



Stencil Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments

Nina Herrmann¹ · Breno A. de Melo Menezes¹ · Herbert Kuchen¹

Received: 12 October 2021 / Accepted: 18 May 2022 / Published online: 23 July 2022
© The Author(s) 2022

Abstract

The development of parallel applications is a difficult and error-prone task, especially for inexperienced programmers. Stencil operations are exceptionally complex for parallelization as synchronization and communication between the individual processes and threads are necessary. It gets even more difficult to efficiently distribute the computations and efficiently implement communication when heterogeneous computing environments are used. For using multiple nodes, each having multiple cores and accelerators such as GPUs, skills in combining frameworks such as MPI, OpenMP, and CUDA are required. The complexity of parallelizing the stencil operation increases the need for abstracting from the platform-specific details and simplify parallel programming. One way to abstract from details of parallel programming is to use algorithmic skeletons. This work introduces an implementation of the MapStencil skeleton that is able to generate parallel code for distributed memory environments, using multiple nodes with multicore CPUs and GPUs. Examples of practical applications of the MapStencil skeleton are the Jacobi Solver or the Canny Edge Detector. The main contribution of this paper is a discussion of the difficulties when implementing a universal Skeleton for MapStencil for heterogeneous computing environments and an outline of the identified best practices for communication intense skeletons.

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

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

Breno A. de Melo Menezes
breno.menezes@uni-muenster.de

Herbert Kuchen
kuchen@uni-muenster.de

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

1 Introduction

High-performance computers consist frequently of multiple computing nodes, each consisting of multicore (CPUs) and potentially several (GPUs). Such hardware is often used in e.g. natural sciences or data science applications. In order to exploit the resources provided by such heterogeneous computing environments, programmers typically have to deal with a challenging combination of low-level frameworks for the different hardware levels such as message passing interface (MPI) [13], OpenMP [17], and CUDA [7]. In addition, this requires investigating the way data is distributed, choosing the number of threads, and a deep understanding of parallel programming. This raises a high barrier for the average programmer to develop an efficient program. Furthermore, without experience programming becomes a tedious and error-prone task. High-level concepts for parallel programming bring multiple benefits. Besides facilitating programming, most frameworks are able to produce portable code which can be used for different hardware architectures.

COLE introduced algorithmic skeletons as one high-level approach to abstract from low-level details [6]. Algorithmic skeletons encapsulate reoccurring parallel and distributed computing patterns, such as Map. They can be implemented in multiple ways e.g. as libraries [3, 11, 12], Domain-specific languages (DSLs) [20], and general frameworks [2, 16]. Although a variety of implementations exists, the research how to efficiently implement skeletons on heterogeneous computing environments is still ongoing.

The MapStencil skeleton is particularly complex as it accesses multiple data points from the same data structure. Race conditions have to be avoided and the pieces of data which are used by distinct processes and threads have to be efficiently transferred. This is especially challenging on heterogeneous computing environments with multiple nodes and GPUs as sending data from one node to another node requires MPI while sending data from one GPU to another GPU might require multiple CUDA operations (and possibly MPI). Furthermore, the efficiency of distinct memory spaces has to be taken into account. For example, the efficiency of the shared memory of the GPU depends among others on the number of GPUs and threads used, how often data points are read in the map operations, and the size of the stencil. Practical applications of the MapStencil skeleton are found e.g. in image processing, the Jacobi solver, or filters in e.g. processing telescope data.

The main contribution of this paper is a discussion of an efficient implementation of the MapStencil skeleton regarding heterogeneous computing environments. To the best of our knowledge, this is the first implementation of this skeleton enabling a *combined use of all the mentioned levels of parallel hardware*, i.e. multiple nodes consisting of multicore CPUs and multiple GPUs. Our paper is structured as follows. Section 2 summarizes related work, Sect. 3 introduces the Muesli, pointing out its concepts and benefits. In Sect. 4 the conceptual implementation of the MapStencil skeleton is discussed. Benchmark applications, as well as experimental results, are presented in Sect. 5. In Sect. 6, we discuss possible extensions. Finally, we conclude in Sect. 7 and briefly point out future work.

2 Related Work

Cheikh et al. [4] mention in their work the difficulties of implementing stencil operations in heterogeneous parallel platforms containing CPUs and GPUs. They recognize problems such as border dependency, which happens when the data is divided into tiles that will be processed by different devices. The constant need for communication between the GPUs in order to update the values that belong to the borders and are necessary for the next calculations are responsible for generating overhead. In order to overcome this problem, the authors suggest improvements in the program formulation, tile division, and tuning the program according to the GPU. In contrast to this work they do not target multi-node environments. Other high-level frameworks also propose the use a specific skeleton for stencil operations (e.g. SkePU [10] and FastFlow [2]). In contrast to our approach, none of them supports the combined use several cluster nodes each having multiple cores and multiple GPUs. SkePU's *MapOverlap* skeleton can be used over vectors (1D) and matrices (2D) and uses an OpenCL backend. The FastFlow stencil operation proposed by Aldinucci et al. [1] proposes a similar approach supporting the use of multiple GPUs. They state that there is no mechanism that allows communication between devices and therefore they propose the use of global memory persistence in order to reduce the need for data transfers between host and devices. Their experimental results expose this problem and show that the execution times are smaller using one GPU when compared to two GPUs for some instances where the cost of communication is higher than the cost of the other calculations. There exist further approaches to optimize stencil operations among others PATUS [5] and Pochoir [18]. However, they are limited in their functionality in contrast to a skeleton framework as they do not support other common programming patterns as e.g. reduce operations which impedes to write a program where multiple operations should be accelerated. Moreover, Pochoir does not support GPU execution and Patus does not support multi node execution. The tuning of stencil operations has been topic to many other publications [14, 21] as well as the investigation on memory spaces of the GPU [15, 19]. The rapid changes in the architecture of Nvidia GPUs especially regarding the sizes of the different memory spaces makes findings not transferable to recent architectures. Moreover, either the performance of single GPUs is tested or the framework is limited to stencil operations.

Our work enriches the research by two aspects. Firstly, not only (one node) multi-GPU setups are tested but the efficiency of heterogeneous computing environments as explained above (i.e. with multiple nodes) are investigated. Secondly, the levels of the GPU memory hierarchy are investigated further, to verify if the reduction in memory latency for shared memory outperforms the time needed to copy the data, for bigger instances.

3 Muenster Skeleton Library

In the beginning, skeletal parallel programming was mainly implemented in functional languages, since it derives from functional programming [6]. Today, the majority of skeleton frameworks are based on C/C++ (e.g. [2, 3, 9]). This is caused

by the good performance of C/C++ but also since imperative and object-oriented languages are prevalent for natural sciences and among HPC developers. Furthermore, frameworks for parallel computing such as CUDA, OpenMP, and MPI integrate seamlessly into C/C++ which allows employing different layers of parallelization.

The C++ library used in our approach is called Muesli [12]. Internally, it is based on MPI, OpenMP, and CUDA. It provides task- and data-parallel skeletons for clusters of multiple nodes, multicore processors, and GPUs. These are among others Fold, multiple versions of Map, Gather, and multiple versions of Zip.

