International Journal of Parallel Programming (2023) 51:172–185
https://doi.org/10.1007/s10766-022-00742-5

Distributed Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments

Nina Herrmann[1] · **Herbert Kuchen**[1]

Received: 25 September 2022 / Accepted: 21 November 2022 / Published online: 7 January 2023

Abstract
Contemporary HPC hardware typically provides several levels of parallelism, e.g. multiple nodes, each having multiple cores (possibly with vectorization) and accelerators. Efficiently programming such systems usually requires skills in combining several low-level frameworks such as MPI, OpenMP, and CUDA. This overburdens programmers without substantial parallel programming skills. One way to overcome this problem and to abstract from details of parallel programming is to use algorithmic skeletons. In the present paper, we evaluate the multi-node, multi-CPU and multi-GPU implementation of the most essential skeletons Map, Reduce, and Zip. Our main contribution is a discussion of the efficiency of using multiple parallelization levels and the consideration of which fine-tune settings should be offered to the user.

Keywords Parallel programming · Skeleton programming · Heterogeneous computing environments · High-level frameworks · Usability

1 Introduction

The field of High Performance Computing (HPC) is growing. Typical HPC hardware offers multiple computing nodes, central processing units (CPUs) and graphics processing units (GPUs) to speed up computations. Programmers have to deal with multiple low-level frameworks to exploit those levels of hardware where expertise for the single frameworks and the combination of those frameworks is required. Examples for those frameworks are Message Passing Interface (MPI) [1], OpenMP [2], and CUDA [3].

✉ Nina Herrmann
nina.herrmann@uni-muenster.de

Herbert Kuchen
kuchen@uni-muenster.de

[1] University of Münster, Leonardo-Campus 3, 48149 Münster, Germany

Constructing a program with those frameworks is time-consuming and error prone. Even if a functioning program was constructed, the lack of knowledge leads to poor design decisions such as not using specific memory spaces, a bad distribution of workload to computational units, or choosing an inappropriate number of threads. Consequently, most programmers have no other option than to rely on high-level parallel programming approaches or to require excessive computation time. Most high-level approaches have multiple benefits such as portable code for different hardware architectures and requiring less maintenance for the end-user as the framework is updated.

Cole introduced algorithmic skeletons as one of the major high-level approaches to abstract from low-level details [4]. Algorithmic skeletons enclose reoccurring parallel and distributed computing patterns, such as Map and Reduce. This concept is wide-spread and beside others implemented as libraries [5–7], domain-specific language (DSLs) [8], and general frameworks [9, 10]. These approaches rarely support the combination of all levels of parallel hardware simultaneously, namely multiple nodes with multiple CPUs (possibly with vectorization) and accelerators.

In the present paper, we will first discuss related high-level frameworks contributing to the parallelization on multiple hard-ware levels in Sect. 2. Section 3 describes the design of our chosen high-level approach, the Muenster Skeleton Library (Muesli), while Sect. 4 presents the implementation of some added features, in particular distributed cubes. The run-times of exemplary programs are shown and discussed in Sect. 5. Finally in Sect. 6, we conclude and point out future work.

2 Related Work

Closest to our work is the skeleton framework SkePU3. In combination with StarPU, it works on heterogeneous clusters. However, it does not (yet) support the combined use of all possible levels of parallelism. Either the program is executed on one node with multiple cores and GPUs [11] or the programs are executed on multiple nodes with single backends (either GPU (OpenCL) or CPU (OpenMP)) [9]. SkePU supports one, two, three, and four-dimensional data structures. Other skeleton frameworks which are in continuous development also do not consider all three layers of parallelization (e.g. FastFlow [10], SkelCL [12], Musket [13]). Hybrid execution of programs on CPUs and accelerators has been the topic in multiple approaches such as SkePU [11], Marrow [14] and Qilin [15]. SkePU and Marrow distribute the load statically between the CPU threads and the GPUs, while Qilin dynamically distributes the working packages. Findings regarding the optimal partitioning of work and data are often hardware and problem-dependent and rarely comparable. Noteworthy, skeletal programming is also used for commercial products such as Intel TBB for multicore CPU parallelism.

In the present paper, we will discuss the distribution of data and computations on multiple nodes, CPUs and GPUs. Our goal is to provide a starting point to automatically distribute workload between different computational units, relieving the programmer from estimating suitable partition ratios. To the best of our knowledge, other approaches either distribute the workload dynamically, which produces

communication overhead, or leave the choice to the programmer, who might not have the expertise to decide on a reasonable split.

