Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Performance portability and evaluation of heterogeneous components of SeisSol targeted to AMD GPUs

Simon Dominick

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Performance portability and evaluation of heterogeneous components of SeisSol targeted to AMD GPUs

Leistungsübertragbarkeit und Beurteilung der verschiedenen Komponenten von SeisSol für AMD Grafikkarten

Author:	Simon Dominick
Supervisor:	Dr. Michael Bader
Advisor:	Ravil Dorozhinskii
Submission Date:	15.04.2021

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

Ich versichere hiermit, dass ich die von mir eingereichte Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Abstract

GPUs (Graphics processing units) are commonly used in high-performance computing to improve the execution time of parallelizable programs. SeisSol, as such a program, can currently only use GPUs manufactured by NVIDIA. In this thesis, we extend the GPU-related libraries to work additionally on GPUs manufactured by AMD. We examine the performance of the extended libraries on the AMD Radeon Instinct MI50 using the Roofline model and compare it to results measured on the NVIDIA Tesla V100. Furthermore, we investigate using multiple AMD GPUs, which communicate via OpenMPI. The Radeon Instinct MI50 achieved a good performance but was slower than the NVIDIA GPU. We detected a problem in the communication between the GPUs, which is restricted to collective communication. This does not affect SeisSol since it uses mainly point-to-point communication.

Contents

1. Introduction

Up until now, GPUs (Graphics processing unit) manufactured by AMD played no significant role in the architecture of high-performance computers. In the list of the 500 most powerful supercomputers published in November 2020 by TOP500, only one system uses AMD GPUs [1]. About 130 other systems incorporating GPUs feature NVIDIA products [1]. But as two new supercomputers featuring AMD GPUs are planned, which are going to achieve more than $1\ EXAFLOP = 10^{18}\ FLOP$ per second [2], namely the Frontier [3] and the El Capitan system [4], GPUs developed by AMD may play a bigger role in the future of high-performance computing. Thus, the ability to run programs using AMD GPUs for accelerating computation opens the possibility to deploy these programs to upcoming systems. As the simulation of natural phenomena is a traditional high-performance computing field, the software package SeisSol represents a candidate for introducing AMD GPU support via the HIP API. The software is used for simulating earthquake dynamics and already capable of using NVIDIA GPUs for accelerated computation. To keep the software package flexible this thesis does not simply port SeisSol to HIP. Instead, the two libraries Device and GemmForge, which provide GPU-related functionality for SeisSol, get extended to support AMD GPUs while retaining compatibility for NVIDIA GPUs.

1.1. Related Work

A directly comparable work does not exist, since this thesis describes the first implementation of SeisSol using AMD GPUs for the computation. But [5] represents the direct predecessor of this thesis. It describes the implementation of the GPU-related features for SeisSol targeting NVIDIA GPUs and both libraries extended in this thesis were created in the context of it. Thus, this work serves as the origin of this thesis. Apart from SeisSol several HPC programs have already been ported for AMD GPUs, like [6], [7], and [8]. Whereas [7] follows a similar approach as this thesis by abstracting the GPU-related API calls with an additional layer, [6] defines the functionality, which can be used for both GPUs in a common folder. The platform-specific implementation is realized in a separate folder for each platform, which interfaces the common folder. For usage of the different platforms, each platform gets explicitly accessed, whereas in our approach an API manages the access to the different platforms.

1.2. Structure

This thesis is structured as followed: In the following chapter 2, background knowledge about GPUs, the software SeisSol, and important libraries used in the thesis gets provided. Chapter 3 introduces the two libraries which provide GPU functionality for SeisSol. Furthermore, performed changes and additions to support AMD GPUs get depicted. These changes get evaluated in chapter 4, where the functionality of both libraries on both platforms gets verified. Additionally, important characteristics of the used AMD GPU for SeisSol get examined via special benchmark tests. The results of those tests get compared to results achieved with a comparable NVIDIA GPU model. Additionally, the usage of multiple GPUs via MPI gets examined. Chapter 5 finalizes the work by concluding the performed changes.

2. Basics

This chapter covers required background knowledge about the used GPUs and APIs provided by the corresponding manufacturer. Furthermore, the theoretical background of the target software SeisSol and two libraries important for multi-node computing get illuminated.

2.1. Hardware

Since a major computational part of SeisSol consists of matrix manipulation, as described later, the ability to simultaneously execute the same operation multiple times is more important than a fast execution of a single operation. This translates into a hardware requirement for the capability of parallelism instead of single-thread performance. This demand is met by GPUs, which contain thousands of moderately fast cores, in contrast to CPUs (Central processing unit), which offer few, fast cores. Since this thesis focuses on porting the GPU-dependent parts of SeisSol to a new manufacturer architecture, namely AMD, there are two manufacturer-specific hardware architectures relevant to the thesis. The next two chapters describe the hardware architectures of the two mainly used GPU models.

2.1.1. NVIDIA

Up until now, SeisSol used NVIDIA GPUs [5] for the computation of the matrix manipulations. Therefore, an NVIDIA GPU is used as a comparison to the new implementation in the thesis. The model used is the NVIDIA Tesla V100 SXM2 16 GB released in 2019, which contains the Volta GV100 architecture. It features 16 GB HBM2 memory with a bandwidth of about 1133 GB/s [9]. The clock speed ranges from 1245 MHz to 1597 MHz if boosted [9]. The Tesla V100 SXM2 models allow a connection between the GPU and other processors with the NVLink technology [10]. NVLink provides higher bandwidth for data exchange between the processors than the expansion bus standard. The Volta architecture supports up to six bidirectional links, where each link delivers up to 25 GB/s [10]. For GPU-to-GPU bidirectional communication up to 300 GB/s by using all six links [10] can be achieved in contrast to 32 GB/s provided by PCIe 3.0 [11]. NVLink also allows the connection of a GPU to a CPU, if the CPU supports NVLink. The achievable bandwidth is dependent on the number of links supported by the CPU. For example, IBMs POWER9 CPU supports up to six links, thus a bidirectional connection with a bandwidth of 300 GB/s to one Volta GPU is possible [12]. The GV100 design features as stated in [10] six GPCs (GPU

Processing Cluster), each of those incorporating seven TPCs (Texture Processing Cluster). A TPC combines two SMs (Streaming Multiprocessor), resulting theoretically in $6\,GCPs * 7\,TPCs * 2\,SMs = 84\,SMs$, but in the Tesla V100, only 80 SMs are activated. One SM houses 64 FP32 cores for single-precision and 32 FP64 cores for double-precision floating-point operations. For integer operations, 64 INT32 cores are included. Furthermore, the architecture has 8 Tensor cores per SM, commonly used for neural networks. A Tensor core computes the operation $D = A * B \ + C$, where A,B,C and D are matrices with a dimension of $4x4$. All the cores of an SM are distributed evenly between four processing blocks. Each block has its warp scheduler, dispatch unit, L0 instruction cache, and a 64 KB-sized register file for storing data needed for the computation. Additionally, each SM has 128 KB of L1 data cache, which can be used for data sharing between the cores, one L1 instruction cache, and four texture units. The 6144 KB-sized L2 cache is accessible by all SMs and used for instructions and data, in contrast to the L0 and L1 caches.

For the execution of instructions, the processing block uses a specialized version of the SIMD (Single instruction multiple data) model [13] called SIMT (Single instruction multiple threads) [10]. The SIMD model describes a computational system where one instruction, for example, add or multiply, gets executed on a set of data. In contrast instructions in multicore CPUs get executed on a single piece of data by the single cores (SISD - single instruction single data), which are the CPU equivalent to processing blocks. The SIMT model refines the SIMD model by executing the instruction with one thread per data piece. In contrast, a vector processor satisfies the SIMD model, but not the SIMT model since a single thread executes the instruction for a vector of data. The following figure depicts these three models:

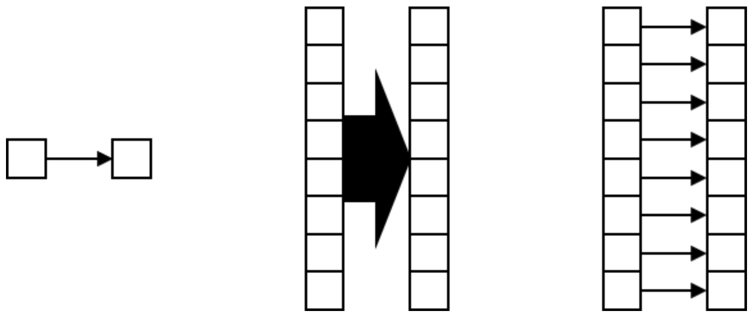

Figure 1: The SISD (left), SIMD (middle) and SIMT (right) model

As described in [10], the GV100 bundles 32 threads into a warp, where every thread executes the same operation simultaneously. To realize branch execution, for example, an if-else statement, each warp has an active mask, determining which thread should be executed. Nonactive threads do not perform any work during the execution step of the warp. Furthermore, the Volta architecture provides a program counter and a call stack for each thread in the warp, in contrast to one program counter and call stack for the

whole warp. This allows for synchronization in smaller than warp thread sizes. Furthermore, it enables a geared execution of branch instructions, meaning that the warp does not have to execute the whole branch at once, but can switch freely between the instructions of multiple branches. Additionally, this technique allows for communication between the threads during branch execution, which would not be possible without the multiple program counters. With only one program counter and call stack, the warp can only synchronize on the warp level, since only the state of the whole warp is known.
Each SM can handle up to 64 active warps, where the warps get divided evenly [14] onto the warp schedulers of the four processing blocks. For the instructions mainly used by SeisSol, namely, float multiplication for the matrices and integer arithmetic for pointers and the control variables of the loops, the Volta architecture needs 4 cycles to complete the execution on a whole warp [14]. Due to the separate INT32 and FP32 cores, a Volta SM can execute integer and floating-point arithmetic in parallel [10]. This allows simultaneous execution of the instructions nested in a loop and updating the control variables for the next step. To hide latency generated by memory transactions and control flow operations, the warp scheduler can choose another warp to be issued to the processing block [15].

2.1.2. AMD

The AMD GPU used for this thesis is the AMD Radeon Instinct MI50 Accelerator released in 2018. It has 32 GB of HBM2 memory width a bandwidth of 1024 GB/s, the clock speed ranges from 1200 up to boosted 1746 MHz [16]. It is connected via PCIe 4.0 x16 lanes, which allow about 64 GB/s [17] bandwidth between GPU and main memory. Additionally, it includes the AMD Infinity Fabric technology, which enables up to 184 GB/s direct GPU to GPU communication in addition to the PCIe connection [17]. The Radeon Instinct MI50 is based on the GCN 5.1 architecture, codenamed Vega. It consists of four shader engines housing 15 CUs (Compute Unit) and one workload manager per shader engine [18]. Each CU contains one SU (Scalar Unit) and 4 VUs (Vector Units) [19]. The SU can execute exactly one integer instruction in SISD fashion, whereas the VUs, which are 16 lanes wide SIMD processing units, can execute integer and floating-point instructions on 16 work items simultaneously [18]. The CU holds 12.5 KB of SGPR (Scalar General-Purpose Register) [18] organized in 32-bit wide entries [19]. For each VU, 64 KB VGPR (Vector General-Purpose Register) are available [18]. Like the SGPR the VGPR is organized in 32-bit entries [19], which can be combined for representing larger values. Furthermore, each CU holds a 16 KB-sized L1 cache, and a 64 KB storage called LDS (Local Data Share) [18]. The LDS is intended for data sharing between VUs [19]. In contrast, the L1 cache is intended for moving data into and computed results out of the CU [19]. The L1 cache of each CU is connected to the 4 MB sized L2 cache [16] shared by all CUs of the GPU and can move data directly between the VGPRs or the LDS and the memory of the GPU [19]. For data sharing between the single CUs, the GPU provides the 64 KB sized GDS (Global Data Share) [19]. For the controlling and distributing of the workload, each CU contains one scheduler [18].

As described in [18], the AMD GPU executes the program in the SIMD fashion, where it groups 64 work items, the AMD terminology for threads, into a wavefront, the AMD equivalent for NVIDIAs warp. Logically all the 64 work items get executed simultaneously on a single VU. Branch divergence is represented by a 64-bit wide execute mask, one bit for each lane, and a 1-bit flag, which indicates if all entries in the mask have the value zero. If the entry in the mask is zero for a certain lane, this lane does not execute any instruction during the execution step of the whole VU [19].

Each VU can hold up to 10 wavefronts in its instruction buffer, making 40 in total for a CU [18]. Since a VU contains only 16 lanes, a wavefront gets executed in a minimum of four steps to complete the instruction for all 64 work items [18]. If a wavefront creates delay, by waiting for a memory transfer, for example, the VU can be switched to another wavefront to hide the latency. If all work items of a wavefront execute the same integer arithmetic instruction, the instruction gets distributed to the SU of the CU. The scheduler of the CU chooses every clock cycle according to the round-robin method another VU to work on [18]. The round-robin method simply divides the time evenly among all VUs. During scheduling, the scheduler can only issue one instruction per wavefront and maximal 5 wavefronts at once. Furthermore, the instructions must differ in their category, for example, SU or VU or memory instructions [18].