Muesli relieves the programmer from low-level details 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 deadlocks. The abstraction level reduces the time needed to implement a program, while not increasing the execution time significantly. The algorithmic skeletons provided rely on two distributed data structures that Muesli provides: distributed arrays (DA) and matrices (DM). In the sequel, we will focus on data-parallel skeletons. Classic examples are Map, Fold, and Zip. 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. In the present paper, we will focus on the recent addition of a MapStencil skeleton. MapStencil gets a user function as an argument that specifies the exact operation to be executed for each element of a distributed matrix. Such a user function can be a sequential C++ function or a C++ functor. Functions and functors use the concept of currying, meaning that their arguments can be supplied one by one rather than all at the same time. Their last arguments are supplied by the skeleton. The snippet in Listing 1 shows the computation of the scalar product of two distributed arrays *a* and *b* (in a slightly simplified syntax).

```

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

```

Listing 1: Scalar product in Muesli.

4 MapStencil Skeleton

The MapStencil skeleton computes a matrix where the value of each element depends on the corresponding element in another matrix and on some of its neighbors within a square with a given height as a parameter, as depicted in Fig. 1.¹

¹ Other stencil shapes such as rectangular or irregular stencils can be handled by using the smallest surrounding square, although this may introduce some overhead.

A stencil determines to which neighbors the argument function of MapStencil is applied in order to determine the value of the considered element. Since the matrix is typically partitioned among the participating nodes, cores, and GPUs, computations at the border of each tile depend on the border of tiles that are assigned to other hardware units (nodes, cores, or GPUs). Thus, accessing them (often) requires time-consuming communication, which should hence be minimized. Since the size of the stencil is a parameter of the skeleton, the implementation of the skeleton has to cope with stencils of different sizes. The design of the skeleton will be explained by firstly discussing possible data distributions, and secondly explaining the user interface.

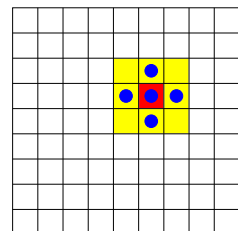
4.1 Data Distribution

As the goal is to design a MapStencil skeleton that works in multi-node and multi-GPU (per node) environments, one of the first considerations is the distribution of data. One possible approach is to distribute the data for the MapStencil skeleton as displayed in Fig. 2. For simplicity, we assume that the number of rows of the processed data structure is a multiple of the number of computational units. If this is not the case some padding has to be applied. The inter-node distribution assigns rows to each node, as displayed on the left-hand side of the figure. Between two nodes, an overlap exists as elements from the neighbor nodes are required for the calculation of the stencil operation. The size of the overlap depends on the stencil size and the size of a row. The row-wise distribution has the advantage that each node has to communicate with at most two other nodes. Moreover, only the upper and lower border need to be exchanged. The right and left values beyond each row can e.g. be treated as neutral values for the calculation. The first and last node only need to exchange the lower or upper border and start with the calculation as soon as they have received the border.

In case that the data would have been distributed into (e.g. square-shaped) blocks, one node might need to communicate with up to eight neighbors. In the case of a set-up with more than four nodes, a block-wise distribution might be beneficial, as fewer elements have to be exchanged. For an $n \times n$ matrix, a 3×3 stencil (we call this size $s = 1$), and a block size of $k \times k$, only up to $4(k + 1)$ elements need to be exchanged per node, rather than up to $2n$ for a row-wise distribution (for natural numbers n and k with $n > 2k > 0$).

For the intra-node distribution, also different approaches can be considered. Here, a row-wise distribution is often more efficient than a block-wise distribution as few computing nodes provide more than four GPUs (per node) and hence $k \geq n/2$ and

Fig. 1 MapStencil: the value of an element as the one depicted in red depends on those values in the surrounding square (here of size 3×3 and depicted in yellow) belonging to the considered stencil (here depicted in blue) (Color figure online)



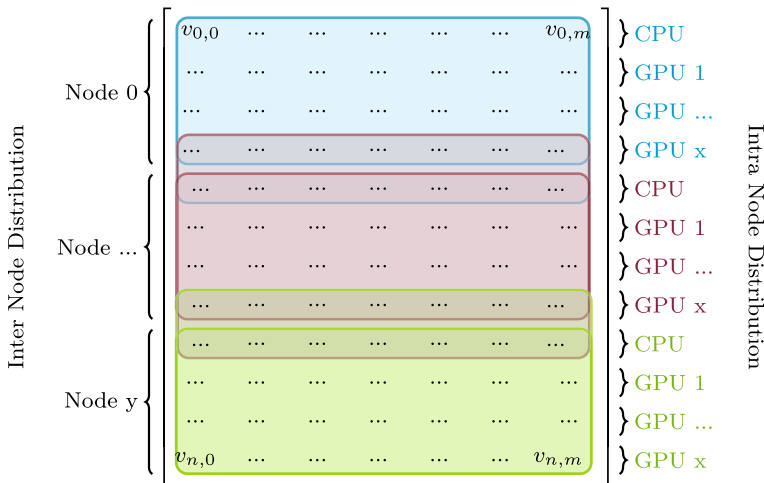


Fig. 2 Data distribution

$2n < 4(k + 1)$. One possible approach targeting a heterogeneous environment with CPUs and GPUs is displayed in Fig. 3. It assigns rows to the (multicore) CPU and the attached GPUs. Obviously, this requires communication depending on the stencil size and the number of rows between CPU and GPUs as well as between GPUs which is not depicted in the figure. In case that multiple cores are available per node, the number of (CPU) rows is divided between the cores.

Another approach is displayed in Fig. 4, using only GPUs. This has the advantage that no borders that are inside the node have to be communicated with the time-expensive device-to-host and host-to-device memory transfers but can be copied with device-to-device copy steps. Moreover, for CUDA-aware MPI versions it is not necessary at all to copy data from the GPU to the CPU and data can also be exchanged between GUPs on different nodes.

A last variation to the data distribution has been made in order to benefit from the shared memory of the GPUs. Shared memory belongs to one CUDA block² of threads started by a GPU. The number of blocks which can start concurrently depends on the GPU used and the number of threads started per block. The latency of shared memory is approximately $100\times$ lower than that of uncached global GPU memory.³ However, using shared memory is only efficient, if the data copied to the shared memory is accessed multiple times as the extra copy step to the shared memory costs time. Another drawback of shared memory is that it cannot be allocated persistently. Shared memory is always limited to one block in one kernel call. This means in case overhead is created it is created in each kernel call and therefore in each iteration.

Stencil operations are considered a good application scenario for using shared memory as one element is likely to be read multiple times. For example for

² Not to be confused with a block of a block-wise distribution.

³ <https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/> accessed 27.04.2021.

