The Algorithmic Skeleton Library Revisited.

The Münster Skeleton Library

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Wirtschaftswissenschaften
durch die Wirtschaftswissenschaftliche Fakultät
der Universität Münster

vorgelegt von
Nina Herrmann

Münster, 2024

Dekan	Prof. Dr. Thomas Langer
Berichterstatter	Prof. Dr. Herbert Kuchen Prof. Dr. Sergei Gorlatch
Datum der Disputation	22. Juli 2024

Abstract

This dissertation explores the development and enhancement of high-level approaches for parallel programming, specifically focusing on the algorithmic skeleton library, Muesli. Motivated by the evolving landscape of hardware and the increasing complexity of data processing, this research aims to make parallel programming more accessible and efficient for programmers with limited knowledge in this field.

The dissertation begins by addressing the need for parallel programming, driven by the ever-growing complexity of algorithms and the increasing amount of data available. It highlights the significance of high-level frameworks that abstract the intricacies of parallel programming, making it easier for developers to write efficient parallel code without deep knowledge of underlying hardware specifics. The research objective is to revise and evaluate Muesli, ensuring it meets the demands of modern heterogeneous computing environments. This involves adapting the framework to support the latest hardware advancements and validating the applicability of existing skeletons for new algorithms.

The applicability is demonstrated by revisiting the currently supported hardware focusing on the hybrid execution of program code on CPUs and GPUs when distributed over multiple nodes. Thereafter, the usefulness for implementing real-world application is proven by firstly implementing the Ant Colony Optimisation algorithm for the Travelling Salesman Problem. Additionally, the generic implementation of a skeleton for stencil operations is discussed in detail showcasing multiple examples, e.g. an implementation of the lattice Boltzmann methods. The thesis concludes with a summary of findings and the academic publications that underpin this research.

This work contributes to the field by providing a robust high-level framework that simplifies parallel programming, adapting to new hardware, and validating its efficacy through practical applications.

Correctness may be theoretical but incorrectness has practical impact.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ACO	Ant Colony Optimization
AMD	Advanced Micro Devices, Inc.
API	Application Programming Interface
BPP	Bin Packing Problem
CPU	Central Processing Unit
CSP	Cutting Stock Problem
DA	Distributed array
DC	Distributed cube
DM	Distributed matrix
DS	Distributed data structure
DSL	Domain Specific Language
DSRM	Design science research methodology
FPGA	Field Programmable Gate Array
FSS	Fish School Search
GPGPU	General-purpose computing on graphics processing units
GPU	Graphics Processing Unit
HPC	High Performance Computing
IBM	International Business Machines Corporation
LBM	Lattice Boltzmann method
LOC	Lines of code
MKP	Multidimensional Knapsack Problem
MPI	Message Passing Interface
Muesli	Münster Skeleton Library
Musket	Muenster Skeleton Tool for High-Performance Code Generation

OpenACC Open Accelerators
OpenCL Open Compute Language
OpenMP Open Multi-Processing

PDE Partial differential equation
PLCube Padded local cube
PLMatrix Padded local matrix

SIMD Single Instuction Multiple Data
SM Streaming multiprocessor

TSP Traveling Salesperson Problem

Part I

Research Overview

1

Exposition

This chapter sets the scene for this dissertation. Section 1.1 motivates the continued development of a high-level library for parallel programming. Subsequently, Section 1.2 specifies the problems challenged by this research and formulates the research objective. Finally, the research design and the outline of this thesis are elaborated in Section 1.3.

1.1 Motivation

Gordon Moore observed that the number of transistors in an integrated circuit doubles approximately every two years [62]. This observation became known as Moore's Law. Due to the physical limits, this law will inevitably come to an end at some point. Instead, software, algorithms, and hardware architectures (e.g. graphene chips) will need to be developed to cope with the growing complexity of algorithms and the increasing amount of data that needs to be processed [51]. While it is not clear whether Moore's Law has already ended, one of the most significant adjustments that can be made immediately is to make efficient use of available hardware by writing efficient parallel programs. Applications that already require parallel programming are, for example, complex image processing algorithms, such as processing satellite or telescope data, algorithms in machine learning, or genome sequencing.

The first approaches to parallel programming were, for example, the implementation of the POSIX standard [44] in 1988 for managing multiple threads in C and C++ [33] and the release of High-Performance Fortran in 1993 [82]. Especially, the rise of General-purpose computing on graphics processing units (GPGPU) increased the interest in creating parallel programs. As interest in parallel programming grew, approaches became more diverse, and alongside efficiency, usability and abstraction became subjects of research [7], [15], [32]. Abstract frameworks enable more programmers to easily write parallel programs without thinking about the details of parallel programming, such as transferring data to distinct memory spaces, managing threads, and thinking about race conditions and deadlocks. One famous approach to high-level parallel programming is algorithmic skeletons. Those were introduced in 1991 by Cole [12]. Algorithmic skeletons

are common programming patterns to abstract from parallel implementation details (e.g., the `map` operation).

Two factors continue to drive the development of high-level parallel programming frameworks. First, developments and innovations by hardware vendors require the continuous adoption of high-level frameworks. Additionally, updates and adaptations to the latest developments in low-level parallel programming languages need to be incorporated. Secondly, new application domains require a rethink of the set of skeletons provided by a skeleton framework. Both aspects reinforce the importance of revisiting algorithmic skeleton frameworks. The Münster Skeleton Library (Muesli) was first presented in 2002 [48], and since then, it has been continuously adapted (e.g., [21], [71]). It serves as a prototype to explore the abstraction layer high-level frameworks can provide to parallel programming and will be enhanced in this thesis.

1.2 Research Objective

The drivers for high-level frameworks in parallel programming are two-fold. The first driver is the continuous need to adjust high-level frameworks to the hardware available on high-performance clusters. Over the previous years, GPU support has become essential for speeding up computations. Modern clusters provide multiple GPUs that are distributed over multiple nodes. Multiple high-level frameworks already implement the support for multiple GPUs distributed over multiple nodes [21] [24] [80]. However, as hardware is constantly changing, it has to be reevaluated if high-level frameworks provide the means to use the hardware efficiently.

The second factor that requires revisiting high-level frameworks is the validation of suitable skeletons for new algorithms that are implemented by users. It has to be evaluated if the current set of skeletons is sufficient to let programmers efficiently implement a variety of algorithms. Therefore, examples that have not yet been implemented with algorithmic skeletons have to be tested for adaptability. Summarising both drivers, the research objective of this dissertation is stated as:

Research objective:

> *The revision and evaluation of an algorithmic skeleton library that provides the means for programmers with little knowledge in parallel programming to implement efficient parallel programs for heterogeneous computing environments.*

The objective is realised by revisiting the artefact (Muesli). Of course, with the emergence of new hardware architectures and algorithms, the findings need to be reevaluated. The work on the artefact yields additional contributions. Examples include the comparison to other high-level frameworks and the optimisation of a parallel implementation of a meta-heuristic.

1.3 Outline and Research Methodology

This research is systematically structured by the design science research methodology (DSRM) to fulfil the research objective. DSRM is chosen out of different methodologies as design science "creates and evaluates IT artefacts intended to solve identified organisational problems." [42, p.77]. Hereby, Muesli is the artefact that is improved and evaluated. The methodology consists of six steps that are shortly explained:

1. *Identify problem and motivate.* The problem to be solved has to be stated and motivated. For the presented work, the problem was already stated, namely, the efficient processing of large data sets with complex algorithms. While the introduction has motivated the overall need for high-level frameworks for parallel programming, the single chapters also motivate the specific aspects chosen.

2. *Define objectives of a solution.* As not all aspects of the problem might be solvable in a single artefact, the objectives of a realisable artefact are defined. The objective was presented in Section 1.2.

3. *Design and develop the artefact.* As the artefact already exists, this thesis discusses the design of the adjustments made to the artefact. The starting point is first explained; thereafter, every chapter elaborates on the design decision made.

4. *Demonstrate the artefact.* To demonstrate the artefact, it has to be applied to an identified problem. Our research programs were developed with high-level frameworks that can be executed in parallel.

5. *Evaluate the artefact.* The demonstration of the artefact has to be evaluated to ensure that the problem was solved in an appropriate manner. For our purpose, this requires to measure the runtime and to evaluate the usability of the library.

6. *Communicate the results.* Last, all findings have to be publicly available. This is realised by publishing this dissertation.

The structure of the thesis and the relationship to the single steps are depicted in Figure 1.1. It consists of two parts: Part I summarises all individual publications that can be found in Part II. In Part I, steps 1,2, and 3 of the DSRM were already stated and realised. Therefore, it builds the foundation for the following chapters:

- Chapter 2 presents relevant programming models for high-, and low-level frameworks. It assumes a basic understanding of parallel programming. The chapter builds the foundation to understand the implementation of the artefact Muesli.
- Thereafter, Chapter 3 introduces the artefact by presenting the supported data structures, skeletons, and additional functions. It also demonstrates an example program. It includes functionality that has been implemented within this work. When necessary, details are explained in the corresponding chapters.
- Given the current state of the art, Chapter 4 inspects further option for the hardware support of Muesli. As multiple nodes equipped with multiple GPUs are already supported, the hybrid usage of the CPU is inspected in more detail.
- Afterwards, Chapter 5 analyses the applicability of Muesli for solving the Traveling Salesperson Problem (TSP) problem with the Ant Colony Optimization (ACO) algorithm, therefore laying a focus on the applications that can be realised with Muesli. Meta-heuristics are commonly used to solve complex problems where a good solution is sufficient.
- To further prove the applicability of Muesli, Chapter 6 firstly explains the implementation of a skeleton that can compute stencil operations and secondly evaluates the created skeleton.
- Last but not least, Chapter 7 finally summarizes all findings.

Part II reproduces the academic publications that are the foundation for this thesis and, therefore, implements Step 6 of the DSRM process, namely to communicate the results.

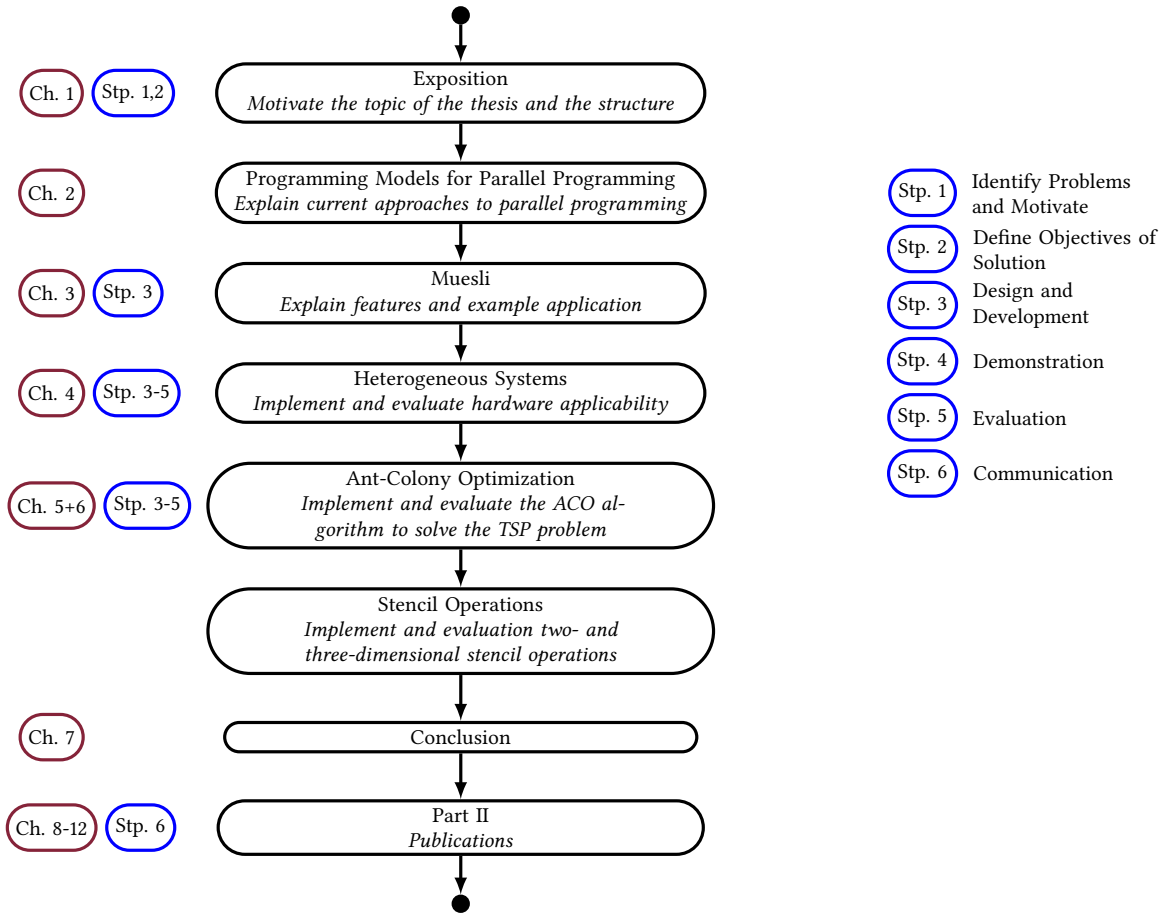


Figure 1.1: Structure of the thesis with references to the chapters and the steps of the DSRM step.

2

Programming Models for Parallel Programming

Even experienced programmers struggle to write parallel programs as, e.g., deadlocks and out-of-memory accesses impede the creation of error-free programs. Moreover, even error-free programs require expertise in parallel algorithms and hardware-specific optimizations to be efficient. Slight changes might have a major impact on the performance. Therefore, multiple approaches originated that aid the creation of parallel code and, therefore, simplify the creation of parallel programs. Approaches can include any tool to abstract from the details of parallel programming, such as frameworks, libraries, or Domain Specific Languages (DSLs). For the sake of simplicity, we will always refer to these as frameworks for high-level approaches and programming models for low-level approaches when we are not talking about the details of a single one. It is assumed that the reader is familiar with computer architectures with multi-core processors and GPUs[1]. This chapter is structured by first presenting a selection of low-level programming models and following up with a presentation of selected high-level frameworks in Section 2.2. In all given examples, keywords that are specific to the one presented are depicted in orange.

2.1 Low-Level Programming Models

The border between a low-level or high-level approach is not straightforward. For instance, CUDA programming code is more abstract than its PTX assembly code representation [2]. However, CUDA code is not high-level for our purposes. In this context, a framework is considered low-level if it either:

1. requires the programmer to manage parallelism and synchronization between individual threads or

[1] An overview of computer architectures can be found in Chapter 2 [8]

[2] `https://docs.nvidia.com/cuda/inline-ptx-assembly/index.html` Accessed 20.05.2024

2. does not relieve the programmer from falling into common traps of parallel programming, such as deadlocks.

With the chosen criteria, high-level frameworks target programmers with little to no expertise in parallel programming. Those could be physicists with basic knowledge of programming. The frameworks presented in this section have been chosen because they have gained popularity in the parallel programming community. Understanding the differences between low-level frameworks is necessary, as this might limit the area of application and features.

The following subsections are structured to first discuss frameworks that target the most elevated level of parallelization, namely connecting multiple nodes in a distributed-memory system (Subsection 2.1.1). The following two subsections present approaches that can be used to generate parallel code for multiple-core and accelerator environments. Next, OpenMP is presented as the most commonly used approach to parallelism on multi-core CPUs (Subsection 2.1.4). Lastly, Subsection 2.1.6 and 2.1.5 present two approaches for generating parallel code on GPUs.

2.1.1 MPI

Message Passing Interface (MPI) is a standard constructed for message-passing in parallel computing environments. It defines a virtual topology and communication and synchronization operations necessary to operate on multiple processes, which can be mapped to nodes, servers, or cores. The first draft was published in 1993, followed by the first release in June 1994 [63]. Since then, multiple versions have been released, adjusting the standard to current needs without introducing big changes.

Amongst others MPICH[3], OpenMPI[4], and Intel MPI[5] implement the standard. Common standards are implemented in C/C++ and Fortran. It is worth mentioning that Python packages written in C/C++ can also use MPI. The performance might vary depending on the implementation [45] and the network used [53]. Moreover, different implementations target different network stacks. Although a choice might provide slight runtime benefits, end-users commonly rely on the software installed on the used cluster[6]. Besides communication between ranks, MPI does not offer any means to write parallel

[3] `https://www.mpich.org/` Accessed 20.02.2024

[4] `https://www.open-mpi.org/` Accessed 20.02.2024

[5] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html` Accessed 20.02.2024

[6] End-users working on local PC, therefore free in their choice of software, rarely have enough nodes to profit from MPI

program code. An example can be seen in Listing 2.1. Lines 7 and 8 extract the overall number of processes (`size`) and the identifier of the current process (`rank`). In case of an error, all processes can be aborted with `MPI_Abort` (l. 12). The `MPI_COMM_WORLD` parameter, used in both statements, specifies the group of ranks that are targeted (called communicator), which in this example includes all processes launched by MPI. Defining custom communicators that only include some processes can speed up communication operations in case operations should be split into groups. The program shows a broadcast operation. One command is sufficient to let the process with the rank 0 send an integer to the other processes and for the other processes to receive the data (l. 17). Note that those operations are synchronous; for example, `MPI_Bcast` returns after all send and receive operations that the local rank participates in have completed. MPI also supports asynchronous operations (e.g. `MPI_Ibcast`). Lastly, `MPI_Finalize` ends the program, freeing all resources allocated by MPI. A complete list of all operations can be found in the specification [27]. This includes multiple communication operations and barriers.

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char *argv[]) {
5    int rank, size, data;
6    MPI_Init(&argc, &argv);
7    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8    MPI_Comm_size(MPI_COMM_WORLD, &size);
9
10   if (size < 2) {
11     fprintf(stderr,"This program requires at least 2 processes\n");
12     MPI_Abort(MPI_COMM_WORLD, 1);}
13   if (rank == 0) {
14     data = 42;
15     printf("Process %d will sent message: %d\n", rank, message);}
16   // Broadcast data from process 0 to all other processes
17   MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
18   if (rank > 0)
19     printf("Process %d received data: %d\n", rank, data);
20
21   MPI_Finalize();
22   return 0;
23 }
```

Listing 2.1: Example for a MPI program with multiple processes executing a broadcast operation.

Listing 2.2 shows an exemplary compiler invocation and execution. The compile command in line 1 is used for the MPICH or the OpenMPI implementation. The argument `-np` defines the number of processes that are started. On HPC cluster, the way `mpirun` is used might vary, as a custom job scheduler (e.g. slurm) might abstract from the command. Before sending data, the process with rank 0 prints line 3; afterwards, the receiving processes can print the received data in arbitrary order as they run in parallel.

```
1 mpicc mpi_bc.cpp -o mpi_bc
2 mpirun -np 5 ./mpi_bc
3 Process 0 will sent message: 42
4 Process 2 received data: 42
5 Process 1 received data: 42
6 Process 4 received data: 42
7 Process 3 received data: 42
```

Listing 2.2: Output of a example for a MPI program with multiple processes.

2.1.2 OpenCL

The open standard Open Compute Language (OpenCL) does not target a specific type of hardware but provides a C++ Application Programming Interface (API) and a DSL to create programs that are executable on CPUs, GPUs, and other types of accelerators. Noteworthy, it also supports Advanced Micro Devices, Inc. (AMD) GPUs. To standardize platforms for parallel execution, the hardware vendor must structure the device in terms of compute units, each with multiple processing elements.

Although OpenCL abstracts from vendor-specific hardware details, it does require the programmer to have knowledge of parallel programming, as the following example shows. OpenCL programs require multiple objects to execute functions in parallel. Firstly, a context needs to be created which might specify a device type (`CPU, GPU, ACCELERATOR`) or use a default type (`CUSTOM, DEFAULT, ALL`) (Listing 2.3 l. 8). Those devices are assigned to a queue that can later execute functions in parallel. A program can have multiple command queues. Noteworthy, the depicted program does not include any error handling. E.g., if the program is executed on a platform without a GPU, it would fail. Next, a data structure is created that is from the class `Buffer`. OpenCL can optimize those data structures depending on the read or write accesses defined as a parameter (in the example, write only). The `Buffer` class always has a fixed size. The most important aspect that clearly defines OpenCL as a low-level framework in our context is that the definition of the parallel operation (ll. 12-15) requires managing threads manually, as the identifier

has to be read. Therefore, a major part of the efficiency of parallelism depends on the programmer's expertise. Programs can also be loaded from external files, usually written with the file ending `.cl`. A `Program` object needs to be built and can then be passed to a `Kernel` object. The naming sounds similar to CUDA functions, which will be explained in Subsection 2.1.5 but can be executed on different types of devices. A `Kernel` object has exactly one function, and parameters can be passed with the function `setArg` (ll. 18+19). Lastly, the actual execution requires to call the function `enqueueTask`. For more complex execution, the function `clEnqueueNDRangeKernel` needs to be used to set data offsets for multiple devices and other parameters. In summary, OpenCL abstracts from vendor-specific aspects but can not abstract from the obstacles of simplifying thread parallelism.

```
1  #include <CL/cl.hpp>
2  #include <iostream>
3
4  int main() {
5    constexpr size_t N = 10;
6    int out[N];
7
8    cl::Context context(CL_DEVICE_TYPE_GPU); // Create OpenCL context and command queue
9    cl::CommandQueue queue(context);
10   cl::Buffer buffOut(context, CL_MEM_WRITE_ONLY, sizeof(out));  // Create buffer for output vector
11   // Load kernel source code, create program, and build it
12   cl::Program program(context,
13     "__kernel void setIndex(__global int* out) {
14       int i = get_global_id(0); out[i] = i;
15     }");
16   program.build();
17
18   cl::Kernel kernel(program, "setIndex", nullptr);  // Create kernel  set argument
19   kernel.setArg(0, buffOut);      // Set arguments for the kernel
20   queue.enqueueTask(kernel);      // Execute kernel and read result
21   queue.enqueueReadBuffer(buffOut, CL_TRUE, 0, sizeof(out), out);
22   return 0;
23 }
```

Listing 2.3: Exemplary OpenCL program that writes the index to an array using the C++ API.

2.1.3 SYCL

SYCL originated from OpenCL and aims to provide a *"consistent language, APIs, and ecosystem in which to write and tune code for accelerator architectures"* [46]. It is a modern C++ API for accelerator programming. Similar to MPI, SYCL provides a standard for which there exist multiple implementations (e.g., AdaptiveCpp[7] or triSYCL[8]). It is no longer based on OpenCL, but itself generates low-level code, for e.g. the Level-Zero API [9]. End users of SYCL can use parallel loops with the option to define ranges for data structures and do not need to worry about memory management. This can be seen in Listing 2.4. SYCL `buffers` are custom data structures that are automatically allocated and migrated between host and device memory, relieving the programmer from memory management. For the example, three `buffers` are created with the data from the C++ data structures created previously (ll. 6-12).

```
1  #include <iostream>
2  #include <CL/sycl.hpp>
3  using namespace sycl;
4  int main() {
5    constexpr size_t N = 5;
6    std::vector<int> a = {0,1,2,3,4};
7    std::vector<int> b = {9,8,7,6,5};
8    std::vector<int> result(N);
9    // Buffers for storing the data.
10   auto a_buf = buffer<int, 1>(a.data(), range<1>(N));
11   auto b_buf = buffer<int, 1>(b.data(), range<1>(N));
12   auto result_buf = buffer<int, 1>(result.data(), range<1>(N));
13
14   queue que(gpu_selector{});    // Create SYCL queue
15   que.submit([&](handler& cgh) {    // Submit command group to the queue
16     auto a_accessor = a_buf.get_access<access::mode::read>(cgh); // Accessor to read a
17     auto b_accessor = b_buf.get_access<access::mode::read>(cgh);
18     auto result_accessor = result_buf.get_access<access::mode::write>(cgh);
19     cgh.parallel_for(range<1>(N), [=](id<1> idx) {      // Kernel to add vectors
20       result_accessor[idx] = a_accessor[idx] + b_accessor[idx];
21     });
22   }); // result_buf [9,9,9,9,9]
23   return 0;}
```

Listing 2.4: Exemplary SYCL program that calculates the sum of two vectors.

[7] https://github.com/AdaptiveCpp/AdaptiveCpp/ Accessed 09.04.2023
[8] https://github.com/triSYCL/sycl Accessed 09.04.2023
[9] https://spec.oneapi.io/level-zero/latest/core/INTRO.html Accessed 20.05.2024

Those buffers can be one, two, or three-dimensional, which can be passed as a template argument. Similar to OpenCL, a `queue` object needs to be created that is mapped to computational units `queues` (l. 14). In lines 16-18, the `accessors` for the data structures are created. Most importantly, the access mode determines whether those are accessed with read or write operations. Thereafter, a parallel for loop can be used with the created accessors, creating the sum of two vectors. Although the parallel for loops builds an additional abstraction layer and can also execute stencil operations by defining ranges for read buffers, SYCL is considered a low-level framework as for the parallel execution on multiple accelerators, the programmer has to manage the distribution of the data structures himself. However, when used on a single accelerator, SYCL could also be considered a high-level framework.

2.1.4 OpenMP

Open Multi-Processing (OpenMP) is a standard and API that enables shared-memory parallelism on CPUs. It is prevalent for most operating systems and CPU-architectures since its Architecture Review Board includes well-known companies such as Arm, AMD, International Business Machines Corporation (IBM), Intel Corporation, Cray Inc., HP Inc., Fujitsu K.K., NVIDIA, NEC, Red Hat, Texas Instruments, and Oracle Corporation [10]. The major advantage of OpenMP is the simplicity of the design and the integration into existing compilers. As depicted in Listing 2.5, it follows the fork-join model, allowing a programmer to mark regions that can be executed in parallel (l. 5, l. 8).

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main() {
4    int shared_array[10] = {0};
5    #pragma omp parallel num_threads(4)
6    {
7      // Increment elements of the shared array using a critical section
8      #pragma omp for
9      for (int i = 0; i < 10; i++) {
10       #pragma omp critical
11       shared_array[i]++;
12     }
13   }
14   return 0;}
```

Listing 2.5: Exemplary OpenMP program that increases each element of an array.

[10] `https://www.openmp.org/about/members/` accessed:12.12.2023

Noteworthy, a programmer does not need to define the number of threads. In the default case, e.g. the `omp parallel` pragma can be used to run code in parallel, and the maximum number of available threads will be used for execution. At first glance, OpenMP looks high-level. However, the example was chosen to illustrate that programmers who lack knowledge of parallel programming can fall into common traps in parallel programming. Line 11 increments each element of an array where the size of the array is larger than the available threads. As threads process multiple elements of the array (which could easily happen for bigger data structures when using the default number of threads), two threads may try to update the same array element simultaneously. Without the `omp critical` clause, an array element could be updated twice. OpenMP has several ways of handling shared memory access and fine-tuning the number of threads, such as pragmas for shared data structures (e.g. `shared()`) and atomic operations [11]. OpenMP is classified as a low-level framework because, although it is easy to use, it can easily lead to typical mistakes by uninformed programmers.

2.1.5 CUDA

The Nvidia corporation develops the parallel computing platform and programming model CUDA to simplify the programming of GPUs for GPGPU. The first version was released on June 15, 2007; it was one of the first frameworks to make general computing on GPUs available to a broad audience. Noteworthy, although CUDA is commonly associated with the programming language extension, the toolkit includes multiple other tools, such as debugging and profiler tools [12]. This overview is focused on the programming model. Most importantly, parallelism is managed by starting `kernels`, which are functions executed cooperatively by blocks of threads. Within the function, each thread has a unique identifier that can be used to implement distinct behaviour.

This is illustrated in Figure 2.1. On the left-hand side, the program first executes the main function. Memory on the GPU is allocated by calling `cudaMalloc` with the bytes required to be allocated and the pointer to access the memory (l. 8). Thereafter, the function executed in parallel is called (l. 9). Noteworthy, for the execution model the arguments passed in the between <<< and >>> denote the number and structure of threads started. In this case, one block with 1024 threads is started. As threads are always grouped in warps of 32 threads, it is reasonable to start a multiple of 32. As depicted in the Figure

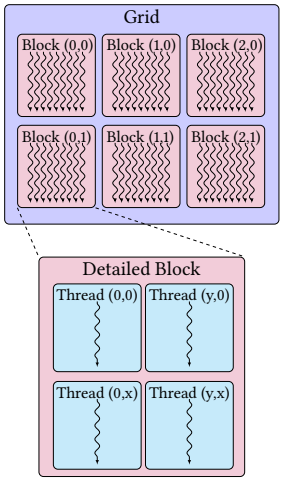
[11] Interested readers might want to have a look at the OpenMP specification [67] to get an idea of the extent of the framework.

[12] A exemplary complete list can be found in the recent CUDA toolkit `https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html` Accessed 10.04.2024

on the right-hand side, one execution *Grid* consists of multiple *Blocks* that have multiple *Threads*. In the program, the grid and thread blocks are started one-dimensional, as the depiction shows threads can also be started two-dimensionally (and three-dimensionally). A program usually requires multiple blocks, as blocks are limited to 1024 threads. In line 3 of the Listing, the current global thread identifier is calculated - `threadIdx.x` is the x-dimensionality of the thread started in the current block, and `blockDim.x` is the number of threads started for each block. Therefore, `blockDim.x*blockIdx.x` calculates the number of threads started by previous blocks. After executing the kernel, the output is copied to an array on the CPU (l. 11).

```
1  #include <iostream>
2  __global__ void write_id(int * &d_out) {
3    int idx = threadIdx.x + blockIdx.x * blockDim.x;
4    d_out[idx] = idx;
5  }
6  void writeIndex(int *& out) {
7    const int ARRAY_SIZE = 1024;
8    const int ARRAY_BYTES = ARRAY_SIZE * sizeof(int);
9    cudaMalloc((void **)&d_out, ARRAY_BYTES);
10   write_id<<<1, ARRAY_SIZE>>>(d_out, ARRAY_SIZE);
11   int h_out[ARRAY_SIZE];
12   cudaMemcpy(h_out, d_in, ARRAY_BYTES, cudaMemcpy↩
       DeviceToHost);
13 }
```

(a) Exemplary CUDA program that writes the index to an array.



(b) Depiction of the thread structure in CUDA.

Figure 2.1: Depiction of a exemplary CUDA program and the CUDA thread model.

Blocks are always limited to 1024 threads. This is important as choosing the number of threads and blocks can have a major impact on the overall performance of the program. A block of threads is executed on a streaming multiprocessor (SM). A SM is a Single Instuction Multiple Data (SIMD)-processor and can therefore execute the same instruction on multiple data points. It has local memory that can be shared between the threads. The number of SM depends on the GPU architecture. With the release of new GPU architectures, the number of threads that can be started in parallel per SM changes due to hardware-specific design decisions. For example, the Turing architecture allows the start of 1024 threads per SM, while the subsequent architecture, Ampere, can start 2048 threads per SM. Another important optimization technique includes the reduction of divergence for threads within one warp.

2.1.6 OpenACC

The open programming standard Open Accelerators (OpenACC) originated in 2012 as the demand for a simpler way to write GPU code rose. Similar to OpenMP, OpenACC allows parallel execution by adding *Pragmas* to the existing programming code. The standard was developed by all companies included in the creation OpenMP that manufacture GPUs. In contrast to CUDA, OpenACC clearly provides an easier syntax for creating parallel programs. In Listing 2.6, the example used for demonstrating CUDA in Listing 2.1a is implemented in OpenACC. The parallelism is implemented in line 6. Especially when parallelizing an existing program, adding pragmas is immediately realizable. However, the dispersion of OpenACC was limited. Firstly, support for OpenACC was only gradually built into common compilers, impeding the adoption. Moreover, users of clusters equipped with GPUs already developed knowledge to create CUDA programs and would invest additional effort to switch to OpenACC programs. Although the OpenACC directives look straightforward, switching to OpenACC can be challenging as programmers still need to think about possible errors caused by the parallelism and the partly misleading naming scheme. For example, while the pragma `parallel loop` forces parallelism, although variables might be written in a nondeterministic manner, the `kernel` directive tells the compiler to look for instructions to execute in parallel [66]. As the unknowledgeable use of pragmas might lead to errors, and certain pragmas require knowledge in parallel programming, OpenACC is still considered low-level. Moreover, the additional layer of abstraction comes with a performance loss, e.g., Hoshino et al. found that for embarrassingly parallel applications OpenACC can reach up to 98% of the performance while more complex programs take about twice as long [43]. The exact slowdown highly depends on the application but discourages users from using OpenACC.

```
1 #include <stdio.h>
2 #include <openacc.h>
3 void writeIndex(int *& out) {
4   const int N = 1024;
5   #pragma acc parallel loop
6   for (int i = 0; i < N; ++i) {
7     out[i] = i;
8   }
9 }
```

Listing 2.6: Exemplary OpenACC program writing the index to an array.

2.2 High-Level Frameworks

Multiple frameworks to abstract from parallel programming details have been established over the last couple of years. An overview of selected high-level frameworks can be seen in Table 2.1. The following criteria were used to select the frameworks from the overview to be presented in more detail:

- The framework has active development in the last two years (either with publications or active work in a public repository code),
- the framework has support for multi-GPU environments,
- the development of the framework is based on C/C++,
- the publication includes reasonable example applications that are not specialized to a specific area.

Due to no active development recently, PARTANS [56], PSkel [70], SkelCL [77], Lift [84], and SkeTo [74] were excluded. Noteworthy, as a spiritual successor, the Rise and Shine project might follow up the research of Lift [78]. Although the second version of FastFlow added support for OpenCL and CUDA [1], the current focus of FastFlow is communication skeletons, which we will not discuss in this dissertation. Therefore, FastFlow is also not further described. Python packages were excluded as they rarely offer the same performance as C++-based approaches for generic operations (ParSL and PySke). DUNE, EPSILOD, and ExaStencil are not listed as they are specialized in stencil operations but will be listed in Chapter 6. For Marrow, no repository could be found. Muenster Skeleton Tool for High-Performance Code Generation (Musket) has also not been enhanced since 2020. However, as it is used as a comparison in Chapter 5, it was decided to describe it briefly. Additionally, Skepu and Celerity were identified as suitable candidates to compete with Muesli. Therefore, the following Subsections will first explain algorithmic skeletons in general. Afterwards, in Subsection 2.2.2, 2.2.3, 2.2.4 SkePu, Celerity, and Musket are presented.

2.2.1 Algorithmic Skeletons

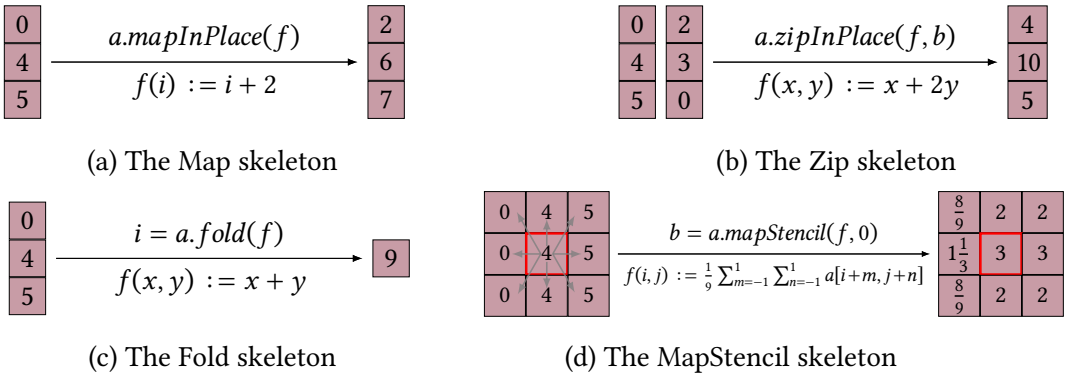
Cole introduced a parallel programming model which encapsulated recurring parallel programming patterns into so-called *Skeletons* [12]. Those patterns are typically split into two, not completely disjoint groups: 1) *data-parallel* and 2) *communication* skeletons. While data-parallel skeletons typically spend most of the time calculating, communication skeletons distribute work and share data between multiple nodes or accelerators.

Name	Hardware	Backend(s)	Publicly Available	Type	Data Structures	Exemplary Parallel Patterns
Celerity	GPUs, Nodes	SYCL Implementation MPI	✓[86]	API, Runtime Environment	1/2/3D	Range Mappers, Reduction, Parallel For
FastFlow [31]	Nodes[a]	MPI, Cereal	✓[89]	C++ Template Library	Native C++	Streams and Pipelines
Lift [34]	GPUs, CPUs	OpenCL	✓[84]	Programming Language and Compiler	1D/2D Array	Generic Stencil, Map, Reduce, Zip, Split, Scatter, Gather, Slide
Marrow [57]	GPU	OpenCL, CUDA[b]	X	C++ Framework	1D [c]	Reduce, Scan, Sort, Filter, Map
Muesli [22]	GPU, CPU, Nodes	MPI, OpenMP, CUDA	✓[87]	C++ Library	1D/ 2D/ 3D	Map, Zip, Reduce, MapStencil
Musket [73]	GPU, CPU, Nodes	MPI, OpenMP, CUDA	✓[88]	DSL	1D/ 2D/ 3D	Map, Zip, Reduce
ParSL [3]	CPU, Nodes, GPU	MPI, Accelerated Libraries	✓[83]	Python Package	Python Datastructures	`python_app` / `bash_app` task management
PySke [55]	CPU, Nodes	MPI	✓[54]	Python Package	SList, SArray, PList, PArray	Map, Reduce, Zip, Map2
SkePU [24]	CPU, GPU, Nodes	OpenCL, CUDA, MPI, OpenMP	✓[23]	Precompiler, C++ Template Library	1D/ 2D/ 3D / 4D	Map, Reduce, MapReduce, Scan, MapOverlap, MapPairs, MapPairsReduce, Call

Table 2.1: Overview of high-level frameworks listing the targeted hardware, the programming frameworks used, and the accessibility.

[a] GPU Support was experimentally implemented in FastFlow 2
[b] Released as a master thesis [92]
[c] An example for a one-dimensional data structure could be found in the most recent publication [92].

Examples of the most commonly known data-parallel skeletons can be seen in Figure 2.2. For example, the *Map* skeleton takes a unary function to each element of a data structure (Figure 2.2a). The `Zip` skeleton combines two data structures; therefore, it requires a binary operation combining two elements into a single element (Figure 2.2b). The `Fold` skeleton, also referred to as Reduce skeleton, takes a binary function and iteratively combines all elements of a given data structure and returns a single element (Figure 2.2c). The `MapStencil` skeleton depicted in Figure 2.2d will be explained in detail in Chapter 6. It reads a defined range of surrounding elements to calculate a new value. Therefore, it can not be categorized into a specific type of function; it reads a variable range of elements from the input data structure. The `MapStencil` skeleton is mostly a data-parallel skeleton. However, as border elements of the data structure have to be communicated between processors, part of the execution time is spent on communication.



(a) The Map skeleton

(b) The Zip skeleton

(c) The Fold skeleton

(d) The MapStencil skeleton

Figure 2.2: Examples for algorithmic skeletons.

More obvious examples of communication skeletons are `Broadcast` and `Gather`. `Broadcast` communicates a data structure or single elements to all computational nodes while `gather` collects elements from each computational node. An example of a more complex communication skeleton is the `Farm` skeleton. It is used to assign tasks to workers efficiently. Ideally, every worker is equally occupied [20]. There are multiple ways to implement skeleton frameworks. Common methods are libraries and DSL, which will be briefly explained. Of course, high-level frameworks can also be implemented in other ways, such as runtime systems and APIs. These might not be mutually exclusive; usually the focus is mentioned in the description of the framework.

Libraries: One way to make the concept available for end users is by implementing a library. Many programming languages support libraries with simple `import` or `include` statements, such as C/C++, Java, and Python. The most important advantage of a library is that existing language programs can be easily extended with the data structures and functions of the library. Moreover, basic functionality, such as reading streaming data, can be taken from the existing functionality of the language.

Domain-Specific Languages: In contrast to general-purpose languages, a DSLs is designed to meet a specific purpose and commonly has a simplified syntax to ease the creation of the program. Most importantly, domain-specific languages have the advantage of parsing, processing, and rewriting the code written in the DSL as each DSL has at least a generator or compiler taking the program and generating a new valid program from it. This also means that the complete program analysis can be considered when generating the program code.

2.2.2 SkePU

The high-level programming approach SkePU supports multiple skeletons for multi-GPU and multicore CPU clusters [24]. It uses a custom precompiler that produces valid C++ code that can be compiled with recent versions of Clang, GCC, or NVCC. Therefore, running SkePU first requires the LLVM and Clang source to build the compiler driver. Possible backend-specifications are `CPU`, `OpenMP`, `CUDA`, or `OpenCL`. This enables not only Nvidia GPUs but also AMD accelerators. All backends can be combined with the StarPU runtime system to target multiple nodes.

Skeletons are supported for custom data containers that can be one, two, three, and four-dimensional. The supported skeletons are: `Map`, `Reduce`, `MapReduce`, `MapOverlap`, `Scan`, `MapPairs`, and `MapPairsReduce`. A pragmatic but rather unusual decision is to additionally provide a generic `call` operation which does not follow any predefined algorithmic pattern but allows the user to process a data structure in a flexible way [13]. [91]

SkePU handles skeletons as objects that have the data structures as parameters. Listing 2.7 implements a program for calculating a heat distribution. A data structure of type matrix is created (l. 7), and thereafter, a skeleton object with the function `heat2D` as a constructor argument is created (l. 11). This example was chosen to outline the advantage of having skeleton objects: in lines 12-15 the object can be configured to support, e.g., in this case, different edge handling and update modes. Finally, the skeleton is called (l. 16).

[13] The interested reader can see an example in the publication [26].

```
1  float heat2D(skepu::Region2D<float> r) {
2          float newval = r(-1,0) + r(1,0) + r(0,-1) + r(0,1);
3          newval /= 4;
4          return newval;
5  }
6  skepu::Matrix<float> calcHeat() {
7          skepu::Matrix<float> domain(size, size, 0);
8          // Fill Matrix with data
9          [...]
10         // Creating the skeleton object.
11         auto update = skepu::MapOverlap(heat2D);
12         update.setOverlap(1, 1);
13         update.setEdgeMode(skepu::Edge::Pad);
14         update.setPad(0);
15         update.setUpdateMode(skepu::UpdateMode::Normal);
16         update(domain, domain);
17         return domain;
18 }
```

Listing 2.7: Exemplary SkePU program simulating two-dimensional heat distribution - adjusted from the Git-repository [23]

2.2.3 Celerity

Celerity is a C++17 API and runtime system that targets clusters with multiple nodes and accelerators [80]. To support multiple hardware vendors, Celerity requires an SYCL implementation running on the target machine, e.g., DPC++ or AdaptiveCpp[14]. E.g. for AdaptiveCpp, the hardware supported includes CPUs with a C++17 OpenMP compiler, Nvidia GPUs, and AMD GPUs that support ROCm can be used [15]. A major advantage of Celerity is that the end user does not need to manage multiple processing units.

In contrast to previous high-level frameworks, Celerity is not built on the idea of algorithmic skeletons. Instead, Celerity provides parallel loops, reduction operations, and range mappers for custom data structures. To illustrate the difference, the previous example of the two-dimensional heat distribution is reimplemented. Next to two data structures, a `queue` is created (Listing 2.8 ll. 1-3). `Queues` receive tasks that are executed one after another. In the given example, a single task is submitted to the queue (l. 6). To access data structures `accessors` need to be defined (ll. 7+8) that distinguish between

[14] Previously known as hipSYCL/OpenSYCL

[15] Interested Readers can check their compatibility `https://rocm.docs.amd.com/projects/install-on-linux/en/latest/reference/system-requirements.html` Accessed: 19.12.2023

read and write access and different access patterns, in this case, the surrounding elements (neighbourhood). The consecutive `parallel_for` code fragment writes the calculated elements to the buffer. Celerity requires the user to identify and handle boundary values (l. 14). Indices can be received with custom methods (ll. 10+11). Even if the creation of queues and handlers takes some time to get used to, Celerity offers data structures and functionalities that abstract from low-level details.

```
1  celerity::distr_queue queue;
2  celerity::buffer<float,2> data{celerity::range<2>(size, size)};
3  celerity::buffer<float,2> swp_data{celerity::range<2>(size, size)};
4  [...] // Initialize data
5
6  queue.submit([&](celerity::handler& cgh) {
7          celerity::accessor dw_buf_read{buf_read, cgh, celerity::access::neighborhood{2,2}, celerity↩
               ::read_only};
8          celerity::accessor dw_buf_write{buf_write, cgh, celerity::access::one_to_one{}, celerity::↩
               write_only, celerity::no_init};
9          cgh.parallel_for<class update>(celerity::range<2>(size,size),  [=, s = size](celerity::item↩
               <2> item) {
10                 const auto x = item.get_id(0);
11                 const auto y = item.get_id(1);
12                 celerity::range<2> range = celerity::range<2>(s, s);
13                 // Check for out-of-bounds access.
14                 if (is_on_boundary(range, 2, item)) {[...]}
15                 float sum += dw_buf_read[{x, y+1}] + dw_buf_read[{x+1, y}] + dw_buf_read[{x, y-1}] +↩
                         dw_buf_read[{x-1, y}];
16                 dw_buf_write[item] = sum/4;
17         });
18 });
```

Listing 2.8: Heat distribution function written in the Celerity runtime

2.2.4 Musket

The DSL Musket solely focuses on creating the means to write parallel programs. The syntax is based on C++, but it is defined using the Xtext framework. Therefore, it uses a parser and an editor that can be incorporated in Eclipse [79]. Therefore, it can provide custom syntax highlighting, code completion, and validation. It generates parallel CPU and GPU code and can scale over multiple nodes. When a musket program is created and transformed into a low-level program, the corresponding CMake files are also generated to ease the execution of the program on a cluster.