3 The *Mue*nster *S*keleton *Li*brary *Muesli*

Originally, skeletal parallel programming was mainly implemented in functional languages since it derives from functional programming [4]. Today, the majority of skeleton frameworks are based on C/C++ [e.g. 7, 9, 10, 16], since the language is known for good performance and interoperability with low-level parallel frameworks such as CUDA, OpenMP, and MPI. Although Python has become popular in many natural science applications, as packages can be easily written and integrated, this does not apply to calculation intense applications as the language entails a major slowdown. Therefore, especially in the HPC context, C/C++ is still the first choice.

The C++ library used for this work is called Muesli [6]. Muesli provides task- and data-parallel skeletons such as Fold, multiple versions of Map, Gather, and multiple versions of Zip. These operations can be used to write programs for clusters of multiple nodes with multicore processors and GPUs. Internally, it is based on MPI, OpenMP, and CUDA. Muesli relieves the programmer from tasks which require expertise in parallel programming, such as the number of threads started and copying data to the correct memory spaces and helps to avoid common errors in parallel programming such as race conditions when accessing data structures. Although the additional abstraction causes some overhead, it does not increase the execution time significantly. In contrast to previous versions, Muesli now supports not only distributed arrays (DA) and matrices (DM) but also distributed cubes (DC) as data structures. Especially in the scientific context, e.g. in computational fluid dynamics, cubes are essential to model 3D objects. A distinctive feature of Muesli is that for Map and Zip, there are in-place variants and variants where the index is used for calculations. Additionally, the MapStencil skeleton allows to update each matrix element depending on its neighbors.

```
1  class Sum : public Functor2<int, int, int>{
2    public: MSL_USERFUNC int operator() (int x, int y)
3                                  const {return x+y;}};
4  Sum sum;
5  auto product = [] (int i, int j) {return i*j;};
6  DA<int> a(3,2);                        // delivers: {2,2,2}
7  DA<int> b = a.mapIndex(sum);           // delivers: {2,3,4}
8  a.zipInPlace(b,product);               // delivers: {4,6,8}
9  int scalarproduct = a.fold(sum);       // delivers: 18
```

Listing 1: Scalar product in Muesli

Listing 1 shows a simple program for calculating the scalar product of the distributed arrays `a` and `b` (in a slightly simplified syntax). Firstly, a distributed array is created in line 6 of Listing ??. A skeleton typically gets a user function as argument, which can be either a C++ function or a C++ functor. We enable currying, i.e. the

arguments of a user function can be supplied one by one rather than all together. For instance, the `MapIndex` skeleton in line 7 of Listing 1 automatically adds the considered array element and its index as additional parameters to the `sum` functor.

4 Data Distribution and Data structures in Heterogeneous Computing Environments

Previous work on Muesli discussing stencil computations already provided the foundation for distributing matrices between computational nodes. This approach is now also used for Map, Zip, Fold and variants of them. This section introduces the data distribution mechanism and the metrics which are used to determine the workload allocated to the computational nodes.

4.1 Distributed Cubes

The added data structure cube is similarly designed to previous data structures in Muesli which makes the syntax easy for programmers. For constructing a distributed cube, at least three arguments have to be passed to define the cube's dimensions. Optionally, a default value can be passed to be filled into all elements of the cube. Listing 2 creates two distributed cubes `a` and `b`, filled with the default values 0 and 1, respectively. The `mapIndexInPlace` skeleton in line 11 adds to each element its row-index, column-index, and its index of the third dimension. In line 12, each value of `b` is added to the corresponding value of `a`.

```
1  class Sum : public Functor2<int, int, int>{
2    public: MSL_USERFUNC int operator() (int x, int y)
3                  const {return x+y;}};
4  class Sum4 : public Functor4<int, int, int, int, int>{
5    public: MSL_USERFUNC int operator() (int i, int j, int x, int y)
6                  const {return i+j+x+y;}};
7  Sum sum;
8  Sum4 sum4;
9  DC<int> a(3,3,3,0);
10 DC<int> b(3,3,3,1);
11 a.mapIndexInPlace(sum4);
12 a.zipInPlace(b,sum);
```

Listing 2: Exemplary cube computation in Muesli

4.2 Segmentation of Data Structures

A simplified version of the approach chosen for the mapStencil skeleton applied to a matrix can be seen in Fig. 1. Each node is responsible for multiple rows of the data structure, and within each node, the data structure is again split between the available CPUs and GPUs in a row-wise manner.