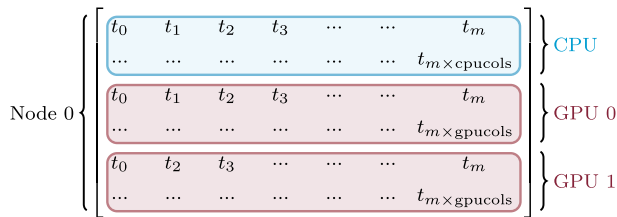


Fig. 3 Intra-node distribution using CPU

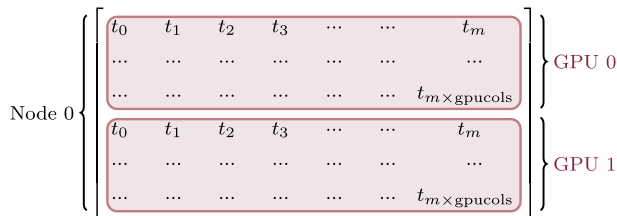


Fig. 4 Intra-node distribution without CPU

numerically solving the heat equation [8], a cross-shaped stencil as in Fig. 1 is required to calculate the current element. This means that each element is read four times. In Conway's game of life⁴ additionally, the diagonal neighbors are read. Hence, one element is read up to nine times. In the Gaussian blur, the stencil size varies depending on the parameters of the Gaussian function.

In order to make the best use of the shared memory, we have used a block-wise distribution of the data on the GPU for this application, as depicted in Fig. 5. Each color represents a block of threads on the GPU. The figure is shortened to increase the readability. For processing, for instance, a 16384×16384 matrix, it was divided into $512 \times 512 = 262,144$ blocks of size 32×32 , such that one block of the matrix corresponds to one CUDA block. In the figure, the surrounding block highlights the elements which need to be read to calculate the next element with a 3×3 -stencil. *nv* stands exemplarily for a neutral value. Due to hardware restrictions, one block never contains more than 32 rows and columns, since CUDA-capable GPUs cannot start more than 1024 threads per block.

The need for a block-wise distribution can be explained by a simple example. When processing a 1024×1024 matrix with one GPU, a row-wise distribution would result in having one row processed by one block. For calculating the heat distribution with a 3×3 stencil, the upper row, the current row, and the following row would need to be loaded for each block into the shared memory, loading 3072 elements in total per block. All elements in the upper and lower row would be read once, and all elements in the current row twice. If in contrast a 32×32 sub-matrix is processed by each block, each block needs to load $(32 \times 4) + 4 = 1028$ elements from the borders and 1024 elements to be processed. This amounts to only 2032 elements. Still, most border elements are only accessed once, but the elements processed are read twice for

⁴ <http://math-fail.com/2010/07/conways-game-of-life-in-html-5.html>.

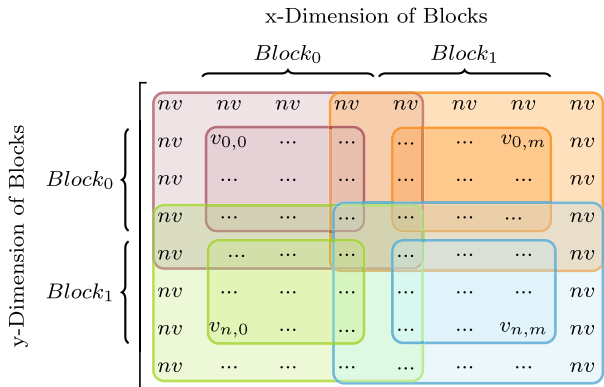


Fig. 5 Shared memory GPU distribution

the corners, three times for the borders, and four times for all 900 elements in the middle of the block. This advantage increases with repeated accesses to the data as e. g. in the game of life and the Gaussian blur.

For a stencil accessing just four elements, the block-wise distribution still reduces the number of elements loaded by $\sim 30\%$ and on average a loaded element is read more often than for a row-wise distribution. However, it has to be investigated experimentally at which point the time saved by shared memory accesses is sufficient to compensate for the time required to load the elements into shared memory.

The depicted distribution has one major disadvantage. Although GPUs are often treated as allowing almost unlimited parallelism, the number of threads which can be started is limited by the architecture. For example, the Nvidia RTX 2080 Ti has 68 streaming multiprocessors. Each multiprocessor can start at most 1024 threads in parallel for the Nvidia 7.5 GPU architecture. Additionally, those threads must be in one block; half a block cannot be processed by one multiprocessor. In the example given, 69,632 threads can run in parallel. For massive data structures, this results in starting threads after the first 69,632 have been finished. For normal Map Operations, this is of minor importance as there exists not a lot of overhead for initializing new blocks of threads. However, using the shared memory can make a huge difference as initializing new blocks requires loading elements again to the shared memory. The Nvidia GTX 2080 Ti has 64K of shared memory per multiprocessor. For big data structures, it is expected to be more efficient to let one thread calculate multiple elements as starting new threads will not directly invoke a parallel computation but produce overhead since the shared memory needs to be initiated again. Instead, it should be faster to start the maximum number of threads which can run in parallel and load more elements into the shared memory to avoid repeated copy calls. During implementation, the available size of the shared memory has to be checked. Figure 6 shows a simplified example of how threads could be reused. In the ideal example of having a 192×2016 data structure and the mentioned GPU, the columns can be split for each block to process 32 columns starting 68 blocks of each 1024 threads (32×32) and therefore exploiting the maximum number of blocks which can be started. Each multiprocessor has 64 KB shared memory (when configuration settings

are set appropriately); it can save up to 16.000 integers. Assuming we have a 3×3 stencil, all 12804 (66×194) elements can be loaded into the shared memory, calculating 12288 elements. As a result, the first block processes all rows of the first 32 columns, making maximum use of the shared memory by calculating four elements per thread.

Obviously, this example is chosen to illustrate the ideal case. It can be generalized that the number of threads started and the number of elements calculated by one thread is dependent on the hardware used and the size of the datastructure.

4.2 Using MapStencil

When computing a new matrix, the MapStencil skeleton must keep the original matrix and cannot just change it in place as old values need to be read.⁵

```

1  JacobiBorder borderFunc(n, m, 75);
2  JacobiSweepFunc(jacobi);
3  jacobi.setStencilSize(1);          // i.e. a 3x3 stencil
4  DM<float> differences(n, m, 0, true);
5  DM<float> board(n, m, 75, true);
6  DM<float> swpBoard(n, m, 75, true);
7  while (globalDiff > EPSILON && numIter < MAXITER) {
8      if (numIter % 50 == 0) {
9          board.mapStencilMM(swpBoard, jacobiFunc, borderFunc);
10         differences = board.zip(swpBoard, differenceFunc);
11         globalDiff = differences.fold(maxFunc, true);
12     } else {
13         if (numIter % 2 == 0) {
14             board.mapStencilMM(swpBoard, jacobiFunc, borderFunc);
15         } else {
16             swpBoard.mapStencilMM(board, jacobiFunc, borderFunc);
17         }
18     }
19     numIter++;
20 }
```

Listing 2: Jacobi method using the MapStencil Skeleton.