A *Musket* program is divided into four parts, namely *meta-information*, *data structure declaration*, *user function declaration*, and *main program declaration*. This can be seen in Listing 2.9. Firstly, multiple configurations can be defined by using the corresponding variables. This example generates code for four nodes, each with 24 cores and four GPUs. Thereafter, the data structures are declared. Musket supports arrays or matrices, those have a data type and size and can either be distributed on the available units or copied on each unit. As primitive values, booleans, integers, doubles, floats, and strings are supported. Next, the user functions look like C++ functions but support only a subset of the operations supported by C++. Here, a value is doubled (ll. 9-11). Lastly, all skeleton calls have to be executed from the main function. The program can not be structured using further methods. Supported skeletons are `Map`, `Zip`, `Fold`, `Gather`, `Scatter` (equivalent to Broadcast), and `ShiftPartitions` (horizontally and vertically). `Map` and `Zip` also have in place and index variants. In the example, firstly, a timer is started (l. 14), and afterwards, the `mapInPlace` skeleton is called (l. 15) with the user function as an argument. Thereafter, the timer is stopped (l. 16).

Although the syntax is fairly easy, this approach has the disadvantage that functionality that is by default available in general-purpose programming languages has to be implemented. For example, Musket does not allow to read data from an input stream.

```
1  #config PLATFORM GPU CPU_MPMD
2  #config PROCESSES 4
3  #config CORES 24
4  #config GPUS 4
5  #config MODE release
6
7  array<int,16384,dist> a;
8
9  int double_values(int i){
10         return i + i;
11 }
12
13 main{
14         mkt::roi_start();
15         a.mapInPlace(double_values());
16         mkt::roi_end();
17 }
```

Listing 2.9: Exemplary Musket program to double the values of an array.

3

Muesli - Muenster Skeleton Library

The first version of Muesli was presented in 2002, introducing a C++ library offering multiple skeletons [48]. Ever since, the main concepts of Muesli have been maintained, but the library has been adjusted to meet the demands of the changing hardware environment and parallel programming frameworks. This includes the support for multi-processors [11] and multi-GPU systems [21]. All presented features within this thesis are entirely compatible with each other. Features developed in previous work might no longer be supported as they have not been compatible with all features of Muesli.

In the following sections, firstly, the data structures of the fourth version of Muesli are presented in Section 3.1; secondly, Section 3.2 gives a detailed description of all available skeletons, and Section 3.3 list additional functionalities provided by Muesli. A basic understanding of C++ and CUDA is required for the following explanations.

3.1 Data Structures

Muesli provides three types of data structures: distributed arrays (DAs), distributed matrices (DMs), and distributed cubes (DCs). In contrast to the built-in data structures of C/C++, the Muesli data structures have skeletons as member functions and other helper functions, simplifying the programming of parallel applications and guaranteeing error-free data accesses. Those can be easily created, requiring the size and, optionally, either a constant value or a pointer that can be passed as the data structure's initial value(s). Listing 3.1 demonstrates the creation of a DA of size 4 with 5 as the initial value, a DM of size 16 with the initial value of 6 at all elements, and a DC with 64 elements which is filled with the data at the pointer `data`. Any data structure provided by Muesli can also be created from an existing data structure (l. 5). This is enabled by implementing the suitable constructors and the Rule of five [16], allowing efficient move operations.

[16] `https://en.cppreference.com/w/cpp/language/rule_of_three` Accessed 12.04.2024

```
1 DA<int> a(4, 5); // delivers a = [5, 5, 5, 5]
2 DM<float> b(4, 4, 6.0f);  // delivers b = [[6.0, 6.0, 6.0, 6.0], [...], [...], [...]]
3 DC<int> c(4, 4, 4, int * data);
4 DM<float> d = b;
```

Listing 3.1: Initialization of Muesli data structures

Constructors: Each data structure has three default constructors. One constructor creates data structures of specific dimensions, requiring the number of dimensions as arguments. The second constructor additionally initialises all elements with a value that can be either a constant value or a pointer. The last constructor has an additional optional value to only distribute complete rows over computational units.
Rule of five: The rule of five includes all necessary methods for efficient copying and moving data structures. Additionally, it includes efficiently assign, destruct, and move operations. Although it is desirable to avoid defining unnecessary operations (C++ Core Guidelines), it is reasonable to duplicate data structures in our context.

The design of the given data structures is ideally suited to inherit properties from a more general class. The relation of the different data structures is displayed in Figure 3.1. Theoretically, every dimensionality can be displayed as a so-called *distributed data structure (DS)*; however, this offloads index calculations to the end user. Therefore, all Skeletons that do not require an index can be represented in the class DS. Index variables, skeletons using the index, and the depending constructors are merely available for DAs, DMs, and DCs classes. For example, constructing a DM with five columns and two rows and executing a `mapInPlace` skeleton is equivalent to creating a DS with ten elements and executing the skeleton. Functions that can be used for every data structure include:

fill: The function either takes a constant value, a pointer of type `T`, or a function and copies either the value to all elements in the DS, all values of the pointer to the corresponding element, or executes the function passed for each element. Therefore, the function has to take one argument, the index, and return the value of type `T`.
get(index)/[index]: Single elements of the data structure can be read with the get function, or the operator `[]` can also be used to access array elements. This function should not be used frequently as this might invoke GPU memory transfers for single elements. Both functions are redefined for multiple dimensions in the corresponding data structures to ease accessing elements.
set(index, value): The set function is also user-friendly since it can be redefined for multiple dimensions to, e.g., pass a row and column argument. Moreover, the value can

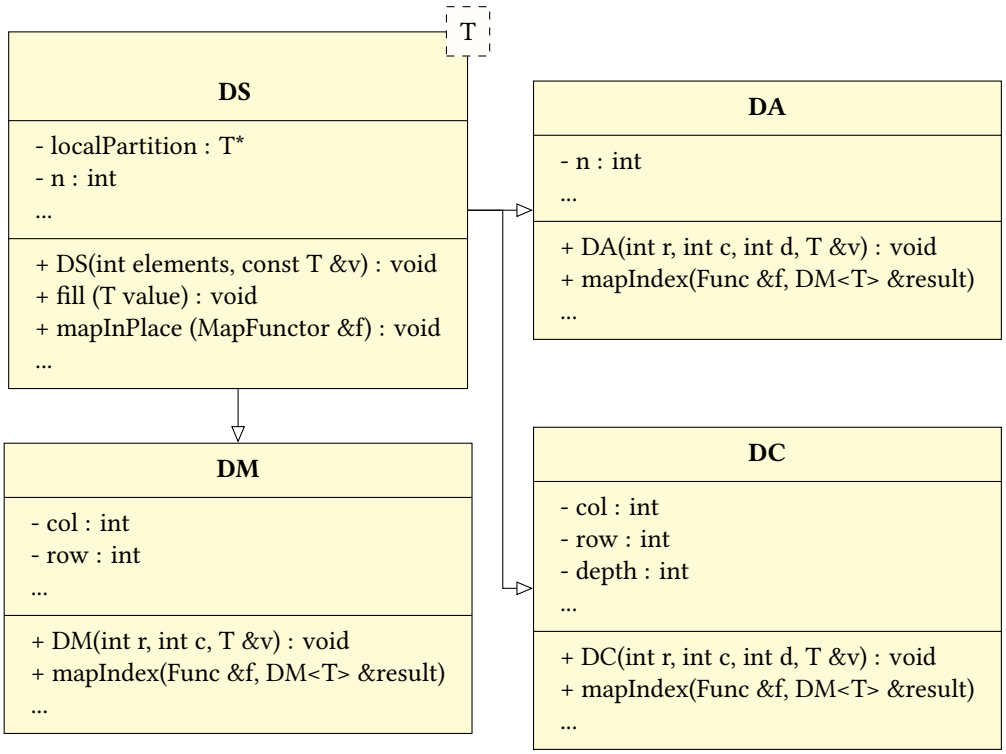


Figure 3.1: Class diagramm of the data structures available in Muesli.

be either a constant value or a pointer.

`isLocal(index)`: The function allows the implementation of distinct behaviour for processes as it can be distinguished if an element is available to the current process.

`updateHost/updateDevice`: If requested, the data structure can be updated manually. This might be necessary for more experienced programmers who want to implement methods themselves or use other libraries. However, this is not necessary. For operations that require updating the data (such as reading elements), the data structures are automatically updated.

`show`: Writing the resulting data structure to a file is repetitive work; therefore, the show function provides an easy-to-use function to either inspect the data structure at the command line or write the data structure to a file. Therefore, the user can optionally define a constant specifying the target output file.

Other functions have been omitted if they are mostly internally used, rarely used, or are skeletons that are independent of the dimensions. The first dimension of the distributed data structure determines the distribution of elements to computational nodes. This means that matrices are distributed in a row-wise manner. Figure 3.2 displays the distribution of

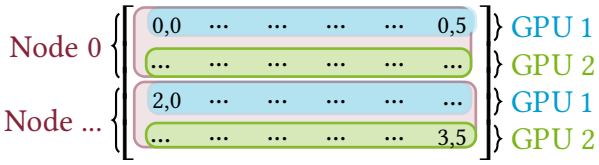


Figure 3.2: Distributing a DM by rows on multiple nodes and GPUs.

a 6×4 matrix with two nodes, each equipped with two GPUs. DC are then distributed by layers. Different distribution mechanisms are also discussed in Chapter 4 and Chapter 6.

3.2 Skeletons

Skeletons can be invoked by calling the corresponding member function of the data structure object. This simplifies the mapping of skeletons to data structures. Each skeleton function receives a functor that can be either an anonymous function or a functor object, specifying the instruction to execute for every element of the data structure. Although using a functor object is slightly more complex, it allows the definition of custom constructors, class variables, and custom methods. Optionally, some skeletons require passing additional data structures. The application of Muesli skeletons is best explained with an example. After the example, all available skeletons are listed.

Listing 3.2 displays a function for calculating the dot product of two vectors. For this purpose, it is assumed that two DAs of the same size are passed as parameters to the function. Firstly, an anonymous function `product` is created that multiplies two elements. The `MSL_USERFUNC` keyword is necessary to declare functions as compatible with CPUs and GPUs. It uses the function type qualifiers `__host__` and `__device__` in case CUDA is available. The product function is used with a `zipInPlace` skeleton in line 7. This skeleton requires passing a data structure of the same size next to the functor variable. The result is written directly to the DA *a*, as it is an in-place variant of the skeleton. Next, a functor object is created (l. 8). The definition of the class `Sum` is given in lines 1-4. The number of arguments required for the operator function is determined by the `Functor` class. This enables the program to show an error without compiling the program in case the functor object has an unsuitable number of parameters for the assigned skeleton. The class could also have constructors, variables, and further methods. Calculating the sum could have also been expressed as an anonymous function; however, it was defined as a class to showcase both scenarios. The result stored in the DA *a* is reduced to a single element by applying the `fold` skeleton with the *sum* object (l. 10).

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

Listing 3.2: Calculation of the dot product with Muesli

As the available skeletons have been described in detail in previous work [19], the following enumeration briefly sketches all skeletons available in Muesli.

Map-Variants: The Map operation executes a unary function on every single element of a data structure. *Muesli* provides all combinations of `InPlace` and `Index` variants for the Map skeleton. The `InPlace` variants of the skeleton operate on a single data structure, namely the data structure executing the call. `Index` variants pass the index/the indices to the user function, allowing it to execute different branches depending on the element's position. Therefore, for calling a `MapIndex` skeleton with a DC, a `Functor4` is required, while a DM requires a `Functor3`.

Zip-Variants: The Zip skeleton combines two data structures with the same dimensions and number of elements into one data structure. This skeleton also offers all combinations of `InPlace` and `Index` variants. Especially with increasing data structures operating in place, reducing the cost of creating data structures is essential. Noteworthy, a skeleton to zip three data structures of the same dimensions was added.

MapStencil: All prior skeletons are independent of other values inside the data structure. In contrast, stencil operations are classified by calculating new elements depending on other elements in the data structure. This operations class is used in image processing, computational fluid dynamics, and mathematics (Jacobi method). Particularities of the skeleton, most importantly, the access to elements for the end-user and the communication of overlapping regions in the back end, are implementation-specific and will be discussed in detail in the dedicated Chapter 6.

Fold: Reducing a data structure results in returning a single element of the given datatype. Therefore, the function or functor passed to the reduce skeleton takes two parameters of the datatype and returns a single element. This is used to find, e.g., the maximum value.

Other frameworks use `Reduce` as an alternative nomination. Noteworthy, it is the only skeleton which requires to have the same return type as the type of the data structure.

`Broadcast:` Especially when working on multiple nodes, distributing input data over multiple nodes requires broadcast data.

`Gather:` When saving the results, the exact opposite operation is required, namely gathering all elements of the data structure. The skeleton can be used in two different manners. Either the data from all processes is written to a pointer, or the result is written to one of the custom Muesli data structures.

3.3 Helper Functions

Helper functions are not accessible through a data structure but are always available in the `msl` namespace. The `initSkeletons` function sets default values for all attributes required to run a basic parallel Muesli program. This includes (but is not limited to):

`total_processes, procid:` The number of available MPI processes and the identifier for the current process are determined automatically with the corresponding MPI functions; the number can be overwritten by passing the number of nodes as an argument to the `mpirun` command.

`max_gpus`, **`num_gpus`**: The maximum of available CUDA GPUs available on the system is checked with (`cudaDeviceProperties`) and by default the number of GPUs used is set to the maximum.

`num_threads` The number of threads used on the CPU is set to the maximum of available threads (`omp_get_max_threads`).

`num_runs:` Runtime measurements might require to execute the program multiple times; this is simplified by setting the number of runs; the default value is one.

`start_time:` The start time of the program is recorded to track the total time.

`threads_per_block:` As a default 1024 threads are started for each GPU-block. Experienced programmers might adjust the number for fine-tuned GPU optimisations.

All variables have matching `getters` and `setters`, allowing knowledgeable programmers to fine-tune and experiment with settings. As the counterpart, the function `terminateSkeletons` should be called at the end of the program. It calls the `MPI_Finalize` function and measures the total runtime. For customised time measurements, the `startTime`, `splitTime`, and `stopTime` functions can be used.

4

Heterogeneous Systems

Enhancing and maintaining skeleton libraries is either related to the functions provided to the end-user, the efficiency of those functions, or the supported hardware. As hardware constantly changes, the hardware supported by a high-level approach needs to be reevaluated. With the emergence of GPGPU computing, one of the major focuses has been to provide GPU support for high-level frameworks. The release of new GPUs still speeds up the program as the number of computational cores increases but rarely requires making major adjustments to high-level frameworks. Other types of accelerators, e.g. Field Programmable Gate Array (FPGA), have recently gained attention but are not commonly available on most clusters. As support for specialised accelerators would only be available to users who have bought them in the past, it can be inferred that those users have expertise in hardware description languages. However, support for other accelerators needs to be constantly reevaluated. Therefore, the commonly available hardware on clusters are CPUs and GPUs. Muesli supports both types of processing units and the connection of multiple nodes with multiple GPUs [21]. Therefore, we found that the improvements that can be made to the hardware support are very limited. The current GPU support merely uses the CPU for communication. Therefore, the major improvement that could be made with the hardware available is to exploit the usage of the CPU. Firstly, the related work for sharing workloads is discussed in Section 4.1; thereafter, the implementation in Muesli is presented (Section 4.2). Next, the implementation is evaluated in Section 4.3 and finally, all findings are concluded in Section 4.4.

4.1 Related Work

Multiple other researchers have invested in load-balancing strategies to keep the CPU occupied when calculating on the GPU. Raju and Chiplunkar have surveyed multiple approaches, putting the focus on energy efficiency [72]. Mittal and Vetter categorise the approaches to hybrid execution by firstly distinguishing between dynamic and static scheduling and secondly on the basis for the workload partitioning [61]. In dynamic approaches, tasks are assigned at runtime to processing units, while static approaches

partition the workload at compile time. The workload partitioning is based on the available memory spaces, the relative performance, the analysis of the task to be executed, or the floating-point precision of the processing units [61]. The findings from the surveys can be used to decide on a method when the drivers that favour hybrid execution have been determined.

In previous research on high-level approaches Öhberg, Ernstsson, and Kessler allowed defining the ratio between a single CPU and a single GPU in SkePU 2 [65]. However, the current version, SkePU 3, does not support hybrid execution. The Marrow framework introduced an approach to load work to multiple GPUs statically [2] and thereafter presented a workload distribution to distribute computations to CPUs and GPUs [75]. The second approach is also calculated statically but targets reoccurring computational trees and calculates loads based on the runtimes from previous runs. Unfortunately, Marrow is not publicly available, making the current status of the framework hardly testable. Wrede and Ernsting developed a variant of Muesli which supports parallel CPU and GPU execution [93]. However, their evaluation was based on specific examples; therefore, the drivers for speedups could not be identified. This impedes finding a meaningful cost model or reasoning an automatic approach to allocate tasks. For this purpose, the implementation was transferred to the current Muesli version. The design and adjustments to the design will be explained in the next section.

4.2 Implementation

To test different CPU-shares, the variable `cpu_fraction` is created, which allows the end-user to set a custom share calculated by the CPU. Users who are unsure how powerful their CPU is can experiment to find a suitable setting. As Muesli is built to either generate a CPU or a GPU parallel program, the program for the GPU has been adjusted to calculate elements on the CPU simultaneously. In the backend of the library, this has led to the following adjustments. Firstly, the partitioning of the data structure has been adjusted. Data structures are segmented equally row-wise when using multiple GPUs. With the percentual share, the number of elements allocated to the CPU are subtracted from the total elements, and the remaining elements are allocated to the available GPUs. This distribution is exemplified in Figure 4.1 for a 4×6 matrix with two nodes, each with one CPU and one GPU. The program was configured to use a CPU-share of one-third. Therefore, each CPU has four elements, and eight elements are assigned to the GPU.

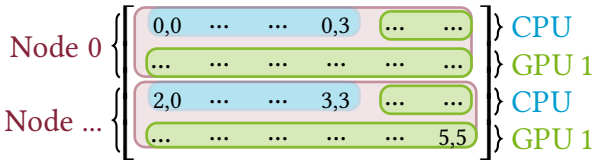


Figure 4.1: Distributing a DM by rows on multiple nodes and computational units.

For keeping track of the indices, each node has a `GPUExecutionPlan` for every GPU allocated to that node. The struct `GPUExecutionPlan` copies the template argument from the data structure and has the following other arguments:

- `long size;` stores the number of elements allocated to one GPU.
- `size_t bytes;` stores the number of bytes allocated on the GPU. This prevents the repeated calculation for copy instructions.
- `int gpuRows,gpuCols,gpuDepth;` stores the number of rows, columns, layers on a single GPU. Those are only stored for suitable data structures.
- `int first;` saves the total index of the first element processed by the first GPU on the node.
- `int firstRow, firstCol, firstDepth;` stores the first global row, column, and layer where the GPU processing starts.
- `int lastRow, lastCol, lastDepth;` stores the last global row, column, and layer where the GPU processing ends.
- `T *h_Data, *d_Data;` save the pointers to the host and device data.

With this minimalist design, the necessary information to execute index calculations are only calculated when a GPU is available.

4.3 Evaluation

Our study is designed to evaluate the suitability of specific skeletons for splitting the calculations. Besides the type of skeleton, the size of the data structure, the hardware setup, and the complexity of the user function are varied as they influence the optimal CPU-share. All tested hardware setups are listed in Table 4.1. Although this is rather unusual, two local PCs are also used as a hardware setup. The runtime measurements of local setups have to be considered carefully, as they might fluctuate. However, it increases the diversity of environments tested, allowing the incorporation of weaker GPUs in the comparison. Even though the primary user group of Muesli are cluster users, it should

Type	Abbr.	GPU	SM	CPU	Cores
Local	GTX750	GeForce GTX 750 Ti	5	Intel(R) Core(TM) i7-4790	4
Local	RTX3070	GeForce RTX 3070 Ti	48	Intel(R) Core(TM) i9-12900H	14
Cluster	RTX2080	GeForce RTX 2080 Ti	68	Zen3(EPYC 7513)	32
Cluster	A100	Nvidia A100	108	Zen3(EPYC 7513)	32

Table 4.1: Overview of used hardware.

not be precluded that Muesli will also be in use on local computers. While the first and the third have already been tested [39], two further hardware setups are added as part of this thesis. The first setup clearly has the weakest GPU with five SM. As this GPU is no longer supported by the current CUDA Toolkit, the experiments were not repeated for this thesis. The second local setup is a relatively recent GPU (released in May 2021). Noteworthy, the CPUs used in the cluster environment are more powerful than the local CPUs. The Nvidia A100 is especially interesting as it has 40 GB of GPU memory, allowing the processing of bigger data structures without intermediate memory transfers.

Two preventative measures were taken to prevent runtime fluctuations. Firstly, each program was executed 25 times, and the average was used. Secondly, within a program, the skeletons were called multiple times to prevent runtimes that were smaller than 2 seconds. The following runtime experiments firstly summarise the results found on the cluster (Subsection 4.3.1). Secondly, Subsection 4.3.2 elaborates on the results found for local environments.

4.3.1 CPU Usage on a Cluster

All cluster experiments were repeated to ensure the topicality of the results. The original results can be seen in Chapter 11 but will also be explained if those deviate from the presented results. Firstly, the reproduced runtimes from the RTX2080 partition are presented. As the memory of the GPU is limited, data sizes from $120^3 - 290^3$ are tested. The results are depicted in Figure 4.2. Jumping in the eye, offloading elements to the CPU is not beneficial for small data structures for *every* skeleton. For the majority (except the `Fold` skeleton), it is a slowdown. For the data size, 120^3, a share of 0.0% has a speedup of factor 2-6 compared to the smallest CPU share tested (0.02%).

For a data size of 170^3, a breakpoint for skeletons *not* using the index can be observed. Using the CPU becomes faster. With an increasing data size, a clear tendency for the `Map`, `MapInPlace`, `text`, and `text` skeleton can be seen. For those skeletons, it is beneficial to offload up to 19% of the calculation to the CPU. Not all shares have been included as they impede the readability of the graph. When increasing the share to 20%, the runtime

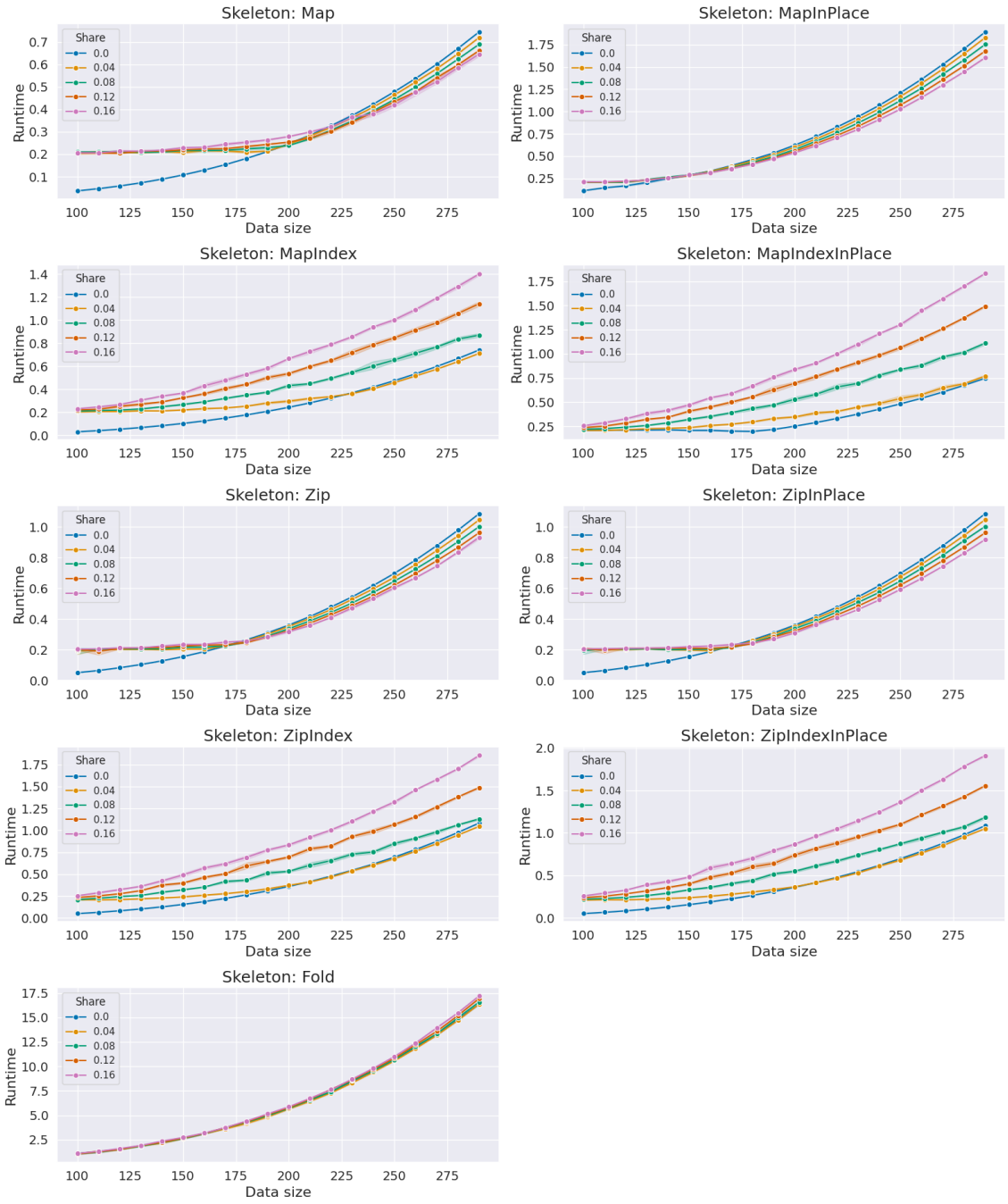


Figure 4.2: Runtimes (in seconds) of benchmark skeleton calls with a simple user function on the RTX2080 partition with varying percentual shares. The data structure size is the side length of the cube.

increases again. A speedup of approximately 1.2 can be reached compared to the GPU variant.

In contrast, skeletons that use the index could not achieve a significant speedup. As those skeletons require the index for each CPU calculation, the corresponding index is repeatedly calculated. They have a clear disadvantage compared to the other skeletons, especially when using a simple user function (e.g. an addition); the index calculation dominates the runtime. With a CPU share of 2% and 4%, a speedup of 1.05 could be achieved, which is neglectable. This result shows an improvement to the previous results as the tendency can be seen more clearly. However, it also strengthens the assumptions that some operations might be faster solely calculated on the GPU. Lastly, the `Fold` skeleton does not show big differences for different CPU shares. As it is a communication-intensive skeleton, the runtime for a CPU program was also measured. The CPU program was significantly faster, requiring only 1.01 seconds for the biggest tested data size (in contrast to 17 seconds). The GPU runtime measurements do not include the time taken for the data transfer.

As for the skeletons using the index, no speedup could be achieved; it had to be concluded that the index calculation was the dominating factor in the calculation. To verify this assumption, the complexity of the user function was artificially increased. Instead of a simple calculation, a loop with 2,000 iterations was executed inside the user function. Although this example is artificial, it is necessary to infer if index variants should never be executed on the CPU or if an analysis of the user function is required.

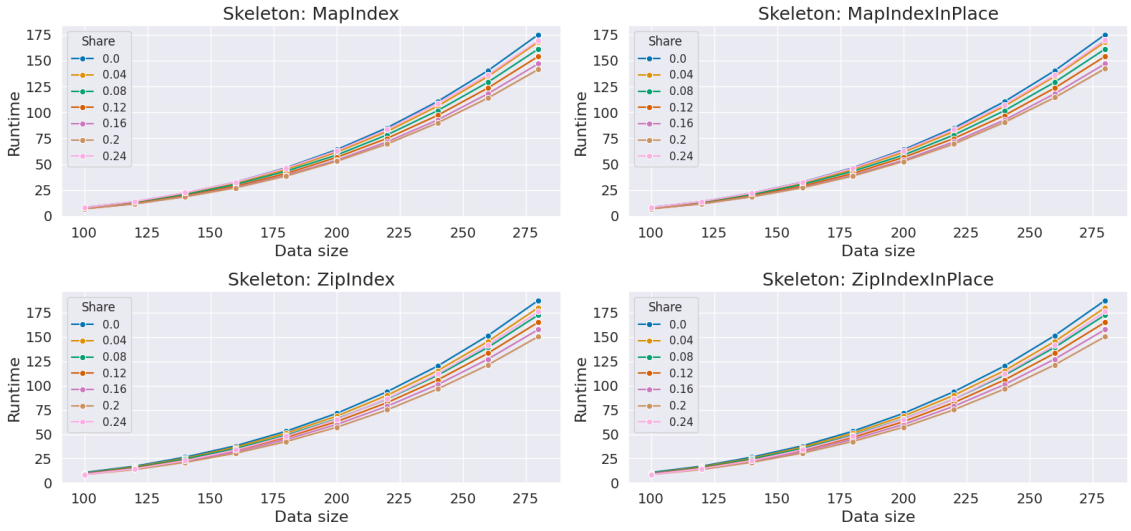


Figure 4.3: Runtimes (in seconds) of benchmark skeleton calls with a complex user function on the RTX2080 partition with varying percentual shares. The data structure size is the side length of the cube.

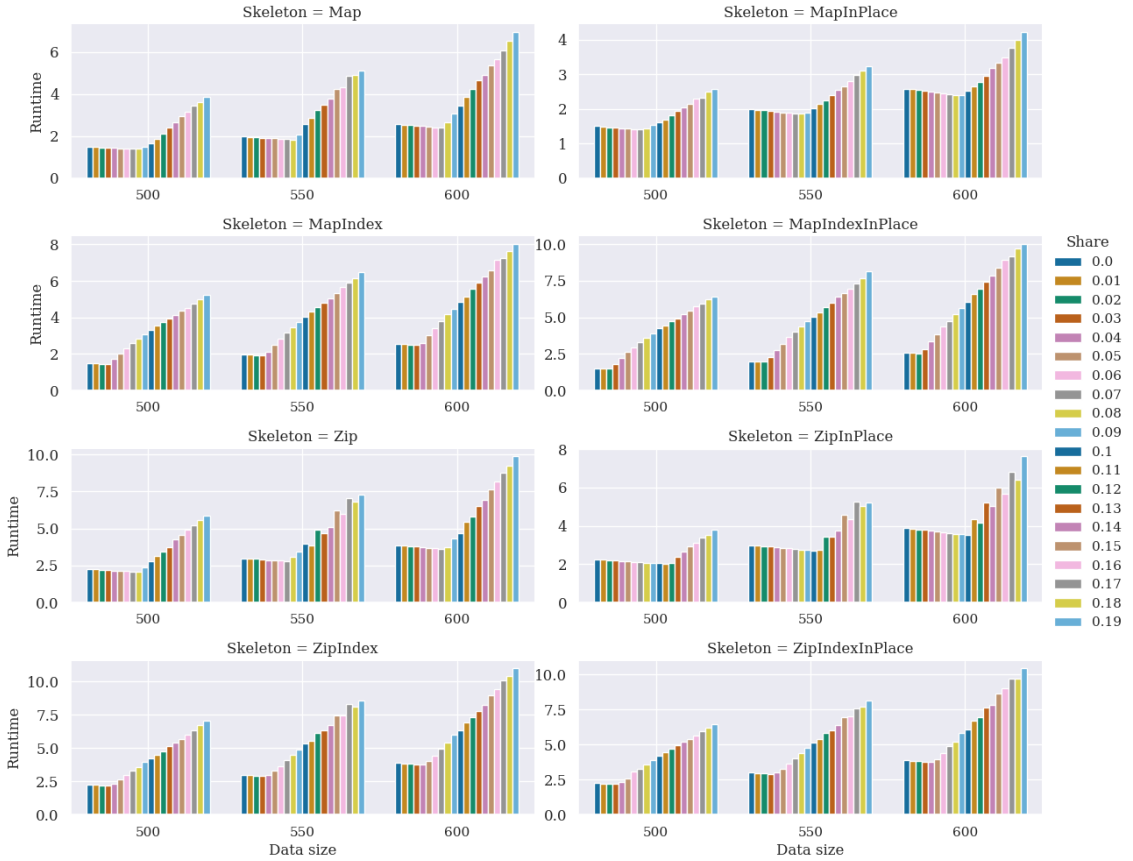


Figure 4.4: Runtimes in seconds of benchmark skeleton calls on a Nvidia A100 grouped by the size of the data structure with varying percentual shares.

With this adjusted program, a speedup of ~1.2 is reached for a data size of 280^3. The program runtimes are depicted in Figure 4.3. As the calculation is the dominating factor, the speedup for the other variants is similar. Also, for smaller data structures that previously did not profit from the CPU-share, a speedup of 1.2 can be achieved for *all* variants.

In contrast to the RTX2080 partition, the A100 partition can process bigger data structures and has more SM while having the same CPU. As the GPU became stronger and the CPU remained the same, the logical consequence would be that for the same program, the optimal CPU-share is lower. This assumption can be confirmed and is depicted in Figure 4.4. Note that the data sizes are the side lengths of a cube. This means the total number of elements processed is 125, 166,375 , and 216 million elements. The graph depicts the program variant with the simple user function. For variants not using the index, the optimum share varies between 7-12%. Noteworthy, the impact of this optimisation is smaller. In the optimal case, a speedup of approximately 1.06 can be achieved. Skeletons using the index of the data structure have an optimal share of

2%-4% with an optimal speedup of 1.03. Again, the only exception is the `Fold` skeleton. Although sharing calculation does not improve the overall runtime for the skeleton, only reducing it with the CPU speeds up the calculation by a factor of 5.

4.3.2 CPU Usage on a local PC

The first GPU tested is the GTX 750 Ti with five SM and compute capability 5.0. The architecture is rather outdated, as the GPU was released in 2014. All those characteristics should benefit offloading calculation to the CPU. The associated CPU has fewer cores, which is the only argument that should impede the effect. Due to the limited parallelism and memory, the size of the data structures and the number of repetitions were decreased in contrast to the program running on the cluster.

Table 4.2 shows an example from the analysed runtimes. It displays the runtimes for the `MapInPlace` skeleton for a parallel CPU, GPU, and mixed program. For this setup, a speedup of up to factor 4.5 can be found for a mixed variant compared to a GPU variant.

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

Table 4.2: Runtimes (in seconds) for the sequential (seq.) program, the CPU (OpenMP) program, the GPU program, and for the optimal mix of CPU and GPU for the `MapInPlace` skeleton on the GTX750 partition. The following columns show speedups of the optimal mix compared to the sequential, the CPU, and the GPU program.

The development of the runtime difference can be seen in Figure 4.5. All skeletons have been tested, but as the findings are repetitive, only an extract of the skeletons is displayed. The findings can be extended to the other skeletons. Not only for the `MapInPlace` skeleton but also for all `Index` and `InPlace` variants of the `Map` and `Zip` skeleton, it can be seen that having a share of 2% has a major impact on the runtime and the speedup. An exception is the `Fold` skeleton with an optimal share at 0.2%. Higher shares were omitted to increase the readability but can be seen in the Appendix Figure A.1. This proves that using weaker GPUs definitely benefits the presented feature. Interestingly, the `Fold` skeleton immediately profits from offloading the calculation. This trend can be observed until 20%. After that, the runtime increases again.

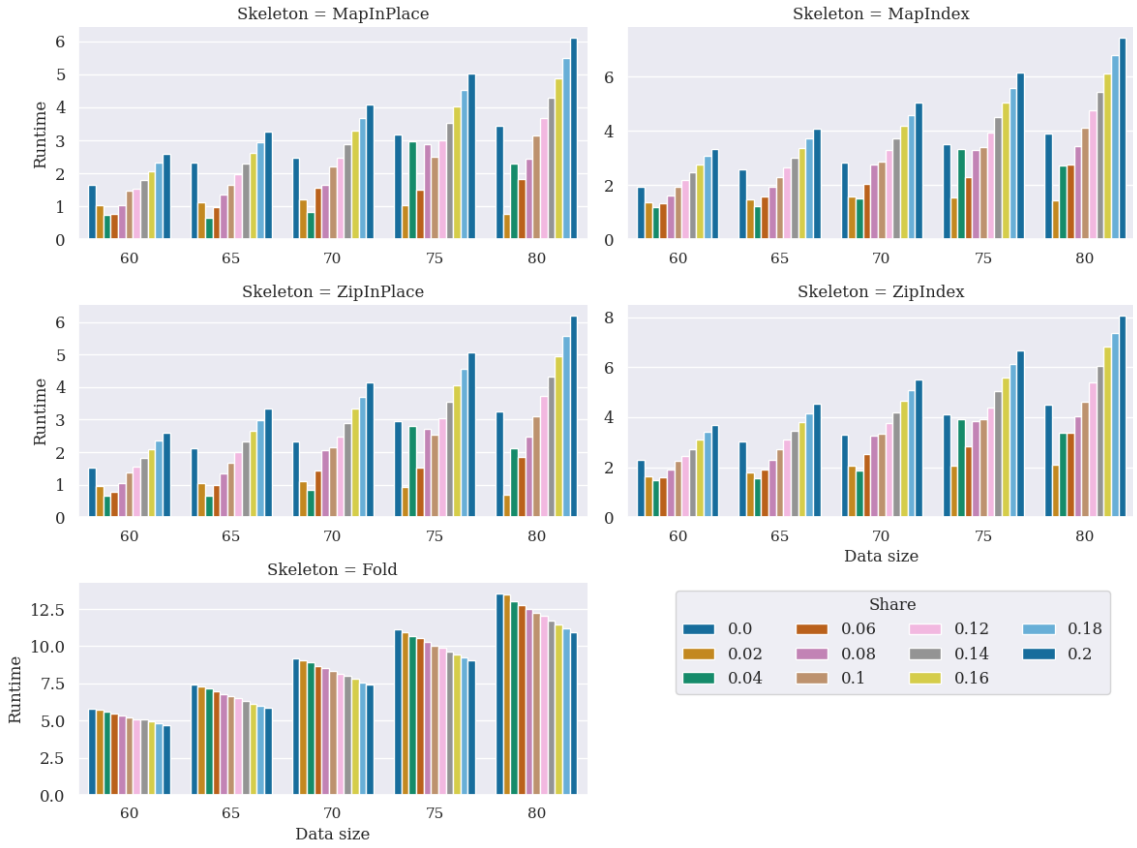


Figure 4.5: Runtimes (in seconds) for different CPU fractions calculated by the CPU for `Map` and `Zip` variants on a GeForce GTX 750 Ti.

GPUs used on local PCs as well as GPUs used in data centres have a steadily increasing number of cores. Therefore, it was decided to pursue the experiment with a more recent local GPU, a GeForce 3070 Ti, as a supplement experiment. As previously listed, the GPU has almost ten times as many SM and has four times as much memory. Therefore, the size of the data structures tested and the number of repetitions were increased. Moreover, the number of cores available on the CPU is higher (four compared to 14), increasing the computational power of the CPU. As the runtimes varied significantly when using a simple user function the runtimes of the complex user function are displayed. The equivalent of the previously presented graph can be seen in Figure 4.6. The depiction displays the CPU-shares up to 30% as the optimal runtime is higher than previously. The optimal percentage varies between 18% and 28% for the `Map` and `Zip` skeletons tested. All other variants can also be seen in the Appendix Figure A.2, but reflect similar results to the depiction displayed here. Interestingly, for this setup, no disadvantage for skeletons using the index could be found. In total, a speedup of 1.3 can be achieved for the skeletons `Map`, `MapInPlace`, `MapIndex`, `MapIndexInPlace`, `Zip`, and `ZipInPlace`. For `ZipIndex`

and `ZipIndexInPlace`, a speedup of 1.33 can be achieved. In contrast, the results for the `Fold` skeleton vary significantly. While the previous experiment clearly showed a benefit of including the CPU, this result could not be confirmed for the newer GPU. This might also be caused by the improved bandwidth; the GeForce GTX 750 Ti has a bandwidth of 86.4 GB/s, while the GeForce RTX 3070 Ti has a bandwidth of 608.3 GB/s. Not displayed in the graph is the runtime of the `Fold` skeleton for 100% CPU-execution. For this setup, the execution on the CPU is slower.

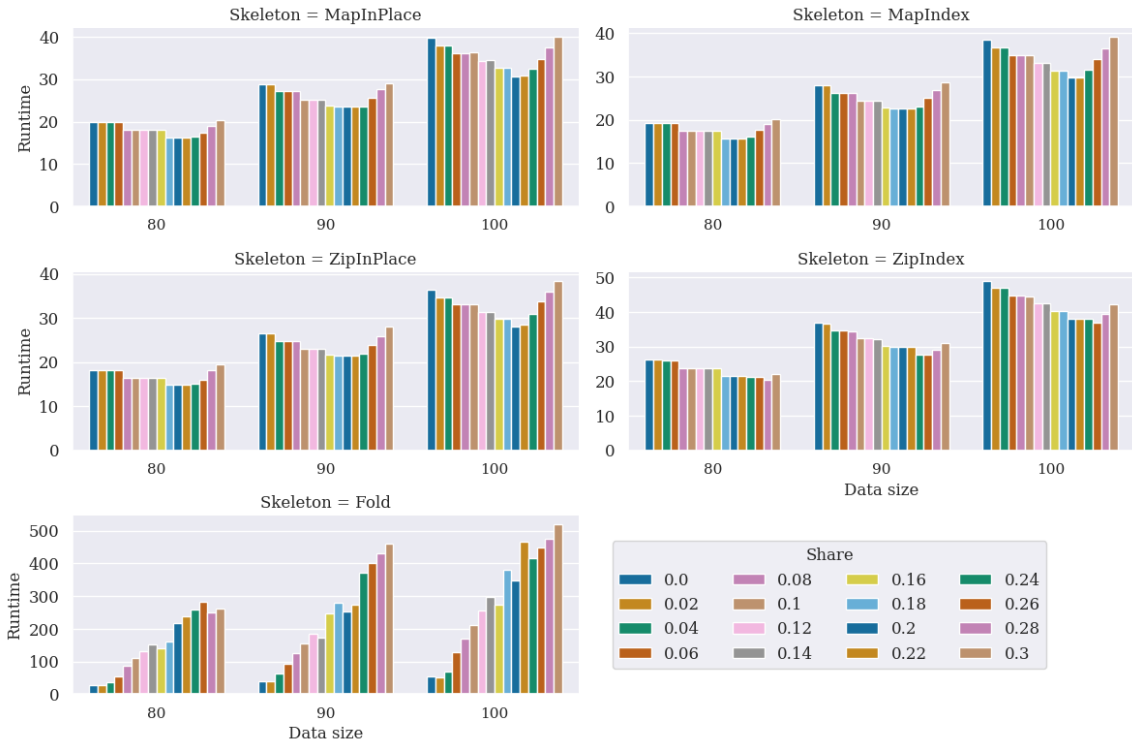


Figure 4.6: Runtimes (in seconds) for different CPU fractions calculated by the CPU for `Map` and `Zip` variants on a GeForce RTX 3070 Ti.

4.4 Conclusion and Outlook

From the finding, it can be concluded that for cluster environments and for modern local workstations, a minor speedup can be achieved for offloading calculations to the CPU. For cluster, this factor was around 1.2, and for a local setup with a more recent GPU 1.3. For example, a program which takes two days can save 8 hours with a factor of 1.2. Therefore, the feature is beneficial for calculations taking days, reducing the total runtime not only by seconds but by minutes or hours.

