



# Optimizing Three-Dimensional Stencil-Operations on Heterogeneous Computing Environments

Nina Herrmann<sup>1</sup> · Justus Dieckmann<sup>1</sup> · Herbert Kuchen<sup>1</sup>

Received: 11 September 2023 / Accepted: 1 May 2024 / Published online: 21 June 2024  
© The Author(s) 2024

## Abstract

Complex algorithms and enormous data sets require parallel execution of programs to attain results in a reasonable amount of time. Both aspects are combined in the domain of three-dimensional stencil operations, for example, computational fluid dynamics. This work contributes to the research on high-level parallel programming by discussing the generalizable implementation of a three-dimensional stencil skeleton that works in heterogeneous computing environments. Two exemplary programs, a gas simulation with the Lattice Boltzmann method, and a mean blur, are executed in a multi-node multi-graphics processing units environment, proving the runtime improvements in heterogeneous computing environments compared to a sequential program.

**Keywords** Skeleton programming · Three-dimensional stencil operations · High-level parallel programming

## 1 Introduction

The field of high performance computing (HPC) is growing as intricate algorithms require more processing power and the demand for handling larger datasets rises. Efficient parallel programs are necessary for evaluating massive datasets. Most HPC environments incorporate numerous nodes equipped with multiple central processing units (CPUs) and graphics processing units (GPUs). Creating programs that combine multiple nodes and accelerators necessitates an understanding of low-level frameworks such as implementations of the message passing interface (MPI) [1], OpenMP [2], and CUDA [3]. Moreover, writing a parallel program is error-prone and tedious, e.g., out-of-memory errors and invalid memory accesses are troublesome to identify even for

---

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

Justus Dieckmann  
justus.dieckmann@uni-muenster.de

Herbert Kuchen  
kuchen@uni-muenster.de

<sup>1</sup> Practical Computer Science, University of Münster, Leonardo-Campus 3, 48149 Münster, Germany

skilled programmers. Furthermore, faultless programs need to be optimized as the selection of memory spaces, distributing data, and thread task allocation are design choices that significantly impact performance but require experience, which scientists usually lack.

Since experts in this field are hard to find, high-level frameworks are a convenient alternative to bypass the tedious creation of parallel programs. High-level frameworks commonly abstract from the distribution of data, provide portable code for different hardware architectures, are adjustable to distinct accelerators, and require less maintenance for the end user. In 1989, COLE introduced algorithmic skeletons enclosing reoccurring parallel and distributed computing patterns as one of the most common approaches to abstract from low-level details [4]. Multiple libraries [5, 6], general frameworks [7, 8], and domain-specific languages [9] use the concept.

The paper contributes to the ongoing work by focusing on a particularly arduous operation, namely the three-dimensional stencil operation. Stencil operations calculate elements depending on neighboring elements within the data structure and therefore require communication between the computational units used. Those operations are essential for, e.g., computational fluid dynamics of gas or airflow. Current high-level approaches are not capable of efficiently updating data in a generalized way and dealing with three-dimensional data structures.

This paper initially presents related work, concentrating on high-level approaches that abstract from problem-specific details (Sect. 2). Section 3 outlines the Muesli library used, while Sect. 4 explains the additional implementation of the three-dimensional skeleton and the examples used to measure the runtime. Section 5 evaluates the work, discussing our runtime experiments on various hardware set-ups, and compares our results to other frameworks. Finally, a summary of our research is given in Sect. 6.

## 2 Related Work

Ongoing work discussing high-level approaches to three-dimensional stencil operations is diverse. The work can be distinguished by examining the types of accelerators used, the operations available, and the maintenance of the frameworks. Table 1 lists research regarding high-level approaches to parallelize stencil operations to the best of the authors' knowledge.

Most related in the area of high-level skeleton programming frameworks, SkePU3 targets multi-node and multi-GPU environments for most skeletons in combination with StarPU. However, for stencil operations (called MapOverlap), the data exchange between the programs is missing for multi-node programs [7]. Celerity is a high-level approach focusing on parallel for-loops, including stencil operations [10]. FastFlow added GPU support but focuses on communication skeletons and misses a comparable stencil operation [8, 11]. Lift handles n-dimensional stencils on single GPUs missing the combination of multiple nodes and accelerators [12]. SkelCL implements a MapOverlap skeleton for multiple GPUs missing multiple nodes [13]. Moreover, it has not been adapted to hardware changes since 2016. The high-level block-structured framework WaLBerla discusses options for optimization in detail. Unfortunately, the experiment section does not discuss the use of multiple GPUs [14]. EPSILOD tests

**Table 1** Overview of frameworks that generate parallel code for stencil operations

Name	Single GPU	Multi-GPU <sup>1</sup>	Generic operations	Publically available	Updated
SkePU [7]	✓	✓ <sup>2</sup>	✓	✓ [23]	Until now
Celerity	✓	✓	✓	✓ [24]	Until now
Lift [12]	✓	X	✓	✓ [25]	Until now
SkeLCL [13]	✓	✓	X	✓ [26]	2016
lbmpy [14] [27]	✓	X	X	✓ [28, 29]	Until now
EPSILOG [15]	✓	✓	X <sup>3</sup>	✓ [30]	Until now
ExaStencil [16]	✓	✓	X	✓ [31]	Until now
DUNE [17]	X	X	X	✓ [32]	Until now
PARTANS [18]	✓	✓	X	X	2013
PSkel [19]	✓	X	X	✓ [33]	2017
OpenACC Extension [20]	✓	X	✓ <sup>4</sup>	X	2017
Palabos [21]	✓ <sup>5</sup> [35]	X	X	✓ [34]	Until now

<sup>1</sup>Published at least one experiment with multiple GPUs<sup>2</sup>Currently not working for the MapStencil skeleton<sup>3</sup>No option to create the stencil required for an LBM implementation<sup>4</sup>Enabled through OpenACC directives<sup>5</sup>(library import for LBM)

prototypes for stencil operations but misses a generic stencil implementation [15]. ExaStencil provides a domain-specific language for stencil operations, therefore it misses the option to parallelize pre- and post-processing steps [16]. DUNE provides the means to write parallel programs in C++ but merely for CPUs [17]. Partans and PSkel developed a framework for stencil operations but have not been adopted [18] [19]. Pereira et al. [20] have expanded OpenACC for single GPUs stencil operations, however, their project is not publically available [20]. Palabos proved that frameworks generating GPU code (npFEM) can be included in the framework to solve the Lattice Boltzmann methods (LBMs) [21], however it is unclear if this feature is still supported. Publications discussing a single method, e.g., the Helmholtz equation [22] focus on the algorithm and do not include accelerators like GPUs.

This work extends the mentioned work, as the presented stencil skeleton is generalizable for multiple applications, allows pre- and post-processing steps, and runs on multiple nodes and accelerators. The implementation of the skeleton is tested with two examples: (1) a version of a LBM (2) a three-dimensional mean blur.

### 3 The Muenster Skeleton Library Muesli

Nowadays, most skeleton frameworks are implemented in C/C++ [6–8, 36–38], as it offers interoperability with multiple parallel frameworks such as OpenMP, MPI, CUDA, and OpenCL and is exceptionally performant. Noteworthy, Python recently gained attention for natural science applications as it provides an easy interface to write packages in C/C++ [14, 27].

