DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Solving the Shallow Water Equations on Heterogeneous Architectures with Kokkos

Dominik Mehringer

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Solving the Shallow Water Equations on Heterogeneous Architectures with Kokkos

Lösen der Flachwassergleichungen auf heterogenen Architekturen mit Kokkos

Author:	Dominik Mehringer
Supervisor:	Prof. Dr. Michael Bader
Advisor:	M.Sc. Alexander Pöppl, M.Sc. Philipp Samfass
Submission Date:	18.05.2020

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 18.05.2020 Dominik Mehringer

Abstract

Parallel computing makes use of various system architectures and hardware configurations, as the application context typically determines suitable machines. It is really challenging for an application programmer to optimize memory management and leverage hardware traits, especially in those cases in which the platform may change. In this bachelor's thesis, the speed-up of computation as well as the performance portability using the framework "Kokkos" in the context of shallow water equations is investigated. The framework generates performance portable code for heterogeneous architectures, which optimizes computation time independently of the underlying hardware. It is achieved by providing an abstraction of the interfaces of the computational devices and using hardware specific characteristics like data layout or memory performance. Furthermore the LRZ-Cluster is used to compare the legacy implementation with the Kokkos implementation using several Intel KNL processors. In order to check performance portability the implementation is also evaluated using different GPU generations. As the results show, Kokkos is indeed generating performance portable code, which is mostly even faster than the legacy approaches.

Contents

1 Introduction

1.1 Motivation

The ever increasing amount of data produced, processed and evaluated by scientific applications on high performance computing clusters requires advancements with respect to architectures and hardware technology. Even though the power of computer systems increased exponentially over the past decades, CPU clock frequencies nowadays stagnate in growth so that the solutions need to be parallelized effectively. As there are several computing devices that can be used for parallelization on hardware-level, such as multicore CPUs (Intel i7), manycore CPUs (Intel Xeon Phi) and GPUs (NVIDIA Titan Xp), different configurations for the platform may occur. It is especially challenging for an application programmer to fully utilize the computational power if the target platform is heterogeneous (in terms of computational devices having differing properties such as the number of cores or energy consumption), because load balancing and synchronization are difficult to implement [1, pp. 1–6].

High Performance Computing (HPC) across different platforms and configurations can lead to portability problems regarding the performance [1, p. 6]. There are many parallelization libraries, like OpenMP, CUDA and MPI, that target specific devices or the communication between the parallel processes. In order to focus on the problem instead of the optimization for the underlying architecture, the Kokkos framework is introduced to tackle this issue and provide an abstraction for different computing devices to ensure an optimized performance on parallel systems without changing the source code. The way of managing and accessing data has a high impact on performance and the data access pattern is dependent on the device. Kokkos provides multidimensional arrays that choose a suitable layout for the specific memory access pattern. This yields several advantages, as it optimizes the code for the system architecture and specific hardware features [2].

1.2 Structure and Related Work

This thesis introduces the theory behind shallow water equations and technical aspects of heterogeneous architectures. Afterwards, the project setup, in which the Kokkos framework is embedded, will be explained and the capabilities of Kokkos are briefly introduced. The following chapters elaborate the integration of Kokkos into the SWE project and the challenges that were faced when addressing performance portability of the legacy code. Evaluating and comparing the results to the legacy code is important in order to measure the impact of the Kokkos enhancements. The last chapter gives a conclusion and an outlook for future work.

There are several applications that make use of Kokkos. The application *miniMD*, for example, is a molecular dynamics project developed at Sandia National Laboratories. It is written in C++ and can be run on any parallel computer due to the scalable implementation [3]. A more detailed list of applications written with the use of Kokkos can be found here [4]. As H. C. Ewards et al. [2] show, the Kokkos implementation is mostly performing better than the legacy implementation when evaluting the performance on mini-applications.

The SWE project is developed for teaching purposes and changes are constantly added. It provides several solvers and approaches for the parallelization of the shallow water equations. Bader et al. [5], for example, use the project for the evaluation of the vectorized augmented Riemann solver. They evaluated their examples using AVX and SSE4, and achieved a massive speed-up, even though the implementation of complex numerical algorithms using SIMD is not as easy as it may seem. Several implementations of different libraries are already evaluated in the thesis' of J. Olden [6] and M. Bogusz [7].

2 Background

2.1 Theory

This section compares different devices and how they can interact in heterogeneous architectures. Afterwards, a brief introduction into shallow water equations is given.

2.1.1 Heterogeneous Architectures

Comparison of Computing Devices

Name	# of Cores	# of Threads	Frequency (Base)
Intel Xeon Platinum	16	32	2200 MHz
NVIDIA Tesla P100	3584	3584	1480 MHz

Table 2.1: Comparison of computing devices for HPC [8], [9].

Central processing units are the most common type of computing devices. They are designed to be versatile and flexible in contrast to the graphics processing unit that has a more specialized instruction set. Low latency guarantees interactivity, but the low number of cores limits the parallel handling of tasks. Therefore the general architecture is mostly designed for serial processing [10].

On the contrary, GPUs are specialized in parallel data processing rather than control and storage. As seen in 2.1, blocks of cores share the same cache. This is important when mapping the work to the cores in order to utilize cache benefits due to the fact that many cores work on the same data. Intelligent work-mapping could increase performance, because data access will be accelerated. A NVIDIA GPU, for example, consists of streaming multiprocessors with private caches (L1) of high bandwidth and many CUDA (Compute Unified Device Architecture) cores per streaming multiprocessor which can perform integer and floating point arithmetic. Each streaming processor can access one cache (L2) that is connected to the DRAM and has an interface (normally PCI-Express) to the host system to communicate with the CPU. With the CUDA programming interface stream processing can be realized for applications in high level

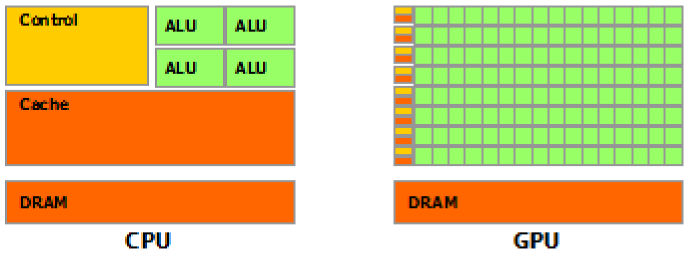

Figure 2.1: Schematic comparison of machine models [13, p. 2].

programming languages like C++. It is done using the API which handles memory management, creates device functions that may get executed by threads and groups threads to blocks which execute the same code in parallel. These global functions can be invoked by the host [11, pp. 600–602], [12].

Connection and Communication

As seen in the previous sections, the architectures and models differ in various ways. One major obstacle is that the memories of the separate devices need to be handled individually, because their (virtual) address space is eventually not shared. Data has to be transferred manually to the respective device which may cause a massive overhead in communication. This issue can also lead to bottlenecks if the interfaces are either too small, the bandwidth is too low or many devices share the same bus. The last aspect is especially problematic if many computational units are involved. The classification of memory can be divided into three types [11]:

- UMA (Uniform Memory Access): The memory access time is equal for each processor.
- NUMA (NonUniform Memory Access): The memory is distinguished in local and peripheral memory. The CPU has a memory module (local), but can access memory of other processors (peripheral).
- COMA (Cache Only Memory Access)

The interconnection network of the multiprocessors is of importance for the communication. The transmission capacity measures the amount of data that the network can transport per second and the bisection bandwidth is calculated by dividing the network in two halves and removing a connecting edge. Afterwards, the new bandwidth needs to be calculated. The bisection bandwidth is the minimum of all combinations. The goal is to maximize bisection bandwidth which is the most important metric, because in the worst case it is the minimum bandwidth. There are several topologies for connecting the processors like star, ring and full interconnection [11, pp. 634–636].

A parallel program which is deployed on many processes and uses distributed memory needs to communicate at some point in execution. Therefore the Message Passing Interface (MPI) is introduced as an abstraction for inter-process communication. Data can be exchanged by sending messages to other processes. The interface provides broadcast and end-to-end communication as well as asynchronous and synchronous operations. Each process knows its rank and the total amount of launched ranks. That enables the programmer to write one program and change the flow of the program depending on this information. There are several other operations provided by the interface, but the ones mentioned are sufficient for the purpose of the project [14].

2.1.2 Shallow Water Equations

Riemann problem and basic equations

The Shallow Water Equations[1] rely on partial differential equations, especially hyperbolic systems. To solve these equations spatial variables as well as a time-dependent one need to be taken into account. The general linear form (with one spatial dimension) is

$$q_t(x,t) + Aq_x(x,t) = 0 \tag{2.1}$$

in which $q \in \mathbb{R}^m$ is a vector of unknown functions and the index represents the partial derivative. The matrix $A \in \mathbb{R}^{m \times m}$ has to meet the following criteria to be hyperbolic:

- Real eigenvalues
- Linear independent eigenvectors

The class of the equations is named conservation laws and has the following simple quasilinear form

$$q_t + f'(q)q_x = 0 \tag{2.2}$$

in which $f(q)$ is called flux function and the conserved quantities are the components of the vector q [15].

In order to approximate the solution the finite volume method is introduced. Finite differences are often used to compute derivatives. The interval is split into cells where q is averaged in the cell. The Riemann problem is a fundamental aspect in solving the

[1]The name will further be abbreviated as SWE

SWE with the finite volume method. q is defined as

$$q(x,0) = \begin{cases} q_l, & \text{if } x < 0, \\ q_r, & \text{if } x > 0. \end{cases} \tag{2.3}$$

with q_l and q_r as the averages of the respective cells at the discontinuity at $x = 0$. This information is used to update the cells by computing the numerical flux. In order to have a stable solution, the CFL condition needs to be satisfied. It has been discovered by showing that the solution of PDEs with the finite differences schema, converges by refining the grid. That stability constraint is given by $\frac{\Delta t}{\Delta x} \max|\lambda| \leq 1$ in which λ is an element of a previously computed set of wave speeds. There are various approaches, like the Roe method, to solve the Riemann problem and these kinds of PDEs, but this section only intends to briefly introduce this topic to understand the application's context [15], [16].