It could be verified that the impact of the speedup depends on the hardware used, the user function, and the data size. The data size can easily be read at compile time. Information on the capability of hardware can be read by comparing the maximum number of threads that can be started by OpenMP and by reading the available SM on the GPU with the `cudaGetDeviceProperties` function. Although the exact power depends on the architecture, it can serve as an indicator for comparing the strength of the computational units. However, the analysis of the user function would require an additional tool that measures the complexity. The creation of such a tool could be a topic for future research.

As it can not be guaranteed that a sufficiently good share can be found automatically, for now, calculations are not automatically shared between the GPU and CPU. All findings made are included in the variable documentation to advise users who might be looking for more optimisation potential.
This chapter is based on the publication:

- Nina Herrmann and Herbert Kuchen. 'Distributed Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments'. In: *International Journal of Parallel Programming* 51.2-3 (2023), pp. 172–185.

5

Ant-Colony Optimization

The efficient solution of combinatorial problems is an ongoing research topic, as those problems can be found in many real-world tasks. In this context, evolutionary heuristics, such as ACO, are particularly relevant as, commonly, the best solution is not required, but a solution that comes close to the optimum is sufficient. Although these solutions usually require less computational power than exact solutions, those calculations can consume a time with growing problem sizes. Self-explanatory, the runtime of meta-heuristics partly depends on choosing adequate parameters, which is the task of meta-heuristic researchers. This work only discusses the possible speedup by implementing parallel programs with high-level frameworks. As an exemplary meta-heuristic, ACO was chosen to be analysed for applicability and extends the research from [59].

The chapter is structured as follows: firstly, the ACO algorithm is explained in Section 5.1, afterwards related work is listed, including optimisation of the ACO algorithm (Section 5.2). Those sections build the base to explain the adjustments made to the algorithm in Section 5.3 and the implementation in Muesli and Musket (Section 5.4). Lastly, all implementations are evaluated regarding runtime and complexity (Section 5.5), and all findings are concluded in Section 5.6.

5.1 Algorithm

The origin of the ACO algorithm lies in the communication used between ants when searching for food. In reality, each ant searches independently, e.g. for food, and leaves a fragrance on the ground called a pheromone trail. The more ants walk on the same path, the stronger the smell of the pheromone trail. This communication mechanism is artificially transferred to ACO algorithms to solve optimization tasks. Noteworthy, the general steps described in the following paragraph refer to the first implementation, called the ant system. There exist variations of this system, such as the division of ants into multiple subgroups [18]. For our showcase, it is sufficient to use the original system, as it could achieve the same quality as other approaches [59]. After describing the generic steps, the algorithm will be applied to the TSP as an exemplary optimization problem.

Algorithm 1: Generic approach for the ACO algorithm

```
1 while !terminationCriteria do
2     for ant ← ants do
3         ant→generate_solution
4     evaluate_solutions
5     update_pheromones
```

Other problems have also been investigated but are not presented here as similar findings could be made [59]. Therefore, for this thesis, the implementation of one problem is discussed in depth.

The generic process of ACO algorithms can be divided into three major steps. Firstly, each ant generates a possible solution to the given problem (Algorithm 1 l. 1). The steps to find a solution are partly based on the pheromones of previous solutions favouring good solutions of earlier iterations, and a random factor is included to explore other parts of the solution space. After multiple solutions have been generated, a second step evaluates the given solution. As the best solution is usually unknown, solutions are primarily evaluated against each other and previously generated solutions, which impedes the evaluation of the significance of the improvement. Afterwards, the pheromones are updated depending on the quality of the solution(s). In some ant systems, all solutions are used, while in others, only the best solution is used. During the update process, a weighting factor determines how much influence the new iteration has. The factor can also be changed depending on the quality of the solution. Possible criteria to terminate the search could be a stagnation in the quality of the solutions or the maximum number of iterations reached.

Parameters that are optimised by heuristic researchers include the weighting of better solutions in calculating the pheromones, the probabilities in the solution-finding step to create more explorative solutions, and the number of ants. Moreover, depending on the problem, variations of the algorithm might be more suitable to search for solutions.

Applying the described steps to a concrete problem, the TSP problem, shows the difficulties in finding generic meta-heuristic approaches and the importance of a flexible framework to adjust to different algorithms. Chapter 3 *Ant Colony Optimisation Algorithms for the Traveling Salesman Problem* in *Ant Colony Optimization* gives an overview of possible algorithms and a qualitative evaluation of the different approaches [17]. In the given problem, the shortest path connecting all cities in one tour needs to be found. Each city can only be visited once, and a complete graph is assumed where each city

connects to each city. To solve this problem, each ant starts at a random initial city and chooses the next city depending on the existing pheromone deposit, the distance to the next city, and a random choice for the next city depending on a probability [17]. Commonly, pheromones are initialised with a value slightly higher than the expected amount of pheromones deposited by ants to prohibit sticking to the first tours constructed [17, p. 70]. More precisely, the probability $p_{i,j}$ of ant k to visit the city j when starting from city i at iteration t can be calculated as follows:

$$p_{i,j}(t) = \frac{[\tau_{i,j}(t)]^{\alpha} \times [\eta_{i,j}]^{\beta}}{\sum_{l \in \mathcal{N}_i^k} [\tau_{i,l}(t)]^{\alpha} \times [\eta_{i,l}]^{\beta}} \quad \text{if} \quad j \in \mathcal{N}_i^k. \tag{5.1}$$

The numerator consist out of $\tau_{i,j}$ being the pheromone at the edge from node i to j multiplied with $\eta_{i,j}$, which is $1/d_{i,j}$, and $d_{i,j}$ is the distance from node i to j. Therefore, the numerator gets higher for close cities with a lot of pheromones between them. α and β are the parameters to determine the weighting of the pheromones in contrast to the distance. Setting α to zero eliminates the influence of the pheromones, therefore increasing the importance of the distance and vice versa. The denominator relativises the nominator compared to all possible cities by dividing it by the sum of pheromones on the edges of all possible cities that can be visited from ant k and city i. Therefore, $\mathcal{N}_i^k$ contains all unvisited nodes that are candidate nodes for the next step. The pheromone update can be specified as follows:

$$\tau_{i,j}(t+1) = (1-p) \times \tau_{i,j}(t) + \sum_{k+1}^{m} \Delta\tau_{i,j}^{k}(t) \tag{5.2}$$

The existing pheromones evaporate by multiplying them with an evaporation rate $(1-p)$ where $0 < p < 1$. Higher rates favour new solutions. The second part of the equation calculates the sum of all pheromones deposited by every ant for the city combination where $\Delta\tau_{i,j}^{k}(t)$ is either 0 in case the city considered is not visited in the defined sequence or $\frac{1}{L^k(t)}$ where $L^k(t)$ is the length of the route constructed by ant k. Ants that generate shorter tours deposit more pheromones, increasing the probability of creating tours using the paths in the next iteration. The probabilistic way allows the creation of diversity in the solution space.

[17] Some algorithms might adjust this behaviour, it is one possible solution to the problem. Please take a look at [17, p.67-82]

5.2 Related Work

Already in 2004, Levine and Ducatelle applied ACO to solve the Bin Packing Problem (BPP) and Cutting Stock Problem (CSP) problem. They outperformed other evolutionary approaches in particular classes of problems. Especially for big problem instances, they highlight the importance of applying heuristics, as finding the best solution is impossible in a reasonable amount of time. However, their focus was on optimizing the algorithm and neglecting the parallelisation of the program. However, this would be beneficial, as generating more solutions enhances the outcome. [52]

Parallel implementations of the ACO algorithm have been investigated several times with an overview provided by Pedemonte, Nesmachnow, and Cancela [68]. The overview is outdated as it was published in 2010, and only a minority of the mentioned work includes accelerators such as GPUs. Interestingly, those parallelisations sometimes also include adjustments to the details of the algorithm to improve data parallelism. For example, Cecilia et al. introduced an I-roulette mechanism holding the closest cities to reduce the initial search space [10]. Delévacq et al. have compared multiple parallelisation strategies. Unfortunately, they neither provide their source code nor the absolute runtimes [14].

This research extends the aforementioned work as it shows that with high-level frameworks, a similar runtime to low-level frameworks can be achieved, simplifying the program's creation and allowing the specialist to adjust the program depending on their needs. Self-evidently, many other approaches provide high-level parallel programming frameworks. However, their applicability must be proven. To the best of our knowledge, besides the work done in Muesli and Musket [59], [93], [73], no skeleton high-level framework comes with an example for metaheuristics itself. Wrede, Menezes, and Kuchen presented an implementation of the Fish School Search and the Particle Swarm Optimisation algorithm in the DSL Musket [94] and Rieger, Wrede, and Kuchen presented an implementation in Muesli [73]. This work is continued by presenting a ACO implementation in Muesli in Section 5.4.

5.3 Parallel Implementation

Deriving an efficient parallel implementation from equations is not straightforward, as multiple data structures need to be read. The parallelisation strategies presented here have been previously introduced [60], [59]. During the implementation, it was assured that the quality of the solutions stayed consistent, meaning that with multiple runs tested, the average of the shortest route did not vary significantly. Independently of the

parallelism strategy, the data containing the cities and the distances of the cities is read. Noteworthy, the distance of the cities is calculated in parallel. Additionally, the x closest cities are calculated and stored in a separate data structure to implement the previously explained I-roulette mechanism [10]. Those calculations only occur initially, so they do not significantly influence the runtime. More importantly, the steps executed for each iteration need to be executed in parallel.

5.3.1 Course-Grained Parallelism

The most apparent parallelisation strategy is to let one thread construct one valid tour, therefore representing one ant. The steps necessary for a single ant to calculate a complete tour are depicted in Algorithm 2. It is assumed that random numbers can be generated without specifying the method used for generating those numbers. Furthermore, multiple data structures must be accessed to calculate the current tour. *closecities* holds the x closest cities for every city. Every ant starts with a random city (l. 1). Firstly, for this city, the probability of going to any of the x closest cities is calculated (ll. 5-11). Two factors significantly influence the likelihood of visiting the city: the distance to the current city and the pheromones deposited on the edges between cities in previous iterations. The two factors can be weighted by changing the variables β and α (ll. 5+6). If the city has been visited previously, both values are set to zero, and the city should not be revisited. The variable sum tracks the overall probability and is later used to relativise the probabilities of visiting the city (l. 12). With this approach, $p_{i,j}$ from the previous Equation 5.1 is calculated. To find the next city for the route, the city is partly randomly chosen by iterating over the x closest cities (ll. 18-22). As soon as the summed-up probability is higher than a random value, the city is chosen to be the next city, and it is written to the routes data structure (l. 24), and the process is repeated to find the next city in the tour. In case the random number generated is allowed to be higher than the sum of probabilities, it can happen that all surrounding cities of the x closest cities have already been visited. Then, an arbitrary city that has not been visited is chosen (ll. 24-26). Therefore, the range for the random number generated influences the probability of exploring completely random cities.

The downsides of this approach are the following:

1. As the L1 and L2 cache available on a GPU is limited, loading multiple data structures into the kernel increases the time for memory accesses. The increasing number of cities strengthens this effect.

Algorithm 2: Pseudo-code for the course-grained parallel tour construction

```
1  c = randomcity;
2  while i < ncities do
3      foreach city_j in closecities do
4          if city_j not visited then
5              eta[city_j] = pow(1/dist[c, city_j], β) ;
6              tau[city_j] = pow(1/phero[c, city_j], α) ;
7              denominator += tau[city_j] * eta[city_j] ;
8          else
9              eta[city_j] = 0 ;
10             tau[city_j] = 0;
11     foreach city_j in closecities do
12         prob[city_j] = tau[city_j] * eta[city_j]/denominator ;
13     if denominator ≠ 0 then
14         Generate a random number r ;
15         sumprob = 0.0 ;
16         foreach city_j in closecities do
17             sumprob+ = prob[city_j] ;
18             if sumprob > r then
19                 c = closecities[city_j] ;
20                 break
21     else
22         foreach cc in ncities do
23             if cc not visited then
24                 c = cc ;
25     routes[i] = c ;
26     i++ ;
27     denominator = 0.0 ;
```

2. The repeated conditional statements lead to thread divergence, slowing down the execution of the kernel.
3. As the number of threads started in parallel is equivalent to the number of ants, the possible parallelism is not exploited. Starting 139.264 ants is an unreasonably high number, as increasing the number of ants at a certain level does not significantly improve the quality of the solution.

5.3.2 Fine-Grained Parallelism

As the problems are quite obvious, an alternative approach to calculate the route was proposed by Menezes et al. [60]. They propose a different approach using a block of threads representing one ant. A block can contain up to 1024 threads, enabling parallel execution of the for-loops mentioned previously. Algorithm 3 abstracts from the details presented in Algorithm 2 to calculate the η and τ values. Importantly, as one ant is presented as one block, all threads within this block calculate the η and τ values in parallel. This means a thread does not calculate all values, but *ncities*/*threads* cities are calculated by one thread, demagnifing the effect of the first problem stated. This approach requires a shared variable sum written in an atomic manner. Afterwards, the threads are synchronised, and one thread calculates the sum (l. 8). The probability calculation is again split between the threads (l. 13). The final steps for the tour construction use the previously shown procedure and are only executed with a single thread per block.

On the one hand, the approach has the advantage of parallelising the calculation for loops. On the other hand, repeated data access to the same data structure from different threads slows down memory access. Checking for already visited cities requires each thread to check the same data structure. Moreover, the additional synchronisation steps also limit the possible parallelism. Lastly, an important part, namely selecting the next city dependent on the probabilities, is still executed on a single-thread basis.

5.3.3 Pheromone Deposition

Independently of the mechanism used to construct a tour, the pheromone has to be calculated. For this purpose, the distance of the tour is calculated by adding up the distances between the cities for each tour constructed. From all distances, the shortest route is calculated. Those steps are skipped in the description as the reduced operation is trivial. More importantly, the deposition of pheromones has to be discussed as the operation was not executed in parallel beforehand [59].

Algorithm 3: Pseudo-code for fine-grained parallelization [60]

```
1  ant_i = blockIdx.x
2  threadIdx.x
3  while i < ncities do
4      foreach city_j in closecities do
5          calculate η and τ /* Algorithm 2 (l.4-10)                    */
6      Synchronize
7      if thread.Idx == 0 then
8          calculate sum(ant_i)
9      Synchronize
10     foreach city_j in closecities do
11         prob[city_j] = τ[city_j] * η[city_j]/sum
12     calc_probability(η, τ, sum)
13     synchronize
14     if threadIdx.x == 0 then
15         chose_next_city(ant_i) /* Algorithm 2 (l.14-23)               */
```

To deposit pheromones, firstly, for every step of every route, the amount of pheromones is saved in the data structure *deltapheromones* where the value depends on the length of the tour constructed (Algorithm 4 $l.9 + 10$). The data structure *routes* contains the valid constructed tours and *dist_routes* the length of each tour. The variable Q can be adapted to either favour that connection exists with a high value or favour the length of the route by having a low value.

Afterwards, the pheromone is calculated by evaporating the previous calculation with the factor RO satisfying $RO > 0 \wedge RO < 1$ (l. 12). A high value favours new solutions. Although calculating the pheromone does not consume most of the time, it is reasonable to analyse if suitable skeletons exist in Muesli to execute at least part of the pheromone deposition in parallel.

5.4 Muesli Implementation

The existing implementations of the course- and fine-grained implementation are accessible on Github [18]. This includes low-level implementations of both strategies with GPU and CPU parallel code. Moreover, it includes a Musket implementation. More explanation on the previous implementation can be found in the publication in Chapter

[18] https://github.com/NinaHerrmann/ant-colony-optimization-project

Algorithm 4: Pseudo-code for pheromone deposition

```
Data: dist_routes
foreach i in nants do
    rlength = dist_routes[i]
    foreach j in ncities do
        cityi = routes[i * ncities + j]
        if j + 1 < ncities then
            cityj = routes[i * ncities + j + 1]
        else
            /* Calculate the distance from the last city to the first.    */
            cityj = routes[i * ncities]
        deltapheromones[cityi, cityj] += Q/rlength
        deltapheromones[cityj, cityi] += Q/rlength
foreach j in ncities * ncities do
    phero[j] = RO * phero[j] + deltaphero[j]
```

9 [59]. In contrast, this section presents a Muesli implementation. All algorithms and listings are reduced to the most relevant information; the interested reader can see the entire program online [19] or in Appendix A.1.

A prominent characteristic of the given problem is that the next city within one route needs to be calculated dependent on the *previous* city. Depicting this dependency is challenging as most data-parallel skeletons are executed on a single-element basis. Dependencies, as in the `MapStencil` skeleton, do not require the subsequent calculation but reading surrounding values. The route calculated during the tour construction needed to be calculated sequentially.

In contrast, to Musket Muesli can make use of C++ `structs`. Therefore, for representing the tour a `struct` called `tour` is used, with an integer pointer representing the tour and a double saving the route length. A noticeable slowdown of the route calculation in the given program is that checking if a city has been visited is, in the worst case, in the runtime class $O(n)$. This can be optimised by changing the data structure. Instead of saving the tour as a time series, the cities are saved as keys and point to the order of their route. For example, the array `[[0→1], [1→-1], [2→0], [3→-1]]` would represent an incomplete tour where city 2 is the initial city followed by city 0 and city 1 and 3 have not been visited. With this data representation, checking if a city has been visited can be done in $O(1)$. As tours were not presented this way previously, the consequences of the adjustment will be explained in the following paragraphs.

[19] `https://github.com/NinaHerrmann/muesli4/blob/cube_map_stencil/examples/aco_tsp_dataset.cpp`

Firstly, we do not calculate the η and τ values in every tour construction but create a DM which stores the product of *eta* and *tau* for every city combination. This calculation is embarrassingly parallel for skeletons as merely the distance and the pheromone need to be called with a `Zip` skeleton to create the new data structure. The user function can be seen in Listing 5.1.

```
1 MSL_USERFUNC double operator()(int from, int next, double dist, double phero) const override {
2     double eta_tau = 0;
3     if (from != next)   // For every unvisited city, calculate the eta and tau value.
4         eta_tau = pow(1/dist, 2) * pow(phero, 1); // assuming 2,1 for alpha and beta
5     return eta_tau;
6 }
```

Listing 5.1: $\eta \times \tau$ calculation as a user function in Muesli implemented with a `Zip` skeleton.

Given the calculated $\eta \times \tau$ values, each ant can start to calculate an independent tour. The concrete construction of a route can be seen in Listing 5.2 and follows the course-grained approach. Implementing the fine-grained approach is hardly realisable due to the synchronisation operations. However, the effect of the missed parallisation is minimised as the η and τ calculation was moved to the previous skeleton. The user function starts by generating a random start city (ll. 2+3). Noteworthy, l. 4 already shows the more efficient way of saving the route, namely saving the city's key for the order. To calculate the next city to be visited, the sum of all $\eta \times \tau$ values of cities belonging to the 32 closest cities that have not been visited is calculated (ll. 9-13). If any of the 32 closest cities have not been visited, the condition $etaTauSum > 0$ is executed (l. 14), and a city from the data set of closest cities is chosen. A random number between 0 and the sum of the $\eta \times \tau$-values is generated to include randomness. The probability of visiting the city is encoded by choosing the city that increases the sum of all probabilities above the randomly chosen number (ll. 15-23). The higher the probability of visiting the city, the more likely the conditional statement in line 21 will be triggered. In case none of the closest cities can be reached, a random city is chosen (l.25), and all consecutive cities are checked. The first city that has not been visited is chosen. The function returns a `struct` of type tour with the calculated distance and the integer pointer with the order of cities visited.

Finding the minimal route is trivial as a functor is required, which takes two tours and returns the tour with a smaller distance. Afterwards, the pheromone needs to be updated (Listing 5.3). As previously explained in Equation 5.2, the δ-pheromone depends on the length of the routes. Therefore, for each city combination, it needs to be checked if the constructed route includes a step where either $city_i$ is directly followed by $city_j$ or $city_j$

is directly followed by $city_i$ (l. 8). The optimal value for Q needs to be adjusted by the heuristic expert, as the value affects the influence short tours have on the pheromone.

```
MSL_USERFUNC T operator()(int ant_index, T build_tour) const override {
    MSL_RANDOM_STATE randomState = msl::generateRandomState(this->seed, ant_index);
    int fromCity = msl::randInt(0, ncities - 1, randomState);
    build_tour[fromCity] = 0;
    double distance = 0;
    for (int i = 1; i < ncities; i++) {
        int nextCity = -1;
        double etaTauSum = 0;
        for (int j = 0; j < IROULETE; j++) {
            int toCity = iroulette[fromCity * IROULETE + j];
            if (build_tour[toCity] == -1)
                etaTauSum += etataus[toCity * ncities + fromCity];
        }
        if (etaTauSum != 0) {
            double rand = msl::randDouble(0.0, etaTauSum, randomState);
            double etaTauSum2 = 0;
            for (int j = 0; j < IROULETE; j++) {
                nextCity = iroulette[fromCity * IROULETE + j];
                if (build_tour[nextCity] == -1)
                    etaTauSum2 += etataus[nextCity * ncities + fromCity];
                if (rand < etaTauSum2)
                    break;
            }
        } else {
            int startCity = msl::randInt(0, ncities - 1, randomState);
            for (int j = 0; j < ncities; j++) {
                if (build_tour[(startCity + j) % ncities] == -1) {
                    nextCity = (startCity + j) % ncities;
                    break;
                }
            }
        }
        build_tour[nextCity] = i;
        distance += distances[nextCity * ncities + fromCity];
        fromCity = nextCity;
    }
    build_tour.setDist(distance);
    return build_tour;}
```

Listing 5.2: Tour construction user function in Muesli. `iroulette` stores the 32 closest cities, `etataus` the previously values calculated and `distance` the distance of every city combination.

The proposed way of calculation has the disadvantage that the DA *routes* needs to be copied on all computational units for read access. This might result in memory contention due to the concurrent read operations. However, in this context, in contrast to the previous approach, not every route step needs to be checked, but the route can be accessed with the city as a key. The calculation of the final pheromone is trivial as it computes the sum of the calculated δ-pheromone and the pheromone calculated in previous iterations. Then, the sum is reduced by an evaporation rate (l. 12).

```
1  MSL_USERFUNC double operator()(int row, int column, double prevphero) const override {
2      double RO = 0.5;
3      int Q = 11340;
4      double deltaphero = 0.0;
5      for (int k = 0; k < nants; k++) {
6          auto rlength = routes[k].getDist();
7          int cityi_VisitIndex = routes[k][row];
8          int cityj_VisitIndex = routes[k][column];
9          if (abs(cityi_VisitIndex - cityj_VisitIndex) == 1)
10             deltaphero += Q / rlength;
11     }
12     return (1 - RO) * (prevphero + deltaphero);}
```

Listing 5.3: Pheromone calculation in Muesli

All the described steps are again displayed in Algorithm 5, showing the skeleton calls. Noteworthy, the skeletons used to initialise the data structures (e.g. the calculation of the distances) are skipped. The functor `fill` resets all routes to point at index -1 for all cities (l. 2). Afterwards, the η and τ values are calculated.

As every following step requires the product of those values, the data structure *etatau* stores the product for every city combination (l. 3). The data structure is, therefore, twice bigger than needed, and nearly all values are stored twice. Storing it in a redundant way simplifies read access as values can be read in parallel, and the calculation takes only a minority of the runtime. The functor `tourConstruction` requires read access to the data structures `iroulette`, `etatau`, and `distance`. Therefore, those data structures are passed as parameters to the function (l. 4). Noteworthy, this requires gathering the data structures from all nodes in case multiple nodes are used. Also, self-evidently writing to data structures passed as arguments is not supported as there is no reasonable way to conjunct those.

Afterwards, the minimum route of the current iteration is found by folding the data structure containing the calculated routes, and the result is compared to the best result of the previous iterations. Calculating the pheromone requires reading the routes

data structure and, inside the user function, firstly, the δ-pheromone is calculated, and afterwards, the sum of the δ-pheromone, and the previous pheromone is evaporated.

Algorithm 5: Muesli skeletons used to parallelize the iterative steps of the ACO algorithm for solving the TSP problem

Data: etatau<double>(*ncities* × *ncities*), phero<double>(*ncities* × *ncities*), distances<double>(*ncities* × *ncities*), iroulette<int>(*ncities* × 32), routes<*route*>(*nants*)

```
1  foreach i in iterations do
2      routes.mapInPlace(fill);
3      etatau.zipIndexInPlace3(distance, pheromone, etataucalc);
4      tourConstruction.setIterationParams(iroulette, etatau, distance);
5      routes.map(tourConstruction);
6      minroute = routes.fold(min);
7      if minroute < alltimeminroute then
8          alltimeminroute = minroute;
9      updatephero.setIterationsParams(routes);
10     phero.mapIndexInPlace(updatephero);
```

5.5 Results

The most important aspects to be tested are the runtime and complexity of the programs. To test the runtime, the handwritten program, the Musket program, and the Muesli program have been executed on a node equipped with a GeForce RTX 2080 Ti and a Zen3 (EPYC 7513) CPU. All times depicted are the average of at least 25 runs. The data sets tested are publicly available [20]. The naming scheme of the datasets indicates the number of cities to be visited. Unfortunately, no dataset that scales linearly could be found. Subsection 5.5.1 summarises the findings of comparing the handwritten implementation and the Musket implementation [59]. Afterwards, Subsection 5.5.2 compares the Muesli implementation to the previous results. All programs are single GPU-programs; however, it should be kept in mind that Muesli is capable of scaling across multiple nodes and GPUs.

[20] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/` Accessed 15.03.2024

5.5.1 Musket Implementation

The previous work showed that the Musket program is as fast as the previously discussed implementations in [60]. For the low-level implementation, no significant difference between the fine-grained and the course-grained tour construction could be found; therefore, merely the course-grained construction is displayed. Figure 5.1 and Table 5.1 summarises the results obtained when comparing the course-grained tour construction from a low-level implementation and the Musket implementation. The results show that for smaller problems, both implementations achieve very similar, almost identical, runtimes. With a growing number of cities and an increasing number of ants, the low-level program achieves slightly shorter runtimes. The values for the most extensive map tested (pr2392) are excluded in this graph since they impede the readability. They can be seen in Table 5.1 and confirm the assumption that the low-level program scales better. For example, when solving the biggest problem instance, the low-level implementation is 0.4% faster with 1024 ants, 3.9% with 2048 ants, 7.9% with 4096 ants, and around 7.9% with 8192 ants. While we can not prove the exact reason for the stagnation, it is most likely that the difference is caused by the different number of threads that started for the kernel calls. While the low-level implementation used 256 threads per block, Musket uses 1024 threads per block. This benefits the execution of smaller ant colonies as those are distributed on multiple SM.

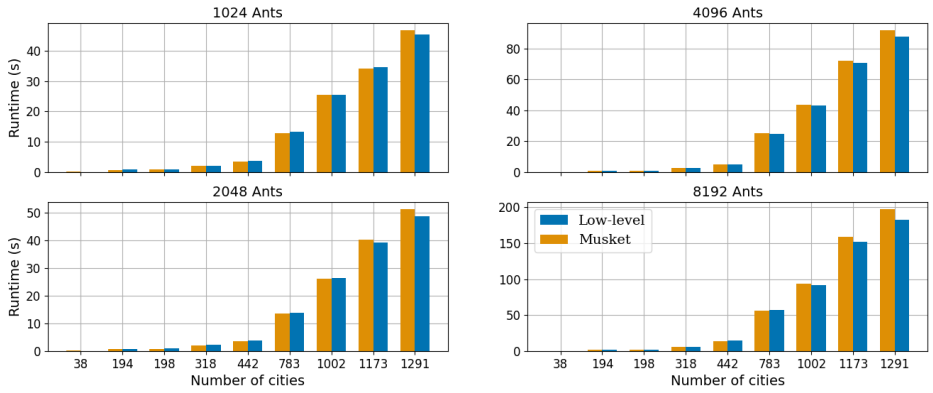


Figure 5.1: Comparison of the execution times of the Musket program (yellow) and the low-level program (blue) on the GeForce RTX 2080 Ti.

Noteworthy, the runtime is not the only metric to compare low and high-level programs. Additionally, lines of code (LOC) are measured. We are aware that neither measurement directly reflects the effort to use the presented approaches. However, it is commonly used as it provides a sufficiently good indicator and is relatively easy to measure. Importantly,

Problem	Ants							
	1024		2048		4096		8192	
	LL	*Musket*	LL	*Musket*	LL	*Musket*	LL	*Musket*
dji38	**0.103**	0.186	0.106	0.189	0.107	0.192	**0.158**	0.199
cat194	0.890	**0.782**	0.896	0.797	1.022	0.938	**1.742**	1.789
d198	0.906	**0.852**	0.914	0.861	1.037	1.001	**1.784**	1.804
lin318	2.217	**2.103**	2.250	2.103	2.726	2.665	**6.175**	6.261
pcb442	3.711	**3.464**	3.783	3.474	5.022	4.961	14.318	**14.224**
rat783	13.365	**12.870**	13.766	13.621	24.651	25.191	57.208	**56.407**
pr1002	**25.409**	25.529	26.307	26.116	43.194	43.657	**91.539**	93.51
pcb1173	34.680	**34.201**	39.201	40.23	70.605	72.321	**151.416**	158.965
d1291	**45.37**	46.744	48.832	51.306	87.808	91.888	**182.741**	197.17
pr2392	**251.412**	252.648	292.913	304.26	428.534	462.256	**899.670**	969.704

Table 5.1: Comparison of execution times for a handwritten program (LL) and the Musket program on the GeForce RTX 2080 Ti.

especially LOC can not reflect the time and effort necessary to implement a parallel program as it can not be assumed that all program lines are equally complex. High-level approaches do not only reduce the LOC but relieve the programmer from challenging decisions, e.g. the allocation of memory spaces and the choice of a number of threads, making single lines easier to implement. For those measurements, every operation related to the data import is excluded from the comparison. In Musket 215 LOC in contrast to 374 LOC in the low-level implementation are required to write a program solving the TSP reducing the lines needed by 43%. This difference is mainly caused by eliminating copy instructions between CPU and GPU and the missing thread calculations. Moreover, the Musket program has a cyclomatic complexity of 54 compared to a slightly higher complexity of 67 for the low-level implementation.

Finishing the analysis, the time spent on different parts of the program was analysed. The split is depicted in Figure 5.2, leaving it without a doubt that the tour construction consumes most of the time. Even for the smallest dataset tested (38 cities), the route calculation takes approximately 80% of the time for 1024 ants and 90% for 8192 ants. For more extensive datasets, the calculation takes around 98% of the time. This outlines the importance of improving the route calculation. Also important to mention is that the total time displayed includes the time to copy the data to the device and initialise data structures, which often takes a noteworthy amount of time.

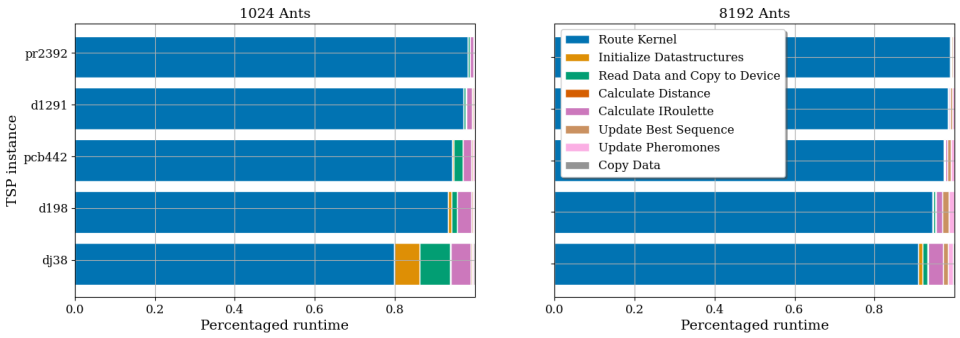


Figure 5.2: Percental runtime of operations of parallel implementation of the ACO algorithm for solving the TSP problem.

5.5.2 Muesli Implementation

The benefit of the redesigned tour construction is conspicuous. As previously explained, the advantage of using C++ `structs` enables the reduction of the algorithm's complexity. Concrete numbers can be seen in Table 5.2. Noteworthy, the Muesli program is also compared to the sequential Muesli program executed on a Skylake (Gold 6140) CPU.

Already, for a small area with 38 cities and 1024 ants, a small speedup of 1.88 can be achieved in contrast to the Musket implementation. This speedup improved significantly, up to 973, for the biggest city and ant colony combination tested. Strikingly, the Musket implementation is even slower than the sequential optimised program for the city pr2392. This emphasises that it was more important to restructure the parallelism than to execute

City	# Ants	Seq. Program	Muesli-Program	Menezes et al.[59]	Speedup (Muesli/Seq.)	Speedup (Muesli/[59])
d38	1024	0.0757	0.0933	0.1757	0.8114	1.8832
	2048	0.1323	0.1315	0.326	1.0061	2.4791
	4096	0.2436	0.217	0.6412	1.1226	2.9548
	8192	0.6262	0.3498	1.2845	1.7902	3.6721
pcb442	1024	4.4647	0.5793	4.6862	7.7071	8.0894
	2048	8.7077	0.6134	8.1128	14.1958	13.226
	4096	16.5164	0.7089	14.815	23.2986	20.8986
	8192	42.9367	0.8708	28.0945	49.3072	32.2629
pr2392	1024	100.6972	7.8075	1885.6974	12.8975	241.5238
	2048	213.6493	8.5712	3662.4705	24.9264	427.2996
	4096	614.0565	10.5288	7151.7166	58.3216	679.2528
	8192	2158.3639	14.411	14028.3833	149.772	973.4497

Table 5.2: Runtimes for the sequential Muesli program, the parallel GPU-Muesli program, and the program provided from Menezes et al. [59].

the program in parallel. Also, in contrast to the sequential program, a speedup of 150 can be reached. As previously discussed, the speedup is limited in case a small number of ants is started, as the parallelism that can be expressed in the tour construction is limited.

As the runtimes are clearly superior, the next evaluation criterion is the LOC. Similar to the previous approach, all operations, including reading of the data, the validation of the route and similar operations, are excluded from the measurements in the Muesli program. The Muesli program has 250 LOC in contrast to 215 (Musket) and 374 (low-level). Although the Musket program is shorter, the runtime advantage and the diversity of implementation options clearly favour Muesli as a high-level framework.

5.6 Conclusion and Outlook

It could be proved that Muesli is suitable for implementing the ACO algorithm. In contrast to another high-level framework Muesli could achieve a major speedup as it provides different data structures. More precisely, the use of C++ structs allowed us to search for already visited cities in $O(1)$. This provided a speedup of ~ 970 for a data set of 2392 cities and 8192 ants. This improvement might not hold true for other problems but strengthens the advantages of Muesli. However, this work did not evaluate if a programmer without knowledge of parallel programming would implement the same program. Therefore, future research that examines the realisations of complex algorithms should also evaluate their usability. As two meta-heuristics have been exemplified, future research should explore other areas of algorithms.
This chapter is based on the publication:

- Breno Menezes, Nina Herrmann, Herbert Kuchen, and Fernando Buarque de Lima Neto. 'High-level parallel ant colony optimization with algorithmic skeletons'. In: *International Journal of Parallel Programming* 49.6 (2021), pp. 776–801.

6

Stencil Operations

Stencil operations are present in many applications. This includes the broad field of numerical simulations (e.g., computational fluid dynamics), partial differential equations, image processing and computer vision, and machine learning. These fields typically have big data sets and operations that are complex enough to require parallelism. The varying parameters cause the difficulty of efficiently parallelising stencil operations. These vary in the size of the stencil, which influences the optimal memory allocation and, the number of memory accesses and the complexity of the function. The last two might favour, e.g., shared memory or tiling. A high-level skeleton for the `MapStencil` operations has to be generic enough to realise all applications while not losing significant performance due to the varying parameters. The following sections will first list related work on high-level approaches to stencil operations, as the Muesli implementation competes with multiple other approaches. Secondly, the user interface (Section 6.2) in Muesli will be explained, followed by the backend implementation in Section 6.3. Lastly, the implementation is evaluated in Section 6.4.

6.1 Related Work

Figure 6.1 displays a simple two-dimensional example of a stencil operation. The element to be calculated is depicted in red and depends on elements inside the surrounding 3 × 3 square (yellow), including the considered stencil (depicted in blue). Related work is defined as any framework that can abstract from the calculation performed and the access pattern. It allows users to use the high-level framework to implement an arbitrary two- or three-dimensional stencil operation and generate a parallel program.

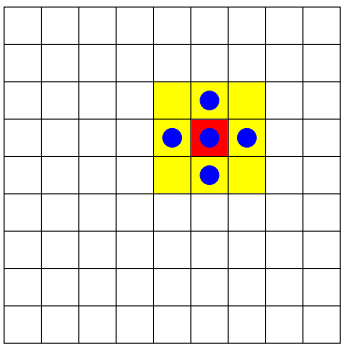


Figure 6.1: Exemplary MapStencil pattern.

The ongoing discourse on high-level strategies for stencil operations is a topic of multiple frameworks. This diversity can be observed by analysing various factors, including the types of accelerators employed,

Name	Single GPU	Multi-GPU[21]	Generic Skeletons	publicly available	updated
Celerity	✓	✓	✓ [22]	✓ [86]	until now
DUNE [4]	X	X	X	✓ [85]	until now
Lift [34]	✓	X	✓	✓ [84]	until now
EPSILOD [9]	✓	✓	X[23]	✓ [90]	until now
ExaStencil [49]	✓	✓	X	✓ [28]	until now
OpenACC Extension [69]	✓	X	✓ [24]	X	2017
PARTANS [56]	✓	✓	X	X	2013
PSkel [70]	✓	X	X	✓ [13]	2017
Palabos [50]	✓ [25]	X	X	✓ [81]	until now
SkelCL [77]	✓	✓	X	✓ [76]	2016
SkePU [24]	✓	✓ [26]	✓	✓ [23]	until now
WaLBerla/lbmpy [6, 5]	✓	X	X	✓ [30, 29]	until now

Table 6.1: Overview of frameworks that generate parallel code for stencil operations.

the range of operations supported, and the maintenance status of the frameworks. For a comprehensive overview, Table 6.1 presents an overview of research endeavours focused on the high-level parallelisation of stencil operations to the best of the author's knowledge.

SkePU3, for instance, is tailored for multi-GPU environments and can target multi-node environments in combination with StarPU for most skeletons. However, it's handling of stencil operations (termed MapOverlap) falls short in facilitating data exchange between programs in multi-node setups [24]. Celerity is a high-level approach, emphasising parallel for-loops, including stencil operations [80]. Lift adeptly manages n-dimensional stencils on single GPUs but lacks support for combining multiple nodes and accelerators [34]. SkelCL, while implementing a MapOverlap skeleton for multiple GPUs, lacks support for multiple nodes and has not seen updates since 2016 [77]. The developers of WaLBerla, a block-structured framework, extensively discuss optimisation options, yet its experiment section overlooks the utilisation of multiple GPUs [6]. EPSILOD tests prototypes for stencil operations but lacks a generic stencil implementation [9]. ExaStencil offers a domain-specific language for stencil operations but misses the opportunity to parallelise pre- and post-processing steps [49]. DUNE facilitates parallel programming in C++ but is restricted to CPUs [4]. Partans and PSkel are frameworks for stencil operations but have not been updated recently [56, 70]. The expansion of OpenACC by Pereira et al.

[26] published at least one experiment with multiple GPUs

targets only a single GPU and is not publicly available [69]. Palabos has demonstrated that frameworks generating GPU code (npFE) can be integrated to address lattice Boltzmann methods (LBMs). Still, they do not encompass generalisable stencil operations for GPUs [50].

6.2 Frontend

In Muesli, the `MapStencil` skeleton can be used for two-dimensional and three-dimensional data structures. Listing 6.1 displays an example of calculating an unweighted blur using the skeleton with a three-dimensional data structure. A user-defined stencil functor has as parameters the data structure used to store the stencil area (padded local cube (PLCube)) and the indices of the currently processed element (l. 2). For two-dimensional data structures, a data structure called padded local matrix (PLMatrix) is used. Elements of the data structure can be accessed without using a separate function, but with the `[]` operator (l. 8). An alternative approach implemented in, e.g. SkePU is to use a relative index allowing accessing data structures by e.g. `cube[0,1,-1]` [25].

```
1  template <size_t radius>
2  MSL_USERFUNC float update(const PLCube<float> &plCube, int x, int y, int z) {
3      float res = 0;
4      // Iterate over all three dimensions - adding the values
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      size_t diameter = radius + radius + 1;
10     // Return the average of all values.
11     return res / (diameter*diameter*diameter);
12 }
13 main () {
14     ...
15     int stencilradius = 2;
16     // Calling the Map Stencil skeleton with the functor as a template parameter.
17     dcp.mapStencil<update<2>>(*dcp2, stencilradius, 0);
18     ...
19 }
```

Listing 6.1: Exemplary functor for the *MapStencil* skeleton calculating a unweighted gaussian blur and the main function calling the skeleton.

In some cases, the relative index might slightly increase the program's readability, which could be realised by making a separate `MapStencilIndex` skeleton available. However, this might cause confusion when distinguishing between a `MapStencil` and `MapStencilIndex` skeleton. Therefore, all `MapStencil` skeletons are index variants in Muesli. Noteworthy, the parameter passing the stencil size (`radius`) could also be a class member instead of a template argument. The design is a matter of preference. When accessing elements from the padded local data structure, it is assumed that no elements outside the stencil radius are accessed. Invalid accesses might result in program termination at runtime. Accessing elements outside the data structure results in returning the neutral value.

The stencil radius and the neutral value of edges are specified in the skeleton call. In l. 17, a distributed cube calls the *MapStencil* skeleton with the stencil function as a template argument. As the first parameter, the skeleton requires a pointer to the output data structure (since the skeleton is not in place). Secondly, the stencil size is passed as a parameter to control the data transfers in the back end, and a neutral value is specified, which is used when the operation accesses out-of-bounds cells. Using a constant value for boundary handling is a straightforward approach that is insufficient to handle more complex stencil operations. Therefore, alternatively, a neutral value functor can be passed to the `MapStencil` skeleton. This approach has the advantage that complex dependencies can be mapped in the function depending on indices and other values. However, using a function for the neutral value produces significant overhead as the values of each iteration have to be calculated repeatedly, as the values might change for every iteration. Listing 6.2 displays such a functor that can be used for solving the partial differential equation (PDE) for a two-dimensional heat distribution where the boundaries have different temperatures. The number of columns must be passed when constructing the functor (l. 3) to handle the element on the right-hand side of the data structure. With this design, arbitrary behaviour can be implemented if needed.

```
1 class NeutralValueFunctor : public Functor2<int, int, float> {
2     public:
3     NeutralValueFunctor(int cols): glob_cols(cols) {}
4     MSL_USERFUNC float operator()(int x, int y) const override { // here, x represents rows.
5         // top, left, and right column must be 100.
6         if (y < 0 || y > (glob_cols - 1) || x < 0) { return 100; }
7         else { return 0;}       // Bottom boundary is 0.
8     }
9     int glob_cols; };
```

Listing 6.2: Exemplary class for the neutral value functor when solving the PDE for a heat distribution with three 100-degree edges and one zero-degree edge.

6.3 Backend Implementation

It is essential to understand the backend implementation to understand the runtime evaluation. The order of operations is displayed in the sequence diagram in Figure 6.2. When writing a program using the `MapStencil` skeleton, the data structure is created first, which can be a DM or a DC. In the example, a DM containing pixel data is created. Calling the skeleton for the first time invokes the creation of an object of the class `plmatrix` for each processing unit. In the given example, it is assumed that only GPUs are used. However, when calculations are offloaded to the CPU, also a PLMatrix for the CPU is created. If a PLMatrix was already created in an earlier `mapStencil()` call, it can be reused if the current stencil size is equal to or less than the one of the previous call. Otherwise, the PLMatrix is recreated. Next, optionally, the size of the PLMatrix is adjusted when the stencil size is adjusted. The data of the PLMatrix is always updated with the `updateData()` function. The function updates the pointer to the current data and communicates the overlapping data points between the computational nodes with the suitable MPI and CUDA operations. When the data is up to date, the user function passed as an argument previously is called for each element, passing the PLMatrix as an argument. Within each user function, the PLMatrix can then be called multiple times to access data structure elements. At the end of the user function, the calculated value is returned to the DM. The usage of the additional data structure allows the encapsulation of only the functionalities which should be available to the user in the user function. Analogously, the PLCubes work in the same way for three dimensions. In the following subsections, two aspects of this design are explained in more detail. Subsection 6.3.1 explains the class PLCube and PLMatrix, and Subsection 6.3.2 explains the communication of overlapping elements.