The used library is called Muenster Skeleton Library (Muesli) [5]. Muesli provides an object-oriented approach that offers one-, two-, and three-dimensional data structures (DA, DM, DC) with skeletons as member functions. The supported skeletons are, for example, multiple versions of Map and Zip (index and inplace variants), Fold, Gather, and, as discussed in this work MapStencil. Internally, MPI, OpenMP, and CUDA are used, which enables simultaneous parallelism on multiple nodes, CPUs, and GPUs. The library can be included with a simple include statement `#include<muesli.h>`. For writing a parallel program, Muesli provides methods to state, among others, the number of processes and GPUs used. Apart from that, Muesli abstracts from parallel programming details by internally distributing the data structures on the available computational units, choosing the number of threads started on the corresponding low-level framework, and copying data to the correct memory spaces. This abstraction also reduces errors commonly made by inexperienced programmers, such as race conditions and inefficient data distribution.

Listing 1 shows a simple program calculating the Scalar product of the distributed arrays `a` and `b` in Muesli. In line 8, a distributed array of size three with a default value of 2 is created. In the skeleton calls in lines 9–11, it can be seen that skeletons have a user function as an argument which can either be a C++ function or a C++ functor. For the `index` variant of `map`, Muesli applies the argument function of `map` to each element of the data structure (here a distributed array (DA)) and its index (line 9). For the `zip` skeleton, the second required data structure is passed as an argument. Lastly, lines 9+11 show that the same function can be used in different contexts, firstly for calculating the sum of the index and the value and secondly as a reduction operator.

```

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

```

Listing 1: Scalar product in Muesli.

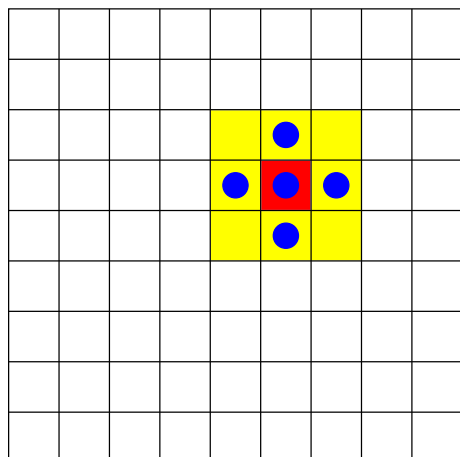
## 4 Three-Dimensional Stencil Operations

Stencil operations are map operations that additionally require reading the surrounding elements of each considered element. Figure 1 displays a two-dimensional stencil with radius one. The peculiarity regarding stencil operations on multiple nodes and accelerators is that each execution of the stencil operation requires updating elements that are shared between computational units. As communication of updated elements requires synchronization between the computational nodes, it decreases the opportunity for executing tasks in parallel within the program. Muesli abstracts from all communication between the computational nodes with a MapStencil skeleton.

### 4.1 Using the MapStencil Skeleton in Muesli

The usage of the 3-dimensional mapStencil skeleton for the end-user is shown in Listing 2. Firstly, a function to be executed on each element is defined (l. 1–10).

**Fig. 1** Exemplary two-dimensional stencil operation



The first argument of the function has to be of type `PLCube` (*PaddedLocalCube*), and the subsequent arguments must be integers for indexing the data structure. The class `PLCube` most importantly offers a `getter`-function taking three index arguments, relieving the end-user from index calculations (l. 8). The presented functor calculates the sum of all elements with a radius of two. It divides the sum by the number of total elements, therefore calculating a mean filter. This functor can be applied to a distributed cube by calling the `mapStencil` skeleton as a member function (l. 13). The skeleton takes the functor as a template argument and requires a distributed cube of the same dimension with the current data,<sup>1</sup> the radius of the stencil<sup>2</sup> and the neutral value for border elements.

```

1  template <size_t radius>
2  MSL_USERFUNC float update(const PLCube<float> &plCube,
3  int x, int y, int z) {
4      float res = 0;
5      for (int mx = x - radius; mx <= x + radius; mx++) {
6          for (int my = y - radius; my <= y + radius; my++) {
7              for (int mz = z - radius; mz <= z + radius; mz++) {
8                  res += plCube(mx, my, mz);}}
9      return res/(radius*radius*radius);}
10 main () { // skipped initialization
11     int stencilradius = 2;
12     dcp1->mapStencil<update<2>>>(*dcp2, stencilradius, 0);}

```

Listing 2: Exemplary functor for the `mapStencil` skeleton.

## 4.2 Implementation of the MapStencil Skeleton

Supplementing the previous chapter implementation details of the `mapStencil` Skeleton will be described to facilitate the reverse engineering of stencil operations for other projects. The description is twofold firstly it discusses the `PLCube` class which allows access to neighbour elements in the user function and secondly the kernel invocation.

Adding the `MapStencil` skeleton to the existing distributed cube (DC) class requires adding two additional member fields: a vector of `PLCubes` and a (maximal) supported stencil radius. As previously mentioned, the `PLCubes` class allows the end-user to abstract from the indexing of the data structure. To make access to the different memory spaces efficient, each computational unit has a separate `PLCube` storing merely the elements needed to calculate the assigned elements. This design choice makes the class flexible to be used for CPUs and for GPUs. It contains the following attributes to provide a light, minimal design:

- `int width, height, depth` - the three dimensions of the data structure,
- `int stencilRadius` radius of the stencil required to calculate the overlapping elements,

<sup>1</sup> A variant of the skeleton which immediately overwrites old values by new values is possible. However, it restricts the order in which computations can take place resulting in wave front parallelism.

<sup>2</sup> Other stencil shapes such as rectangular or irregular stencils can be handled by using the smallest surrounding cube, this may introduce overhead.

- `int neutralValue` used when the index is outside of the data structure,
- `T* data`, `T* topPadding`, `T* bottomPadding` CPU or GPU pointer for current data,
- four integers to save global indexes for the start and end of main and padding data areas.