Exemplarily, the usage will be described by the computation of the static heat distribution. An excerpt of the program is shown in Listing 2. MapStencil requires three parameters: the input matrix to be processed, a functor for the stencil computation, and a functor for computations at the borders. At the start (line 1–3), the functors are initialized. For the stencil functor, the stencil size is set. Its parameter 1 is the distance from the center, i.e. we use a 3×3 stencil. The stencil functor could be enhanced by taking stencils with different sizes for width and height. However, this was considered a routine piece of work as the same data distribution as previously described can be used. For the border functor (line 1), the size of the matrix and the default value (here 75) are passed as arguments. Afterwards, three $n \times m$ distributed matrices are created (lines 4–7): `board` and `swpBoard` are used in turns to store

⁵ A variant of the skeleton which immediately overwrites old values by new values is however possible and could be applied, for instance, for implementing the Gauß-Seidel method for solving systems of linear equations.

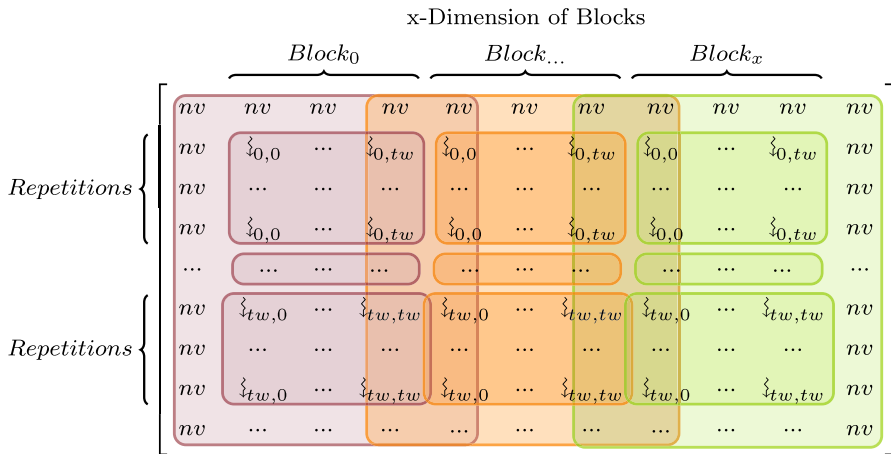


Fig. 6 Improved shared memory GPU distribution

the input and output heat matrix of each stencil computation, respectively, while differences is used to store their difference. The stencil computation needs to be repeated until either the maximal difference between two subsequent matrices is smaller than the desired EPSILON or the maximal number of iterations is reached (line 7). Every 50 iterations, it is checked how much the two data structures differ. Their difference is calculated by a Zip skeleton and from the resulting distributed matrix its maximal element is computed using a Fold skeleton (lines 10–11). One advantage of Muesli is that it differentiates between operations which require to update data and operations which already have the appropriate data on the computational unit. This feature enables to optimize the transition between different skeleton calls, as not all data is transferred between host and GPUs.

```

1  class JacobiBorder: public Functor2<int, int, float> {
2      public:
3          JacobiBorder(int glob_rows, int glob_cols, float default)
4              : glob_cols_(glob_cols), glob_rows_(glob_rows), default_(
5                  default) {}
6          MSL_USERFUNC
7          float operator()(int x, int y) const {
8              if (y < 0 || y > (glob_cols_ - 1)) {return 100;}
9              if (x < 0) {return 100;}
10             if (x > (glob_rows_ - 1)) {return 0;}
11             return default_;
12         }
13     private:
14         int glob_rows_;
15         int glob_cols_;
16         int default_ = 75;
17     };

```

Listing 3: Example of a functor for processing the border.

```

1  class JacobiSweepFunctor: public DMapStencilFunctor<float, float,
    JacobiNeutralValueFunctor> {
2      public:
3          JacobiSweepFunctor() : DMapStencilFunctor(){}
4          MSL_USERFUNC
5          float operator()(int rowIndex, int colIndex, PLMatrix<float> *
            input) const {
6              float sum = 0;
7              for (int i = -stencil_size; i <= stencil_size; i++) {
8                  if (i == 0)
9                      continue;
10                 sum += input.get(rowIndex+i, colIndex);
11                 sum += input.get(rowIndex, colIndex+i);
12             }
13             return sum / 4;
14         }
15 };

```

Listing 4: Example of stencil functor.

The functors for computing the border and the stencil can be seen in Listing 3 and Listing 4, respectively. In our example, we assume 100 degree centigrade at the top, left, and right border of the board and 0 degree centigrade at the bottom. The usage of a functor allows to define more complicated patterns such as cyclic or toroidal patterns as the functor can have arbitrary attributes (e. g. `glob_rows_`).

For the stencil functor, one argument is the distributed input matrix. By applying `get` to it with a row and a column index as parameters, the matrix element at this position can be accessed. In case that the row or column index is smaller than zero or bigger than the number of rows or columns minus 1, the framework internally uses the border functor.

5 Experimental Results

We have tested the MapStencil skeleton with the three example applications mentioned in Sect. 4, namely the Jacobi method, the Gaussian blur, and the game of life. We have measured the run-times and speedups for different hardware configurations with different numbers of nodes and different numbers of GPUs per node. We have also investigated different fractions of the computations assigned to the CPUs. Moreover, we have investigated the use of shared memory on the GPUs and different block sizes.

For the experiments, the HPC machine Palma II was used.⁶ Palma II has multiple nodes equipped with different GPUs. We have used configurations with up to four nodes with Ivybridge (E5-2695 v2) CPUs. Each node has 24 CPU cores and 4 GeForce RTX 2080 Ti GPUs.

For the first experiments, the program implementing the Jacobi method shown in Sect. 4 was used. Moreover, an implementation of the game of life has been considered in our experiments. It is played on a board of cells. Depending on the surrounding cells a considered cell will be alive or not after one iteration. In contrast

⁶ <https://confluence.uni-muenster.de/display/HPC/GPU+Nodes>.

to the Jacobi method, where the stencil accesses four neighbor elements, the calculation of the stencil for the game of life reads all eight surrounding elements and the current element.

5.1 CPU Usage

Firstly, we consider a configuration consisting of a single node with a CPU and up to 4 GPUs. We allocate different fractions of the calculation to the CPU and the rest to the GPUs. The findings are displayed in Table 1. In order to keep the dataset smaller, the run-times on two GPUs have been excluded in Table 1. The Jacobi solver is set to perform a maximum of 5000 iterations. All program executions were checked to perform the same number of iterations in order to ensure that different floating-point rounding behavior does not influence the comparison.

It can be seen that with a rising percentage of elements calculated on the CPU the run-time increases linearly. Even for small experimental settings having a 512×512 matrix, the version including the CPU requires 0.02 s without the CPU, 0.9 s when only calculating 5% on the CPU. When calculating 25% on the CPU this increases to 3.75 s. This finding repeats for bigger data structures and also when using multiple GPUs. Therefore, the CPU should only be used for communication and the GPU for calculation for compute intense tasks. As the differences increase for bigger data sizes, we conclude that with the communication involved in the MapStencil skeleton the produced overhead for including the CPU outweighs the advantages of outsourcing calculations to the CPU. This finding might not hold true for skeletons which do not require communication. In the sequel, we will merely use the CPU for communication purposes and let the GPUs do the calculations. For this configuration, the variant using four GPUs first performs worse than on just one GPU for a small matrix, but it gains slightly for bigger data sizes. This effect will be discussed in-depth in the next subsection.