6.3.1 Data Access

The padded local classes (PLCube and PLMatrix) allow access to individual elements of the data structure without producing significant overhead by providing a minimal design. To make the accesses to the different memory areas efficient, each computational unit has its own class that stores only the elements needed to calculate the elements assigned to it. The behaviour of the class is automatically adjusted for CPUs or GPUs. It contains the following attributes to provide a light, minimal design:

- `int width, height (depth)` representing the necessary dimensions of the data structure,

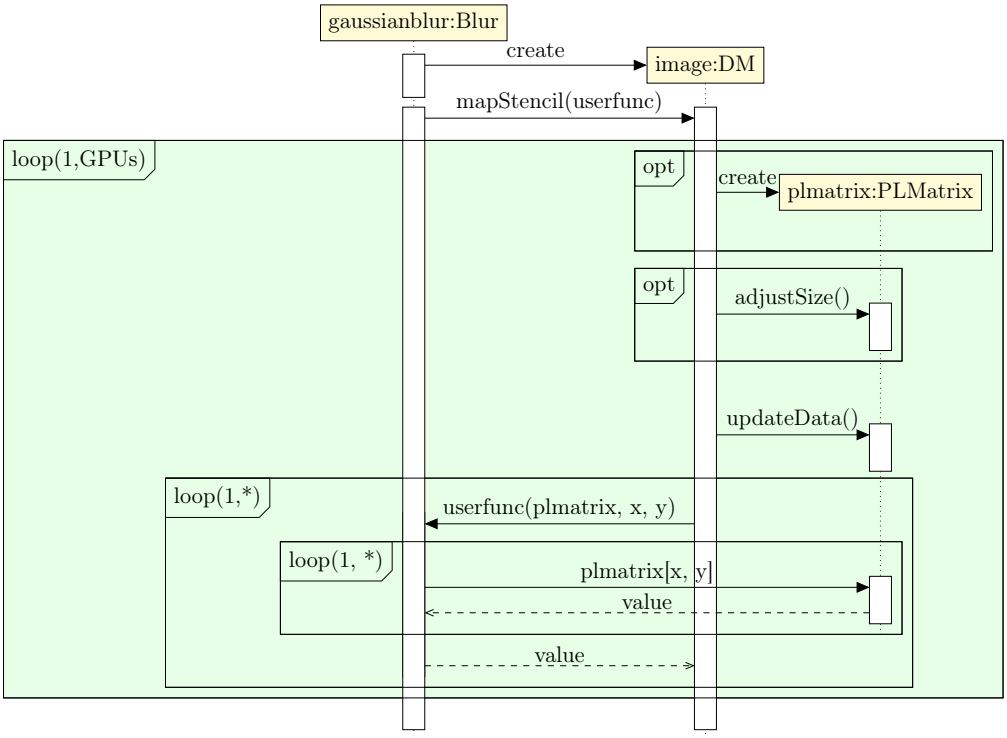


Figure 6.2: Sequence diagram of the *MapStencil* skeleton being used for a Gaussian blur image filter.

- `int stencilRadius` radius of the stencil required to calculate the overlapping elements,
- `T* data, T* topPadding, T* bottomPadding` CPU or GPU pointer to the current data where `T` is the template arguments specifying the data type,
- `T neutralValue` used when the index is outside of the data structure,
- four additional integers to save global indexes for the start and end of main and padding data areas.

Most importantly, the subscript operator is implemented, taking two or 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. This design has the disadvantage that all accesses to memory require conditional statements to check if the element is stored in the top or bottom padding. However, it comes with the advantage of merely updating the data at the pointer, circumventing repeated copy steps for each iteration.

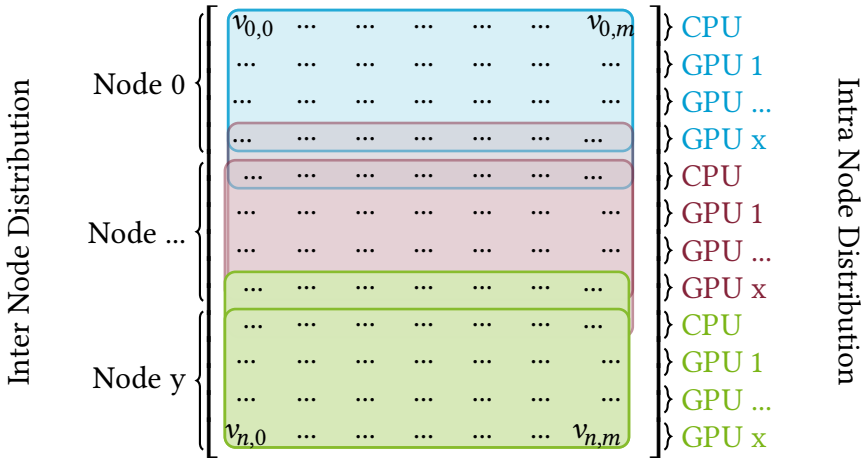


Figure 6.3: Distributing a DM by rows on multiple nodes and computational units

6.3.2 Boundary Management

The most important aspect regarding the backend implementation is to minimise the time spent on data transfers while still being compatible with heterogeneous computing environments. The following paragraphs explain the alternatives tested in Muesli.

Firstly, Figure 6.3 displays a partition by rows. The major advantage of the approach is that it requires only a few operations to update boundary data (less than two on average) and that the distribution is trivial with an increasing number of computational units. The major drawback is that depending on the data structure and the number of computational nodes used, the number of elements that need to be communicated increases.

Previous research has achieved speedups by changing the distribution to a block-wise distribution. A simple example can explain the mathematical reasoning for a block-wise distribution. Assuming a 3×3 stencil, when processing a 1024×1024 matrix with, e.g. four GPUs, a distribution by rows would result in a need to copy 1024 elements for the first and last and 2048 elements for each of the two middle GPUs, which need to update both top and bottom boundaries, resulting in exchanging $6156/4 = 1539$ elements on average. In contrast, a distribution by blocks would result in exchanging $1024 + 3$ elements for each block. This effect increases for bigger stencil sizes and has a major effect when scaling to a larger number of computational units.

However, the number of elements transferred between computational units is not the only factor influencing the runtime and sometimes, it is not the determining factor. Continuing the previous example, the concurrent read operations for elements in the data structure might lead to read contention, reducing the operations executed in parallel. Moreover, the initialisation cost rises when a node has to communicate up to eight borders

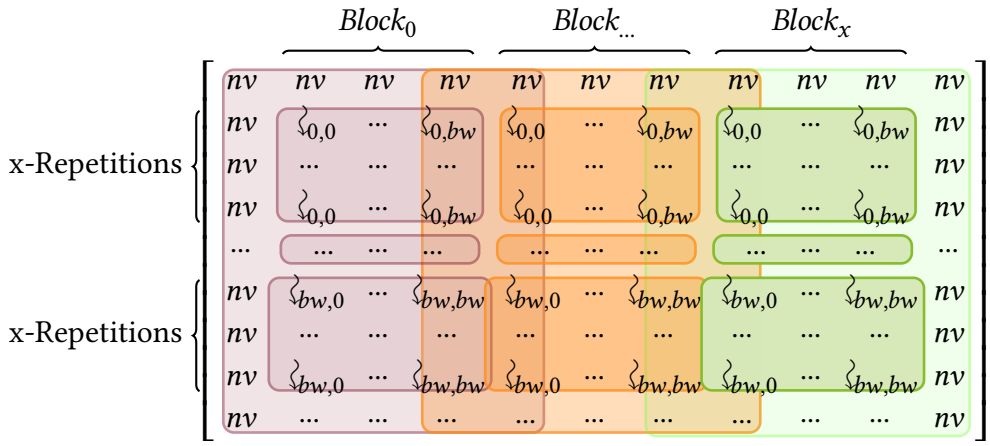


Figure 6.4: Block-wise distribution with the repetitive calculation of elements.

[27], in contrast to two. Lastly, block-wise distribution requires specialised handling for a non-square number of computational units and data structures.

As no clear decision could be made, it was decided that for the distribution between nodes, a row-wise distribution is reasonable, as clusters with a square number of GPU-nodes are rarely available. The same applies to the computational units on a node. In contrast, GPU can start 60-100 blocks in parallel, increasing the benefits of a block-wise distribution [64]. Figure 6.4 combines two optimisation techniques commonly used for GPUs. Firstly, the data structure is split into blocks. Secondly, instead of just one element, multiple elements are calculated per thread. Especially for bigger data structures, the number of values to be calculated preponderates the number of threads that can be started in parallel. In this case, it produces overhead to start new blocks of threads, making it faster to let one thread calculate multiple elements. In the figure, this is depicted as *Repetitions*. Thread $0, 0$ calculates x elements. Commonly, this is also referred to as the tile width. Choosing the optimal tile width depends on multiple factors, such as the number of accesses to other elements in the data structure and the usage of local and shared memory [28]. *bw* represents the x and y-dimensions (assuming they are equal) of the threads started in one block. Both designs have been implemented in Muesli to test the different approaches. However, the block-wise distribution was limited to the GPU.

[27] The corner elements require to communicate to diagonal nodes.
[28] https://developer.nvidia.com/cuda-toolkit-archive - accessed 22.11.2023

6.4 Evaluation

The evaluation of the *MapStencil* skeleton is extensive as the different options drafted are evaluated separately. Therefore, firstly, the examples used are shortly introduced, and their differences are highlighted (Subsection 6.4.1). Thereafter, the single experiments are split into three features. Firstly, offloading the calculation to the CPU is tested in Subsection 6.4.2. Secondly, the usage of shared memory is investigated (Subsection 6.4.3). Thereafter, the scaling behaviour over multiple nodes and GPUs is tested in Subsection 6.4.4. As that experiment shaped the design of the final implementation of the `MapStencil` skeleton, the evaluation is concluded by comparing the implementation to two other frameworks.

Table 6.2 displays the nodes from the cluster Palma II used for the evaluation. The abbreviation will be used during the evaluation.

Abbreviation	Nodes	GPU	SM	CPU	Cores
cpuskylake	143	-	-	Skylake (Gold 6140)	36
cpuzen	12	-	-	Zen2 (EPYC 7742)	128
A100SXM	2	Nvidia A100 SXM	108	Zen3 (EPYC 7343)	64
RTX2080	5	GeForce RTX 2080 Ti	68	Zen3(EPYC 7513)	32
A100	5	Nvidia A100	108	Zen3(EPYC 7513)	32

Table 6.2: Overview of used hardware for the evaluation of the MapStencil skeleton

6.4.1 Examples

Within this evaluation, four examples are used. Firstly, a two-dimensional implementation of a Jacobi solver is implemented. More concretely, this example simulates a plate with four borders, three having a source of heat with 100 degrees and one having 0 degrees. For this example, the previously mentioned neutral value functor is used, as the values of the borders differ. Hence, this example comes with the overhead of calling a function for the neutral value.

Secondly, a two and three-dimensional blur is implemented. This example is especially interesting as the size of the stencil can vary quite reasonably; e.g., in image processing, different radii are reasonable for smoothing an image. Both examples are perfectly suited for parallelism, as no conditional statements are required.

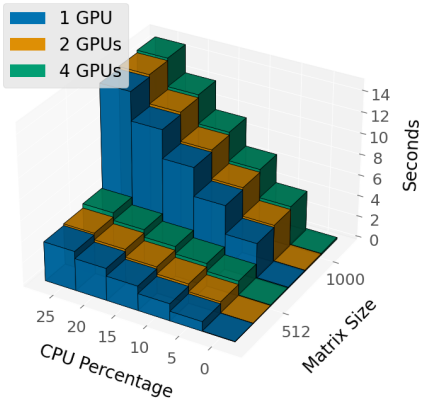
Next, an implementation of Conway's Game of Life is tested. It is a zero-player game which calculates the steps of a cellular automaton where a cell dies or remains dead if it has less than two neighbours, becomes alive if it has exactly three neighbours, stays alive if it has two or three neighbours, and dies if it has more than three neighbours [29]. The most significant difference is that Conway's Game of Life has multiple conditional statements impeding the parallelism on GPUs and is based on integers instead of floating point values.

Lastly, the LBM commonly used for fluid simulation is implemented for our final comparison. Our example simulates a room with a constant source of gas from a corner of the room and barriers within this room that reflect the gas. Therefore, a D3Q19-Grid was used, where D is the number of dimensions and Q is the number of neighbours. The exact formula and program can be found in Chapter 12.

6.4.2 CPU Usage

For testing the feature of distributed calculation presented in Chapter 4, we distributed the calculation of the Jacobi Solver on a node equipped with eight GeForce RTX 2080 Ti and a Zen3 (EPYC 7513) CPU. The solver was set to perform 5000 iterations. The program has been tested against one, two and four GPUs with multiple CPU fractions. An extract of the results can be seen in Figure 6.5a and Table 6.5b. Note that the figure does not depict the same matrix size as this would impede the readability; therefore, the table lists the runtimes for a 5000×5000 matrix. Moreover, the figure includes the data for two GPUs. To be safe, we also tested smaller CPU-shares than depicted. However, no speedup compared to the GPU only variant could be achieved for any tested combinations. The fastest runtimes are highlighted in bold in Table 6.5b. 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. It can be concluded that with a rising percentage of elements calculated on the CPU, the runtime increases linearly. This finding also holds for bigger data structures and when using multiple GPUs. As the runtime difference increases for bigger data sizes, it is concluded 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.

[29] `https://conwaylife.com/`



(a) Runtimes for different CPU fractions calculated by the CPU.

		Matrix size			
		512 × 512		5000 × 5000	
		GPUs			
		1	4	1	4
CPU %	0	**0.0291**	**0.0505**	**1.5244**	**1.4389**
	5	0.9121	1.4287	77.8069	76.9942
	10	1.6239	2.1743	146.3872	145.4040
	15	2.3374	2.8821	215.5982	214.1764
	20	3.0690	3.5955	284.9108	285.1225
	25	3.7524	4.3260	356.1769	357.0563

(b) Runtimes (in seconds) for different fractions calculated by the CPU for the Jacobi Solver.

Figure 6.5: Evaluation of different fractions calculated by the CPU for the Jacobi Solver.

6.4.3 Shared Memory

Using shared memory reduces the time needed to access elements while requiring time to load the elements into the shared memory space. Therefore, when using the shared memory on the GPU, a row-wise distribution is a disadvantage as more elements can be reused with a block-wise distribution. This design is also commonly referred to as tiling. In the work laying the foundation for this chapter [41], the shared memory was tested with multiple of the listed examples. Interested readers can either inspect the publication or see exemplary numbers for the Jacobi method in Appendix A.3. It was concluded that the advantage of using shared memory for small stencil sizes was neglectable; therefore, only the most intuitive example, the Gaussian blur, is presented in this subsection. For this purpose, a 6000 × 4000 pixel large image was processed with ten iterations. The stencil size was varied from a 3 × 3 stencil to a 21 × 21 stencil. As an additional optimisation technique, multiple elements per thread were processed. This technique is commonly used when the number of elements to be calculated exceeds the number of threads that can run in parallel, and the size of the shared memory allows the load of all required elements into the memory space. The maximum number of elements that can be processed can be calculated as follows:

$$\left\lfloor \frac{\frac{elements}{(tdim_x + (ss_x - 1))} - (ss_y - 1)}{tdim_y} \right\rfloor \tag{6.1}$$

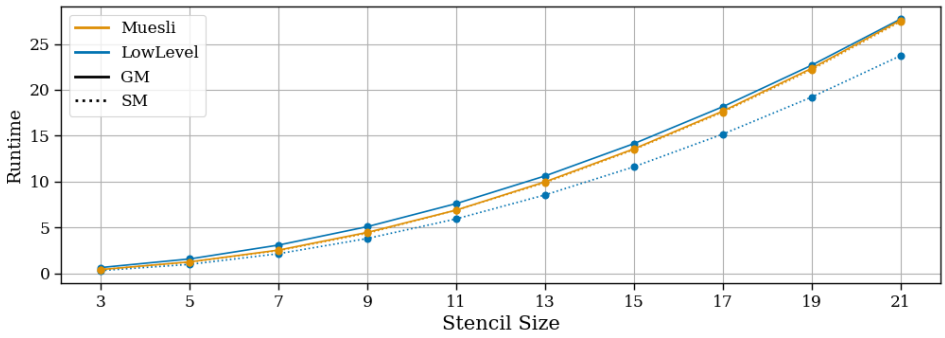


Figure 6.6: Runtimes (in seconds) on a single GPU using shared memory in contrast to the global memory for the Gaussian Blur for a low-level and a Muesli program.

where *elements* is the number of elements that can be stored in the shared memory, ss_x, ss_y the x and y dimensionality of the stencil and $tdim_x, tdim_y$ the thread dimension. For the example of a GeForce RTX 2080 Ti, this equation can be used as follows: 12288 floating-point numbers of type `float` can be loaded into the 48 KB of shared memory. Assuming a stencil size of 9×9 and 1024 threads $\lfloor \frac{\frac{12288}{32+8} - (8)}{32} \rfloor = 9$ elements can be calculated per thread. This technique also reduces the elements accessed from different threads, as the elements needed for calculating an element and its neighbour are close to each other. Within this experiment, the number of elements processed per thread varied from 1 to 8 elements. No runtime improvement could be found for a higher number of elements per thread. The Muesli program is compared to a hand-written CUDA implementation for verification purposes.

Table 6.3 and Figure 6.6 depict the runtimes of the best configurations of the two programs (mostly six elements per thread). With an increasing stencil size, the advantage of using the shared memory is visible for the low-level implementation but not for the Muesli implementation. The impact of the speedup can be seen in Table 6.3 in boldface, with the optimum runtime also highlighted in boldface. While the low-level program achieves a speedup of 1.29 and 1.17, the Muesli program shows no significant speedup. The most important difference leading to a slow-down of the Muesli program is the generic handling of border elements, which includes multiple conditional statements impeding the parallelism when loading the elements into the shared memory. One might wonder why the difference is relatively low. Although shared memory accesses are approximately 100 times faster than global memory accesses, modern GPUs have efficient L1 and L2 caches. Those are controlled by the compiler. The GeForce RTX 2080 Ti has 64 KB L1 Cache per SM, allowing it to store 16384 floats. Keywords like `__constant__` or

Stencil size	Muesli					
	GM	SM 4^2	SM 8^2	SM 16^2	SM 32^2	Speedup
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**
Stencil size	Low-level					
	GM	SM 4^2	SM 8^2	SM 16^2	SM 32^2	Speedup
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**

Table 6.3: Extract of the Runtimes (in seconds) on a single GPU using shared memory in contrast to the global memory for the Gaussian Blur for a low-level and a Muesli program.

`__restrict__` additionally ease the automatic detection of relevant memory. Therefore, modern compilers can partly compensate for missing optimisation techniques by using caches efficiently.
In summary, neither using the shared memory nor offloading calculation to the CPU could significantly speed up the most straightforward high-level version of the `MapStencil` skeleton using global memory and no specific optimisation techniques. Therefore, for all further experiments, the version of the `MapStencil` skeleton using the global memory will be used.

6.4.4 Hardware Scaling with Global Memory

To evaluate our results' scalability, the Jacobi solver and the Game of Life were tested with up to four nodes and four GPUs each. An extract of the runtimes is displayed in Table 6.4 for the Jacobi method and Table 6.5 for the Game of Life. In both tables, a comparison to a sequential C++ implementation can be found running on a single Skylake (Gold 5118) CPU.
It can be seen that for a single node for small matrix sizes (512×512 matrix), using two or four GPUs actually produces a slow-down of the program compared to a single GPU. The initialisation and the communication between the GPUs outweigh the advantage of starting more threads concurrently. Two factors strengthen this effect. Firstly, the cluster does not have direct communication between GPUs but requires always to include the CPU for communicating boundaries. Secondly, e.g. for four GPUs, merely 65536 elements are calculated per GPU. Therefore, the time spent on calculation is very small, and the possible parallelism is not exploited. In contrast, for bigger data sizes (10000×10000 matrix), using four GPUs improves the speedup from ~ 38 to ~ 51. However, ideally,

matrix size	seq. C++	1 Node 1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
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
10000^2	3544.270	93.156	**38.047**	73.815	48.015	69.092	**51.298**
matrix size	seq. C++	2 Nodes 1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
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
10000^2	3544.270	48.285	73.404	40.636	87.221	39.388	89.984
matrix size	seq. C++	4 Nodes 1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
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
10000^2	3544.270	31.435	**112.749**	29.443	120.379	27.551	**128.642**

Table 6.4: Runtimes (in seconds) and speedups on multiple nodes and GPUs per node only using GPU global memory for the Jacobi Solver. Numbers in boldface are mentioned in the text. All speedups are relative to the sequential program.

using four GPUs would result in a speedup close to four, while for this example, the improvement is far less than linear.

When increasing the number of nodes, it can be verified that the CPU is the bottleneck of the speedup. In contrast to a speedup of ~51 achieved for one node with four GPUs, two nodes with two GPUs achieve a speedup of ~87 and four nodes with each one GPU achieve a speedup of ~113. Therefore, when using multiple nodes with one GPU each, the optimal speedup of 4 cannot be reached, but nearly a factor of 3 can be reached (93.16/31.44). Increasing the number of GPUs on multiple nodes runs into the same effect as previously, as the data transfer on a single CPU produces a bottleneck. Scaling on four GPUs for two nodes and four nodes only achieves a speedup of 1.23 and 1.14 compared to the single GPU variant.

The implementation of the Game of Life was configured to be executed 20000 times as the runtime on a single GPU should not be significantly lower than in the previous example, and for the example, multiple repetitions are reasonable. The runtimes were generated using the same hardware setup. As the findings vary considerably from the previous one, those are explained separately. For a single GPU, a speedup of ~178.6 can be achieved for a small data set and ~265 for a bigger data set. The Game of Life user

matrix		1 Node					
size	seq. C++	1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
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**
matrix		2 Nodes					
size	seq. C++	1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
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
matrix		4 Nodes					
size	seq. C++	1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
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

Table 6.5: Runtimes (in seconds) and speedups on multiple nodes and GPUs per node only using GPU global memory for the Game of Life. Numbers in boldface are mentioned in the text.All speedups are relative to the sequential program.

function has eight addition operations and two conditional statements. Therefore, it is slightly more complex than the Jacobi method (four additions and one division).

The program takes more than double the time for a small data set with 256×1024 cells per GPU when using four GPUs. Noteworthy, the runtimes are still relatively small (< 3s). Interestingly, the opposite finding regarding the scaling behaviour on multiple nodes can be found for bigger data sizes. While the Jacobi method scaled better across nodes, for the Game of Life, the speedup for four GPUs on a single node (~816) is better than for one GPU on four nodes (~509), and slightly inferior to the speedup on two nodes with two GPUs (839). Scaling across more than four GPU still reduces the runtime but suffers from the communication. For example, scaling from a single GPU to two nodes with two GPUs (therefore, quadruplicating the hardware) improves the runtime by a factor of ~3.1, while scaling to two nodes with four GPUs each (octuplicating the hardware) improves the runtime by a factor of ~4.3. The GPUs are not fully occupied, and the additional overhead of initialising a GPU on another node and transferring the data slows down the overall program.

While it is easily explainable that using eight computational nodes on an 8192×8192 matrix produces a significant communication overhead, the findings for the scaling behaviour are unexpected. Two reasons explain the effect. Firstly, calling the neutral value functor scales better across multiple CPUs as the padding is filled on the CPU and,

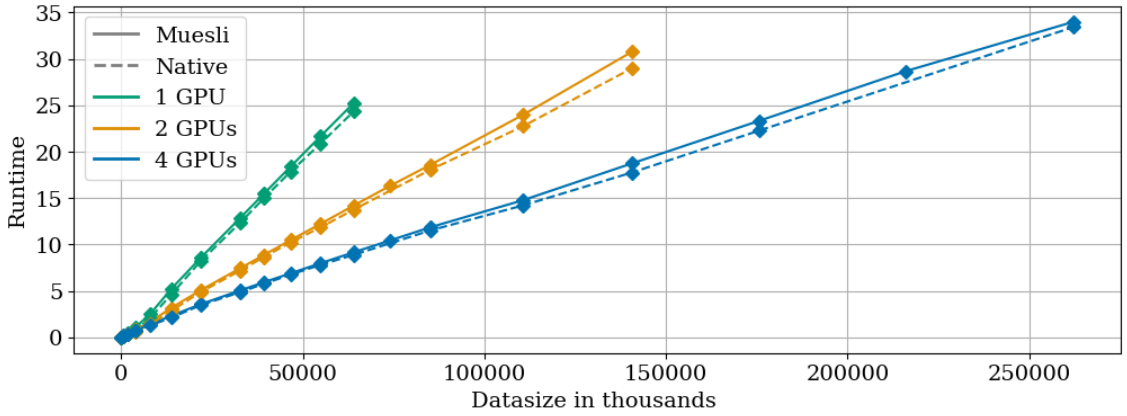


Figure 6.7: Runtime in seconds of a multi-GPU Muesli program and native (CUDA) implementation of the LBM on a single node equipped with GeForce RTX 2080 Ti GPU.

therefore, is a bottleneck when using multiple GPUs. Secondly, the data transfers are smaller for the Game of Life as integer values are used. This favours GPU transfers as the initialisation costs of MPI data transfers are higher.

Next, the scaling behaviour of the examples for three-dimensional data structures is evaluated. As more elements are processed and user functions get more complex, a better scaling behaviour is expected. For the LBM experiment, Figure 6.7 displays the runtime in seconds compared to a low-level implementation using up to four GPUs[30]. Although the native implementation is always slightly faster, the Muesli program performs similarly. The runtimes for bigger data sets are not displayed for one and two GPUs to improve the readability and to highlight the continuous similar runtimes of Muesli and the hand-written implementation. Also worth mentioning is that the biggest data sizes do not fit into the available memory on a single GPU or two GPUs.

The scalability on multiple nodes is similar to previous results. It can be seen that the speedup when adding more hardware resources is not linear due to the communication overload. A slow-down can be seen, e.g. when comparing two nodes with two GPUs to two nodes with four GPUs (highlighted in boldface red). When scaling on two computational units (merely communicating one boundary) either distributed over two nodes or on the same node, a speedup of 1.72 (one node) and 1.91 (two nodes) can be reached (highlighted in boldface green).

Last, but not least, we wanted to evaluate the scaling by measuring the runtime of a delightfully parallel program, namely a mean blur, and separately measuring the time

[30]The implementation was part of a bachelor thesis [16] and adjusted afterwards `https://github.com/NinaHerrmann/ba-native`

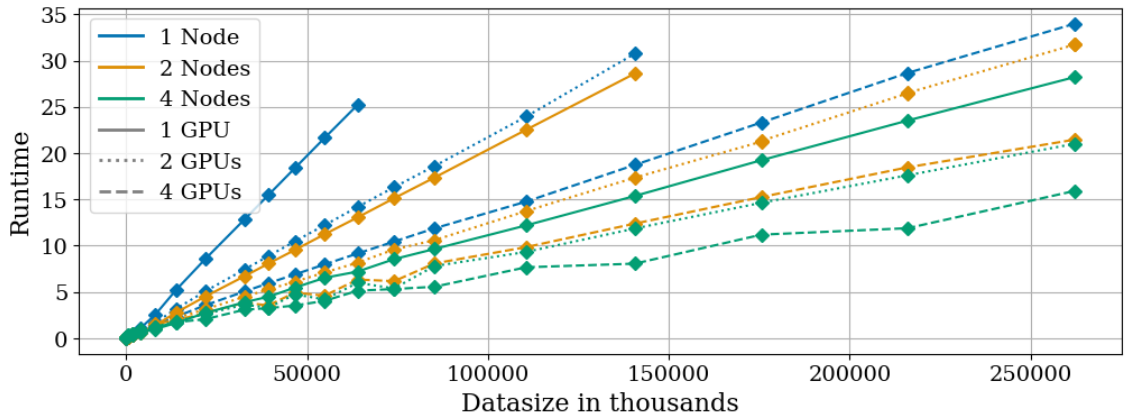


Figure 6.8: Runtimes of the LBM Muesli program on multiple nodes and multiple GPUs on the RTX2080 partition.

Cube size	Seq. C++	1 Node					
		1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
160^3	135.612	1.136	119.391	0.831	163.107	0.764	177.528
320^3	1125.32	12.858	**87.521**	7.447	**151.107**	5.055	222.597
640^3	9261.54	-	-	-	-	33.99	272.477
Cube size	Seq. C++	2 Nodes					
		1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
160^3	135.612	0.661	205.019	**0.637**	212.975	**0.726**	186.879
320^3	1125.32	6.714	**167.62**	4.492	250.516	3.842	292.884
640^3	9261.54	31.731	291.879	31.734	291.849	21.462	431.538
Cube size	Seq. C++	4 Nodes					
		1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
160^3	135.612	0.628	216.041	**0.868**	156.319	**0.945**	143.553
320^3	1125.32	3.893	289.084	3.623	310.576	3.087	364.478
640^3	9261.54	28.22	328.188	21.02	440.606	15.918	581.835

Table 6.6: Speedup for the parallel implementation of the LBM gas simulation on the RTX2080 partition.

Stencil radius	Data size	1 GPU	4 GPUs	4 GPUs -Com[a]	Speedup	Speedup -Com	8 GPUs	8 GPUs -Com[a]	Speedup	Speedup -Com
9	120^3	4.54	1.47	1.35	3.09	3.35	1.04	0.79	4.37	5.74
	280^3	60.58	16.44	15.85	3.68	3.82	9.71	8.47	6.24	7.15
	400^3	192.61	51.59	50.34	3.73	3.83	29.52	26.96	6.52	7.14
	560^3	563.20	148.68	146.41	3.79	**3.85**	83.38	78.58	6.75	7.17
13	120^3	13.59	4.20	4.03	3.23	3.37	2.53	2.17	5.38	6.27
	280^3	252.54	69.24	68.32	3.65	3.70	37.92	35.97	6.66	7.02
	400^3	836.59	222.12	220.38	3.77	3.80	118.89	115.15	7.04	7.27
	560^3	2464.76	643.33	640.16	3.83	**3.85**	338.46	331.65	**7.28**	**7.43**

Table 6.7: Speedup for the parallel implementation of the mean blur for multiple GPUs on the RTX2080 partition.

[a] Runtimes without communication between nodes and GPUs.

spent on communication. In contrast to the LBM, the example does not have conditional statements in the user function. Therefore, thread divergence does not take place. This advantage profits the speedup of the program when comparing the scaling across GPUs and nodes. For example, when scaling across eight GPUs, a speedup of 7.28 can be reached (including the communication), and without the communication overhead (not including synchronisation), a speedup of 7.43 can be achieved (Table 6.7 highlighted in green). Also, the speedup, even including communication, is more striking for bigger stencil sizes, as the growing complexity of the user function outweighs the communication. When scaling over four GPUs, the speedup for the stencil sizes 9 and 13 is even similar (also highlighted in green). Similar findings could be made when scaling across multiple nodes [37].

Supplementary Figure 6.9 compares the runtimes of the mean blur using the same number of accelerators, either having them on the same node or distributed over multiple nodes. For the first comparison, it can be clearly seen that two nodes with four GPUs are faster than a single node with eight GPUs. Although additional MPI calls have to be made, the parallel data transfer to the GPU done by the multiple nodes compensates for the time required for the calls. In contrast, having a single node with two GPUs is faster than two nodes with each a single GPU, as the additional MPI transfer is necessary.

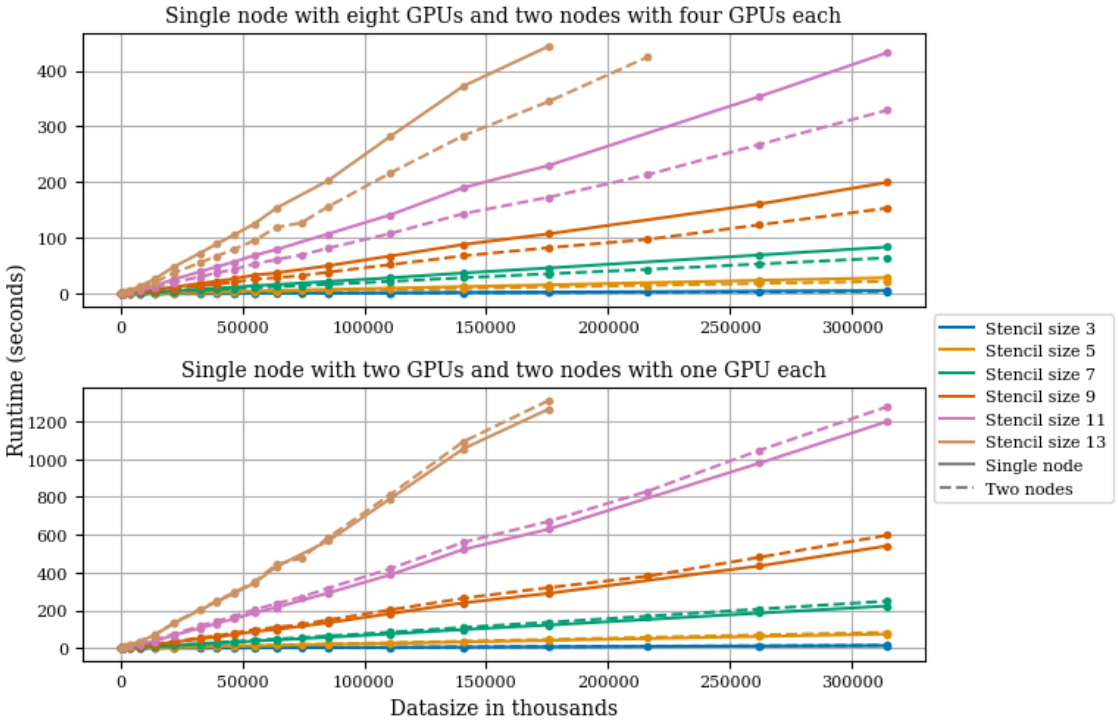


Figure 6.9: Runtimes of the Blur Muesli program on the RTX2080 partitions with similar hardware.

6.4.5 Framework Comparison

The final design of the `MapStencil` skeleton can now be compared to other frameworks. As the LBM is the example closest to a real-world application, and it can also be compared to specialised libraries, it was used for the comparison. From the selection of frameworks listed in Section 6.1, Palabos and the lbmpy library were chosen as specialised frameworks that should generate efficient code. For a multi-GPU program, Celerity was chosen. SkePU was also considered, but problems were encountered when scaling across multiple GPUs with the skeleton.

Although Palabos is specialised for the LBM, the Muesli implementation outperforms Palabos by a factor of up to four for bigger data sizes, as highlighted in boldface in Table 6.8. The lbmpy python package has a C++ backend implementing a GPU parallel version of the LBM. Muesli always outperforms lbmpy by at least a factor of 1.8. This factor decreases with bigger data sets but could not be investigated further as lbmpy consumes significantly more memory. For the given experiment, lbmpy could compute a data structure with a maximum of 280^3 elements while using the same variable type, Muesli could process data structures with up to 400^3 elements.

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

Table 6.8: Comparison of Palabos, lbmpy and Muesli on the cpuzen partition for CPU programs and on the 2080 partition for GPU programs. All GPU programs are executed on a single GPU.

In contrast to Muesli that uses the CUDA native NVCC compiler, Celerity uses the Clang-based compiler of hipSYCL. When testing, the program created with Muesli outperformed the one generated with Celerity (Figure 6.10). This was rather surprising as the Celerity runtime builds a task graph and a command graph, allowing us to optimise programs while generating them. During the investigation for the cause, it was found that the SYCL compiler did not unroll some for loops, which is an optimisation that the NVCC compiler could do automatically. Manually inserting a `#pragma unroll` in the program resulted in a faster program, as depicted in the graph. However, it can be recorded that without manual adjustments, Muesli performed better than Celerity on four GPU.

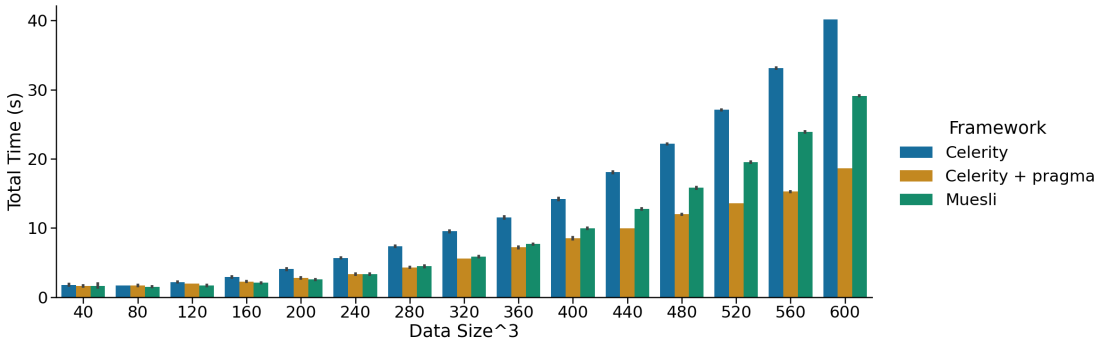


Figure 6.10: Runtime comparison of a multi-GPU Muesli program and a multi-GPU Celerity program of the LBM on the A100SXM partition.

6.5 Conclusion and Outlook

The extensive evaluation of the `MapStencil` skeleton allowed the design of a generic skeleton that can be used for multiple example applications. From the experiments, it can be inferred that using a neutral value functor has a major impact on the possible speedup. Offloading elements to the CPU could not speed up the computations. For the shared memory, it can be concluded that Muesli can not achieve the speedup achieved with a low-level implementation. The overhead of loading elements generically to the shared memory can not be diminished. When scaling across multiple nodes, the efficiency depends on the problem and the data size. For the rather trivial example of a mean blur, it could be shown that also for bigger stencil sizes, nearly the optimal speedup is achieved over eight GPUs. The cost of the communication depends on the hardware used, clearly showing that nodes with more than four GPUs should start multiple MPI processes to parallelise the communication to the GPU. Lastly, Muesli was compared to three other frameworks, showing that it can not only compete with them but outperform those. A minor optimisation could also be to provide more means to define the neutral value.

While the stencil skeleton has been detailed and tested, it highlights another research gap that often is neglected, namely the comparison to other frameworks. While a comparison was provided, this could be enhanced by comparing a variety of applications with multiple frameworks.

This chapter is based on the publication:

- Nina Herrmann, Breno Menezes, and Herbert Kuchen. 'Stencil calculations with algorithmic skeletons for heterogeneous computing environments'. In: *International Journal of Parallel Programming* 50.5-6 (2022), pp. 433–453,
- Nina Herrmann, Justus Dieckmann, and Herbert Kuchen. 'Optimizing Three-Dimensional Stencil-Operations on Heterogeneous Computing Environments'. Accepted for publication in the International Journal of Parallel Programming. 2024.

7

Conclusion

In this thesis, the algorithmic skeleton library Muesli has been revisited. Multiple research streams have been invested. The contributions are listed in Section 7.1. Afterwards, Section 7.2 list the limitations of this research. Last but not least, Section 7.3 provides a perspective regarding future research opportunities.

7.1 Contributions

This research enhances the exploration of opportunities for high-level frameworks. Firstly, the hardware that can be used by Muesli was inspected. From our analysis, it could be concluded that other types of accelerators can be neglected as they are commonly unavailable in clusters. Therefore, this work focused on the optimisation of the available hardware, namely to inspect the exploitation of the CPU when using the GPU. From the executed experiments, indicators could be found that favour the offloading of calculation to the CPU. Increasing data structure sizes and more complex user functions favour a higher CPU-share. While for medium-strong GPUs, a speedup of 1.2 could be reached, the A100 GPU showed nearly no benefit. As the complexity of the user function can not be analysed in the current setup, the indicators are not reliable enough to implement an automatic determination of the optimum CPU-share. Nevertheless, the documented findings can guide users of Muesli to experiment with the functionality. Noteworthy, Muesli by design is one of the few high-level frameworks that can scale over multiple nodes equipped with multiple accelerators and, therefore, can parallelise over complex hardware setups. The efficiency is slightly limited as it can not efficiently exploit multi-core CPUs when using the GPU program.

Secondly, the applicability of Muesli for different programs was tested. Firstly, the ACO algorithm was implemented for solving the TSP problem in Muesli. It was found that for Muesli, merely a `Zip` skeleton that processes three data structures had to be added. Especially the support for C++ data structures eased the implementation, which is a major advantage compared to Musket. With the adjusted data structure, Muesli was

impressively faster than the previous implementation in Musket. It clearly showed that Muesli is suitable for implementing a meta-heuristic such as ACO.

Next, Muesli was extended with a `MapStencil` skeleton. This skeleton was extensively tested with multiple hardware setups and examples. Those examples included the Game of Life, a blur, and an implementation of the LBM. It could be found that offloading calculations to the CPU does not speed up the program. Compared to a low-level implementation, Muesli could not achieve the same performance when using the local memory of the GPU with relatively big stencil sizes; for all other examples, Muesli could achieve similar performance to a low-level implementation. The diversity of the examples also highlights the broad applicability of the skeleton. Lastly, for the LBM example, Muesli was faster than the three other frameworks tested, two of them being specialised in the LBM.

Finally, it can be said that Muesli was successfully revisited and adjusted to the current developments in parallel programming. It could be shown that Muesli can be used to implement a broad spectrum of applications and support heterogeneous computing environments. Both aspects were critical to fulfilling the stated research objective.

7.2 Limitations

Although the conducted research discussed multiple aspects, it also has its limitations. Firstly, the usability of Muesli was not tested. Although Muesli is designed to the best of the author's knowledge, it was not proven that Muesli is easy to use. Especially for the implementation of the ACO method, it is questionable if an average programmer would create the same program.

Moreover, the hardware setups tested were limited by the computational nodes available on the cluster Palma II of the University of Münster. This also limited the software stack available, e.g. for testing multiple MPI versions. Moreover, due to the setup, AMD GPUs could not be tested. Moreover, the comparisons were limited to a selection of frameworks with only a couple of examples. Therefore, it can not be guaranteed that all findings can be transferred to other examples and hardware setups.

7.3 Outlook

As hardware is constantly changing, developments for new accelerators and hardware architectures will always be a topic for high-level approaches. FPGA, recent developments in the area of graphene chips, and the ongoing research on quantum computers are the most prominent examples to showcase perspectives for new hardware developments.

As algorithms in the area of artificial intelligence are becoming indispensable, the efficient implementation of algorithms in this area would emphasise the applicability of Muesli further. Of course, other types of programs and algorithms would also be valuable.

Another interesting topic that has been neglected in the past is the complete comparison of high-level frameworks. This would include runtime experiments for multiple examples and supplement those findings with a usability study and a performance comparison of the programs that could be created by those users. Ideally, this comparison would also include frameworks not discussed in this dissertation due to the selection criteria. For example, the performance of functional languages could be used in combination with parallelism to speed up computation. An example was already presented [35].

Finally, the development of a precompiler or runtime system for Muesli could open up new possibilities. First, an analysis of the Muesli program could benefit the offloading to the CPU as the share could be based on the complexity of the user function. Moreover, such an architecture could help to implement automatic optimisations such as not scaling across multiple nodes in case it requires too much communication or recommending hardware that is especially suitable for the given program.

References

[1] Marco Aldinucci, Marco Danelutto, Maurizio Drocco, Peter Kilpatrick, Claudia Misale, Guilherme Peretti Pezzi, and Massimo Torquati. 'A parallel pattern for iterative stencil+ reduce'. In: *The Journal of Supercomputing* 74 (2018), pp. 5690–5705.

[2] Fernando Alexandre, Ricardo Marques, and Hervé Paulino. 'On the support of task-parallel algorithmic skeletons for multi-GPU computing'. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC '14. Gyeongju, Republic of Korea: Association for Computing Machinery, 2014, pp. 880–885.

[3] Yadu Babuji et al. 'Parsl: Pervasive Parallel Programming in Python'. In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 25–36.

[4] Peter Bastian et al. 'The Dune framework: Basic concepts and recent developments'. In: *Computers & Mathematics with Applications* 81 (2021). Development and Application of Open-source Software for Problems with Numerical PDEs, pp. 75–112.

[5] Martin Bauer, Harald Köstler, and Ulrich Rüde. 'lbmpy: Automatic code generation for efficient parallel lattice Boltzmann methods'. In: *Journal of Computational Science* 49 (2021).

[6] Martin Bauer et al. 'waLBerla: A block-structured high-performance framework for multiphysics simulations'. In: *Computers and Mathematics with Applications* 81 (2021). Development and Application of Open-source Software for Problems with Numerical PDEs, pp. 478–501.

[7] Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. 'A survey of high-level parallel programming models'. In: *Heriot-Watt University, Edinburgh, UK* 1.2 (2013).