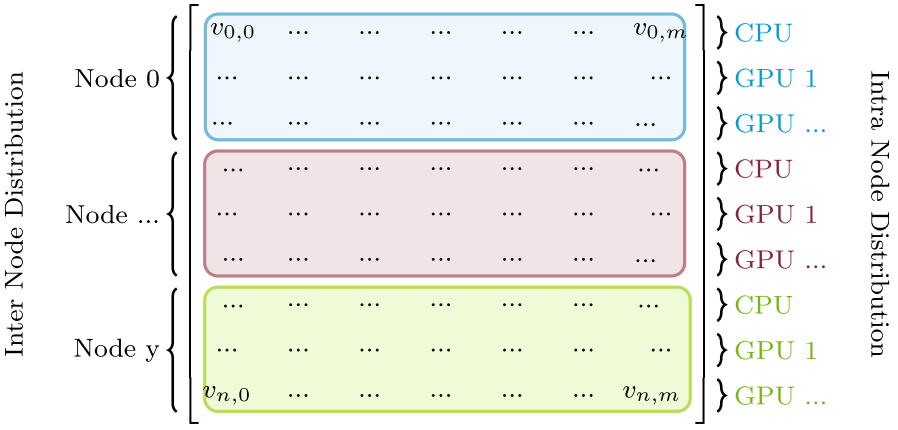


Fig. 1 Data distribution

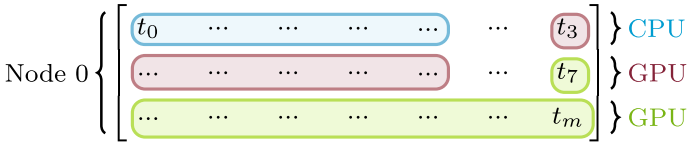
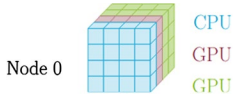


Fig. 2 Intra-node distribution using multiple accelerators

Fig. 3 Intra-node distribution of a Cube



In the context of stencil calculations, it was reasonable to distribute complete rows or rectangles of data to minimize the required data transfers for communicating border values. Therefore, *always* complete rows were distributed. For skeletons such as Map and Zip, where the calculation does not depend on neighbor values, it is not necessary to distribute complete rows, as incomplete rows do not degrade the execution time. Data transfers are rarely needed. Therefore, it is assumed that distributing complete rows is less important than equally splitting the workload. Figures 2 and 3 demonstrate how incomplete rows are distributed and how the concept is transferred to a cube. This presentation also portions the amount of work (elements calculated) unequally for the two GPUs. Prospectively, Muesli automatically calculates suitable workload splits to relieve the programmer from the fine-tuning the splitting work.

4.3 Work-Load Partitioning

The current implementation uses the number of cores of the used GPU to allocate more elements to GPUs, which can start more threads concurrently. This relieves the user from some low-level details for exploiting the available hardware. More

Table 1 Overview of used Hardware

Type	Number Nodes	Per Node			
		GPU-type	GPUs	CPU-type	CPUs
Local	1	Quadro K620	1	Intel(R) Core(TM)	1
		GeForce GTX 750 Ti	1	i7-4790 CPU 8 cores	
Cluster	2	GeForce RTX 2080 Ti	4	Zen3(EPYC 7513)	1
				24 cores	

precisely, CUDA provides `DeviceProperties` which, amongst others, state the number of multiprocessors available. To calculate the number of cores the function `_ConvertSMVer2Cores(props.major, props.minor) * props.multiProcessorCount;` has to be used as the number of multiprocessors is dependent on the version of the GPU. However, a good approximation of the maximum possible parallelism can be calculated with this reference number. In the future, this number might also be dependent on the version of the GPU to prefer newer GPUs. Besides splitting the workload between multiple GPUs the fraction which is calculated by the CPU has to be automatically chosen by the library. The experimental results section evaluates which partition is reasonable for different skeletons, determining good default values for different calculation patterns.