5.2 Experiments on Multiple Nodes and GPUs Using Global Memory

In order to study the usefulness of using several GPUs and nodes, we will consider a bigger problem size. As we have seen, the use of multiple GPUs does not pay off, if the considered problem size is too small. Thus, we now let our Jacobi solver perform up to 10,000 iterations also on bigger matrices and only finish early if the results do not differ by more than 0.001. However, for the tested data sizes the program never finished early. This setting is used on a different number of nodes and GPUs per node. On the GPUs, we use global memory only. The run-times of the parallel program on 1–4 nodes with 1–4 GPUs each are displayed in Fig. 7. Some of the numbers are additionally listed in Table 2. In this table, a comparison to a sequential C++ implementation can be found running on a Skylake (Gold 5118) CPU and the corresponding speedups.

From the numbers, it can be seen that the run-time for small problem sizes (512×512 matrix) does not improve when adding more hardware. In this case, the initialization overhead outweighs the advantage of starting more threads concurrently. This changes with a growing data size. As can be seen in Fig. 7, the program

Table 1 Run-times (in s) for different fractions calculated by the CPU for the Jacobi Solver

| Matrix | 512 × 512 | | 1000 × 1000 | | 5000 × 5000 | |
|--------|---------------|---------------|-------------|---------|---------------|---------------|
| | 1 GPU | 4 GPUs | 1 GPU | 4 GPUs | 1 GPU | 4 GPUs |
| 0 | 0.0291 | 0.0505 | 0.0966 | 0.1579 | 1.5244 | 1.4389 |
| 5 | 0.9121 | 1.4287 | 3.3705 | 3.8408 | 77.8069 | 76.9942 |
| 10 | 1.6239 | 2.1743 | 6.1082 | 6.5869 | 146.3872 | 145.4040 |
| 15 | 2.3374 | 2.8821 | 8.8477 | 9.3290 | 215.5982 | 214.1764 |
| 20 | 3.0690 | 3.5955 | 11.5590 | 12.0882 | 284.9108 | 285.1225 |
| 25 | 3.7524 | 4.3260 | 14.2981 | 14.7681 | 356.1769 | 357.0563 |

Best values are depicted in bold

running on a single node becomes slower than the multinode programs for bigger matrix sizes, as more computations can be distributed between the nodes. Moreover, the programs using multiple GPUs are faster than a single GPU program. However, the improvement is far less than linear. For the biggest tested data size, it can be seen that for one node a speedup of ~ 38 is reached which does not significantly differ from the speed-up achieved for 5000 rows and columns processed. Adding more GPUs for one node improves the speedup to ~ 51 . The speedup is limited as the communication between the GPUs has to be synchronous. Adding more nodes improves the speedup clearly at a data size of 5000 rows and columns. Beforehand the initialization overhead restrains the advantage from the calculation. After that point, the program scales better. On 4 nodes the run-time is divided by 3. In contrast adding four GPUs reduces the run-time by $\sim 0.25\%$ ($1 - (69.1/93.2)$).

As a second example, the game of life was considered. The sequential program ran also on a Skylake (Gold 5118) CPU. The game of life was configured to calculate 20000 iterations. The interesting difference is that the game of life reads more elements for the stencil calculation. Moreover, in contrast to the Jacobi Solver, the difference to the previous iteration is not checked. Hence, the calculation takes more time in contrast to the required communication. As can be seen in Table 3, this alters the speedup. The speedup is considerably better than for the Jacobi Solver. Partly, it has to be taken into account that the corresponding sequential program requires more time. But mentionable for the 4096×4096 matrix more speedup is achieved than for the $10,000 \times 10,000$ matrix of the Jacobi Solver (Table 4).

Adding more GPUs for the smallest problem size leads to a slow down. For a matrix with 4096 rows and columns adding more GPUs improves the speedup from 262.318 to 430.864 for two GPUs and to 527.098 for four GPUs. For a matrix with 8192 rows and columns adding more GPUs improves the speedup from 265.071 to 495.936 for two GPUs and to 816.368 for four GPUs. This highlights that with an increasing data size it is advantageous to use more GPUs. For two nodes the speedup increases from 265.071 to 503.238 for one GPU for the biggest tested data size. However, the repeated transfer operations make the speedup improvement from one node to two nodes slightly worse, for four GPUs from 816.368 to 1142.374. This effect becomes more obvious when comparing the run-time to four nodes. In that

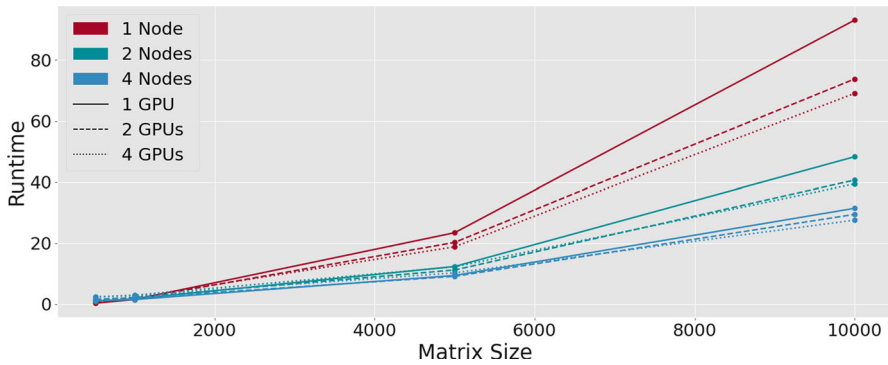


Fig. 7 Run-times (in s) on multiple nodes and GPUs per node using GPU global memory only for the Jacobi Solver

Table 2 Run-times (in s) and speedups on multiple nodes and GPUs per node only using GPU global memory for the Jacobi Solver

| Matrix size | Seq. C++ | 1 GPU | Speedup | 2 GPUs | Speedup | 4 GPU | Speedup |
|---------------------|----------|--------|----------------|--------|---------|--------|----------------|
| <i>1 Node</i> | | | | | | | |
| 512 ² | 9.406 | 0.298 | 31.532 | 0.608 | 15.475 | 1.417 | 6.639 |
| 1000 ² | 35.488 | 1.497 | 23.702 | 1.642 | 21.616 | 2.172 | 16.341 |
| 5000 ² | 888.922 | 23.420 | 37.955 | 20.237 | 43.925 | 18.793 | 47.302 |
| 10,000 ² | 3544.270 | 93.156 | 38.047 | 73.815 | 48.015 | 69.092 | 51.298 |
| <i>2 Nodes</i> | | | | | | | |
| 512 ² | 9.406 | 0.955 | 9.854 | 1.378 | 6.827 | 2.326 | 4.043 |
| 1000 ² | 35.488 | 1.700 | 20.873 | 2.138 | 16.600 | 2.904 | 12.217 |
| 5000 ² | 888.922 | 12.334 | 72.071 | 11.185 | 79.472 | 12.218 | 72.758 |
| 10,000 ² | 3544.270 | 48.285 | 73.404 | 40.636 | 87.221 | 39.388 | 89.984 |
| <i>4 Nodes</i> | | | | | | | |
| 512 ² | 9.406 | 1.097 | 8.573 | 1.609 | 5.846 | 2.331 | 4.035 |
| 1000 ² | 35.488 | 1.469 | 24.156 | 2.028 | 17.498 | 2.775 | 12.786 |
| 5000 ² | 888.922 | 9.390 | 94.668 | 9.092 | 97.774 | 10.243 | 86.780 |
| 10,000 ² | 3544.270 | 31.435 | 112.749 | 29.443 | 120.379 | 27.551 | 128.642 |

Numbers in bold face are mentioned in the text

case, each node processes only 2048 rows for the 8192×8192 matrix. The GPUs are not fully occupied and the additional overhead of initializing a GPU on another node and transferring the data slows down the overall program.