2.1.3. Comparison of the Architectures

The comparison in this section is based on the hardware specifications, whereas a comparison of the GPUs based on measurements is included in the evaluation chapter. Table 1 lists easily comparable hardware specifications. For the deeper comparison of the 2 GPUs, this section focuses on the SM and CU. The VUs of a CU are comparable with the processing blocks of an SM since both have the capacity for working on 16 threads or work items simultaneously and both, the SM and CU house four of the respective components.

As an advantage, the SM features a scheduler for every processing block, whereas the CU has only one scheduler for all four VUs. A processing block gets instantly the next available warp scheduled after it completes the execution of a warp. The scheduler of the CU however works every clock cycle on a different VU. If an instruction takes n cycles, where $n \,\%\, 4 \neq 0$ or no wavefront is available for execution during the scheduling, the VU stands still for 3 cycles until the scheduler is able to schedule on this VU again, even if a wavefront would be available in the meantime. Thus, a CU can have an idle VU although a wavefront would be ready for execution, whereas the SM can put a processing block instantly to work if there are warps available.

	Tesla V100	**Radeon Instinct MI50**
Memory Size	16 GB	32 GB
Memory Type	HBM2	HBM2
Bandwidth	1133 GB/s	1024 GB/s
Number of SMs (NVIDIA)/CUs (AMD)	80	60
FP32 Cores per SM (NVIDIA) Lanes per CU (AMD)	64	64
Total number of FP32 cores (NVIDIA) Total number of VU lanes (AMD)	5120	3840
Shared Memory per SM (NVIDIA) LDS per CU (AMD)	96 KB	64 KB
Register File per SM (NVIDIA) VGPR per CU (AMD)	256 KB	256 KB
Boosted clock speed	1597 MHz	1746 MHz
Theoretical FP32 performance (based on boosted clock speed)	15.7 TFLOPs	13.41 TFLOPs

Table 1: Comparison of hardware specifications

Another advantage of the SM is the set of separate INT32 cores in every processing block. They enable the parallel execution of FP32 and INT32 instructions for warps. In contrast, the CU only incorporates the SU, which is shared between all four VUs. Furthermore, the SU can only execute on one piece of data (SISD) in contrast to the 16 INT32 cores, which can work on a set of data (SIMD). The SU is only used if all 64 work items of a wavefront perform the same operation with the same parameters and if a wavefront of a VU already occupies the SU, wavefronts residing on the other VUs cannot use the SU. Hence the SM can execute a floating-point and integer instruction simultaneously for every processing block, whereas the CU can only execute both operations if the SU is not occupied and the integer instruction is the same for the whole wavefront.

The architectures differ in the grouping size of work items or threads. Both, the CU and SM have the capacity for four simultaneously executed groups. But NVIDIA bundles 32 into a warp and AMD 64 into a wavefront. This is not automatically an advantage for one of the architectures. If a program cannot utilize the size of 64 work items, the VU still executes all 64 work items and unused lanes perform no useful work. But if the capacity is used, the VU may perform more work at once than a processing block. This is dependent on the executed instruction: For example, the VU needs, as the processing block, four cycles for the FP32 FMA (Fused Multiply-Add), but it can process the doubled amount of work items or threads than its NVIDIA counterpart. This leads to another, closely related aspect. The CU can handle more threads than an SM, namely 2560 to 2048, since the CU can hold up to 40 wavefronts with 64 threads each in its buffers, the SM can hold 64 warps with 32 threads each. But due to the grouping size of a wavefront, this does not automatically translate to an advantage of the AMD GPU. If the program cannot utilize all 64 slots, many work items remain empty.

Additionally, the SM houses 8 Tensor cores, whereas the CU has no comparable counterpart. But for SeisSol the Tensor cores pose no advantage because it employs an algorithm optimized for the used matrix sizes, which does not use Tensor cores. The used algorithm gets explained in chapter 3.2.

Apart from the SMs and CUs the Tesla V100 has another advantage over the Radeon Instinct MI50. The Tesla V100 can be connected to the CPU via NVLink, whereas the Radeon Instinct MI50 is connected via PCIe 4.0. However, the connection to the CPU via NVLink is not mandatory, it can be used only for connecting GPUs, like the Infinity Fabric of AMD. But NVLink offers, depending on the number of used links, up to 300 GB/s, Infinity Fabric only 184 GB/s. The PCIe 4.0 connection to the CPU of the Radeon Instinct MI50 is also slower than NVLink with 64 GB/s, but if the Tesla V100 is not connected via NVLink, it only offers 32 GB/s with PCIe 3.0.

2.2. API

Unlike in the early days of scientific GPU programming, where researchers had only access to the GPU via graphical APIs and formulated their calculations as a shader program in shading languages [20], today both NVIDIA and AMD provide APIs especially designed for non-graphics related usage of GPUs. These are usable by more common programming languages, such as C++. Both manufacturers follow the same basic concepts: To distinguish between the CPU and GPU scope, the CPU scope is called the host side, and the GPU scope the device side. Consequently, functions running on the CPU are called host functions and functions running on the GPU device functions. A special case of device functions is the kernel functions. They are callable by the host side, in contrast to non-kernel device functions, which are only callable by the device side. Furthermore, the kernel functions represent the entry point for the GPU computation. Like the functions, the memory also gets differentiated between host memory and device memory.

The basic structure of a program invoking a kernel function for GPU computation is the same for both APIs: Firstly, the required memory gets allocated on the GPU and the required data gets transferred. Then the kernel function will be called and executed. After the execution, the results get copied back to the CPU and the allocated device memory gets deallocated. But not every algorithm profits by running on the GPU. The GPU clock speed is much slower than a CPU clock speed, but the GPU excels in parallel processing. Furthermore, the copying of the data creates latency, which is reducible to a certain amount by concurrent copying and executing, which is supported by the GPUs. Thus, only heavy parallelable algorithms, for example, gemms (General Matrix Multiplication), can profit when processed on GPUs.

2.2.1. CUDA

CUDA (Compute Unified Device Architecture) [21] is the name of the API provided by NVIDIA. It was released in 2006 as a library for C and is now available for multiple programming languages on Windows and Linux. It provides language extensions for marking functions and variables for GPU or CPU usage and functions for controlling the data and program flow. Files using the CUDA-specific features need to include the cuda.h file and to be compiled by the nvcc compiler. The code snippet below shows a simple CUDA C++ program to demonstrate important CUDA features. The program sums up two vectors and stores the result in a separate vector.

```
#include <cuda.h>

#define SIZE 100

__global__ void matrixMultiplicationKernel(int* a, int* b, int* c){
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        while(tid < SIZE){
                c[tid] = a[tid] + b[tid];
                tid += blockDim.x * gridDim.x;
        }
}

int main(){

        int host_a[SIZE], host_b[SIZE], host_c[SIZE];

        for(int i = 0; i < SIZE; i++){
                host_a[i] = i;
                host_b[i] = 2 * i;
        }

        int *dev_a, *dev_b, *dev_c;

        cudaMalloc(&dev_a, SIZE * sizeof(int));
        cudaMalloc(&dev_b, SIZE * sizeof(int));
        cudaMalloc(&dev_c, SIZE * sizeof(int));

        cudaMemcpy(dev_a, host_a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, host_b, SIZE * sizeof(int), cudaMemcpyHostToDevice);

        matrixMultiplicationKernel<<<32,32>>>(dev_a, dev_b, dev_c);

        cudaMemcpy(host_c, dev_c, SIZE * sizeof(int), cudaMemcpyDeviceToHost);

        cudaFree(dev_a);
        cudaFree(dev_b);
        cudaFree(dev_c);
38.         return 0;
39.
40. }
```

Line 1 contains the include statement for the CUDA library. In lines 5 – 11 the kernel function gets defined. It is marked by the `__global__` keyword. The GPU runs the kernel function in blocks of threads. The blocks get assigned to an SM and the threads are distributed onto warps, where each processing block executes the kernel function. To coordinate the computation, each thread and block get numbered. The programmer can access these indices by the `threadId` and `blockId` variables provided by `cuda.h` as shown in line 6. For example, the fifth thread of the second block works on the index $4 + 1 * 32 = 36$. In case the problem dimension is bigger than the amount of called threads, the index `tid` gets incremented in line 9. The maximum number of threads is determined by the hardware limits of the GPU. In lines 24 - 26 the memory on the GPU gets allocated by the `cudaMalloc` function and the needed data gets transferred in lines 28 – 29 using the `cudaMemcpy` function. Both functions need the size of the data, `SIZE * sizeof(int)`, and the copy function needs additional information about the direction of the transfer. In this case, the copy goes from the host to the device, represented by `cudaMemcpyHostToDevice`. In line 31 the kernel function gets launched by the CPU, marked by the `<<<32,32>>>` expression after the function name. The first number determines the number of launched blocks, the second the number of threads per block. After the launch, the resulting data gets copied back in line 33, this time the direction is `cudaMemcpyDeviceToHost`, and the memory on the GPU gets deallocated in lines 35 – 37 via `cudaFree`. Apart from this simple example, the CUDA library offers more functionality like synchronization of threads or streams, which goes beyond the scope of the thesis.

2.2.2. HIP

AMD released its API HIP (Heterogeneous-compute Interface for Portability) [18] in 2016. It currently is only available for Linux and C++ but support for Fortran is in development [18]. It offers features similar to CUDA. The code snippet below shows the same program as in the CUDA section, but this time written with HIP instead of CUDA. Since HIP was designed to enable an easy transition from CUDA to HIP the resemblance of HIP to CUDA is apparent. The HIP function calls are syntactically like the CUDA versions except for `hip` in front instead of `cuda`. Their functionality is also the same, as the syntactical similarity indicates. Hence only differences aside from the `cuda` to `hip` transition require additional explanation.

```
1.  #include "hip/hip_runtime.h"
2.
3.  #define SIZE 100
4.
5.  __global__ void matrixMultiplicationKernel(int* a, int* b, int* c){
6.          int tid = hipThreadIdx_x + hipBlockIdx_x * hipBlockDim_x;
7.          while(tid < SIZE){
8.                  c[tid] = a[tid] + b[tid];
9.                  tid += blockDim.x * gridDim.x;
10.         }
11. }
12.
13. int main(){
14.
15.         int host_a[SIZE], host_b[SIZE], host_c[SIZE];
16.
17.         for(int i = 0; i < SIZE; i++){
18.                 host_a[i] = i;
19.                 host_b[i] = 2 * i;
20.         }
21.
22.         int *dev_a, *dev_b, *dev_c;
23.
24.         hipMalloc(&dev_a, SIZE * sizeof(int));
25.         hipMalloc(&dev_b, SIZE * sizeof(int));
26.         hipMalloc(&dev_c, SIZE * sizeof(int));
27.
28.         hipMemcpy(dev_a, host_a, SIZE * sizeof(int), hipMemcpyHostToDevice);
29.         hipMemcpy(dev_b, host_b, SIZE * sizeof(int), hipMemcpyHostToDevice);
30.
31.         hipLaunchKernelGGL(matrixMultiplicationKernel, 32, 32,
32.                                                 0, 0, dev_a, dev_b, dev_c);
33.
34.         hipMemcpy(host_c, dev_c, SIZE * sizeof(int), hipMemcpyDeviceToHost);
35.
36.         hipFree(dev_a);
37.         hipFree(dev_b);
38.         hipFree(dev_c);
39.         return 0;
40.
41. }
```

One main difference is the include statement in line 1, here the HIP runtime needs to be included. The only difference in the kernel function is the nomenclature of the thread and block IDs in line 6, but they work like the CUDA variant. A more distinct difference is the launch call of the kernel function in lines 31 – 32. It is launched by the `hipLaunchKernelGGL` macro, which takes the function name and the launch dimensions (blocks and threads) as the first arguments. The following two arguments are not important for the example, they specify the amount of additional shared memory required by the kernel and the stream ID it should run into, where zero corresponds to the default stream. The remaining arguments are the function parameters for the kernel function. The kernel function gets executed in a CU, where the threads of the blocks get organized into wavefronts and distributed to the VUs. Like NVIDIA AMD has a hardware-dependent limit for the number of threads launched by a kernel.

The HIP environment offers command-line tools for an automatic port from CUDA to HIP code: `hipify-perl` and `hipifiy-clang`. The first tool is faster, but the programmer may have to port more code manually. The second one requires, as the name indicates, an installation of the clang compiler, but works more precisely and generates additional information like warnings about the port procedure [18].