5 Experimental Results

We have tested varying distribution possibilities with the distributed cubes for the skeletons Map, MapInPlace, MapIndex, MapIndexInPlace, Fold, Zip, ZipIndex, ZipInPlace and ZipIndexInPlace. The distributions include multi-node multi-GPU set-ups and different fractions of calculations which are assigned to the CPU. The run-times of all experiments are the result of calling skeletons multiple times. For the experiments, the HPC machine Palma II[1] and, for comparison, an ordinary 8-core PC with two different GPUs were used (Table 1). With Palma, we used the GeForce RTX 2080 Ti GPUs partition equipped with 2 nodes each with 4 GPUs and a Zen3 (EPYC 7513) CPU. Each node has 24 CPU cores. To provide generalizable results, each skeleton was tested on a stand-alone basis. For this purpose, we used multiple sizes and CPU-fractions. For running the sequential version on the HPC, a single Broadwell (E5-2683 v4) CPU was used. We let each skeleton run 25 times (without data transfers between them) to produce meaningful run-times for the calculations. The PC was used to have a comparison for the discussion of a suitable CPU-fraction. GPUs with fewer streaming multiprocessors can start fewer threads concurrently, making the use of the CPU more reasonable. The PC is equipped with one Quadro K620, one GeForce GTX 750 Ti, and an eight core Intel(R) Core(TM)

[1] https://confluence.uni-muenster.de/display/HPC/GPU+Nodes

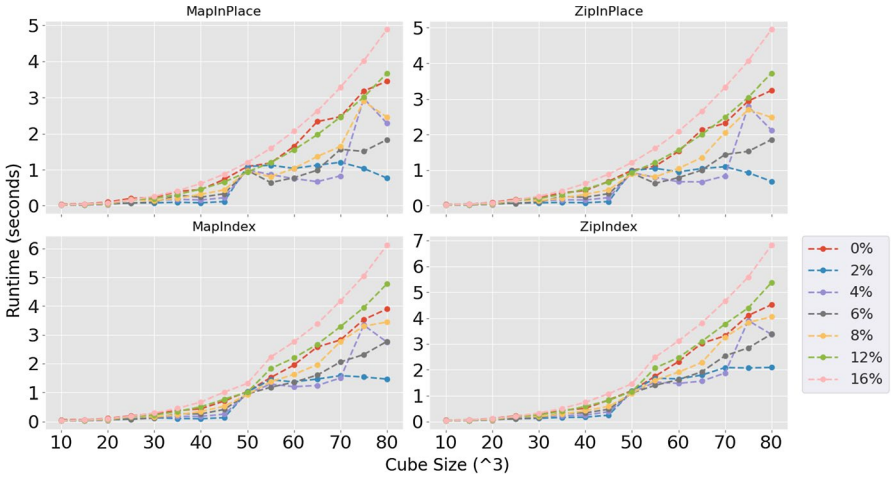


Fig. 4 Run-times (in s) for different CPU-fractions calculated by the CPU for Map- and Zip variants on the PC

i7-4790 CPU with 3.60GHz. The sequential version only used one of the available cores and no GPU.

5.1 CPU Usage on the PC

Allocating a fraction of the work to the CPU did not speed up the Map Stencil skeleton as Map Stencil has communication overhead for transferring the padding between different computational units after each skeleton call. Hence it is reasonable to test CPU-fractions for skeletons which require less communication. Map and Zip are suitable examples as calculations only depend on the current element. Figure 4 displays some results of running Map and Zip on the PC. As can be seen with increasing data size, all skeletons are optimal at a CPU-fraction of 2%. CPU-fractions greater than 16% are not displayed as their run-time is increasing as expected. Interestingly, at a data size of 50^3, all run-times are nearly equal and, from that point on, show clearly a difference. In Table 2 exemplary speedups for the mixed usage of the CPUs and the GPU are listed. Our results aim to automatically identify those changing points for the end-user to adjust the generated code to the system. In this context, it is especially noteworthy that eight cores available on the PC provide some significant computation power. Still, the fraction allocated to the CPU is small. Hence hardware with less cores should, by default, not use the CPU for Map and Zip.