5.3 Experiments on Multiple Nodes and GPUs Using Shared Memory

Another aspect to inspect is the run-time when using shared memory. Thus, we varied how many threads are started, which affects how many elements are loaded

Table 3 Run-times (in s) and speedups on multiple nodes and GPUs per node only using GPU global memory for the game of life

| Matrix size | Seq. C++ | 1 GPU | Speedup | 2 GPUs | Speedup | 4 GPUs | Speedup |
|-------------------|-----------|--------|----------------|--------|---------|--------|-----------------|
| <i>1 Node</i> | | | | | | | |
| 1024 ² | 193.444 | 1.083 | 178.643 | 1.330 | 145.490 | 2.370 | 81.614 |
| 4096 ² | 2957.470 | 11.275 | 262.318 | 6.864 | 430.864 | 5.611 | 527.098 |
| 8192 ² | 11891.900 | 44.863 | 265.071 | 23.979 | 495.936 | 14.567 | 816.368 |
| <i>2 Nodes</i> | | | | | | | |
| 1024 ² | 193.444 | 2.077 | 93.123 | 2.989 | 64.722 | 4.456 | 43.412 |
| 4096 ² | 2957.470 | 6.627 | 446.262 | 4.720 | 626.525 | 5.730 | 516.132 |
| 8192 ² | 11891.900 | 23.631 | 503.238 | 14.166 | 839.476 | 10.410 | 1142.374 |
| <i>4 Nodes</i> | | | | | | | |
| 1024 ² | 193.444 | 2.565 | 75.424 | 4.037 | 47.916 | 6.940 | 27.876 |
| 4096 ² | 2957.470 | 6.427 | 460.170 | 8.328 | 355.141 | 11.968 | 247.117 |
| 8192 ² | 11891.900 | 23.344 | 509.416 | 18.282 | 650.481 | 18.588 | 639.758 |

Best values are depicted in bold

into the shared memory. In Fig. 8 the run-times for starting 64 threads (8×8 tile) and 256 (16×16 tile) threads are shown. The run-time slightly improves when loading more elements in contrast to fewer elements. However, it is still slower than the global memory program. The loading of the elements into the shared memory produces too much overhead to speed up the program. This overhead is produced for each kernel call. For the biggest data size, the run-times for one GPU are 93.156 s without shared memory and 100.205 s with shared memory, while for four GPUs the run-time increases from 69.092 s to 73.108 s. For multiple nodes this difference decreases, however it is expected that with multiple calls also in this setting overhead is produced. Additionally to the proposed optimization in the data distribution, the

Table 4 Run-times (in s) on multiple nodes and GPUs per node using GPU global memory (GM) versus shared memory (SM) for the Jacobi Solver

| Matrix size | 1 GPU | | | 4 GPUs | | |
|---------------------|---------------|-------------------|--------------------|---------------|-------------------|-------------------------|
| | GM | SM 8 ² | SM 16 ² | GM | SM 8 ² | SM 16 ² size |
| <i>1 Node</i> | | | | | | |
| 512 ² | 0.298 | 0.307 | 0.307 | 1.417 | 1.399 | 1.371 |
| 1000 ² | 1.497 | 1.615 | 1.616 | 2.172 | 2.091 | 1.786 |
| 10,000 ² | 93.156 | 100.205 | 99.569 | 69.092 | 73.108 | 73.146 |
| <i>2 Nodes</i> | | | | | | |
| 512 ² | 0.954 | 0.890 | 0.907 | 2.326 | 2.317 | 2.258 |
| 1000 ² | 1.700 | 1.670 | 1.675 | 2.905 | 2.630 | 2.599 |
| 10,000 ² | 48.285 | 51.402 | 51.311 | 39.388 | 38.673 | 39.675 |

Best values are depicted in bold

CUDA configurations were specified to `cudaDeviceSetCacheConfig(cudaFuncCachePreferShared)` and `cudaDeviceSetSharedMemConfig(cudaSharedMemBankSizeEightByte)`. The first setting prefers shared memory, which means that more shared memory space is available. The second setting sets the bank size to eight bytes to support floating-point number access. Both settings did not significantly affect the run-time.

As previously said, the performance of the shared memory depends on multiple factors, among others how often elements are read. Hence, it was expected, that the game of life should profit from the shared memory, as more elements are read. The results only slightly change for using the shared memory for calculating the game of life, as can be seen in Table 5. This means reading one element up to nine times does not suffice in our case to speed up the program significantly. For one GPU the run-time increases from 44.86 to 48.55 s and 47.60 s. This difference is slightly smaller than the previous one but still slower than using the global memory.

To further examine the shared memory the Gaussian blur is added as an example. The Gaussian blur uses the Gaussian function to reduce detail in images. This perfectly suits the application context, as the stencil size can be varied as a parameter of the function. The results evaluate the run-time of the calculation of the Gaussian blur over 100 iterations, with a 512×512 pixel-sized image. The number of iterations had to be increased to ensure that the run-times are meaningful. The run-times are listed in Table 6 and shown in Fig. 9. The listed stencil sizes go from 11×11 to 21×21 stencils. This means for calculating one pixel 121–441 elements have to be read.