Furthermore, HIP code can be compiled for NVIDIA GPUs with the nvcc compiler. The HIP compiler hipcc invokes depending on the platform the nvcc compiler for CUDA or the hip-clang compiler for AMD GPUs. To realize the multiplatform support in the code the `hip_runtime.h` header itself includes depending on the platform either a separate header for HIP or CUDA. The CUDA header includes the actual `cuda.h` header and several other headers, that contain macros and function definitions that redirect the HIP library calls to CUDA calls, for example, `hipMalloc` to `cudaMalloc`. The HIP-specific header contains the actual code for the AMD platform. Currently HIP supports not all CUDA features, a list of the supported features can be found in the documentation of HIP. To execute a program including HIP on an AMD GPU an installation of the ROCm[1] open-source software platform is required.

2.3. SeisSol

SeisSol is software for simulating earthquakes. Its mathematical model solves the elastic wave equation:

$$\frac{\partial Q}{\partial t} + A\frac{\partial Q}{\partial x} + B\frac{\partial Q}{\partial y} + C\frac{\partial Q}{\partial z} = 0$$

by a combination of the discontinuous Galerkin (DG) and the arbitrary high-order derivatives (ADER) method as described in [22]. The elastic wave equation is dependent on time and space, where the DG method discretizes it in the space and the ADER scheme in the time domain resulting in a discrete model. Space dependent matrices A, $B,$ and C represent the physical properties of the represented material and Q is a vector of unknowns, namely stress and velocity components. For the simulation, the simulated area gets divided into a mesh of tetrahedrons and advances in timesteps, computing the changes in the tetrahedrons at each timestep. The combination of all tetrahedrons represents the simulation of the whole area. The model allows different granularity in the mesh represented by tetrahedrons of different sizes throughout the domain [22]. A less important area of the simulation can be represented by bigger tetrahedrons, thus reducing the number of tetrahedrons, and needed computation. For a similar granularity in time, SeisSol allows for different time stepping in the tetrahedrons called local time-stepping (LTS) as described in [23]. Each tetrahedron stores its time step $\Delta t \in \mathbb{N}^+$ and current time t and may only update, if its $t + \Delta t$ is smaller than the $t + \Delta t$ of its neighbors.

[1] https://rocmdocs.amd.com/en/latest/index.html#

Due to that tetrahedrons with lower importance can be set to bigger time steps thus reducing the amount of the total time steps. If LTS is not applied, all tetrahedrons must match the time step of the tetrahedron needing the highest accuracy, i.e., smallest step size [24], resulting in more timesteps and thus needed computation.

For the computation of the updates in the tetrahedrons, the solution for the vector of unknowns Q gets approximated by a polynomial represented by a time-dependent matrix containing the coefficients of the polynomial called degrees of freedom and a set of space-dependent, orthogonal basis functions of degree N. N also determines the convergence order of the whole ADER-DG scheme. Furthermore, applying the DG scheme, the elastic wave equation gets multiplied with a test function of the same family as the basis function and integrated over the current tetrahedron. Additionally, a numerical flux representing the influence of the neighboring tetrahedrons is added. For the actual computation, all tetrahedrons get transformed via a transformation matrix into a single reference tetrahedron of known. The transformation into the reference tetrahedron allows a more efficient computation since several parts of the resulting equation can be computed beforehand for the reference tetrahedron, which gets used for the update of every tetrahedron.

The time integration of degrees of freedom gets computed via the ADER scheme. Hereby the solution of the elastic wave equation gets approximated regarding time using a Taylor series up to the order N, which is the same as the degree N of the basis functions used for the DG. The time derivative of the Taylor series gets replaced by space derivatives using the Cauchy–Kovalewski procedure.

The desired convergence order has the be chosen before the compilation and the degree N gets determined based on the convergence order. Via N several dimensions of the matrices used for the computation get defined. This is used to achieve higher performance by generating custom, optimized code for the matrix multiplication depending on the known matrix information and targeted hardware. SeisSol supports computation only on CPUs and optional GPU offloading for the matrix multiplications. For this thesis, only the offloading to the GPU is considered. The generation of the kernel and kernel call functions code is done by the GemmForge library, which got introduced for NVIDIA GPUs in prior work [5] and is covered in chapter 3.2.

SeisSol supports computation on multi-node systems. Therefore, the tetrahedrons of the mesh get clustered based on their LTS time step as described in [23]. As a restriction, a tetrahedron of the cluster C_l may only be surrounded by tetrahedrons of the time clusters C_{l-1}, C_l or C_{l+1}, where l marks the time step. Then the mesh gets divided into partitions, where a partition may contain tetrahedrons from more than one time cluster. Furthermore, a partition includes additional memory space for the neighbors of the tetrahedrons located on the border of the partition. This is needed since a tetrahedron requires the input of its neighbors for its update. These partitions get distributed onto the nodes of the HPC cluster. The nodes compute the updates for their assigned partitions independently and exchange data with other nodes if necessary.

2.4. OpenMPI

For the communication between the nodes, SeisSol uses an open-source implementation of the MPI standard called OpenMPI. The MPI (Message Passing Interface) standard specifies a communication system for parallel processes [25]. As the name indicates, the communication between the processes is realized via messages. MPI manages all the actual communication for the program so that the developer only needs to invoke MPI-specified communication functions. For the messages MPI includes reliability of transmission, meaning that a message will always be received correctly and in the right order if several messages got sent consecutively. The programmer must only focus on program errors and resource errors, which may occur due to exceeded system resources. For the organization of the communication MPI associates each process with a rank, a unique integer value in the running context. This rank is used as the destination and source for the messages. The message passing is realized via two function types for point-to-point communication: one for sending the message and one for receiving. Both function types contain information about the sent data and a communication envelope. Naturally, the send-function contains the actual data, and the receive-function a buffer for the data to be stored in. Additionally, the receive function contains status information about the message, including an error message, in case an error occurred during the send. The envelope contains the information for the addressing. It has two ranks, one for the source and one for the destination. Additionally, a tag and a communicator, defining the communication context, must be included. A message can only be received if the send-function and receive-function both define the same envelope. However, a receiver may accept any rank and tag in the envelope, but the communication context must be identical to the sender. The communication context restricts the message to a group of processes. The processes can be organized in groups to resemble programming patterns if necessary. MPI features blocking, which ensures that the program execution waits until the message got successfully stored or sent, and non-blocking variants, allowing the program to continue execution instantly after the call of the function, of the send/receive functions. For both variants, four modes are included, which define when the actual sending is started depending on a call of a matching receive function. Additionally, collective communication, for example, one rank broadcasting to all other ranks in a communication context, functions, and much more functionality, which goes beyond the scope of this thesis, are also included in MPI.

2.5. UCX

For the transportation of the OpenMPI messages, SeisSol uses the open-source framework Unified Communication X, abbreviated UCX. UCX acts as an interface between the communication model of the program and the used hardware as described in [26], [27], and [28]. In this case, the used communication model is MPI, the following figure depicts the influence of UCX in the communication between the MPI processes:

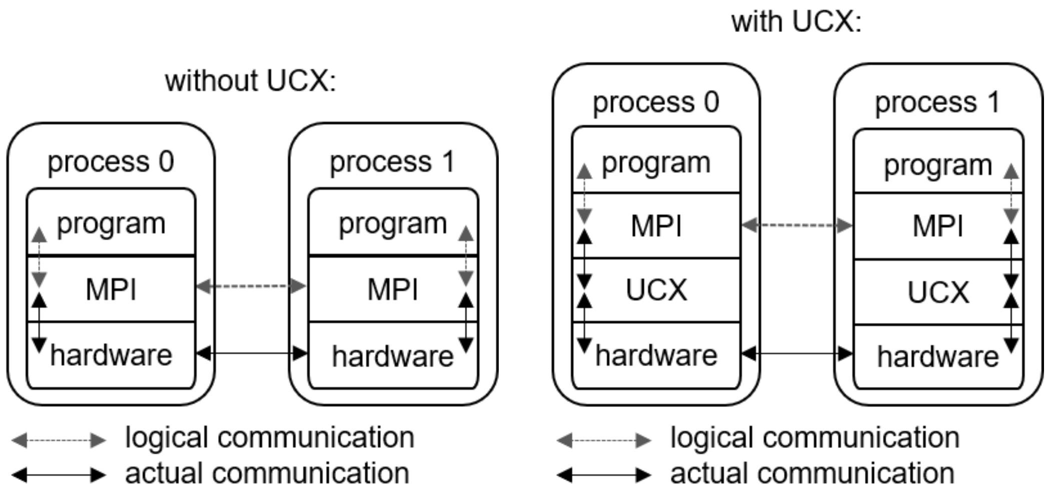

Figure 2: Communication between the processes

As a main advantage, UCX takes over the hardware support from MPI but introduces a small performance loss due to the additional layer. MPI can use interfaces defined by UCX to communicate via different transportation technologies without performing adjustments based on the transportation medium. Thus, the portability of the program gets easier. During its installation, UCX automatically detects available transport systems already installed on the machine. It supports many communication models and hardware transportation types and allows easy integration of new models and technologies. Furthermore, it enables a mixture of different communication models, where the models can communicate via the UCX interface. It features three major frameworks:

UCP (Protocols), the high-level API: UCP provides interfaces for the communication libraries. Communication libraries can use the provided interfaces for their communication without knowing the underlying hardware or which transport type gets used by UCX. UCP is used for the initialization of the UCX framework and managing the communication. Hereby it fulfills several tasks. Firstly, it chooses the best transportation medium. It considers all available network devices and transport options and chooses the best for the task, for example, the ROCm environment for transporting to an AMD GPU. Hereby performance and scalability are the crucial factors, but the user can restrict the options by editing a UCX environment variable for devices and transport options. Secondly, UCP uses multi-rail communication if available. The multi-rail model uses several network devices for the transmission of one message. The message gets distributed onto the network devices according to their performance characteristics. The distribution of messages can be uneven for network devices with different performance characteristics. It supports up to four rails but uses two by default. The user can change the number of rails via an environment variable, setting this variable to 1 results in no multi-rail usage. Furthermore, it handles the fragmentation of messages. If all preparations are made, it passes the message via the underlying UCT layer to the selected hardware. UCP consists of several interfaces, providing functionality for the different communication models [27][29]:

- Initialization: These interfaces collect information about the used hardware and initialize the UCP endpoints, which get used as source and destination by the messages.
- RMA: Here single-sided communication functions get provided, needed by the OpenSHMEM library for example.
- AMO: These interfaces enable atomic operation on remote memory, needed by PGAS models.
- Tag Matching: These provide the ability to give messages tags to make messages distinguishable, which is needed by MPI models.
- Active message: These enable callback functions to be executed by a receiver if it gets a message.
- Collectives: These interfaces provide synchronization operations and group communication like broadcasting and all-to-all exchange, needed by MPI models.
- Stream: In these interfaces the data gets treated as a stream flowing over the transportation layer, implying ordered data. This is needed by BSD-socket-based models.

UCT (Transport), the low-level API: UCT represents the transport layer, which gets used by the UCP for the actual transportation. It supports several connection types like PCIe and NVIDIAs NVLink. To minimize performance losses overusing the designated transport directly, UCT relies on hardware drivers provided by hardware manufacturers. It provides 3 communication types: The short operation, for small messages, which can be sent completely at once. For bigger messages, it provides buffered copy and send operations, where the message gets sent in several parts via a buffer. This also provides an option for non-contiguous transportation. Direct memory to memory communication is supported via zero-copy operations.

UCS (Services), the service layer: UCS provides utilities for efficient communication like commonly used data structures and memory management tools.

All three APIs can be used independently as they were designed to work together but not rely on each other. To configure UCX after installation the user can modify a variety of environment variables. UCX is included in OpenMPI and MPICH (another open-source implementation of MPI) since version 3.0 (OpenMPI) and version 3.3 (MPICH) respectively.

3. Implementation

SeisSol relies on two libraries for the GPU offloading: The Device[2] library, which was designed to provide a common interface for different GPU APIs, and GemmForge, generating custom kernel and GPU kernel launcher functions for gemms. Both libraries already existed before the thesis and got extended and modified during the thesis.

3.1. The Device library

As mentioned, the C++ Device library provides an interface that passes the GPU-related calls to the currently needed GPU API depending on the hardware. SeisSol as the caller does not need to handle GPU API calls as the Device library handles all the GPU API-related work and returns platform-independent error messages and memory pointers to SeisSol.

3.1.1. Consideration of using HIP

At the first glance, the HIP library can be considered as a good substitution for the Device library, since it is designed to support both AMD and NVIDIA GPUs. However, HIP does not support all of CUDA's functionality. For example, the CUDA function `cudaMemPrefetchAsync` is not supported by HIP and thus would not be available if HIP were used. The Device library uses the CUDA function for usage on NVIDIA hardware and provides a workaround for AMD GPUs. Hence the Device library can maximize performance for each platform if the related API provides a better implementation of functionalities. Secondly, the Device API includes additional functionality around the GPU API calls, which get explained in chapter 3.1.3. If HIP were used instead, these additions would have to be implemented in a separate library resulting in a Device-like library without the advantages of the Device library. Furthermore, the Device library is extendable for a new GPU interface or hardware in the future, for example, OneAPI i.e., the Intel implementation of SYCL standard. A new interface just needs to implement the functionality required by the Device library as well as the compilation process needs to be configured for the new API. SeisSol itself would require no changes in the code to use a new API or hardware.