Most importantly, the `getter`-function is implemented, taking three integers as arguments and returning the suitable value. This is either the neutral value or the corresponding element from the CPU or GPU memory. Assuming GPUs are used as accelerators, the skeleton updates the current data structure in case the data is not up to date (Listing 3, l.6). Afterwards, it synchronizes the `PLCubes` inside one node, and the data between multiple nodes (l.7, l.9). Foreach GPU used, the `mapStencilKernelDC` kernel is called executing the functor on the appropriate part of the overall data structure. In any other case (multiple nodes and CPU), it is only necessary to synchronize the nodes (l. 21) and call the functor with the corresponding arguments (l. 29).

```

1  template<typename T>
2  template<msl::DCMapStencilFunctor<T> f>
3  void msl::DC<T>::mapStencil(msl::DC<T> &result,
4                             size_t stencilSize, T neutralValue) {
5      #ifdef __CUDACC__
6          this->updateDevice();
7          syncPLCubes(stencilSize, neutralValue);
8          msl::syncStreams();
9          syncPLCubesMPI(stencilSize);
10         for (int i = 0; i < this->ng; i++) {
11             cudaSetDevice(i);
12             dim3 dimBlock(muesli::threads_per_block);
13             dim3 dimGrid((this->plans[i].size + dimBlock.x - 1)
14                           / dimBlock.x);
15             detail::mapStencilKernelDC<T, f><<<dimGrid, dimBlock, 0,
16             muesli::streams[i]>>>(result.plans[i].d_Data,
17             this->plCubes[i], result.plans[i].size);
18         }
19         msl::syncStreams();
20         result.setCpuMemoryInSync(false);
21     #else
22         syncPLCubesMPI(stencilSize);
23     #ifdef _OPENMP
24     #pragma omp parallel for
25     #endif
26     for (int k = 0; k < this->nLocal; k++) {
27         int l = (k + this->firstIndex) / (ncol*nrow);
28         int j = ((k + this->firstIndex) - l*(ncol*nrow)) / ncol;
29         int i = (k + this->firstIndex) % ncol;
30         result.localPartition[k] = f(this->plCubes[0], i, j, l);
31     }
32     #endif
33 }
```

Listing 3: Implementation of the `mapStencil` skeleton.

### 4.3 Example Applications for Three-Dimensional Stencil Operations

Two examples are used to evaluate our implementation: an implementation of the LBM and a mean blur. The LBM is used for fluid simulations e.g. the distribution of gas. It distinguishes between the collision and the streaming step, which alternate in continuous simulations [39, p. 61ff.]. In the streaming step, gas particles move from one cell to another. The fluid flow caused by the colliding particles is calculated in the collision step. The distribution function  $f_i(x, t)$  calculates for a cell  $x$  and a timestamp  $t$  how many particles move in the next step to neighbor  $i$ . Index 0 corresponds to the cell itself.  $f_i^*$  defines the distribution after the collision of the particles (see formula (2)).  $\Delta t$  is the period to be simulated.

$$f_i(x + c_i \Delta t, t + \Delta t) := f_i^*(x, t) \quad (1)$$

For the collision steps, the Bhatnagar–Gross–Krook-operator is used.  $\tau$  is a constant defining the convergence of the simulation. Thus  $\tau$  influences the viscosity of the gas.

$$f_i^*(x, t) := f_i(x, t) - \frac{\Delta t}{\tau} (f_i(x, t) - f_i^{\text{eq}}(x, t)). \quad (2)$$

The equilibrium state is calculated by

$$f_i^{\text{eq}}(x, t) := w_i \rho \left( 1 + \frac{u \cdot c_i}{c_s^2} + \frac{u \cdot c_i}{2c_s^4} + \frac{u \cdot u}{2c_s^2} \right), \quad (3)$$

where  $w_i$  are the weights of the chosen grid and  $c_i$  is the position of the neighbor cells relative to the main cell. The constant number  $c_s$  is the sound velocity of the model. The mass density  $\rho$  and the puls density  $u$  are defined by

$$\rho(x, t) = \sum_i f_i(x, t), \quad \rho u(x, t) := \sum_i c_i f_i(x, t). \quad (4)$$

For the implementation of the LBM, a D3Q19-Grid was used, D being the number of dimensions and Q the number of neighbors. Both steps (collision and streaming) are combined in one `mapStencil` call. Noteworthy, the implementation has to consider that single cells can be marked as blocked, simulating objects that are barriers to the flow of gas or as distributing constantly gas. Therefore, special cells are marked with `Not a Number` values. To simulate this behavior without requiring additional storage, the handling of the floating point numbers is extended. According to the IEEE-754 Standard, each floating point number that has a maximal exponent with a mantissa that is not equal to zero is considered `Not a Number`. The most significant bit of the mantissa of  $f_0$  is set so that the number is definitely understood as NaN. The remaining bits of the mantissa can then be used freely to store other data. In the code, bit masks and a struct with bit-fields are defined in the code to access this information as easily as possible (Listing 4).



```

1  const int FLAG_OBSTACLE = 1 << 0;
2  const int FLAG_KEEP_VELOCITY = 1 << 1;
3  typedef struct {
4      unsigned int mantissa : 23;
5      unsigned int exponent : 8;
6      unsigned int sign : 1;
7  } floatparts ;

```

Listing 4: Handling of barriers and streaming cells.

The data stored for each cell is an `array<float, Q>`. `Q` is a constant number for the neighbor cells and the cell itself. This type is abbreviated in the following listing with `cell_t`. Moreover, it is abstracted from the three-dimensional vector operations (l. 28, 29, 31, 33). The user function starts by transforming the current value of the cell into the single float parts (l. 4). In case it is a cell that distributes gas (`FLAG_KEEP_VELOCITY`), the cell remains without changes (l. 5–7). In any other case, for all neighbor cells, the current amount of particles is read (l. 10–12). In the collision step, all cells which are obstacles reverse the airflow (l. 16–22). All other cells calculate the particles streaming from the next cells (l. 27–34). Noteworthy, the function contains multiple conditional statements impeding the parallelism on GPUs as the threads execution diverges.

```

1  MSL_USERFUNC cell_t update(const PLCube<cell_t> &plCube,
2                               int x, int y, int z) {
3      cell_t cell = plCube(x, y, z);
4      auto* parts = (floatparts*) &cell[0];
5      if (parts->exponent == 255 && parts->mantissa
6          & FLAG_KEEP_VELOCITY) {
7          return cell;
8      }
9      // Streaming.
10     for (int i = 1; i < Q; i++) {
11         cell[i] = plCube(x + (int) offsets[i].x,
12                         y + (int) offsets[i].y, z + (int) offsets[i].z)[i];
13     }
14     // Collision.
15     if (parts->exponent == 255 && parts->mantissa
16         & FLAG_OBSTACLE) {
17         if (parts->mantissa & FLAG_OBSTACLE) {
18             cell_t cell2 = cell;
19             for (size_t i = 1; i < Q; i++) {
20                 cell[i] = cell2[opposite[i]];
21             }
22         }
23         return cell;
24     }
25     float p = 0;
26     vec3f vp {0, 0, 0};
27     for (size_t i = 0; i < Q; i++) {
28         p += cell[i];
29         vp += offsets[i] * cellwidth * cell[i];
30     }
31     vec3f v = p == 0 ? vp : vp * (1 / p);
32     for (size_t i = 0; i < Q; i++) {
33         cell[i] = cell[i] + deltaT / tau * (feq(i, p, v)
34             - cell[i]);
35     }
36     return cell;
37 }

```

Listing 5: User function of the LBM.

The second example used is a mean filter commonly used for smoothing images (user function in Listing 2). Aside from images, filters are commonly used to pre-process data to reduce noise. This might also be applied to signal processing and other application contexts. This example application has the advantage that the stencil size (i.e. radius) can be varied, as depending on the context different stencil sizes are reasonable. Moreover, this program does not require if clauses which potentially slow down the program.

