DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Performance Portability and Evaluation of Heterogeneous Components of SeisSol Targeted to Upcoming Intel HPC GPUs

Ludwig Kratzl

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Performance Portability and Evaluation of Heterogeneous Components of SeisSol Targeted to Upcoming Intel HPC GPUs

Portierung und Leistungsanalyse heterogener Komponenten von SeisSol für zukünftige Intel HPC GPUs

Author:	Ludwig Kratzl
Supervisor:	Prof. Dr. Michael Bader
Advisor:	M. Sc. Ravil Dorozhinskii
Submission Date:	15/07/2021

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

Garching, 15/07/2021 Ludwig Kratzl

Acknowledgments

I would like to pay my special regards to my supervisor Prof. Dr. Michael Bader and my advisor Ravil Dorozhinskii for their great guidance throughout this work. My sincere thanks also goes to three members of the Intel cooperation for their fast and productive support. I would also like to thank the Department for Geophysics at the Ludwig-Maximilian University for providing me access to the Heisenbug machine. Last, I want to thank my family and friends, especially my sister Franziska Kratzl, who has inspired and motivated me to study.

Abstract

For the first time in over 20 years, Intel is selling discrete graphics cards, including products for high-performance computing, scheduled for release in 2022. This thesis investigates programming models for the upcoming Intel GPUs and selects the Sycl standard, provided by oneAPI and hipSYCL, to port the heterogeneous components of SeisSol. The modules in question in SeisSol are analyzed and extended, and their efficiency is assessed using a Roofline Model analysis. Similar experiments are performed using Sycl with CUDA as a backend to test whether this combination can replace all native GPU approaches. This work demonstrates that already the low-power series of Intel's discrete GPUs can reach over 85 % of their peak performance using this portability, making them superior to integrated graphics chips. However, further measurements also show that Intel MPI needs to be improved before it can be used on supercomputers. With an Nvidia RTX 3090 and Intel Arria 10 FPGA, it is proven that Sycl's cross-platform approach works but that its efficiency heavily depends on the implementation of the standard.

Kurzfassung

Erstmals seit über 20 Jahren verkauft Intel diskrete Grafikkarten, darunter auch Produkte für High-Performance Computing, die 2022 erscheinen sollen. Die vorliegende Arbeit untersucht Programmiermodelle für die erscheinenden Intel-GPUs und selektiert den Sycl-Standard, bereitgestellt durch oneAPI und hipSYCL, für die Portierung von SeisSols heterogenen Komponenten. Die betreffenden Module in SeisSol werden analysiert und erweitert und die Effizienz der Portierung mithilfe eines Roofline-Diagramms durchgeführt. Ähnliche Experimente werden mit Sycl und CUDA als Backend durchgeführt, um zu prüfen, ob diese Kombination alle nativen GPU-Ansätze ersetzen kann. Diese Arbeit demonstriert, dass schon die Low-Power Serie von Intels diskreten GPUs an 85 % ihrer Spitzenleistung mit der Portierung herankommt und damit den integrierten Grafikchips überlegen ist. Allerdings zeigen weitere Messungen auch, dass Intel MPI vor dem Einsatz auf Supercomputern verbessert werden muss. Mit einer Nvidia RTX 3090 und Intel Arria 10 FPGA wird belegt, dass Sycls Crossplattform Ansatz funktioniert, aber dessen Effizienz stark von der Implementierung des Standards abhängt.

Contents

1. Introduction

In 2019, Intel announced the 12th generation of its GPUs, including the first discrete Graphics Cards for over twenty years. The new series of products, Intel Iris Xe (X^e), contains a revisited Instruction Set Architecture as well as models for office applications with low power consumption (Xe-LP), computer games, and general high-performance (Xe-HP and Xe-HPG), and HPC (Xe-HPC). While the Xe-LPs are already available by the Iris Xe Max "DG1", Intel did not release representatives of the other GPUs by the time of this thesis. However, the Leibniz Rechenzentrum (LRZ) recently announced to convey its SuperMUC-NG into a new phase[1]: To fit the needs of modern AI computations, they are going to expand the 15th fastest supercomputer in the world with the Iris Xe-HPC GPU called "Ponte Vecchio". This announcement could change future HPC architectures, as until now, most HPC centers rely on Nvidia GPUs: As described in Figure 1.1, from the top 500 supercomputers worldwide, 147 of them used accelerators or co-processors in November 2020, of which over 95 % were coming from the Nvidia product stack. Consequently, HPC applications are often developed with CUDA, the native language and API for Nvidia GPUs, which does not apply to Intel. Therefore, LRZ's breakthrough decision comes with the price that any CUDA code must be ported before running it on its new cluster. This especially concerns strategic partners of the LRZ like the Technical University of Munich (TUM), which uses the SuperMUC-NG as its base supercomputer.

One affected application will be SeisSol, an open-source software for numerical simulations of earthquakes and seismic waves. SeisSol was initially designed with an MPI+OpenMP model, targeted to computations on CPUs and accelerators like Co-Processors [1]. However, recent changes in SeisSol took a step towards Performance Portability by substituting OpenMP with CUDA, resulting in a doubled speedup by exploiting native programming features like shared memory and launch bounds [2]. Similar work has already been preceded with HIP, the programming language and toolkit for AMD graphics cards [3]. Nevertheless, neither HIP nor CUDA can be employed to target Intel GPUs. Therefore, SeisSol is not tunable to the future Intel GPUs running in LRZ or other supercomputers, as OpenMP only supports limited capabilities to access the offload device.

This thesis analyzes existing heterogeneous components of SeisSol and applies all necessary changes to adapt them to Intel GPUs. The work is structured as follows:

Chapter 2 introduces the topic by first summarizing the known hardware details of the existing and upcoming Intel GPUs. After that, available programming models for them are discussed and it is shown that Sycl, a cross-platform standard to program accelerators, fits best for portability. Additionally, an overview of available implementations of Sycl is given,

[1]`https://www.lrz.de/presse/ereignisse/2021-05-04-SuperMUC-NG-Phase-2_ENG/`

and hipSYCL and oneAPI are selected for compilation and performance evaluation. Next, the numerical scheme of SeisSol, based on a high-order Discontinuous Galerkin method and its implementation [4] solving the underlying system of Hyperbolic Partial Differential Equations, is summarized. With that, the necessity for high-performance parallelism and fine-tuned GPU kernels that justify the employment of native or low-level programming concepts like Sycl over OpenMP is explained. Finally, the term Performance Portability is discussed, and a Roofline Model analysis is chosen as success criteria for the main compute kernels.

Chapter 3 examines three submodules for heterogeneous access: Device, a facade for accelerator accesses, Yateto, providing a DSL to express tensor operations, and GemmForge, a code generator for batched General Matrix Multiplications (GEMMs), which implements the DSL of Yateto. It explains changes that were crucial in order to add support to the Sycl standard and focuses on how these modifications apply to use Sycl not only on Intel GPUs but also on FPGAs and arbitrary accelerators.

Chapter 4 evaluates Performance Portability by various benchmarks: At first, a parallel Jacobi Relaxation solver is performed on Intel's DevCloud with one and two Iris Xe Max "DG1" GPUs using Intel MPI. This shows that GPU-aware Intel MPI is available and is suitable for SeisSol, however needs further optimization before productive utilization. At the same time, the chapter demonstrates that DG1 is already able to get close to peak performance on dense GEMM operations and shows its superiority against SoC designs like the 9th Generation UHD Graphics P630. Running SeisSol proxy implementation and two benchmarks establishes a proof-of-concept of all changes and proofs that SeisSol will be able to run on the new SuperMUC-NG. Furthermore, explorative performance analyses using Intel Arria 10 FPGAs and two Nvidia RTX 3090 GPUs are performed, showing that the Sycl standard could replace all native GPU code in the future from a functional perspective, but not from a performance one. At last, this thesis ends with a summary and conclusion and describes future work which should be done once the new Intel GPUs are available.

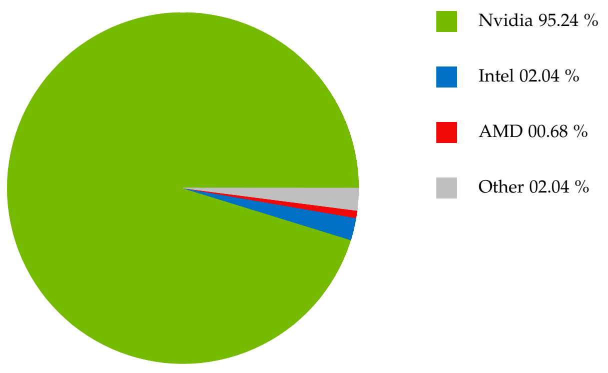


Figure 1.1.: Market share of vendors of accelerators within the top 500 supercomputers. Total count of centers using GPUs or Co-processors: 147. Nvidia is significantly more often in use (140 of 147). Data extracted from `https://www.top500.org/statistics/` using the ranking from November 2020. Accessed 05/21.

2. Fundamentals

This chapter summarizes fundamentals necessary to port SeisSol for the new Intel GPUs. First, Intel's latest integrated graphics chip (Gen11) and parts of its performance characteristics are examined. The following subsection accomplishes the same for all known details of Iris Xe series and compares these two generations among each other. Then, available programming models and compilers for any Intel GPU are discussed, and Sycl, together with oneAPI and hipSYCL, is selected for the remaining parts of this thesis. The following section dives deeper into heterogeneous components of SeisSol, mainly Device, Yateto, and GemmForge: It explains their structure and implementation details and justifies them with the help of SeisSol's Numerical Scheme, which is established in the same section. With this in mind, methods to evaluate all changes in terms of performance are analyzed, and Roofline Model analysis, in combination with several benchmarks and tests, is chosen as success criteria.

2.1. Intel GPUs

Contrary to what is popular assumed, Intel has been developing GPUs for over twenty years. For example, they released their first own discrete Graphics Card in 1998 with the i740 and i752, only three years after Nvidia but without success. However, they continued their work towards that: The Extreme Graphics and GMA series were mainly used as chipsets on a motherboard independent of other components like a CPU, resulting in limited performance aspects. With Intel HD Graphics (Generation 5 and later), Intel switched to a System on a Chip (SoC) architecture. From then, Intel processors did not only contain CPUs with several cores but also a GPU. This design led to a huge triumph in the GPU market: Though Figure 1.1 demonstrates the dominance of Nvidia in terms of HPC, Figure 2.1 shows the victory of the SoC architecture in the consumer market. In 2020, nearly 70% of all GPUs were Intel products (however, note that this is different when only considering discrete GPUs). In 2010, Intel again made a step towards discrete accelerators with the Xeon Phi manycore architecture. Initially inspired by previous GPU researches, this design has now been canceled in favor of Iris Xe series. The following subsections examine the latest GPU series, Gen11, and compare its characteristics with known details of the upcoming Iris Xe product stack.

2.1.1. Gen11 Architecture

All investigations in this subsection are based on Intel architectural white papers [5] [6], if not otherwise indicated. Figure 2.2 gives a graphical overview of the subsequent explanations.

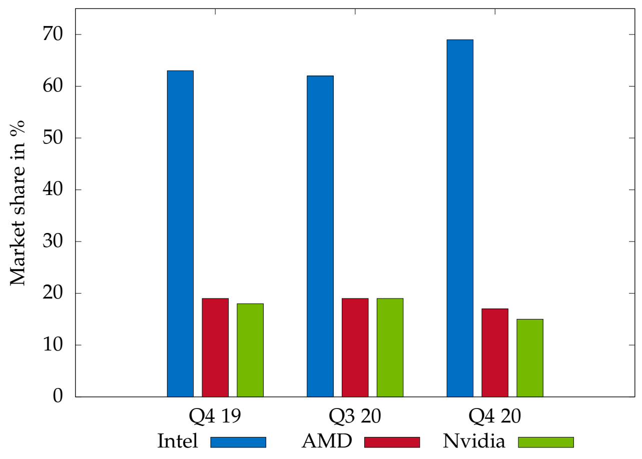


Figure 2.1: Market share of vendors of overall installed GPUs. Intel's proportion is notable greater than AMD's or Nvidia's. Data extracted from `https://www.jonpeddie.com/press-releases/gpu-shipments-soar-once-more-in-q4`. Accessed 05/21.

Gen11 architecture structures itself, similar to its predecessor Gen9, in a single slice connected via a Ring Bus with CPU Cores and a coherent shared Last-Level-Cache (LLC). Each slice contains an L3 cache and slice-common components as well as eight subslices.

A subslice clusters eight execution units together with an L1 and L2 texture sampler cache and a shared local memory (SLM) of 64 KB. Here, Gen9 differs from Gen11: The SLM of Gen9 is attached outside of the subslice. Also, all subslices contain a dataport, a general-purpose memory unit for load and store operations with 64 Byte read and write bandwidth per second, and a thread dispatcher that assigns threads to execution units.

An Execution Unit (EU) is the main basic block of an Intel GPU. It is a computational unit consisting of two SIMD ALUs pipelined across seven available hardware threads. The ALUs have a vector width of 128 Bit, allowing up to four FP32 (32-bit floating point) or eight FP16 (16-bit floating point) operations simultaneously. With Fused Multiply Add (FMA) commands, a single EU can reach up to 16 FP32 or 32 FP16 operations per cycle: 2 (ALUs) $\cdot$ (4 or 8 SIMD-width) $\cdot$ 2 (FMA) $=$ 16 or 32 operations. Therefore, a complete GPU results in a theoretical peak performance of $8 \cdot 8 \cdot 16, 32 = 1024$ FP32 or 2048 FP16 operations per cycle, respectively.

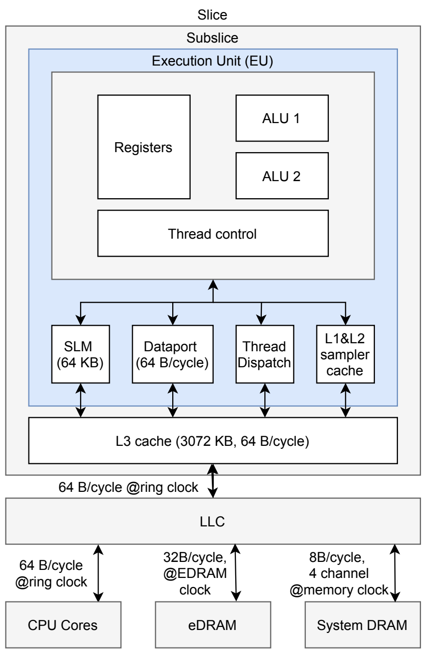


Figure 2.2.: Schematic structure of the Intel Gen11 architecture: A slice groups into subslices that cluster several execution units (EUs). The GPU is connected with the CPU by a shared LLC

The memory bandwidth, on the other hand, is limited to its hierarchy. However, L1 and L2 sampler caches become only active in texture or image surface sampling, which is typically not relevant for high-performance computations. Therefore, the bandwidth of the GPU is initially constrained by the 3 KB L3 cache of a subslice. Next follows the LLC (3 MB) shared by CPU and GPU and an optional embedded DRAM (eDRAM), serving as an additional cache level. Last, the connection of LLC to the system's memory control limits the memory bandwidth of all caches: For Gen11, this is 8 B per cycle on four channels, resulting in a theoretical bandwidth of system memory frequency times 4 times 8 Byte.