The SWE are defined by following equations:

- $h_t + (hu)_x = 0$ in which h denotes the depth of the fluid and hu the momentum. The momentum is the flow rate and called discharge.
- $(hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x = 0$ is deduced by putting the hydrostatic law $\frac{1}{2}\rho gh^2$ into the conservation of momentum equation $(\rho hu)_t + (\rho hu^2 + p)_x$. Density ρ is canceled out [15, p. 254].

The combination of these equations yields, according to [15, p. 254], the SWE in one dimension

$$\begin{bmatrix} h \\ hu \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}_x = 0 \tag{2.4}$$

Dimensional splitting

In order to compute the numerical solutions for multidimensional systems, *dimensional splitting* is used to reduce the problem on the one-dimensional case. This is done by splitting the spatial dimensions and aligning the results to the coordinate axis. The two-dimensional case (which is the relevant case in the simulation) $q_t + Aq_x + Bq_y$ is split into *x-sweeps* $q_t + Aq_x$ and *y-sweeps* $q_t + Bq_y$. The shallow water equations are generalized from the one-dimensional case and defined in equation 2.5 [15, pp. 429 – 444].

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = 0 \tag{2.5}$$

In the first step, the one-dimensional case is solved with the respective cell information in x-direction for each y. The equation 2.6 is the update of the cell on a grid. F denotes the numerical flux of a cell and Q the cell average [15, pp. 429 – 444].

$$Q^*_{ij} = Q^n_{ij} - \frac{\Delta t}{\Delta x}(F^n_{i+1/2,j} - F^n_{i-1/2,j}) \tag{2.6}$$

The y-sweep makes use of the intermediate values Q^* to compute the final cell update Q^{n+1} with G^* as numerical flux in y-direction [15, pp. 429 – 444].

$$Q^{n+1}_{ij} = Q^*_{ij} - \frac{\Delta t}{\Delta x}(G^*_{i,j+1/2} - G^*_{i,j-1/2}) \tag{2.7}$$

2.2 SWE Project

The application used to explore and test the capabilities of Kokkos is based on a project of the chair. It is called SWE and already implements basic features for solving shallow water equations. Firstly, SConstruct is the build system that needs to be modified in order to set flags or include libraries (in our case Kokkos). SWE simulates two-dimensional domains and therefore uses a submodule which implements one-dimensional solvers. The FWave and AugRie solver are both implementing the Riemann method of solving PDEs. The Kokkos approach uses the HLLE solver. The mathematics are described in 2.1.2.

Secondly, various methods for the two-dimensional simulations are located in the directory "/src/blocks". The method of dimensional splitting will be enhanced with the Kokkos framework and stored in a separate source file. Basically, this methods applies the solver on two spatial dimensions by splitting the problem in x and y direction. The solver is applied for each cell in one direction and after that in the other direction (so-called sweeps) on the discretized grid (2.1.2).

The rest of the files are different scenarios on which the simulation is applied to. The writer source files are used to write checkpoints, which can be used to restore the simulation.

2.3 Kokkos

The Kokkos framework is a library for C++ or Fortran applications developed at Sandia National Laboratories. Its main goal is to provide an ecosystem to generate performance portable code for various scientific applications. As seen in 2.1.1, it can be arbitrarily complex to fully utilize the whole system performance on architectures with different kinds of GPU(s) and CPU(s). To achieve high performance, Kokkos leverages architecture-specific characteristics, e.g. data layout. Kokkos' purpose is to tackle this issue and provide an abstraction layer to the underlying (heterogeneous) architecture. Therefore Kokkos provides different patterns for (parallel) code execution, data management and performance portable data access. Furthermore hierarchical threading and atomicity is addressed [2].

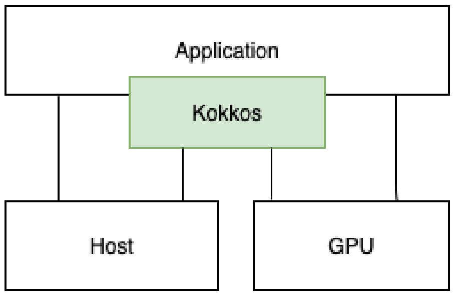

Figure 2.2: Kokkos as a uniform adapter for computational devices.

The code samples, 2.3 and 2.4 shown below, illustrate the difference in the implementation of a matrix multiplication performed in CUDA and OpenMP. This part will not give a deep insight into CUDA programming, but in addition to the kernel, memory allocation, clean up and kernel calls are needed [12]. OpenMP also allows to use the GPU for multithreading and has several more features which are not covered in this thesis. For demonstration purposes only a simplified, small subset of the specification is shown [17].

```
struct {
    int width;
    int height;
    float *elements;
} Matrix;
__global__ void matMul(Matrix A, Matrix B, Matrix C) {
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = 0; i < A.width; ++i) {
        Cvalue += A.elements[row * A.width + i]
                * B.elements[i * B.width + col];
    }
    C.elements[row * C.width + col] = Cvalue;
}
```

Figure 2.3: CUDA kernel of a matrix multiplication. Kernel is executed on GPU cores. Code adapted from [13, p. 24].

```
void matMul(Matrix A, Matrix B, Matrix C) {
  #pragma omp parallel for
  for (int row = 0; row < A.height; row++) {
    for (int col = 0; col < A.width; col++) {
        float Cvalue = 0;
        for (int i = 0; i < A.width; i++) {
            Cvalue += A.elements[row * A.width + i]
                * B.elements[i * B.width + col];
        }
        C.elements[row * C.width + col] = Cvalue;
    }
  }
```

Figure 2.4: OpenMP implementation of a matrix multiplication. Actions are performed on the host [17].

```
// Declaration of a matrices
Kokkos::View<float**> a("matA", hA, wA);
Kokkos::View<float**> b("matB", hB, wB);
Kokkos::View<float**> c("matC", hC, wC);

// Multidimensional iteration
auto range_policy = Kokkos::MDRangePolicy<Kokkos::Rank<2>>({0, 0},
                                                          {hC, wC});

// Perform matrix multiplication
Kokkos::parallel_for(range_policy, [=] (const int row, const int col) {
    float Cvalue = 0;
    for (int i = 0; i < wA; i++) {
        Cvalue += a(row, i) * b(i, col)
    }
    c(row, col) = Cvalue;
});
```

Figure 2.5: Implementation of a matrix multiplication performed by Kokkos [2].

One can clearly see that there are significant differences in the approaches. OpenMP uses preprocessor directives that automatically parallelize given code for the current architecture depending on the setup. The code is then run on the host if the target is not specified to offload on the GPU [17]. On the contrary, in the CUDA programming model, the host needs to initialize the data, transfer it to the device with dedicated instructions, execute the kernels and then fetch the results from the device memory [13]. Kokkos provides a much more convenient interface to unify this disparity, visualized in figure 2.2, while still having the freedom to write host and device functions. This is especially interesting in terms of software engineering, because the application programmer is shielded from hardware details and even the APIs of different parallel programming models. The target offloading happens at compile time by either choosing the kernel execution space or using the default execution space which is much more fault-tolerant, because the source code does not need to get changed.

The computation gets mapped on cores by providing a range which represents the amount of work, the index on which the core operates and the function that gets computed by the unit. The function gets passed as a functor object. Kokkos provides the function *parallel_for* which gets passed the total amount of iterations or a range policy as its first argument and a functor as its last argument. The functor can be expressed as a lambda-function to be more concise. It is important to capture by value, but one needs to be careful to not copy large data containers. The lambda gets the current index / indices as parameter. If not only work needs to be performed on data, but also the data is reduced to a single value, Kokkos provides *parallel_reduce*. Now the lambda has an additional parameter as accumulator (thread-private) and the *parallel_reduce*-function has a reference to the total reduced value as its last parameter. The assigned values for the thread-private references can be reduced afterwards according to a Kokkos or user defined reduction function. Figure 2.5 demonstrates the structure of the *parallel_for*-function [2].

Another important concept are views for handling multidimensional arrays, because the layout for arrays may differ for different devices. Accessing the view follows a common design pattern, which is used in numerical libraries. Kokkos provides many classes for different implementations of the memory space, in order to manage and boost performance for devices with differing memory performance attributes, like texture caches in GPUs. Deep copies of views into other memory spaces can be performed, which is constraint by the *SpaceAccessability*-struct. The abstraction of execution and memory space leads to varying configurations for the computation and storage in heterogeneous architectures. Table 2.2 shows the relevant execution and memory spaces that were used in the project. Kokkos provides several more containers, like the *DualView* which provides a host mirror if the view is not located in host memory space [2].

Execution Space	Memory Space
Cuda	CudaSpace
OpenMP	HostSpace
PThread	CudaHostPinnedSpace
Serial	CudaUVMSpace

Table 2.2: Kokkos distinguishes between several execution and memory spaces. The table shows the relevant spaces for the project [2].

With Kokkos it is also possible to explicitly build hierarchical parallel execution patterns. Hierarchical parallelism can be useful for machines with many computing devices, where computing cores can be grouped and have access to a fast shared memory. These characteristics can be leveraged by Kokkos. A node based cluster with multicore CPUs, for example, supporting hyperthreads and vector instructions has 4 levels of parallelism. There are different types of abstraction levels that can be mapped by Kokkos on specific hardware features. Therefore the concept of thread teams is used which logically (or even physically) groups threads that are synchronized and share memory. The threads are indexed by the league rank, an arbitrary integer, and the team rank which is constraint by the hardware. New teams are launched once a team is finished. Policies are used to instantiate teams by providing the league size and team size, which can automatically be determined with *AUTO*. The respective lambda gets the team member handle. In the kernel function, team members can access the same scratch memory, allocate space in the memory and generally perform the same actions as on the global memory. After the team is finished, the scratch memory gets cleaned up. Loops can be nested and each level can access the ranks. The execution policy *TeamThreadRange* and *ThreadVectorRange* can be used for team based nested parallelism over specific ranges. *PerTeam* and *PerThread* allow atomic operations in *single* nested regions [2].