[8] Günther Bengel. *Masterkurs Parallele und Verteilte Systeme*. ger. 2., erw. und aktualisierte Aufl. Lehrbuch. Wiesbaden: Springer Vieweg, 2015. ISBN: 9783834816719.

[9] Manuel de Castro, Inmaculada Santamaria-Valenzuela, Yuri Torres, Arturo Gonzalez-Escribano, and Diego R Llanos. 'EPSILOD: efficient parallel skeleton for generic iterative stencil computations in distributed GPUs'. In: *The Journal of Supercomputing* (2023), pp. 1–34.

[10] José M. Cecilia, José M. García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. 'Enhancing data parallelism for ant colony optimization on GPUs'. In: *Journal of Parallel and Distributed Computing* 73.1 (2013), pp. 42–51.

[11] Philipp Ciechanowicz and Herbert Kuchen. 'Enhancing muesli's data parallel skeletons for multi-core computer architectures'. In: *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*. IEEE. 2010, pp. 108–113.

[12] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. English. MIT Press, 1991.

[13] Alyson Deives and Rodrigo Rocha. *PSkel*. `url`: `https://github.com/pskel/pskel` (visited on 13/02/2024).

[14] Audrey Delévacq, Pierre Delisle, Marc Gravel, and Michaël Krajecki. 'Parallel Ant Colony Optimization on Graphics Processing Units'. In: *Journal of Parallel and Distributed Computing* 73.1 (2013). Metaheuristics on GPUs, pp. 52–61.

[15] Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño. 'A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era'. In: *IEEE Transactions on Parallel and Distributed Systems* 23.8 (2012), pp. 1369–1386.

[16] Justus Dieckmann. *Implementierung und Optimierung von dreidimensionalen Stencil-Skeletten mithilfe paralleler Verarbeitung*. 2023-02-14.

[17] Marco Dorigo and Thomas Stutzle. *Ant Colony Optimization*. A Bradford Book. The MIT Press, 2019.

[18] Marco Dorigo and Thomas Stützle. 'Ant Colony Optimization: Overview and Recent Advances'. In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. Cham: Springer International Publishing, 2019, pp. 311–351.

[19] Steffen Ernsting. *Data parallel algorithmic skeletons with accelerator support*. eng. Münster, 2016.

[20] Steffen Ernsting and Herbert Kuchen. 'A scalable farm skeleton for hybrid parallel and distributed programming'. In: *International Journal of Parallel Programming* 42 (2014), pp. 968–987.

[21] Steffen Ernsting and Herbert Kuchen. 'Algorithmic skeletons for multi-core, multi-GPU systems and clusters'. In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138.

[22] Steffen Ernsting and Herbert Kuchen. 'Algorithmic skeletons for multi-core, multi-GPU systems and clusters'. In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138.

[23] August Ernstsson, Johan Ahlqvist, Mahder Gebremedhin, and Henrik Henriksson. *SkePU*. url: https://github.com/skepu/skepu/ (visited on 18/04/2024).

[24] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. 'SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters'. In: *International Journal of Parallel Programming* 49.6 (2021), pp. 846–866.

[25] August Ernstsson and Christoph Kessler. *SkePU*. url: https://skepu.github.io (visited on 18/04/2024).

[26] August Ernstsson, Lu Li, and Christoph Kessler. 'SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems'. In: *International Journal of Parallel Programming* 46 (1 Feb. 2018), pp. 62–80.

[27] MPI Forum. *MPI: A Message-Passing Interface Standard*. 2021. url: https://www.mpi-forum.org/docs/ (visited on 21/05/2024).

[28] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). *ExaStencil*. url: https://github.com/lssfau/ExaStencils (visited on 13/02/2024).

[29] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). *lbmpy*. url: https://i10git.cs.fau.de/pycodegen/lbmpy (visited on 13/02/2024).

[30] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). *WaLBerla*. url: https://i10git.cs.fau.de/walberla/walberla (visited on 13/02/2024).

[31] Mehdi Goli and Horacio González-Vélez. '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*. 2013, pp. 148–156.

[32] González-Vélez, Horacio and Leyton, Mario. 'A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers'. In: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160.

[33] The Open Group. *pthread.h(0p) — Linux manual page*. url: https://www.man7.org/linux/man-pages/man0/pthread.h.0p.html (visited on 05/08/2024).

[34] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 'High Performance Stencil Code Generation with Lift'. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 100–112.

[35] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 'Futhark: purely functional GPU-programming with nested parallelism and in-place array updates'. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 556–571. ISSN: 0362-1340. DOI: 10.1145/3140587.3062354. URL: https://doi.org/10.1145/3140587.3062354.

[36] Nina Herrmann, Justus Dieckmann, and Herbert Kuchen. 'Optimizing Three-Dimensional Stencil-Operations on Heterogeneous Computing Environments'. Contribution presented at the 16th International Symposium on High-Level Parallel Programming and Applications, June 29-30, Cluj-Napoca, Romania, 2023.

[37] Nina Herrmann, Justus Dieckmann, and Herbert Kuchen. 'Optimizing Three-Dimensional Stencil-Operations on Heterogeneous Computing Environments'. Accepted for publication in the International Journal of Parallel Programming. 2024.

[38] Nina Herrmann and Herbert Kuchen. 'Distributed Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments'. Contribution presented at the 15th International Symposium on High-Level Parallel Programming and Applications, July 7-8, Porto, Portugal. 2022.

[39] Nina Herrmann and Herbert Kuchen. 'Distributed Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments'. In: *International Journal of Parallel Programming* 51.2-3 (2023), pp. 172–185.

[40] Nina Herrmann, Breno Menezes, and Herbert Kuchen. 'Stencil calculations with algorithmic skeletons for heterogeneous computing environments'. Contribution presented at the 14th International Symposium on High-Level Parallel Programming and Applications, July 12-13, Online Event. 2021.

[41] Nina Herrmann, Breno Menezes, and Herbert Kuchen. 'Stencil calculations with algorithmic skeletons for heterogeneous computing environments'. In: *International Journal of Parallel Programming* 50.5-6 (2022), pp. 433–453.

[42] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. 'Design Science in Information Systems Research'. In: *MIS Quarterly* 28 (2004), pp. 75–105. (Visited on 17/07/2019).

[43] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. 'CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application'. In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 2013, pp. 136–143.

[44] 'IEEE Standard Portable Operating System Interface for Computer Environments'. In: *IEEE Std 1003.1-1988/INT, 1992 Edition* (1988), pp. 1–40.

[45] Kawthar Shafie Khorassani, Chen-Chun Chen, Bharath Ramesh, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. 'High Performance MPI over the Slingshot Interconnect'. In: *Journal of Computer Science and Technology* 38.1 (2023), pp. 128–145.

[46] Khronos Group. *SYCL*. `url`: `https://www.khronos.org/sycl/` (visited on 20/02/2024).

[47] Christos Kotsalos, Jonas Latt, and Bastien Chopard. 'Palabos-npFEM: Software for the simulation of cellular blood flow (digital blood)'. In: *arXiv preprint arXiv:2011.04332* (2020).

[48] Herbert Kuchen. 'A Skeleton Library'. In: *Euro-Par 2002 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 620–629.

[49] Sebastian Kuckuk and Harald Köstler. 'Whole Program Generation of Massively Parallel Shallow Water Equation Solvers'. In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 2018, pp. 78–87.

[50] Jonas Latt et al. 'Palabos: Parallel Lattice Boltzmann Solver'. In: *Computers and Mathematics with Applications* 81 (2021). Development and Application of Open-source Software for Problems with Numerical PDEs, pp. 334–350.

[51] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. 'There's plenty of room at the Top: What will drive computer performance after Moore's law?' In: *Science* 368.6495 (2020).

[52] John Levine and Frederick Ducatelle. 'Ant colony optimization and local search for bin packing and cutting stock problems'. In: *Journal of the Operational Research Society* 55.7 (2004), pp. 705–716.

[53] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Peter Wyckoff, and D K. Panda. 'Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics'. In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. SC '03. Phoenix, AZ, USA: Association for Computing Machinery, 2003, p. 58.

[54] Frédéric Loulergue and Philippe Jolan. *PySke*. `url: https://github.com/pyske/PySke` (visited on 13/02/2024).

[55] Frédéric Loulergue and Jolan Philippe. 'Automatic Optimization of Python Skeletal Parallel Programs'. In: *Algorithms and Architectures for Parallel Processing*. Ed. by Sheng Wen, Albert Zomaya, and Laurence T. Yang. Cham: Springer International Publishing, 2020, pp. 183–197.

[56] Thibaut Lutz, Christian Fensch, and Murray Cole. 'PARTANS: An autotuning framework for stencil computation on multi-GPU systems'. In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013).

[57] Ricardo Marques, Hervé Paulino, Fernando Alexandre, and Pedro D. Medeiros. 'Algorithmic Skeleton Framework for the Orchestration of GPU Computations'. In: *Euro-Par 2013 Parallel Processing*. Ed. by Felix Wolf, Bernd Mohr, and Dieter an Mey. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 874–885.

[58] Breno Menezes, Nina Herrmann, Herbert Kuchen, and Fernando Buarque de Lima Neto. 'High-level parallel ant colony optimization with algorithmic skeletons'. Contribution presented at the 13th International Symposium on High-level Parallel Programming and Applications, July 9-10, Porto, Portugal. 2020.

[59] Breno Menezes, Nina Herrmann, Herbert Kuchen, and Fernando Buarque de Lima Neto. 'High-level parallel ant colony optimization with algorithmic skeletons'. In: *International Journal of Parallel Programming* 49.6 (2021), pp. 776–801.

[60] Breno Menezes, Herbert Kuchen, Hugo A. Amorim Neto, and Fernando B. de Lima Neto. 'Parallelization Strategies for GPU-Based Ant Colony Optimization Solving the Traveling Salesman Problem'. In: *2019 IEEE Congress on Evolutionary Computation (CEC)*. 2019, pp. 3094–3101.

[61] Sparsh Mittal and Jeffrey S. Vetter. 'A Survey of CPU-GPU Heterogeneous Computing Techniques'. In: *ACM Comput. Surv.* 47.4 (July 2015).

[62] Gordon E Moore. 'Cramming more components onto integrated circuits'. In: *Electronics* 38.8 (1965), pp. 114–117.

[63] 'MPI: A message passing interface'. In: *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. 1993, pp. 878–883.

[64] NVIDIA Corporation & affiliates. *CUDA C++ Programming Guide*. Version 12.4. url: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model (visited on 13/05/2024).

[65] Tomas Öhberg, August Ernstsson, and Christoph Kessler. 'Hybrid CPU–GPU execution support in the skeleton programming framework SkePU'. In: *The Journal of Supercomputing* 76.7 (2020), pp. 5038–5056.

[66] OpenAcc Community. *OpenAcc*. url: https://openacc-best-practices-guide.readthedocs.io/en/latest/ (visited on 11/04/2024).

[67] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. url: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf (visited on 20/02/2024).

[68] Martin Pedemonte, Sergio Nesmachnow, and Héctor Cancela. 'A survey on parallel ant colony optimization'. In: *Applied Soft Computing* 11.8 (2011), pp. 5181–5197.

[69] Alyson D Pereira, Márcio Castro, Mario AR Dantas, Rodrigo CO Rocha, and Luís FW Góes. 'Extending OpenACC for efficient stencil code generation and execution by skeleton frameworks'. In: *2017 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2017, pp. 719–726.

[70] Alyson D. Pereira, Luiz Ramos, and Luís F. W. Góes. 'PSkel: A stencil programming framework for CPU-GPU systems'. In: *Concurrency and Computation: Practice and Experience* 27.17 (), pp. 4938–4953.

[71] Michael Poldner and Herbert Kuchen. 'Optimizing skeletal stream processing for divide and conquer'. In: vol. PL. 2008, pp. 181–189.

[72] K Raju and Niranjan N Chiplunkar. 'A survey on techniques for cooperative CPU-GPU computing'. In: *Sustainable Computing: Informatics and Systems* 19 (2018), pp. 72–85.

[73] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. 'Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons'. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19. Limassol, Cyprus: Association for Computing Machinery, 2019, pp. 1534–1543.

[74] Shigeyuki Sato and Hideya Iwasaki. 'A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming'. In: *Programming Languages and Systems*. Ed. by Zhenjiang Hu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 79–94.

[75] Fábio Soldado, Fernando Alexandre, and Hervé Paulino. 'Towards the transparent execution of compound opencl computations in multi-cpu/multi-gpu environments'. In: *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I 20*. Springer. 2014, pp. 177–188.

[76] Michel Steuwer. *SkelCL*. url: https://github.com/skelcl/skelcl (visited on 13/02/2024).

[77] Michel Steuwer and Sergei Gorlatch. 'SkelCL: a high-level extension of OpenCL for multi-GPU systems'. In: *The Journal of Supercomputing* 69.1 (2014), pp. 25–33.

[78] Michel Steuwer, Thomas Koehler, Bastian Köpcke, and Federico Pizzuti. 'RISE & shine: Language-oriented compiler design'. In: *arXiv preprint arXiv:2201.03611* (2022).

[79] The Eclipse Foundation. *Xtext Documentation*. https://eclipse.org/Xtext/documentation/. 2020.

[80] Peter Thoman, Florian Tischler, Philip Salzmann, and Thomas Fahringer. 'The celerity high-level api: C++ 20 for accelerator clusters'. In: *International Journal of Parallel Programming* 50.3-4 (2022), pp. 341–359.

[81] Université de Genève. *palabos*. url: https://gitlab.com/unigespc/palabos (visited on 13/02/2024).

[82] Rice University. *High Performance Fortran*. url: http://hpff.rice.edu/versions/index.htm (visited on 05/08/2024).

[83] University of Chicago. *Parsl*. url: https://github.com/Parsl/parsl (visited on 13/02/2024).

[84] University of Edinburgh, University of Glasgow, University of Münster. *lift*. url: https://github.com/lift-project/lift (visited on 13/02/2024).

[85] University of Heidelberg. *DUNE*. url: https://gitlab.dune-project.org/core (visited on 13/02/2024).

[86] University of Innsbruck. *Celerity*. url: https://github.com/celerity/celerity-runtime (visited on 13/02/2024).

[87] University of Muenster. *Muesli4*. url: https://github.com/NinaHerrmann/muesli4 (visited on 18/04/2024).

[88] University of Muenster. *Muesli4*. url: https://github.com/wwu-pi/musket_dsl (visited on 18/04/2024).

[89] University of Pisa, University of Turin. *FastFlow*. url: https://github.com/fastflow/fastflow (visited on 18/04/2024).

[90] University of Valladolid. *EPSILOD*. url: https://gitlab.com/trasgo-group-valladolid/controllers/ (visited on 13/02/2024).

[91] *Userguide SkePU*. https://skepu.github.io/docs/userguide_skepu.pdf. (Visited on 19/12/2023).

[92] Pedro de Almeida Amaral Ramos Valente. 'A Cuda Backend For Marrow and its Optimisation via Machine Learning'. MA thesis. NOVA University Lisbon, 2022.

[93] Fabian Wrede and Steffen Ernsting. 'Simultaneous CPU–GPU execution of data parallel algorithmic skeletons'. In: *International Journal of Parallel Programming* 46.1 (2018), pp. 42–61.

[94] Fabian Wrede, Breno Menezes, and Herbert Kuchen. 'Fish school search with algorithmic skeletons'. In: *International Journal of Parallel Programming* 47 (2019), pp. 234–252.

Part II

Included Publications

8

Publication Overview

The findings presented in Part I are the summary of the individual contribution to the publications listed in Part II and, when necessary, include further execution time measurements. All papers presented are published under the terms of the Creative Commons CC BY license, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. The listed publications have always been first presented at the International Symposium on High-Level Parallel Programming and Applications and then published in the International Journal of Parallel Programming. The listed ratings are based on the Core ranking [31].

Journal publication

No.	Publication	CORE
J1	Chapter 9 [59]: Breno Menezes, Nina Herrmann, Herbert Kuchen, and Fernando Buarque de Lima Neto. 'High-level parallel ant colony optimization with algorithmic skeletons'. In: *International Journal of Parallel Programming* 49.6 (2021), pp. 776–801	B
J2	Chapter 10 [41]: Nina Herrmann, Breno Menezes, and Herbert Kuchen. 'Stencil calculations with algorithmic skeletons for heterogeneous computing environments'. In: *International Journal of Parallel Programming* 50.5-6 (2022), pp. 433–453	B
J3	Chapter 11 [39]: Nina Herrmann and Herbert Kuchen. 'Distributed Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments'. In: *International Journal of Parallel Programming* 51.2-3 (2023), pp. 172–185	B
J4	Chapter 12 [37]: Nina Herrmann, Justus Dieckmann, and Herbert Kuchen. 'Optimizing Three-Dimensional Stencil-Operations on Heterogeneous Computing Environments'. Accepted for publication in the International Journal of Parallel Programming. 2024	B

[31] `https://portal.core.edu.au/jnl-ranks/` and `https://portal.core.edu.au/conf-ranks/`

Conference Contributions

No.	Publication	CORE
C1	Chapter 9 [58]: Breno Menezes, Nina Herrmann, Herbert Kuchen, and Fernando Buarque de Lima Neto. 'High-level parallel ant colony optimization with algorithmic skeletons'. Contribution presented at the 13th International Symposium on High-level Parallel Programming and Applications, July 9-10, Porto, Portugal. 2020	C
C2	Chapter 10 [40]: Nina Herrmann, Breno Menezes, and Herbert Kuchen. 'Stencil calculations with algorithmic skeletons for heterogeneous computing environments'. Contribution presented at the 14th International Symposium on High-Level Parallel Programming and Applications, July 12-13, Online Event. 2021	C
C3	Chapter 11 [38]: Nina Herrmann and Herbert Kuchen. 'Distributed Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments'. Contribution presented at the 15th International Symposium on High-Level Parallel Programming and Applications, July 7-8, Porto, Portugal. 2022	C
C4	Chapter 12 [36]: Nina Herrmann, Justus Dieckmann, and Herbert Kuchen. 'Optimizing Three-Dimensional Stencil-Operations on Heterogeneous Computing Environments'. Contribution presented at the 16th International Symposium on High-Level Parallel Programming and Applications, June 29-30, Cluj-Napoca, Romania, 2023	C

9

High-Level Parallel Ant Colony Optimization with Algorithmic Skeletons

Breno A. de Melo Menezes [*] · **Nina Herrmann**[*] · **Herbert Kuchen**[*] · **Fernando Buarque de Lima Neto**[†]

Citation Breno Menezes, Nina Herrmann, Herbert Kuchen, and Fernando Buarque de Lima Neto. 'High-level parallel ant colony optimization with algorithmic skeletons'. In: *International Journal of Parallel Programming* 49.6 (2021), pp. 776–801.

Abstract Parallel implementations of swarm intelligence algorithms such as Ant Colony Optimization (ACO) have been widely used to shorten the execution time when solving complex optimization problems. When aiming for a GPU environment, developing efficient parallel versions of such algorithms using CUDA can be a difficult and error-prone task even for experienced programmers. To overcome this issue, the parallel programming model of *Algorithmic Skeletons* simplifies parallel programs by abstracting from low-level features. This is realized by defining common programming patterns (e.g. map, fold, and zip) that later on will be converted to efficient parallel code. In this paper, we show how algorithmic skeletons formulated in the domain-specific language *Musket* can cope with the development of a parallel implementation of ACO and how that compares to a low-level implementation. Our experimental results show that *Musket* suits the development of ACO. Besides making it easier for the programmer to deal with the parallelization aspects, *Musket* generates high-performance code with similar execution times when compared to low-level implementations.

Keywords Algorithmic Skeletons · Ant Colony Optimization · High Performance Computing.

[*]University of Münster, Germany
[†]University of Pernambuco, Brazil

9.1 Introduction

Nature-inspired metaheuristics have been widely used to solve complex optimization problems [23]. When tackling combinatorial problems, approaches using Ant Colony Optimization (ACO) have been widely exploited and can be often found in literature [16]. Initially proposed by Marco Dorigo in his Ph.D. thesis, it is inspired by the social behavior of ant colonies when searching for new sources of food and their ability to find the shortest path between the colony and the food [9]. ACO was initially created to solve problems such as the Traveling Salesperson Problem (TSP), achieving satisfactory results.

In ACO, the process of building a solution is done in several steps, including a certain number of probability calculations. The workload in this process is proportional to the size of the problem and the number of possibilities to be explored. Considering the TSP problem, the number of possible tours grows exponentially as the number of nodes in the graph increases. The same occurs for packing problems such as the Multidimensional Knapsack Problem (MKP) and the Bin Packing Problem (BPP), where the number of combinations rises quickly as more items have to be packed. In order to explore more of these possibilities in the search space, more ants are required in the colony and, therefore, the computational costs increase substantially. Knowing that a considerable part of the computations is performed during the solution construction phase (path construction or packing) and that the runtime increases when tackling a bigger instance of such problems with several ants, it is mandatory to speed up the program in order to run the algorithm in a reasonable amount of time without losing the quality of the solutions.

Parallel implementations of ACO have been introduced aiming for different high-performance hardware, such as multi-core CPUs and GPUs. Low-level frameworks such as OpenMP, MPI, and CUDA provide many tools for programmers, assisting the development of such parallel versions of the algorithm. The tools provided by CUDA help programmers develop parallel programs for Nvidia GPUs. Nevertheless, some expertise is necessary to generate high-performance code. Programmers must be aware of data transfers, synchronization points, and many other issues that make the development of the program difficult and error-prone.

Aiming to ease the development of such parallel algorithms, high-level parallelization tools provide means to profit from the use of high-performance hardware without the issues inherent to low-level programming. For example, some tools allow the use of predefined common programming patterns, known as algorithmic skeletons [4], here referred to as skeletons. They represent common operations, such as *map, zip*, and *reduce*, and can be used in a program. Musket converts those patterns to parallel code. This

way, the programmer's job is to translate the methods from the original algorithm into predefined operations which are translated to parallel code.

Muenster Skeleton Tool for High-Performance Code Generation (Musket) is an approach based on a Domain Specific Language (DSL) created to speed-up the development of parallel programs [20]. By using it, programmers are able to create code by first writing it in the DSL *Musket* and then converting the program into parallel CPU or GPU code. Created as a general purpose tool, *Musket* has already been applied and tested in several problems including metaheuristics, presenting promising performance results compared to other parallelization approaches [28].

In this paper, we investigate the use of *Musket* to create a parallel GPU version of ACO in order to understand and compare how it relates to low-level implementation in terms of performance and development complexity. The identification of positive and negative sides of using the general purpose structures available in *Musket* may also serve as a base for the future development of the framework.

Our paper is organized as follows: The basics about ACO are displayed in Section 9.2. In Section 9.3 we give an overview of related work. The description of *Musket* and how it was applied in this work can be found in Section 9.4. Experiments are detailed in Section 9.6 together with the results. In Section 9.7 we put together the conclusions of this work and point out future work.

9.2 Ant Colony Optimization

ACO is a metaheuristic in which artificial ants cooperate with each other in order to solve complex discrete optimization problems. It was initially tailored to solve the TSP, therefore it requires some adaptations in order to be applied in other contexts, such as the BPP and MKP. Details about each adaptation and the GPU implementation are described in the following.

9.2.1 ACO Solving the Traveling Salesman Problem

In the case of TSP, the objective is to find the shortest tour in a graph, starting from a random node, visiting each node once and only once, and coming back to the original node [10, 11]. In order to solve such a problem, each ant in the colony tries to create a tour and at the end of each iteration, they share their success through pheromone deposits. More successful ants, the ones that generated shorter tours, deposit more pheromones on the visited edges. The pheromone is what will attract other ants in the

following iteration, helping them to generate similar tours based on the success of ants from previous iterations.

The process described above can be divided into two steps which compose the execution of ACO, namely tour construction and pheromone deposit. During the tour construction, each ant must create a tour starting at a random node. The decision where to go next from the current node i is done in a probabilistic manner, taking into account the distance and the amount of pheromone between the current node and a candidate node. The probability is calculated as shown in Equation 9.1.

$$p_{i,j} = \frac{[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}}{\sum_{l \epsilon N}[\tau_{i,l}]^{\alpha}[\eta_{i,l}]^{\beta}} \quad \forall j \epsilon N \tag{9.1}$$

where α and β are the parameters used to determine the influence of pheromone quantity and distance between the nodes over the probability, respectively. $\tau_{i,j}$ is the pheromone at the edge from node i to j, $\eta_{i,j}$ is $1/d_{i,j}$, where $d_{i,j}$ is the distance from node i to j. N is the set of unvisited nodes that can follow node i. $p_{i,j}$ is the probability that the ant goes from node i to j. The current node i and visited nodes have a probability equal to zero.

Once each ant has created its tour, the fitness of each ant will be equal to the total distance traveled. Afterward, the pheromone update will take place. The first step is the pheromone evaporation where each edge loses a certain quantity of pheromone according to the following assignment (Equation 9.2).

$$\tau_{i,j} := (1 - \rho) * \tau_{i,j} \tag{9.2}$$

where, $\rho \in [0..1]$ defines the evaporation rate and $\tau_{i,j}$ is the amount of pheromone between nodes i and j. ρ is used to control the amount of pheromone, enabling the algorithm to focus more on new trails. After the evaporation, it is time for each ant to deposit pheromone on the tour it has created, according to the following assignment (Equation 9.3):

$$\tau_{i,j} := \Delta t + \tau_{i,j} \tag{9.3}$$

where $\Delta t = 1/q_k$, and q_k is the length of the round tour of ant k. By doing so, ants that generated shorter tours deposit more pheromone at visited edges than the ones that generated longer tours.

This process is repeated through as many iterations as needed. Although the pheromones are used to attract ants and help them create tours similar to previous successful ones, the probabilistic way of choosing the next step allows ants to create distinct paths and therefore generate diversity.

9.2.2 ACO Solving the Bin Packing Problem

The bin packing problem (BPP) is an NP-complete combinatorial optimization problem where a set of items with a given volume has to be packed into as few bins with a fixed capacity as possible. A certain quantity tells how many identical copies of an item are available. Being a combinatorial problem, the BPP can also be solved using ACO. The goals are similar but the process is a bit different because of the weight limitation imposed in the BPP and the method used to build a solution. In this section, these differences will be explained together with the approach used in this work to build a solution. The algorithm used here follows the approach of Levine and Ducatelle [17].

Differences between both problems emerge directly from the setup. When solving the BPP, the algorithm has to ensure continuously that the capacity of the currently considered bin is not violated, while the TSP has no such restrictions. Nevertheless, the basic structure of the algorithm remains unchanged. Modifications of the algorithm come from the fact that the order of inclusion is not important anymore. Instead, the grouping of items that fit together in one bin is relevant. Therefore, all items included in the current bin should be considered when calculating the probability of selecting the next item. Algorithm 15 illustrates the process of calculating τ and η.

Once τ and η are calculated for each item, the probability of choosing item j is calculated in the same way as explained previously. Items that have already been included ($item.volume = 0$) and items that would exceed the bin's capacity will be excluded and their probabilities are set to zero. If there are still items available but none fits into the current bin, a new bin is started. The process is repeated until all items have been packed and the fitness of the solution is equal to the number of bins used in this solution.

Using the same concept, the pheromone deposit process considers the fitness of one solution and re-visits each of the bins created to deposit pheromones. If items A, B and C are part of a bin created in a solution that resulted in good fitness, the connections between these items will receive the same high amount of pheromone. This method of depositing pheromones, informs ants in future packing phases that packing A, B and C together helped create a good solution and increases the chances of them being packed together again.

Algorithm 6: Calculate Probabilities Pseudo-code

```
1  current_bin = empty
2  avail_capacity = capacity
3  for j ← items do
4      phero_sum = 0.0
5      while (j.quantity > 0) do
6          if (j.volume <= avail_capacity) then
7              avail_capacity j.volume
8              for i ← current_bin do
9                  phero_sum += pheromones[i][j]
10             current_bin = current_bin ∪ {j}
11         else
12             current_bin = empty
13             avail_capacity = capacity
14     τ_j = phero_sum/number_of_items_in_current_bin
15     η_j = j.volume^β
```

Other parts of the algorithm, such as the pheromone evaporation, are performed in the same way as mentioned previously.

9.2.3 ACO Solving the Multidimensional Knapsack Problem

The multidimensional knapsack problem (MKP) is as the previously discussed optimization problems NP-hard. In its original form, a set of items ($J = \{1, 2, ..., n\}$) is given together with a set of constraints ($I = \{1, 2, ..., m\}$). As each item has a value, the goal is to select a subset of J aiming to maximize profit respecting all given constraints. The approach used in this work to solve the MKP is inspired by the one proposed by Soh-Yee Lee and Yoon-Teck Bau [22]. In this approach, the probability of ant k adding a new item to the knapsack is calculated considering the partial solution ($\tilde{S}_k(t)$) constructed until step t. It can be calculated using Equation 9.4:

$$p_j^k(t) = \begin{cases} \dfrac{[\tau_j(t)]^\alpha \cdot [\eta_j(\tilde{S}_k(t))]^\beta}{\sum_{l \in A_k(t))}[\tau_l]^\alpha [\eta_l(\tilde{S}_k(t))]^\beta}, \text{if } j \in A_k(t) \\ 0, \text{otherwise} \end{cases} \tag{9.4}$$

where $p_j^k(t)$ is the probability of ant k choosing item j at step t, $A_k(t)$ is the set of available items for ant k at step t, $\tau_j(t)$ is the amount of pheromones assigned to item j

at step t and $\eta_j(\tilde{S}_k(t))$ is the heuristic factor which is calculated considering the value of item j and how it complies with the constraints as shown in Equation 9.5:

$$\eta_j(\tilde{S}_k(t)) = \frac{v_j}{\bar{\delta}_j(k,t)} \tag{9.5}$$

where v_j is the value of item j and $\bar{\delta}_j(k,t)$ is the average tightness considering all constraints and the actual state of the partial solution $\tilde{S}_k(t)$. The tightness of an item j for a certain constraint i is defined by Equation 9.6:

$$\delta_{ij}(k,t) = \frac{r_{ij}}{c_i - \sum_{l \in \tilde{S}_k(t)} r_{il}} \tag{9.6}$$

where r_{ij} is the size of item j for constraint i and c_i is the capacity of the knapsack for dimension i. Summing the values for each dimension and dividing it by the number of constraint dimensions m gives the average tightness $\bar{\delta}_j(k,t)$.

Once all probabilities are calculated, the same probabilistic method as mentioned before is used to select the next item to be inserted in the knapsack. The process is repeated until the knapsack is full and none of the remaining items fits. Differently from the BPP, the solution proposed for the MKP can be smaller than the size of the initial set of objects.

In the approach used in this work, pheromone values are an association with one item and a construction step. Unlike the BPP, where groups of items are important, in MKP the order in which items are added has more value. The amount of pheromone to be deposited to item j by ant k is calculated using Equation 9.7:

$$\Delta\tau_j^k(t) = \begin{cases} QL_k, \text{if item } j \text{ is used by ant } k \\ 0, \text{otherwise} \end{cases} \tag{9.7}$$

where Q is constant (defined by $1/\sum_{j=1}^{n} p_j$) and L_k is the fitness of the solution constructed by ant k, which is equal to the sum of the values of all items inserted.

9.2.4 GPU-ACO

Targeting a GPU environment, we present here one possible implementation approach for parallelizing ACO using CUDA. The concepts explained in this section are applied in the same way for both problems investigated, adapting only to their peculiarities. The steps that compose the ACO algorithm as described above are quite simple and the general process makes the algorithm suitable for parallelization. Even so, extra care is required when dealing with the same steps in a parallel way.

The CUDA framework makes it possible to run sequential instructions on the CPU, while the computational-intensive tasks can run on the GPU in parallel. In our approach, the whole algorithm runs on the GPU and, therefore, operations such as routing and pheromone updates are declared as CUDA kernels. One advantage of this approach is that no data transfer between host and device is necessary along the iterations. These data transfers are performed at the beginning (reading data and copying to GPU) and at the end of the execution (retrieving results from GPU to host). A general view of the proposed implementation used to solve the TSP is represented in Algorithm 7.

Algorithm 7: Pseudo-code GPU-ACO

```
1  initialize_ACO
2  copy_data_to_Device();
3  initialize_GPU
4  calculate_distance_kernel«<n_blocks, n_threads»>(...);
5  create_random_generators_kernel«<n_blocks, n_threads»>(...);
6  n_threads = warp_size;
7  n_blocks = n_ants / warp_size;
8  iterations = 0;
9  while (iterations++ < max_iterations) do
10     solution_construction_kernel«<n_blocks, n_threads»>(...);
11     pheromone_evaporation_kernel«<n_cities,n_cities»>(...);
12     pheromone_deposit_kernel«<n_ants,n_cities»>(...);
13 copy_results_to_Host();
14 clear_GPU();
```

The first step of this implementation is the initialization of the structures that compose the problem. It includes reading the data from a file that contains the x and y coordinates of each city present in the graph of the TSP instance or the volumes and quantities of the items for the BPP. Afterward, the data can be copied to the GPU and, already on the device side, other data structures necessary to run ACO can be created directly on the GPU, i.e. random number generators, distance matrix, pheromone matrix and route matrix. After everything is created and initialized properly, the algorithm can loop through its iterations and perform the solution constructions (tour or packing) and pheromone updates.

The solution construction is the first step in an iteration and also the most demanding task of ACO. In order to create a parallel version of it, a straightforward approach was chosen where the calculations necessary for calculating one solution (e.g. one route) are performed by one thread. The number of CUDA blocks will be determined by dividing

the number of ants in the colony by the desired number of threads per block, meaning that it can be changed and adapted according to the necessity and capabilities of the hardware. The simplicity of implementing this approach is one of its positive aspects. Once there is a sequential implementation of ACO, it is not too difficult to port it to CUDA using this approach.

After the solution-construction phase, the pheromone update is broken down into two different steps. The first one is the pheromone evaporation, in which each connection in the graph loses a certain amount of pheromone as explained before in Equation 9.2. In our parallel implementation, one CUDA block is generated for each city and, inside this block, one thread is generated for each other city in order to decrease the amount of pheromone (Listing 9.1). For the BPP problem, the same approach is chosen with connections between each type of item available to be packed in bins.

```
1 __global__ void evaporation_kernel(double* c_phero) {
2   int edge_index = blockIdx.x * blockDim.x + threadIdx.x;
3   double RO = 0.5; //Evaporation rate = 50%
4
5   if(blockIdx.x !=  threadIdx.x){  // no edge from city_i to itself
6     c_phero[edge_index] = (1 - RO) * c_phero[edge_index];
7   }
8 }
```

Listing 9.1: Evaporation kernel (TSP)

The pheromone deposit phase starts with one CUDA block assigned to one ant in the colony. Each thread inside a block is responsible for updating an edge visited by the ant in the tour constructed previously. As the pheromone matrix is stored in the GPU's global memory and is being updated by different threads, racing conditions might appear. In order to overcome that, the pheromone deposit is performed using CUDA's atomic operations, guaranteeing the integrity of the data without losing performance. For the BPP problem the pheromone value of items that are packed together in good solutions in one bin is increased. In this way, items that are often packed together in good solutions have a connection with a higher pheromone value.

The process is repeated for a number n of iterations. In the end, the best results are copied to the host, and the execution is ended. With such a simple approach it is already possible to achieve a considerable speedup when compared to sequential implementations.

9.3 Related Work

The use of ACO to solve combinatorial problems has been already deeply investigated. Levine and Ducatelle [17], used pure ACO to solve many instances of the BPP and Cutting Stock Problem (CSP). They state that exact methods work well for small instances of such problems, but would require too much time to solve bigger instances. In this situation, ACO comes as a convenient tool that is able to achieve good results in a reasonable amount of time. Furthermore, they also state the need to have a certain number of evaluations in order to achieve good results. For bigger instances, the necessity of having these evaluations together with the complexity of generating solutions results in a notable increase in the execution times. Although they do not apply any parallelization, it would come to hand when solving these bigger instances as the execution take already a considerable amount of time compared to the smaller problems.

Low-level parallel implementations of ACO have been already investigated in the literature. The approaches include mainly the introduction of data parallelism into the code, like in the work of Uchida et al. [25]. Other works focus on improvements in the algorithm that would favor data parallelism. Cecilia et al. developed a new mechanism called I-Roulette in order to enhance parallelism during the path creation process. Furthermore, they introduce strategies for parallelization of the pheromone update process suitable for GPU architectures [3].

Another approach to reduce the execution time of the algorithm is to improve the parallelization itself. Instead of modifying the algorithm by introducing new mechanisms that would simplify the execution, the idea is to make better use of the hardware available. This can be achieved by organizing the structures needed by the algorithm in an optimal way and dividing the work in such a way that hardware use would be optimized. This can be done for example by introducing different levels of parallelism. In ACO, this can be done by parallelizing not only the job of each ant but also all internal calculations that belong to tour construction as demonstrated by Menezes et al. [18]. Also, in another work, the same authors investigate the use of atomic operations to enhance the process of updating the pheromone matrix [5]. The results indicate that different levels of parallelism can be useful according to the size of the problem and that the use of atomic operations can speed up the pheromone update phase.

Rieger et al. introduced *Musket*, a DSL for parallel programming [20]. Their idea is to offer a language with algorithmic skeletons built in and with a syntax similar to C++ in order to help programmers to write high-performance distributed parallel programs without the need of expertise in low-level frameworks. High-performance low-level code

for different architectures (Multi-core CPUs, GPUs, or clusters) is generated from Musket files. The authors point out the benefits of using a DSL compared to other high-level approaches. Also, among some examples, the Fish School Search (FSS) metaheuristic is used as a benchmark. Further analysis of FSS and *Musket* is done by Wrede et al.[28]. Both studies show the possibility of using such general-purpose tools for application in the metaheuristics field. The major criticism of using high-level frameworks has been the possible loss in performance. The present paper serves to evaluate the performance of a skeleton-based implementation and the handwritten implementation previously discussed [18, 5].

There are also many other approaches providing high-level parallel programming based on algorithmic skeletons, including Fastflow [1], SkePU [19], Muesli [12, 13], eSkel [2] and many others.

9.4 Musket

Musket is a DSL which enables programmers to develop parallel applications and generate optimized code without requiring knowledge about low-level parallel programming frameworks. For interested readers, the code can be found in a public repository [7].

The syntax of *Musket* is based on C++ which is widely used for high-performance computing. It was defined using the Xtext framework and it includes a parser and an editor that can be incorporated into Eclipse [24]. In this way, programmers can use helpful features such as syntax highlighting, code completion, and validation. Creating parallel programs in *Musket* is simplified in multiple ways. Common parallel programming structures are simplified by representing them as skeletons. Moreover, the division and allocation of data structures to distinct processes is done by the code generator which transforms Musket code to C++ with CUDA operations in case that GPU code is generated. Furthermore, it is totally abstracted from specifying the number of threads to be started. Those and other advantages become more apparent by illustrating an exemplary program (Listing 9.2).

A *Musket* program is divided into four parts, namely *meta-information*, *data structure declaration*, *user function declaration*, and *main program declaration*. The *meta-information* block (lines 1-5) specifies for which type of hardware code should be generated. Firstly, the platform argument distinguished between a program for GPUs or CPUs. In the case of stating multiple platforms, multiple programs are created. For our application context, only the GPU code generator is required. Afterward, the number of processes, cores,

and GPUs that shall be used are specified. Information about the targeted architecture is essential to generate a distribution for data structures that is efficient and to organize the parallel execution of the skeletons.

In *Musket*, global data structures are declared before writing functions in the *data structure declaration* block (line 7). On the one hand, for each additional data structure type that is offered, the effort for the implementation rises. On the other hand, it is possible to include additional information for the data structures e.g. on the data distribution. Here, the elements of the array are distributed among the GPUs. Available distribution modes are *dist* for distributed, *copy* to make the whole data structure available for all processes, and local which is a specific form of copy where no global copy needs to be created. Similar to the specification of arrays, matrices can be created, which require the same parameters despite having two parameters to specify the number of rows and columns.

```
1  #config PLATFORM GPU CPU_MPMD
2  #config PROCESSES 4
3  #config CORES 24
4  #config GPUS 4
5  #config MODE release
6
7  array<int,16384,dist> a;
8
9  int double_values(int i){
10         return i + i;
11 }
12
13 main{
14         mkt::roi_start();
15         a.mapInPlace(double_values());
16         mkt::roi_end();
17 }
```

Listing 9.2: Musket code sample

The *user-function declaration* part (lines 9-11) includes custom user functions which will be passed as arguments to the skeleton call and in the final program executed among the nodes and cores available. Inside user functions, programmers can make use of control structures, such as *if-else statements* and *for loops*, moreover, a selection of C++ library methods and external functions are available. Global structures declared in the previous section can be used either with a local index or a global index. Moreover, local variables can be created.

In the last part, similar to C programs, a main function is declared which defines the entrance of the program. In the example, lines 13-17 contain the *main program declaration*. There, general instructions of the program are listed using control structures, musket functions, and parallelization instructions in the form of algorithmic skeletons. Musket functions are typically used functions for writing parallel programs which do not need to be executed in parallel, for example measuring the runtime and getting maximal and minimal values of data types. For example, in line 14 the time measurement is started. Wrapping such functions relieves the user of the framework to think about target-specific functions. In order to simplify parallelization, *Musket* offers (different versions of) the Skeletons *fold*, *map*, *reduce* (which in contrast to fold only accepts a selection of commonly used reduction operators), *zip*, *gather*, *scatter* and *shift partition*. For *zip* and *map*, in-place and index variants and their combinations are available. The given example doubles the value of every element of the array. The implementation of the ACO algorithm will show how those structures can also be used for more complex programs.

The written DSL code will then be transformed into low-level code. With each program generated (in the case of multiple platforms), CMake Files and execution scripts are generated. The generated code is not meant to be further adjusted.

9.5 Our Proposal

ACO is a metaheuristic which is suitable for parallelization. Many tasks, such as the path or packing construction, are independent of each other as each ant is able to create its own solution without interference from other ants. Even though, the task of creating a parallel version for it can be quite challenging. A few steps require extra care since reduction steps are executed before performing general calculations. For example, as shown in Equation 9.1, the probability is calculated using the product of the amount of pheromone and the distance divided by the sum of all products. Furthermore, steps like the pheromone-update phase include changes that shall be performed by each ant in the colony over the pheromone matrix, which is a structure used by the whole colony. Therefore, the programmer must be careful to avoid race conditions and perform the right data transfers without harming performance.

In order to overcome such difficulties, *Musket* is helpful. Generally, high-level frameworks have the advantage that the user does not need expertise in the specific area (in this context parallel programming). Musket DSL code is also more concise than e.g. C++ code with calls to a (skeleton) framework.

However, using a DSL also has its disadvantages. The developer of the DSL has to decide which functionalities are essential. Missing necessary functionalities limit the user of the DSL. While, in contrast, including too many functionalities increases the complexity of the code generator and confuses inexperienced users.

In order to evaluate the usability and practicability of a high-level framework for the exemplary case of the ACO algorithm, a *Musket* program will be compared to a handwritten program. The aspects of major interest are the performance of the programs and the complexity of the syntax and structure provided by the different approaches. As part of the comparison the creation process of a program implementing the ACO algorithm in *Musket* will be described, to illustrate the advantages and disadvantages of using a high-level framework.

9.5.1 Musket-ACO

In the following, interesting aspects of the high-level implementation of the ACO algorithm will be discussed. Of particular interest is how the single steps proposed in the abstract solution approach in Algorithm 7 are translated to one or multiple skeletons. Taking two problems enriches the analysis since similarities of the implementations independent from the problem can be highlighted.

The first difference between the algorithms is that for solving the TSP problem firstly the distances between all cities are calculated, and secondly, the 32 closest cities are determined. Those calculations speed up the route-searching process of the algorithm since it favors close cities. The steps are executed once; therefore, they can be considered as data pre-processing. For the BPP and the MKP, no data pre-processing is necessary with the used data set.