## 5 Evaluation

As the purpose of high-level parallel frameworks is to facilitate writing efficient parallel programs, the speedup needs to be measured. For this purpose, the presented exemplary

**Table 2** Overview of used hardware

Identifier	Nodes	Per node		Per computational unit		GPU/CPU-type
		GPUs	Max. CPU-threads	Mem. (GB)	Cores	
Normal	136	–	36	92	18	Skylake (Gold 6140)
zen2	12	–	128	496	64	Zen2 (EPYC 7742)
gpu2080	5	8	32	11	4352	GeForce RTX 2080 Ti
gpuhgx	2	8	32	80	6912	Nvidia A100 SXM

programs, a mean filter and an LBM implementation, are executed on the HPC machine Palma II.<sup>3</sup> Table 2 lists the hardware specification of the partitions used. Those are two GPU-partitions and two CPU-partitions. To provide meaningful results, all parallel programs are executed ten times. Runtimes are measured using MPI\_Wtime.

For testing CPU-parallelization, the *zen2* partition is used, which is equipped with 12 nodes, each with one Zen2 (EPYC 7742) CPU with 64 cores. For running the sequential version, a single Skylake (Gold 6140) CPU starting a single thread is used (*normal*). It is not possible to run sequential programs on the *zen2* partition as all sequential programs have to run on the *normal* partition.

For the GPU-programs, the *gpu2080* and *gpuhgx* partitions are used. The *gpu2080* partition has five nodes, each with 8 GeForce RTX 2080 Ti GPUs. The *gpuhgx* partition is equipped with two nodes, each with 8 A100 SXM GPUs. These partitions, most importantly, vary in the maximum memory and the number of cores. The *gpu2080* partition has more nodes. However, each GPU has 11GB VRAM and 4352 cores allowing less parallelization than more powerful GPUs (such as the A100) can provide. In contrast, the *gpuhgx* partition has 80GB VRAM per GPU, allowing bigger data structures to be processed, and has more cores (6912) to speed up the program. Using different GPUs also contributes to proving the universal applicability of the MapStencil skeleton in Muesli.

Next to a sequential program, the Muesli-programs solving the LBM are compared to a native implementation. The native program is written to the best of our knowledge which can be categorized as having 2–3 years of experience in C/C++ programming, with a focus on optimization.<sup>4</sup>

## 5.1 LBM

The LBM is used to simulate fluid flow in the three-dimensional space. Consequently, it is reasonable to run an experiment that does not only execute the mapStencil skeleton one time but has multiple iterations simulating multiple dispersion steps. 200 iterations were chosen for every experiment to compare run times between different data sizes.

Data sizes were chosen to completely utilize the available storage. For the LBM, each cell requires 76 bytes, as each cell stores 19 32-bit floating point numbers. For the calculation, one data structure to read and one to write is necessary. The largest

<sup>3</sup> <https://confluence.uni-muenster.de/pages/viewpage.action?pageId=27755336>.

<sup>4</sup> For further reference the program code can be viewed <https://github.com/justusdieckmann/ba-native>.

**Table 3** Runtimes (seconds) and speedups for the parallel implementation of the LBM gas simulation for CPU programs on the zen2 partition

Data size	Sequential	1 Node	Speedup	4 Nodes	Speedup	8 Nodes	Speedup
120 <sup>3</sup>	56.00	1.30	42.96	0.49	114.58	0.31	180.16
400 <sup>3</sup>	2243.63	22.45	99.93	11.2	200.35	6.75	332.26
440 <sup>3</sup>	2994.47	30.35	98.66	16.68	179.53	8.36	358.31
520 <sup>3</sup>	4971.15	47.65	104.33	24.57	202.36	13.56	366.53
640 <sup>3</sup>	9261.54	88.53	104.62	44.93	206.11	24.04	385.18
800 <sup>3</sup>	18572.90	173.88	106.81	86.00	215.96	45.59	407.36
960 <sup>3</sup>	34880.80	300.44	<b>116.10</b>	149.23	<b>233.74</b>	77.11	<b>452.37</b>

Highest runtimes are given in bold

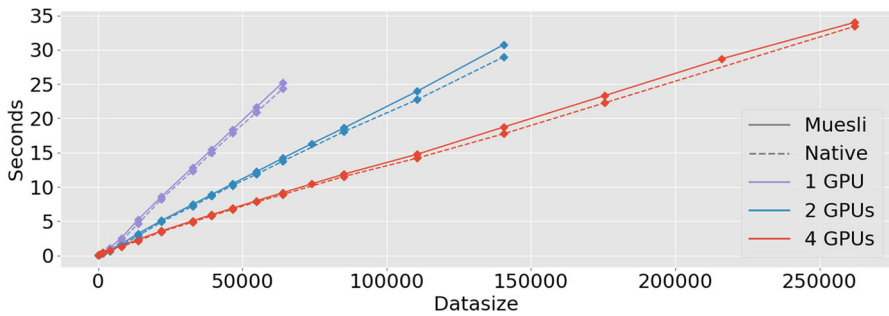
theoretically possible data structure size for a given amount of memory can be simply calculated by:

$$d(\text{gb}) := \sqrt[3]{\text{gb} \cdot \frac{2^{30}}{2 \cdot 76}} \quad (5)$$

This results in a maximum side length of 426 for the RTX 2080 Ti GPU and 826 for the A100 SXM. Although the CPU partition would support bigger data structures, the data size was not increased to the maximum as the speedup converged, and the runtime of the sequential program became unreasonable high (approximately 10h for calculating the LBM simulation for a data size of 960<sup>3</sup>).

As the CPU has 64 cores, this is caused by multithreading. In total, in the optimal case, 128 threads are started on the 64 cores available. In this scenario, it should be considered that the calculations are easy to execute in parallel as all data resides on the memory of the single CPU, accessible for all threads. Using four nodes requires communicating the border values, thus requiring operations that cannot be executed in parallel. Although more elements need to be communicated with increasing data sizes, the share of operations that require communication is decreasing, thus allowing more parallelism. This can be seen as the best speedup that can be achieved with four nodes for the biggest tested data size (960<sup>3</sup>). Eight nodes achieve a speedup of 452. Table 3 shows that for the CPU-zen2 partition, a speedup of 116 can be reached with a single CPU.

In contrast, the GeForce RTX 2080 Ti achieves a speedup of 88 with a single GPU. The maximum speed up which can be achieved by GPUs depends on the GPU used. The GeForce RTX 2080 Ti has 68 streaming multiprocessors (SMs), each capable of executing 64 threads in parallel (4352 in total). Although additional threads may be scheduled, the hardware does not have physical cores to start the threads in parallel. Multiple factors limit the possible speedup. Most importantly, threads with diverging execution branches cannot be executed in parallel, limiting the 64 threads executed in parallel per SM.



**Fig. 2** Runtime comparison of a multi-GPU Muesli program and native (CUDA) implementation of the LBM on a single-node of the gpu2080 partition

The A100 has 108 SMs, allowing more threads to be executed in parallel. The maximum speedup achieved is roughly 353.