This chapter is only supposed to give a brief introduction to the capabilities of Kokkos as there are many more functionalities. The parallelization back-end can be selected when building the project and there are many additional options. For further information the wiki of the project [18] can be consulted.

3 Implementation

This chapter will explain the integration of Kokkos in the SWE environment, the design decisions and the approaches used.

3.1 Basic classes

At first the *SWE_DimensionalSplittingKokkos* class is added which implements the dimensional splitting method described in 2.1.2 with Kokkos. Therefore it inherits from *SWE_Block* and provides the respective interface. As datatype, for the float matrix that is given *SWE_Block* as a template argument, *Float2DKokkos* is chosen which will be described below.

The main function is located in *swe_kokkos.cpp*, respectively *swe_kokkos_mpi.cpp* for the MPI implementation with Kokkos, and reads the command line arguments used for the scenario:

- t: Simulation time in seconds
- n: Snapshots that need to be written
- x: Cells in horizontal direction
- y: Cells in vertical direction
- o: File name of the output
- Optional arguments if a specific scenario should be taken

  b: Bathymetry file

  d: Displacement file

This information is then processed to construct the initial state of the simulation scenario. Before the simulation is set-up, Kokkos needs to be initialized. Then the simulation is computed, files are written and Kokkos gets finalized. The code is put in a different scope to make sure that every object used by Kokkos is destructed before it is finalized. The following code snippet demonstrates these steps:

```
// ...
Kokkos::initialize();
{
  SWE_DimensionalSplittingKokkos simulation(args);
  simulation.init(scenario);
   while (simulationIsNotFinished) {
     // ...
     simulation.computeNumericalFluxes();
     simulation.updateUnknowns();
     // ...
   }
}
Kokkos::finalize();
// ...
```

Figure 3.1: Main sequence of the simulation.

In order to smoothly integrate the view data-structures worked on by Kokkos into the legacy code, the class *Float2DKokkos* is created and used as an adapter. It is required that on the one hand the object can be natively indexed like a simple two-dimensional array and on the other hand accessed like a View. These requirements allow the legacy code to be unchanged, because both accessing variants are pointing on the same memory location, which speeds-up computation time by administrating the memory effectively (no deep copy or synchronization is needed). Figure 3.2 shows the structure of the adapter. The adapter only lives in the host space to ensure that the memory alignment is appropriate to the raw pointer layout.

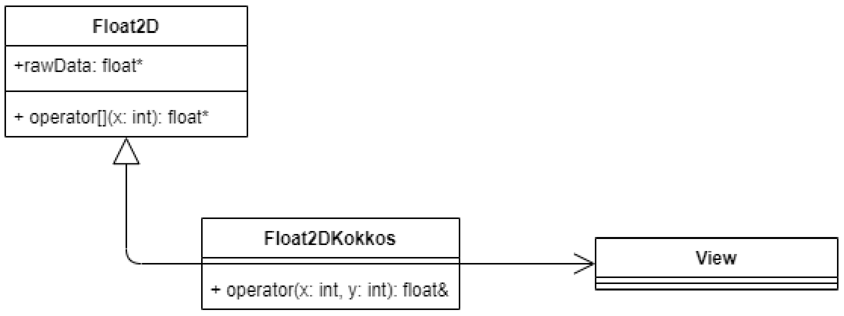
Float2D
+rawData: float*
+ operator[](x: int): float*
Float2DKokkos
+ operator(x: int, y: int): float&
View

Figure 3.2: Structure of the adapter for the access of legacy code.

If CUDA execution space and CUDA unified virtual memory is available, the unified memory is used instead of the host memory. It is advantageous for development and performance, but every aspect needs to be considered. The pivotal advantage is that memory management is getting handled by the graphics device and no additional copy instructions between the different memory spaces are needed in the application. But also caching and data locality benefit from automatic page migration due to the low-latency local memory. This is especially beneficial for the GPUs high bandwidth memory which surpasses 720 GB/s on modern devices. Nevertheless, with virtual memory also comes problematic scenarios that will penalize poor memory management. Page faults may occur when the GPU memory is working at full capacity and therefore this needs to be kept in mind for large scale simulations [19].

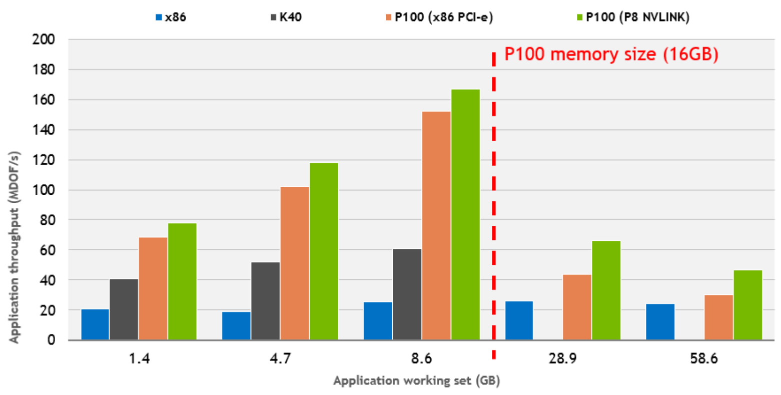

Figure 3.3: Performance comparison of unified memory [19].

One problem that occurs when using CUDA parallelization is that the kernel function is dependent on the solver, which is the vectorized *HLLE* approach and located in the solver-submodule. This does not affect the host back-ends, because the solver is host-accessible and can therefore be invoked inside the kernel. When running the kernel on CUDA back-end, it is problematic to call the host-based solver to compute the net updates, because the kernel gets compiled as a device function and therefore must not invoke the host-based sover-function. This can be avoided by providing the HLLE solver as a device function.

3.2 Compile-Time Choices

In order to reuse existing functionality, several options need to be set at compile-time. Firstly, if the code gets compiled for CUDA devices, the selected solver gets exchanged to a CUDA based HLLE solver. This is only applicable for the flat parallelization approach. This is necessary, because, as mentioned before, the default solver is only usable on the host execution space. *HLLEFunCuda* instead provides the annotations that allow the code to be run on the graphics device and the code can be called from inside the functor. The solver is stateless which is necessary to avoid race conditions or gratuitous synchronization. Only the host-shared data (views of the unknowns) is set to *CudaUVMSpace* or *CudaHostPinnedSpace* so that the relevant data is host-accessible. This is caused by the execution space, because encapsulation leads to host based code which must not be used in a CUDA kernel. For the unknowns that reside in the base class *SWE_Block*, a mirror-like view is provided that is used for the computation. These views need to be synchronized back to the base class when the time-step is written. The CUDA call of other device functions must be protected by the *__CUDA_ARCH__* preprocessor directive definition.

Secondly, if CUDA is not enabled, the execution space is set to the default host execution space if not otherwise specified. The internal views of the unknowns are a shallow copy of the respective base class members. The *HLLEFun* is used as host based solver. It is necessary to use this solver independently of the execution space to evaluate the results appropriately.

3.3 Implementation Approaches

```
template<typename shared_mem_space,
         typename working_mem_space,
         typename kernel_exec_space>
class SWE_DimensionalSplittingKokkos :
         public SWE_Block<Float2DKokkos<shared_mem_space>> {};
```

Figure 3.4: Schematic declaration of the Kokkos flat parallelization class extending the base class SWE_Block.

The main implementation of *SWE_DimensionalSplittingKokkos* adapts the dimensional splitting legacy implementation and uses a flat parallelization (no explicit exploit of hierarchical structures) approach. Kokkos provides different approaches for parallelization like hierarchical parallelization and multidimensional iterations. The hierarchical parallelism approach is only available for non-CUDA computation, because the lambda needs to capture the *this* pointer, which is not yet supported by the NVCC wrapper provided by the development team. The reductions automatically forces a stream synchronization of the device executed kernels when using the CUDA execution space. The class is fully templated as it can be seen in 3.4, so that the user can either set the

- shared memory space: host-accessible memory space that can eventually be accessed by the CUDA execution space (if enabled) or has a deep copy mechanism to fetch the computed unknowns residing in the working memory space,
- working memory space: memory location where the views for temporary results reside and
- kernel execution space: the space where the kernel is run

or use the default values. The template parameters are checked for compatibility at compile-time and a suitable synchronization mechanism between the views is selected. Instead of controlling the used parallelism by the template parameter, the two approaches are put in two different classes, because it makes the code much more clear and lean, which supports maintainability and extendability [2].

3.3.1 Flat Parallelism

The flat parallelism, as mentioned before, only uses streams of depth one. This is useful for simple, non-hierarchical algorithms and less hierarchical architectures. The class definition is depict in figure 3.4.

At first, *parallel_reduce* is used to compute the horizontal net updates (x-sweep). The data is streamed over a multidimensional range and the maximal horizontal wave speed is reduced to compute the maximum time-step, according to the CFL condition, in the next step. The native reduction function *Max<float>(...)*, provided by Kokkos, is used in this case.

Afterwards, the intermediate Q^* states are computed (see 2.6) with Kokkos' *parallel_for*. The y-sweep is then performed analogously to the x-sweep and reduced, with the same *Max<float>(...)* reduction function, on the maximum vertical wave speed. The reduction is not necessarily needed, but the reduced value can be used for debugging purposes. Lastly, the unknowns (height and momenta) are updated according to the Euler method.