It can be seen that starting less than 64 threads (8×8 tile) results in a major slowdown. Starting not enough threads slows down the whole program as the parallelism cannot be exploited. Starting 64 or 256 threads results in a minor

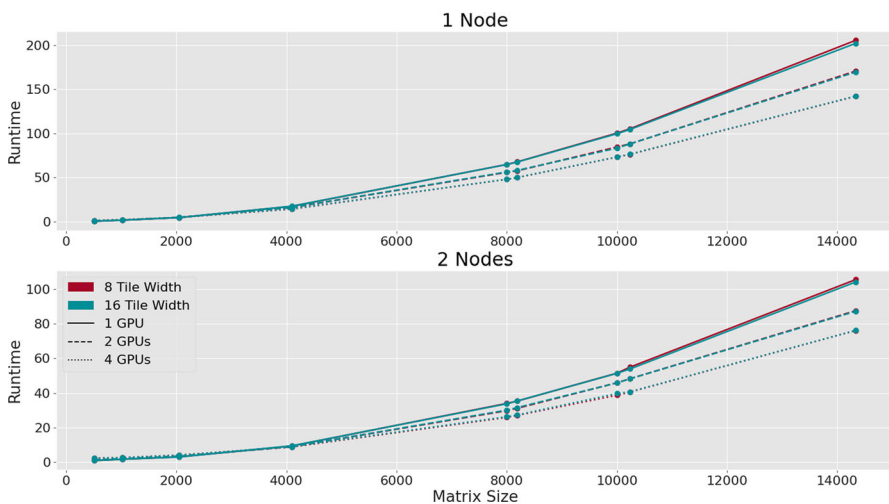


Fig. 8 Run-times (in s) on multiple nodes and GPUs per node using GPU shared memory for the Jacobi Solver

speedup. For the tested problem, 64 threads are the best solution found. However, the speedup is less than 1.09 having only minor effects on the run-time. Starting more threads slows down the run-time, which could be due to bank conflicts. To underline our finding it is compared to a hand-written parallel program. Table 7 lists the speedups for the different tile sizes. Those run-times only include the calculation of the Gaussian blur and not the data transfers from CPU to GPU. As can be seen, the differences are minimal.

Another application context is to process a larger picture. We changed the processed picture to a 6000×4000 image, making it plausible to let one thread calculate multiple data points as proposed by other publications (such as e.g. [15, 19]). We tested the program with letting one thread calculate one to eight elements. More elements are not reasonable as the shared memory is limited. From the run-time, the optimal tile size was chosen, which was in all displayed cases four elements per thread. The results can be seen in Table 8. Letting one thread calculate multiple elements slightly improved the low-level program's run-time. Depending on the application context, shared memory might provide more speedup. However, it did not significantly accelerate this application.

This observation can be attributed to the fact that in contrast to previous hardware, newer GPUs have a big L2 Cache. The access time for L2 Cache is close to shared memory and reduces the potential for shared memory to speed up the program. Therefore, findings from previous publications have to be verified on newer hardware as we could not reproduce their results. In conclusion, only stencil operations with a very big stencil size or repeated access to the elements should use the shared memory.

6 Discussion and Outlook

The major focus of our work was the data distribution between nodes and computational units. Our findings could be extended by analyzing n -dimensional data structures (with $n > 2$). We presume that for n -dimensional data structures,

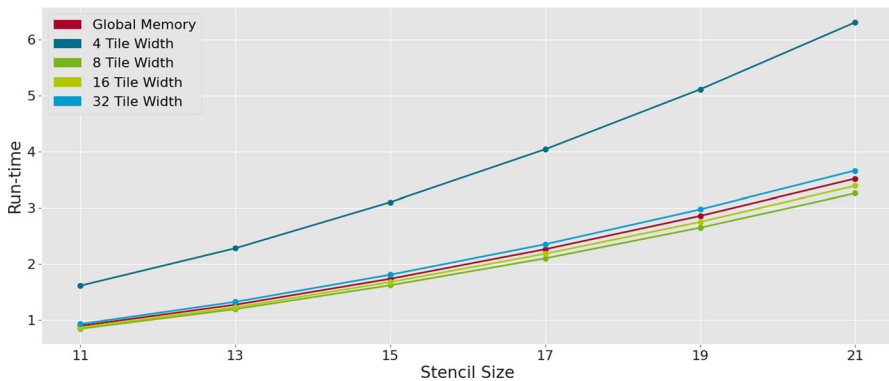
Table 5 Run-times (in s) on multiple nodes and GPUs per node using GPU global memory (GM) versus shared memory (SM) for the game of life

| Matrix size | 1 GPU | | | 4 GPUs | | |
|-------------------|---------------|-------------------|--------------------|--------|-------------------|--------------------|
| | GM | SM 8 ² | SM 16 ² | GM | SM 8 ² | SM 16 ² |
| <i>1 Node</i> | | | | | | |
| 1024 ² | 1.083 | 1.361 | 1.314 | 2.370 | 4.221 | 4.238 |
| 4096 ² | 11.274 | 12.279 | 11.953 | 5.611 | 19.588 | 19.058 |
| 8192 ² | 44.863 | 48.552 | 47.606 | 14.567 | 70.207 | 69.336 |
| <i>2 Nodes</i> | | | | | | |
| 1024 ² | 2.077 | 1.770 | 1.765 | 4.456 | 3.857 | 3.823 |
| 4096 ² | 6.627 | 7.509 | 6.747 | 5.730 | 5.452 | 5.684 |
| 8192 ² | 23.631 | 26.356 | 24.614 | 10.410 | 11.100 | 10.403 |

Best values are depicted in bold

Table 6 Run-times (in s) and speedups on a single node using GPU global memory and shared memory for the Gaussian blur (muesli)

| Stencil-size | Seq. C++ | GM | Tile size | | | | | Speedup (to GM) | Speedup seq. |
|--------------|----------|-------|----------------|----------------|----------------|-----------------|-----------------|-----------------|--------------|
| | | | 2 ² | 4 ² | 8 ² | 16 ² | 32 ² | | |
| 11 | 129.73 | 0.899 | 6.603 | 1.616 | 0.848 | 0.869 | 0.931 | 1.061 | 153.020 |
| 13 | 183.30 | 1.276 | 9.309 | 2.284 | 1.197 | 1.229 | 1.327 | 1.066 | 153.146 |
| 15 | 241.93 | 1.739 | 12.679 | 3.105 | 1.624 | 1.681 | 1.812 | 1.071 | 148.999 |
| 17 | 308.87 | 2.267 | 16.547 | 4.046 | 2.104 | 2.187 | 2.356 | 1.078 | 146.801 |
| 19 | 383.32 | 2.861 | 20.942 | 5.116 | 2.651 | 2.753 | 2.976 | 1.079 | 144.584 |
| 21 | 467.19 | 3.523 | 25.859 | 6.312 | 3.262 | 3.397 | 3.667 | 1.080 | 143.226 |

**Fig. 9** Run-times (in s) for the Gaussian blur with different tile sizes on one GPUs using GPU global memory or GPU shared memory**Table 7** Speedups for a low-level implementation in contrast to Muesli on a single node using GPU global memory and shared memory for the Gaussian blur

| Stencil-size | GM | Tile size | | | | |
|--------------|-------|----------------|----------------|----------------|-----------------|-----------------|
| | | 2 ² | 4 ² | 8 ² | 16 ² | 32 ² |
| 11 | 1.013 | 1.072 | 1.021 | 1.032 | 1.025 | 0.992 |
| 13 | 1.009 | 1.055 | 1.010 | 1.017 | 1.011 | 0.972 |
| 15 | 1.001 | 1.050 | 1.009 | 1.013 | 1.012 | 0.976 |
| 17 | 0.997 | 1.040 | 1.001 | 1.008 | 0.999 | 0.919 |
| 19 | 0.987 | 1.040 | 1.004 | 1.008 | 0.999 | 0.969 |
| 21 | 0.982 | 1.039 | 1.004 | 1.001 | 0.998 | 0.960 |

block-wise distribution becomes more efficient. For example, a three-dimensional data structure of size $64 \times 64 \times 64$, which should be distributed on eight nodes either the data could be distributed fully exploiting one dimension and then switching to the next dimension, which would result in having blocks of size $8 \times 64 \times 64$ per node.