In order to check whether a low-level program is significantly faster, a native implementation was programmed to be compared against the Muesli program. As can be seen in Fig. 2, the implementations are close to each other. In contrast to the native implementation, Muesli has a slight overhead. However, as it is very small, the differences are expected and neglectable. Runtimes for bigger data structures are not included for one GPU and two GPUs to increase the readability of the graph. Also worth mentioning is that the native implementation has 544 lines of code without implementing MPI inter-node communication. In contrast, the Muesli program has 246 lines of code and can be run on multiple nodes. Moreover, it should be considered that writing native code is arduous as each line requires fine-tuning.

Regarding the scalability on multiple GPUs, an extract of the achieved speedups can be seen in Table 4. A speedup of 1.7 compared to a single GPU version can be reached for two GPUs, and for four GPUs, a speedup of 2.75 is achieved. To ensure that the communication causes the overhead, the time for the update function was measured separately. Without the communication, a speedup of 1.94 and 3.88 was achieved, which can be attributed to the synchronization of streams. Although the speedup is limited by the communication operations, using multiple GPUs also has the advantage of being able to process bigger data structures, since there is more memory available.

Considering multiple levels of parallelism, the program can also run on multiple nodes equipped with multiple GPUs. Runtimes are depicted in Fig. 3 and Table 5. The columns in the table distinguish between the runtimes including all communication between devices and nodes and runtimes only measuring the calculation on the accelerators. Most eye-catching in the figure, the runtimes for four nodes and four GPUs are not linear but show a switching pattern. This is caused by not splitting the data structure into complete slices but into incomplete slices (e.g., 640 has 40 slices per GPU while 680 has 42.5). The program can handle this. However, the communicational effort rises.

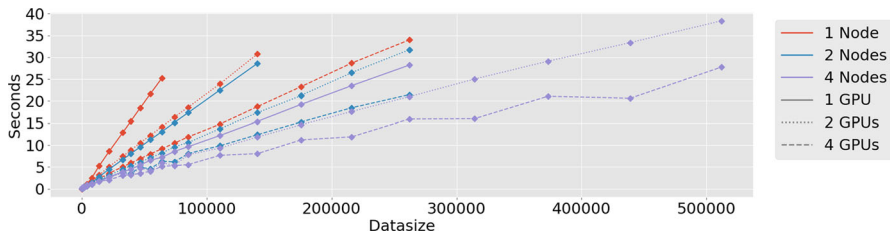
**Table 4** Speedup for the parallel implementation of the LBM gas simulation on the gpu2080 partition

Data size	Sequential	1 GPU	Speedup	2 GPUs	Speedup	4 GPUs	Speedup	8 GPUs	Speedup
400 <sup>3</sup>	2243.63	25.23	88.93	14.20	158.06	9.16	244.92	9.06	247.51
520 <sup>3</sup>	4971.15	–	–	30.76	161.62	18.74	265.32	16.65	298.53
640 <sup>3</sup>	9261.54	–	–	–	–	33.99	272.48	28.78	321.75
800 <sup>3</sup>	18572.9	–	–	–	–	–	–	50.68	366.45

**Table 5** Speedup for the parallel implementation of the LBM gas simulation for multiple nodes on the gpu2080 partition

Data size	GPUs	1 N	1 N-Com	4 Ns	4 Ns-Com <sup>1</sup>	Speedup	Speedup
280 <sup>3</sup>	1	8.59	8.59	2.74	2.16	3.14	3.97
280 <sup>3</sup>	4	3.58	2.23	2.06	0.39	1.73	5.74
400 <sup>3</sup>	1	25.23	25.23	7.21	6.32	3.50	3.99
400 <sup>3</sup>	4	9.16	6.50	5.14	1.70	1.78	3.82

<sup>1</sup> Runtimes without communication between nodes and GPUs

**Fig. 3** Runtimes of the LBM Muesli program on multiple nodes and multiple GPUs on the gpu2080 partition**Table 6** Runtimes in seconds and speedups of CPU program for the mean filter with stencil radius 2 on the zen2 partition

Data size	Runtime (s)			Speedup		
	Sequential	4 Nodes	8 Nodes	4 Nodes	8 Nodes	
120	86.85	0.64	0.46	136.68	190.30	
280	1135.88	6.82	3.95	166.43	287.84	
400	3333.38	19.91	10.23	167.43	325.97	
560	9188.60	49.91	26.40	184.00	348.08	

## 5.2 Mean Filter

Using the LBM implementation as an exemplary program has multiple downsides. Firstly, one factor that has a major influence on the generalizability of the skeleton stays constant—namely, the stencil radius. This factor influences the number of elements that need to be communicated between the computational nodes. Therefore, it is essential to vary this factor to analyze the performance of the skeleton. Moreover, the implementation of the LBM has multiple conditional statements like branch operations slowing down the possible parallelism. In contrast, the mean blur does not contain any if-branches.

Firstly, the CPU parallelism is discussed. Table 6 lists the speedup for one, four, and eight nodes compared to a sequential program. Noteworthy, the optimal time is not achieved by using 128 threads but by using 64 threads. As the instructions are easily executed in parallel, scheduling threads that are executed when other threads are idle is no longer beneficial. For four nodes, a speedup of 184 can be reached. As can be

**Table 7** Speedup for the parallel implementation of the mean blur for a single node of the gpu2080 partition

Stencil radius	Data size	Runtime 1 GPU	Speedup Seq/1 GPU	Runtime 4 GPU	Speedup 1 / 4 GPUs	4 GPUs -Com <sup>1</sup>	Speedup 1/4-Com
2	120 <sup>3</sup>	0.16	542.83	0.07	2.16	0.05	3.51
	400 <sup>3</sup>	4.46	<b>747.39</b>	1.44	3.09	1.15	3.89
	800 <sup>3</sup>	43.65	–	12.14	3.60	10.95	3.99
10	120 <sup>3</sup>	8.24	560.50	2.64	3.12	2.50	3.29
	280 <sup>3</sup>	131.51	508.92	35.98	3.66	35.20	3.74
	800 <sup>3</sup>	3953.38	–	1022.75	3.87	1016.71	3.89

<sup>1</sup> Runtimes in seconds without communication between nodes and GPUs

Highest runtimes are given in bold

seen, for rising data sizes, the speedup improves as fewer communication operations are required. The same applies to programs using eight nodes reaching a speedup of 348.

Secondly, the speedup for the GeForce RTX 2080 Ti GPUs is measured. The program has fewer conditional statements, avoiding branch divergence. The speedup for a single GeForce RTX 2080 Ti GPUs for a data size of 400<sup>3</sup> is 747, significantly better than for the LBM implementation. Tables 7 and 8 list the runtimes and speedups for a small stencil radius of two (reading 125 elements per calculation) for one and two nodes with each one or four GPUs. Including communication operations, a speedup of 3.6 is reached. To ensure that this is caused by the communication operations, the runtime spent on calculation is measured separately. The speedup depicted in the last column reaches 3.99, which is close to an optimum of 4. As communication operations require synchronization, the optimal speedup is hardly achievable. In contrast, scaling across nodes improves the speedup for a single GPU from 747 to 1253. Comparing the runtimes without communication, 43.65 s are nearly doubled from 24.88. Scaling from one node to two nodes is more efficient, as merely one overlapping data region needs to be communicated. The speedup for four GPUs behaves similarly to the above-explained behavior.