2.1.2. Iris Xe Architecture

The Iris Xe series is an iteration of Intel's Gen12 architecture. This subsection summarizes architectural details of the new Xe series based on the announced products of the LP series and an architectural overview shown at the Architecture Day 2020 [7]. One of the main benefits of the new architectural style are so-called tiles. Every tile resembles a single Iris Xe, that are together interconnected via a so-called XeLink. This feature is crucial to compute clusters like considered in this work. The following explanations show how the other known specifications apply to the four Xe sub-architectures.

- **Iris Xe LP**: By the time of writing, LP is the only released series. Intel Iris Xe Graphics G7 follows the SoC design, whereas Iris Xe Max DG1 (Discrete Graphics 1) is the standalone version equipped with 4 GB of memory. Both have a single slice and no additional tiles. One main difference between these GPUs and Gen11 is the number of subslices and execution units. In the case of Iris Xe LP, six subslices contain 16 EUs on each slice, whereas in Gen11, there are 8 for both. Hence, the minimum number of total EUs is 96 compared to 64 per slice. Using a boost clock of 1650 MHz and a memory clock of 2133 MHz, these two GPUs reach a theoretical peak bandwidth of 68.3 GB/s or 2.5 FP32 OP/s. Iris Xe Max SG1 (Server Graphics 1) is another model based on the aforementioned specifications but containing 8 GB of onboard DDR4 memory.

- **Iris XE HPG**: Iris Xe HPG DG2 is an improved variant of DG1 announced for the end of 2021. Internet leaks[1] reveal variants with 128, 256, and 512 EUs. It can be speculated that these high counts of EUs get achieved by multiple slices within a GPU. Assuming the same clock, ALU count, and SIMD width as with the LP series, 3.4 to 13.5 TFlop/s are reachable. However, other leaks state a clock speed of 2.2 GHz, making the DG2 a direct competitor to Nvidia's RTX 3070 in terms of Flop/s.

- **Iris XE HP**: The HP series comes with up to four tiles that could scale performance linearly with the peak one of XE HPG products. Apart from that, no details are known at the time of writing.

- **Iris XE HPC**: A GPU series with its codename "Ponte Vecchio" is designed to run on supercomputers like SuperMUC-NG (compare Chapter 1) or Aurora. The first details were announced at the HPC Devcon 2019[2]: 16 compute tiles distributed across two blocks combine Rambo cache and I/O link tiles. A total number of EUs was not unveiled so far. Therefore, no assumptions about the ultimate peak performance are viable.

[1] `https://www.tomshardware.com/news/intel-dg2-gpu-specifications-show-serious-potential`
[2] `https://www.intel.com/content/www/us/en/events/hpcdevcon/overview.html`

2.1.3. Intel GPUs in HPC

Today, Intel GPUs are not widespread in HPC applications because of their limited performance characteristics. However, there is still research on them. For example, Gera et al. investigate the performance characteristics of Gen9 architecture, showing that these GPUs can reach their peak performance [8]. Deakin and McIntosh-Smith examine Iris Pro 580 Graphics (Gen9) in combination with Sycl [9]. They demonstrate that the peak bandwidth and execution time is equal to OpenCL. Additionally, they show that Sycl can also produce competitive performance compared to native performance models. Li et al. compare the Xeon Phi architecture with an Nvidia Tesla GPU, however, without accessing an integrated graphics chip [10]. Similar has been performed by Theodoro et al. [11]. Christgau and Steinke, on the other hand, perform a closely related work to this thesis: They port their tsunami simulation easyWave from CUDA to Sycl using Intel's compatibility tool, showing acceptable Performance Portability, but also areas for improvements of oneAPI [12]. To the best of the author's knowledge, no other investigations on the available Iris Xe products have been carried out in the area of HPC so far.

2.2. Intel GPU Programming

This section discusses programming models for Intel GPUs and compares their syntax by a Saxpy (Single-Precision A·X Plus Y) example and emphasizes features or missing concepts with help of the literature. Starting with the most widely used, OpenMP, it continues with OpenCL, Intel's direct programming concept so far, and ends with the Sycl standard. Its implementations provided by oneAPI and hipSYCL are then selected for the rest of this work.

2.2.1. OpenMP

OpenMP is a standardized language extension for C/C++ and Fortran and allows quick parallelization by parallel regions, work-sharing constructs, or task-based thread scheduling. It is assumed that most readers are familiar with the general concepts of OpenMP. Therefore, the next paragraph focuses on GPU offloading.

Starting from version 4.0, OpenMP can be used to offload code to non-host devices like FPGAs and GPUs with the help of target regions, which are assignable to a device indexed by the OpenMP runtime. Parallel regions within a target region can be nested into teams that create a kernel-like hierarchization into blocks and threads similar to CUDA. However, a runtime system is responsible for distributing work between compute units. There exists a broad variety of compilers, for instance, GCC, Intel CC, or LLVM with Clang, which support OpenMP offloading, making OpenMP highly portable. Appendix A.1 shows a Saxpy benchmark implemented with OpenMP offload, including data mapping of host to device pointers and vice versa. In direct comparison to Sycl or OpenCL, OpenMP can achieve better performance results [9]. Additionally, it could already get close to the peak performance with Intel's Xeon Phi series at an early stage of the offload support [13]. On the other hand, OpenMP can also create higher runtimes in several applications than their native counterparts,

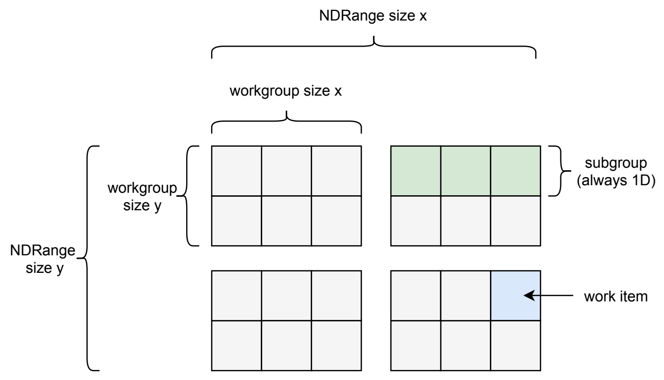


Figure 2.3: Example for OpenCL and Sycl kernel indexing. A 2D range of (6,4) groups into four (3,2) workgroups. Each workgroup consists of two subgroups (green) of length three, containing the smallest component, a work item (blue). Note that OpenCL defines global and local ranges for their kernels instead of local block counts and their thread sizes as in CUDA or HIP.

like CUDA [14], and require deep low-level analysis and custom features to tune a code for a specific platform via local shared memory, FMA operations, or register bounds [15]. Because of this and the fact that SeisSol requires these features to minimize the execution time spent on numerical macro kernels, the usage of OpenMP is declined for Intel GPUs in SeisSol.

2.2.2. OpenCL

OpenCL is a cross-platform standard developed by the Khronos Group primarily used for parallel or heterogeneous systems. By the time of writing, version 3.0 is the latest release. In the following, only crucial ideas of OpenCL C are summarized, as the rest of this thesis dives deeper into its concepts with the Sycl abstraction of OpenCL. Further readings are available in the standard [16] or tutorials [17] [18]. The OpenCL standard consists of four models: The platform, memory, execution, and programming model.

The platform model defines a host device (typically a CPU) that is always provided by the runtime. Hence, there is at least one execution environment guaranteed to be available. Additionally, the host acts as a master thread, connecting multiple OpenCL devices that queue so-called commands or command groups. A command, on the other hand, is a small sub-program, mostly called a *kernel*, that gets executed by one of the available devices.

The execution model assigns an n-dimensional index space to a kernel, closely related to blocks and threads in CUDA or teams and threads in OpenMP. This hierarchical structure is called an NDRange (N-dimensional range) and is used to distribute work to hardware threads of a device. For common OpenCL devices like GPUs, N is limited to three dimensions. An NDRange structures itself into workgroups (blocks/teams) consisting of work items (threads) performing the actual work. Therefore, the global NDRange equals the count of workgroups times the size of each one. Figure 2.2.2 illustrates this idea using a 2D range of size (6,4). There is also an intermediate 1D structure between workgroups and work items, called subgroups. Subgroups were introduced in version 2.0 of the standard and are used to cluster work items and assign them to a single hardware thread (comparable to a SIMD lane). However, as they do not impact this work, further readings are referred to [19].

The memory model defines three main types of memory: Private memory is only visible to a single work item, whereas local memory gets shared within a workgroup. Finally, there is global memory accessible by a complete NDRange. For Intel GPUs, these memories are mapped with the same memory available in the hardware (compare Figure 2.2 in the last section). OpenCL also allows declaring global memory as so-called unified shared or host memory. Unified shared memory is managed by the runtime and gets automatically transferred between host and device via page fault mechanisms when accessed. On the contrary, host memory follows the same idea, but data is transferred via a PCI-E bus and does not migrate back from the device.

A combination of the discussed three models results in the OpenCL programming model available by the OpenCL framework. Traditionally, this model is expressed with the C programming language, but a C++ extension is also available [20]. Various vendors provide OpenCL for their products, including Nvidia and Intel, resulting in good portability. Additionally, Fang et al. showed that OpenCL gets similar performance results as CUDA under fair condition [21]. Nevertheless, OpenCL is not used to port SeisSol for Intel GPUs for three reasons: Firstly, OpenCL C comes with a large overhead of boilerplate code, which makes dynamic code generation cumbersome. That issue is furthermore emphasized by the necessity to separate kernel and host code. Due to the length of an OpenCL code, no example is provided in the appendix of this work, but it is referred to the book of Ravishekhar and Bhattacharyya, where Saxpy is implemented in Chapter 1 [18] and stays exemplary for these concerns. Secondly, there is a modern C++17 programming abstraction to OpenCL called Sycl that recently received new attention with Intel's oneAPI platform. oneAPI contains a Sycl compiler called *data-parallel c++* (dpcpp), available in both a packaged and open-source variant, that already received good results while porting CUDA code to Intel's own products [12]. Therefore, it is believed that the performance gap between Sycl and OpenCL or OpenMP, like reported by Silvia et al. [22], will minimize or not apply to Intel's software stack in the future. Thirdly, Sycl today allows different backends than OpenCL, for example, CUDA, which allows to create native applications.

2.2.3. Sycl

Sycl [23] is another Khronos standard, initially designed on top of the OpenCL backend. However, as mentioned in the section before, it also allows different backends like CUDA, increasing portability by reducing dependencies on OpenCL. Sycl is a fully C++17 conform standard, which means that no additional syntactical extension is needed to create and launch a kernel, but only Sycl libraries. With its single-source approach (device and host code are mixable), Sycl can highly reduce lines of code within a program, speed up development time, and improve the readability of heterogeneous code [24]. Due to its relation to OpenCL, Sycl follows the same models. Thus, all explanations of platform, execution, and memory from the previous section also apply to Sycl. Obviously, the programming model deviates as OpenCL C can't access C++ features like lambdas or template functions. Listing A.2 implements a Saxpy benchmark using Sycl and raw C pointers. In direct comparison to OpenCL, the brevity and conciseness of the Sycl code can be highlighted, assuming the reader is familiar with C++ lambda expressions. If not, introductory works to Sycl help to explain this part in detail [25]. By the time of writing, there exist at least four Sycl implementations, including Codeplay's

ComputeC++, Xilinx's TriSYCL, hipSYCL provided by the University of Heidelberg [26], and Intel's oneAPI. For this thesis, hipSYCL and oneAPI are selected as development platforms for three reasons: Firstly, both are open-source available and installable on HPC machines without additional configuration. For hipSYCL, there is also a spack package available, adding more convenience to test setups like used in this work. Spack is an HPC package manager, which installs an application and all its dependencies from source code, which is important as the targeted test systems do not allow changes without administrative access. Secondly, both of them support CUDA as code backend, which is used to ensure the correctness of SeisSol by comparing the results obtained with Sycl with the native CUDA ones. Thirdly, and most importantly, it is expected that Intel's Sycl compiler is highly tuned for Intel devices, and therefore will be available on all HPC systems that employ Intel GPUs like the "Ponte Vecchio". Hence, evaluations using oneAPI should create an outlook on what is expectable in the future in terms of performance results.

2.3. SeisSol

SeisSol is an open-source "scientific software for the numerical simulation of seismic wave phenomena and earthquake dynamics"[3]. An example of a real-world application of SeisSol is the 1994 Northbridge or the 2004 Sumatra-Andaman earthquake. This section introduces SeisSol and its heterogeneous components by providing an overview of its numerical scheme build upon a High-Order Discontinuous Galerkin (DG) method using ADER time integration and Local or Global Time Stepping (LTS/GTS). Hence, the necessity of small GEMM kernels generated during compilation is justified. The subsequent section provides a big picture of SeisSol's heterogeneous components distributed to three libraries, namely, Device, Yateto, and GemmForge that are analyzed and extended in Chapter 3 with Sycl compliant code. The last section defines the term Performance Portability and how it is applied in this thesis.

2.3.1. Numerical Scheme

SeisSol provides a numerical solver for the three-dimensional elastic wave equation driven by velocities and stresses. This can be modeled by a system of linear hyperbolic Partial Differential Equations (PDEs) as expressed in Equation 2.1.

$$\frac{\partial q}{\partial t} + A(s)\frac{\partial q}{\partial x} + B(s)\frac{\partial q}{\partial y} + C(s)\frac{\partial q}{\partial z} = 0 \tag{2.1}$$

$q(s,t) = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{yx}, \sigma_{xz}, u, v, w)$ is a nine-component vector containing stresses σ_{ij} and particle velocities u, v, w in the Cartesian space $s = (x, y, z) \in \mathbb{R}^3$. A stress σ_{ij} determines the force on a plane i within a body in direction j. A, B, C are space-dependent Jacobi matrices adding the Lamé parameters λ and σ as well as the mass density ρ. The Lamé parameters are used to define the relationship between stress and strain in elastic deformation.

[3]`http://www.seissol.org/`

SeisSol uses a Discontinuous Galerkin (DG) method for spatial discretization. In the DG method, 2.1 is transformed into its weak form by multiplying it with test functions Φ and integrating it over the domain Ω. The result is then semi-discretized into finite elements Ω_k that present a new state in a time step and are linked by a numerical flux. SeisSol applies an adaptive grid of tetrahedral meshes, allowing to vary the mesh size depending on the importance of geological regions. Additionally, SeisSol combines the DG method with the ADER scheme for time discretization as described by Käser and Dumbser [27] [28] which leads to a high-order representation of $\hat{Q}$ at each time step t_n of tetrahedron Ω_k shown in Equation 2.2. Note that the following descriptions are based and summarized on the work of Heinecke et al. [1].

$$\hat{Q}_k^n(s) = (\phi_1, ..., \phi_{B_O})(\xi(s))Q_k^n \tag{2.2}$$

This equation combines orthogonal polynomial basis functions $\phi_j(\xi), j \in 1, \ldots, B_O$, which are space-dependent, and time-dependent degrees of freedom $Q_k^n \in \mathbb{R}^{B_O \times 9}$. The count of bases B_O is determined by the convergence order O. A typical size for O is six, resulting in $B_6 = 56$ basis functions. Therefore, this characteristic size is used later in the benchmarks for a Roofline Model analysis. ξ is a translation of an arbitrary Cartesian coordinate $s = (x, y, z)$ into a reference point, here a tetrahedron of a mesh. The fully discretized update scheme in SeisSol for a degree of freedom at time $t_{n+1} = t_n + \Delta t$ based on Q_k^n is provided in Equation 2.3.

$$Q_k^{n+1} = Q_k^n + V_k - \sum_{i=1}^{4} F_{k,i} \tag{2.3}$$

Note that if the time steps Δt are equally for every tetrahedron, one speaks of Global Stepping (GTS), but also individual values are allowed (Local Time Stepping, LTS). The partitioning of tetrahedrons into equivalent time steps is one of the key ideas of SeisSol

The heart of the update scheme form *Compute Kernels* denoted as V, F, consisting of tensor operations and depending on time $T_k^{n,\Delta t}$. That explains and justifies the focus of this work on optimizing that part of SeisSol's numerical scheme, as these kernels are executed enormously often, depending on the simulated time and count of tetrahedrons in the mesh. Some of them are illuminated in the following to emphasize these considerations. More detailed explanations can be found in previous work [29].