In contrast to Map and Zip, creating a new data structure and the Fold skeleton are less calculation intense. Hence it was expected that CPU variants would be faster. Figure 5 displays both. Creating a data structure does not require a lot of time. It can be seen that for smaller sizes, the sequential and CPU only version are faster as no GPU memory needs to be allocated. However, for growing data sizes, they are similar. All CPU and GPU mixed variants show no significant difference in their

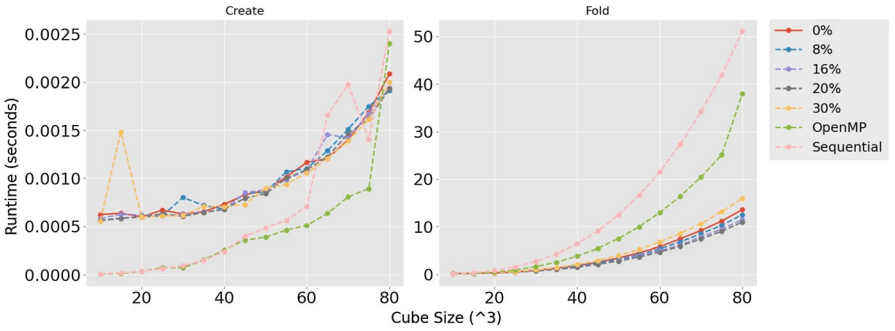


Fig. 5 Run-times (in s) for different CPU-fractions calculated by the CPU

Table 2 Run-times (in s) for the sequential, the OpenMP only version, the GPU only version, and for the optimal mix of CPU and GPU for MapInPlace on the PC. The column CPU % shows which CPU fraction was used in the optimal mix. The following columns show speedups of the optimal mix compared to the sequential version, the OpenMP only version, and the GPU only version

size (^{3})	Run-time				CPU % of Opt. Mix	Speedup		
	Seq.	OpenMP	GPU	Opt. Mix		Seq.	OpenMP	GPU
50	13.09	7.47	1.09	0.94	0.12	13.98	7.98	1.16
60	22.60	12.87	1.64	0.75	0.04	30.12	17.15	2.19
70	35.88	20.38	2.47	0.83	0.04	43.42	24.66	2.99
80	53.65	30.71	3.44	0.76	0.02	70.14	40.15	4.50

run-times. As they are finished in milliseconds, this aspect has little influence on the overall run-time of a larger application. In contrast, the Fold skeleton requires a lot of time (around 12–17 s for parallel programs). In contrast to the previous skeletons, Fold performs best for 20 % CPU-fraction and achieves with this setting a speedup of 1.2 compared to the GPU only version for 80^3 elements.

5.2 CPU Usage on HPC Machine

In contrast to the PC, Palma has relatively strong GPUs. A GeForce RTX 2080 Ti can start up to 69,632 threads in parallel, while the aforementioned GPUs can start 6,144 and 10,240 threads in parallel, respectively. Hence, using the CPU is expected to be less beneficial. In contrast to the PC, skeletons were called up to 10,000 times as otherwise the run-time would have been too short. Figure 6 depicts the run-times for the MapIndex, ZipIndex, MapIndexInPlace, and ZipIndexInPlace skeleton. Two major observations can be made. Firstly, for InPlace variants, no speedup is achieved when using the CPU. This underlines that with extremely powerful GPUs the CPU should not be used for calculations. Secondly, non-InPlace variants show a rather mixed behaviour. No CPU-fraction is clearly winning, but the winner changes almost randomly with the cube size and the differences are small.

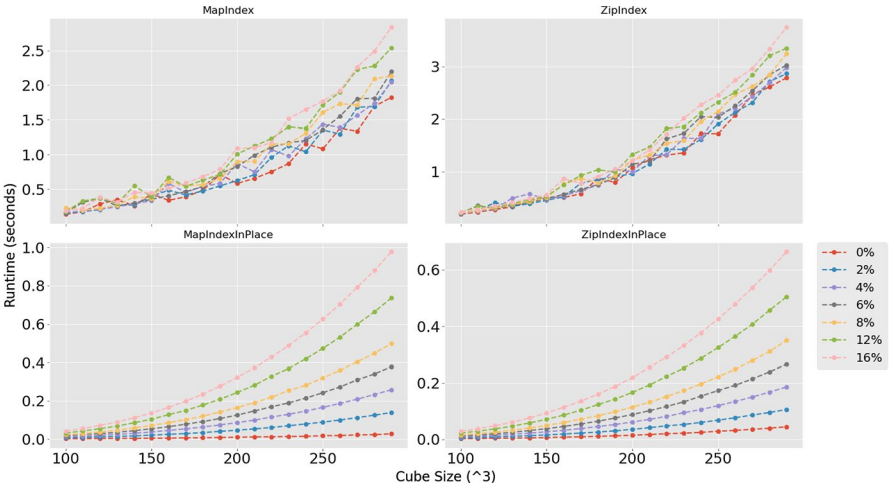


Fig. 6 Run-times (in s) for different CPU-fractions calculated by the CPU for Map- and Zip variants on the HPC Palma