[2] https://github.com/SeisSol/Device/

3.1.2. Design

The design of the Device library is based on a mixture of two structural patterns. It provides the `AbstractAPI` struct, which acts as both Facade [31] and Adapter [32]. The platform-dependent declaration is made in a separate subclass for each platform. These classes implement only access to those GPU API functions that are necessary for SeisSol, therefore acting as a facade to the GPU APIs. This gets achieved by wrapping functions that require no special datatypes. For functions requiring special datatypes, it enables access without the need to include the GPU APIs, thus acting as an adapter. It converts GPU-specific datatypes to standard C++ datatypes or manages the actual control of these data types and provides control functions to the user. The actual implementation of the subclasses is made in several classes, each covering one aspect of functionality, for example, memory allocation or copying data between host and device. For access to the `AbstractAPI` struct, the library provides the `DeviceInstance` singleton. Singleton is a design pattern, where only one object of the class can exist at any given time in the context of a running program [33]. If the program tries to create a new, separate object, it will receive a reference to the already existing object. Based on the singleton pattern of the `DeviceInstance`, only one object of the platform-dependent implementation of `AbstractAPI` exists at any given time. This is important for the additional functionality provided by the Device library, which gets described in the next chapter.

For the HIP support, a new subclass of `AbstractAPI` and a set of classes containing the implementation were added in the scope of the thesis. Most of the code was portable using `hipify-perl` since mostly only prefix `cuda` needed to be replaced with `hip`. But some parts required workarounds due to missing HIP counterparts. A special case is the usage of NVToolsExt, which is an additional API provided by NVIDIA [30]. It improves logging and visualization for the analysis of CUDA programs. Since no HIP alternative could be found, functions relying on it were left empty. This poses no problem for the functionality since the API is only used for performance analysis.

The compilation process of the library is modeled using CMake. The process can be split up into three phases shown in the following figure:

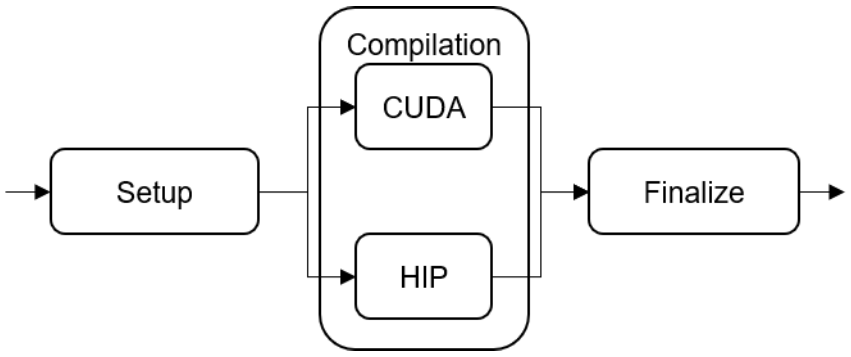

Figure 3: The Device compilation process

Setup: During the first phase CMake checks if the desired variables are set. The user must define the variables `REAL_SIZE_IN_BYTES` and `DEVICE_BACKEND`. The first one determines the used precision whereas the second variable describes the used programming API and takes CUDA or HIP as correct values. For further optimization, during the compilation, the user must also define the `DEVICE_SUB_ARCH`. This variable takes the version label of the SM or CU of the targeted GPU architecture as values.

Compilation: As the name indicates, this phase covers the actual compilation. Depending on the values of `DEVICE_BACKEND` the progress shifts either to CUDA or HIP compilation. While the HIP path got added during this thesis, the CUDA path already existed upfront.

The CUDA path firstly loads the CUDA CMake package, containing CMake settings specialized for CUDA compilation. Then it tries to find the library NVToolsExt. If it is not found, the Device library gets compiled without the support for NVToolsExt. Afterward, the compilation flags for the nvcc compiler get set. To compile the CUDA implementation of `AbstractAPI`, the nvcc compiler gets invoked by the `cuda_add_library`. Lastly, some CUDA specific include libraries and paths get set.

The HIP path has the same structure, but with major differences. Since HIP is not yet a part of the default CMake package, the path to the installation of HIP must be set manually. Additionally, the helper CMake files located in the HIP library must be included. In contrast to the CUDA path, the HIP path defines compiler flags for both CUDA and HIP, since HIP may be used to compile for an NVIDIA GPU. These flags get defined in dedicated variables for each platform and an additional variable for flags used by both. The variable `CMAKE_HIP_CREATE_SHARED_LIBRARY` must be set to ensure the correct compilation as a library and not as an executable file. The hipcc compiler gets invoked like nvcc using `hip_add_library`, which is included in the CMake helper files from the HIP library. Based on the platform the link to the hip library gets set in different ways, for CUDA it can be set via a property variable in CMake, for HIP a direct path to the library must be set.

Finalize: In the last stage compilation flags common to both paths get set. This includes a flag determining the desired precision. Lastly, platform-independent include directories get defined.

3.1.3. Functionality

The Device library includes additional functionality apart from the wrapping of GPU API calls. It gathers information about the amount of allocated memory and copied data. This gets collected by incrementing statistics variables after the invocation of GPU API calls. Furthermore, it performs error checks after each GPU API call, logging arising error messages. Aside from informative functionality, it provides functions for managing a temporary memory located on the GPU, which is organized as a stack. A set number of bytes get allocated on the GPU and managed by the library. The programmer can get a pointer to available temporary memory, which then gets marked as used. If the programmer does not need it anymore, the memory gets unmarked but not deallocated. Thus, the programmer does not need to allocate and deallocate memory for temporary usage and can potentially reduce the execution time, since less time is used on memory management. For access to GPU streams, the library provides a circular stream buffer. Streams are a concept for interweaving data copying from or to GPU and calculation on the GPU. This can result in an increased performance since due to hardware characteristics copies can be performed in parallel with computation on the GPU. Instead of using one block of data and one kernel, a program can issue several small kernels working on smaller pieces of data to different streams. The GPU can copy the data required by one stream during the execution of the kernel of another stream. The Device library can hold up to a predefined number n of streams. If the program tries to access the stream $n + 1$, it gets redirected to the first stream, hence it acts as a circular buffer. The library provides a function for synchronizing a single stream or all streams at once. In SeisSol streams are primarily used to launch several small kernels, which work on independent data.

3.2. GemmForge

For generating efficient kernels GPU SeisSol deploys the GemmForge[3] library. It is written in python 3 and includes a generator for copy-add-scale operations and gemms (**ge**neral **m**atrix **m**ultiplication). It produces a program text for the kernel function and the corresponding kernel call function. Since the matrices involved in the computations are small, the maximum dimension used in this thesis is $56x56$, GemmForge can employ

[3] https://github.com/ravil-mobile/gemmforge/

optimizations and thus outperform GPU standard libraries for matrix multiplication, as shown in [5]. GemmForge relies on the standard FP32 or FP64 cores, whereas NVIDIAs standard libraries often use Tensor cores for their matrix multiplication. For the computation of $C = A * B + C$, where A,B and C are matrices, GemmForge loads matrix B in the shared memory, which is possible due to the small size of the matrix, for large matrices shared memory maybe not sufficient. Each thread loads one element of the same column of A into its register. Thus, the column size of A defines the number of active threads launched. Each thread performs the multiplication on the same row of B. This row now can be broadcasted to all threads from shared memory, which is faster than each thread accessing the row consecutively [5]. For further optimization, GemmForge relies on information about the used GPU provided before the compilation. It calculates how many matrix multiplications can be done by a single block based on estimations of available shared memory and registers. If possible, it launches kernels in a two-dimensional manner, where the size of the second dimension corresponds to the number of matrix multiplications in a block. The size of the first dimension is based on the number of threads needed for one column and is always a multiple of the warp or wavefront size. Hardware analysis and generation of a kernel are performed in the `GemmGenerator` class. To generate a kernel a programmer must provide hardware and matrix characteristics. Hardware characteristics contain manufacturer and specific GPU architecture. Matrix characteristics contain the dimensions of the involved matrices, which must be suitable for multiplication, otherwise, an error message gets printed. Furthermore, they contain scale factors for matrix A and C, a Boolean for each matrix indicating if it should be transposed and an addressing mode, defining how pointers to matrix entries are assembled. For loading of shared memory and a possible transposition, GemmForge provides the `loader` module, which contains classes specifying the sizes and loading of the shared memory. Since GemmForge was originally designed only for NVIDIA GPUs, a way of supporting both HIP and CUDA was introduced for this thesis, which gets explained in the following two chapters.

3.2.1. The Architecture class

GemmForge uses the `Architecture` class defined in the file `arch.py` to hold hardware-specific data and characteristics for different GPU architectures. The class provides `produce` function, which determines the used GPU by two strings, one containing the name of the manufacturer and the other containing the name of the GPUs architecture. Then it returns an `Architecture` object filled with the corresponding hardware characteristics. The class provides information about the size of warps or wavefronts, the sizes of the different memory types, and how many threads and blocks can be launched on a single CU or SM. For AMD support the `produce` function got modified to distinguish between the two hardware manufacturers and the hardware characteristics of the Radeon Instinct MI50 and its successor Instinct MI100 got specified. The class can be easily extended for new architectures versions from NVIDIA or AMD, it only needs the specific hardware characteristics defined. For a new manufacturer, a new distinction must be included containing the characteristics of its GPUs.

3.2.2. Architecture lexicons

During the generation of the kernel in the `GemmGenerator` class, GemmForge used the hardcoded CUDA specific thread and block variables and kernel call. To support multiple manufacturer-specific APIs, the `arch_lexic` (an abbreviation of architecture lexicon) module got implemented. It is designed according to the Factory method design pattern. In the factory function, a direct creation of an object of the desired type, in this case, lexicons, gets replaced by a factory function, which returns the correct object depending on the current context, in this case, hardware manufacturer. The module defines the abstract superclass `AbstractArchLexic`, which defines the needed variable names and functions for accessing and combining them into often needed combinations. Platform-dependent implementation is realized in subclasses, containing the corresponding variable names for the thread and block variables. Furthermore, subclasses implement a function for creating kernel launch function call. For the creation of an `ArchLexic` object, the factory function `arch_lexic_factory` is provided, which returns a lexicon corresponding to the used platform. The module can be easily extended for new APIs, if the new APIs programming model is a CUDA-like programming model, using thread and blocks to organize the distribution of the threads created by the kernel on the hardware. The `arch_lexic` module got introduced in several already existing classes, replacing the hardcoded CUDA-specific strings. The `arch_lexic` is used in several places – i.e., `GemmGenerator`, `CsaGenerator`, and `loaders`, which allows to fully get rid of all CUDA hardcoded keywords and thus make GemmForge more flexible and extensible for future changes.

4. Evaluation

For evaluation of the implementation, we examine the libraries GemmForge and Device. Thereby we assess if the kernels generated by GemmForge work properly and measure and evaluate delivered performance on the AMD Radeon Instinct MI50. The measured performance gets compared to results measured on NVIDIAs Tesla V100 to evaluate the delivered performance of the AMD GPU. To assess if the extension of the Device library was successful, we created a benchmark by implementing a Jacobi solver. This benchmark serves furthermore as an evaluation of the combination of HIP and OpenMPI, to ensure SeisSol could get executed on multi-GPU systems and leverage available performance.

4.1. Roofline Model

For performance evaluation of the kernel functions generated by GemmForge, we use the Roofline model presented in [34]. This model requires measurements of bandwidth and kernels performance, for which GemmForge provides benchmark tests. For the thesis, these tests got ported to HIP. This chapter first describes the theoretical background of the Roofline model and then its application to the gemms.

4.1.1. Theoretical background

The Roofline model provides a method for evaluating the performance of programs running on multicore systems. Originally designed for multicore CPUs in 2008, it can also be used for the analysis of programs using GPUs for computation since it is based on concepts occurring in both CPU and GPU, namely floating-point and memory performance. The model can be divided into two steps: Firstly, the creation of a hardware-dependent graph based on measurements taken on the processor. Secondly, derivation of the program into a form useable by the model and measuring the performance of the program.