$T_k^{n,\Delta t}$ describes the *Time Kernel* and approximates time-predicted values of the degrees of freedom in $[t_n, t_{n+1}]$ of a tetrahedron k by a Taylor expansion. The time derivatives of this expansion are expressed by a recursive scheme using multiplication of stiffness matrices, mass matrices, the previous result of the recursion, and a linear combination $\in \mathbb{R}^{9 \times 9}$ of A, B, C.

V_k describes the *Volume Kernel* depending on $T_k^{n,\Delta t}$, following a similar scheme as the time derivatives of it.

$F^{k,i}$ expresses the numerical *Flux Kernel* for a tetrahedron Ω_k over its faces $i = 1, \ldots, 4$ depending on $T_k^{n,\Delta t}$. Again, there are tensor operations by multiplying unique flux matrices, face-local matrices, and the time kernel T. For completeness, F can also express the *Dynamic Rupture Kernel* or boundary conditions, which are explained in detail in preceded work [29].

2.3.2. Heterogeneous Components

The last subsection summarized the underlying numerical scheme and how it is expressed by tensor operations. Considering SeisSol and GPUs, these are implemented by three modules, as illustrated in Figure 2.4, that together enable GPU access: Yateto, GemmForge, and Device. Device is essentially a primary API endpoint to prepare and synchronize computations of the numerical scheme. Yateto generates code to express tensor operations with GEMM operations and corresponding calls GEMM libraries under the assumption of using it in the context of an ADER-DG method. GemmForge, on the other hand, implements GEMM code generation for GPUs, until now only using CUDA and HIP, and leverages performance by exploiting knowledge of the application domain. One main advantage of this design is that one does not need to change code within SeisSol if adding a new accelerator API. In the following, the most important aspects of all repositories are illuminated and how they apply to Intel GPUs. Chapter 3 then use this information to analyze and implement necessary changes.

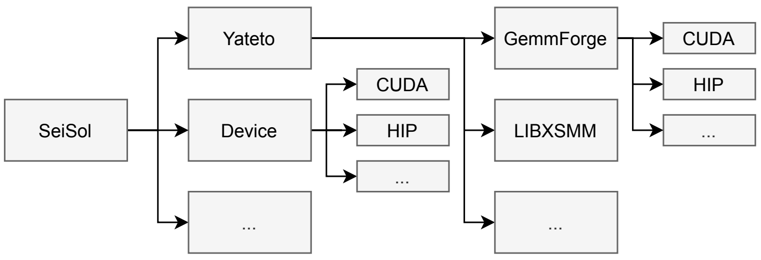


Figure 2.4: Summary of the crucial heterogeneous components in SeisSol: Yateto provides a DSL to express the numerical scheme implemented by GemmForge or other GEMM libraries. Device presents a facade to a typical GPU programming model like Sycl.

Device

Device[4] implements Facade [30] and Adapter [30] design patterns for accelerator accesses. It provides an API abstracting common GPU programming concepts, including memory allocation, copying data from or to the device, and synchronization, but also a custom stack on the global memory of a device and a stream buffer for out-of-order tasks. Additionally, there is a sub-interface for common GPU algorithms containing data initialization and scaling, reduction, and batched array manipulation[5]. As stated in Chapter 1, there have been implementations of this facade in CUDA [2] and HIP [3] so far. For Intel GPUs, a new adaptee needs to be implemented in Sycl.

Device is primarily an API to prepare and finalize data for GEMMs but is also responsible for synchronizing kernels and balance a workload to queues. SeisSol already calls all methods of the facade in the right order and produces numerical correct results with the CUDA implementation [2]. Therefore, if the Sycl implementation behaves exactly like the CUDA one, the results must be logically correct. A deeper analysis of what that means for the implementation is available in Chapter 3. Moreover, the repository contains a parallel Jacobi benchmark and several test suites that can be used or extended to ensure correctness and

[4] `https://github.com/SeisSol/Device`

[5] A batched array is considered as a contiguously allocated area of memory containing logically separated containers.

for evaluation. The benchmark is implemented with an MPI+X model, where X can be any GPU programming model or language wrapped by Device. Thereby, it may demonstrate that the targeted MPI implementation is Intel-GPU aware. This is especially important, as SeisSol relies on the same design idea and does not work without GPU-aware MPI.

Yateto

Yateto[6] (Yet Another Tensor Toolbox) is a code generator, providing a domain-specific language (DSL) to express tensor operations [4]. Yateto was explicitly designed for generating kernels establishing them via 2D tensor (or GEMM) operations used in Partial Differential Equation (PDE) solvers, particularly in the ADER-DG method. Because of this, it is possible to reason about generated kernels executing on hardware, which does not apply to a generalized code generator. This includes, for example, that most matrices fit into low-level caches or that all dimensions stay fixed during runtime. Moreover, the application context immediately justifies the focus of optimizing these operations as they are the building block of solving the ADER-DG scheme. Yateto delegates the generation of kernels to GEMM libraries, for instance, LIBXSMM[7], targeting SIMD Intel CPU architectures, or GemmForge, targeting Nvidia [2] and AMD GPUs [3]. For GPUs in general, Yateto allows passing batched GEMMs, increasing the time data remains on a GPU. A batched GEMM operation is defined in Equation 2.4 and summarized based on previous work [2].

$$C_e = \alpha \cdot Op(A_d) \cdot Op(B_f) + \beta \cdot C_e \tag{2.4}$$

A, B, C are real matrices such that Matrix-Matrix-Multiplication is defined. e is the index of the current batch operation, d is either e or omitted if A is constant throughout the batches. The same holds for f and B. $Op(M)$ is either M^T or M. Note that two special cases of GEMMs, Copy-Scaled-Add (CSA) and initialization, are treated individually by Yateto, which becomes important when changing GemmForge later in this thesis. Additionally, note that all matrices are stored in column-major forward due to the origin of SeisSol in Fortran.

Considering Intel GPUs, Yateto does not need major changes, as the current design already allows to switch between the AMD and Nvidia GPU architectures. Therefore, the rest of this thesis does not focus on Yateto and assumes that an additional option for Intel GPUs is already available.

GemmForge

GemmForge[8] is SeisSol's code generator for batched GEMM operations on a GPU, as mentioned in the section before. GemmForge allows specifying a GPU architecture, which is passed to a GEMM, CSA, or initialization code generator. Exploiting the programming model of the target architecture, the code is generated following the same optimization ideas on any GPU model, which are summarized in the following based on preceded work [2].

[6]`https://github.com/SeisSol/yateto`
[7]`https://github.com/hfp/libxsmm`
[8]`https://github.com/ravil-mobile/gemmforge`

Inspired by Rivera et al. [31], GemmForge splits a single matrix-matrix multiplication into a sum of outer products, distributed parallelly to up to three Nvidia warps. Dorozhinskii and Bader denote these as *team*, and the first m threads of a team *active threads*. The size of a team is determined by the count of rows of $Op(A_d)$. Each load or store of any column (recall the column-major layout of SeisSol) of A or C by a team then leads to coalesced memory access. Furthermore, each active thread loads one element of the same column of A into a register and combines it with a row of matrix B, which is loaded into the shared memory of a GPU. At the beginning of a kernel execution, intermediate results of a multiplication are stored in registers and copied back to global memory at the end of the entire product. These resulting fine-tuned kernels are further optimized by loop-unrolling and CUDAs `launch_bounds` hint for the compiler, limiting the number of registers per thread to avoid register spilling.

Evaluating these techniques, the CUDA implementation of GemmForge outperforms native HPC libraries like cuBLAS[9] and can reach the theoretical peak performance of a GPU [2]. This is attributable to the fact that these types of kernels are 1) memory bound and 2) allow the techniques as described above due to the known structure of the matrices at runtime. This does not apply to general GEMMs like those processed by cuBLAS. GemmForge's HIP implementation follows the same ideas and also gets close to peak performance [3].

Regarding Intel GPUs, the architectural layer of the code generation has to be extended to include object-oriented concepts of C++ and to skip optimizations that are not expressible by Sycl (like register limitations). However, no specialized GEMM implementations for Sycl are created in this work as there is not much experience on performance characteristics of Intel GPUs in the literature yet. Instead, the existing technique as described above is employed, even though it might not be optimal for Intel GPUs in general. The correctness of the implementation is ensured by an existing test-suite for GEMM operations and new suites for CSA and initialization created in this work. The next section summarizes how the described changes and evaluations for Sycl and Intel GPUs refer to Performance Portability.

2.3.3. Performance Portability

Performance Portability is a non-standardized term in computer science for a) providing a correct piece of software to a new platform and b) evaluating performance relative to the new underlying architecture at once. Pennycook et al. define Performance portability as follows: Performance Portability is "a measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set" [32]. This work stays to a single problem on two platforms for Performance Portability: A single and a double GEMM benchmark provided by GemmForge, that maps to a compute kernel solved by SeisSol in a typical setup and runs on an Iris Xe Max, Gen9 UHD Graphics P630, and Nvidia RTX 3090. Therefore, the suggested metric of their work is not applied. Instead, evaluations of performance results are performed directly by a Roofline Model analysis [33].

A Roofline Model is a visualization for performance characteristics of both a device and an algorithm. It plots operational intensity (in floating-point operations per byte) against

[9] `https://docs.nvidia.com/cuda/cublas/index.html`

measured performance (in Flop/s) of an algorithm, thus showing if its performance is either memory-bound (i.e. limited by the bandwidth) or compute-bound (i.e. limited by the count of operations a CPU can execute per cycle). An operational intensity can be received by hardware performance counters or estimated by dividing the numbers of Flops by the count of bytes processed in the algorithm. On the other hand, the theoretical peak performance on both labels is retrieved either with the specification of a device or by microbenchmarks. The latter can create fairer conditions for GPUs, as the theoretical bandwidth is not achievable by default floating-point data types. The closer an algorithm gets to these peak performances, the more efficient it uses the underlying hardware.

For the parallel Jacobi and SeisSol itself, benchmarks provided by these applications are executed on the two Intel GPUs, a single Intel Arria 10, and a compute cluster containing two Nvidia RTX 3090. Chapter 4 introduces the execution environments as well as the parameter and the output of the algorithms in detail.

3. Analysis and Design

The last chapter introduced SeisSol's heterogeneous components, Yateto, Device, and GemmForge, and gave an idea of what changes are required to support Intel GPUs with the Sycl programming model. This chapter analyzes Device and GemmForge in detail and designs a new implementation using the existing CUDA code as a template such that the Sycl code is functionally equivalent to it. Next, important implementation details are provided, for example, individual reductions for both hipSYCL and oneAPI, and the correctness of all changes is ensured by various test suites.

3.1. Device

Device provides a Singleton [30] interface facade for an arbitrary accelerator API and common GPU algorithms. Figure 3.1 shows a higher-level overview of the structure of its CUDA implementation, which is summarized in the following.

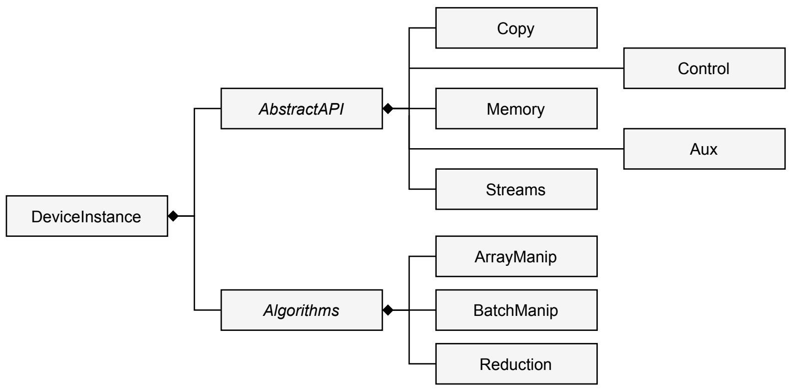


Figure 3.1: Structure of CUDA components of Device. `AbstractAPI` hides the implementation of native features like data transfer in `Copy` or custom ones like a GPU stack provided by `Memory`. `Algorithms` encapsulates typical parallel algorithms like reduction.

`Memory` implements the allocation and deletion of global, shared, and page-locked memory. Additionally, it includes a custom GPU stack on the global memory used for quick data access in SeisSol's compute kernels with push and pop operations. `Copy` transfers 1D or padded 2D data from a host to a device or vice versa, either synchronously or asynchronously. `Streams` maintains CUDA device streams which are similar to a kernel queue in Sycl, allowing to introduce further levels of parallelism. All streams are ordered and managed by a circular queue buffer implemented by `Streams`. `Control` initializes and selects a CUDA device by a certain id number and synchronizes it. `Aux` contains a test kernel and a method for error checking.

`ArrayManip` and `BatchManip` are parts of the `Algorithms` component and implement shortcut methods for batched or non-batched array initialization and scaling, whereas `Reduction` aggregates an array depending on an arithmetic operator.

In the following subsections, this architecture is used as a template for the Sycl implementation. They answer, which design concepts of Sycl are different and why they prevent performing an automatic conversion, for example with Intel's compatibility tool[1]. For that, CUDA API is compared with Sycl API, and inconsistencies between the programming models are extracted. Subsequently, changes in the architecture (compared to Figure 3.1) are designed, implemented, and verified.

3.1.1. CUDA compared to Sycl

This subsection illuminates differences and similarities between CUDA and Sycl needed for designing the Sycl implementation of Device and, later in the chapter, GemmForge. These considerations remain with the most crucial aspects received by the author's experience during this work. Note that there are more ways to express aspects from one language within the other, for example, explicit device management in CUDA or subgroups in Sycl. However, as they do not play a role in later changes, they are not consulted in the following explanations based on CodePlay's migration guide[2].

One difference between Sycl and CUDA is that Sycl is designed to access arbitrary devices, whereas CUDA is not. That already leads to a break in the programming concepts: As described in Section 2.2.2, the Sycl platform model holds next to a host device all other devices (for example, CPUs or FPGAs) that are accessible. Therefore, in Sycl, a programmer is responsible for selecting a GPU over all other devices. In CUDA, there is no need for such filtering, as only Nvidia GPUs are available. Furthermore, CUDA automatically detects and provides the best device on a machine. Sycl needs a device selector or a lookup of the underlying platforms to express the same concept.

Next, CUDA has a global static context presented by the runtime. Therefore, launching a kernel does not require calling any C++ objects or holding a context like a Sycl queue. Additionally, kernels in CUDA per default execute on a default stream, i.e., passing a stream explicitly to a kernel is not necessary for CUDA. Sycl does not have such implicit queuing. Furthermore, Sycl streams are per default out-of-order, which means that any kernel passed to a queue can be executed in any order. That is not the case with CUDA, where streams are in-order.