In Listings 9.3, 9.4 and 9.5 extracts of the program are shown. Both programs nest those steps in a for loop to find a good solution. The number of iterations is left to the programmer. A program that stops when a sufficiently good result is achieved would also be feasible. Both programs start in line 1 with the most calculation-intensive step. For solving the TSP problem each ant calculates one possible route to visit all cities, for the BPP problem each ant packs bins until all items are packed. The parallelization of those methods is obvious as each ant can independently perform calculations. The return value of the two methods `route_kernel` and `packing_kernel` is different. The program solving the TSP problem writes the sequence of visited cities into an array and does not save the overall distance. The program solving the BPP problem returns the number of bins used. The functions require different parameters, however, from these two problems it can be

concluded that the first part of solving a problem with the ACO algorithm is feasible by mapping each result an ant produces to a result array.

```
antss.mapIndex(route_kernel());
d_fitness.mapIndexInPlace(update_delta_phero());
int new_bestroute = d_fitness.reduce(min);
antss.mapIndex(update_best_sequence(bestroute));
city.mapIndex(update_phero());
```

Listing 9.3: Skeleton calls of TSP program

```
d_fitness.mapIndexInPlace(packing_kernel(itemtypes, itemcount, BETA, bin_capacity));
int new_best_fitness = d_fitness.reduce(min);
if (new_best_fitness < best_fitness)
best_fitness = new_best_fitness;
d_phero.mapIndexInPlace(evaporation_kernel(itemtypes, EVA));
d_fitness.mapIndexInPlace(update_pheromones_kernel(itemtypes, itemcount, bin_capacity));
```

Listing 9.4: Skeletons calls of BPP program

```
d_ant_fitness.mapIndexInPlace(generate_solutions(n_objects, n_constr));
best_fitness = d_ant_fitness.reduce(max);
d_ant_fitness.mapIndexInPlace(update_best_solution(best_fitness,n_ants, n_objects));
d_pheromones.mapIndexInPlace(evaporate(evaporation));
d_ant_solutions.mapIndexInPlace(pheromone_deposit(n_objects, n_ants));
```

Listing 9.5: Skeletons calls of MKP program

Obviously, the program solving the TSP uses five skeleton calls while the program solving the BPP uses four skeleton calls. This shows that although the same algorithm is used the steps for finding a solution are adjusted depending on the problem solved. In this case, the changes are caused due to the differences in the steps executed to find a good solution and measure the fitness of a solution.

The differences in finding a good solution are that for the TSP problem good results are found by choosing from the 32 closest cities. For the BPP problem, good results are found by first packing the heaviest object and proceeding by favoring as heavy objects. Both approaches are influenced by the pheromone and a random factor. The fitness of the TSP problem is measured by the overall distance while the fitness of the BPP problem is measured by the number of bins. Therefore, for the BPP problem fitness is required during calculating a solution. Moreover, calculating the fitness for the BPP problem does not require a lot of memory space as it is sufficient to increase an integer for each bin used. In contrast, this is not the case for the TSP problem. Calculating the distance of the whole route is not necessary as any route with all cities is a valid solution. Moreover,

calculating the fitness while finding a solution would result in additional read operations of the data structure storing the distance between different cities. Therefore, it would be required to load an array that stores the distance between all cities in the limited Graphics Processing Unit (GPU) memory. Instead, the implementation loads an array with the closest 32 cities calculated in the data pre-processing which requires remarkably less memory. This is essential as fewer threads can be started with less memory available.

Therefore, the program solving the TSP problem uses another map skeleton to calculate the distance of each route found by ants (Listing 9.3 l.2). Depending on the distances, values for updating the pheromone are saved. This skeleton is not necessary for the BPP problem as the number of bins used are already saved in the data structure `d_fitness`. Both programs continue by finding the minimum of all found solutions (Listing 9.3 l.3, Listing 9.4 l.2). For other problems this could also be the maximum (e.g. the Knapsack problem searches for the maximum value of packed items).

For the solution of the BPP problem, the following steps are to evaporate the previous pheromone, and to update the pheromone between items (Listing 9.4 l.5+6). For the TSP problem it was decided to evaporate the pheromone while updating the pheromone in the same skeleton call (Listing 9.4 l.5). No runtime differences could be found for having two or one skeleton call. Starting two skeletons allows more parallelization in the generated code as all entries can be changed at the same time, but cost more time as the start of an additional kernel requires time. Having one skeleton call saves time for starting kernels, but allows for less parallelization as not all entries are updated in parallel but dependent on the fitness all entries used by that solution are updated by one thread.

As a result, the *Musket* implementation of ACO is very similar to the low-level approach. Adjustments have been made according to the necessities and restrictions imposed by Musket. For example, atomic operations applied in the pheromone update phase were not used in the high-level implementations because they are not implemented in *Musket* until now. For this specific case, the lack of this feature has no impact on the final program since other approaches can be used to perform the same steps. One approach is to run the pheromone updates sequentially which would in theory consume more time. The second approach is to perform the updates in parallel without any lock controls. In practice, the sequential approach does not consume much more time and does not change the fact that the pheromone update consumes a minimal fraction of the total execution time. For the parallel approach, there is a small chance of two distinct threads trying to update the same value and, if they do, the loss of information is minimal and has no major impact on the behavior of the algorithm.

Furthermore, parts of the code needed to be split into separate steps in order to fit Musket's structure. The implemented programs for the problems varied in the number of skeletons used, even so, they are based on the same algorithm.

9.6 Our Case Study

The comparison between the parallel implementations of ACO used in this work can be done from different perspectives. As we propose the use of a high-level parallelization, the first point to analyze is the applicability of the tool. As mentioned in the previous section, the skeletons available in *Musket* suffice to create high-level programs of ACO. Another aspect that can be discussed is the usability of *Musket*. In *Musket*, 215 lines were needed to solve the TSP, 175 to solve the BPP, and 168 to solve the MKP, compared to 374, 356 and 334 lines of the low-level implementation, respectively. For this comparison, the code methods that read the data from files were excluded in both programs.

Furthermore, the cyclomatic complexity was measured for both programs [21]. We are aware that it does not reflect directly the effort to implement the programs, but it depicts evidence and is commonly used because of its simplicity. In this evaluation, the *Musket* program has a cyclomatic complexity of 54 compared to a slightly higher complexity of 67 of the low-level implementation, meaning that it has less linearly independent paths, which makes it simpler and easier to maintain.

Most of the difference between the codes from both versions comes from the *main* method since kernel calls do not have to be written in *Musket*. But most importantly it is abstracted from all data transfers, which consume several code lines in the low-level program. Moreover, it should be considered that the lines-of-code metric is admittedly debatable since it fails to evaluate how complex the written lines are. In addition to requiring only 57% and 49% of the lines of code compared to the low-level program, *Musket* abstracts from complex decisions such as choosing the number of threads or moving data between the CPU and GPU. Therefore, it could be argued that creating a *Musket* program does not only require fewer lines of code but additionally is written faster since the programmer is relieved from complex low-level decisions as in a low-level implementation using pure CUDA.

Another aspect to be evaluated in this work is the runtime. In order to test this aspect in both parallel implementations mentioned in this work, an NVIDIA GeForce RTX 2080 Ti accelerator containing 4352 CUDA cores, 11 GB memory, and running CUDA 7.5 was

used. Furthermore, we applied the implementations to two different problems to evaluate the behavior in different scenarios. The programs and results are publicly available [6].

9.6.1 TSP Experiments

The first experiments performed in this work include instances of TSP taken from popular repositories with different graph sizes [27, 26]. The selected TSP instances have different numbers of vertices so that the performance of low- and high-level ACO implementations can be evaluated on various problem sizes (Table 9.1). Unfortunately, the selection from the different repositories does not grow linearly in size. While some maps are very close in size, e.g. qa194 and d198, other maps have big differences, e.g. d1291 and pr2392.

Instance	dj38	qa194	d198	lin318	pcb442	rat783	pr1002	pcb1173	d1291	pr2392
# vertices	38	194	198	318	442	783	1002	1173	1291	2392

Table 9.1: TSPLIB

For experimental purposes, each version was tested using different colony sizes (1024, 2048, 4096, and 8192 ants) for all TSP instances in order to simulate different levels of computational load. An important remark is that in this work, the fitness achieved is not relevant and the focus of the analysis is rather on the execution time. Since in essence, both implementations represent the same algorithm and just vary in the parallelization approach and both achieve similar fitness values when using the same setup.

Aiming at a fair comparison between both approaches, the execution times registered in the experiments are denoted in seconds and represent the whole execution of the algorithm, including the initialization process and data transfers between the host and GPU. The runtimes are the average of 30 runs excluding the first runs due to the warm-up of the GPU. Figure 9.1 puts the values beside each other graphically for an easier comparison.

The results show that for the smaller problems, where fewer resources are needed, both implementations achieve very similar, almost identical, results. When more resources are needed, the low-level version tends to scale better and provide shorter execution times. The values for the last and biggest map are excluded in this graph since they impede the readability and will be discussed afterward.

Intuitively, the graph underlines how close the runtime values from the low-level program and the *Musket* program are. Also, as the TSP instance increases in size, the low-level program tends to be slightly faster, especially with higher values for the colony

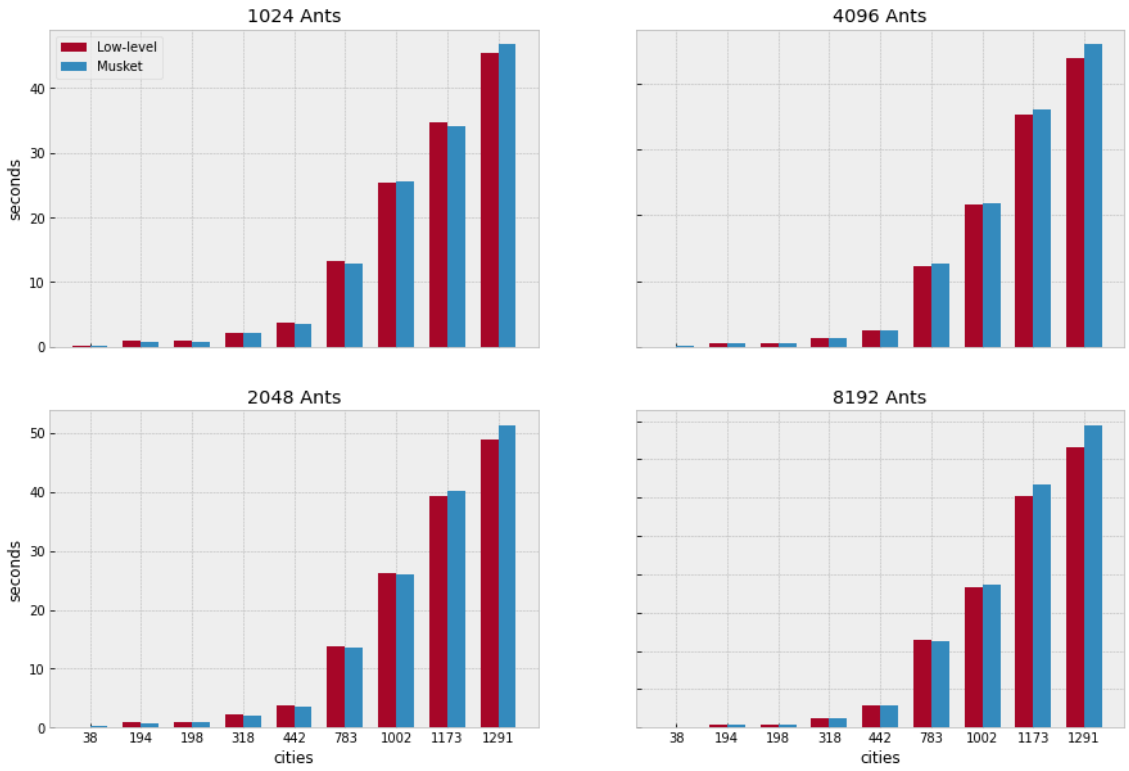


Figure 9.1: TSP execution times comparison

size. For example, when tackling the biggest problem, the low-level implementations are 0.4% faster with 1024 ants, 3.9% with 2048, 7.9% with 4096, and around 7.9% with 8192.

The execution times follow a similar pattern also for the biggest problem tackled (pr2392). Figure 9.2 illustrates this. This pattern appears in all test cases and is directly connected to how the programs are organized. In the low-level version, the operations are executed specifically for a certain task, against general-purpose operations present in the *Musket* program.

Table 9.2 shows the overall execution times for both parallel ACO implementations considering the different problems and setups.

Another factor that affects the execution times is the setup regarding the number of blocks and block size. As CUDA does not accept more than 1024 threads per block for most architectures and some kernels used a block size equal to the number of cities, some balancing was necessary. Using more blocks with fewer threads each enables CUDA to run the algorithm but it also adds some overhead, which explains the growth in the execution times when changing from 4096 to 8192 ants. Adaptations to solve this matter are easily done in the low-level program which generates a program with a better configuration of numbers of blocks and threads, which fits the problem, compared to high-

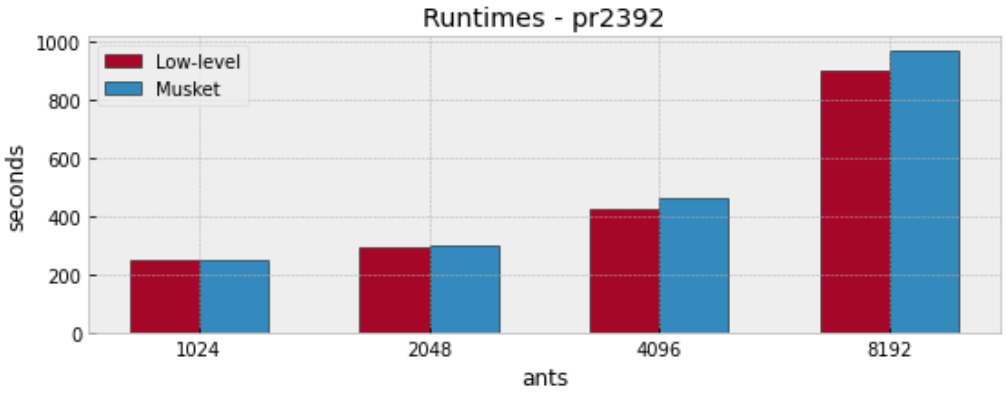


Figure 9.2: Execution times for pr2392

	GeForce RTX 2080 Ti							
	1024		2048		4096		8192	
Problem	LL	*Musket*	LL	*Musket*	LL	*Musket*	LL	*Musket*
dji38	0.103	0.186	0.106	0.189	0.107	0.192	0.158	0.199
cat194	0.890	0.782	0.896	0.797	1.022	0.938	1.742	1.789
d198	0.906	0.852	0.914	0.861	1.037	1.001	1.784	1.804
lin318	2.217	2.103	2.250	2.103	2.726	2.665	6.175	6.261
pcb442	3.711	3.464	3.783	3.474	5.022	4.961	14.318	14.224
rat783	13.365	12.870	13.766	13.621	24.651	25.191	57.208	56.407
pr1002	25.409	25.529	26.307	26.116	43.194	43.657	91.539	93.51
pcb1173	34.680	34.201	39.201	40.23	70.605	72.321	151.416	158.965
d1291	45.37	46.744	48.832	51.306	87.808	91.888	182.741	197.17
pr2392	251.412	252.648	292.913	304.26	428.534	462.256	899.670	969.704

Table 9.2: Execution times comparison: low-level vs. *Musket*

level approaches. Of course, the kernel instructions can be changed in order to optimize the execution time, but for comparison purposes, only the numbers of blocks and threads were changed. In order to investigate further the runtime differences between both implementations, the runtime of single kernels was isolated and investigated separately.

The most important and time-consuming step in ACO is the tour construction. Performed many times during the execution, it is affected by the colony size and also the graph size. Therefore, special attention was given to the time spent by each parallel implementation on creating routes. Figure 9.3 shows for the example of 1024 ants the proportional amount of time spent on each kernel by the *Musket* implementation and the low-level implementation. Obviously, even for the smallest map the route-construction kernel requires for both programs by far most of the runtime. Therefore, we investigated the calculations of the route.

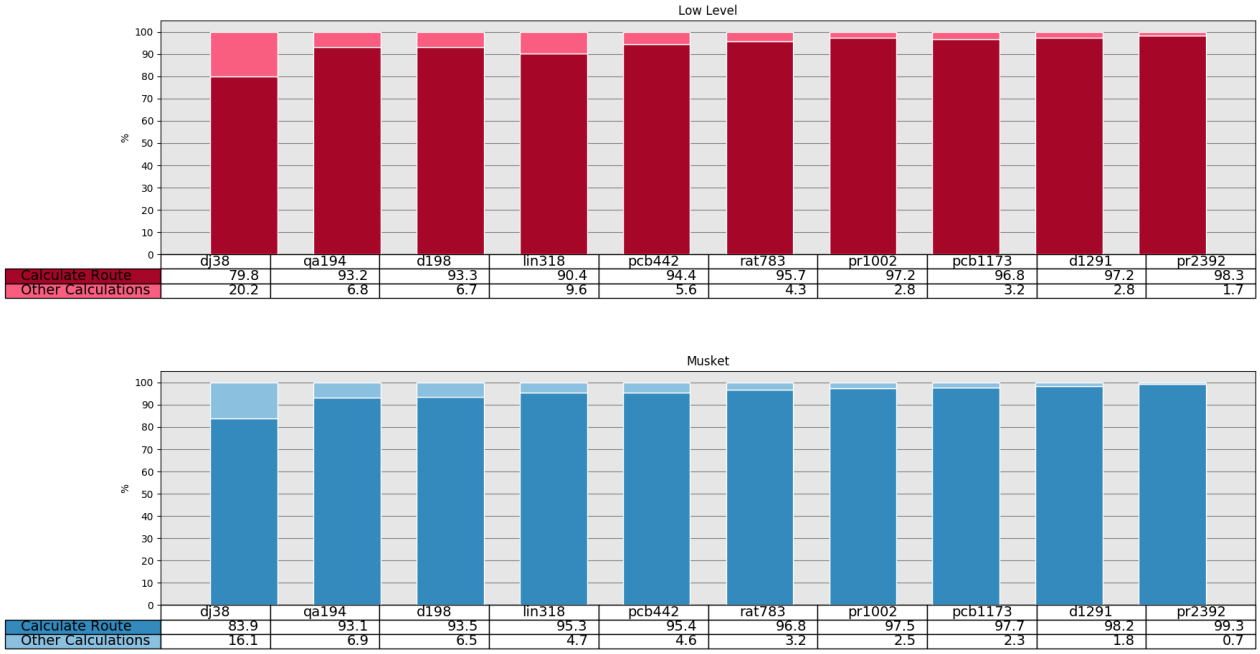


Figure 9.3: Proportional execution times of route calculation

In order to compare the two implementations regarding the tour construction step, we have investigated the average time spent in the tour construction per iteration as shown in Figure 9.4. The graphs show similar results to the total execution times mentioned previously and it is no wonder since the tour construction is the reason for a great part of the general execution times shown before.

The kernels responsible for executing the other steps of the algorithm have almost equal execution times for both implementations. Furthermore, they represent a small, almost irrelevant, part of the whole execution time. Therefore no deeper analysis becomes necessary.

9.6.2 BPP Experiments

The BPP instances used in this work were extracted from different sources from literature [8, 14]. They vary not only in the number of items to be packed but also in the degree of difficulty to solve. Details about each instance can be seen in Table 9.3.

In order to evaluate the performance of both implementations and compare it with the results of the TSP experiments, similar setups were used in the BPP experiments. The same numbers are used for the colony sizes. Furthermore, the BPP experiments were also executed in a second GPU, the NVIDIA Tesla V100, so that the programs could be

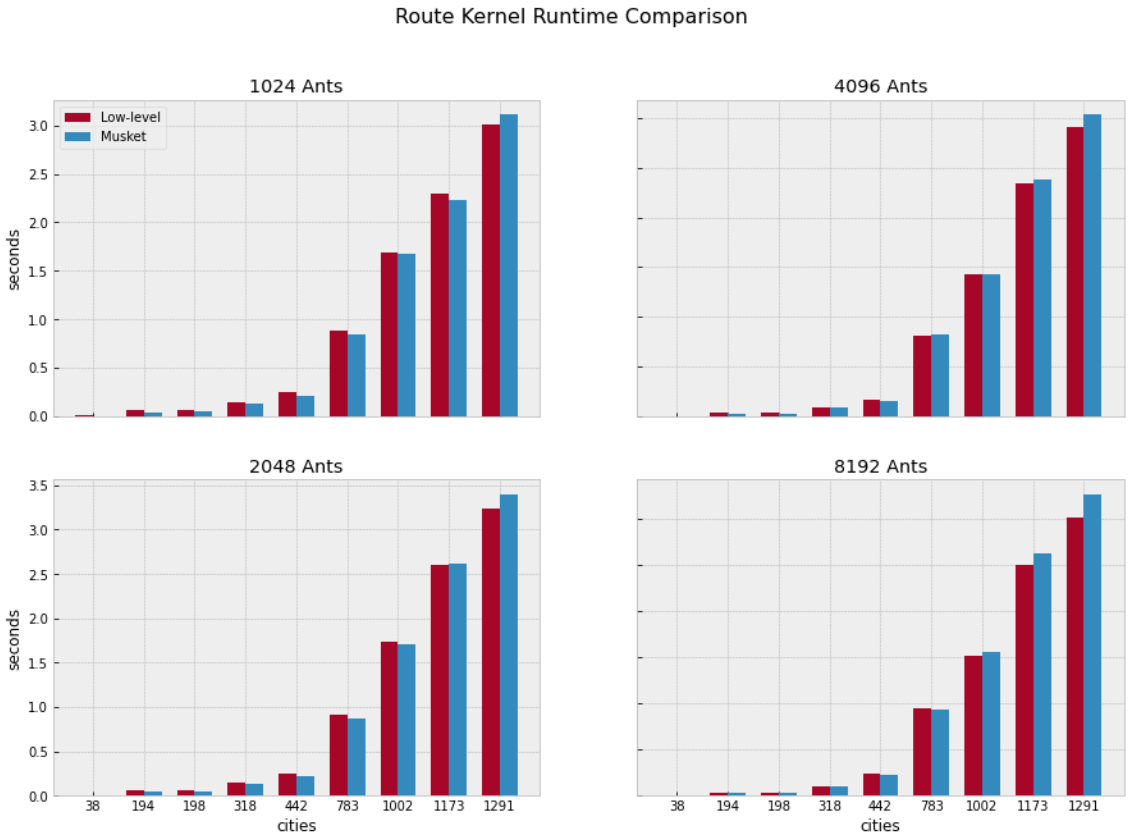


Figure 9.4: Execution times comparison for the tour construction kernel

Instance	0	1	2	3	4	5
# item types	50	166	358	522	712	915
total # items	60	201	402	600	801	1002
bin capacity	1000	2456	7552	16256	31616	65088

Table 9.3: BPP

evaluated using devices with different configurations. The average execution times for each of the BPP instances are displayed in Figure 9.5.

The results show that the low-level implementation has slightly shorter execution times for most of the test cases and for both GPUs. Also, both implementations presented a reduction in the execution times to a similar degree when using the Tesla V100. Similarly to the TSP results, the tendency of having slightly shorter execution times when running the low-level program is more pronounced when observing the results from the bigger instances with more items to be packed. Table 9.4 includes the execution times and also the percentage comparison from the low-level to the musket program, both using the GeForce RTX 2080 Ti.

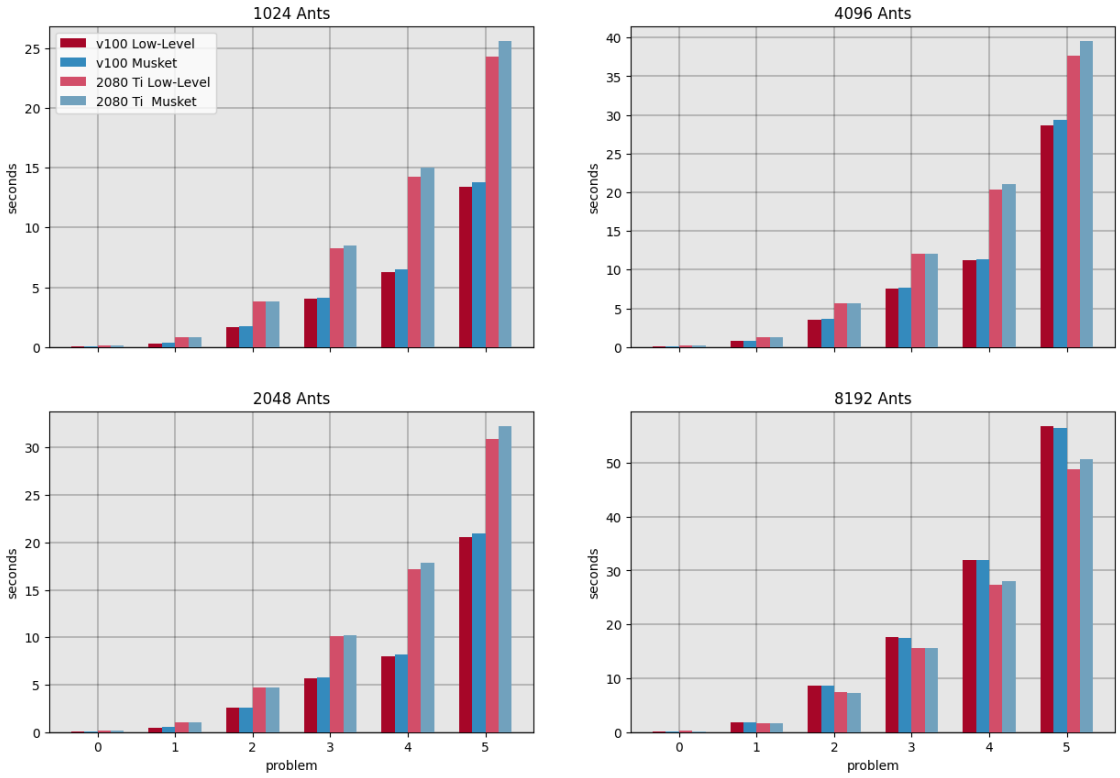


Figure 9.5: BPP execution times

	GeForce RTX 2080 Ti											
	1024			2048			4096			8192		
P.	LL	*Musket*	%	LL	*Musket*	%	LL	*Musket*	%	LL	*Musket*	%
0	0.14	0.14	0.99	0.15	0.16	1.02	0.17	0.18	1.08	0.21	0.21	1.00
1	0.81	0.82	1.02	1.01	1.05	1.05	1.21	1.29	1.06	1.59	1.69	1.07
2	3.82	3.87	1.01	4.70	4.72	1.00	5.65	5.68	1.01	7.40	7.38	1.00
3	8.26	8.49	1.03	10.11	10.25	1.01	12.06	12.15	1.01	15.64	15.75	1.01
4	14.21	15.07	1.06	17.23	17.93	1.04	20.32	21.14	1.04	27.35	28.16	1.03
5	24.24	25.60	1.06	30.94	32.28	1.04	37.70	39.60	1.05	48.85	50.82	1.04

Table 9.4: BPP execution times: proportional comparison

It is also interesting to observe that the execution times for both implementations scale differently as the colony gets bigger in the BPP experiments when compared to the TSP experiments. The speedup rates for the BPP experiments remain quite stable, independently of the colony sizes. This might be explained by the fact that the data is structured differently for the bin packing problem when compared to the TSP. In the BPP, most of the data are integer values, which occupy less space and favor internal calculations. Furthermore, the data structures that store the information regarding the items to be packed are also smaller due to the repetition of items, impacting directly the

space required by each thread and reducing the number of probability calculations during the packing phase. Having less demanding threads to execute makes it possible for the GPU to run more threads simultaneously, meaning that doubling the colony size will not necessarily double the execution time.

In order to investigate further, we measured also the time spent during the packing phase of the algorithm. Figure 9.6 shows the values for Problem 3.

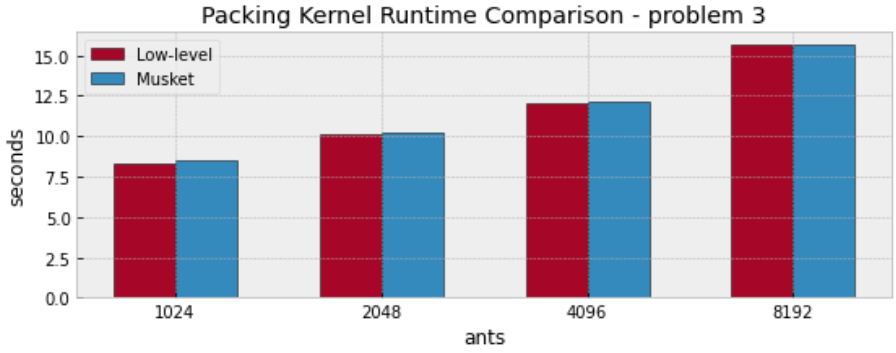


Figure 9.6: Packing kernel execution times - problem 3

9.6.3 MKP Experiments

The MKP instances used in this work were extracted from the OR-Library, first described in [15]. The library offers a couple of instances of the MKP with different numbers of objects and different numbers of constraints. Details about each instance can be seen in Table 9.5.

Instance	1	2	3	4	5	6	7
Objects	6	10	15	20	28	39	50
Constraints	10	10	10	10	10	5	5
Optimal	3800	8706.1	4015	6120	12400	10618	16537

Table 9.5: MKP problems

For these experiments, the same ACO setup was used as in the experiments previously mentioned. Regarding the hardware used, a third GPU was used in the benchmarks, namely the Quadro RTX 6000. By doing so, the evaluation of both implementations was extended and the performance could be compared also for different scenarios since the GPUs have different specs. A comparison of the execution times for the low-level and musket implementations is displayed in Figure 9.7.

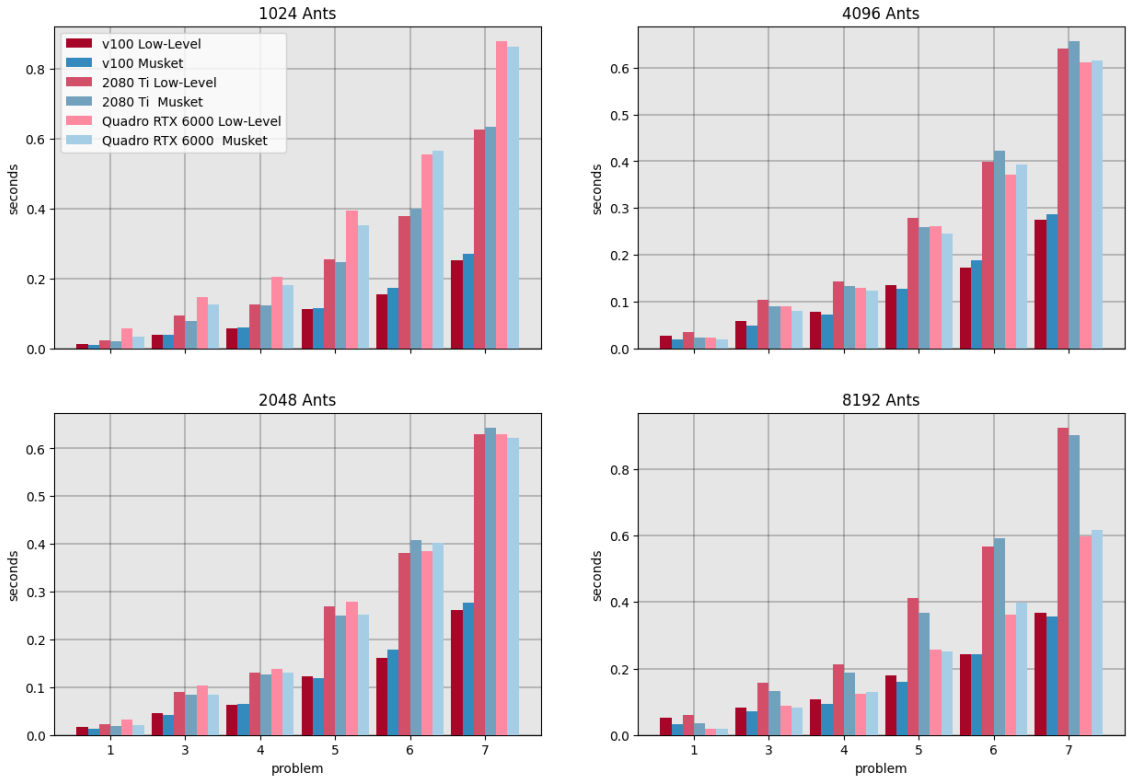


Figure 9.7: MKP execution times - GeForce RTX 2080 Ti, Tesla V100 and Quadro RTX 6000

The execution time graph shows very similar performances for both implementations when using the same GPU. The differences are in the magnitude of hundredths of a second and did not follow any pattern. The size of the MKP instances and the overall short execution times show that the problems did not pose a greater challenge to the programs which makes it difficult to identify where the differences come from. In addition to that, the solution sizes differ, which adds an uncertainty factor to how long a step to build a solution should take.

Performance differences between implementations using different GPUs can also be seen in Figure 9.7. It is interesting to observe that the runtimes from the Musket implementation scale are in the same proportion as those from the low-level implementation for all colony sizes.

Figure 9.8 puts the execution times from the experiments using the Quadro RTX 6000 GPU into another perspective. The graph shows how the execution times are similar for both implementations and how they remain unchanged even when the colony size is doubled. For the same problem, the execution times grow only when the colony size is increased to 8192 ants. This behavior shows how similar the workload generated by

the implementation using Musket is when compared to the low-level implementation. For both, the workload generated was not enough to fully occupy the GPU even when the colony size was doubled. It took 8192 ants to generate a workload that would fully occupy the Quadro RTX 6000 GPU and its 4608 cores.

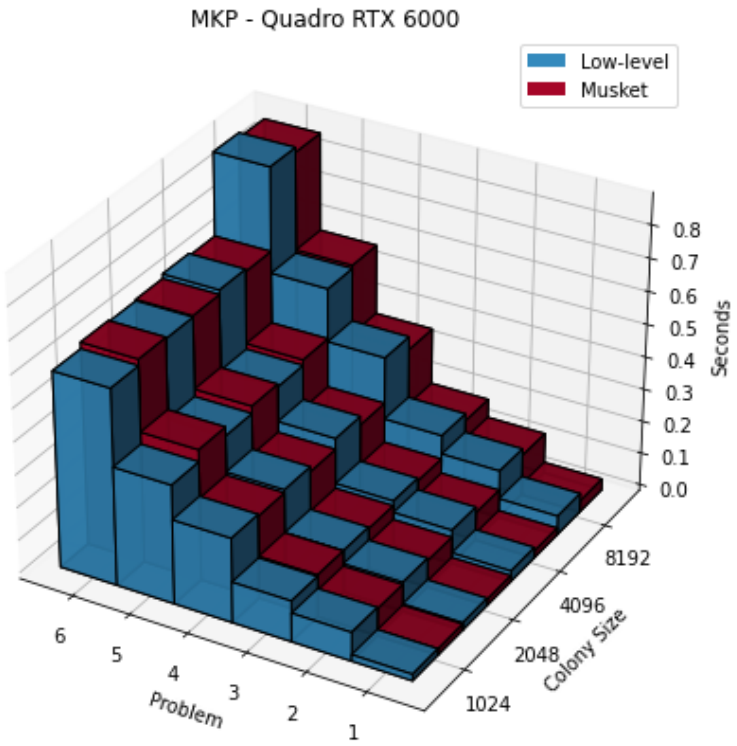


Figure 9.8: MKP execution times - Quadro RTX 6000

As an overall result, plenty of similarities between the execution times of both implementations investigated in this work can be observed for all three problems. They show how the use of *Musket* can simplify the development of parallel programs, as the use of general purpose skeletons provided out of the box suffices to develop a parallel version of ACO in fewer lines of code and on a much lower complexity level when compared to the low-level CUDA implementation without impairing the performance.

9.7 Conclusion

The use of a high-level parallelization approach can be of great help for programmers aiming to run swarm intelligence algorithms on high-performance hardware, such as graphical processing units. In this work we have evaluated *Musket* as an approach for

the parallelization of the ACO algorithm in order to identify the pros and cons of using such a tool regarding the development aspect and also the performance aspect when compared to a low-level implementation.

Considering the development aspect, in its actual state, the skeletons embedded in *Musket* provide enough features for the development of parallel ACO programs. Furthermore, the experiments have shown that *Musket* offered some advantages in terms of simplicity, requiring fewer skills to develop a high-performance parallel version of ACO. Not only fewer lines of code were necessary, but it is also much simpler to program without having the concerns that regard the parallel aspects of programming a CUDA-based version of the code, such as data initialization, data transfers, and the allocation of blocks and threads.

In terms of runtime, the ACO version implemented using *Musket* achieved reasonably good execution times compared to the low-level CUDA-based implementation for both problems investigated here. As algorithm enhancements and handcrafted adaptations to a certain problem instance were left aside, both implementations were evaluated in equal conditions. By doing so, we were able to observe how the *Musket* implementation reacts in scenarios where the colony size was increased and more resources were needed. In these experiments, a bit more overhead was generated but nothing that would compromise *Musket*'s overall performance. In the end, there is a positive balance, as the results showed that we were able to simplify the implementation phase without compromising the runtime of the experiments.

The ACO version used in this work was idealized to be a simple implementation for a single GPU environment. Many optimizations could be introduced in order to enhance the performance of the algorithm e.g. the usage of shared memory. Also, if the goal is to run in a new environment such as multiple GPUs or multiple computational nodes with multiple GPUs, more complex changes are necessary, which can be tricky even for experienced programmers. In this aspect, *Musket* has the advantage that the same program can be used to generate code for different architectures once there is a code generator for it.

Future works include the evaluation of different hardware, such as multiple GPUs on one computational node and also a cluster environment with many nodes and many GPUs per node. Furthermore, we want to further investigate the possibility of enhancing *Musket* to provide metaheuristic-specific skeletons in order to make better use of the hardware and reduce even more the execution times for such problems.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 'Fastflow: high-level and efficient streaming on multi-core'. In: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017).

[2] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. 'Flexible skeletal programming with eSkel'. In: *European Conference on Parallel Processing*. Springer. 2005, pp. 761–770.

[3] José M. Cecilia, José M. García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. 'Enhancing data parallelism for ant colony optimization on GPUs'. In: *Journal of Parallel and Distributed Computing* 73.1 (2013), pp. 42–51.

[4] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[5] Breno Augusto De Melo Menezes, Luis Filipe De Araujo Pessoa, Herbert Kuchen, and Fernando Buarque De Lima Neto. 'Parallelization strategies for GPU- ased ant colony optimization applied to TSP'. In: *Advances in Parallel Computing* 36 (2020), pp. 321–330.

[6] Breno Augusto De Melo Menezes and Nina Herrmann. *Ant Colony Optimization Project*. `https://github.com/brenoamm/ant-colony-optimization-project`. Last Change: 24.03.2021. 2021. `url`: `https://github.com/wwu-pi/HLPP2020_ACO_Programs`.

[7] Breno Augusto De Melo Menezes and Nina Herrmann. *Musket Repository*. `https://github.com/wwu-pi/musket_dsl`. Last Change: 04.05.2020. 2020. `url`: `https://github.com/wwu-pi/musket_dsl`.

[8] M. Delorme, M. Iori, and S. Martello. 'Bin packing and cutting stock problems: Mathematical models and exact algorithms'. In: *European Journal of Operational Research* 255 (2016), pp. 1–20.

[9] Marco Dorigo. 'Optimization, Learning and Natural Algorithms[in Italian]'. PhD thesis. Milan: Dipartimentodi Elettronica, Politecnico di Milano, 1992.

[10] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. 'Ant colony optimization'. In: *IEEE computational intelligence magazine* 1.4 (2006), pp. 28–39.

[11] Marco Dorigo and Gianni Di Caro. *Ant colony optimization: a new meta-heuristic*. 1999.

[12] Steffen Ernsting and Herbert Kuchen. 'Algorithmic skeletons for multi-core, multi-GPU systems and clusters'. In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138.

[13] Steffen Ernsting and Herbert Kuchen. 'Data parallel algorithmic skeletons with accelerator support'. In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299.

[14] E. Falkenauer. 'A hybrid grouping genetic algorithm for bin packing'. In: *J. of Heuristics* 2 (1996), pp. 5–30.

[15] J.E. Beasley. 'OR-Library: distributing test problems by electronic mail'. In: *Journal of the Operational Research Society*. 1990, pp. 1069–1072. `url`: `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

[16] Nikolaos Ath Kallioras, Konstantinos Kepaptsoglou, and Nikos D. Lagaros. 'Transit stop inspection and maintenance scheduling: A GPU accelerated metaheuristics approach'. In: *Transportation Research Part C: Emerging Technologies* 55 (2015), pp. 246–260.

[17] John Levine and Frederick Ducatelle. 'Ant colony optimization and local search for bin packing and cutting stock problems'. In: *Journal of the Operational Research Society* 55.7 (2004), pp. 705–716.

[18] Breno A.M. Menezes, Herbert Kuchen, Hugo A. Amorim Neto, and Fernando B. De Lima Neto. 'Parallelization Strategies for GPU-Based Ant Colony Optimization Solving the Traveling Salesman Problem'. In: *2019 IEEE Congress on Evolutionary Computation, CEC 2019 - Proceedings* (2019), pp. 3094–3101.

[19] Tomas Öhberg, August Ernstsson, and Christoph Kessler. 'Hybrid CPU–GPU execution support in the skeleton programming framework SkePU'. In: *The Journal of Supercomputing* 76.7 (2020), pp. 5038–5056.

[20] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. 'Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons'. In: *Proceedings of the ACM Symposium on Applied Computing* Part F147772 (2019), pp. 1534–1543.

[21] Fabrizio Riguzzi. *A Survey of Software Metrics*. Tech. rep. 1996.

[22] Soh-Yee Lee and Yoon-Teck Bau. ‘An ant colony optimization approach for solving the Multidimensional Knapsack Problem’. In: *2012 International Conference on Computer and Information Science (ICCIS)*. IEEE, June 2012, pp. 441–446. url: http://ieeexplore.ieee.org/document/6297286/.

[23] El-Ghazali Talbi. *Metaheuristics: from design to implementation*. Hoboken, NJ, USA: John Wiley & Sons, 2009.

[24] The Eclipse Foundation. *Xtext Documentation*. https://eclipse.org/Xtext/documentation/. 2020.

[25] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. ‘Accelerating ant colony optimisation for the travelling salesman problem on the GPU’. In: *International Journal of Parallel, Emergent and Distributed Systems* 29.4 (2014), pp. 401–420.

[26] Heidelberg University. *Discrete and Combinatorial Optimization*. https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/XML-TSPLIB/instances/. Accessed: 14.03.2018.

[27] University of Waterloo. *National Traveling Salesman Problems*. http://www.math.uwaterloo.ca/tsp/world/countries.html. Accessed: 14.03.2018.

[28] Fabian Wrede, Christoph Rieger, and Herbert Kuchen. ‘Generation of high-performance code based on a domain-specific language for algorithmic skeletons’. In: *The Journal of Supercomputing* 0123456789 (2019).

10

Stencil Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments

Nina Herrmann* · **Breno A. de Melo Menezes *** · **Herbert Kuchen***

Citation Nina Herrmann, Breno Menezes, and Herbert Kuchen. 'Stencil calculations with algorithmic skeletons for heterogeneous computing environments'. In: *International Journal of Parallel Programming* 50.5-6 (2022), pp. 433–453.

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.

*University of Münster, Germany

Keywords Parallel Programming· Skeleton Programming· Heterogeneous Computing Environments· High-Level Frameworks· Stencil Operations.

10.1 Introduction

High-performance computers consist frequently of multiple computing nodes, each consisting of multicore Central Processing Units (CPUs) and potentially several Graphics Processing Units (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 [11, 12, 3], Domain Specific Languages (DSLs) [20], and general frameworks [16, 2]. 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 10.2 summarizes related work, Section 10.3 introduces the Münster Skeleton Library (Muesli), pointing out its concepts and benefits. In Section 10.4 the conceptual implementation of the MapStencil skeleton is discussed. Benchmark applications, as well as experimental results, are presented in Section 10.5. In Section 10.6, we discuss possible extensions. Finally, we conclude in Section 10.7 and briefly point out future work.

10.2 Related Work

Cheıkh 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 Aldınucci 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 [21, 14] 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.