These steps are all put in different functions to reuse them when building other communication structures like MPI around it. MPI and Kokkos are discussed later in this chapter.

3.3.2 Hierarchical Parallelism

Hierarchical parallelism can be used for dimensional splitting, because the loops are not tightly nested and each team member accesses data-structures that are contiguous in memory which positively affects the computation time owing to the shared scratch memory. The basic structure is analog to the one used in the flat parallelism approach

```
template<typename shared_mem_space,
         typename working_mem_space,
         typename kernel_exec_space>
class SWE_DimensionalSplittingKokkosHierarchy :
         public SWE_Block<Float2DKokkos<shared_mem_space>> {};
```

Figure 3.5: Schematic declaration of the hierarchy approach class extending the base class SWE_Block.

and as it can be seen in figure 3.5. Generally, for every sweep the league size is equal to the number of cells in x-direction and Kokkos sets the appropriate team-size. That enforces a mapping of the team member rank onto the x-coordinate of the current cell. Edge cases like a low amount of cells in x-direction are not handled, because they do not occur in real simulations as grid-sizes mostly have a rectangular shape. *league_rank()* returns the current x-index (team member rank) of the thread team. The nested reduction gets the team member handle and the iteration range. Afterwards, the net updates get automatically distributed over the threads of the team. Computing the intermediary Q^* states and the y-sweep is similar to the x-sweep except the ranges get adjusted accordingly. Lastly, the unknowns get updated using *parallel_for* with the same policy.

3.3.3 MPI and Kokkos

```
template<typename shared_mem_space,
         typename working_mem_space,
         typename kernel_exec_space>
class SWE_DimensionalSplittingMpiKokkos :
      public SWE_DimensionalSplittingKokkos<shared_mem_space,
                                            working_mem_space,
                                            kernel_exec_space> {};
```

Figure 3.6: Schematic declaration of the MPI implementation extending the base class SWE_DimensionalSplittingKokkos.

When using MPI and Kokkos, MPI has to be set-up before Kokkos is initialized. In order to use both in the context of the project, a class *SWE_DimensionalSplittingMpiKokkos* is implemented. It extends the basic implementation of *SWE_DimensionalSplittingKokkos* as illustrated in 3.6 and reuses the methods (x-sweep, y-sweep, computeMaxTimeStep and updateUnknowns) used for the computation of the simulation. That allows the simulation to use MPI communication with the already implemented Kokkos based computation. Kokkos can not scale one kernel over multiple GPUs and therefore MPI needs to be used to fully utilize the system hardware. That allows the simulation to use multiple GPUs that can even be of different generations. Due to the fact that after each iteration all MPI processes have to be synchronized to exchange data, the computing power needs to be similar, otherwise the more powerful GPUs get blocked by the MPI barrier. When using CUDA, the ghost cells of the block need to be synchronized with the host to exchange data between neighboring ranks, because the data is located in the CUDA memory space, which is not host-accessible. It is implemented by using the *deep_copy* function to copy the data to the host-accessible data layer before the data gets send to the neighbor ranks. After receiving the updates of the other ranks, the affected cells get copied back to the CUDA memory space. In order to save time, when transferring the data from device memory to host memory, one sub-view gets created for each edge, which is a subset of the original view and only the connected edges perform a deep copy. The flow of the main loop for every iteration is schematically illustrated in the activity diagram 3.7. The manual memory synchronization is not optimal yet and the *DualView* can help achieving a more performant implementation. Therefore the *Float2DKokkos* structure has to be changed and the usage of this view has to be investigated. The shared memory space must not be the *CudaUVMSpace*, because several problems occur and the simulation gets corrupted [2].

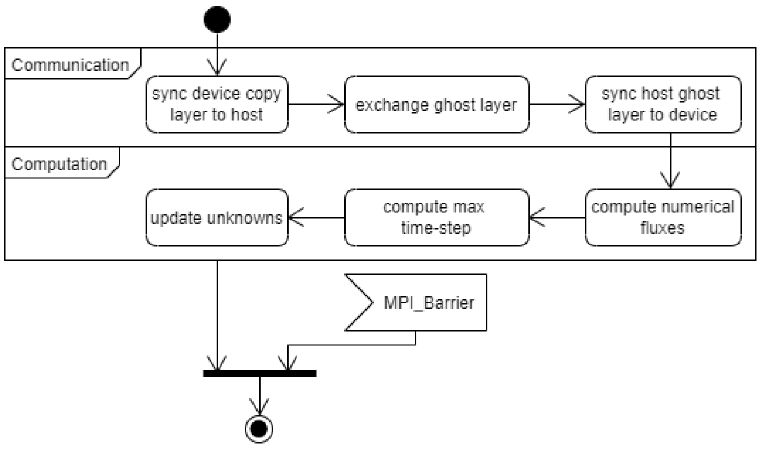

Figure 3.7: Actions are performed in every iteration of the simulation. Communication and computation can clearly be separated. If the relevant data is already host-accessible, the synchronization is skipped.

4 Evaluation

In this chapter the previous implementations of Kokkos in the SWE environment will be evaluated. Therefore different setups of machines, Kokkos settings and scenerios are used for testing. Host-based implementation gets compiled with the Intel compiler version 19.0 and the CUDA implementation with the NVCC wrapper provided by Kokkos.

For evaluating the results, different types of configurations ought to be used to demonstrate performance portability and the leverage of hardware features. The used machines have the following specifications:

	ATSCCS	CoolMUC-3
OS	Ubuntu 18.04	SUSE Linux Enterprise Server 12
Processor	Intel Core i7 3th Gen	Intel Xeon Phi (Knights Landing)
GPU	NVIDIA Quadro P400	
	NVIDIA Titan Xp	
Memory	15 GiB	96 GB DDR4 80.8 GB/s per node
		16 GB HBM 460 GB/s per node

Table 4.1: Machine specifications [20].

	Quadro P400	Titan Xp	Core i7 3th Gen	Xeon Phi (KNL)
Cores	1792	3840	4	64
Threads	1792	3840	8	256
Nom. freq.	1.202 GHz	1.405 GHz	3.4 GHz	1.3 GHz
L1 Cache	48 KB per SM	48 KB per SM	32 KB	32 KB per core
L2 Cache	2 MB	3 MB	256 KB	512 KB per core
FLOPS FP32	5.304 TFLOPS	12.15 TFLOPS		

Table 4.2: Detailed information of the computational devices used in the evaluation [20], [21], [22], [23].

4.1 Flat Parallelism

This section compares the legacy execution with OpenMP and CUDA to the respective execution with the Kokkos based implementation. Figure 4.1 shows different computing devices executing the same code. This demonstrates that the kernel can be deployed, without changing the code, on several devices. As seen in the plot, the simulation executes on different platforms with differing architectures and scales linearly with varying time-steps. It is as expected, because with a fixed grid size, the workload in every iteration is kept constant and only the number of iterations is increased.

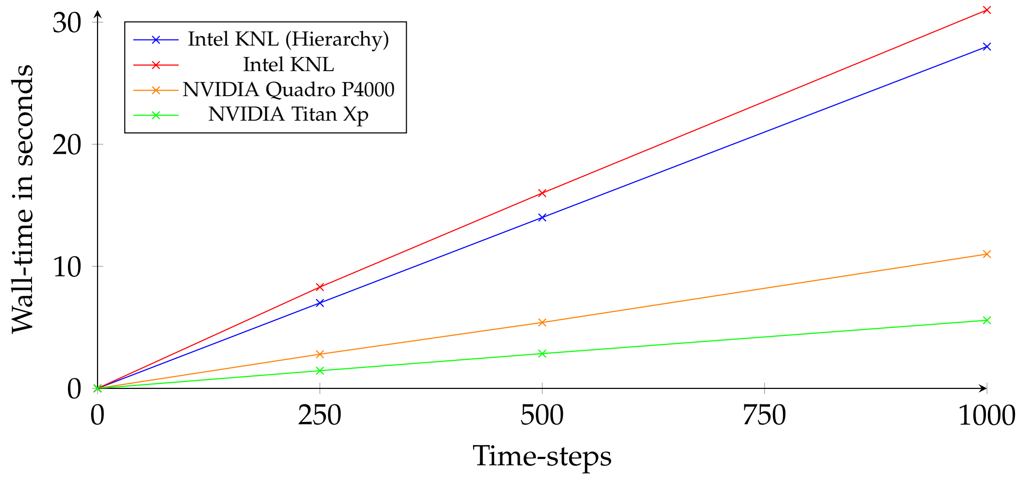

Figure 4.1: Wall-time for different architectures using Kokkos on a 500 x 500 grid

At first the GPU execution model should be investigated to check if the execution model perceived matches the actual execution of the code on the device. The tool *NVIDIA Visual Profiler* shows different profiling data, such as memory management, UVM metrics, GPU meta data, and for each executed kernel the respective execution time, etc. It can be seen that the simulation is executed as expected.

As seen in figure 4.2 it is not applicable to use UVM as shared and working memory space. This leads to massive performance penalities as computation time increases more than four times compared to 4.3 due to page faults and the resulting data transfers. The profilers metric *Thrashing-Throttling* indicates that the same pages are frequently read or written to by CPU and GPU. This could originate in the software design, because the legacy code is also accessing the data from host-side, such as when computing the maximum time-step or even the Kokkos library performs the reduction in host-space. This information can be gathered by investigating the template classes. The thrashing therefore results in a lot of data migration. The gap left and right of the kernel in figure

4.3 is due to the data transfer and the written checkpoint. Both simulations are executed on a 500 × 500 grid with 100 time-steps. The results implicate that the memory space should be chosen explicitly depending on where the data actually resides and is mainly processed.