Regarding bigger stencil radiuses, the runtimes for a stencil radius of 10 are listed. This requires reading 9261 elements per calculation. This extreme example is chosen to observe the runtime and speedup when not all elements can be loaded in caches.

Besides changing the hardware, the stencil radius was adjusted to discuss the impact on the performance. With an increasing stencil radius, the calculation of one element requires more read and write operations. For a stencil radius of two, the sum of 125 elements is calculated, and self-explanatory, this grows cubic. Figure 4 depicts the influence on the runtime. Although the number of elements processed grows cubic, the runtime does not grow cubic.

Moreover, it ascertains that the speedup improves with a growing share of calculation operations. This is detailed listed in Table 9. Similar to the CPU program, the speedup with and without communication is measured. For bigger stencil radiuses, the speedup comes closer to the optimum. Although more elements need to be commu-



**Table 8** Speedup for the parallel implementation of the mean blur for two nodes of the gpu2080 partition

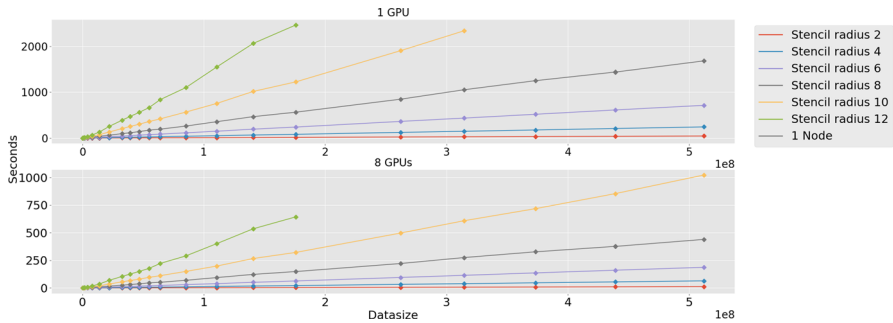
Stencil radius	Data size	Runtime 1 GPU	Speedup Seq/1 GPU	Runtime 4 GPU	Speedup 1 / 4 GPUs	1 GPU -Com <sup>1</sup>	4 GPUs -Com <sup>1</sup>	Speed-up 1/4-Com
2	120 <sup>3</sup>	0.15	579.02	0.08	2.00	0.09	0.03	3.08
	400 <sup>3</sup>	2.66	<b>1253.15</b>	1.12	2.38	2.31	0.65	3.53
	800 <sup>3</sup>	24.88	–	7.41	3.36	22.09	5.81	3.80
10	120 <sup>3</sup>	5.48	842.20	1.87	2.94	4.02	1.50	2.69
	280 <sup>3</sup>	75.39	887.78	20.41	3.69	65.68	18.68	3.52
	800 <sup>3</sup>	2149.16	–	541.64	3.97	2009.69	510.79	3.93

<sup>1</sup>Runtimes without communication between nodes and GPUs  
Highest runtimes are given in bold

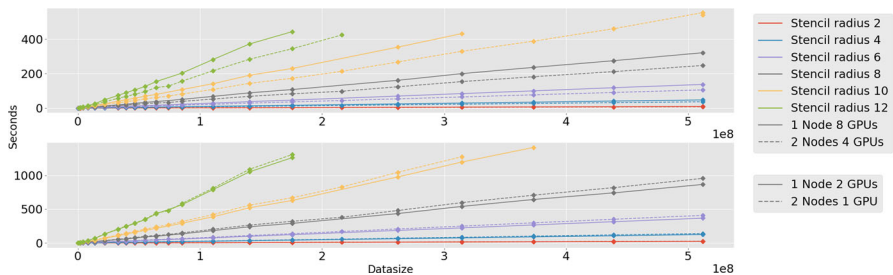
**Table 9** Speedup for the parallel implementation of the mean blur for multiple GPUs on the gpu2080 partition

Stencil radius	Data size	1 GPU	4 GPUs	4 GPUs -Com <sup>1</sup>	Speedup	Speedup	8 GPUs	8 GPUs -Com <sup>1</sup>	Speedup	Speedup
8	120 <sup>3</sup>	4.54	1.47	1.35	3.09	3.35	1.04	0.79	4.37	5.74
	280 <sup>3</sup>	60.58	16.44	15.85	3.68	3.82	9.71	8.47	6.24	7.15
	400 <sup>3</sup>	192.61	51.59	50.34	3.73	3.83	29.52	26.96	6.52	7.14
	560 <sup>3</sup>	563.20	148.68	146.41	3.79	3.85	83.38	78.58	6.75	7.17
12	120 <sup>3</sup>	13.59	4.20	4.03	3.23	3.37	2.53	2.17	5.38	6.27
	280 <sup>3</sup>	252.54	69.24	68.32	3.65	3.70	37.92	35.97	6.66	7.02
	400 <sup>3</sup>	836.59	222.12	220.38	3.77	3.80	118.89	115.15	7.04	7.27
	560 <sup>3</sup>	2464.76	643.33	640.16	3.83	3.85	338.46	331.65	7.28	7.43

<sup>1</sup>Runtimes without communication between nodes and GPUs



**Fig. 4** Runtimes of the Blur Muesli program on the gpu2080 partition



**Fig. 5** Runtimes of the Blur Muesli program on the gpu2080 partitions with similar hardware

nicated between the nodes, the intensity of the calculation requires more time which dominates the total runtime. For a stencil radius of 12 (processing 15.625 elements per thread). The operations are still very performant as elements are automatically written in GPU caches, allowing efficient data access. As multiple combinations of hardware settings are tested, it is interesting to compare having the same number of GPUs distributed on a different number of nodes. For example, having one node with two GPUs, in contrast to having two nodes with one GPU, has the downside of having one less CPU. However, it has the advantage of allowing GPU to GPU communication. Two comparisons are displayed in Fig. 5. The first figure compares a one-node eight GPUs program to a two-node four GPUs program for different stencil radii. As can be seen, the two-node program is faster. This is caused by parallelizing communication as some operations are executed by the CPU instead of communicating between only GPUs. In contrast, the second figure shows that when a single node program with two GPUs is compared to a two nodes program with each one GPU. The single node program is slightly faster as a communication operation between GPUs is faster than an MPI communication between two nodes.