10.3 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. 3, 9, 2]. 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 High Performance Computing (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 10.1 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 Sum sum;
5 auto product = [] (int i, int j) {return i*j;};
6 DA<int> a(3,2);                    // delivers: {2,2,2}
7 DA<int> b = a.mapIndex(sum);       // delivers: {2,3,4}
8 a.zipInPlace(b,product);           // delivers: {4,6,8}
9 int scalarproduct = a.fold(sum);   // delivers: 18
```

Listing 10.1: Scalar product in Muesli.

10.4 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 Figure 10.1[32]. 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.

10.4.1 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

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

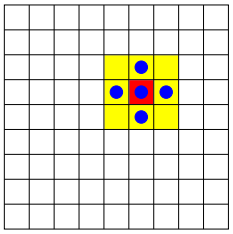


Figure 10.1: MapStencil: the value of an element as the one depicted in red depends on those values in the surrounding square (here of size 3×3 and depicted in yellow) belonging to the considered stencil (here depicted in blue).

possible approach is to distribute the data for the MapStencil skeleton as displayed in Figure 10.2. 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 $2n < 4(k + 1)$. One possible approach targeting a heterogeneous environment with CPUs and GPUs is displayed in Figure 10.3. 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.

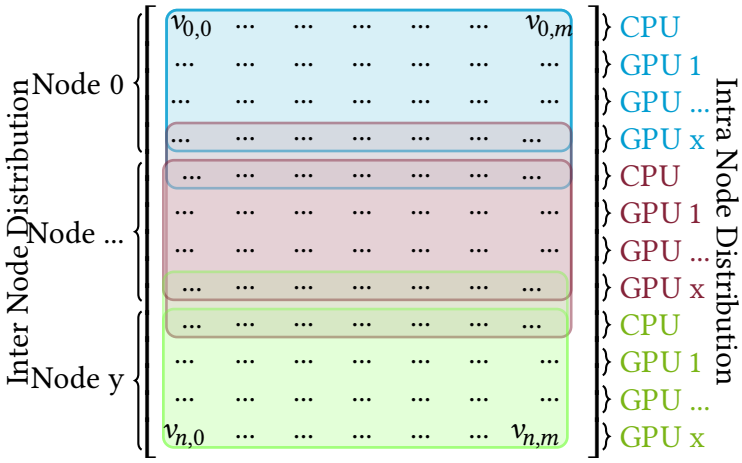


Figure 10.2: Data distribution

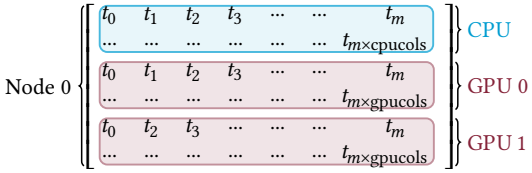


Figure 10.3: Intra-node distribution using a CPU.

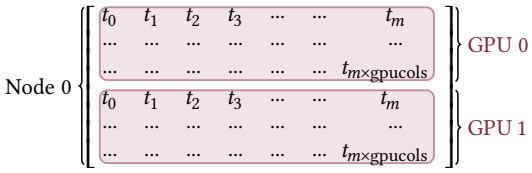


Figure 10.4: Intra-node distribution without a CPU.

Another approach is displayed in Figure 10.4, 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 GPUs 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 block[33] 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 100x lower than that of uncached global GPU memory[34]. 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

[33] Not to be confused with a block of a block-wise distribution.
[34] https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/ accessed 27.04.2021

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 numerically solving the heat equation [8], a cross-shaped stencil as in Figure 10.1 is required to calculate the current element. This means that each element is read four times. In Conway's game of life[35] 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 Figure 10.5. 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 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

[35] http://math-fail.com/2010/07/conways-game-of-life-in-html-5.html

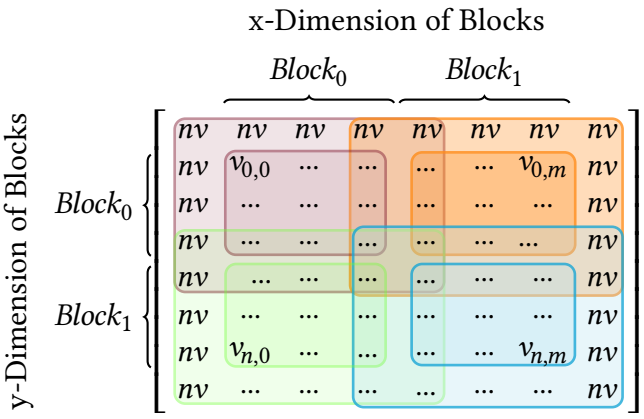


Figure 10.5: Shared memory GPU distribution.

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 10.6 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,

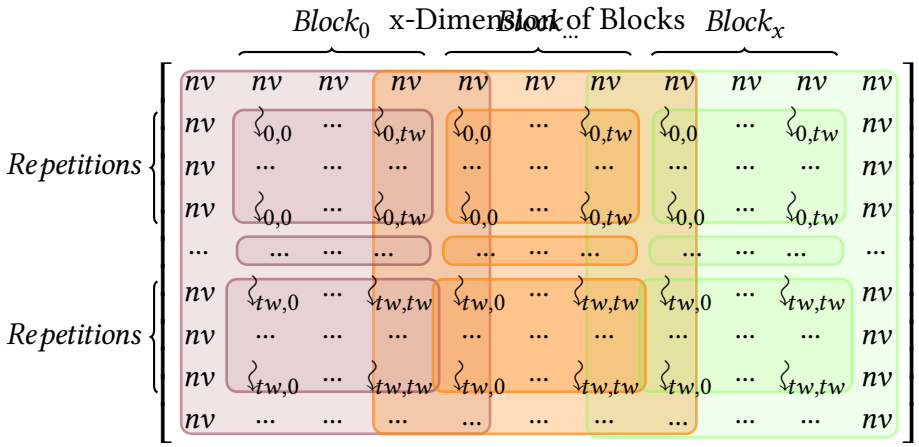


Figure 10.6: Improved shared memory GPU distribution.

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 data structure.

10.4.2 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.[36]

```
1  JacobiBorder borderFunctor(n, m, 75);
2  JacobiSweepFunctor 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, jacobiFunctor, borderFunctor);
10                 differences = board.zip(swpBoard, differenceFunctor);
11                 globalDiff = differences.fold(maxFunctor, true);
```

[36] 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.

```
12          } else {
13                  if (numIter % 2 == 0) {
14                          board.mapStencilMM(swpBoard, jacobiFunctor, borderFunctor);
15                  } else {
16                          swpBoard.mapStencilMM(board, jacobiFunctor, borderFunctor);
17                  }
18          }
19          numIter++;
20  }
```

Listing 10.2: 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 10.2. 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 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_(default) {}
5     MSL_USERFUNC
6     float operator()(int x, int y) const {
```

```
7         if (y < 0 || y > (glob_cols_ - 1)) {return 100;}
8         if (x < 0) {return 100;}
9         if (x > (glob_rows_ - 1)) {return 0;}
10        return default_;
11    }
12    private:
13    int glob_rows_;
14    int glob_cols_;
15    int default_ = 75;
16 };
```

Listing 10.3: Example of a functor for processing the border.

```
1 class JacobiSweepFunctor: public DMMapStencilFunctor<float, float, JacobiNeutralValueFunctor> {
2     public:
3     JacobiSweepFunctor() : DMMapStencilFunctor(){}
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 10.4: Example of a stencil functor.

The functors for computing the border and the stencil can be seen in Listing 10.3 and Listing 10.4, 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 a ttributes (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.

10.5 Experimental Results

We have tested the MapStencil skeleton with the three example applications mentioned in Section 10.4, namely the Jacobi method, the Gaussian blur, and the game of life. We have measured the runtimes 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[37]. 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 Section 10.4 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 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.

10.5.1 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 10.1. In order to keep the dataset smaller, the runtimes on two GPUs have been excluded in Table 10.1. 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 runtime 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

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

matrix	512 × 512		1000 × 1000		5000 × 5000	
CPU %	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

Table 10.1: Runtimes (in seconds) for different fractions calculated by the CPU for the Jacobi Solver.

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.

10.5.2 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 10000 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 runtimes of the parallel program on 1-4 nodes with 1-4 GPUs each are displayed in Figure 10.7. Some of the numbers are additionally listed in Table 10.2. 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 runtime 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 Figure 10.7, the program running on a

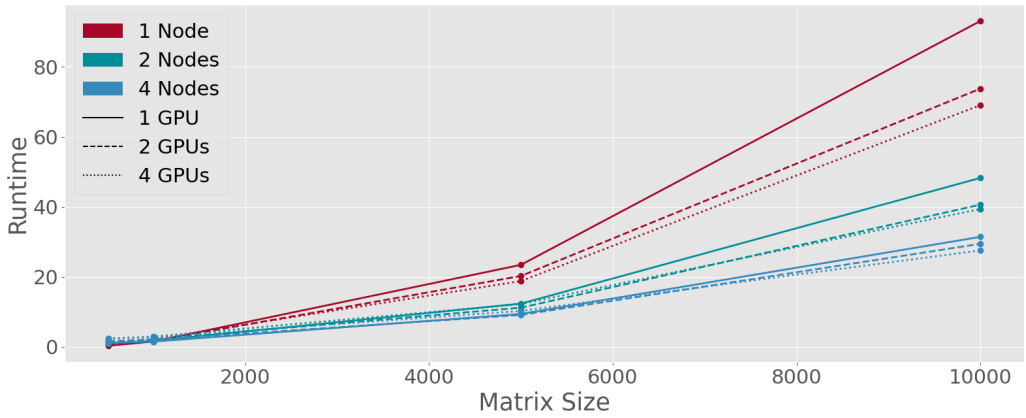


Figure 10.7: Runtimes (in seconds) on multiple nodes and GPUs per node using GPU global memory only for the Jacobi Solver.

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 runtime is divided by 3. In contrast adding four GPUs reduces the runtime 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 10.3, 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 10000×10000 matrix of the Jacobi Solver.

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

matrix size	seq. C++	1 Node					
		1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
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
10000^2	3544.270	93.156	**38.047**	73.815	48.015	69.092	**51.298**
matrix size	seq. C++	2 Nodes					
		1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
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
10000^2	3544.270	48.285	73.404	40.636	87.221	39.388	89.984
matrix size	seq. C++	4 Nodes					
		1 GPU	Speedup	2 GPUs	Speedup	4 GPU	Speedup
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
10000^2	3544.270	31.435	**112.749**	29.443	120.379	27.551	**128.642**

Table 10.2: Runtimes (in seconds) and speedups on multiple nodes and GPUs per node only using GPU global memory for the Jacobi Solver. Numbers in bold face are mentioned in the text.

matrix size	seq. C++	1 Node 1 GPU	Speedup	2 GPUs	Speedup	4 GPUs	Speedup
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**
matrix size	seq. C++	2 Nodes 1 GPU	Speedup	2 GPUs	Speedup	4 GPUs	Speedup
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**
matrix size	seq. C++	4 Nodes 1 GPU	Speedup	2 GPUs	Speedup	4 GPUs	Speedup
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**

Table 10.3: Runtimes (in seconds) and speedups on multiple nodes and GPUs per node only using GPU global memory for the Game of life.

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 runtime to four nodes. In that 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.

10.5.3 Experiments on Multiple Nodes and GPUs Using Shared Memory

Another aspect to inspect is the runtime when using shared memory. Thus, we varied how many threads are started, which affects how many elements are loaded into the shared memory. In Figure 10.8 the runtimes for starting 64 threads (8×8 tile) and 256 (16×16 tile) threads are shown. The runtime slightly improves when loading more elements in contrast to fewer elements. However, it is still slower than the global memory

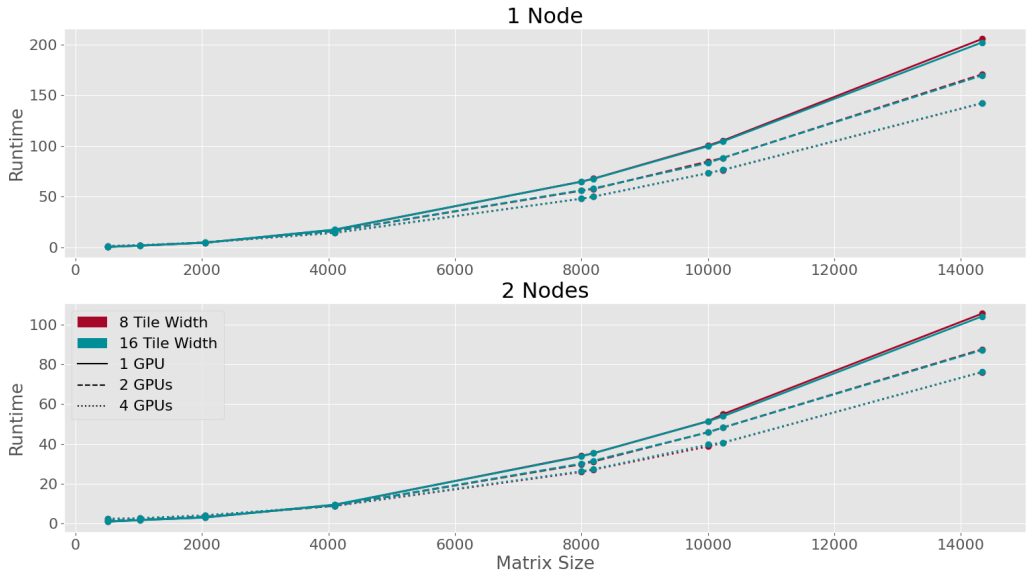


Figure 10.8: Runtimes (in seconds) on multiple nodes and GPUs per node using GPU shared memory for the Jacobi Solver.

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 runtimes for one GPU are 93.156 s without shared memory and 100.205 s with shared memory, while for four GPUs the runtime 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 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 runtime.

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 10.5. 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 runtime increases from 44.86 s 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.

matrix size	1 Node					
	1 GPU			4 GPUs		
	GM	SM 8^2	SM 16^2	GM	SM 8^2	SM 16^2size
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
10000^2	**93.156**	**100.205**	**99.569**	69.092	73.108	73.146
matrix size	2 Nodes					
	1 GPU			4 GPUs		
	GM	SM 8^2	SM 16^2	GM	SM 8^2	SM 16^2
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
10000^2	48.285	51.402	51.311	**39.388**	**38.673**	**39.675**

Table 10.4: Runtimes (in seconds) on multiple nodes and GPUs per node using GPU global memory (GM) vs. shared memory (SM) for the Jacobi Solver.

matrix size	1 Node					
	1 GPU			4 GPUs		
	GM	SM 8^2	SM 16^2	GM	SM 8^2	SM 16^2
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
matrix size	2 Nodes					
	1 GPU			4 GPUs		
	GM	SM 8^2	SM 16^2	GM	SM 8^2	SM 16^2
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

Table 10.5: Runtimes (in seconds) on multiple nodes and GPUs per node using GPU global memory (GM) vs. shared memory (SM) for the game of life.

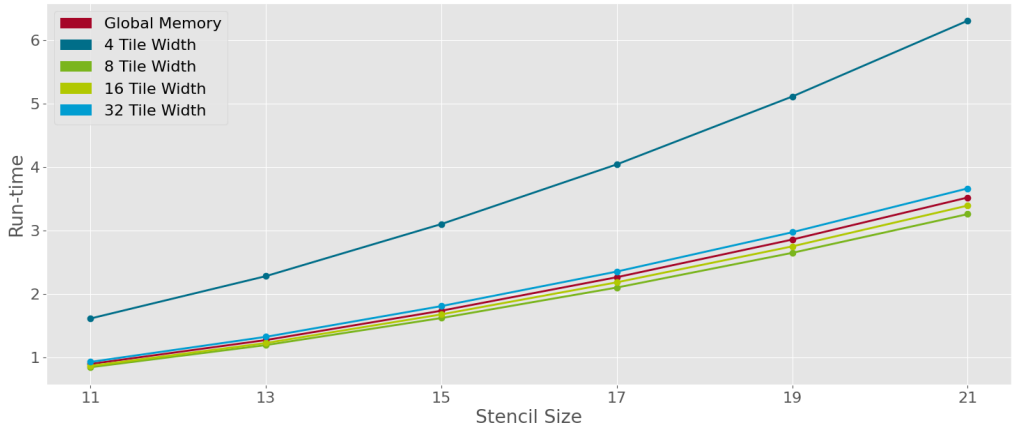


Figure 10.9: Runtimes (in seconds) for the Gaussian blur with different tile sizes on one GPUs using GPU global memory or GPU shared memory.

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 runtime 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 runtimes are meaningful. The runtimes are listed in Table 10.6 and shown in Figure 10.9. The listed stencil sizes go from 11×11 to 21×21 stencils. This means for calculating one pixel 121 to 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 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 runtime. Starting more threads slows down the runtime, which could be due to bank conflicts. To underline our finding it is compared to a hand-written parallel program. Table 10.7 lists the speedups for the different tile sizes. Those runtimes 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 runtime, the optimal tile size was chosen, which was in all displayed cases four elements per thread. The results can be seen in

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

Table 10.6: Runtimes (in seconds) and speedups on a single node using GPU global memory and shared memory for the Gaussian blur (muesli).

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

Table 10.7: 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	Muesli					
size	GM	SM 4^2	SM 8^2	SM 16^2	SM 32^2	Speed-up
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
stencil	LowLevel					
size	GM	SM 4^2	SM 8^2	SM 16^2	SM 32^2	Speed-up
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

Table 10.8: Runtimes (in seconds) using GPU global memory (GM) vs. shared memory (SM) for the Gaussian blur letting each thread calculate the optimal number of elements.

Table 10.8. Letting one thread calculate multiple elements slightly improved the low-level program's runtime. 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.

10.6 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, block-wise distribution becomes more efficient. For example, a three-dimensional data structure of size 64×64×64, 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 handled 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.

10.7 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 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.

References

[1] Marco Aldinucci, Marco Danelutto, Maurizio Drocco, Peter Kilpatrick, Guilherme Peretti Pezzi, and Massimo Torquati. 'The loop-of-stencil-reduce paradigm'. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 3. IEEE. 2015, pp. 172–177.

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 'Fastflow: high-level and efficient streaming on multi-core'. In: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017).

[3] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. 'Flexible skeletal programming with eSkel'. In: *European Conference on Parallel Processing*. Springer. 2005, pp. 761–770.

[4] Taieb Lamine Ben Cheikh, Alexandra Aguiar, Sofiene Tahar, and Gabriela Nicolescu. 'Tuning framework for stencil computation in heterogeneous parallel platforms'. In: *The Journal of Supercomputing* 72.2 (2016), pp. 468–502.

[5] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 'Automatic code generation and tuning for stencil kernels on modern shared memory architectures'. In: *Computer Science-Research and Development* 26.3 (2011), pp. 205–210.

[6] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[7] NVIDIA Corporation. *CUDA*. `https://developer.nvidia.com/cuda-zone`. 2021. (Visited on 10/05/2021).

[8] John Crank and Phyllis Nicolson. '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. 1. Cambridge University Press. 1947, pp. 50–67.

[9] Kento Emoto, Sebastian Fischer, and Zhenjiang Hu. 'Generate, Test, and Aggregate'. In: *Programming Languages and Systems*. Ed. by Helmut Seidl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 254–273.

[10] Johan Enmyren and Christoph W Kessler. '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*. 2010, pp. 5–14.

[11] Steffen Ernsting and Herbert Kuchen. 'Algorithmic skeletons for multi-core, multi-GPU systems and clusters'. In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138.

[12] Steffen Ernsting and Herbert Kuchen. 'Data parallel algorithmic skeletons with accelerator support'. In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299.

[13] MPI Forum. *MPI Standard*. `https://www.mpi-forum.org/docs/`. 2021. (Visited on 10/05/2021).

[14] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 'High performance stencil code generation with lift'. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2018, pp. 100–112.

[15] Xinxin Mei and Xiaowen Chu. 'Dissecting GPU Memory Hierarchy Through Microbenchmarking'. In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2017), pp. 72–86.

[16] Tomas Öhberg, August Ernstsson, and Christoph Kessler. 'Hybrid CPU–GPU execution support in the skeleton programming framework SkePU'. In: *The Journal of Supercomputing* 76.7 (2020), pp. 5038–5056.

[17] OpenMP. *OpenMP The OpenMP API specification for parallel programming*. `https://www.openmp.org/`. 2021. (Visited on 10/05/2021).

[18] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. 'The pochoir stencil compiler'. In: *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. 2011, pp. 117–128.

[19] Ben Van Werkhoven, Jason Maassen, and Frank J Seinstra. 'Optimizing convolution operations in cuda with adaptive tiling'. In: *2nd Workshop on Applications for Multi and Many Core Processors (A4MMC 2011)* (2011).

[20] Fabian Wrede, Christoph Rieger, and Herbert Kuchen. 'Generation of high-performance code based on a domain-specific language for algorithmic skeletons'. In: *The Journal of Supercomputing* 76.7 (2020), pp. 5098–5116.

[21] Yongpeng Zhang and Frank Mueller. 'Auto-generation and auto-tuning of 3D stencil codes on GPU clusters'. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 2012, pp. 155–164.

11

Distributed Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments

Nina Herrmann* · **Herbert Kuchen***

Citation Nina Herrmann and Herbert Kuchen. 'Distributed Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments'. In: *International Journal of Parallel Programming* 51.2-3 (2023), pp. 172–185.

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

Keywords Parallel Programming · Skeleton Programming · Heterogeneous Computing Environments · High-Level Frameworks · Usability.

*University of Münster, Germany

11.1 Introduction

The field of High Performance Computing (HPC) is growing. Typical HPC hardware offers multiple computing nodes, Central Processing Units (CPUs) and Graphics Processing Units (GPUs) to speed up computations. Programmers have to deal with multiple low-level frameworks to exploit those levels of hardware where expertise for the single frameworks and the combination of those frameworks is required. Examples for those frameworks are Message Passing Interface (MPI) [9], OpenMP [12], and CUDA [4].

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

Cole introduced algorithmic skeletons as one of the major high-level approaches to abstract from low-level details [3]. Algorithmic skeletons enclose reoccurring parallel and distributed computing patterns, such as Map and Reduce. This concept is wide-spread and beside others implemented as libraries [6, 7, 2], Domain Specific Languages (DSLs) [16], and general frameworks [8, 1]. These approaches rarely support the combination of all levels of parallel hardware simultaneously, namely multiple nodes with multiple CPUs (possibly with vectorization) and accelerators.

In the present paper, we will first discuss related high-level frameworks contributing to the parallelization on multiple hard-ware levels in Section 11.2. Section 11.3 describes the design of our chosen high-level approach, the Münster Skeleton Library (Muesli), while Section 11.4 presents the implementation of some added features, in particular distributed cubes. The runtimes of exemplary programs are shown and discussed in Section 11.5. Finally in Section 11.6, we conclude and point out future work.

11.2 Related Work

Closest to our work is the skeleton framework SkePU3. In combination with StarPU, it works on heterogeneous clusters. However, it does not (yet) support the combined use of all possible levels of parallelism. Either the program is executed on one node with multiple cores and GPUs [11] or the programs are executed on multiple nodes with single backends

(either GPU (OpenCL) or CPU (OpenMP)) [8]. SkePU supports one, two, three, and four-dimensional data structures. Other skeleton frameworks which are in continuous development also do not consider all three layers of parallelization (e.g. FastFlow [1], SkelCL [15], Musket [13]). Hybrid execution of programs on CPUs and accelerators has been the topic in multiple approaches such as SkePU [11], Marrow [14] and Qilin [10]. SkePU and Marrow distribute the load statically between the CPU threads and the GPUs, while Qilin dynamically distributes the working packages. Findings regarding the optimal partitioning of work and data are often hardware and problem-dependent and rarely comparable. Noteworthy, skeletal programming is also used for commercial products such as Intel TBB for multicore CPU parallelism.

In the present paper, we will discuss the distribution of data and computations on multiple nodes, CPUs and GPUs. Our goal is to provide a starting point to automatically distribute workload between different computational units, relieving the programmer from estimating suitable partition ratios. To the best of our knowledge, other approaches either distribute the workload dynamically, which produces communication overhead, or leave the choice to the programmer, who might not have the expertise to decide on a reasonable split.

11.3 The *M*uenster *S*keleton *L*ibrary *M*uesli

Originally, skeletal parallel programming was mainly implemented in functional languages since it derives from functional programming [3]. Today, the majority of skeleton frameworks are based on C/C++ [e.g. 8, 2, 5, 1], since the language is known for good performance and interoperability with low-level parallel frameworks such as CUDA, OpenMP, and MPI. Although Python has become popular in many natural science applications, as packages can be easily written and integrated, this does not apply to calculation intense applications as the language entails a major slowdown. Therefore, especially in the HPC context, C/C++ is still the first choice.

The C++ library used for this work is called Muesli [7]. Muesli provides task- and data-parallel skeletons such as Fold, multiple versions of Map, Gather, and multiple versions of Zip. These operations can be used to write programs for clusters of multiple nodes with multicore processors and GPUs. Internally, it is based on MPI, OpenMP, and CUDA. Muesli relieves the programmer from tasks which require expertise in parallel programming, such as the number of threads started and copying data to the correct memory spaces and helps to avoid common errors in parallel programming such as

race conditions when accessing data structures. Although the additional abstraction causes some overhead, it does not increase the execution time significantly. In contrast to previous versions, Muesli now supports not only distributed arrays (DA) and matrices (DM) but also distributed cubes (DC) as data structures. Especially in the scientific context, e.g. in computational fluid dynamics, cubes are essential to model 3D objects. A distinctive feature of Muesli is that for Map and Zip, there are in-place variants and variants where the index is used for calculations. Additionally, the MapStencil skeleton allows to update each matrix element depending on its neighbors.

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

Listing 11.1: Scalar product in Muesli.

Listing 11.1 shows a simple program for calculating the scalar product of the distributed arrays `a` and `b` (in a slightly simplified syntax). Firstly, a distributed array is created in line 6. A skeleton typically gets a user function as argument, which can be either a C++ function or a C++ functor. We enable currying, i.e. the arguments of a user function can be supplied one by one rather than all together. For instance, the `MapIndex` skeleton in line 7 of Listing 11.1 automatically adds the considered array element and its index as additional parameters to the sum functor.

11.4 Data Distribution and Data structures in Heterogeneous Computing Environments

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

11.4.1 Distributed Cubes

The added data structure cube is similarly designed to previous data structures in Muesli which makes the syntax easy for programmers. For constructing a distributed cube, at least three arguments have to be passed to define the cube's dimensions. Optionally, a default value can be passed to be filled into all elements of the cube. Listing 11.2 creates two distributed cubes `a` and `b`, filled with the default values 0 and 1, respectively. The `mapIndexInPlace` skeleton in line 11 adds to each element its row-index, column-index, and its index of the third dimension. In line 12, each value of `b` is added to the corresponding value of `a`.

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

Listing 11.2: Exemplary cube computation in Muesli.

11.4.2 Segmentation of Data Structures

A simplified version of the approach chosen for the mapStencil skeleton applied to a matrix can be seen in Figure 11.1. Each node is responsible for multiple rows of the data structure, and within each node, the data structure is again split between the available CPUs and GPUs in a row-wise manner.

In the context of stencil calculations, it was reasonable to distribute complete rows or rectangles of data to minimize the required data transfers for communicating border values. Therefore, *always* complete rows were distributed. For skeletons such as Map and Zip, where the calculation does not depend on neighbor values, it is not necessary to distribute complete rows, as incomplete rows do not degrade the execution time. Data transfers are rarely needed. Therefore, it is assumed that distributing complete rows is less important than equally splitting the workload. Figures 11.2 and 11.3 demonstrate how incomplete rows are distributed and how the concept is transferred to a cube. This

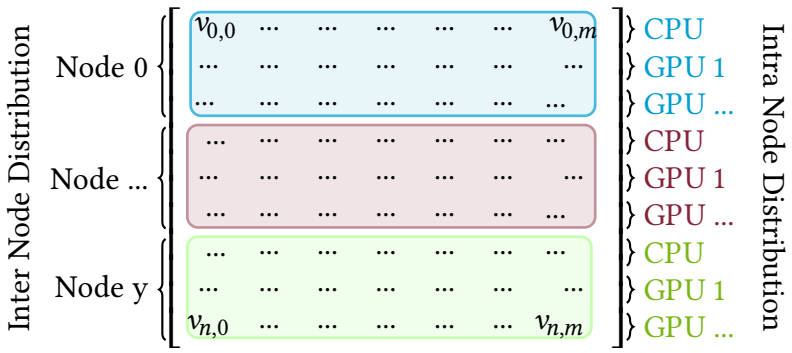


Figure 11.1: Data distribution

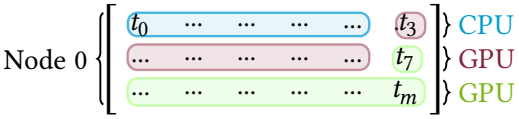


Figure 11.2: Intra-node distribution using multiple accelerators.

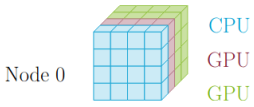


Figure 11.3: Intra-node distribution of a cube

presentation also portions the amount of work (elements calculated) unequally for the two GPUs. Prospectively, Muesli automatically calculates suitable workload splits to relieve the programmer from the fine-tuning the splitting work.

11.4.3 Work-Load Partitioning

The current implementation uses the number of cores of the used GPU to allocate more elements to GPUs, which can start more threads concurrently. This relieves the user from some low-level details for exploiting the available hardware. More precisely, CUDA provides `DeviceProperties` which, amongst others, state the number of multiprocessors available. To calculate the number of cores the function `_ConvertSMVer2Cores(props.major, props.minor) * props.multiProcessorCount;` has to be used as the number of multiprocessors is dependent on the version of the GPU. However, a good approximation of the maximum possible parallelism can be calculated with this reference number. In the future, this number might also be dependent on the version of the GPU to prefer newer GPUs. Besides splitting the workload between multiple GPUs the fraction which is calculated by the CPU has to be automatically chosen by the library. The experimental

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

Table 11.1: Overview of used hardware

results section evaluates which partition is reasonable for different skeletons, determining good default values for different calculation patterns.

11.5 Experimental Results

We have tested varying distribution possibilities with the distributed cubes for the skeletons Map, MapInPlace, MapIndex, MapIndexInPlace, Fold, Zip, ZipIndex, ZipInPlace and ZipIndexInPlace. The distributions include multi-node multi-GPU set-ups and different fractions of calculations which are assigned to the CPU. The runtimes of all experiments are the result of calling skeletons multiple times. For the experiments, the HPC machine Palma II[38] and, for comparison, an ordinary 8-core PC with two different GPUs were used. With Palma, we used the GeForce RTX 2080 Ti GPUs partition equipped with 2 nodes each with 4 GPUs and a Zen3 (EPYC 7513) CPU. Each node has 24 CPU cores. To provide generalizable results, each skeleton was tested on a stand-alone basis. For this purpose, we used multiple sizes and CPU-fractions. For running the sequential version on the HPC, a single Broadwell (E5-2683 v4) CPU was used. We let each skeleton run 25 times (without data transfers between them) to produce meaningful runtimes for the calculations. The PC was used to have a comparison for the discussion of a suitable CPU-fraction. GPUs with fewer streaming multiprocessors can start fewer threads concurrently, making the use of the CPU more reasonable. The PC is equipped with one Quadro K620, one GeForce GTX 750 Ti, and an eight core Intel(R) Core(TM) i7-4790 CPU with 3.60GHz. The sequential version only used one of the available cores and no GPU.

11.5.1 CPU Usage on the PC

Allocating a fraction of the work to the CPU did not speed up the Map Stencil skeleton as Map Stencil has communication overhead for transferring the padding between different

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

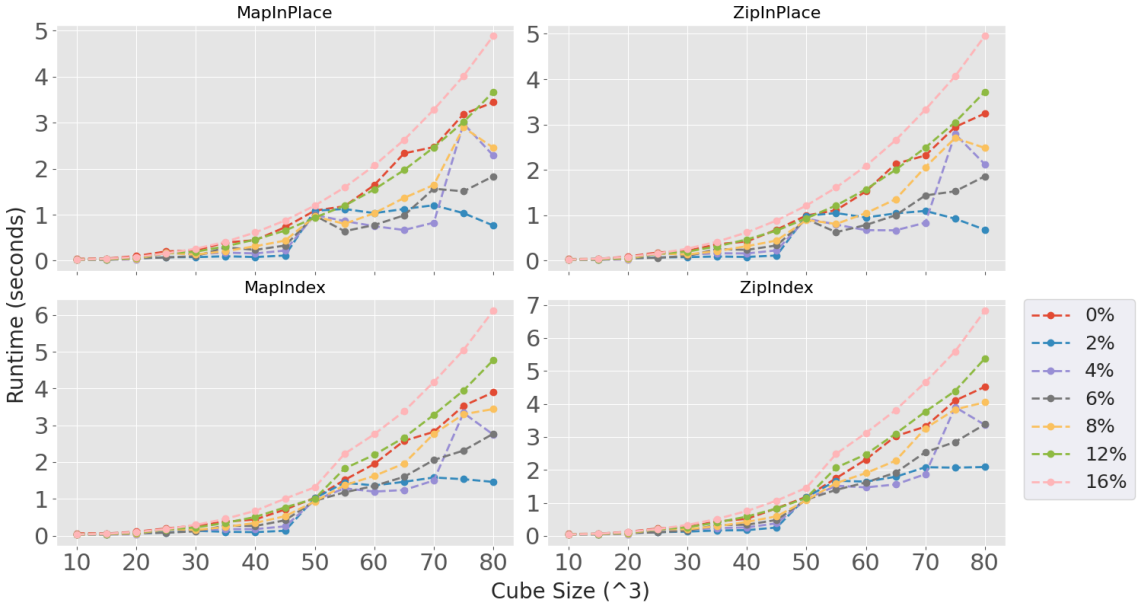


Figure 11.4: Runtimes (in seconds) for different CPU-fractions calculated by the CPU for Map- and Zip variants on the PC.

computational units after each skeleton call. Hence it is reasonable to test CPU-fractions for skeletons which require less communication. Map and Zip are suitable examples as calculations only depend on the current element. Figure 11.4 displays some results of running Map and Zip on the PC. As can be seen with increasing data size, all skeletons are optimal at a CPU-fraction of 2%. CPU-fractions greater than 16% are not displayed as their runtime is increasing as expected. Interestingly, at a data size of 50^3, all runtimes are nearly equal and, from that point on, show clearly a difference. In Table 11.2 exemplary speedups for the mixed usage of the CPUs and the GPU are listed. Our results aim to automatically identify those changing points for the end-user to adjust the generated code to the system. In this context, it is especially noteworthy that eight cores available on the PC provide some significant computation power. Still, the fraction allocated to the CPU is small. Hence hardware with less cores should, by default, not use the CPU for Map and Zip.

In contrast to Map and Zip, creating a new data structure and the Fold skeleton are less calculation intense. Hence it was expected that CPU variants would be faster. Figure 11.5 displays both. Creating a data structure does not require a lot of time. It can be seen that for smaller sizes, the sequential and CPU only version are faster as no GPU memory needs to be allocated. However, for growing data sizes, they are similar. All CPU and GPU mixed variants show no significant difference in their runtimes. As they are finished in milliseconds, this aspect has little influence on the overall runtime of a

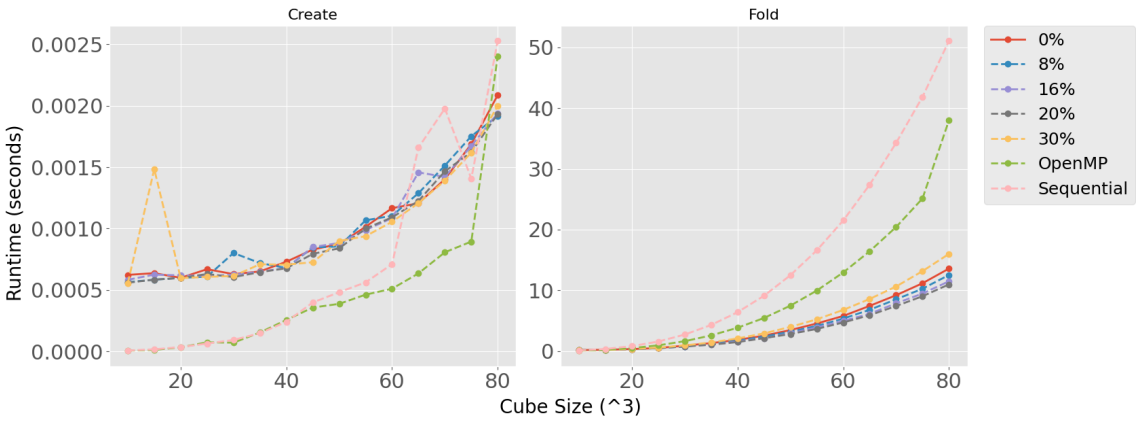


Figure 11.5: Runtimes (in seconds) for different CPU-fractions calculated by the CPU.

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

Table 11.2: Runtimes (in seconds) for the sequential, the OpenMP only version, the GPU only version, and for the optimal mix of CPU and GPU for MapInPlace on the PC. The column CPU % shows which CPU fraction was used in the optimal mix. The following columns show speedups of the optimal mix compared to the sequential version, the OpenMP only version, and the GPU only version.

larger application. In contrast, the Fold skeleton requires a lot of time (around 12-17 s for parallel programs). In contrast to the previous skeletons, Fold performs best for 20 % CPU-fraction and achieves with this setting a speedup of 1.2 compared to the GPU only version for 80^3 elements.

11.5.2 CPU Usage on High-Performance Computing Machine

In contrast to the PC, Palma has relatively strong GPUs. A GeForce RTX 2080 Ti can start up to 69,632 threads in parallel, while the aforementioned GPUs can start 6,144 and 10,240 threads in parallel, respectively. Hence, using the CPU is expected to be less beneficial. In contrast to the PC, skeletons were called up to 10,000 times as otherwise the runtime would have been too short. Figure 11.6 depicts the runtimes for the MapIndex, ZipIndex, MapIndexInPlace, and ZipIndexInPlace skeleton. Two major observations can be made. Firstly, for InPlace variants, no speedup is achieved when using the CPU. This underlines

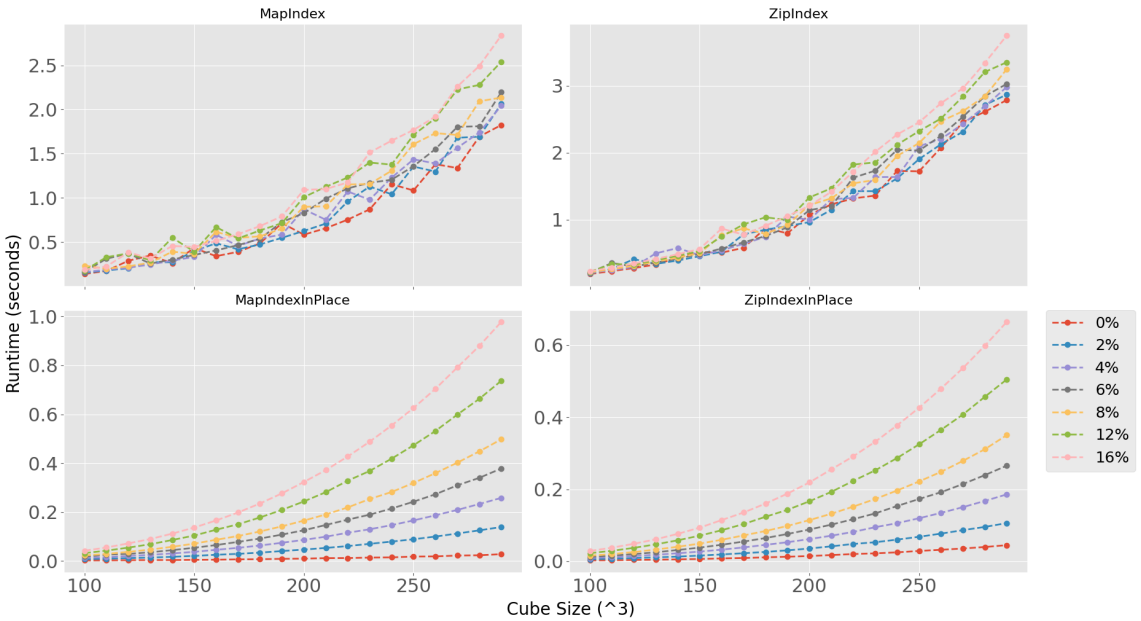


Figure 11.6: Runtimes (in seconds) for different CPU-fractions calculated by the CPU for Map- and Zip variants on the high-performance cluster Palma.

that with extremely powerful GPUs the CPU should not be used for calculations. Secondly, non-InPlace variants show a rather mixed behaviour. No CPU-fraction is clearly winning, but the winner changes almost randomly with the cube size and the differences are small.

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

For the Fold skeleton the ideal CPU fraction is hard to determine. Figure 11.7 shows that all GPU programs perform only as good as the OpenMP program. Conclusively outsourcing calculation to the CPU produces similar runtimes. However, allocating 40-50% of the calculation to the CPU is the best fit in most cases.

11.5.3 Multi-Node and Multi-GPU on the PC

As the PC has just eight cores and only two GPUs, only a moderate speedup is possible by adding more nodes. We have measured the performance of one MPI process with one GPU, one MPI process with two GPUs, and two MPI processes with one GPU each.

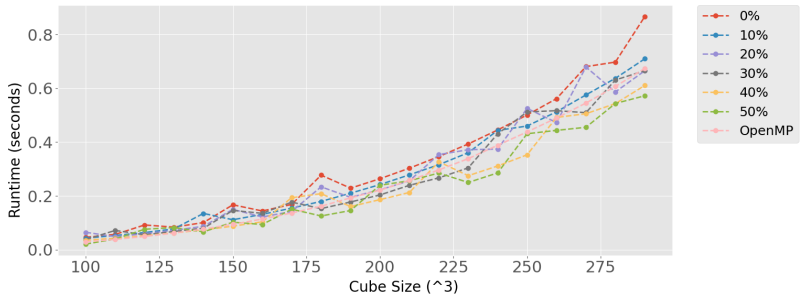


Figure 11.7: Runtimes (in seconds) for different CPU-fractions calculated by the CPU for the Fold skeleton on the high-performance cluster Palma.

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

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

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

Table 11.3: Runtimes (in seconds) and speedups on multiple nodes and GPUs for ZipInPlace on the PC.

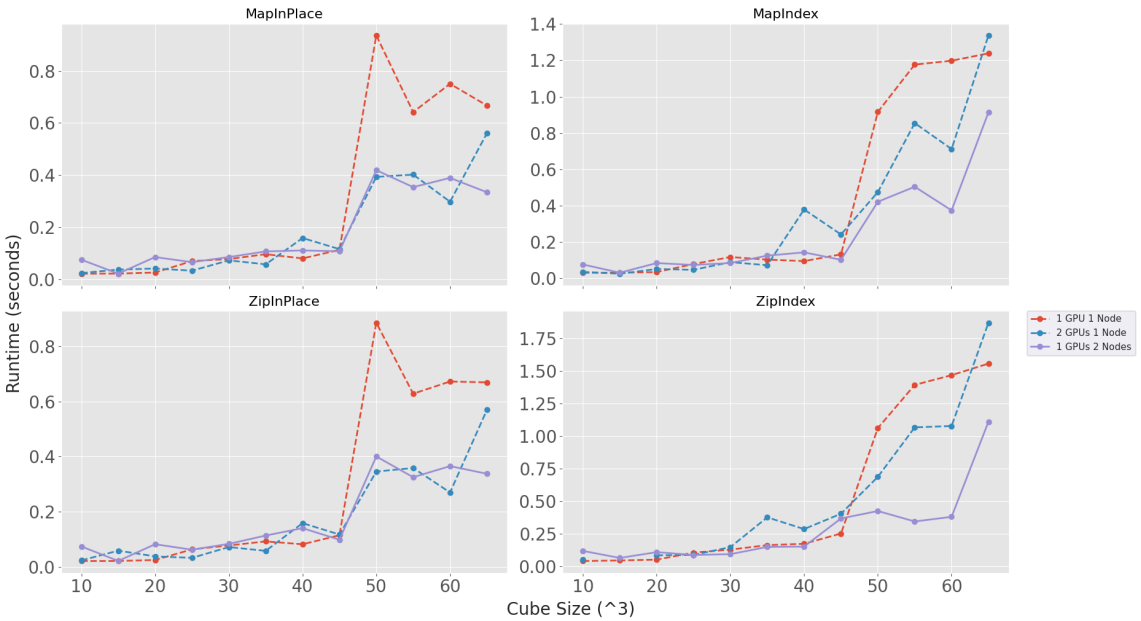


Figure 11.8: Runtimes (in seconds) for multiple nodes and GPUs for Map- and Zip variants on the PC.

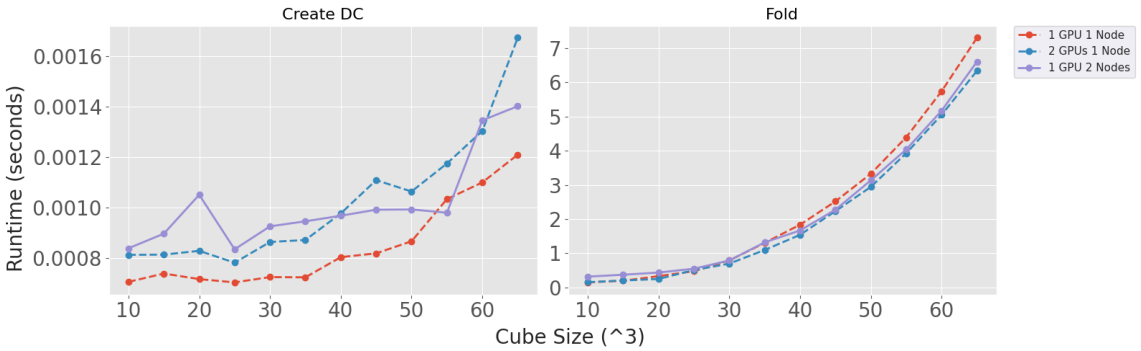


Figure 11.9: Runtimes (in seconds) for multiple nodes and GPUs for Map- and Zip variants on the PC.

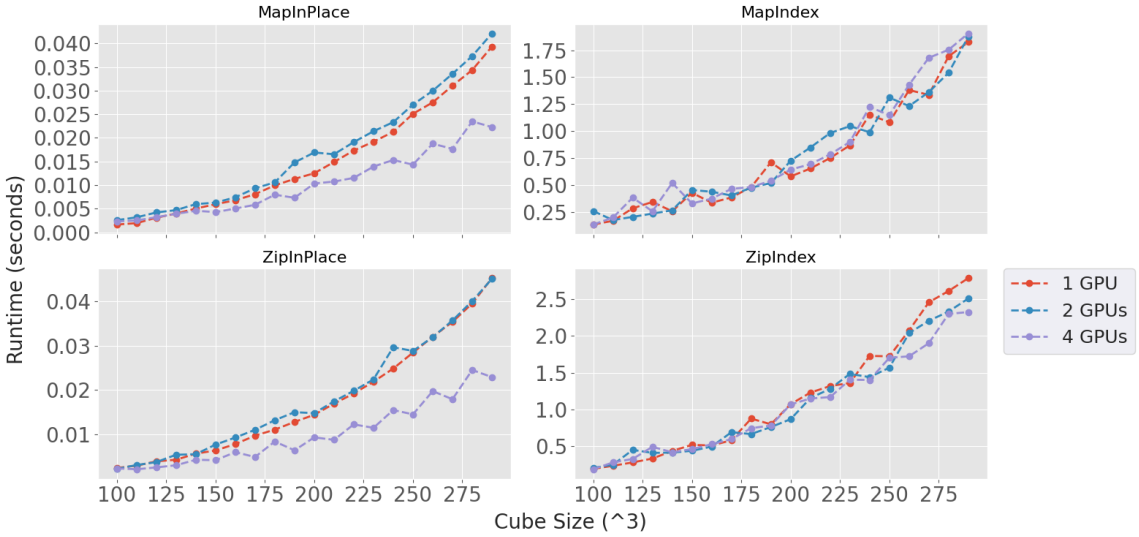


Figure 11.10: Runtimes (in seconds) for multiple GPUs for Map- and Zip variants on the high-performance cluster Palma.

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

On Palma, setups with up to four GPUs per node can be tested. Although initializing additional GPUs produces overhead, the calculations can be distributed and more computational power can be used. Results for different skeletons can be seen in Figure 11.10. For non-InPlace variants, there is nearly no visible speedup between the different GPU versions. The creation of new data structures is dependent on the CPU. Hence using multiple GPUs on one node does not speed up the runtime for skeletons which require the creation of new data structures. In contrast for InPlace skeletons the four GPU variant shows a significant advantage compared to the one GPU variant. However, we cannot understand why the two GPU variant is not faster than the variant using one GPU. Although calling a skeleton produces some overhead, this should not outweigh the calculation time. Therefore, more investigation in this area is required.

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

11.6 Conclusions and Future Work

We have shown that in specific hardware settings and for selected skeletons, using the CPU in addition to the GPUs can speed up the program. This is especially relevant for Map and Zip skeletons as they do not require communication between CPU and GPU in

Figure 11.11: Runtimes (in seconds) for multiple GPUs for the Fold skeleton on the high-performance cluster Palma.

contrast to the Fold skeleton and the creation of data structures. We have also shown that for strong GPUs, it is often more efficient to let the GPUs do all calculations.

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

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 'Fast-flow: high-level and efficient streaming on multi-core'. In: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017).

[2] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. 'Flexible skeletal programming with eSkel'. In: *European Conference on Parallel Processing*. Springer. 2005, pp. 761–770.

[3] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[4] NVIDIA Corporation. *CUDA*. `https://developer.nvidia.com/cuda-zone`. 2021. (Visited on 10/05/2021).

[5] Kento Emoto, Sebastian Fischer, and Zhenjiang Hu. 'Generate, Test, and Aggregate'. In: *Programming Languages and Systems*. Ed. by Helmut Seidl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 254–273.

[6] Steffen Ernsting and Herbert Kuchen. 'Algorithmic skeletons for multi-core, multi-GPU systems and clusters'. In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138.

[7] Steffen Ernsting and Herbert Kuchen. 'Data parallel algorithmic skeletons with accelerator support'. In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299.

[8] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. 'SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters'. In: *International Journal of Parallel Programming* 49.6 (2021), pp. 846–866.

[9] MPI Forum. *MPI Standard*. `https://www.mpi-forum.org/docs/`. 2021. (Visited on 10/05/2021).

[10] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 'Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping'. In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2009, pp. 45–55.

[11] Tomas Öhberg, August Ernstsson, and Christoph Kessler. 'Hybrid CPU–GPU execution support in the skeleton programming framework SkePU'. In: *The Journal of Supercomputing* 76.7 (2020), pp. 5038–5056.

[12] OpenMP. *OpenMP The OpenMP API specification for parallel programming*. `https://www.openmp.org/`. 2021. (Visited on 10/05/2021).

[13] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. 'Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons'. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2019, pp. 1534–1543.

[14] Fábio Soldado, Fernando Alexandre, and Hervé Paulino. 'Towards the transparent execution of compound OpenCL computations in multi-CPU/multi-GPU environments'. In: *European Conference on Parallel Processing*. Springer. 2014, pp. 177–188.

[15] Michel Steuwer and Sergei Gorlatch. 'SkelCL: a high-level extension of OpenCL for multi-GPU systems'. In: *The Journal of Supercomputing* 69.1 (2014), pp. 25–33.

[16] Fabian Wrede, Christoph Rieger, and Herbert Kuchen. 'Generation of high-performance code based on a domain-specific language for algorithmic skeletons'. In: *The Journal of Supercomputing* 76.7 (2020), pp. 5098–5116.

12

Optimizing Three-Dimensional Stencil-Operations on Heterogeneous Computing Environments

Nina Herrmann* · **Justus Dieckmann*** · **Herbert Kuchen***

Citation Nina Herrmann, Justus Dieckmann, and Herbert Kuchen. 'Optimizing Three-Dimensional Stencil-Operations on Heterogeneous Computing Environments'. Accepted for publication in the International Journal of Parallel Programming. 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.

*University of Münster, Germany

12.1 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) [27], OpenMP [36], and CUDA [28]. 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 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 [9]. Multiple libraries [12, 6], general frameworks [13, 2], and domain-specific languages [39] 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 (Section 12.2). Section 12.3 outlines the Muesli library used, while Section 12.4 explains the additional implementation of the three-dimensional skeleton and the examples used to measure the runtime. Section 12.5 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 Section 12.6.

12.2 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 12.1 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 [13]. Celerity is a high-level approach focusing on parallel for-loops, including stencil operations [37]. FastFlow added GPU support but focuses on communication skeletons and misses a comparable stencil operation [2, 15]. Lift handles n-dimensional stencils on single GPUs missing the combination of multiple nodes and accelerators [17]. SkelCL implements a MapOverlap skeleton for multiple GPUs missing multiple nodes [35]. 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 [5]. EPSILOD tests prototypes for stencil operations but misses a generic stencil implementation [7]. ExaStencil provides a domain-specific language for stencil operations, therefore it misses the option to parallelize pre- and post-processing steps [21]. DUNE provides the means to write parallel programs in C++ but merely for CPUs[3]. Partans and PSkel developed a framework for stencil operations but have not been adopted [25] [31]. Pereira et al. have expanded OpenACC for single GPUs stencil operations, however, their project is not publically available[30]. Palabos proved that frameworks generating GPU code (npFEM) can be included in the framework to solve the Lattice Boltzmann methods (LBMs) [22], however it is unclear if this feature is still supported. Publications discussing a single method, e.g., the Helmholtz equation [16] 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.

[41] Currently not working for the MapStencil skeleton
[42] No option to create the stencil required for an LBM implementation
[43] enabled through OpenACC directives

Name	Single GPU	Multi-GPU[39]	Generic Operations	Publically Available	Updated
SkePU [13]	✓	✓ [40]	✓	✓ [34]	until now
Celerity	✓	✓	✓	✓ [8]	until now
Lift [17]	✓	X	✓	✓ [24]	until now
SkelCL [35]	✓	✓	X	✓ [33]	2016
lbmpy [5][4]	✓	X	X	✓ [38] [23]	until now
EPSILOD [7]	✓	✓	X[41]	✓ [11]	until now
ExaStencil [21]	✓	✓	X	✓ [14]	until now
DUNE [3]	X	X	X	✓ [10]	until now
PARTANS [25]	✓	✓	X	X	2013
PSkel [31]	✓	X	X	✓ [32]	2017
OpenACC Extension [30]	✓	X	✓ [42]	X	2017
Palabos [22]	✓ [43]	X	X	✓ [29]	until now

Table 12.1: Overview of frameworks that generate parallel code for stencil operations.

12.3 The *Muenster Skeleton Library Muesli*

Nowadays, most skeleton frameworks are implemented in C/C++ [13, 2, 6, 26, 1, 18], 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++ [4, 5].

The used library is called Münster Skeleton Library (Muesli) [12]. 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 12.1 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 12.1: Scalar product in Muesli.

12.4 Three-Dimensional Stencil Operations

Stencil operations are map operations that additionally require reading the surrounding elements of each considered element. Figure 12.1 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 pro-

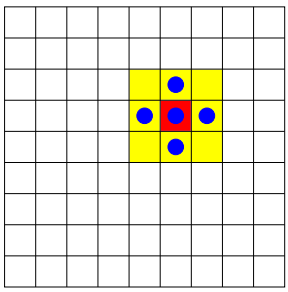


Figure 12.1: Exemplary two-dimensional stencil operation.

gram. Muesli abstracts from all communication between the computational nodes with a MapStencil skeleton.

12.4.1 Using the MapStencil Skeleton in Muesli

The usage of the 3-dimensional `mapStencil` skeleton for the end-user is shown in Listing 12.2. Firstly, a function to be executed on each element is defined (l. 1-10). 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 [44], the radius of the stencil[45] 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 12.2: Exemplary functor for the `mapStencil` skeleton.

12.4.2 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

[44] 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.

[45] Other stencil shapes such as rectangular or irregular stencils can be handled by using the smallest surrounding cube, this may introduce overhead.

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,
- `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 12.3, 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 12.3: Implementation of the `mapStencil` skeleton.

12.4.3 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 [20, 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 (12.2)). Δt is the period to be simulated.

$$f_i(x + c_i\Delta t, t + \Delta t) := f_i^*(x,t) \tag{12.1}$$

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

$$f_i^*(x,t) := f_i(x,t) - \frac{\Delta t}{\tau}\left(f_i(x,t) - f_i^{\text{eq}}(x,t)\right). \tag{12.2}$$

The equilibrium state is calculated by

$$f_i^{\text{eq}}(x,t) := w_i \rho \left(1 + \frac{u \cdot c_i}{c_s^2} + \frac{u \cdot c_i}{2c_s^4} + \frac{u \cdot u}{2c_s^2}\right), \tag{12.3}$$

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 ρ and the puls density u are defined by

$$\rho(x,t) = \sum_i f_i(x,t), \qquad \rho u(x,t) := \sum_i c_i f_i(x,t). \tag{12.4}$$

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 12.4).