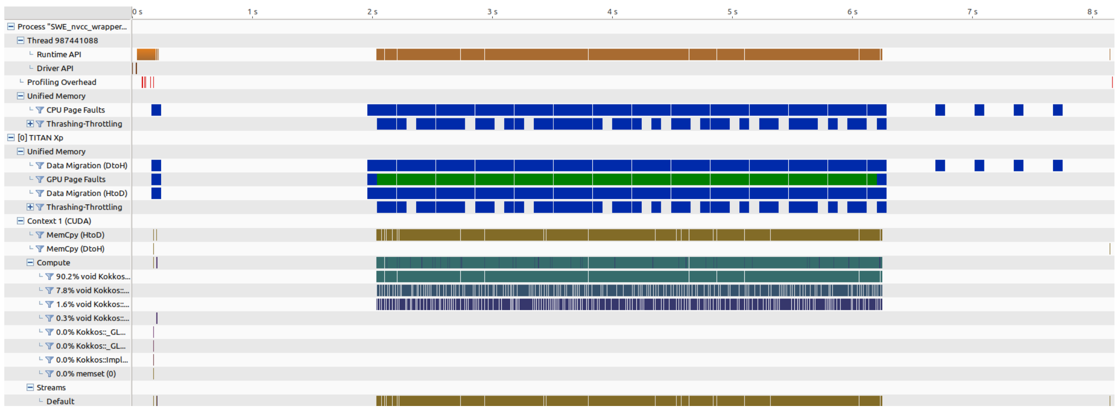

Figure 4.2: UVM as shared and working memory used in the kernels. Wall-time increases due to page faults and data migration.

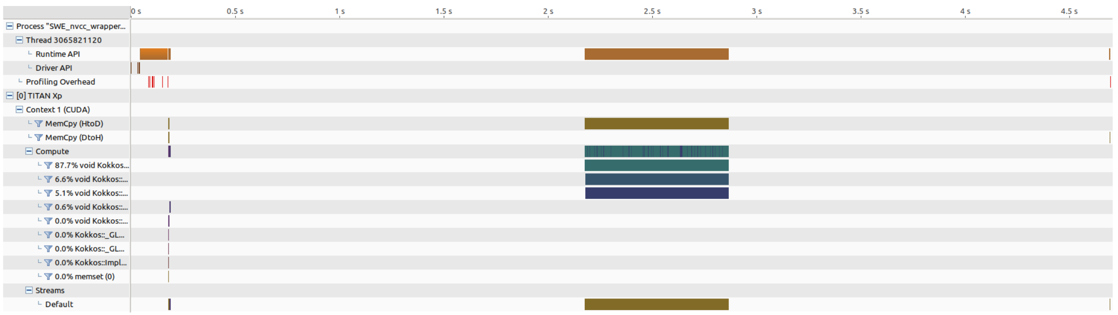

Figure 4.3: Working memory space is set to CUDA space and the host-accessible memory space to CudaHostPinnedSpace. Wall-time is less than in figure 4.2, because the data is only deep-copied before and after the simulation, and no thrashing occurs due to explicitly setting the memory space.

Figure 4.4 demonstrates that the number of cell updates per second increases until the grid size reaches 450 × 450 for both computing devices. After reaching the maximum at approximately 450 cells, the amount of cell updates per second decreases again. This could be due to the loss of cache benefits, because the parameters (six floats) worked on by the solver do not fit into the cache. The Titan Xp, for example, has 48 KB of L1 Cache for each (30 in total) streaming multiprocessor and 3 MB of L2 Cache which results in 4.44 MB of total cache capacity [21]. The amount of data used by the solver can be calculated by multiplying the grid size with the number of parameters times four byte (single-precision floating point): $450 \times 450 \times 6 \times 4Byte = 4.86MB$. It can be seen that the size of the needed memory surpasses the cache capacity and therefore the cache benefits are fully exploited. Increasing the grid size, and therefore the amount of processed data, would lead to more memory traffic and a decrease in performance. The calculation for the KNL processor is analog to the Titan Xp and yields the same result.

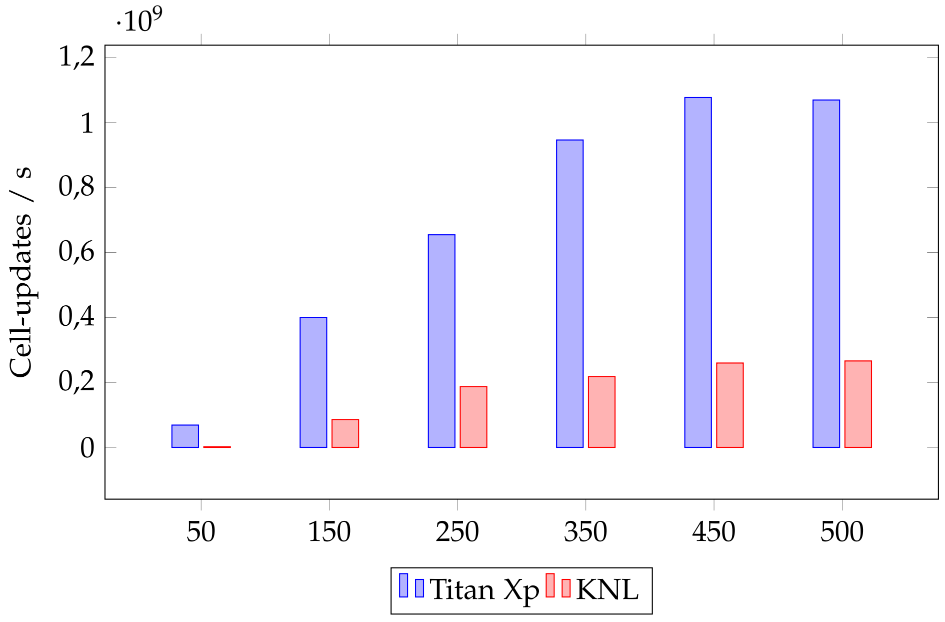

Figure 4.4: Number of cell updates per second using a NVIDIA Titan Xp and a KNL processor. The simulation consists of 15 time-steps on a quadratic grid.

Figure 4.5 shows the speed-up of the simulations compared to the respective legacy implementation. It can be seen that for small grid sizes the Kokkos implementation is faster which could be due to the hardware-awareness of Kokkos and therefore the exploitation of fast memory like (texture) caches. In the interval of approximately 450 to 550 the legacy code is performing better than the Kokkos implementation. This situation might be caused by the already full exploitation of fast caches and a slightly negative influence of the Kokkos framework overhead. Afterwards, the speed-up increases again and peaks at around 1000×1000. For large grid sizes the host implementation converges to a speed-up of around 7 %. The Kokkos overhead does not significantly influence the overall performance anymore due to the increased workload on the computational device and the memory. Therefore the benefits of Kokkos surpass the legacy implementation by a constant factor The GPU speed-up is not displayed for large grid sizes due to the jump of the wall-time for grid sizes beyond 1456×1456.

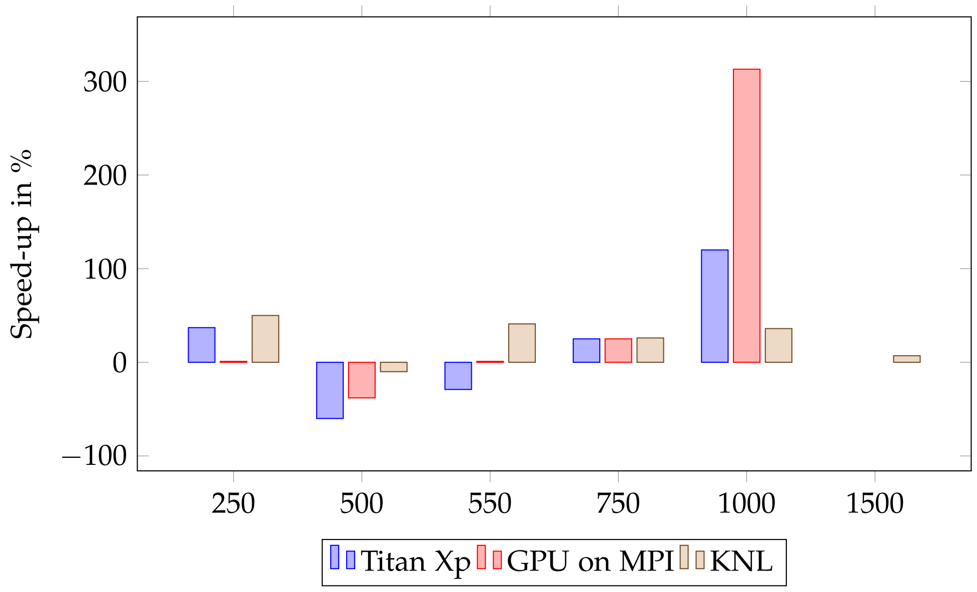

Figure 4.5: Speed-up of Kokkos compared to the respective legacy code as baseline.

4.1.1 OpenMP and Kokkos

Figure 4.6 shows the roofline model of the application executed on 256 threads on one KNL processor. It is an insightful model to measure performance. The arithmetic intensity (AI) is the number of FLOPS divided by the bytes transferred by the program. The model relates the maximum number of FLOPS, the peak memory bandwidth and the arithmetic intensity in order to identify bottlenecks [24, pp. 542 – 547]. As seen in the figure, the program (the red dot) is bounded by the maximum bandwidth. It implicates that the program is not using the full computational power of the system, because it requests data faster than the memory can provide. Nevertheless, the program optimizes data access, because there is only a small space between the DRAM bandwidth bound and the program. Improving data locality and cache usage could possibly lead to a slightly better performance. The AI is less than in the GPU model, because the tool has taken the whole program as reference and not only the solver used for the computation of the net updates.