The index space of CUDA is very similar to Sycl except in terms of notation. In CUDA, for example, a work item is called a *thread*, a work-group a *block*, and an NDRange a *grid*. Calculations of a global thread index in Sycl follow the same regulations as in CUDA in all dimensions, but Sycl also has convenient functions and objects to circumvent these computations. An example for 1D indexing using CUDA and a verbose and abbreviated variant in Sycl is provided in Equation 3.1 to 3.3. On the other hand, Figure 3.2 compares CUDA and Sycl 2D index space labeled with method calls necessary to access the information.

[1]`https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compatibility-tool.html`

[2]`https://developer.codeplay.com/products/computecpp/ce/guides/sycl-for-cuda-developers/migration`

Note that both the figure and equation assume that all Sycl calls are performed on an `nd_item` within a kernel function. For CUDA, the same is asserted but without the need for the latter.

$$
\begin{aligned}
&\text{blockIdx.x} \cdot \text{blockDim.x} + \text{threadIdx.x} && (3.1)\\
\equiv\, &\text{get\_group().get\_id(0)} \cdot \text{get\_local\_range(0)} + \text{get\_local\_id(0)} && (3.2)\\
\equiv\, &\text{get\_global\_id(0)} && (3.3)
\end{aligned}
$$

However, one essential difference is the way how the index space is created in both programming models. A CUDA kernel launcher requires a specification of m_k blocks and n_k threads per dimension k. That results during runtime in a grid size of $m_k \times n_k$, or m_k blocks of size n_k. Sycl reverses this idea: A kernel function takes a (global) NDRange (or grid in the CUDA terms) of size $m_k \times n_k$ per dimension and a local size n_k of all workgroups (blocks). This difference of launching a kernel index has to be considered while porting CUDA to Sycl code.

CUDA and Sycl share the same ideas regarding the memory model as described in Section 2.2.2 but express it with different syntax. For example, shared memory can be allocated within a kernel in CUDA but outside of a kernel in Sycl. Nevertheless, Sycl allows two different data management strategies: Raw C pointers, which are unconstrained accessible via pointer arithmetics, and memory buffers. Memory buffers provide an abstraction of memory and are fully maintained by the Sycl runtime. They are not directly accessible by index operators but rather with an accessor object. Buffers and accessors in combination with out-of-order queues enable the Sycl runtime to build dependency graphs. Similar to OpenMP tasking, Sycl can then schedule kernels with this graph and may add more parallelism.

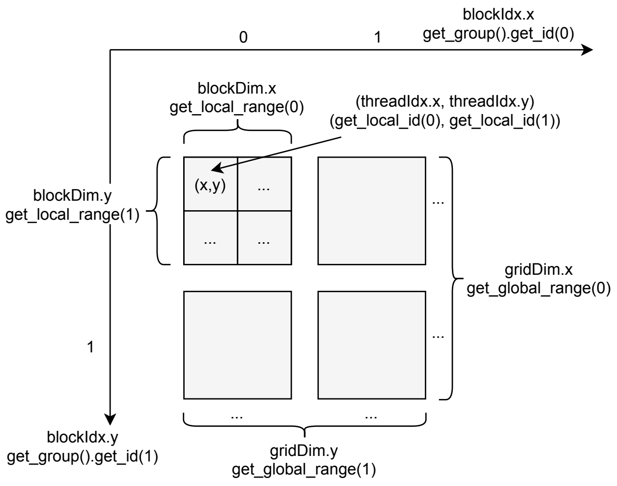


Figure 3.2.: CUDA (first line) and Sycl (second line) indexing in comparison for a 2D index space. Except for different wording and additional object hierarchies in Sycl, both follow the same design idea. Note that all methods must be called within a kernel. Sycl additionally requires to call the methods on an `nd_item`.

Furthermore, CUDA default memory copy operations are synchronous, whereas asynchronous ones require a different set of functions. In Sycl, those operations need to be submitted on a queue when using raw pointers. Any queuing in Sycl returns an event that is executed asynchronously and can be waited for. Alternatively, there are wait operations on a queue that synchronize it completely with the host. However, synchronizing a single queue does not synchronize the complete device, as there can be multiple queues per device. Comparing to CUDA, synchronizing Nvidia GPUs is commonly performed either per device completely or per single stream.

3.1.2. Design and Implementation

Using the CUDA architecture as a template and aware of the discrepancies and similarities between both programming languages, the Sycl part of Device is designed and implemented as follows:

- A container class called `DeviceContext` is created, which is instantiated during the initialization phase of the `Control` component and filled with each available definition of a concrete accelerator installed at a given system. However, to enable GPU-aware MPI, all non-level zero backends are excluded regarding Intel GPUs, as it is only available on the latter one. The filled vector is sorted by a newly introduced environment variable, `PREFERRED_DEVICE_TYPE`, requiring a value of either GPU, CPU, or FPGA. This allows to set and switch devices at runtime by an index: Because the context class holds all references that are specific per device (like the queue buffer or the stack), pointers to them are updated once a device was changed. Nevertheless, it is still a user's responsibility to distribute the work to the maximum count of the same device types.
- For reasons of the Single-Responsibility principle of Software Engineering, dedicated classes `Stack` and `DeviceCircularStreamBuffer` are created, instead of including them directly in `Memory` or `Stream`. Note that all queues in the buffer are created in-order.
- All data managements in `Memory` or `Copy` are handled via C raw pointers instead of buffers. Buffers are great for automatic dependency management, but this is not a use-case for SeisSol and therefore, they are not used for this work. Additionally, omitting buffers reduces the overhead which is caused by the runtime with them.
- Furthermore, any memory copy operations are synchronized, except if methods are explicitly marked as asynchronous. Additionally, to the best of the author's knowledge, there are no 2D memory copy operations for strided memory transfer in Sycl. Due to that, they are implemented manually via a for-loop.
- Any kernel indexing in `Algorithms` is directly converted using Figure 3.2. The global kernel index space is created by multiplying CUDA blocks and threads in every dimension, whereas the local range inherits CUDA block size.
- Instead of checking error codes as returned from CUDA API, an asynchronous exception handler is passed to any queue. Furthermore, all assertions on the integrity of a method are expressed with exceptions.
- Two implementations of `Reduction` are provided: The first one bases on the Sycl reduction library implemented by oneAPI. The second one is a custom implementation for platforms like hipSYCL that do not implement this part of the standard.
- The build system compiles all Sycl code with O3 optimization. Furthermore, Ahead-of-Time (AoT) compilation is activated if any preferred device is set. That can increase the performance of a kernel as it does not have to be assembled at runtime via Just-In-Time

compilation. Additionally, AoT compilation is mandatory for FPGAs. All other changes in the build files are excluded here but are available at the public GitHub repository for interested readers.

Appendix A.3 shows an updated Figure 3.1 that summarizes the design changes in the API component regarding Sycl based on the descriptions above.

3.1.3. Correctness

Correctness of the implementation is ensured by three examples and test suites. The first example submits a test kernel and transfers mock data between the host and a device, whereas the second example is an *HelloWorld* test for GPU-aware MPI. For both tests, a limitation was encountered on a local Intel HD Graphics 620. It does not have hardware support for double-precision floating points. These types are only supported if `IGC_EnableDPEmulation` environment variable is set to 1. The same holds for the Iris Xe Max GPU[3]. However, this does not impact SeisSol, as the precision is definable at compile-time. Both tests pass their requirements on the local HD Graphics P620, but note that Intel MPI requires setting `I_MPI_OFFLOAD` environment variable to 2 before running MPI with device pointers.

The parallel Jacobi benchmark is a more complex example implemented for CPUs and GPUs, which is summarized based on preceded work [3] in the following. It approximates a solution of a system of diagonal-dominant linear equations $Ax = b$ by converging to a fixpoint given by $\phi(x_k) = x_k + M^{-1}r$. M contains diagonal elements of A, and $r = b - Ax_k$ is the residual value of a current convergence step. The implementation in the repository splits computations of matrix-vector products and distributes these across all MPI ranks. Each rank is assigned to a single GPU. Intermediate and final results are propagated with MPI collectives. For reproducibility, the benchmark uses a fixed band matrix A, solution vector x and initial approximation x_0 stated in Equation 3.4. Additionally, the maximum error is set to $\epsilon = 10^{-7}$.

$$A = \begin{pmatrix} 2 & -1 & 0 & & 0 & 0 & 0 \\ -1 & 2 & -1 & \dots & 0 & 0 & 0 \\ 0 & -1 & 2 & & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & & 2 & -1 & 0 \\ 0 & 0 & 0 & \dots & -1 & 2 & -1 \\ 0 & 0 & 0 & & 0 & -1 & 2 \end{pmatrix}, x_0 = \begin{pmatrix} -10 \\ \vdots \\ -10 \end{pmatrix}, x = \begin{pmatrix} -1 \\ \vdots \\ -1 \end{pmatrix} \tag{3.4}$$

Running this example on Intel HD Graphics P620 and one or two Iris Xe Max with row sizes $\{10^k \mid k = 3,4,5,6,7\} \cup \{5 \cdot 10^k \mid k = 3,4,5,6\}$ of A using oneAPI in the packaged version result in equal outcomes using Intel MPI, indicating that the Sycl implementation is correct and Intel GPU-aware MPI works. Similar experiments were performed using native CUDA,

[3]`https://community.intel.com/t5/Intel-DevCloud/Iris-Xe-MAX-node-is-missing-double-precision-support/td-p/1247876?profile.language=de`

oneAPI (open-source), and hipSYCL with one and two RTX 3090 and OpenMPI without UCX layer. The same results were obtained with that setup, as well.

Last, all available tests for the `Reduction` part of the algorithms library were executed. Additionally, they were extended by suites for the custom implementation of 2D memory copy as well as tests for array manipulations. As all tests are passing, the correctness of `Algorithms` is also verified under the assumption that they do not contain errors.

3.2. GemmForge

GemmForge is a Python module for generating batched GEMM kernels as introduced in Section 2.3.2. Parallel to this thesis, GemmForge was redesigned by the community compared to its original structure described by Dorozhinskii [2]. The following section always refers to the design provided by the tag v0.0.204, for which Figure 3.3 shows a truncated overview of its architecture.

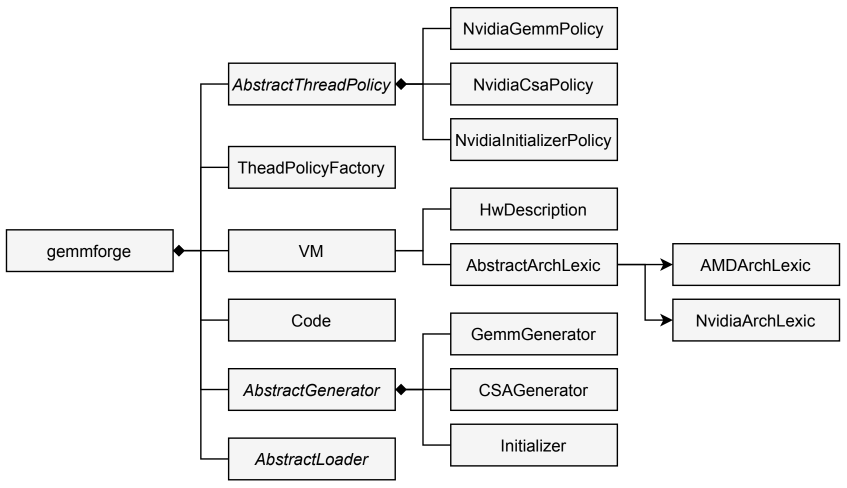


Figure 3.3: Architectural style of GemmForge. `VM` stores important hardware details of a vendor model. `AbstractThreadPolicy` defines a strategy to create a kernel index space. `AbstractGenerator` and `AbstractLoader` generate batched GEMM kernels with the help of the `Code` dictionary.

`VM` is a container class for a GPU vendor. It contains `HwDescription` storing hardware details like the size of shared memory or the maximum count of threads for a GPU model. `AbstractArchLexic` is a facade for vendor-specific language syntax used by the code generators. `AbstractThreadPolicy` is a Strategy Pattern [30] exploiting these data to define an indexing space of a kernel. Default policies are specific for Nvidia GPUs created by a Factory [30] class but can be extended for other GPU manufacturers. `AbstractGenerator` and `AbstractLoader` are base classes for batched GEMM code generation as described in Section 2.3.2. `AbstractGenerator` analyzes the underlying `VM` and thread policy to create a kernel launcher, a kernel itself, and a belonging header. `AbstractLoader` is a sub-generator for loading data into shared memory. `Code` serves as a dictionary for C++ code generation. `CSAGenerator`, `GemmGenerator`, and `Initializer` are the respective generator implementations.

One issue thereby is that the GemmForge implementation is tailored to CUDA and HIP. HIP and CUDA share almost all programming concepts except for minor syntactical differences.

Therefore, only the bare necessities like thread indexing within a kernel are abstracted in `AbstractArchLexic`. Other concepts like launching kernels, creating an index space with dimension objects, or synchronization within a block are not abstracted, resulting in a non-disjoint separation between GPU and host code. The following section discusses two designs that change the architecture of GemmForge such that GPU and host code are fully separated.

3.2.1. Design

The main challenge of GemmForge design is to eliminate mutually exclusive concepts in CUDA and Sycl that are difficult to generalize and that prevent to solely add a third implementation to `AbstractArchLexic`. For example, declaration of shared memory in CUDA takes place within a kernel but in Sycl outside of it. Or: Sycl kernels are typically expressed with anonymous lambda methods, whereas CUDA uses functions labeled with a language extension. All these discrepancies are not issues in Device, where each implementation is isolated from the other ones.

To tackle these problems, two design options are considered: The first one targets to completely refactor all three generators by introducing a Composite Pattern [30] in combination with GPU and host code Builders [30] to express a formal grammar for each programming language. That design implements a fully GEMM-independent code generation package for default C++, CUDA, HIP, and Sycl code. Furthermore, Sycl and CUDA code is interchangeable and allows transferring CUDA to Sycl step-by-step, if necessary. Appendix A.4 emphasizes these ideas by a class diagram.

The second design aims to add more levels of abstraction into `AbstractArchLexic` and move all GPU-specific code to overridden methods of derived classes. At the same time, all non-shared information needed during the generation phase in the parameters of such a method is unified. For example, the method in `AbstractGenerator`, which creates a kernel, is extended by a count of requested local memory. This parameter then acts as an offer that can be consumed or not. In this case, `SyclArchLexic` can use this hint to declare the local memory during the kernel setup, whereas `NvidiaArchLexic` ignores it. Moreover, the same parameter is passed to a standalone version for creating local memory within the kernel code. The CUDA architecture lexicon accepts this parameter, whereas the Sycl one, on the other hand, skips that method completely.

This work follows the second variant for the following reason: The first design requires a complete re-implementation of GemmForge with a complex structure of many different classes implementing the Composite and Code Builder patterns. An experimental setup[4] using C# confirms any concerns on this. Therefore, and due to uncertainties accompanying that design regarding time limitations, the ideas of redesigning GemmForge were discarded.