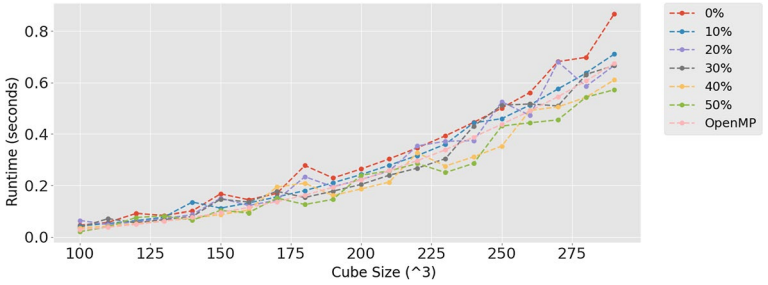


Fig. 7 Run-times (in s) for different CPU-fractions calculated by the CPU for the Fold skeleton on the HPC Palma

In contrast to the previous experiments, the size of the cube was increased to make use of all threads which can be started (max. 290^3=24,389,000 elements). Non-InPlace skeletons require to create a new data structure where the results are stored. As the skeletons beside using one or two data structures use the same user-function, the effect must be produced by creating and writing to a different data structure. The effect of having longer run-times for non-InPlace skeletons could also be observed for the PC which required double the amount of time compared to InPlace variants.

For the Fold skeleton the ideal CPU fraction is hard to determine. Figure 7 shows that all GPU programs perform only as good as the OpenMP program. Conclusively outsourcing calculation to the CPU produces similar run-times. However, allocating 40–50% of the calculation to the CPU is the best fit in most cases.

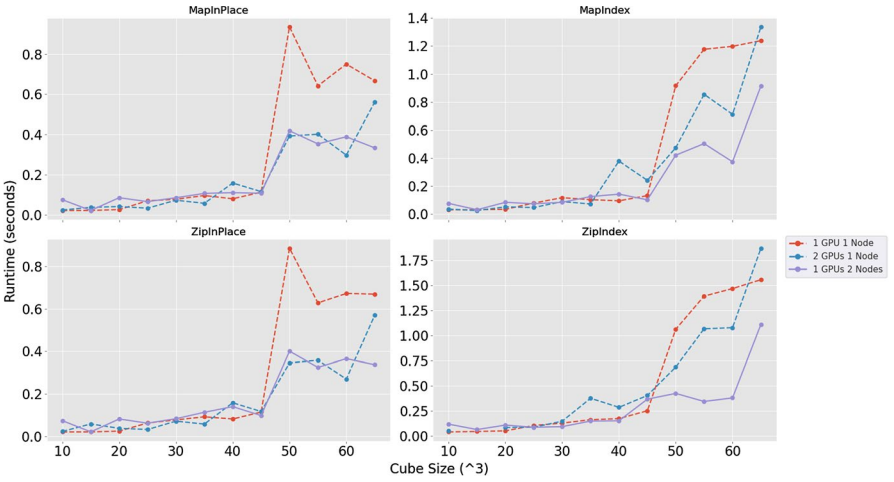


Fig. 8 Run-times (in s) for multiple nodes and GPUs for Map- and Zip variants on the PC

Table 3 Run-times (in s) and speedups on multiple nodes and GPUs for ZipInPlace on the PC

size (ˆ3)			1 Node		2 Nodes		
	Seq.	OpenMP	1 GPU	2 GPUs	2 GPUs	Optimal	Speedup
45	9.24	5.50	0.11	0.12	0.10	0.10	93.94
55	16.88	10.02	0.63	0.36	0.33	0.33	51.89
65	27.96	16.55	0.67	0.67	0.34	0.34	82.75