The graph defines the upper bound (roofline) for achievable performance on the hardware. It is plotted on a two-dimensional domain where the y-dimension depicts performance in FLOP/s. The x-dimension represents the operational intensity of the program in FLOP/byte, which is going to be explained shortly. The graph is derived from a combination of two hardware characteristics: maximum performance of the multicore system in FLOP/s and achievable bandwidth for memory movement between main memory and caches of the processor cores in byte/s. The maximum performance is represented as a horizontal line at $y = maximumPerformance$. The bandwidth however corresponds not directly to one of the axes, but can be represented by a combination of both axes:

$$\frac{\frac{FLOP}{second}}{\frac{FLOP}{byte}} = \frac{\frac{1}{second}}{\frac{1}{byte}} = \frac{byte}{second}$$

Thus, the bandwidth is represented by $y = maximumBandwidth * x$. The combination of both functions results in the graph $g(x)$, which can be described as followed, where x_c donates x coordinate of the intersection of both functions:

$$g(x) = \begin{cases} maximumBandwidth * x \ if \ x < x_c \\ maximumPerformance \ otherwise \end{cases}$$

The values of $maximumBandwidth$ and $maximumPerformance$ can be theoretical limits or measured by benchmarks. If a graph is constructed for given hardware it can be used for all program evaluations run on this hardware. For new hardware, a new graph must be constructed.

For performance evaluation of the program, the operational intensity of the kernel must be calculated first. The operational intensity I_k of a kernel can be described as:

$$I_k = \frac{Number\ of\ floating\ point\ operations}{Number\ of\ bytes\ moved\ between\ main\ memory\ and\ cache}$$

But since we work on GPUs, we use arithmetic intensity instead of operational intensity and thus define I_k as followed:

$$I_k = \frac{Number\ of\ floating\ point\ operations}{Number\ of\ bytes\ moved\ between\ GPU\ memory\ and\ GPU\ registers}$$

The arithmetic intensity is used since the data is already resident on the GPU during execution of the kernel since it gets copied onto GPU memory before the invocation of the kernel. If the intensity I_k of the kernel is known, a measurement in FLOP/s can be made and plotted in the graph as the point x_k:

$$x_k = (I_k; measuredPerformance)$$

If the measured performance is significantly lower than $g(I_k)$, i.e.,x_k is not near the graph, the model indicates that the kernel function can be optimized further.

4.1.2. Application to AMD's Radeon Instinct MI50

For the application of the Roofline model on the kernels generated by GemmForge, the graph $g(x)$ must be created for AMDs Radeon Instinct MI 50. As maximum performance, the theoretical performance of 13.41 TFLOP/s provided by [16] is used. The maximum bandwidth gets measured on the Radeon Instinct MI50. Therefore, the ported version of the bandwidth benchmark created for measurements of the NVIDIA Tesla V100 in [5] gets used. In the benchmark, a predefined amount of GB gets allocated twice on the GPU. A kernel copying the data from one allocation to the other gets executed. The execution time of the kernel gets measured to achieve a GB/s value. To even out performance peaks and lows the kernel gets repeatedly executed and the measured time averaged depending on the number of repeats. For the port, the file `global.cu` got a HIP equivalent called `global.cpp`. The file contains the explained kernel function and additional utility for measurements. Furthermore, the compilation process got a new path for HIP compilation. To start a test the number of repeats and number of GBs must be defined in an input file. To establish a bandwidth value for the Roofline model the benchmark got executed with different combinations of GBs and number of repeats. The number of GBs ranged from 0.5 to 8 GBs, incrementing in steps of 0.5 GBs, whereas the number of repeats ranged from 100 to 1000, incrementing in steps of 100 repeats, resulting in 160 tests in total. The test parameters are widespread to generate a result that is not restricted to a single use case but to get a more uniform value for different use cases. A selection of test results is shown in the following figure:

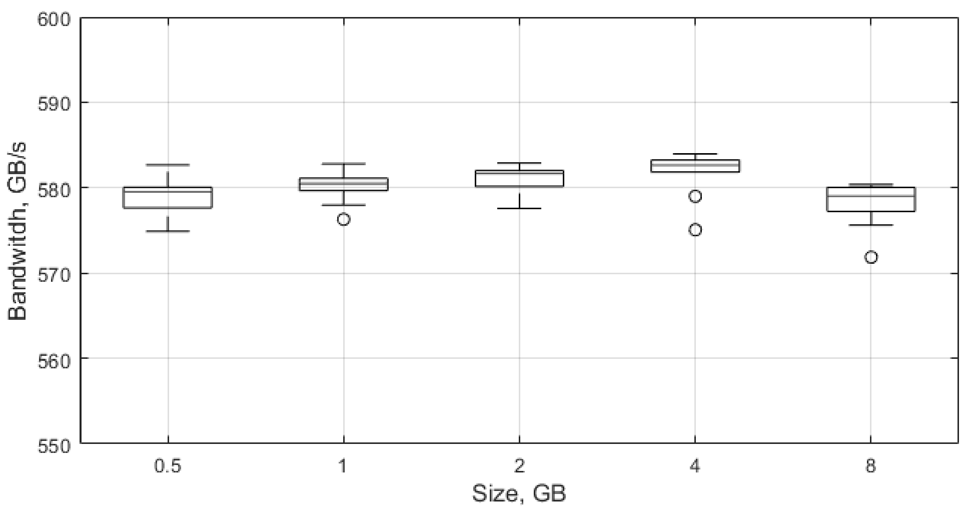

Figure 4: Selection of bandwidth measurements

The results are spread in a small range with only a few outliers. The average bandwidth of all tests was about 580 GB/s, which is about 56,6% of the theoretical performance of

1024 GB/s documented in [16]. Reasons for this discrepancy require deeper investigation, which goes beyond the scope of the thesis.
Since both maximum bandwidth and performance are known, the value of x_c can be calculated as:

$$x_c = \frac{13.41 \frac{TFLOP}{second}}{0.58 \frac{TB}{second}} \approx 23.1 \ \frac{FLOP}{Byte}$$

And thus, the function $g(x)$ can be expressed as:

$$g(x) = \begin{cases} 0.58 * x \ if \ x < \ 23.1 \\ 13.41 \ otherwise \end{cases}$$

For the performance measurement of the kernels, the benchmark as designed in [5] is used. The benchmark calculates the following term, where L and A are $56x9$ sized matrices, whereas matrix B has the dimension $9x9$ and matrix D $56x56$:

$$L = D * A * B + L$$

Since GemmForge only generates kernels of the form $C = A * B + C$, computation is realized in two kernels, where the first kernel computes $T = A * B$ and stores the result in the $56x9$ sized matrix T. The second kernel finalizes the result with the computation $L = D * T + L$. For the plotting of measured results, the arithmetic intensity of both kernels must be calculated first. The intensity I_{k1} of the first kernel can be described as followed, where p is $4 \ Byte = \frac{32 \ bit}{8}$ (float) or $8 \ byte \ = \ \frac{64 \ bit}{8}$ (double) depending on precision:

$$I_{k1} = \frac{9 * 9 * 2 * 56}{(2 * 56 * 9 + 9 * 9) * p}$$

The devisor can be derived from the dimensions of the involved matrices, where A, which gets loaded from global memory into registers, and T, which gets written back to global memory, are $56x9$, hence $2 * 56 * 9$. The matrix B, which is also loaded from global memory, has the dimension $9x9$, thus the addition of $9 * 9$. The multiplication with p translates the result into a number of bytes. For the determination of the dividend the computational part of the generated kernel must be considered. Pointer arithmetic does not contribute to the total number of floating-point operations, since they are integer operations:

```
1.  if (hipThreadIdx_x < 56) {
2.        float Results[9] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
3.        float Value;
4.        for (int k = 0; k < 9; ++k) {
5.          Value = GlobMatA[hipThreadIdx_x + 56 * k];
6.          for (int n = 0; n < 9; ++n) {
7.            Results[n] += Value * ShrMatB[k + 9 * n];
8.          }
9.        }
10.       for (int n = 0; n < 9; ++n) {
11.         GlobMatC[hipThreadIdx_x + 56 * n] = Results[n];
12.       }
13.  }
```

The floating-point operations are in line 7, one addition and one multiplication. These 2 operations are embedded in a nested for-loop in line 6, which is repeated 9 times. The superior for-loop in line 4 is also repeated 9 times. The if statement in line 1 causes execution of the loop in line 4 for 56 times since it represents the parallelization in HIP fashion. The loop in line 10 does not contribute to the number of floating-point operations since it only contains variable assignments and no computation. Thus, the whole number of floating-point operations accumulates to $2 * 9 * 9 * 56$ for the first kernel.

The arithmetic intensity I_{k2} of the second kernel is calculated in a similar fashion. The devisor can be derived from involved matrices too, but with some special cases: matrix D can be ignored since it is the same for all work items in the wavefront. Matrix L appears two times in the calculation since it gets loaded from global memory and after computation back into it. Thus, the number of elements in the devisor can be expressed as $3 * 56 * 9$, since both L and T have the dimension $56x9$. The dividend is again derived from the generated kernel:

```
1.  if (hipThreadIdx_x < 56) {
2.        float Results[9] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
3.        float Value;
4.        for (int k = 0; k < 56; ++k) {
5.          Value = GlobMatA[hipThreadIdx_x + 56 * k];
6.          for (int n = 0; n < 9; ++n) {
7.            Results[n] += Value * ShrMatB[k + 56 * n];
8.          }
9.        }
10.       for (int n = 0; n < 9; ++n) {
11.         GlobMatC[hipThreadIdx_x + 56 * n] = Results[n] +
12.                       GlobMatC[hipThreadIdx_x + 56 * n];
13.       }
14.  }
```

Again 2 floating-point operations are in line 7, but this time the superior loop in line 4 is repeated 56 times. The embedded loop in line 6 is repeated 9 times like in the first kernel. In this case, the loop in line 10, which gets repeated 9 times contributes to the number of operations since it contains an addition. The loops in lines 4 and 10 get both executed, like in the first kernel, 56 times. Thus, the dividend can be represented as $56 * (2 * 56 * 9 + 9)$ and the formula for I_{k2} as:

$$I_{k2} = \frac{56 * (2 * 56 * 9 + 9)}{(3 * 56 * 9) * p}$$

For performance measurement of both kernels, the simple-gemm benchmark of GemmForge was used. The benchmark generates a kernel function via `GemmGenerator` class of GemmForge for a given precision and matrix dimensions of the three matrices used in the kernel function. Precision and dimensions must be defined before compilation since the generation of the kernel gets invoked during the compilation process and if another precision or dimension is required, the benchmark must be recompiled. Additionally, the user can define, like in the bandwidth benchmark, the number of repeats for the kernel and the number of GBs, which determines depending on the used matrices the number of elements on which the kernel gets applied. The benchmark checks the correctness of the kernel function results by comparing them to a computation on CPU. After the correctness check, the kernel gets executed the user-defined number of times and averages measured performance over the number of repeats. The averaged results get printed in GFLOP/s. For the usage of the extended GemmForge library, the python file invocating `GemmGenerator` got extended with a new command-line argument that determines the manufacturer. The creation of the generated filename got modified via an if-else statement to differ between manufacturers and add the corresponding file ending. Additionally, the generated file gets the correct include statement inserted, depending on the manufacturer. The file containing the actual invocation of the kernel needed no editing due to the design of the benchmark. It only must include the generated header file, which contains the declaration of the generated kernel call function. The compilation process got extended with a separate HIP compilation path.
The benchmark got executed with 500 repeats and a number of GBs which correspond to 250000 elements, to get a reliable measurement for the Roofline model. The number of GBs must be changed depending on precision and the executed kernel. Both kernels got measured in single and double precision. The measurement was repeated 20 times for each configuration and the result averaged over the number of repeats. To achieve optimal performance, the kernel call was tweaked for each of the four configurations via the GPU properties defined in the `arch.py` file. This results in different y-dimensions in the wavefronts for each configuration, as depicted in table 2.

