing systems. J Parallel Distrib Comput 74(12):3228–3239 Domain-specific languages and high-level frameworks for high-performance computing
15. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to GPU codes. In: 2012 Innovative Parallel Computing (InPar), pp 1–10
16. Grewe D, O'Boyle MFP (2011) A static task partitioning approach for heterogeneous systems using opencl. In: Knoop J (ed) Compiler construction. Springer, Berlin, pp 286–305
17. Gschwandtner P, Durillo JJ, Fahringer T (2014) Multi-objective auto-tuning with insieme: optimization and trade-off analysis for time, energy and resource usage. In: Euro-Par 2014 Parallel Processing, LNCS, vol 8632. Springer, Berlin, pp 87–98
18. Kegel P, Steuwer M, Gorlatch S (2012) dOpenCL: towards a uniform programming approach for distributed heterogeneous multi-/many-core systems. In: Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), pp 174–186
19. Khronos OpenCL Working Group: The OpenCL C Specification. Version 2.0. http://www.khronos.org/opencl. Accessed May 2019
20. Kim J, Seo S, Lee J, Nah J, Jo G, Lee J (2012) SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing (ICS). San Servolo Island, Venice, Italy, pp 341–352
21. Lee J, Samadi M, Mahlke S (2015) Orchestrating multiple data-parallel kernels on multiple devices. In: 2015 International Conference on Parallel Architecture and Compilation (PACT), pp 355–366
22. Lee J, Samadi M, Park Y, Mahlke S (2013) Transparent CPU–GPU collaboration for data-parallel kernels on heterogeneous systems. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT'13. IEEE Press, Piscataway, NJ, USA, pp 245–256
23. Louis-Noel Pouchet: PolyBench/GPU. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ (2012). Accessed May 2019
24. Luk CK, Hong S, Kim H (2009) Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp 45–55
25. MPI Forum: MPI: a message-passing interface standard, Version 3.0. http://www.mpi-forum.org. Accessed May 2019
26. nVidia: CUDA zone. https://developer.nvidia.com/cuda-zone. Accessed May 2019
27. nVidia: OpenCL SDK. https://developer.nvidia.com/opencl. Accessed May 2019
28. OpenACC: OpenACC Standard. https://www.openacc.org/. Accessed May 2019
29. OpenMP: OpenMP specification. https://www.openmp.org/. Accessed May 2019
30. Planas J, Badia RM, Ayguadé E, Labarta J (2013) Self-adaptive OMPSS tasks in heterogeneous environments. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp 138–149
31. Raca V, Mehofer E (2015) Device-sensitive framework for handling heterogeneous asymmetric clusters efficiently. In: 26th IEEE International Symposium on Computer Architecture and High Performance Computing. Florianopolis, Brazil, pp 181–188
32. Raca V, Mehofer E, Hudec M (2016) Optimal time and energy efficient work distributions in heterogeneous systems. In: 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). Heraklion, Greece, pp 440–447

33. Rasch A, Bigge J, Wrodarczyk M, Schulze R, Gorlatch S (2019) dOCAL: high-level distributed programming with OpenCL and CUDA. J Supercomput
34. Shen J, Varbanescu AL, Lu Y, Zou P, Sips H (2016) Workload partitioning for accelerating applications on heterogeneous platforms. IEEE Trans Parallel Distrib Syst 27(9):2766–2780
35. Shen J, Varbanescu AL, Martorell X, Sips H (2015) Matchmaking applications and partitioning strategies for efficient execution on heterogeneous platforms. In: 2015 44th International Conference on Parallel Processing, pp 560–569
36. Steuwer M, Kegel P, Gorlatch S (2011) Skelcl—a portable skeleton library for high-level gpu programming. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp 1176–1182
37. Top500: Top 500 list. https://www.top500.org/lists/2019/06. Accessed July 2019
38. Wen Y, Wang Z, O'Boyle MFP (2014) Smart multi-task scheduling for openCL programs on CPU/GPU heterogeneous platforms. In: 2014 21st International Conference on High Performance Computing (HiPC), pp 1–10
39. Yoo AB, Jette MA, Grondona M (2003) Slurm: simple linux utility for resource management. In: Feitelson D, Rudolph L, Schwiegelshohn U (eds) Job scheduling strategies for parallel processing. Springer, Berlin, pp 44–60
40. Young JW (1974) A first order approximation to the optimum checkpoint interval. Commun ACM 17(9):530–531