5.3 Multi-Node and Multi-GPU on the PC

As the PC has just eight cores and only two GPUs, only a moderate speedup is possible by adding more nodes. We have measured the performance of one MPI process with one GPU, one MPI process with two GPUs, and two MPI processes with one GPU each (Figs. 8, 9). For the figure, the optimal CPU fraction has been taken, which varied between 0.02 and 0.04 %. Minor run-time decreases are caused by data sizes closer to a multiple of the maximum number of threads that can run in parallel. In this case, fewer threads are idle. Again at the breaking point of 50^3 elements, it is beneficial to use multiple GPUs or multiple nodes. Using two MPI processes with one GPU each is better than using two GPUs with one process. Again it can be seen that non-InPlace skeletons require more time caused by the additional creation of a data structure. It can be seen that in specific hardware settings, using the CPU (in addition to the GPU) can speed up the program (Table 3). This is especially relevant for Map and Zip skeletons as they do not require communication between CPU and GPU in contrast to the Fold skeleton and the creation of data structures. It can also be seen that for strong GPUs, it is often more efficient to let the GPU do all calculations.

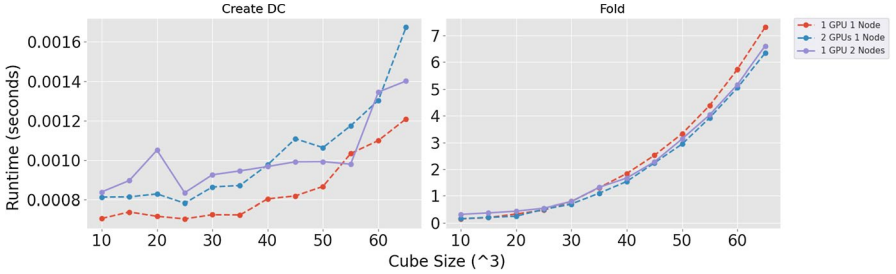


Fig. 9 Run-times (in s) for multiple nodes and GPUs for Map- and Zip variants on the PC

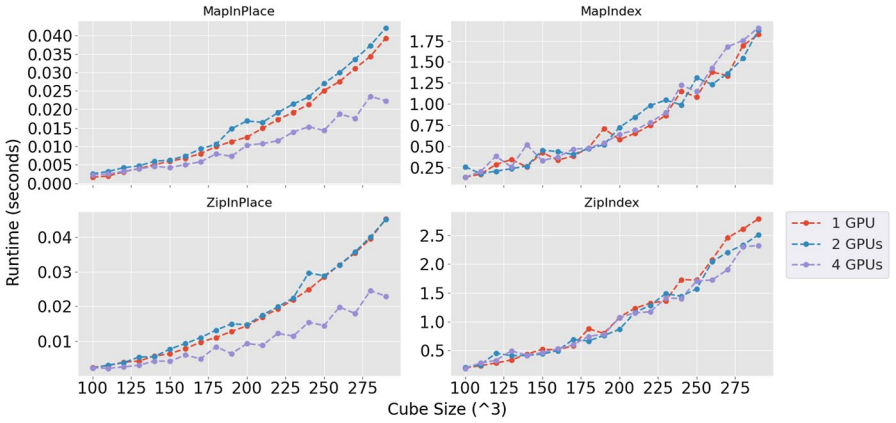


Fig. 10 Run-times (in s) for multiple GPUs for Map- and Zip variants on the HPC Palma

For the Fold skeleton, only a minor speedup could be achieved on the PC. At 25^3 elements, the multi-node and multi-GPU variants become faster than the single GPU variant. However, both variants are only slightly faster than the single GPU program. Creating distributed cubes is fastest on one GPU. Multi-node and multi GPU programs have the disadvantage of requiring multiple calls to allocate memory, which creates some overhead.

5.4 Multi-Node and Multi-GPU on an HPC Machine

On Palma, setups with up to four GPUs per node can be tested. Although initializing additional GPUs produces overhead, the calculations can be distributed and more computational power can be used. Results for different skeletons can be seen in Figs. 10 and 11. For non-InPlace variants, there is nearly no visible speedup between the different GPU versions. The creation of new data structures is dependent on the CPU. Hence using multiple GPUs on one node does not speed up the run-time for skeletons which require the creation of new data structures. In contrast for InPlace skeletons the four GPU variant shows a significant advantage compared to the one