[4]`https://github.com/ZaubererHaft/GemmForge-Experimental`

3.2.2. Implementation

At the chosen design, a new architecture lexicon `SyclArchLexic` is added and all GPU-specific method calls in the generators are deferred to `AbstractArchLexic` or `Code`. Furthermore, the existing methods are extended with offers to mutually exclusive concepts discarded or consumed by vendor-specific implementations as described in the section before. Additionally, passing a non-null stream or queue pointer as a parameter is asserted in the generated kernel launcher. Hence, the method always gets the right context where to submit a GEMM operation. So far, this was either null or a stream from the `CircularBuffer` of the Device submodule passed as a parameter by SeisSol. Within CUDA a null (or zero) stream is valid for a kernel launcher, as it resembles the default stream. Using Sycl, one can't make assumptions about a device and need a correct platform chosen by Device (compare Section 3.1.1). Figure 3.4 illustrates the solution to that problem by a shortened kernel launcher generated by GemmForge. Initially, the change broke the compatibility to SeisSol, but a fix was provided by the community during this work, such that SeisSol always passes a stream that is different from `null`.

```
void gemm(const float * A, const float * B,
          float * C, unsigned NumElements,
          void* streamPtr) {
  dim3 Block(64, 1, 1);
  dim3 Grid((NumElements + 1 - 1) / 1, 1, 1);
  cudaStream_t stream = (streamPtr != nullptr) ?
       static_cast<cudaStream_t>(streamPtr) : 0;
  kernel<<<Grid,Block,0,stream>>>(A, B, C,
       NumElements);
  CHECK_ERR;
  }
```

```
void gemm(const float * A, const float * B,
          float * C, unsigned NumElements,
          void* streamPtr) {
  range<3> Block(64, 1, 1);
  range<3> Grid((NumElements + 1 - 1) / 1, 1, 1);
  if (streamPtr == nullptr)
    throw std::invalid_argument();

  queue *stream = static_cast<queue *>(streamPtr);
  kernel(stream, Grid, Block, A, B, C, NumElements);
}
```

Figure 3.4.: Shortened GEMM kernel launcher of CUDA (left) and Sycl (right) generated by GemmForge. Pointers `A`, `B`, `C` contain `NumElements` contiguously batched matrices. Active threads per kernel are determined by the y-index range, whereas z index range is not set by the GEMM generator. The Sycl code asserts a valid context provided by the `streamPtr` and implicitly checks errors by exceptions thrown by the runtime.

The remaining part of the Sycl implementation in the respective generators is derived from the CUDA code similar as performed in Section 3.1.2, except for one difference. In Device, the thread indexing is directly converted by the corresponding methods (recall Figure 3.2 for an overview). However, this direct indexing causes issues using Sycl with CUDA backend[5]: The linear index in CUDA allows up to $2^{31} - 1$ threads along x-axis. However, the CUDA implementation for Sycl used by oneAPI and hipSYCL does not map the x-axis of Sycl to the x-axis in CUDA but to the z-axis, which is limited to $2^{16} - 1$. In a typical setup of SeisSol, more than 2 GB of data is used. Assume matrix sizes of $C \in \mathbb{R}^{56\times56}$, $A \in \mathbb{R}^{56\times9}$, $B \in \mathbb{R}^{9\times9}$ with single precision. This results in $2 \cdot 1024^3\ \mathrm{B} \div ((56 \cdot 56 + 56 \cdot 9 + 9 \cdot 9) \cdot 4\ \mathrm{B}) \approx 144281$

[5]`https://github.com/intel/llvm/issues/1388`

`NumElements`, each requiring 64 threads to process the biggest matrix A with a quadratic size of 56. Therefore, a direct conversion raises a `CUDA_ERROR_INVALID_VALUE`, as the maximum number of threads in y, z direction in CUDA is exceeded.

To solve this problem, an additional abstraction layer is introduced for thread indexing in each generator provided by `AbstractArchLexic`. For CUDA and HIP, the current implementation is adopted without changes. For Sycl, the indexing strategy in GemmForge is switched as follows: The local size of a range is passed also to the global one. However, the global range is extended by multiplying the z-size with the count of threads, and not the x-size. That results in a re-ordering of the blocks along the z-axis. Figure 3.5 illustrates this change using a generic global space and a fixed local space of 64 along x-axis.

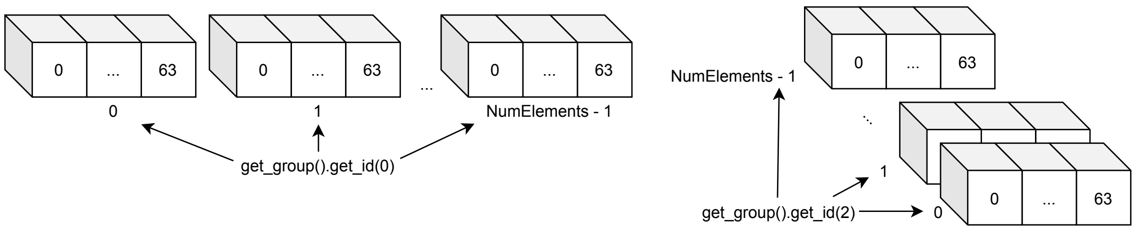


Figure 3.5.: Generalized index space for CUDA (left) and Sycl (right) used in GemmForge kernel launcher. Because the CUDA backend for Sycl maps the z-axis in Sycl to the x-axis in CUDA when using a 3D range, it is necessary to exchange both during GEMM code generation using Sycl.

Due to a lack of information about Iris Xe HPC series, no separate thread policy classes for Intel are added. It is to believe that tuning the LP series is not productive for the upcoming high-performance products. In Chapter 4, all performance analyses are therefore performed with the default thread policy. Regardless of that, the hardware descriptions of Iris Xe Max and UHD Graphics P630, as found in Section 2.1.2, are added to compute the active threads for the GEMM operation.

3.2.3. Correctness

To verify all changes, GemmForge test suite for GEMM operations is executed, containing setups for transposed and non-transposed matrices A, B, C as well as different values for α, β. The tests fill these matrices with random numbers and compare the resulting matrix C with a simple CPU implementation producing C' using their absolute difference $|c_{ij} - c'_{ij}|$. A test fails if this difference is higher than a fixed maximum error $\epsilon = 10^{-5}$. Furthermore, similar tests are implemented for `CSAGenerator` and `Initializer` generators but with the unused matrices removed.

The tests were executed on Iris Xe Max and UHD Graphics P630 using oneAPI in the packaged version as well as on Nvidia RTX 3090 using oneAPI (open-source variant) and hipSYCL. All `GemmGenerator` tests passed on all machines. The same held for `CSAGenerator` and `Initializer` using RTX 3090. However, P630 and DG1 raised a runtime exception `CL_INVALID_WORK_GROUP_SIZE` when executing these tests using oneAPI or created illegal results with hipSYCL. Investigating this problem using the open-source variant of oneAPI[6] reveals that the total local size calculated with `get_local_range(0)` · `get_local_range(1)` · `get_local_range(2)` must not exceed `PI_KERNEL_GROUP_INFO_WORK_GROUP_SIZE` provided by an accelerator. For P630, this value is set 256, for DG1, it is 512, which can be reproduced by running `clinfo`[7]. That means that a local size of a kernel range is limited to 256 or 512 for these devices, respectively. However, `CSAGenerator` and `Initializer` index their matrices fully within the local index. For example, initialization of batched matrices $A_k \in \mathbb{R}^{56\times 9}$ creates a local indexing space of at least $64 \cdot 9 = 576$ when aligning the row size to 64. That already exceeds the maximum local size of both devices.

For DG1, this problem is circumventable by not aligning the x-size to a multiplier of Nvidia warp size of 32, which allows running the tests and SeisSol proxy application. In contrast to that, P630 can't run both applications because even with unaligned row sizes, a minimum of $56 \times 9 = 504$ threads per workgroup is required. For more resource-intensive simulations, both limitations on the local size are a drawback compared to Nvidia, where a thread block is limited to 1024 threads[8]. Nevertheless, it is expectable that Intel HPC GPUs allow a higher maximum number of threads per workgroup, maybe similar to Nvidia. On that account, the index generation is not changed such that it is compatible with the integrated graphics chips.

[6] `https://github.com/intel/llvm/blob/sycl/sycl/source/detail/error_handling/enqueue_kernel.cpp`

[7] `http://manpages.ubuntu.com/manpages/bionic/man1/clinfo.1.html`

[8] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, Section 2.2

4. Evaluation and Discussion

This chapter evaluates all introduced changes and their effect on SeisSol using two test systems: Intel DevCloud[1] and Heisenbug[2] provided by the Geophysics department of the Ludwig-Maximilian University (LMU). The DevCloud allows requesting multiple compute nodes, equipped with one to many Iris Xe Max, integrated UHD Graphics P630, or even Arria 10 FPGAs. Heisenbug system comes without a job scheduler for their two Nvidia RTX 3090 and a single AMD EPYC 7662 64-Core Processor. Both types of GPUs were already used for verifying the correctness of all changes in the previous sections, but without focusing on platform details. Therefore, Table 4.1 gives an overview of the most important performance characteristics and properties crucial for the subsequent measurements.

	Intel DevCloud		Heisenbug
	Iris Xe Max	UHD Graphics P630	RTX 3090
Peak FP32 Flops	2.50 TFlop/s	460.8 GFlop/s	35.58 TFlop/s
Max. threads per block	512	256	1024
Peak bandwidth in GB/s	68.30	41.60	936.20
Sycl platform	oneAPI (packaged)		oneAPI (open-source), hipSYCL, CUDA
MPI implementation	Intel MPI		OpenMPI w/o UCX layer

Table 4.1.: Performance characteristics and properties of all test systems. The Intel Iris LP series is designed for test setups and office applications. Thus, the limited performance is not surprising compared to RTX 3090 or other high-end Nvidia GPUs.

Note that the Intel PAC With Arria 10 GX FPGA is not included in the table. Its specifications can be found at the Intel documentation[3]. The following examinations are performed on these systems in the next sections:

- The first experiment executes the parallel Jacobi benchmark of Device on DevCloud using a single Iris Xe Max and UHD Graphics P630. The goal of it is to give an impression of performance differences between discrete and integrated graphics chips of Intel. Additionally, it demonstrates that the multi-platform approach of Sycl truly works by executing the benchmark on Intel Arria 10 and Nvidia RTX 3090.

[1]`https://devcloud.intel.com/oneapi/home/`

[2]`https://www.geophysik.uni-muenchen.de/research/geocomputing/heisenbug`

[3]`https://ark.intel.com/content/www/us/en/ark/products/149169/intel-pac-with-arria-10-gx-fpga.html`

- Next, the Jacobi benchmark is investigated using two Iris Xe Max to assess the performance of Intel MPI regarding GPU-awareness. Running the same experiment on RTX 3090 with all platforms compares the performance results within the two Sycl implementations hipSYCL and oneAPI (open-source) and to native CUDA using OpenMPI without UCX layer. This experiment is then discussed as a Proof-of-Concept (PoC) that Sycl could replace the Device submodule in the future.
- Subsequently, all specifications for the Roofline Model analysis are measured by first executing a microkernel benchmark, representing a typical computation in SeisSol, to get the actual bandwidth for both Intel GPUs. Then, two GEMM kernels are executed to obtain the performance of the portability in terms of Flop/s and used to create the Roofline Model for both devices.
- To evaluate the performance of a Sycl GEMM kernel without containing native CUDA features like `launch_bounds`, the Roofline Model analysis is also executed on the Nvidia GPU. Again, this experiment is used as a PoC to demonstrate that Sycl can replace all native implementations, but here in GemmForge, instead of Device.
- SeisSol proxy application evaluates the main compute kernels of SeisSol. Running it on Iris Xe Max as well as on the Nvidia machine serves as a criterion for the success of porting SeisSol to Intel GPUs. Real simulations are not executable on DG1 due to the highly limited performance of this GPU, as already sketched in Section 3.2.3 and emphasized with Table 4.1. The latter aims to give an idea of what performance could be expectable on related Intel cards in contrast to RTX 3090 in the future.
- Simulating LOH1 and TPV5 benchmark of SeisSol on the Heisenbug server using CUDA, hipSYCL, and oneAPI (open-source) generates comparable outputs for stresses σ, velocities v, or slip rates in the fault of an earthquake[4]. An investigation of some of them by an error analysis ensures the overall correctness of the implementation. Furthermore, the same benchmarks are performed without I/O to show the entire performance of the GPU with Sycl and CUDA.

The experiments were not aimed to compare performance between the Intel and Nvidia GPUs, as they are not competitive at the current state. Additionally, differences between GPUs and CPUs are not assessed, as this was already part of prior work [2] [3]. Also, the generated CUDA code by oneAPI and hipSYCL is not analyzed in detail, as it was not the scope of this thesis. However, this is an interesting topic for future work, as the following sections show large differences in performance and numerical correctness. Note that any code was compiled with `O3` optimization level but no other additional compiler hints than stated in the last chapter. Furthermore, note that all numbers are rounded to two decimal places.

[4]A good definition of these and other geophysical terms can be found in `https://earthquake.usgs.gov/learn/glossary/`.

4.1. Device

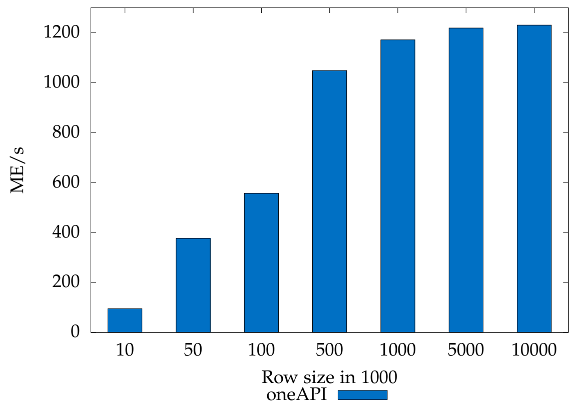
In the first experiment, the parallel Jacobi as described in Section 3.1.3 is executed. Again, a row size of $\{10^k \mid k = 3,4,5,6,7\} \cup \{5 \cdot 10^k \mid k = 3,4,5,6\}$ is used for A. The benchmark reports its runtime as an absolute performance value and the count of entries processed relative to the time across all GPUs (ranks). This throughput is defined in Equation 4.1.

$$ME/s := \left(\sum_{r=0}^{n-1} \frac{maxIter \cdot load_r}{t_r}\right) \div n \tag{4.1}$$

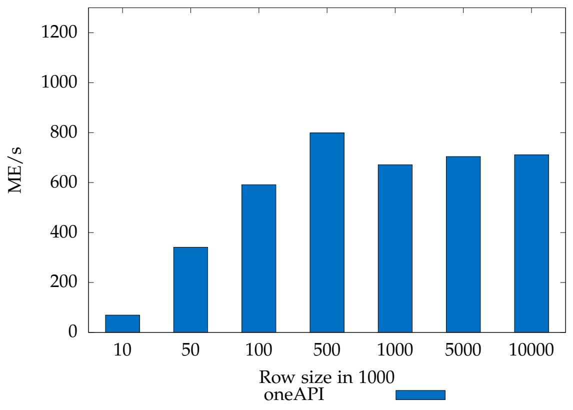
where n is the total number of ranks (GPUs), indexed by r, $maxIter$ is the constant count of fixpoint iterations. $load_r$ and t_r describe the workload per rank (GPU) and the overall time in μs spent in the compute kernels and synchronization with the host, respectively. Because $maxIter$ is constant 10^6 and the minimum number of rows is 1000 for all tests, 4.1 is denoted as Million Elements per second (ME/s) and used as the primary performance counter in the measurements. The total runtime can be obtained easily by rearranging the equation if required.