	Single precision	**Double precision**
1st kernel	1	16
2nd kernel	16	8

Table 2: The different y-dimensions of the wavefronts

With measured performances and arithmetic intensities calculated, the point x_k can be created for each configuration:

$$x_{k1-SP} = (I_{k1-SP} = 2.08, measuredPerformance = 1.339)$$
$$x_{k1-DP} = (I_{k1-DP} = 1.04, measuredPerformance = 0.619)$$
$$x_{k2-SP} = (I_{k2-SP} = 9.41, measuredPerformance = 3.989)$$
$$x_{k2-DP} = (I_{k2-DP} = 4.70, measuredPerformance = 1.851)$$

Combined with $g(x)$ created with the bandwidth test, these result into the Roofline model depicted in figure 5:

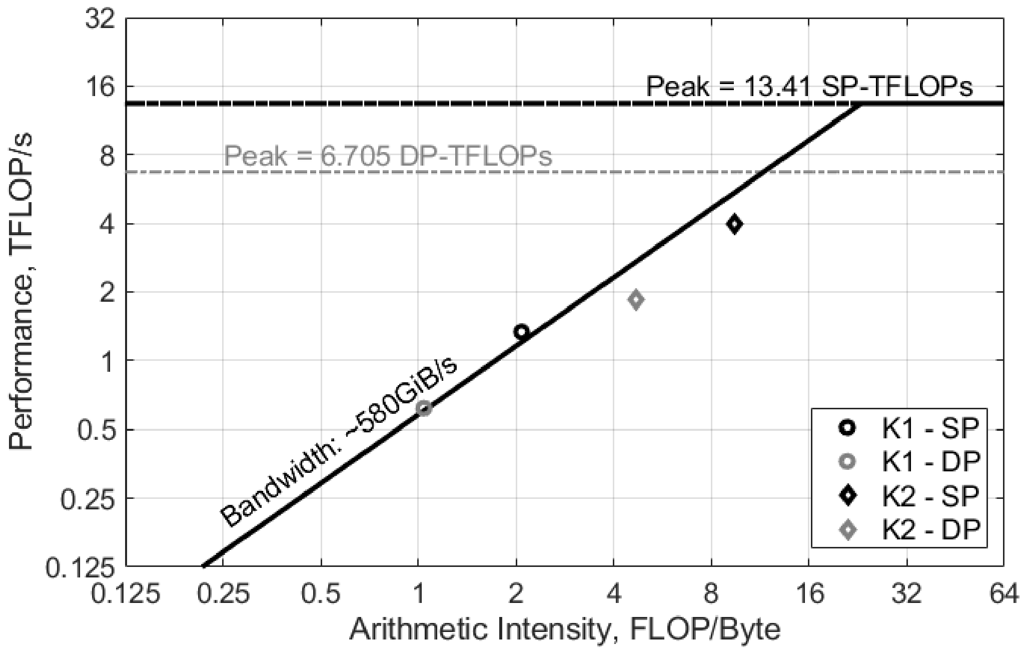

Figure 5: Roofline model analysis for the AMD Radeon Instinct MI50.

In comparison to the Roofline model predicted maximum performances the measured performances of the first kernels surpass the expectation. For single-precision, the measured average performance of $1.339\,\mathrm{TFLOP}/s$ is about 11% higher than the predicted maximum performance $g(I_{k1-SP}) = 1.2064\,TFLOP/s$. For double-precision, the difference is smaller. The measured performance is only 1% higher than the predicted $g(I_{k1-DP}) = 0.6032\,TFLOP/s$.

The second kernels however achieve a significantly lower performance than the predicted performance. Single precision performance is about 27% lower than the predicted $g(I_{k2-SP}) = 5.4578\,TFLOP/s$. The difference for double precision is even bigger with about 33% lower than $g(I_{k2-DP}) = 2.726\,TFLOP/s$.

4.1.3. Comparison to NVIDIAs Tesla V100

For the comparison between the manufacturers, a Roofline model analysis for NVIDIAs Tesla V100 with the same kernels is needed. Such an analysis was already performed in [5] and the measured values are used to create the Roofline model depicted in the following figure:

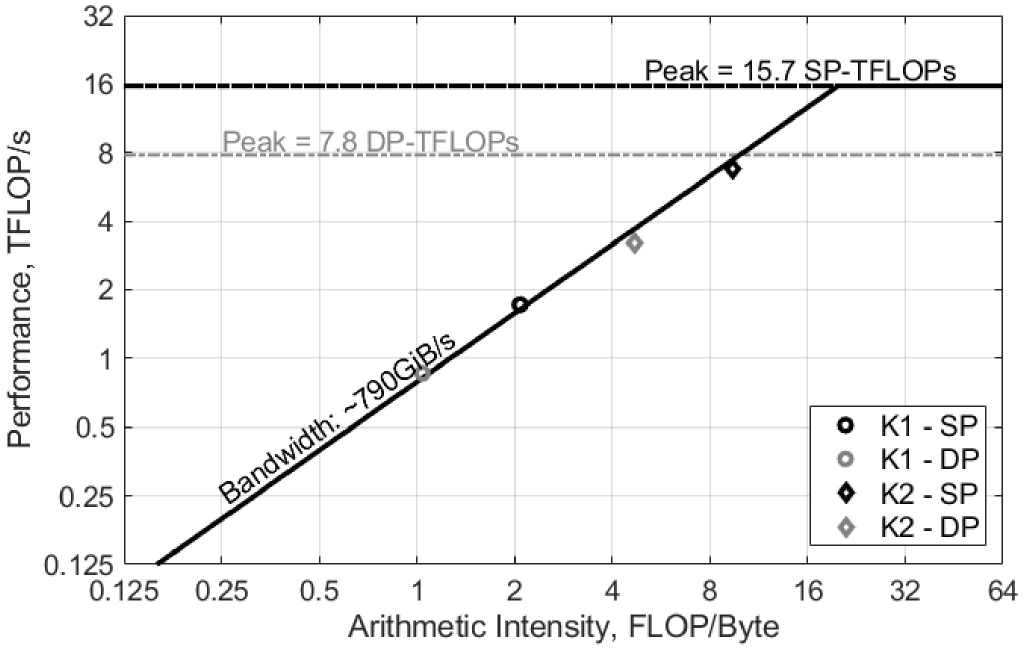

Figure 6: Roofline model analysis for the NVIDIA Tesla V100

As described in chapter 2.1.3, the maximum performance of the Tesla V100 is higher than the Radeon Instinct MI50s. But in both cases, maximum, achievable performance is limited by the measured bandwidth of the respective GPU since the arithmetic intensities of the kernels are located under the slope in the figures. Maximum theoretical performance is not important for the comparison of these results. The measured bandwidth of the Tesla V100 is significantly higher than Radeon Instinct MI50s although both have a similar theoretical maximum bandwidth. The NVIDIA GPU achieves about 69.7% of its theoretical performance, whereas the Radeon Instinct MI50 only achieves 56.6%. Thus, the kernels can achieve higher performance on the Tesla V100. This assumption is confirmed by the measured values in [5]:

$$x_{k1-SP} = (I_{k1-SP} = 2.08, measuredPerformance = 1.722)$$
$$x_{k1-DP} = (I_{k1-DP} = 1.04, measuredPerformance = 0.861)$$
$$x_{k2-SP} = (I_{k2-SP} = 9.41, measuredPerformance = 6.768)$$
$$x_{k2-DP} = (I_{k2-DP} = 4.70, measuredPerformance = 3.210)$$

Also, measured values for the second kernels are better compared to the predicted values of the Roofline model, they only have about 10% lower performance for single precision and 14% for double precision. In contrast, the second kernels on the Radeon Instinct MI50 show about 30% lower performance. The measurements of the first kernels have a similar ratio to the anticipated performance for both GPUs. The comparison shows that the kernels perform better on the Tesla V100. If the low bandwidth of the AMD GPU would be increased, the GPU may yield a comparable performance, since performance is limited by bandwidth in both cases.

4.2. Jacobi Solver

To ensure the correct functionality of the Device library, a Jacobi solver, as a good parallelizable algorithm, got implemented in the Device library in the scope of the thesis. The implementation uses OpenMPI combined with UCX to enable the distribution of the problem onto several processors. Thus, the Jacobi solver serves additionally as a benchmark for using multiple GPUs for computation. Like the Device library itself, Jacobi solver can be compiled for single or double precision. For compilation, the user must define the required precision, GPU manufacturer, and used GPU architecture. Furthermore, the solver can be compiled with or without MPI. During the compilation phase, both the CPU and the GPU solver get compiled and can be selected during the invocation of the program.

4.2.1. The Math

The Jacobi solver is an iterative method to approximate the solution of a linear system:

$$Ax = b$$

Hereby matrix A contains the coefficients of the linear system, whereas vector x contains the unknowns and b the solutions of the single terms. For approximation, matrix A gets decomposed into two matrices D and LU via $A = D + LU$. D is a diagonal matrix containing the diagonal entries of A and LU is a combination of the lower and upper triangle part of A. Computation of a single step can then be expressed as:

$$x^{k+1} = D^{-1} * \left(b - \left(LU * x^k\right)\right)$$

For the first step $k = 0$ the first solution x^0 must be guessed. The steps can be repeated a defined number of times or until a predefined discontinuation criterion is reached. For this implementation, the criterion is defined as:

$$b - \left(A * x^{\mathrm{k}}\right) < \varepsilon$$

This term describes that solution x^k of the current step, k is sufficiently close to the actual solution x, where maximum difference is defined via ε.

4.2.2. The CPU implementation

The solver got implemented in two steps: firstly, a version computing only on the CPU got implemented. This implementation got modified to use GPUs via kernel functions for parts of computation in the second step. The CPU implementation serves as a correctness comparison for the GPU port.
The implementation uses sparse ELLPACK form [35] for the representation of involved matrices. This form only saves non-zero entries of the matrix and corresponding indices in two vectors. Furthermore, additional information like the dimension of the matrix and the maximum number of non-zero entries per row is required for representing the original matrix. The advantages of the format are a smaller amount of used storage and a shorter iteration over elements since only non-zero entries get traversed. But the advantages only apply for sparse matrices, nearly filled matrices do not benefit from representation in sparse ELLPACK form. For this implementation of a Jacobi solver, only quadratic matrices get used. For parallelization, vectors and matrices get split up and equally distributed onto the single MPI processes. The distribution is shown in figure 7, where the first MPI process works on the first k entries of the vector and the next MPI process in the next portion. Finally, the last MPI process works on the last portion.

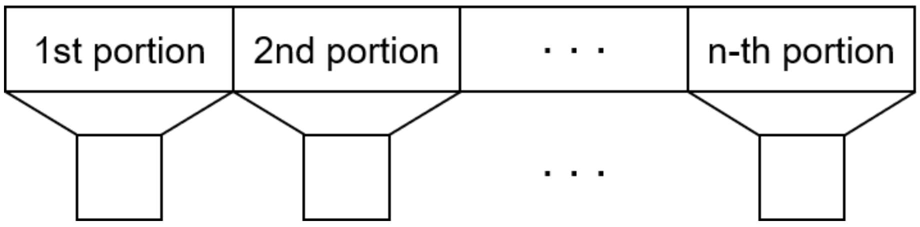

Figure 7: Distribution of problem domain onto MPI processes

The actual Jacobi solver using the CPU is implemented in the `solver` function of the namespace `host`. The function takes the involved matrix and vectors as parameters. Additionally, information for the controlling like the maximum number of repeats, the specified ε, etc. gets passed to the function. The function sets up the matrices D^{-1} and LU and starts the computation of the approximation. The computation is split into the three steps $tmp = LU * x^k$, $x^k = b - tmp$ and $x^{k+1} = D^{-1} * x^k$, where tmp represents a variable for storing the results of the first step. The steps are realized in two functions, one for the matrix-vector multiplication and one for the vector arithmetic. The third computation step get computed using vector manipulation function instead of matrix multiplication function, since the matrix D^{-1} can be represented as a vector, which contains the diagonal entries of D^{-1}. Since all other entries of D^{-1} are zero, no information is lost by representation as a vector. After completion of the three computation steps all entries of the new x^k gets collected from all MPI processes and redistributed between the MPI processes. This gets realized by the collective MPI operation `MPI_Allgatherv`. These steps get repeated until either a predefined maximum number of repeats is reached, or approximation is sufficiently close to the result.

To check the proximity of the current approximation x^k the program calculates every I steps the value of $r = b - \left(A * x^{\mathrm{k}}\right)$. The value of I must be set before the invocation of the computation. For the computation of r the function calculates $b - \left(A * x^{\mathrm{k}}\right)$ and takes the entry with the highest absolute value of the resulting vector as value for r. If $r < \varepsilon$, the computation can be stopped since the next step will not provide a significant improvement of the result. Furthermore, the program takes the opportunity to print out the number of the current iteration and the value of r to give the user an update about the progress. If the computation has stopped, the resulting x^{k} and the duration of the computation gets printed out to the user.

The solver can work without MPI if desired to measure the impact of the communication through MPI on the duration of the program. Therefore, the duration of the computation and the communication get separately measured.

4.2.3. The GPU port

For the port to the GPU, a new Jacobi solver got implemented in the `Solver` class in the namespace `gpu`. The GPU solver follows roughly the structure of the original solver with slight changes and additions. It relies on the same input parameters as the CPU version and follows the same distribution method for the problem domain shown in figure 7. Hereby each MPI process gets one GPU assigned, which executes the kernel functions for its MPI process. The access to GPU API functions gets realized with the Device library, whereas the invocation of kernel functions is handled by platform independently declared kernel call functions. Thus, only the implementation of the kernel itself and the definition of the corresponding call function must be implemented for NVIDIA and AMD, the remaining code can be used for both manufacturers.