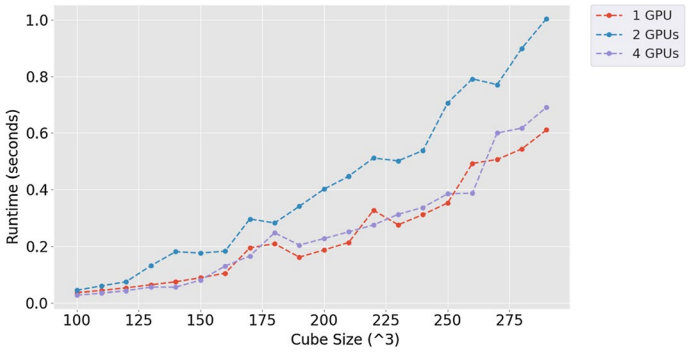


Fig. 11 Run-times (in s) for multiple GPUs for the Fold skeleton on the HPC Palma

GPU variant. However, we cannot understand why the two GPU variant is not faster than the variant using one GPU. Although calling a skeleton produces some overhead, this should not outweigh the calculation time. Therefore, more investigation in this area is required.

Interestingly, for the Fold skeleton the single GPU program and the four GPU program have approximately the same run-time. This finding can be used for a default default implementation of Fold only on one GPU with the best found CPU-fraction.

6 Conclusions and Future Work

We have shown that in specific hardware settings and for selected skeletons, using the CPU in addition to the GPUs can speed up the program. This is especially relevant for Map and Zip skeletons as they do not require communication between CPU and GPU in contrast to the Fold skeleton and the creation of data structures. We have also shown that for strong GPUs, it is often more efficient to let the GPUs do all calculations.

As future work, we plan to use our observations for automatically choosing a suitable data distribution. This could facilitate the usage of Muesli for inexperienced programmers. There is however still some way to go, before we can reach this goal. As could be seen, different hardware requires a different distribution of data and calculations. Therefore, we aim to fine-tune Muesli to the specific hardware. In addition to making use of hardware details which are available at run-time, a precompiler could care about the distribution of data and work, and might integrate regular data-structures. SkePU is using a precompiler, and the authors mentioned the usefulness of a static code analysis by the precompiler to autotune a program [11]. However, this has not been fully implemented yet to the best of our knowledge. The autotuner would have to take the complexity of the user functions into account.

Author Contribution Both Authors have published the work together. Runtime measurements have mainly been done by Nina Herrmann, writing has been done by both.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. MPI Forum. Mpi standard. https://www.mpi-forum.org/docs/ (2021). Accessed: 10.05.2021
2. OpenMP. Openmp the openmp api specification for parallel programming. https://www.openmp.org/ (2021). Accessed: 10.05.2021
3. NVIDIA Corporation. Cuda. https://developer.nvidia.com/cuda-zone (2021). Accessed: 10.05.2021
4. Cole, M.I.: Algorithmic skeletons: structured management of parallel computation. Pitman, London (1989)
5. 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)
6. Ernsting, S., Kuchen, H.: Data parallel algorithmic skeletons with accelerator support. Int. J. Parallel Program. **45**(2), 283–299 (2017)
7. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eskel. In: European Conference on Parallel Processing, pp. 761–770. Springer (2005)
8. Wrede, F., Rieger, C., Kuchen, H.: Generation of high-performance code based on a domain-specific language for algorithmic skeletons. J. Supercomput. **76**(7), 5098–5116 (2020)
9. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: Skepu 3: Portable high-level programming of heterogeneous systems and hpc clusters. Int. J. Parallel Program. **49**(6), 846–866 (2021)
10. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. Programming multi-core and many-core computing systems, parallel and distributed computing (2017)
11. Öhberg, T., Ernstsson, A., Kessler, C.: Hybrid cpu-gpu execution support in the skeleton programming framework skepu. J. Supercomput. **76**(7), 5038–5056 (2020)
12. Steuwer, M., Gorlatch, S.: Skelcl: a high-level extension of opencl for multi-gpu systems. J. Supercomput. **69**(1), 25–33 (2014)
13. Rieger, C., Wrede, F., Kuchen, H.: Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, pp. 1534–1543 (2019)
14. Soldado, F., Alexandre, F., Paulino, H.: Towards the transparent execution of compound opencl computations in multi-cpu/multi-gpu environments. In: European Conference on Parallel Processing, pp. 177–188. Springer (2014)
15. Luk, C. K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 45–55. IEEE (2009)
16. Emoto, K., Fischer, S., Hu, Z.: Generate, test, and aggregate. In: Seidl, H. (ed.) 1Programming Languages and Systems, pp. 254–273. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)