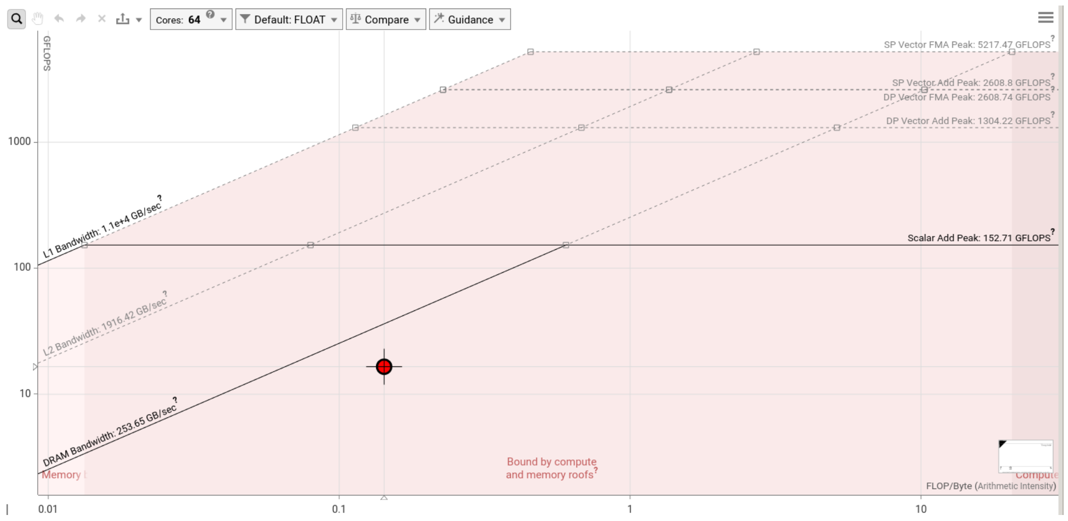

Figure 4.6: Roofline model using one KNL. It is captured by the Intel Advisor XE [25].

Plot 4.7 compares the measured speed-ups, alluding to Amdahls law, of the Kokkos implementation to the legacy results. It is executed on one KNL processor [26]. The difference between the legacy and Kokkos implementation could occur due to a better load balancing on the cores and threads. When building Kokkos, it can be optimized for a specific processor and aggressive vectorization can be enabled, so that it knows the underlying hardware and can therefore leverage performance characteristics. In this case, Kokkos is optimized for vector instructions and the KNL processor which would also explain the difference in the speed-up.

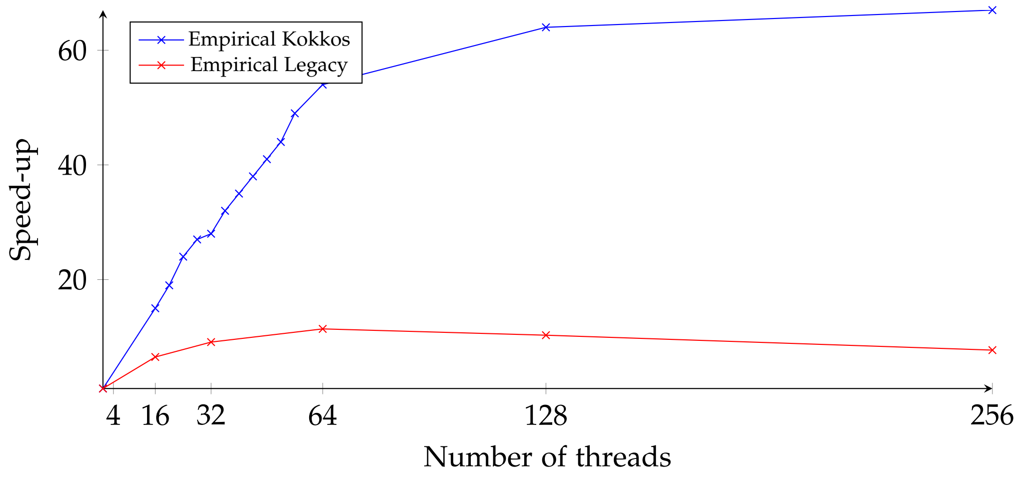

Figure 4.7: Speed-ups of empirical results alluding to Amdahls law [26].

Plots 4.8 and 4.9 show the execution wall-time of the OpenMP legacy code compared to Kokkos with OpenMP back-end enabled. It can clearly be seen that Kokkos is performing better than the legacy implementation. The quadratic behavior of the plot is as expected, because the grid size grows quadratically too. The plots illustrates that the Kokkos implementation is not performing worse than the legacy implementation and, as it can be seen later, is performance portable to other devices / platforms.

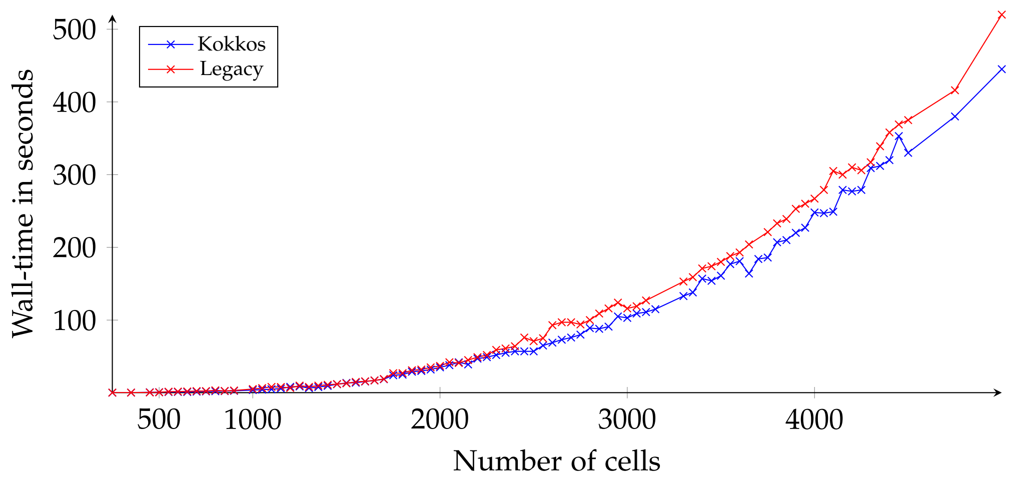

Figure 4.8: Wall-time comparison of legacy and Kokkos implementation on one KNL processor. The simulation consists of 15 time-steps.

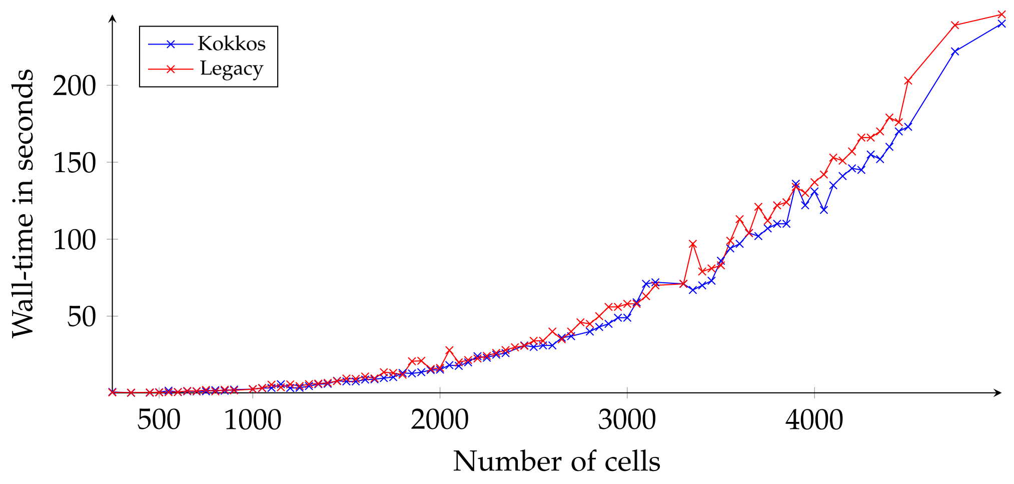

Figure 4.9: Wall-time comparison of legacy and Kokkos implementation on 128 MPI processes on two KNL processors using a quadratic grid. The simulation consists of 15 time-steps.

4.1.2 CUDA and Kokkos

Figure 4.10 depicts the roofline model executed on the Titan Xp. The maximum bandwidth is 416266.1 MB/s, the arithmetic intensity of the kernel is approximately 3.06 and the maximum number of FLOPS performed by the Titan Xp are 12.5 TFLOPS. It can be seen that the kernel is also bound by the bandwidth.

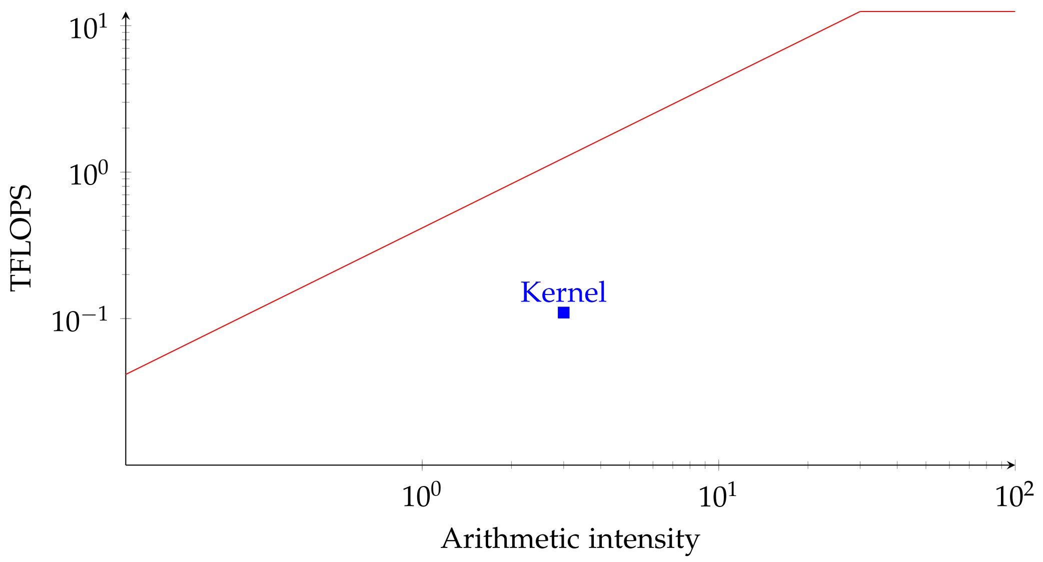

Figure 4.10: Roofline model for the execution of the simulation on the Titan Xp with grid dimension 500 × 500 for 5 time-steps. Arithmetic intensity is 3.06.