```
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 12.4: 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 12.5: User function of the LBM.

The second example used is a mean filter commonly used for smoothing images (user function in Listing 12.2). 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.

12.5 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 programs, a mean filter and an LBM implementation, are executed on the HPC machine Palma II[46]. Table 12.2 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`.

		per node		per computational unit		
Identifier	Nodes	GPUs	CPU-threads	mem. (GB)	cores	GPU/CPU-type
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

Table 12.2: Overview of used hardware.

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.

[46] https://confluence.uni-muenster.de/pages/viewpage.action?pageId=27755336

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[47].

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

$$d(\text{gb}) := \sqrt[3]{\text{gb} \cdot \frac{2^{30}}{2 \cdot 76}} \tag{12.5}$$

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 10 hours for calculating the LBM simulation for a data size of 960^3).

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

[47]For further reference the program code can be viewed https://github.com/justusdieckmann/ba-native

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 (960^3). Eight nodes achieve a speedup of 452. Table 12.3 shows that for the CPU-zen2 partition, a speedup of 116 can be reached with a single CPU.

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

Table 12.3: Runtimes (seconds) and speedups for the parallel implementation of the LBM gas simulation for CPU programs on the zen2 partition.

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.

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 Figure 12.2, 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

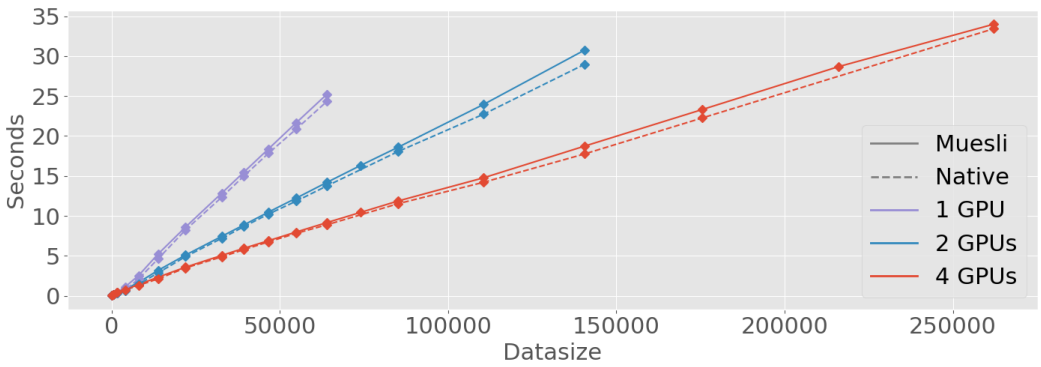


Figure 12.2: Runtime comparison of a multi-GPU Muesli program and native (CUDA) implementation of the LBM on a single-node of the gpu2080 partition.

and can be run on multiple nodes. Moreover, it should be considered that writing native code is arduous as each line requires fine-tuning.

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

Table 12.4: Speedup for the parallel implementation of the LBM gas simulation on the gpu2080 partition.

Regarding the scalability on multiple GPUs, an extract of the achieved speedups can be seen in Table 12.4. 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 Figure 12.3 and Table 12.5. 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

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

Table 12.5: Speedup for the parallel implementation of the LBM gas simulation for multiple nodes on the gpu2080 partition.

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.

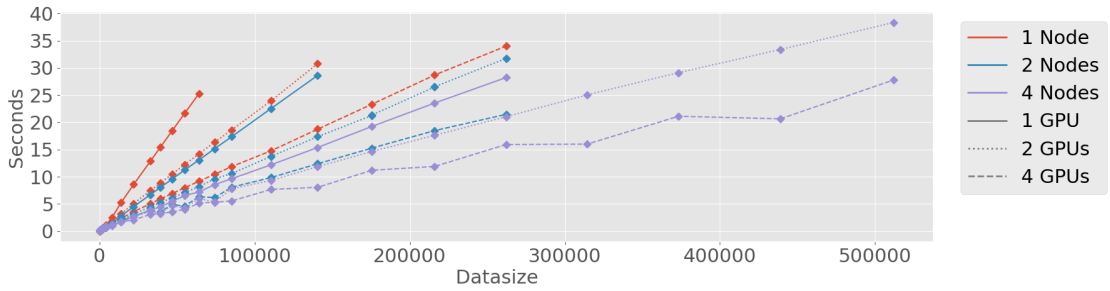


Figure 12.3: Runtimes of the LBM Muesli program on multiple nodes and multiple GPUs on the gpu2080 partition.

12.5.2 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

	Runtime (s)			Speedup	
Data size	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

Table 12.6: Runtimes in seconds and speedups of a CPU program for the mean filter with stencil radius 2 on the zen2 partition.

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 12.6 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 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.

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

Table 12.7: 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	1 GPU -Com[1]	4 GPUs -Com[1]	Speed-up 1/4-Com
2	120^3	0.15	579.02	0.08	2.00	0.09	0.03	3.08
	400^3	2.66	**1253.15**	1.12	2.38	2.31	0.65	3.53
	800^3	24.88	-	7.41	3.36	22.09	5.81	3.80
10	120^3	5.48	842.20	1.87	2.94	4.02	1.50	2.69
	280^3	75.39	887.78	20.41	3.69	65.68	18.68	3.52
	800^3	2149.16	-	541.64	3.97	2009.69	510.79	3.93

Table 12.8: Speedup for the parallel implementation of the mean blur for two nodes of the gpu2080 partition.

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 400^3 is 747, significantly better than for the LBM implementation. Table 12.7 and 12.8 list the runtimes and speedups for a small stencil

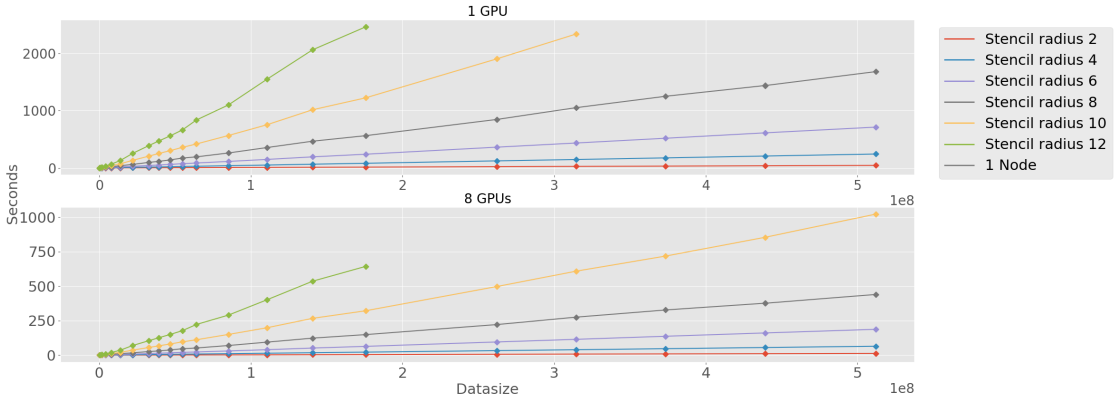


Figure 12.4: Runtimes of the Mean Blur Muesli program on the gpu2080 partition.

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 seconds 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 12.4 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 12.9. 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 communicated between

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

Table 12.9: Speedup for the parallel implementation of the mean blur for multiple GPUs on the gpu2080 partition.

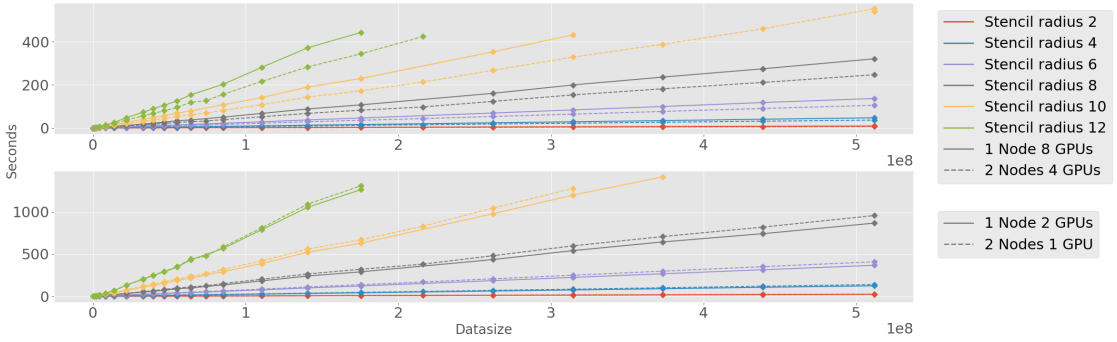


Figure 12.5: Runtimes of the Blur Muesli program on the gpu2080 partitions with similar hardware.

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 Figure 12.5. The first figure compares a one-node eight GPUs program to a two-node four GPUs program for different stencil radiuses. 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.

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

Table 12.10: Runtimes in seconds and speedups for a mean filter on the gpuhgx partition

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 GPUs (2309.08). According to the previous approach, the runtime was measured with and without communication (Table 12.10). Even with the communication, the speedup is close to the optimum. For four GPUs, a speedup of 3.65 and 3.93 can be reached for 400^3 and 800^3 elements. For eight GPUs, a speedup of 5.18 and 7.55 is reached.

12.5.3 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 12.11 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 280^3 was the maximum size possible while the Muesli program could run cubes of size 400^3.

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 (Figure 12.6) as the SYCL compiler did not unroll some for loops. Manually inserting a `#pragma unroll`

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

Table 12.11: 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.

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.

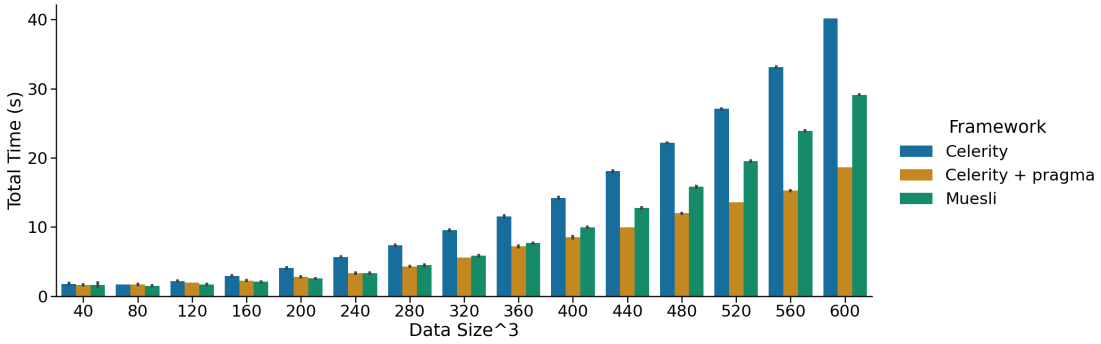


Figure 12.6: Runtime comparison of a multi-GPU Muesli program and a multi-GPU celerity program of the LBM on the gpuhgx partition.

12.6 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.

References

[1] Enrique Alba, Gabriel Luque, Jose Garcia-Nieto, Guillermo Ordonez, and. 'MALLBA: a software library to design efficient optimisation algorithms'. In: *International Journal of Innovative Computing and Applications* 1.1 (2007), pp. 74–85.

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 'Fastflow: high-level and efficient streaming on multi-core'. In: *in: Programming multi-core and many-core computing systems, parallel and distributed computing* (2017), pp. 261–280.

[3] Peter Bastian et al. 'The Dune framework: Basic concepts and recent developments'. In: *Computers & Mathematics with Applications* 81 (2021). Development

and Application of Open-source Software for Problems with Numerical PDEs, pp. 75–112.

[4] Martin Bauer, Harald Köstler, and Ulrich Rüde. 'lbmpy: Automatic code generation for efficient parallel lattice Boltzmann methods'. In: *Journal of Computational Science* 49 (2021), p. 101269.

[5] Martin Bauer et al. 'waLBerla: A block-structured high-performance framework for multiphysics simulations'. In: *Computers & Mathematics with Applications* 81 (2021). Development and Application of Open-source Software for Problems with Numerical PDEs, pp. 478–501.

[6] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. 'Flexible Skeletal Programming with eSkel'. In: *Euro-Par 2005 Parallel Processing*. Ed. by José C. Cunha and Pedro D. Medeiros. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 761–770.

[7] Manuel de Castro, Inmaculada Santamaria-Valenzuela, Yuri Torres, Arturo Gonzalez-Escribano, and Diego R Llanos. 'EPSILOD: efficient parallel skeleton for generic iterative stencil computations in distributed GPUs'. In: *The Journal of Supercomputing* (2023), pp. 1–34.

[8] *Celerity*. url: https://github.com/celerity/celerity-runtime (visited on 13/02/2024).

[9] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Computer science thesis. London: Pitman, 1989.

[10] *DUNE*. url: https://gitlab.dune-project.org/core (visited on 13/02/2024).

[11] *EPSILOD*. url: https://gitlab.com/trasgo-group-valladolid/controllers/-/tree/epsilod_JoS22 (visited on 13/02/2024).

[12] Steffen Ernsting and Herbert Kuchen. 'Data parallel algorithmic skeletons with accelerator support'. In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299.

[13] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. 'SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters'. In: *International Journal of Parallel Programming* 49.6 (2021), pp. 846–866.

[14] *ExaStencil*. url: https://github.com/lssfau/ExaStencils (visited on 13/02/2024).

[15] Mehdi Goli and Horacio González-Vélez. '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*. 2013, pp. 148–156.

[16] Ronald Gonzales, Yury Gryazin, and Yun Teck Lee. 'Parallel FFT algorithms for high-order approximations on three-dimensional compact stencils'. In: *Parallel Computing* 103 (2021), p. 102757.

[17] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 'High Performance Stencil Code Generation with Lift'. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 100–112.

[18] Yuki Karasawa and Hideya Iwasaki. 'A Parallel Skeleton Library for Multi-core Clusters'. In: *2009 International Conference on Parallel Processing*. 2009, pp. 84–91.

[19] Christos Kotsalos, Jonas Latt, and Bastien Chopard. 'Palabos-npFEM: Software for the simulation of cellular blood flow (digital blood)'. In: *arXiv preprint arXiv:2011.04332* (2020).

[20] Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. *The Lattice Boltzmann Method: Principles and Practice*. eng. Graduate Texts in Physics. Cham: Springer International Publishing AG, 2016.

[21] Sebastian Kuckuk and Harald Köstler. 'Whole Program Generation of Massively Parallel Shallow Water Equation Solvers'. In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 2018, pp. 78–87.

[22] Jonas Latt et al. 'Palabos: Parallel Lattice Boltzmann Solver'. In: *Computers & Mathematics with Applications* 81 (2021). Development and Application of Open-source Software for Problems with Numerical PDEs, pp. 334–350.

[23] *lbmpy*. url: `https://i10git.cs.fau.de/pycodegen/lbmpy` (visited on 13/02/2024).

[24] *lift*. url: `https://github.com/lift-project/lift/tree/master` (visited on 13/02/2024).

[25] Thibaut Lutz, Christian Fensch, and Murray Cole. 'PARTANS: An autotuning framework for stencil computation on multi-GPU systems'. In: 9.4 (Jan. 2013).

[26] Ricardo Marques, Hervé Paulino, Fernando Alexandre, and Pedro D. Medeiros. 'Algorithmic Skeleton Framework for the Orchestration of GPU Computations'. In: *Euro-Par 2013 Parallel Processing*. Ed. by Felix Wolf, Bernd Mohr, and Dieter an Mey. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 874–885.

[27] *MPI Standard*. https://www.mpi-forum.org/docs/. (Visited on 24/02/2023).

[28] NVIDIA. *CUDA*. https://developer.nvidia.com/cuda-zone. (Visited on 24/02/2023).

[29] *palabos*. url: https://gitlab.com/unigespc/palabos (visited on 13/02/2024).

[30] Alyson D Pereira, Márcio Castro, Mario AR Dantas, Rodrigo CO Rocha, and Luís FW Góes. 'Extending OpenACC for efficient stencil code generation and execution by skeleton frameworks'. In: *2017 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2017, pp. 719–726.

[31] Alyson D. Pereira, Luiz Ramos, and Luís F. W. Góes. 'PSkel: A stencil programming framework for CPU-GPU systems'. In: *Concurrency and Computation: Practice and Experience* 27.17 (), pp. 4938–4953.

[32] *PSkel*. url: https://github.com/pskel/pskel (visited on 13/02/2024).

[33] *SkelCL*. url: https://github.com/skelcl/skelcl (visited on 13/02/2024).

[34] *skepu*. url: https://github.com/skepu/skepu/ (visited on 13/02/2024).

[35] Michel Steuwer and Sergei Gorlatch. 'SkelCL: a high-level extension of OpenCL for multi-GPU systems'. In: *The Journal of Supercomputing* 69.1 (2014).

[36] *The OpenMP API specification for parallel programming*. https://www.openmp.org/. (Visited on 24/02/2023).

[37] Peter Thoman, Florian Tischler, Philip Salzmann, and Thomas Fahringer. 'The celerity high-level api: C++ 20 for accelerator clusters'. In: *International Journal of Parallel Programming* 50.3-4 (2022), pp. 341–359.

[38] *WaLBerla*. url: https://i10git.cs.fau.de/walberla/walberla (visited on 13/02/2024).

[39] Fabian Wrede, Christoph Rieger, and Herbert Kuchen. 'Generation of high-performance code based on a domain-specific language for algorithmic skeletons'. In: *The Journal of Supercomputing* 76.7 (2020), pp. 5098–5116.

A

Appendix

A.1 Heterogeneous Systems

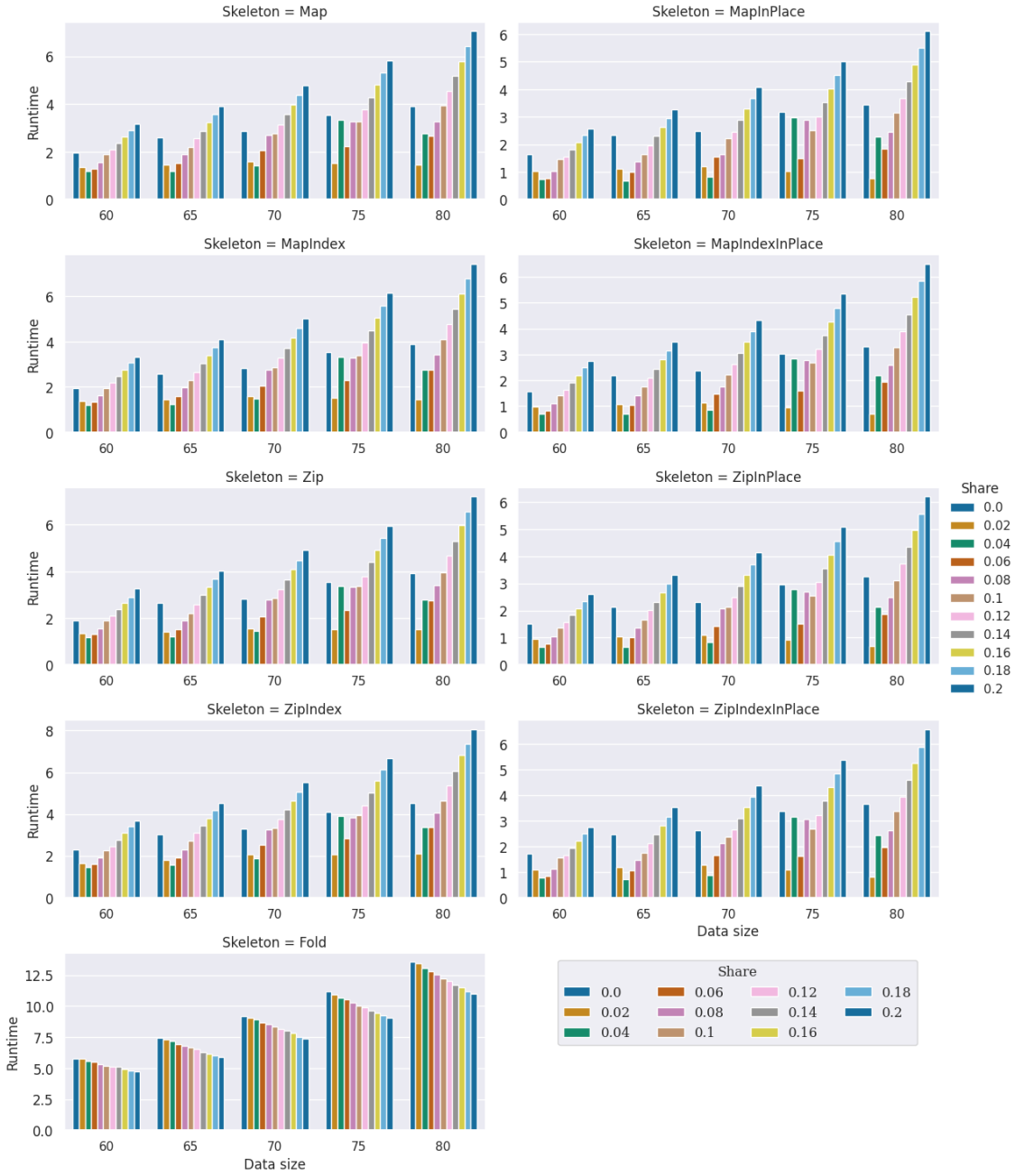


Figure A.1: Runtimes (in seconds) for different CPU fractions calculated by the CPU for Map- and Zip variants on a GeForce RTX 750 Ti.

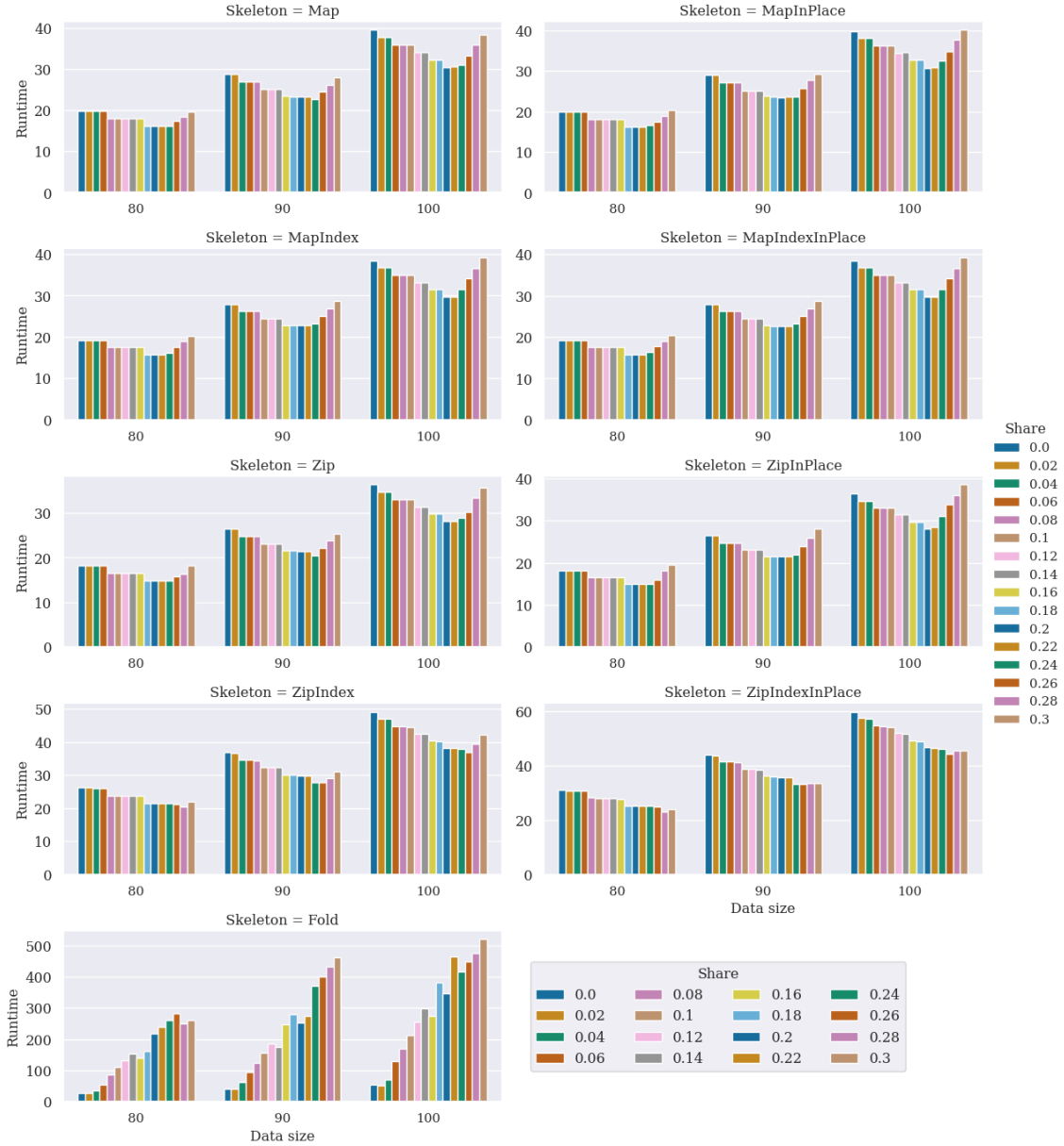


Figure A.2: Runtimes (in seconds) for different CPU fractions calculated by the CPU for Map- and Zip variants on a GeForce RTX 3070 Ti.

A.2 Ant-Colony-Optimization

Listing A.1 displays the full function of the Ant Colony Optimization (ACO) program solving the Traveling Salesperson Problem (TSP) program.

```
template<typename T>
void aco(int iterations, int nants, DA<city> &cities, int ncities, const std::string& importFile ) {
    msl::startTiming();
    msl::DM<double> phero = createPheroMatrix(ncities);
    msl::DM<double> distance(ncities, ncities, 0);
    CalcDistance calcdistance(cities.getUserFunctionData());
    distance.mapIndexInPlace(calcdistance);
    distance.updateHost();
    msl::DM<int> iroulette = createIRoulette(distance, ncities);
    msl::DM<double> etatau(ncities, ncities, 0);
    msl::DA<T> routes(nants, {});
    Fill<T> fill(-1, ncities);
    Min<T> min(ncities);
    T minroute;
    EtaTauCalc etataucalc;
    int veryGoodSeed = (int) time(nullptr);
    TourConstruction<T> tourConstruction(ncities, veryGoodSeed);
    UpdatePhero<T> updatephero(nants);
    double dsinit = msl::stopTiming();
    T alltimeminroute = {};
    for (int i = 0; i < iterations; i++) {
        routes.mapInPlace(fill);
        // Write the eta tau value to the data structure.
        etatau.zipIndexInPlace3(distance, phero, etataucalc);
        tourConstruction.setIterationParams(iroulette.getUserFunctionData(),
        etatau.getUserFunctionData(),
        distance.getUserFunctionData());
        routes.mapIndexInPlace(tourConstruction);
        // Get the best route.
        minroute = routes.foldCPU(min);
        if (i == 0) {
            alltimeminroute = minroute;
        } else {
            if (minroute[ncities+1] < alltimeminroute[ncities+1])
                alltimeminroute = minroute;
        }
        updatephero.setIterationsParams(routes.getUserFunctionData());
        // Update the pheromone.
```

```
39          phero.mapIndexInPlace(updatephero);
40      }
41      double calctime = msl::stopTiming();
42      printf("s;d;s;f;f;f;", importFile.c\_str(), nants, "singlekernel",
43      calctime, dsinit+calctime, alltimeminroute.getDist());
44
45      routes.updateHost();
46      if (CHECKCORRECTNESS) {
47          checkminroute(nants, minroute.getDist(), routes);
48          checkvalidroute(routes, ncities, nants);
49      }
50  }
```

Listing A.1: ACO implementation in Muesli

Table A.1 displays all data collected for the comparison of the Muesli Program and the Program provided in [59].

A.3 Stencil Operations

The number of iterations for the Jacobi method was increased to 10000 iterations, and additionally, a 10000×10000 matrix was added. We included a check to ensure that the results do not differ by more than 0.001. However, for the tested data sizes, the program never reached that point. In Table A.2, the runtimes for starting 64 threads (8×8 tile) and 256 (16×16 tile) threads are shown. Starting a 32×32 tile produced similar results and is therefore not depicted in the table. The runtimes highlighted are the fastest. No clear tendency between the variants tested can be seen.

City	# Ants	Muesli	Program [59]	Speedup
d38	1024	0.0933	0.1757	1.8832
	2048	0.1315	0.326	2.4791
	4096	0.217	0.6412	2.9548
	8192	0.3498	1.2845	3.6721
c194	1024	0.2495	1.3557	5.4337
	2048	0.2644	2.4732	9.354
	4096	0.2971	4.8358	16.2767
	8192	0.3571	9.5298	26.6866
d198	1024	0.2492	1.3688	5.4928
	2048	0.261	2.5417	9.7383
	4096	0.3088	4.8534	15.717
	8192	0.3608	9.5866	26.5704
lin318	1024	0.4106	2.8236	6.8768
	2048	0.4112	5.0164	12.1994
	4096	0.4756	9.4517	19.8732
	8192	0.5832	18.1343	31.0945
pcb442	1024	0.5793	4.6862	8.0894
	2048	0.6134	8.1128	13.226
	4096	0.7089	14.815	20.8986
	8192	0.8708	28.0945	32.2629
rat783	1024	1.2126	11.6556	9.6121
	2048	1.3155	18.8896	14.3593
	4096	1.5129	33.0573	21.8503
	8192	1.9354	61.5159	31.7846
lu980	1024	1.2932	16.3033	12.6069
	2048	1.5612	25.5904	16.3915
	4096	2.0739	43.9838	21.2083
	8192	3.2948	80.3289	24.3805
pr1002	1024	1.7688	17.7638	10.0429
	2048	1.9396	27.9204	14.3949
	4096	2.2342	47.6588	21.3315
	8192	2.8749	86.9366	30.2399
pcb1173	1024	2.2129	213.4618	96.4625
	2048	2.4558	402.093	163.732
	4096	2.8677	767.8529	267.7591
	8192	3.7619	1495.0848	397.4281
d1291	1024	2.6502	284.276	107.2659
	2048	2.8965	539.1764	186.1476
	4096	3.3563	1029.2567	306.6641
	8192	4.4007	2000.2605	454.5323
pr2392	1024	7.8075	1885.6974	241.5238
	2048	8.5712	3662.4705	427.2996
	4096	10.5288	7151.7166	679.2528
	8192	14.411	14028.3833	973.4497

Table A.1: Runtimes (in seconds) for the Muesli program and the program provided by Menezes et al. [59]

Matrix size	1 Node					
	1 GPU			4 GPUs		
	GM	SM 8^2	SM 16^2	GM	SM 8^2	SM 16^2
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**
10000^2	**93.156**	100.205	99.569	**69.092**	73.108	73.146
Matrix size	2 Nodes					
	1 GPU			4 GPUs		
	GM	SM 8^2	SM 16^2	GM	SM 8^2	SM 16^2
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**
10000^2	**48.285**	51.402	51.311	39.388	**38.673**	39.675

Table A.2: Runtimes (in seconds) on multiple nodes and GPUs per node using GPU global memory (GM) vs. shared memory (SM) for the Jacobi Solver.

Affidavit

I declare in lieu of an oath that I have independently written the submitted dissertation with the title *'The Algorithmic Skeleton Library Revisited.'*. I have documented the participation of co-authors by means of a declaration in accordance with § 15 paragraph 5 sentence 3 of the Promotionsordnung. I have not made use of any sources and aids other than those indicated by me. I have marked all paragraphs taken verbatim or in spirit from the writings of other authors. I assure that my dissertation has not already been submitted elsewhere as an examination paper.

Münster, 5th November 2024 Nina Herrmann

Nina Herrmann

Date of birth:	December 07th, 1995
Place of birth:	Hamburg, Germany
Nationality:	German

Education

since 12/2019	PhD candidate at the School of Business and Economics University of Münster, Germany
10/2017 – 09/2019	Master of Science in Information Systems University of Münster, Germany
10/2014 – 09/2017	Bachelor of Science in Information Systems University of Münster, Germany

Academic Career

since 2019	*Research Assistant* at the Chair for Practical Computer Science, Department of Information Systems, University of Münster, Germany
04/2016 – 11/2019	*Student Assistant* at the ZHL Digital, University of Münster, Germany

List of Publications

Since the year 2019, the following scientific publications that I authored and co-authored have been published or are accepted.

1. Breno Menezes, Nina Herrmann, Herbert Kuchen, and Fernando Buarque de Lima Neto. 'High-level parallel ant colony optimization with algorithmic skeletons'. In: *International Journal of Parallel Programming* 49.6 (2021), pp. 776–801

2. Nina Herrmann, Breno Menezes, and Herbert Kuchen. 'Stencil calculations with algorithmic skeletons for heterogeneous computing environments'. In: *International Journal of Parallel Programming* 50.5-6 (2022), pp. 433–453

3. Nina Herrmann and Herbert Kuchen. 'Distributed Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments'. In: *International Journal of Parallel Programming* 51.2-3 (2023), pp. 172–185

4. Nina Herrmann, Justus Dieckmann, and Herbert Kuchen. 'Optimizing Three-Dimensional Stencil-Operations on Heterogeneous Computing Environments'. Accepted for publication in the International Journal of Parallel Programming. 2024