The program was also tested on the A100 partition allowing even bigger data sizes to be processed by a single GPU. In contrast to a single GeForce RTX 2080 Ti GPUs (Speedup 747), the A100 has a speedup of 1122.34 for a data size of  $400^3$  elements, and 3680.54 for four GPU (2309.08). According to the previous approach, the runtime was measured with and without communication (Table 10). Even with the communication, the speedup is close to the optimum. For four GPU, a speedup of 3.65 and 3.93 can

Table 10 Runtimes in seconds and speedups for a mean filter on the gpubgx partition

Stencil radius	Data size	1 GPU	4 GPUs	Speedup	8 GPUs	Speedup	4 GPUs -Com <sup>1</sup>	Speedup	8 GPUs -Com <sup>1</sup>	Speedup
2	400 <sup>3</sup>	2.97	0.91	3.28	1.14	2.61	0.75	3.95	0.38	7.77
	800 <sup>3</sup>	24.00	6.57	<b>3.65</b>	4.64	<b>5.18</b>	6.02	3.99	3.03	<b>7.92</b>
8	400 <sup>3</sup>	103.60	27.16	3.81	15.19	6.82	26.56	3.90	13.35	7.76
	800 <sup>3</sup>	1016.62	258.47	<b>3.93</b>	134.69	<b>7.55</b>	256.47	3.96	129.55	<b>7.85</b>

<sup>1</sup>Runtime without communication between nodes and GPUs  
Highest runtimes are given in bold

**Table 11** Comparison of Palabos, lbmpy and Muesli on the zen2 partition for CPU programs and on the 2080 partition for GPU programs. All GPU programs were run on a single GPU

Data size	CPU Muesli	Palabos	Speedup	GPU Muesli	lbmpy	Speedup
40 <sup>3</sup>	0.48	0.33	0.68	0.02	0.13	5.91
80 <sup>3</sup>	0.58	1.36	2.34	0.14	0.62	4.54
120 <sup>3</sup>	1.3	3.32	2.55	0.44	1.72	3.94
160 <sup>3</sup>	2.02	7.11	3.52	1.14	3.55	3.12
200 <sup>3</sup>	3.26	12.93	3.97	2.52	6.52	2.59
240 <sup>3</sup>	5.02	21.13	<b>4.2</b>	5.23	10.93	2.09
280 <sup>3</sup>	8.1	32.36	4	8.59	16.01	1.86

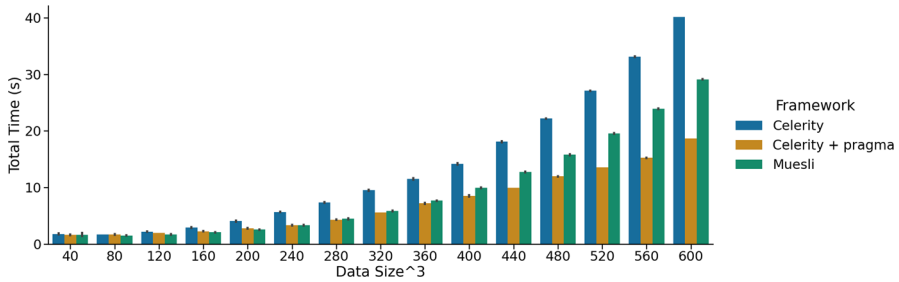
Highest runtimes are given in bold

be reached for 400<sup>3</sup> and 800<sup>3</sup> elements. For eight GPU, a speedup of 5.18 and 7.55 is reached.

### 5.3 Framework Comparison

Ongoing work on stencil calculations claims to exploit the parallelism on the used hardware. Therefore, it is essential to compare our implementation to other frameworks listed in the related work section. Firstly, we chose to compare our work to Palabos. Additionally, lbmpy was included in the comparison on a single GPU. In Table 11 it can be seen that Muesli is up to 4.2 times faster than the recent CPU version of Palabos. The GPU programs were compared on a single GPU of the gpu2080 partition. Muesli always outperforms lbmpy however, the speedup decreases. Noteworthy, it was impossible to test bigger data sizes on the partition as lbmpy consumes significantly more memory. For the lbmpy library 280<sup>3</sup> was the maximum size possible while the Muesli program could run cubes of size 400<sup>3</sup>.

As those frameworks can not scale over multiple GPUs the multi-GPU program was compared to an implementation using the Celerity runtime. This comparison is influenced by the compiler as Muesli uses the NVCC compiler which sometimes generates superior code compared to the Clang-based compiler of hipSYCL, which we used as the SYCL implementation for building the Celerity code. During the analysis of the program, it was discovered that the pure Celerity program performed slower than Muesli (Fig. 6) as the SYCL compiler did not unroll some for loops. Manually inserting a `#pragma unroll` inside the user function of the gas simulation results in a better performance of Celerity. As those results are mixed a real comparison would require more examples and a variety of hardware compositions to compare the efficiency of the communication operations.



**Fig. 6** Runtime comparison of a multi-GPU Muesli program and a multi-GPU celerity program of the LBM on the gpubgx partition

## 6 Conclusion

We have presented the implementation and experimental evaluation of a three-dimensional MapStencil skeleton. The implementation was tested with two example programs: (1) an LBM implementation and (2) a mean filter. Both examples were tested in complex hardware environments equipped with multiple nodes, CPU cores, and accelerators.

For the LBM implementation, a speedup of 116 for one node and 452 for four nodes can be reached, confirming that for complex functions, the communication between nodes has only a minor influence on the speedup. In contrast, using multiple GPUs for complex functions does not provide the expected speedup. For a single GPU, merely a speedup of 88 can be reached. This speedup scales for two GPUs (161), but as the communication rises, the speedup does not scale according to the number of accelerators. This is caused by the complex instruction flow as the conditional statements cause thread divergence. Abstracting from the communication, the program scales across multiple nodes.

In contrast, running the mean filter example shows that non-diverging programs do not benefit from multithreading with CPUs, having a speedup of 180 for four nodes. In contrast, a GPU can reach a speedup of 747, benefiting from warps of threads that can run the same instruction in parallel. The communication between GPUs decreases the speedup from 7.7 to 5.4, taking approximately 30% of the runtime. When increasing the stencil radius, the speedup improves as the larger radius improved the GPU utilization, amortizing GPU threads' latency. This finding is confirmed for two different types of GPUs an A100 partition and a GeForce RTX 2080 Ti partition. Moreover, it is confirmed that inter-GPU communication is faster than inter-node communication.

Overall, it is shown that stencil operations are particularly relevant for inclusion in high-level frameworks, as they are commonly used across domains and the implementation is not feasible for inexperienced programmers in a reasonable amount of time, also due to the complex communication operations. The speedup achieved proves that high-level frameworks can provide the means to produce parallel stencil programs without knowledge of parallel programming.

**Acknowledgements** The cluster Palma II enabled the runtime experiments.

**Author Contributions** (1) Nina Herrmann, (2) Justus Dieckmann, (3) Herbert Kuchen. 1 wrote most of the manuscript text. 2 and 3 wrote part of the manuscript text. Figures were created by 1. The Muesli implementation was created by 1 and 2, the native implementation was created by 2. The runtime experiment was set up and executed by 1. All authors reviewed the manuscript.

**Funding** Open Access funding enabled and organized by Projekt DEAL. This project is not funded.

**Data Availability** Muesli and the low-level implementation are available at Github (<https://github.com/NinaHerrmann/muesli4>) (<https://github.com/justusdieckmann/ba-native>). Complete runtime datasets are not published.