4.1.1. Single Accelerator

Figure 4.1 shows the ME/s averaged by ten iterations achieved with Iris Xe Max (left) and UHD Graphics P630 (right). Both experiments were executed on one GPU ($n = 1$) with disabled MPI, using single precision floating point values.



(a) Iris Xe Max DG1



(b) UHD Graphics P630

Figure 4.1.: Average throughputs measured with the parallel Jacobi benchmark using single precision on a discrete Intel Iris Xe Max (Gen12, left) and with the integrated UHD Graphics P630 (Gen9, right).

The figure omits row sizes < 10000 for better readability. Both GPUs produce similar performance values for lower row sizes ($10^4, 5 \cdot 10^4, 10^5$). The absolute deviation (in ME/s) of Iris Xe Max to UHD Graphics is $25.82, 35.24, -34.83$, respectively. However, the integrated graphics chip already reaches its peak performance at a row size of $5 \cdot 10^5$ that is furthermore significantly lower than DG1 (1048.62 ME/s vs. 798.96 ME/s). For larger row sizes (e.g., $10 \cdot 10^6$), the count of processed elements is nearly doubled (1230.60 ME/s vs. 711.27 ME/s) considering Iris Xe Max. These measurements indicate that the discrete LP series profits from its 32 additional EUs and a subslice-shared L3 cache and thus receives better performance counters. One can conclude that already LP series performs better on large data sets, so similar is expectable from the HPC GPUs.

Figure 4.2 shows the same experiment (average of 10 iterations, single precision, no MPI) compiled with native CUDA (green), oneAPI (open-source, blue), and hipSYCL (red) but at this time executed on a single RTX 3090. Again, very small row sizes are excluded. At lower row sizes (up to $5 \cdot 10^5$), a repetitive pattern is observable occurring at all measurements with the Nvidia card in this work: CUDA implementation results in the best performance, followed by the code generated by oneAPI (open-source) and hipSYCL with its `syclcc` compiler, respectively.

Nevertheless, in this trial, the differences between all three platforms start to decrease at a row size greater or equal than $5 \cdot 10^5$. In fact, oneAPI in the open-source variant and hipSYCL seem to profit from these large matrices and even obtain higher performance than native CUDA. Using 14737.91 ME/s achieved by CUDA with a matrix size of 5 Million as a reference value, oneAPI has 14.76 % and hipSYCL 6.20 % higher throughput per second. Due to the scope and time limitations of this work, the origins of this discrepancy are not analyzed. Instead, the experiment is used as proof that Sycl can replace CUDA in terms of functional correctness and, in some cases, also performance. However, it additionally shows that the results are highly dependent on the selected Sycl implementation, as hipSYCL, for example, processes only half as many elements as oneAPI at smaller matrices.

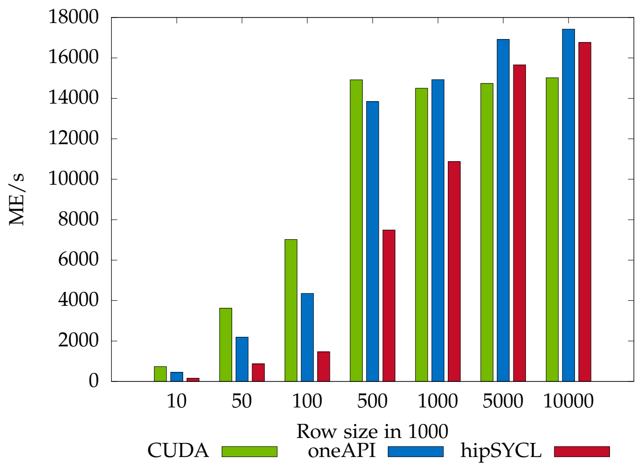


Figure 4.2: Average throughputs measured with the parallel Jacobi benchmark on a single Nvidia RTX 3090 with single precision, using native CUDA and Sycl compiled with oneAPI (open-source) and hipSYCL. Whereas CUDA is far ahead at the beginning, oneAPI and hipSYCL profit from very large matrix sizes.

For explorative purposes, results are also provided in Figure 4.3 using an Intel Arria 10 FPGA. FPGAs differ a lot from CPUs and GPUs, as they consist of configurable integrated circuits. During the compilation process, a program is mapped to the spatial architecture of logical units. Parallelism during execution is received by pipelining and not on instruction or data-level. Keeping the pipeline filled with instructions is one of the challenges of tuning FPGAs, which is exacerbated by branches in the control flow of a program.

Executing the compiled benchmark with the same row sizes as above results in considerably lower throughput in contrast to the GPUs. However, interpreting FPGAs and their performance characteristics are not part of this work and require special attention [34]. Nevertheless, future work could analyze it and define use cases for their deployment. FPGAs might be a valuable extension for SeisSol, as they can exceed the performance of GPUs in imaging and other applications [35] [36]. The cross-platform approach of Sycl could allow to exploit and evaluate this in direct comparison with graphics chips or CPUs.

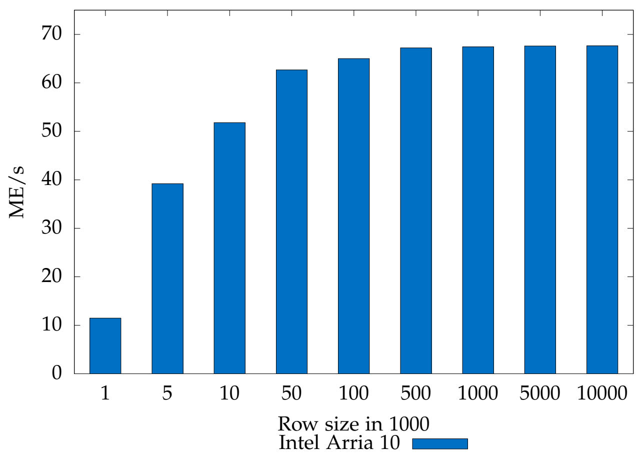


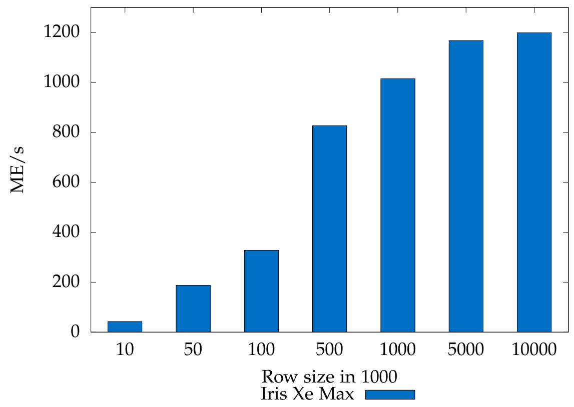
Figure 4.3: Average throughputs measured with the parallel Jacobi benchmark on a single Intel Arria 10 FPGA with single precision and compiled with oneAPI (packaged). It proves the cross-platform approach of Sycl, however, the benchmark reaches almost its peak performance already at a row size of $5 \cdot 10^4$.

4.1.2. Multi GPU

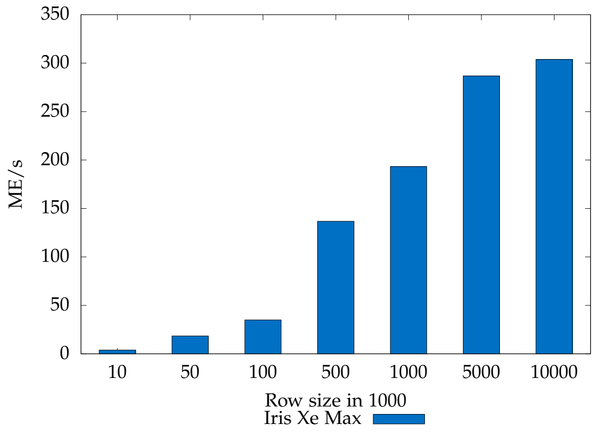
The second experiment repeats the measurements of the last section, but now with two Iris Xe Max and two RTX 3090. Here, UHD Graphics P630 and Arria 10 are skipped, as the focus lies on the performance of Intel MPI regarding GPU-awareness. For RTX 3090, all CUDA measurements are compared against the two Sycl implementations.

Using multiple GPUs, the Jacobi benchmark reports the averaged throughput across all ranks as shown in Equation 4.1. In the following, this is denoted as *compute results*. Additionally, it logs *communication results*, which follow the same formula as in Equation 4.1 but t_r is now the time in μs spent on gathering all intermediate results of an approximation x_i from each rank and sending the combined value to all others (execution time of `MPI_Allgatherv`).

Figure 4.4 shows *compute* and *communication* speed for two Iris Xe Max. A performance drop of about half ME/s compared to the single GPU execution is observable at low row sizes ($10^4, 5 \cdot 10^4, 10^5$) that relativize with higher ones. That means that the execution time stays roughly the same compared to a single GPU though the workload has been halved. Therefore, additional resources for launching the kernels do not pay off in small payloads for GPUs. Similar results are measured using two RTX 3090 (see Figure 4.5). Here, it is also worth noting that higher throughputs of oneAPI (open-source) and hipSYCL for bigger matrix sizes ($5 \cdot 10^6, 10 \cdot 10^6$) vanish using a multi-GPU setup and stay below a native CUDA implementation. Nevertheless, oneAPI gets close to CUDA (96.30 % and 98.15 % relative performance), whereas hipSYCL varies more (87.55 % and 92.98 %).



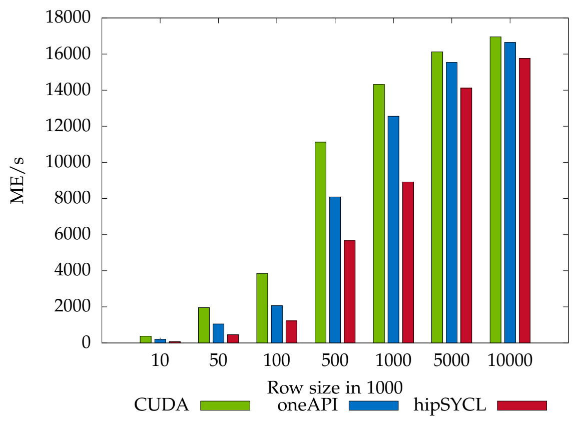
(a) Compute Results



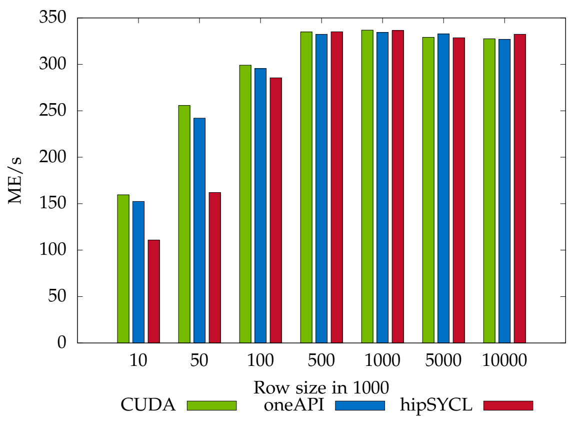
(b) Communication Results

Figure 4.4.: Average throughputs across all ranks measured with the parallel Jacobi benchmark using single precision on two discrete Intel Iris Xe Max. Left: Processed elements relative to the time spent in the compute kernels. Right: Processed elements compared to the time spent gathering and broadcasting intermediate results of a solution vector. Here, the GPU implementation of Intel MPI has a distinct performance drop in comparison to OpenMPI.

During execution, a significant impact on the overall execution time of two Iris Xe Max, driven by communication over MPI layer, was experienced. This can be attributed to the current state of Intel MPI using GPU pointers. By the time of writing, Intel representatives confirmed that the support for GPU-aware MPI is still under development and is going to be finalized with the release of Iris Xe HPC series. Therefore, Intel MPI does currently not perform as well as other GPU-aware MPI implementations, e.g. OpenMPI. The investigation of communication results obtained with two RTX 3090 using OpenMPI 3.1.5 without UCX layer is illustrated in Figure 4.5 and confirms the aforementioned statement. Here, the communication results are significantly better for lower values of row sizes than the ones obtained with Intel MPI. For example, results obtained for a row size of 10^4 are about 20 times faster at native CUDA in terms of communication speed (3.85 ME/s against 159.59 ME/s). The difference decreases with increasing row sizes (59.18 % at $5 \cdot 10^5$, 42.63 % at 10^6) and ends with 7.24 % at the highest row count, which means that communication overheads diminish at large data sets. Thus, the experiment leads to the conclusion that collective operations like `MPI_Allgatherv` have a bad impact on the execution time at the current state of Intel MPI. Nevertheless, this reported drawback is limited to collectives, which do not affect SeisSol since the major communication is executed via point-to-point operations. It will be a task for future work to check the efficiency of its implementation once the Intel HPC GPUs become available.



(a) Compute Results



(b) Communication Results

Figure 4.5.: Average throughputs across all ranks measured with the parallel Jacobi benchmark compiled with native CUDA, oneAPI (open-source) and hipSYCL using single precision on two RTX 3090. Left: Processed elements relative to the time spent in the compute kernels. Right: Processed elements compared to the time spent gathering and broadcasting intermediate results of the approximation. The GPU implementation of OpenMPI is faster on any row size than Intel MPI.

4.2. GemmForge

The second part of the evaluations performs a Roofline Model analysis for Iris Xe Max, UHD Graphics P630, and RTX 3090 by executing three microkernels: A simple bandwidth test and a single and double GEMM kernel.

The bandwidth test manually copies data from an array P to an array Q using the maximum allowed workgroup size and divides the transferred payload by the time in seconds. This determines the effective bandwidth obtained for floating-point data format, which is to be preferred for the Roofline Model analysis. Indeed, the theoretical bandwidth given by a manufacturer data sheet is only achievable by other data types than single-precision floats (e.g. doubles), which are not used in the test kernels. Therefore, using the effective bandwidth causes realistic and fair conditions for the analyses.

The single and double GEMM kernels are derived from the following equation as described by Dorozhinskii and Bader [2]:

$$C_e = D \cdot A_e \cdot B_e + C_e \tag{4.2}$$

where $C, A \in \mathbb{R}^{56\times9}, B \in \mathbb{R}^{9\times9}, D \in \mathbb{R}^{56\times56}$. D represents a constant matrix for all batches e (e.g. a stiffness or mass matrix as introduced in Section 2.3.1). As this equation contains two GEMMs ($D \cdot A_e$ or $A_e \cdot B_e$), the entire equation is split up into two kernels using a temporary storage T_e per batch:

$$T_e = A_e \cdot B_e \tag{4.3}$$

$$C_e = D \cdot T_e + C_e \tag{4.4}$$

Equation 4.3 represents a single GEMM and 4.3 together with 4.4 a double GEMM kernel. In both kernels, the maximum number of batches e_{max} is determined by the requested count of memory, similar to the explanations in Section 3.2.3. Additionally, the same experiment is repeated using a single RTX 3090 to compare the two Sycl implementations against native CUDA. All following measurements are executed with single precision and repeated 20 times.

4.2.1. Roofline Model Analysis with Intel GPUs

Figure 4.6 shows the measured bandwidths for both Iris Xe Max and UHD P630 using various requested data set sizes. The medians of the results span more while using the integrated GPU (28.19 GB/s to 29.69 GB/s) in contrast to the discrete one (56.05 GB/s to 55.91 GB/s). For Roofline Model analysis, the maximum value is filtered across all data sets, creating a more strict upper-bound limit to it: 29.90 GB/s obtained with 3.5 GB of requested elements respective 57.75 GB/s with 0.5 GB. This is approximately 84.56 % of the theoretical peak bandwidth for Iris Xe Max respective 71.88 % for UHD P630.