The plot 4.11 depicts two different GPU models and their performance for different grid sizes. As seen in the plot, the Titan Xp is around two to three times more powerful than the Quadro P4000. This is due to the amount of FLOPS that can be processed by each GPU. The execution with the MPI version is faster for each rank compared to the single execution. But the wall-time for the simulation is negatively influenced by the less powerful GPU and therefore performs worse than simulating solely on the Titan Xp. The plot shows only the wall-time of the worst rank without including the communication of the MPI ranks. The MPI ranks are balanced in a 2:1 split so that the grid of the Titan Xp is double the size of the Quadro GPU. The legacy code is taken from another repository, which is provided by the chair of scientific computing [27].

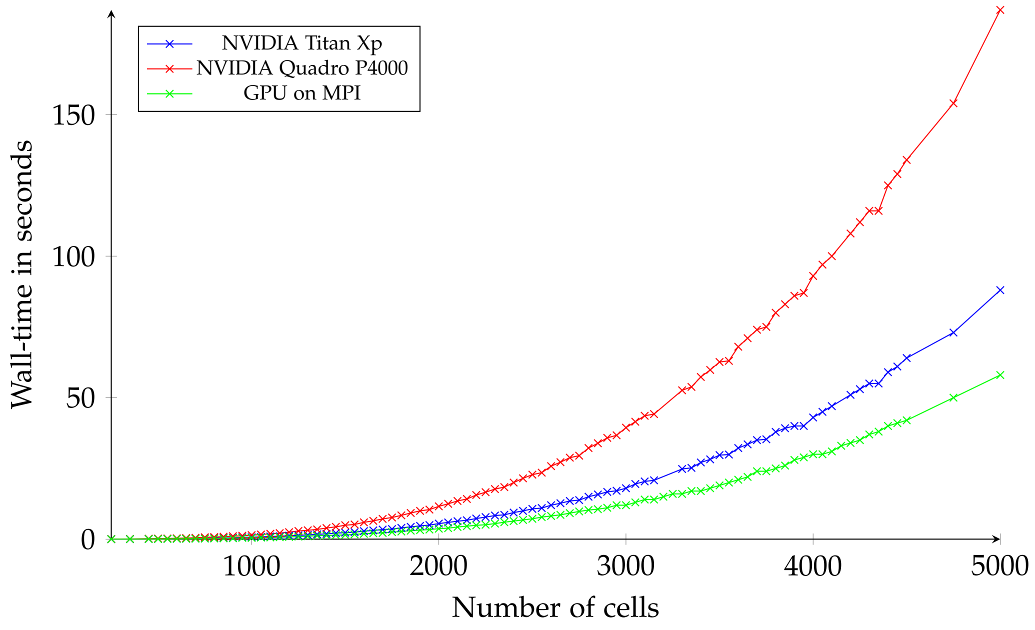

Figure 4.11: Wall-time comparison of different GPU models for different quadratic grids with 15 time-steps. Titan Xp can process twice the amount of FLOPS compared to the Quadro P400 and that reflects in the simulation wall-time. The MPI measurement therefore splits the grid domain in a 2:1 ratio.

Plot 4.12 shows the wall-time of the CUDA implementation for Kokkos compared to the legacy implementation using the CUDA-API. The spatial domains of the legacy code differ from the other evaluations, because the implementation only allows the grid sizes to be multiples of the tiling size which is 16 in this case. On the contrary, Kokkos is much more flexible as it allows arbitrary grid sizes. As seen in 4.12, Kokkos scales according to the expectation whereas the legacy implementation has a jump between the quadratic grid sizes of 1440 and 1456 cells. This is not caused by the kernels which only take approximately 1.4 seconds, but by a blocking CPU.

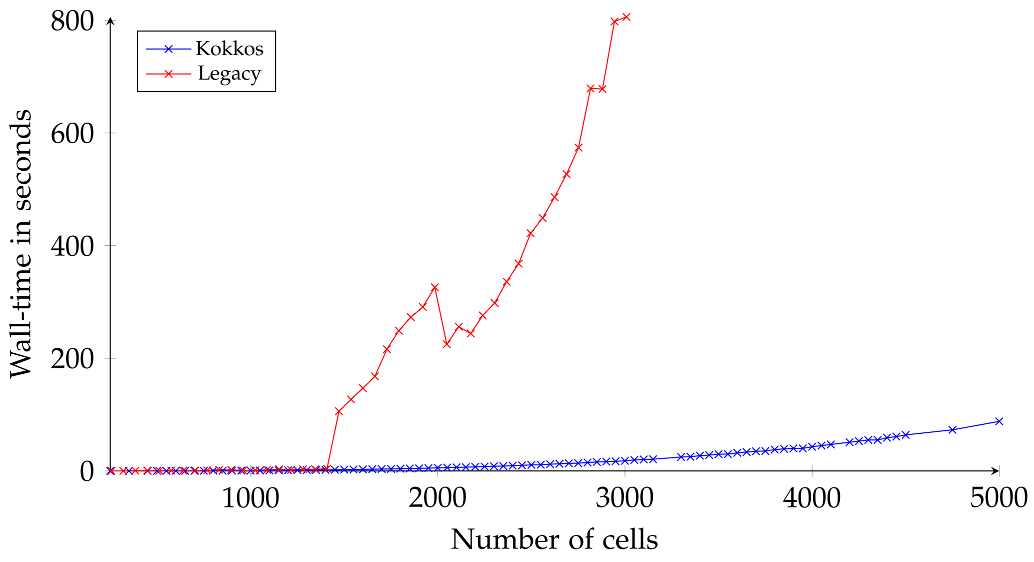

Figure 4.12: CUDA wall-time comparison of Kokkos and the legacy implementation on a quadratic grid with 15 time-steps. Legacy implementation jumps at 1456 due to a blocking CPU not of a extended kernel execution time.

Due to fact that each core has to be mapped onto several cells and the number of cells increases quadratic in each step, the wall-time is suggested to increase by $\theta(cell\_count^2)$. This assumption seems to be confirmed by the measurements taken and visualized in the respective plots.

4.2 Hierarchical Parallelism

The hierarchical parallelism approach is only available for host execution spaces. Therefore the evaluation only compares the wall-time of the hierarchical approach to the non-hierarchical Kokkos implementation and the legacy code. As seen in 4.13 and 4.14, the hierarchical approach performs slightly better than the flat version. This might be due to the fact that the KNL processor is build upon the hierarchical paradigm described in the Kokkos section. The implementation explicitly exploits vector instructions, because the computation of the Q^* states and the update of the unknowns are vectorized with *ThreadVectorRange*. The results would suggest to prefer the hierarchical implementation, but on the contrary it is less portable to other devices in the context of the project due to legacy code support and the missing class member capturing.

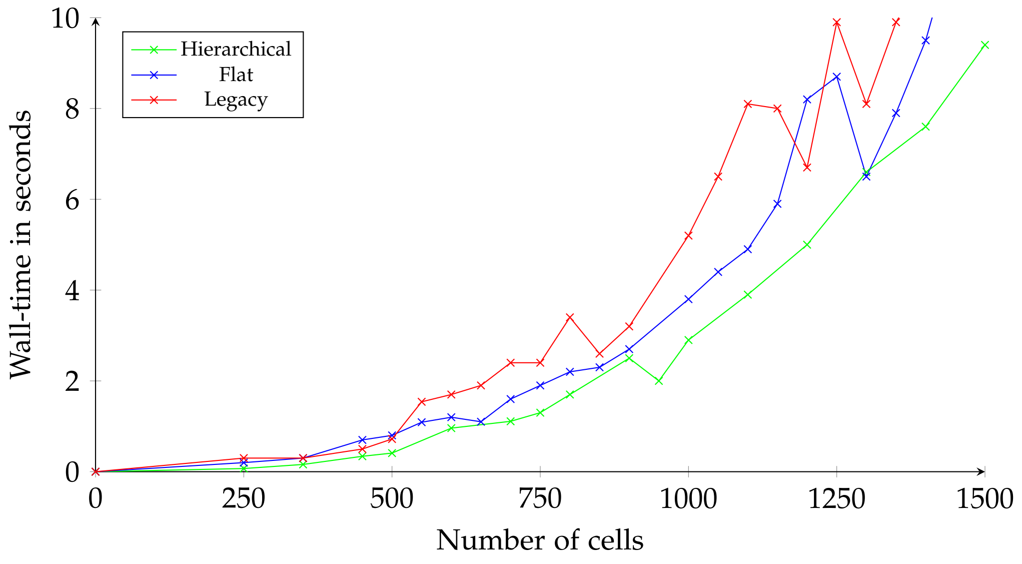

Figure 4.13: Wall-time comparison of legacy, Kokkos and hierarchical implementation on 64 threads on a KNL processor using a quadratic grid. The simulation consists of 15 time-steps.