The GPU solver starts the same way as the original with the setup of the matrices, but computation cannot be started immediately after setup since data must first be copied to GPU memory. This is handled by a separate function, which uses the Device library for a platform-independent code. But before the invocation of this function, the program must select the right GPU. For the selection, a Device function is used with the rank of the current MPI process as selection criteria. After completion of the copies, the solver can

be started. As in the CPU, version computation is dissected in three steps. These steps get realized in kernel functions, where a version of each kernel is implemented once for NVIDIA GPUs and once for AMD GPUs. The GPU parallelism is hereby realized by concurrent access on the vector entries by the GPU threads instead of a loop iteration over them. For the matrix-vector multiplication, concurrent access is applied to the rows of the matrix. For each kernel function, a kernel call function is implemented, which specifies the number of threads per block, or work items per wavefront in AMD terminology. To cover the whole vector and matrices, the appropriate number of those blocks or wavefronts get calculated and used during the launch of the kernel. Like in the CPU version the computed result gets collected and distributed between the MPI processes. Hereby the data gets directly shared between the involved GPUs. Likewise, the computation of r combined with an output of the value and the current number of iterations follows a predefined interval of repeats. In contrast to the original solver, the computation is realized mainly by using kernel functions. Only the selection of the highest absolute entry for the value of r gets executed on the host side.
After the computation is completed, which follow the same conditions as the original, the computed approximation x^{k} gets copied back to the host side. Afterward, allocated memory on the GPUs gets freed using a separate function, which uses Device functions for deallocation.

4.2.4. Test Results

Since this Jacobi solver is used for testing, it always runs with the same values for A,x^0 and the solution x for the linear equation:

$$A = \begin{bmatrix} 2 & -1 & 0 & & 0 & 0 & 0 \\ -1 & 2 & -1 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2 & & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & & 2 & -1 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 2 & -1 \\ 0 & 0 & 0 & & 0 & -1 & 2 \end{bmatrix} \quad x^0 = \begin{bmatrix} -10 \\ -10 \\ \vdots \\ -10 \end{bmatrix} \quad x = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

The vector b gets computed by $b = A * x$ at the start of the program. The user can define the number of rows for x^0, which automatically defines the dimensions of the quadratic matrix A. Furthermore, the solver type to be used, CPU or GPU, the maximum number of iterations, the number of iterations after which the output gets printed, and the value of ε must be defined in an input file. The program shows a correct behavior if the output vector x^k converges to vector x. If the used solver is compiled with MPI, the user must define the number of MPI processes during the invocation of the program. If the solver is compiled without MPI, the solver always runs with one process.

For our first tests, we chose 100000 as the maximum number of iterations and 10000 as the number of iterations for the intermediate output. The discontinuation criterion is set to $\varepsilon = 10^{-7}$. Only the number of rows got modified between the tests since parallelization of multiple MPI processes is performed on the rows as depicted in figure 9. Furthermore, GPU computation parallelizes the access to the rows too. Thus, with modifying the number of rows an impact of both parallelization types can be observed. For the tests, two versions of the Jacobi solver were used: one compiled without MPI and one compiled with MPI. Both versions got compiled for single precision. The comparison of both versions shows the impact of MPI on performance.

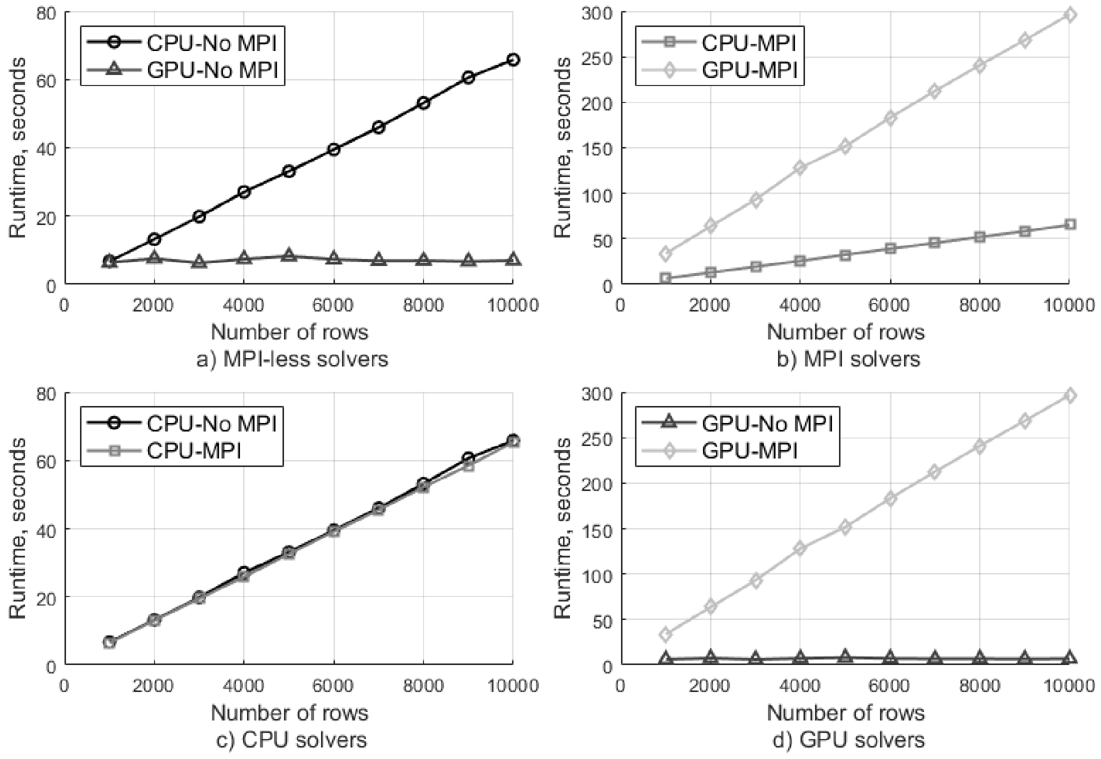

Figure 8: Runtime of Jacobi solver with one process on the Radeon Instinct MI50

Figure 8 shows measurements taken with an MPI and MPI-less version of the solver, arranged in four subfigures. Only one MPI process was used to keep the results comparable to the MPI-less solver since this only uses one process. OpenMPI was used as MPI implementation, using UCX for transportation of messages between MPI processes. The measurements were taken once with only CPU computing and once with GPU offloading. The used CPU is an AMD EPYC 7742 featuring 64 cores at 2.25 GHz base clock speed [36].

Figure 8 a) shows the performance gain of GPU offloading. The runtime increases proportional to the number of rows for the MPI-less CPU version, whereas the MPI-less GPU version delivers nearly constant runtime at about seven seconds for all numbers of rows tested. In figure 8 c) the runtime of CPU versions with and without MPI gets compared. The runtimes show insignificant differences for all test configurations, which is the desired behavior of MPI. Figure 8 b) compares the behavior of both MPI solvers. The GPU solver shows significantly worse performance than the CPU solver, which stands in contrast to the results of the MPI-less solvers depicted in figure 8 a). Furthermore, the MPI using GPU solver is about 42 times slower than the MPI-less GPU solver as depicted in figure 8 d). These bad results could indicate a problem in the combination of AMD GPUs and OpenMPI.

To further investigate the problem, the performance of computation and communication got separately analyzed. Additionally, these performances got measured on an NVIDIA GPU as a comparison baseline. Furthermore, a second AMD GPU got used to check if the problem is confined to the Radeon Instinct MI50. AMD's Instinct MI100, as the current model of AMDs Instinct series, features twice the number of CUs than the Radeon Instinct MI50 and thus twice the number of VU lanes [37]. They operate at a lower clock speed than the Radeon Instinct MI50, namely 1000 MHz base and 1502 MHz boosted clock speed [38]. The theoretical single-precision performance of the MI100 reaches up to 23.1 TFLOPs [37]. For this comparison, the number of rows was changed, the number of iterations and discontinuation criterion remained the same as in the first tests. Fewer tests with bigger steps in the number of rows were performed. The tests with NVIDIAs Tesla V100 and AMDs Instinct MI100 were, like the tests with the Radeon Instinct MI50, performed with the MPI implementation OpenMPI and UCX. The kernel got invocated with the same launch dimensions on all three GPUs. For the examination of computation and communication, the time of computation and communication got separately measured, and the number of rows divided by the measured time. If several GPUs are involved, the number of updates per second got averaged over the number of GPUs. The measurements were performed with the ROCm version 4.1.0 for the Radeon Instinct MI50 and version 4.0.0 for the Instinct MI100.

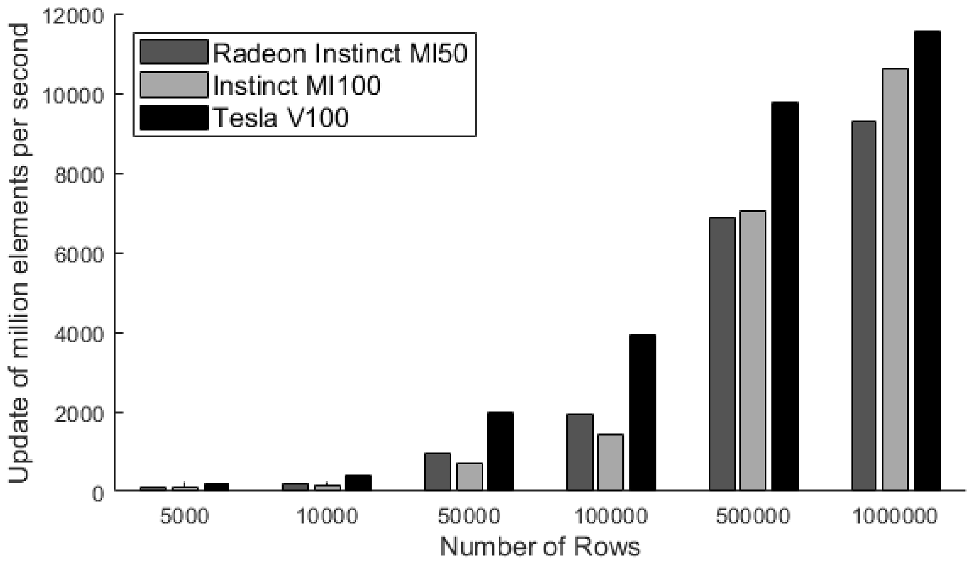

Figure 9: Average throughput of GPUs without MPI

Figure 9 shows the average compute throughput of the three GPUs with one process without MPI for different numbers of rows. The throughput increases with the problem size, which indicates that the GPUs perform more work in a similar time. A comparison of the AMD GPUs show, that the newer Instinct MI100 performs worse at small problem sizes, where the Radeon Instinct MI50 is up to 1.25 times faster. But at bigger problem sizes the newer GPU overtakes its predecessor. The Tesla V100 achieves higher throughput than both AMD GPUs, roughly 1.25 to 2 times the Radeon Instinct MI50. The different ratios of the results are depicted in table 3. These ratios serve as a baseline to which the computational ratios of the MPI solver can be compared to.

Number of Rows	**5000**	**10 000**	**50 000**	**100 000**	**500 000**	**1 000 000**
Tesla V100 to Radeon Instinct MI50	1.90	2.01	2.05	2.04	1.42	1.25
Tesla V100 to Instinct MI100	2.02	2.38	2.75	2.75	1.39	1.09
Instinct MI100 to Radeon Instinct MI50	0.94	0.84	0.75	0.74	1.03	1.14

Table 3: Throughput ratios for the GPUs without MPI

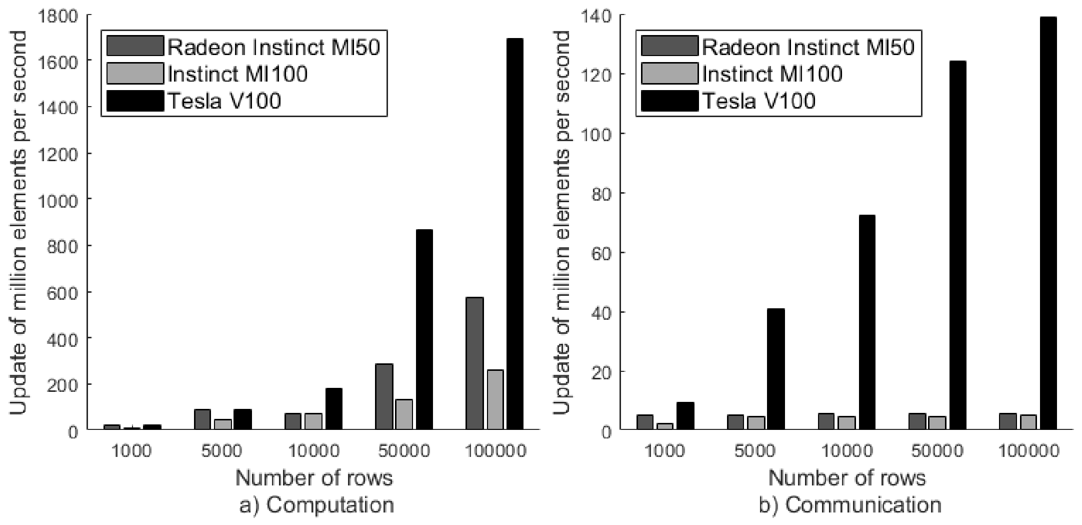

Figure 10: Throughput of GPUs with two MPI processes

Figure 10 depicts the average throughput per GPU for computation and communication measured while employing two GPUs via two MPI processes. The value for computational throughput decreases since the value gets averaged over the number of used GPUs and each GPU works only on half the number of rows.
Like in the MPI-less tests, the Instinct MI100 performs worse in computational throughput than its predecessor for the low number of rows. In almost all test configurations the Radeon Instinct MI50 is 2 times faster than its successor. Again, the NVIDIA GPU is faster than both AMD GPUs, the ratio is even bigger for the tested configurations, up to 3 times than the Radeon Instinct MI50. For the communication, however, the GPUs show different behavior. Whereas the communicational throughput increases with each step for the NVIDIA GPU, the AMD GPUs show a constant throughput regardless of the number of rows. Hence, the ratio increases between the NVIDIA GPU and an AMD GPU with each step, up to 26.14 times between the Tesla V100 and Radeon Instinct MI50. The ratios for all configurations are depicted in table 4.