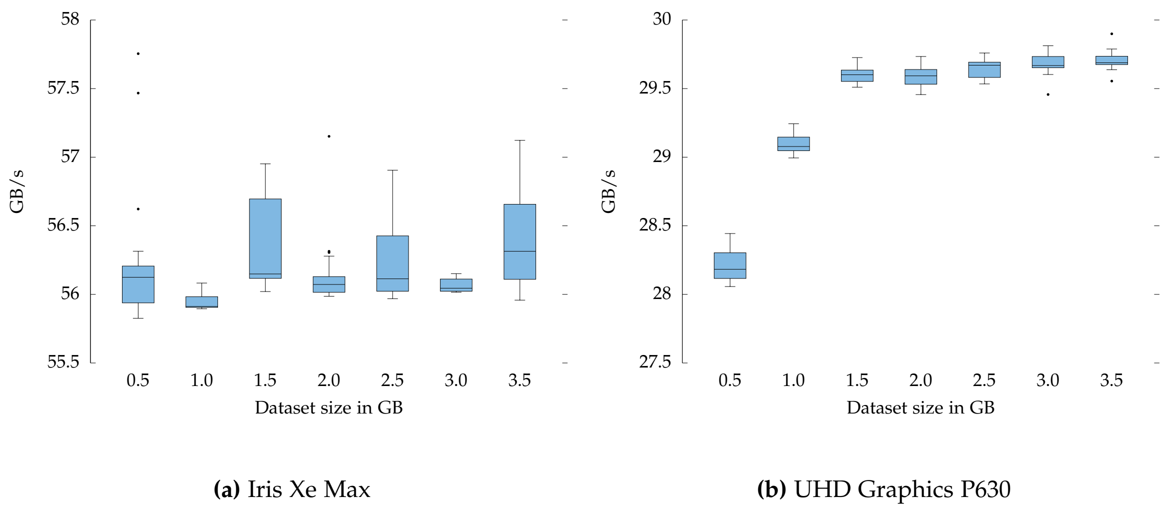


Figure 4.6.: Bandwidth measurements by microkernel using Intel Iris Xe Max (left) and UHD P630 (right). Note the different range but the same scaling along the y axis. The discrete graphics card has a more balanced median, whereas the integrated GPU varies higher at smaller data set.

The computational intensity of the single and double GEMM kernel is estimated with the help of Equation 4.3 and 4.4. For reasons of simplicity, an arbitrary single batch $e = 1$ is considered. Since $\alpha = 1$ and $\beta = 0$, GemmForge skips these operations and only multiplies A and B. Let i be a row of A and j a column of B. The respective result T_{ij} is calculated by a sum of parallel outer products, resulting in 9 multiplications and additions, or in total $9 \cdot 2$ floating-point operations. This product is repeated per column $1 \leq j \leq 9$ and row $1 \leq i \leq 56$ of A, resulting in $56 \cdot 9 \cdot 9 \cdot 2$ Flops for the first kernel. The transferred bytes are estimated accordingly: Two matrices T, A of size 56×9 and one matrix of size 9×9, all containing a single-precision floats of 4 B. In total, this results in $(2 \cdot 56 \cdot 9 + 9 \cdot 9) \cdot 4$ B per single GEMM.

The established operational intensity is given in Equation 4.5.

$$\frac{56 \cdot 9 \cdot 9 \cdot 2 \text{ Flop}}{(2 \cdot 56 \cdot 9 + 9 \cdot 9) \cdot 4 \text{ B}} \approx 2.08 \frac{\text{Flop}}{\text{B}} \tag{4.5}$$

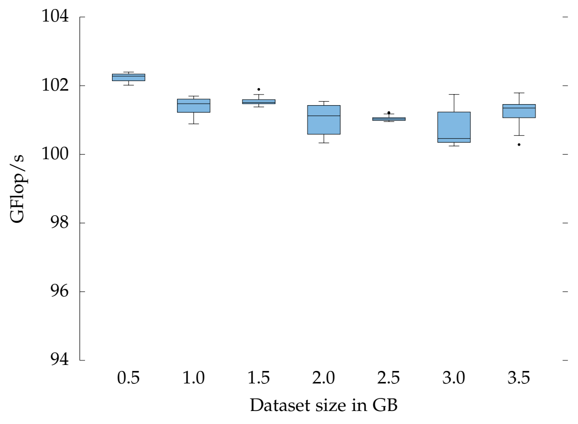
For the double GEMM Kernel, Equation 4.4 is first considered isolated. Multiplying D with T_e requires $59 \cdot 2$ operations per element. Due to the size of T_e, this is repeated $56 \cdot 9$ times and needs furthermore $56 \cdot 9$ additions to add matrix C_e. It can be speculated that matrix D stays in GPU local memory during the entire batch operation. Therefore, the size of D is not added it into the transferred bytes and thus only the memory transfers caused by loading and storing matrix C_e and T_e are taken into account. The full estimation is stated in the following equation:

$$\frac{56 \cdot 9 \cdot 56 \cdot 2 + 56 \cdot 9 \text{ Flop}}{(3 \cdot 56 \cdot 9) \cdot 4 \text{ B}} \approx 9.42 \frac{\text{Flop}}{\text{B}} \tag{4.6}$$

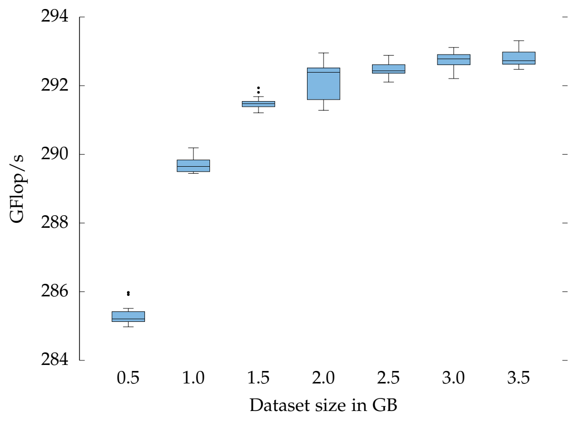
Combining 4.5 and 4.6 together results in the overall operational intensity for the double GEMM benchmark. For that, the nominators and denominators are summed up, respectively:

$$\frac{(56 \cdot 9 \cdot 9 \cdot 2) + (56 \cdot 9 \cdot 56 \cdot 2 + 56 \cdot 9) \text{ Flop}}{((2 \cdot 56 \cdot 9) + (9 \cdot 9 + 3 \cdot 56 \cdot 9)) \cdot 4 \text{ B}} \approx 6.35 \frac{\text{Flop}}{\text{B}} \tag{4.7}$$

The observed peak performance of both kernels is shown in Figure 4.7 for Iris Xe Max and 4.8 for UHD P630 using various requested data sets. The measured Flops of the single GEMM benchmark span from 100.24 to 102.40 GFlop/s using DG1 and from 48.36 to 50.12 using the integrated chip. Similar to the experiments performed in Section 4.1, almost twofold performance differences between these GPUs can be observed.

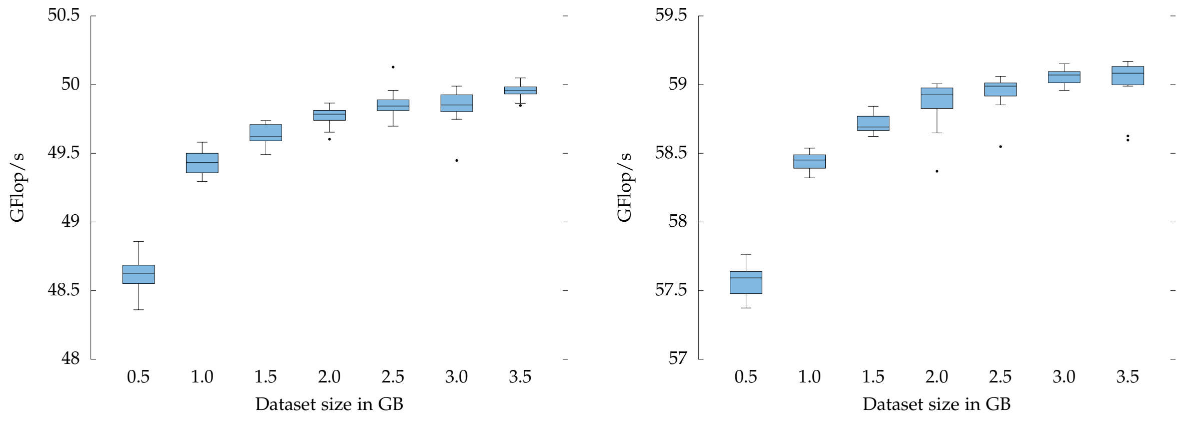


(a) Single GEMM



(b) Double GEMM

Figure 4.7.: Flop/s obtained by the single (left) and double (right) GEMM benchmark on Intel Iris Xe Max using single precision and various batch sizes. The maximum performance of the double GEMM almost triples compared to the single one. For the Roofline Model, the accomplished Flop/s are averaged over all data sets



(a) Single GEMM

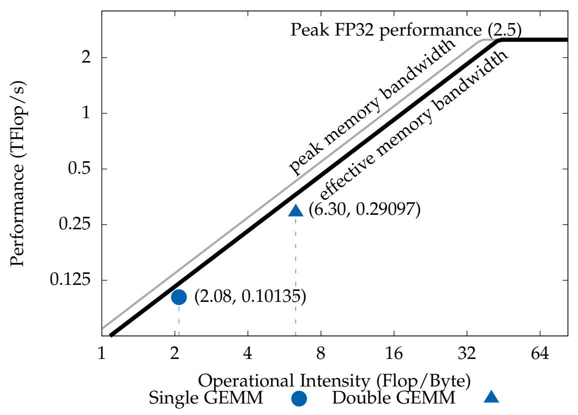
(b) Double GEMM

Figure 4.8.: Flop/s obtained by the single (left) and double (right) GEMM benchmark on the Intel UHD Graphics P630 using single precision and various batch sizes. The performance of the GPU stagnates at the double GEMM.

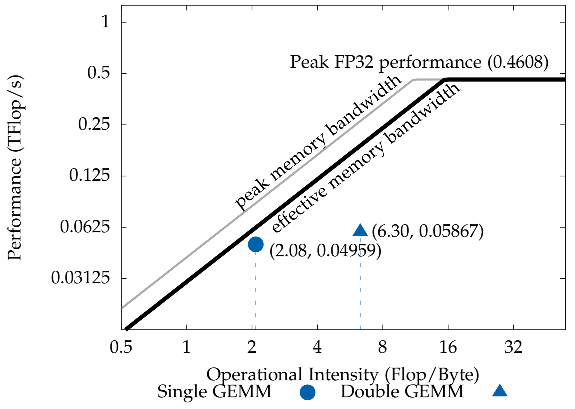
Looking at the double GEMM benchmark, this discrepancy even increases fivefold, which is due to the fact that UHD P630 is slowed down overall with the introduction of the second GEMM operation. The average Flop/s over all data sets obtained with Iris Xe Max increase by 189.4 % from 101.35 to 293.31, whereas it is only 18 % for UHD P630 (49.6 vs. 58.57). This can be explained by the fact that Gen9 UHD P630 shares its local memory throughout the entire slice. That increases latency, as all communication must pass the dataport (compare Section 2.1.1). On the other hand, Gen12 Iris Xe assigns (similar to Gen11) each subslice its own local memory, which mitigates this issue and increases the efficiency of the shared memory.

For the Roofline Model analysis, the GFlop/s are averaged across all datasets. These are 101.35 and 290.97 for Iris Xe and 58.67 and 49.59 for UHD Graphics. Using the peak Flop/s and the maximum bandwidth of Table 4.1, the Roofline models are built. The maximum operational intensity for Iris Xe Max is $\frac{2.5 \cdot 10^3\,\text{GFlop/s}}{68.30\,\text{GB/s}} \approx 36.60\,\text{Flop/B}$, whereas the effective is $\frac{2.5 \cdot 10^3\,\text{GFlop/s}}{57.7544\,\text{GB/s}} \approx 43.92\,\text{Flop/B}$. Applying the same reasoning, UHD Graphics P630 results in an operational intensity of 11.07 Flop/B or 15.41 Flop/B, respectively. Combining all these results together creates the Roofline diagram plotted in Figure 4.9.

From a visual perspective, one can already derive that both GPUs get close to their peak performance if considering solely the single GEMM benchmark. The double GEMM benchmark is not as close as the single GEMM in the case of Iris Xe GPU but still seems to be relatively near to the roof. However, results obtained with UHD P630 are almost twice as far away. More precisely, the slope of the roof is determined by $p_G \div i_G$, where p_G is the theoretical peak performance in GFlop/s and i_G is the effective operational intensity in Flop/Byte of GPU G and multiplied with the estimated operational intensity to obtain the exact theoretical value. Table 4.2 lists the measured and the maximum reachable performance of both GPUs for both kernels, confirming these observations.



(a) Intel Iris Xe



(b) Intel UHD Graphics P630

Figure 4.9.: Roofline Model for Iris Xe Max (left) and integrated Intel UHD Graphics P630 (right). In both diagrams, the single GEMM benchmark gets close to the effective peak performance. The double GEMM benchmark, on the other hand, gets near to the roof only with the discrete GPU, whereas the integrated one is significantly far away from that. This is attributable to the slice-shared USM of Gen9 that decreases the efficiency of the local memory.

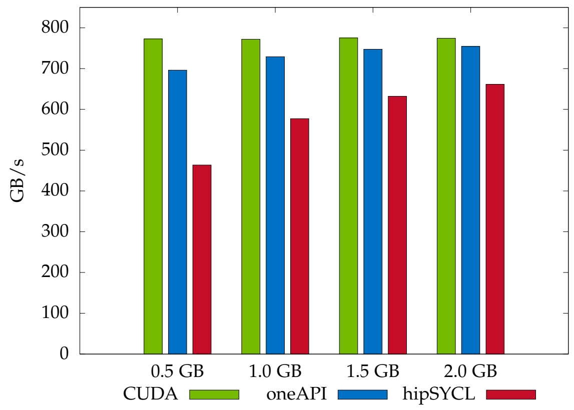
The efficiency of both Intel GPUs remain behind Nvidia (104.5 % of the maximum performance) [2] and AMD (111.0 %) [3] while comparing the single GEMM benchmark with single precision. In these preceded works, the second part of the equation was measured in isolation and is therefore not directly comparable with the double GEMM benchmark used in this thesis. However, there were performance losses around 10 % (Nvidia) and 27 % (AMD) with this variant, as well. Therefore, one can assume that the combined kernel is also slower across any GPU and thus Iris Xe achieves a satisfactory intermediate result, which is furthermore reinforced by the investigations in the next section. Additional improvements could be achieved by better utilization of the subslices by introducing a dedicated thread policy. At the same time, it is expectable that the Sycl standard implemented by oneAPI moves more strongly towards Intel GPUs, and thus, additional optimizations like `launch_bounds` might be introduced.