Assuming each node has 8 GPUs and each GPU has a split of $1 \times 64 \times 64$. In contrast splitting the data block-wise each node has $32 \times 32 \times 32$ elements. Continuing, each GPU has one block of $16 \times 16 \times 16$ elements. Assuming a 3×3 and a 9×9 stencil, the first distribution requires each GPU to load 8,972, 42,560 ($3 \times 66 \times 66 - 4096$, $9 \times 72 \times 72 - 4096$) surrounding elements. In contrast, the block-wise distribution requires to load only 1,736, 9,728 ($18 \times 18 \times 18 - 4096$, $24 \times 24 \times 24 - 4096$) elements.

Irregular stencils are another interesting topic. We can handle them by embedding them into the next biggest surrounding square, but this may introduce some overhead. Avoiding this overhead is difficult in a library such as Muesli. An optimization could be handled by e.g. a pre-compiler that analyzes patterns of irregular accesses and chooses an appropriate data exchange scheme.

7 Conclusions and Future Work

MapStencil computes a matrix where the value of each element depends on the corresponding element in another matrix and on some of its neighbors. We have added a MapStencil skeleton to the algorithmic skeleton library Muesli. The corresponding implementation supports all typical hardware levels, i.e. a cluster consisting of several nodes each providing one or more multicore CPUs and possibly multiple GPUs. To the best of our knowledge, there is no MapStencil implementation yet, which also supports any combination of these hardware levels. In the present paper, we have focussed on the data distribution, the load distribution between CPU and GPUs, the usefulness of multiple nodes and GPUs for MapStencil, and the question whether MapStencil should use GPU shared memory. From the point of view of the users, MapStencil processes a whole distributed matrix in one step. They do not have to bother about the transformation of global indexes to local ones and vice versa or about complicated communication steps between GPUs and CPUs or between nodes.

As an example program, the Jacobi solver for systems of linear equations has been used. The results showed for the case of many iterations that using the CPU in addition to the GPUs slows down the program significantly. For this reason, in the

Table 8 Run-times (in s) using GPU global memory (GM) versus shared memory (SM) for the Gaussian blur letting each thread calculate the optimal number of elements

| Stencil size | GM | SM 4 ² | SM 8 ² | SM 16 ² | SM 32 ² | Speed-up |
|------------------|-------|-------------------|-------------------|--------------------|--------------------|----------|
| <i>Muesli</i> | | | | | | |
| 11 | 6.9 | 13.25 | 6.83 | 6.89 | 7.02 | 1.01 |
| 21 | 27.57 | 52.98 | 27.11 | 27.42 | 28.01 | 1.02 |
| <i>Low level</i> | | | | | | |
| 11 | 7.61 | 11.82 | 5.95 | 5.92 | 6.02 | 1.29 |
| 21 | 27.71 | 50.86 | 23.87 | 23.63 | 24 | 1.17 |

Best values are depicted in bold

following experiments, the CPU was merely used to communicate between nodes. This is of course different in a setting without GPUs. Next, a version using the global memory of the GPU was tested with up to four nodes each equipped with up to four GPUs. For the program implementing the Jacobi solver, a single GPU achieves a speed-up of ~ 38 . Four nodes with one GPU each achieve a speed-up of ~ 112 . To complement the Jacobi solver, the game of life was also considered in our experiments. One of the most important differences is that the game of life reads nine elements to calculate the stencil while the Jacobi Solver reads only four elements. This increases the proportion of the required calculation time and therefore improves the speedup. For a single GPU a speedup of ~ 265 could be achieved and for four GPUs a speedup of ~ 816 . Using the GPU shared memory could not achieve a notable benefit. As part of the experiments, the number of elements loaded into the shared memory was tested. An increasing number of elements loaded could neither improve the results for the game of life nor for the Jacobi solver. Letting one thread calculate multiple elements improved the speedup slightly. As future work, we want to experiment with CUDA-aware MPI. Unfortunately, the MPI version on our hardware is not CUDA-aware. As soon as we get a CUDA-aware MPI, we would like to check out how far this enables improvements of our skeleton implementations.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Aldinucci, M., Danelutto, M., Drocco, M., Kilpatrick, P., Pezzi, G.P., Torquati, M.: The loop-of-stencil-reduce paradigm. In: 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 3, pp. 172–177. IEEE (2015)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing (2017)
3. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eSkel. In: European Conference on Parallel Processing, pp. 761–770. Springer (2005)
4. Cheikh, T.L.B., Aguiar, A., Tahar, S., Nicolescu, G.: Tuning framework for stencil computation in heterogeneous parallel platforms. *J. Supercomput.* **72**(2), 468–502 (2016)
5. Christen, M., Schenk, O., Burkhart, H.: Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Comput. Sci. Res. Dev.* **26**(3), 205–210 (2011)
6. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Pitman, London (1989)
7. Corporation, N.: Cuda. <https://developer.nvidia.com/cuda-zone> (2021). Accessed 10 May 2021
8. Crank, J., Nicolson, P.: A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. In: Mathematical Proceedings of the Cambridge Philosophical Society, vol. 43, pp. 50–67. Cambridge University Press (1947)

9. Emoto, K., Fischer, S., Hu, Z.: Generate, test, and aggregate. In: Seidl, H. (ed.) *Programming Languages and Systems*, pp. 254–273. Springer, Berlin (2012)
10. Enmyren, J., Kessler, C.W.: Skepu: a multi-backend skeleton programming library for multi-gpu systems. In: *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, pp. 5–14 (2010)
11. 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)
12. Ernsting, S., Kuchen, H.: Data parallel algorithmic skeletons with accelerator support. *Int. J. Parallel Prog.* **45**(2), 283–299 (2017)
13. Forum, M.: Mpi standard. <https://www.mpi-forum.org/docs/> (2021). Accessed 10 May 2021
14. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 100–112 (2018)
15. Mei, X., Chu, X.: Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.* **28**(1), 72–86 (2017). <https://doi.org/10.1109/TPDS.2016.2549523>
16. Öhberg, T., Ernstsson, A., Kessler, C.: Hybrid CPU-GPU execution support in the skeleton programming framework SkePU. *J. Supercomput.* **76**(7), 5038–5056 (2020)
17. OpenMP: OpenMP the openMP API specification for parallel programming. <https://www.openmp.org/> (2021). Accessed 10 May 2021
18. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.K., Leiserson, C.E.: The pochoir stencil compiler. In: *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 117–128 (2011)
19. Van Werkhoven, B., Maassen, J., Seinstra, F.J.: Optimizing convolution operations in cuda with adaptive tiling. In: *2nd Workshop on Applications for Multi and Many Core Processors (A4MMC 2011)* (2011)
20. 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)
21. Zhang, Y., Mueller, F.: Auto-generation and auto-tuning of 3d stencil codes on GPU clusters. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pp. 155–164 (2012)