## Declarations

**Conflict of interest** The authors declare that they have no Conflict of interest.

**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. MPI Standard: <https://www.mpi-forum.org/docs/>. Accessed 24 Feb 2023
2. The OpenMP API specification for parallel programming. <https://www.openmp.org/>. Accessed 24 Feb 2023
3. NVIDIA: CUDA: <https://developer.nvidia.com/cuda-zone>. Accessed 24 Feb 2023
4. Cole, M.I.: Algorithmic skeletons: structured management of parallel computation. Computer science thesis. Pitman, London (1989)
5. Ernsting, S., Kuchen, H.: Data parallel algorithmic skeletons with accelerator support. *Int. J. Parallel Prog.* **45**(2), 283–299 (2017)
6. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eskel. In: Cunha, J.C., Medeiros, P.D. (eds.) *Euro-Par 2005 Parallel Processing*, pp. 761–770. Springer, Berlin, Heidelberg (2005)
7. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: Skepu 3: portable high-level programming of heterogeneous systems and hpc clusters. *Int. J. Parallel Prog.* **49**(6), 846–866 (2021)
8. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: Pillana, S., Xhafa, F. (eds.) *Programming multi-core and many-core computing systems, parallel and distributed computing*, pp. 261–280. Wiley, London (2017)
9. 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)
10. Thoman, P., Tischler, F., Salzmann, P., Fahringer, T.: The celerity high-level api: C++ 20 for accelerator clusters. *Int. J. Parallel Prog.* **50**(3–4), 341–359 (2022)
11. Goli, M., González-Vélez, H.: Heterogeneous algorithmic skeletons for fast flow with seamless coordination over hybrid architectures. In: *2013 21st euromicro international conference on parallel, distributed, and network-based processing*, pp. 148–156 (2013)
12. Hagedorn, B., Stoltzfus, L., Steuer, M., Gorchach, S., Dubach, C.: High performance stencil code generation with lift. In: *Proceedings of the 2018 international symposium on code generation and optimization*. CGO 2018, pp. 100–112. Association for Computing Machinery, New York, NY (2018)
13. Steuer, M., Gorchach, S.: Skelcl: a high-level extension of opencl for multi-gpu systems. *J. Supercomput.* **69**(1), 25–33 (2014)

14. Bauer, M., Eibl, S., Godenschwager, C., Kohl, N., Kuron, M., Rettinger, C., Schornbaum, F., Schwarzmeier, C., Thönnies, D., Köstler, H., Rüde, U.: walberla: A block-structured high-performance framework for multiphysics simulations. *Comput. Math. Appl.* **81**, 478–501 (2021)
15. Castro, M., Santamaria-Valenzuela, I., Torres, Y., Gonzalez-Escribano, A., Llanos, D.R.: Epsilod: efficient parallel skeleton for generic iterative stencil computations in distributed gpus. *J. Supercomput.* **79**(9), 9409–9442 (2023)
16. Kuckuk, S., Köstler, H.: Whole program generation of massively parallel shallow water equation solvers. In: 2018 IEEE international conference on cluster computing (CLUSTER), pp. 78–87 (2018)
17. Bastian, P., Blatt, M., Dedner, A., Dreier, N.-A., Engwer, C., Fritze, R., Gräser, C., Grüninger, C., Kempf, D., Klöfkom, R., Ohlberger, M., Sander, O.: The dune framework: basic concepts and recent developments. *Comput. Math. Appl.* **81**, 75–112 (2021)
18. Lutz, T., Fensch, C., Cole, M.: PARTANS: an autotuning framework for stencil computation on multi-GPU systems. *ACM Trans. Archit. Code Optim.* **9**(4), 1–24 (2013)
19. Pereira, A.D., Ramos, L., Góes, L.F.W.: Pskel: a stencil programming framework for cpu-gpu systems. *Concurr. Comput. Pract. Exp.* **27**(17), 4938–4953 (2015)
20. Pereira, A.D., Castro, M., Dantas, M.A., Rocha, R.C., Góes, L.F.: Extending openacc for efficient stencil code generation and execution by skeleton frameworks. In: 2017 international conference on high performance computing & simulation (HPCS), pp. 719–726. IEEE (2017)
21. Latt, J., Malaspinas, O., Kontaxakis, D., Parmigiani, A., Lagrava, D., Brogi, F., Belgacem, M.B., Thorimbert, Y., Leclaire, S., Li, S., Marson, F., Lemus, J., Kotsalos, C., Conradin, R., Coreixas, C., Petkantchin, R., Raynaud, F., Beny, J., Chopard, B.: Palabos: parallel lattice Boltzmann solver. *Comput. Math. Appl.* **81**, 334–350 (2021)
22. Gonzales, R., Gryazin, Y., Lee, Y.T.: Parallel fft algorithms for high-order approximations on three-dimensional compact stencils. *Parallel Comput.* **103**, 102757 (2021)
23. Skepu: <https://github.com/skepu/skepu/>. Accessed 13 Feb 2024
24. Celerity: <https://github.com/celerity/celerity-runtime>. Accessed 13 Feb 2024
25. Lift: <https://github.com/lift-project/lift/tree/master>. Accessed 13 Feb 2024
26. SkelCL: <https://github.com/skelcl/skelcl>. Accessed 13 Feb 2024
27. Bauer, M., Köstler, H., Rüde, U.: lbmpy: automatic code generation for efficient parallel lattice boltzmann methods. *J. Comput. Sci.* **49**, 101269 (2021)
28. WaLBerla: <https://i10git.cs.fau.de/walberla/walberla>. Accessed 13 Feb 2024
29. Lbmpy: <https://i10git.cs.fau.de/pycodegen/lbmpy>. Accessed 13 Feb 2024
30. EPSILOD: [https://gitlab.com/trasgo-group-valladolid/controllers/-/tree/epsilod\\_JoS22](https://gitlab.com/trasgo-group-valladolid/controllers/-/tree/epsilod_JoS22). Accessed 13 Feb 2024
31. ExaStencil: <https://github.com/lssfau/ExaStencils>. Accessed 13 Feb 2024
32. DUNE: <https://gitlab.dune-project.org/core>. Accessed 13 Feb 2024
33. PSkel: <https://github.com/pskel/pskel>. Accessed 13 Feb 2024
34. Palabos: <https://gitlab.com/unigesp/palabos>. Accessed 13 Feb 2024
35. Kotsalos, C., Latt, J., Chopard, B.: Palabos-npfem: software for the simulation of cellular blood flow (digital blood). arXiv preprint [arXiv:2011.04332](https://arxiv.org/abs/2011.04332) (2020)
36. Marques, R., Paulino, H., Alexandre, F., Medeiros, P.D.: Algorithmic skeleton framework for the orchestration of gpu computations. In: Wolf, F., Mohr, B., Mey, D. (eds.) Euro-Par 2013 parallel processing, pp. 874–885. Springer, Berlin, Heidelberg (2013)
37. Alba, E., Luque, G., Garcia-Nieto, J., Ordonez, G., Leguizamon, G.: Mallba: a software library to design efficient optimisation algorithms. *Int. J. Innovative Comput. Appl.* **1**(1), 74–85 (2007)
38. Karasawa, Y., Iwasaki, H.: A parallel skeleton library for multi-core clusters. In: 2009 international conference on parallel processing, pp. 84–91 (2009)
39. Krüger, T., Kusumaatmaja, H., Kuzmin, A., Shardt, O., Silva, G., Viggien, E.M.: The lattice Boltzmann method: Principles and Practice. Graduate texts in physics. Springer, Cham (2016)