	Number of Rows	1000	5000	10 000	50 000	100 000
Computation	**Tesla V100 to Radeon Instinct MI50**	0.81	1.04	2.52	3.02	2.97
	Tesla V100 to Instinct MI100	1.60	2.08	2.58	6.71	6.44
	Instinct MI100 to Radeon Instinct MI50	0.50	0.50	0.98	0.45	0.46
Communication	**Tesla V100 to Radeon Instinct MI50**	1.86	7.87	13.62	23.34	26.14
	Tesla V100 to Instinct MI100	4.08	9.12	15.31	26.08	29.04
	Instinct MI100 to Radeon Instinct MI50	0.46	0.86	0.89	0.90	0.90

Table 4: Throughput ratios for the GPUs with two MPI processes

Compared to the ratios of the MPI-less tests depicted in table 3, the computational ratios of the test depicted in table 4 are worse for the AMD GPUs but still close. The bad ratios of the AMD GPUs in communication however clearly indicate a problem in the communication between the GPUs via OpenMPI. Thus, the bad performance of the MPI using the Jacobi solver observed in the first tests was mainly caused by the communication.

To further investigate the communication problem, the collective communication used in the Jacobi solver gets compared to point-to-point MPI communication. Therefore, the OSU benchmarks for MPI get used [39]. Firstly, the OSU benchmark osu_allgatherv [39] covering the used MPI function `MPI_Allgatherv` was performed using the different GPUs. The benchmark measures latency for messages of increasing size between the communication participants [39]. A selection of the results is shown in table 5.

Message Size	Tesla V100	Radeon Instinct MI50	Instinct MI100
1	50.53	7.18	3.17
32	50.34	12.15	7.68
1024	51.46	135.27	115.85
32768	75.29	2807.71	1718.21
1048576	487.22	92183.18	53876.01

Table 5: Average Latency of osu_allgatherv in µs

These values show similar behavior to the Jacobi solver. For bigger message sizes the latency increases significantly for AMD GPUs, whereas the Tesla V100 shows only a moderate increase. At the biggest message size tested, the NVIDIA GPU has a 189.20 times lower latency than the Radeon Instinct MI50 and 110.58 times lower than the Instinct MI100. To check the latency of point-to-point communication, the OSU benchmark osu_latency was performed, which measures the one-way latency of the communication [39].

Message Size	Tesla V100	Radeon Instinct MI50	Instinct MI100
1	0.36	0.34	1.83
32	28.54	3.53	3.76
1024	29.38	9.13	54.18
32768	52.06	13.56	8.34
1048576	338.64	145.79	45.16

Table 6: Average Latency of osu_latency in µs

The measurements depicted in table 6 however show different behavior than the collective communication results. The AMD GPUs have lower latency than the NVIDIA GPU at bigger message sizes. The Radeon Instinct MI50s latency is 2.32 times lower, the Instinct MI100s 7.50 times. This behavior is reinforced by measurements taken with the osu_bw benchmark, which measures the bandwidth of point-to-point communication [39]. For communication, osu_bw uses non-blocking MPI functions [39].

Message Size	Tesla V100	Radeon Instinct MI50	Instinct MI100
1	0.06	0.45	0.74
32	2.02	11.59	9.65
1024	63.36	91.06	18.87
32768	1672.74	2868.30	3613.31
1048576	5079.01	7575.60	46859.91

Table 7: Measured point-to-point bandwidth using osu_bw in MB/s

The measurements of the benchmarks show that the communication problem is restricted to collective communication and does not affect point-to-point communication. An enquire to the AMD support regarding the different behaviors of collective and point-to-point communication with OpenMPI confirmed the observation. For point-to-point communication, OpenMPI can use the direct path between the AMD GPUs. But the collective communication via OpenMPI requires a feature, which is not yet implemented in the HIP library. Therefore, the messages get copied to the host side, which performs the collective communication and copies the results back to the GPUs. This intermediate step generates the observed latency. The required feature of HIP however is currently in development according to the AMD statement.

5. Conclusion and Future Work

In this thesis, we extended the libraries Device and GemmForge, which cover the GPU-related functionalities of SeisSol. The libraries are now compatible with AMD GPUs while retaining the ability to run on NVIDIA GPUs. For confirmation of the successful extension, we ran benchmarks for each library. Furthermore, these benchmarks evaluated the performance characteristics of an AMD GPU and served as a comparison to the existing NVIDIA implementation. Therefore, a comparable NVIDIA GPU, namely the Tesla V100, was used, which showed some advantages over AMD's Radeon Instinct MI50.

The Roofline model analysis showed that the generated kernels perform worse on the Radeon Instinct MI50 than on the Tesla V100, but the performance was limited by low memory bandwidth. Comparable performance on the AMD GPU may be achieved by a further investigation of the deviation of the measured bandwidth to the theoretical. Furthermore, the performance of the kernels can be improved by increasing the arithmetic intensity. This can be achieved by combining several kernels into one to increase the number of floating-point operations at a similar rate of moved bytes, as proposed in [5].

The Jacobi solver revealed a problem in the collective communication of AMD GPUs performed by using OpenMPI. However, the OSU benchmarks showed, that this problem is limited to collective communication and does not affect non-blocking point-to-point communication. This behavior got confirmed by AMD support. However, this communication problem does not affect SeisSol since the majority of MPI communication is realized by the unaffected non-blocking point-to-point communication.

For running earthquake simulations using SeisSol on AMD GPUs, the next generation of AMD GPUs may pose an opportunity. Based on information from multiple online sources, e.g. [40], the Instinct MI200 is going to feature managed memory, which is used by SeisSol running on NVIDIA GPUs.

Furthermore, the extension of both libraries poses an opportunity for further work. OneAPI[4], which includes the SYCL programming model [41], is such a candidate, which unites different accelerator architectures, including GPUs, under a single API.

[4] https://www.oneapi.com/

List of Figures

List of Tables

Bibliography

[1] TOP500: https://www.top500.org/statistics/list/ using “Accelerator/CP Family” as Category and the November 2020 release

[2] AMD: https://www.amd.com/en/products/exascale-era

[3] Oak Ridge National Laboratory: https://www.amd.com/en/products/exascale-era

[4] Hewlett Packard Enterprise: https://www.hpe.com/us/en/newsroom/press-release/2020/03/hpe-and-amd-power-complex-scientific-discovery-in-worlds-fastest-supercomputer-for-us-department-of-energys-doe-national-nuclear-security-administration-nnsa.html

[5] Ravil Dorozhinskii and Michael Bader. 2020. SeisSol on Distributed Multi-GPU Systems: CUDA Code Generation for the Modal Discontinuous Galerkin Method. *International Conference on High Performance Computing in Asia-Pacific Region.*

[6] Yuhsiang M. Tsai, Terry Cojean, Tobias Ribizel and Hartwig Anzt. 2020. Preparing Ginkgo for AMD GPUs – A Testimonial on Porting CUDA Code to HIP. *Parallel Processing Workshopseuro-par 2020 International Workshops, Warsaw, Poland, August 24–25, 2020*

[7] Cade Brown, Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. 2020. Design, Optimization, and Benchmarking of Dense Linear Algebra Algorithms on AMD GPUs. *2020 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2020*

[8] Yujiang Bi, Yi Xiao, WeiYi Guo, Ming Gong, Peng Sun, Shun Xu, and Yi-bo Yang. 2020. Lattice QCD GPU Inverters on ROCm Platform. *EPJ Web Conf., 245 (2020) 09008*

[9] TechPowerUp: https://www.techpowerup.com/gpu-specs/tesla-v100-sxm2-16-gb.c3471

[10] NVIDIA. 2017. NVIDIA Tesla V100 GPU Architecture. Retrieved from: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[11] NVIDIA. 2020. NVIDIA V100 Tensor Core GPU. Retrieved from: https://images.nvidia.com/ content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf

[12] IBM. 2018. IBM Power System AC922 Introduction and Technical Overview. Retrieved from: http://www.redbooks.ibm.com/redpapers/pdfs/redp5472.pdf

[13] Michael J. Flynn. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers, VOL. c-21, NO. 9*

[14] Zhe Jia, Marco Maggioni, Benjamin Staiger and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv:1804.06826*

[15] NVIDIA: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[16] TechPowerUp: https://www.techpowerup.com/gpu-specs/radeon-instinct-mi50.c3335

[17] AMD. 2020. AMD Radeon Instinct MI50. Retrieved from: https://www.amd.com/system/files/documents/radeon-instinct-mi50-datasheet.pdf

[18] Paul Bauman, Noel Chalmers, Nick Curtis, Chip Freitag, Joe Greathouse, Nicholas Malaya, Damon McDougall, Scott Moe, René van Oostrum and Noah Wolfe. 2019. Introduction to AMD GPU Programming with HIP. Retrieved from: https://www.olcf.ornl.gov/wp-content/uploads/2019/09/AMD_GPU_HIP_training_20190906.pdf

[19] AMD. 2020. “Vega” 7nm Instruction Set Architecture. Retrieved from: https://developer.amd.com/wp-content/resources/Vega_7nm_Shader_ISA.pdf

[20] Jason Sanders and Edward Kandrot. 2010. CUDA by Example. Addison Wesley

[21] NVIDIA. 2017. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Retrieved from: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[22] Michael Dumbser and Martin Käser. 2006. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes – II. The three-dimensional isotropic case. *Geophys. J. Int.* 167, 1 (2006), 319–336.

[23] Alexander Breuer, Alexander Heinecke and Michael Bader. 2016. Petascale Local Time Stepping for the ADER-DG Finite Element Method. *2016 IEEE International Parallel and Distributed Processing Symposium*

[24] Michael Dumbser, Martin Käser and Eleuterio F. Toro. 2006. An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes – V. Local time stepping and p-adaptivity. *Geophys. J. Int.* (2007) 171, 695–717

[25] Message Passing Interface Forum. 2015. MPI: A Message-Passing Interface Standard Version 3.1. Retrieved from: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[26] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar. 2015. UCX: An Open Source Framework for HPC Network APIs and Beyond. Retrieved from: https://www.openucx.org/wp-content/uploads/2015/09/ucx_hoti20151.pdf

[27] OpenUCX Documentation: https://www.openucx.org/documentation/

[28] UCX. https://openucx.github.io/ucx/api/v1.9/html/index.html

[29] UCX. https://openucx.github.io/ucx/api/v1.9/html/md__hpc_mtr_scrap_users _yosefe _ucx_docs_doxygen_design.html#UCP

[30] NVIDIA. https://docs.nvidia.com/nsight-visual-studio-edition/nvtx/index.html

[31] Refactoring Guru. https://refactoring.guru/design-patterns/facade

[32] Refactoring Guru. https://refactoring.guru/design-patterns/adapter

[33] Refactoring Guru. https://refactoring.guru/design-patterns/singleton

[34] Samuel Webb Williams, Andrew Waterman and David A. Patterson. 2008. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. Retrieved from: https://www2.eecs.berkeley.edu/Pubs /TechRpts/2008/EECS-2008-134.html

[35] https://people.math.sc.edu/Burkardt/data/sparse_ellpack/sparse_ellpack.html

[36] AMD. https://www.amd.com/en/products/cpu/amd-epyc-7742

[37] AMD. https://www.amd.com/en/products/server-accelerators/instinct-mi100

[38] TechPowerUp. https://www.techpowerup.com/gpu-specs/radeon-instinct-mi100.c3496

[39] Ohio State University. http://mvapich.cse.ohio-state.edu/benchmarks/

[40] Hardware Times. https://www.hardwaretimes.com/amd-instinct-mi200-in-h2-2021-4x-mi200-gpus-paired-per-epyc-trento-cpus-via-infinity-fabric-3-0/

[41] Intel. 2020. OneAPI Specification Release 1.0-rev-3. Retrieved from: https://spec.oneapi.com/versions/latest/oneAPI-spec.pdf