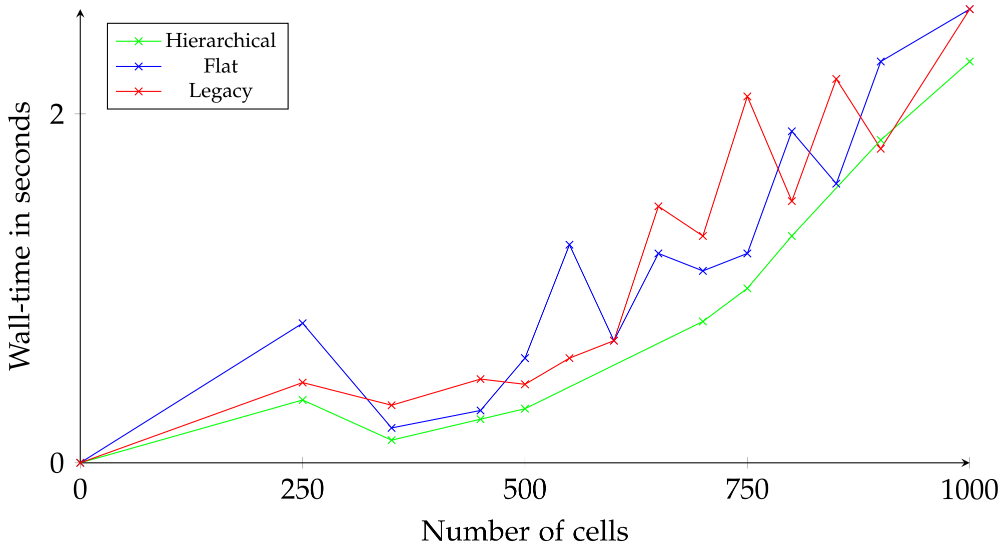

Figure 4.14: Wall-time comparison of legacy, Kokkos and hierarchical implementation on 128 threads on two KNL processors using a quadratic grid. The simulation consists of 15 time-steps.

5 Conclusion

In this thesis the Kokkos framework was explored in the context of shallow water equations. The implementation demonstrated that the usage of Kokkos makes it easy to deploy the code on several devices, while not having any performance losses. Even the support, due to the seamless integration into the given project structure, of the legacy project was guaranteed and only small additions, such as the *Float2DKokkos*-adapter, had to be supplemented.

The evaluation showed the performance portability for different devices and platforms. It advocates the use of Kokkos in scientific applications, as code can be written once, but run on several devices. Not only is it advantageous in terms of software engineering aspects, but it also lets the application programmer focus on the problem instead of hardware details.

The development with Kokkos has several advantages, such as a clear and concise documentation of the core features of the framework, an easy-to-use API and continuous updates of the project repository. But there are also some disadvantages, especially when using CUDA or more recent developed features. The CUDA development was difficult, because a lot of information had to be collected by checking the issue list on the project repository. Particularly, the missing support for class member capturing when using the NVCC wrapper, which is necessary when compiling for CUDA, blocked the development of the hierarchical approach for CUDA execution space. Additionally to that, the debugging of the project was often complicated due to the heavy usage of templates by Kokkos.

Complementary to the CUDA implementation the use of the ROCm platform can be investigated and compared to the different back-ends provided by Kokkos. This is particularly interesting, because it allows a comparison of different vendors and this can point out which GPU performs better depending on the overall configuration, use-case and price-performance ratio. Furthermore, it should be evaluated how the use of performance homogeneous GPUs impacts the speed-up.

Another important step for the project is to split the computation and communication part. That allows integrating new frameworks in the projects and reusing already implemented communication / computation classes. Therefore it has to be investigated what the differences are and one general concept has to be consolidated. The possible structure of the reengineered software model could possibly look like shown in figure

5.1. After reengineering it is easy to add new methods by just extending the respective abstract class.

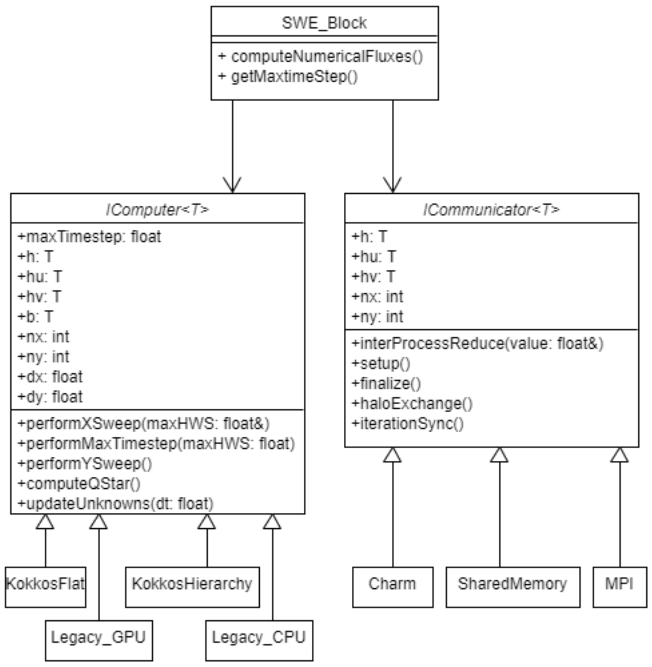

Figure 5.1: SWE reengineered class model

After all, Kokkos' convenient interface advocates it as a framework for parallel computing. The evaluation confirms this statement as already illustrated in the measurements.

List of Figures

List of Tables

Bibliography

[1] P. Czarnul, *Parallel Programming for Modern High Performance Computing Systems*. Boca Raton, Fla: CRC Press, 2018, ISBN: 978-1-138-30595-3.

[2] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2014.07.003`.

[3] P. C. Steve Plimpton and C. Trott. (2020). miniMD, [Online]. Available: `https://github.com/mantevo/minimd` (visited on 05/13/2020).

[4] Kokkos. (2020). Applications, [Online]. Available: `https://kokkos.org/applications/` (visited on 05/14/2020).

[5] M. Bader, A. Breuer, W. Hölzl, and S. Rettenberger, "Vectorization of an augmented riemann solver for the shallow water equations," in *Proceedings of the 2014 International Conference on High Performance Computing and Simulation (HPCS 2014)*, W. W. Smari and V. Zeljkovic, Eds., IEEE, 2014, pp. 193–201.

[6] J. Olden, "Performance Analysis of SWE Implementations based on modern parallel Runtime Systems," Bachelor's Thesis, Technical University of Munich, 2018.

[7] M. Bogusz, "Exploring Modern Runtime Systems for the SWE Framework," Bachelor's Thesis, Technical University of Munich, 2019.

[8] "Whitepaper: NVIDIA Tesla P100," NVIDIA, Tech. Rep. WP-08019-001_v01.1.

[9] Intel. (2019). Intel® Xeon® Platinum 8253 Processor, [Online]. Available: `https://www.intel.com/content/www/us/en/products/processors/xeon/scalable/platinum-processors/platinum-8253.html/` (visited on 05/12/2020).

[10] B. Caulfield. (2009). What's the Difference Between a CPU and a GPU? [Online]. Available: `https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/` (visited on 05/12/2020).

[11] A. S. Tanenbaum and T. Austin, *Rechnerarchitektur : Von der Digitalen Logik Zum Parallelrechner*. Cambridge: Pearson Education, 2014.

[12] M. Harris. (2012). An Easy Introduction to CUDA C and C++, [Online]. Available: `https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/` (visited on 05/12/2020).

[13] N. Corporation. (2019). CUDA C++ Programming Guide, [Online]. Available: `https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf` (visited on 05/13/2020).

[14] T. Rauber and G. Rünger, "Message-passing-programmierung," in *Parallele Programmierung*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 207–273, ISBN: 978-3-540-46548-5. DOI: `10.1007/978-3-540-46548-5_5`.

[15] R. J. LeVeque and L. R. J., *Finite Volume Methods for Hyperbolic Problems*. Cambridge: Cambridge University Press, 2002, ISBN: 978-0-521-00924-9.

[16] E. F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics - A Practical Introduction*. Berlin Heidelberg: Springer Science & Business Media, 2009, ISBN: 978-3-540-49834-6.

[17] *OpenMP Application Programming Interface*, version 5.0, Nov. 2018.

[18] H. C. Edwards, C. R. Trott, and D. Sunderland. (2020). Kokkos Wiki, [Online]. Available: `https://github.com/kokkos/kokkos/wiki` (visited on 05/15/2020).

[19] N. Sakharnykh. (2016). Beyond GPU Memory Limits with Unified Memory on Pascal, [Online]. Available: `https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/` (visited on 05/12/2020).

[20] LRZ. (2020). CoolMUC-3, [Online]. Available: `https://doku.lrz.de/display/PUBLIC/CoolMUC-3` (visited on 05/15/2020).

[21] TECHPOWERUP. (2020). NVIDIA TITAN Xp, [Online]. Available: `https://www.techpowerup.com/gpu-specs/titan-xp.c2948` (visited on 05/15/2020).

[22] ——, (2020). NVIDIA Quadro P4000, [Online]. Available: `https://www.techpowerup.com/gpu-specs/quadro-p4000.c2930` (visited on 05/15/2020).

[23] ——, (2020). Intel Xeon Phi 7210F, [Online]. Available: `https://www.techpowerup.com/cpu-specs/xeon-phi-7210f.c2042` (visited on 05/15/2020).

[24] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design - The Hardware/software Interface*. San Francisco, Calif: Morgan Kaufmann, 2013, ISBN: 978-0-124-07726-3.

[25] Intel. (2019). Intel® Advisor Tutorial for Using the Automated Roofline Chart to Make Optimization Decisions, [Online]. Available: `https://software.intel.com/content/www/us/en/develop/documentation/advisor-tutorial-roofline/top.html` (visited on 05/13/2020).

[26] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS ’67 (Spring), Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485, ISBN: 9781450378956. DOI: `10.1145/1465482.1465560`.

[27] T. C. of Scientific Computing. (2020). The Shallow Water Equations teaching code, [Online]. Available: `https://github.com/TUM-I5/SWE` (visited on 05/12/2020).