	Single GEMM			Double GEMM		
	Measured	Theor.	%	Measured	Theor.	%
Iris Xe Max	101.35	118.40	85.60	290.97	358.61	81.14
UHD Graphics P630	49.59	62.20	79.73	58.67	188.39	31.67

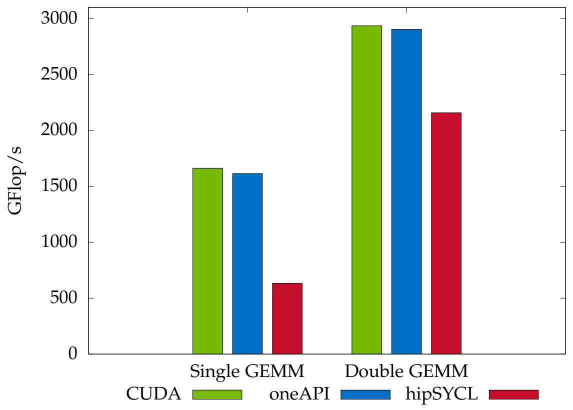
Table 4.2.: Analytical results (in GFlop/s) of the Roofline Model analysis. The benchmarks reach more than 80 % of Intel Iris Xe effective peak performance for both kernels, whereas they can't exploit the Intel UHD P630 effectively.

4.2.2. Roofline Model Analysis with CUDA backend

In the following, similar experiments are repeated using Sycl compiled with CUDA backend. The obtained results are represented in a shortened manner and without discussing them at a deeper level. Recall that all experiments are performed with single precision and repeated 20 times.



(a) Bandwidth



(b) Benchmarks

Figure 4.10.: Averaged Bandwidths (left) and performance of GEMM benchmarks (right) obtained with Nvidia RTX 3090 measured with native CUDA, oneAPI (open-source), and hipSYCL using single precision. Though oneAPI is very close to CUDA, hipSYCL can't reach the same performance. This discrepancy may be explained by the different underlying PTX generators.

The bandwidth measurements for all three platforms are plotted in Figure 4.10a with fewer data sets sizes than for Intel GPUs. Again, the maximum value accomplished at 2.0 GB of occupied memory is selected for the Roofline Model analysis, which is 774.72 GB/s. Once more, the data show a three-stage difference between all platforms. The average Flop/s of both the single and double GEMM using a 2 GB dataset are available in Figure 4.10b.

In this measurement, oneAPI (open-source) scores very closely to CUDA (97.13 % of 1662.29 GFlop/s in the single GEMM and 98.95 % of 2935.52 GFlop/s in the double GEMM benchmark). hipSYCL, on the other hand, doesn't seem to be able to apply the code optimizations (38.07 % and 73.51 %, respectively). This difference between the two

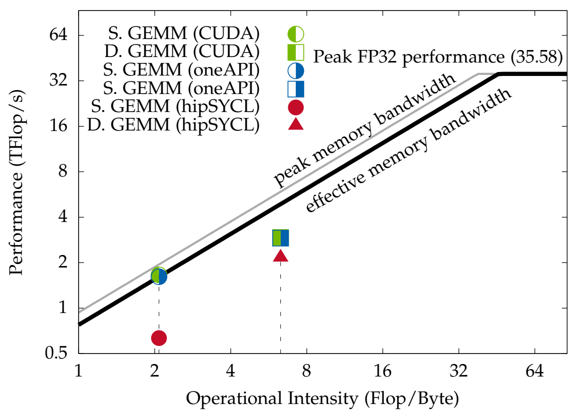


Figure 4.11.: Roofline Model for native CUDA, oneAPI (open-source), and hipSYCL. oneAPI and CUDA slightly exceed the effective memory bandwidth with the single GEMM, but stay below with the double GEMM benchmark. hipSYCL remains behind the theoretical and effective limits.

Sycl implementations seems to depend on the underlying compilers of oneAPI and hipSYCL: hipSYCL calls the CUDA compiler nvcc for PTX generation, whereas oneAPI uses clang. The huge difference between native CUDA and hipSYCL is therefore surprising.

Last, Figure 4.11 plots all measurements in a Roofline Model. CUDA and oneAPI slightly exceed the effective bandwidth with the single GEMM (103.16 % and 100.20 % compared to the theoretical peak, respectively). On the other hand, there is a need for improvement of the code generation considering the double GEMM benchmark (60.15 % and 59.52 %). hipSYCL remains at the bottom (39.28 % and 44.20 %). This model demonstrates that the performance of Sycl with oneAPI is on par with CUDA for these particular benchmarks. However, it must be noted that even a marginal difference scales to a large one when these kernels are executed tremendously often, which is to be expected when running SeisSol.

4.3. SeisSol

This section evaluates SeisSol performance with all introduced changes into its heterogeneous components and thus allows to compare different programming models, i.e. CUDA and Sycl. First, the proxy variant of SeisSol is executed to illustrate the expectable performance of its compute kernels and then the simulation with LOH1 and TPV5 benchmarks.

4.3.1. SeisSol-Proxy

The proxy application emulates SeisSol's single-node performance by running the main compute kernels as introduced in Section 2.3.1. The proxy is parameterizable with individual kernels and aims to give an overview of the overall performance in SeisSol, including all optimization techniques.

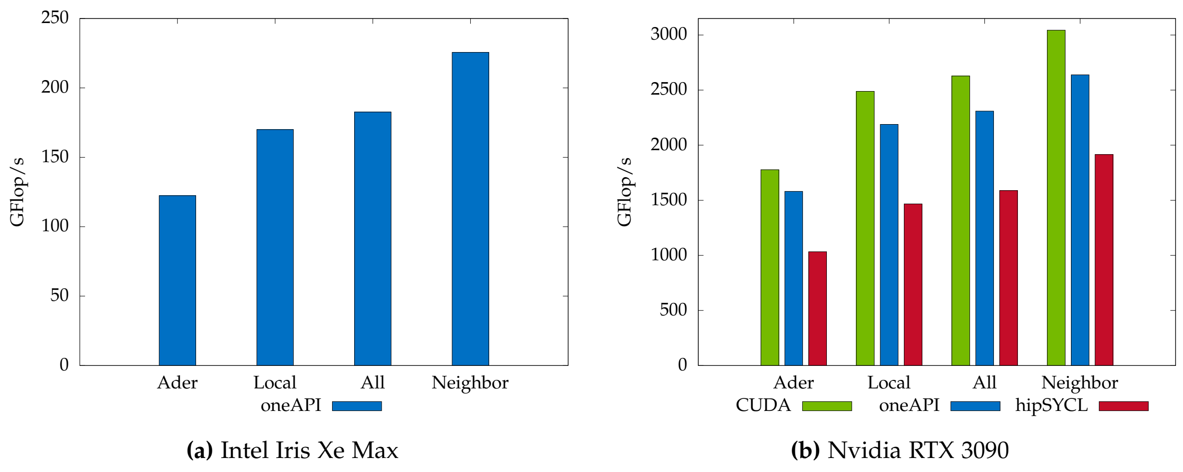


Figure 4.12.: Hardware Flops of the main compute kernels of SeisSol executed by the proxy application using 100000 cells, 40 time steps, single precision and a convergence order of six. Left: performance results obtained with Intel Iris Xe Max. Right: Nvidia RTX 3090. The slightly better performance of CUDA compared to oneAPI (open-source) pays off with the frequent repetitions of these kernels.

Figure 4.12 shows the performance counters of all generated kernels using Intel Iris Xe Max (left) and Nvidia RTX 3090 (right) using 100000 cells, 40 time steps, a convergence order of six, and single-precision averaged by ten repetitions. As mentioned in the sections before, SeisSol can't run on UHD Graphics P630 due to the limited local thread sizes per workgroup. Similar to preceded work [2], *Ader* is denoted as the implementation of the *Time kernel* and *Local* or *Neighbor* as respective implementations of Equation 2.3. Because Iris Xe is a low-end GPU and thereby was not designed for the execution of large-scale simulations, it is assumed that comparing its results from the proxy with other GPUs is not productive. Therefore, Figure 4.12a stays without additional comments and future studies could fruitfully use it a as a reference. On the other hand, Figure 4.12b shows the performance obtained with RTX 3090: As expected in the previous section, the small difference between oneAPI (open-source) and CUDA scales with a larger count of iterations, here up to 13.31 % at the *Neighbor* kernel. Therefore, one can conclude that, is not a suitable replacement for CUDA at the current state of SeisSol, in spite of the fact that Sycl and oneAPI can unify and ease programming across Nvidia and Intel. The same holds for hipSYCL, where the performance deviations are particularly large compared to native CUDA (37.07 % at the *Neighbor* kernel).

4.3.2. Benchmarks

The last experiment executes LOH1[5] and TPV5 benchmarks[6] of SeisSol. The Layer Over Half-space (LOH1) simulates elastic wave propagation driven by a source point for earthquake nucleation. As suggested by Harris et al. [37], TPV5 nucleates the earthquake rupture in a square zone of the fault surface. While evolving away from the nucleation, the rupture encounters two artificial square patches with initial stress conditions different from the fault. Both benchmarks are executed solely on the Heisenbug machine using 24 cores and two RTX 3090. However, these measurements show that Sycl-standardized code works correctly. Thus, SeisSol should also run accurately on Intel HPC GPUs.

	LOH1		TPV5	
Max. ϵ_{rel}	u, v, w	σ_{ij}	u, v, w	σ_{ij}
oneAPI (open-source)	0.4	$7.24 \cdot 10^{-6}$	0.87	0.19
hipSYCL	0.0	0.0	0.0	0.0

Table 4.3.: Relative maximum error compared to native CUDA occurred at LOH1 (left half) and TPV5 (right half) benchmark using oneAPI (open-source) and hipSYCL. Whereas hipSYCL produces exact results, oneAPI results in a numerical error, which can be attributed to the different PTX code generation.

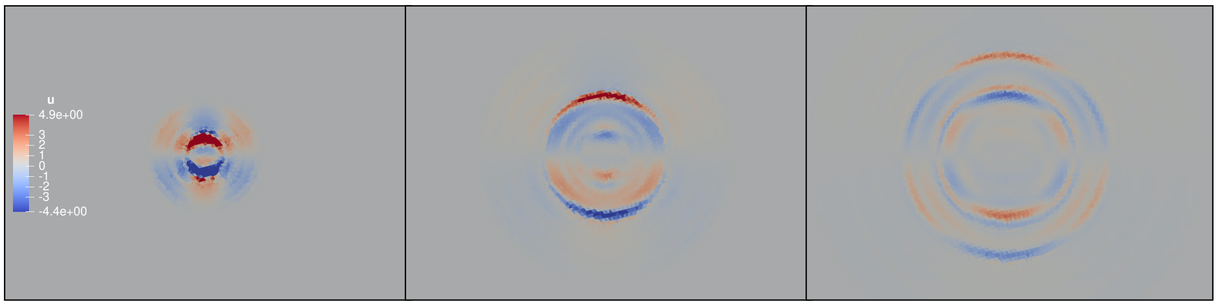
First, the numerical results of both simulations are examined by comparing the output for stresses σ_{ij} and the particle velocities u, v, w to ensure the overall correctness of this work. SeisSol allows writing its output over time in Hierarchical Data Format[7] (HDF5) files, which

[5] `https://seissol.readthedocs.io/en/latest/pointsource.html`
[6] `https://seissol.readthedocs.io/en/latest/tpv5.html`
[7] `https://www.hdfgroup.org/solutions/hdf5/`

can be compared block and element-wise with a constant δ by `h5diff`. As the σ_{ij} are measured in Mega Pascal (MPa), whereas the u, v, w are normalized, the data is grouped into stresses and velocities for analysis. The following explanations are also summarized in Table 4.3.

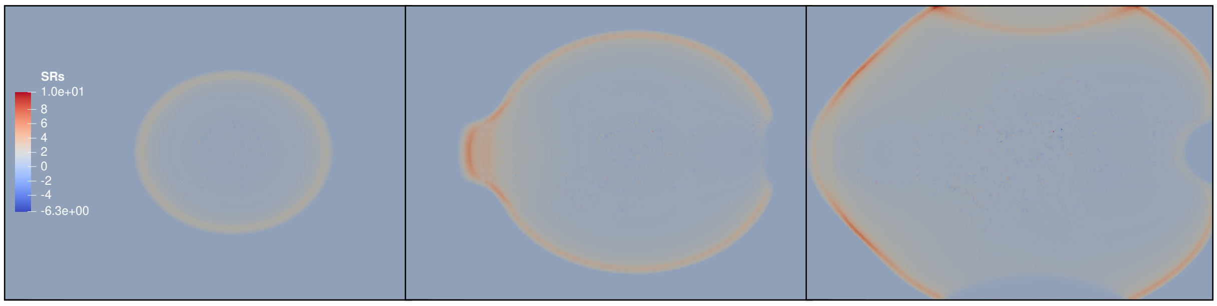
Considering oneAPI (open-source) and LOH1, an absolute maximum error $\epsilon_{abs} \approx 0.01 - 0.0006 = 0.004$ or $\epsilon_{rel} \approx \frac{2}{5}$ of u at position [5 12571] compared to native CUDA is observable. For σ_{xy} at [6 12571], it is $\epsilon_{abs} \approx 2.76253 \cdot 10^{11} - 2.76251 \cdot 10^{11} = 2 \cdot 10^{6}$ or $\epsilon_{rel} \approx 7.24 \cdot 10^{-6}$ as the overall maximum error. For hipSYCL, the velocities and stresses match exactly the CUDA results ($\epsilon = 0.0$).

With TPV5 and oneAPI (open-source), the measurements deviate with $\epsilon_{abs} \approx 0.0131 - 0.0017 = 0.0114$ and $\epsilon_{rel} \approx 0.87$ at u and position [13 1608610] respective $\epsilon_{abs} \approx 3.6021 \cdot 10^{7} - 2.9084 \cdot 10^{7} = 6.937 \cdot 10^{6}$ and $\epsilon_{rel} \approx 0.19$ with σ_{yy} at [20 1663914]. Again, the results are exact with hipSYCL.



(a) u at $t = 1$ s **(b)** u at $t = 2$ s **(c)** u at $t = 3$ s

Figure 4.13.: Particle velocity u in LOH1 benchmark at $t = 1$ s (left), $t = 2$ s (center), and $t = 3$ s (right) calculated by oneAPI and rendered in ParaView. From a visual perspective, the results match with native CUDA.



(a) SRs at $t = 3$ s **(b)** SRs at $t = 4$ s **(c)** SRs at $t = 5$ s

Figure 4.14.: Slip rates along strike direction (SRs) in TPV5 benchmark at $t = 3$ s (left), $t = 4$ s (center), and $t = 5$ s (right) calculated by oneAPI and rendered in ParaView. From a visual perspective, these rates match with the output reported in the SeisSol documentation.

These discrepancies are attributable to the different PTX code generators of hipSYCL and oneAPI as mentioned in the section before. Nevertheless, since hipSYCL produces exact results and the error of oneAPI stays within an acceptable range (it involves only a negligible amount of meshes), it is expected that portability is working correctly for SeisSol on both platforms. From a visual perspective, this can be confirmed by comparing the outputs visualized in ParaView. Figure 4.13 and 4.14 show two exemplary values evolving over time for both benchmarks compiled with oneAPI. Comparing these outputs with the example figure stated in the SeisSol documentation[8] and using CUDA, no differences are perceptible.

The performance results of both benchmarks are available in Figure 4.15 for the same experiment but with disabled output. Recall that in any simulation in SeisSol, there are also calculations on the CPU involved. Due to this, the Flop count of the two benchmarks differs in general, because LOH1 has lower computational dependencies on CPUs than TPV5. Additionally note that dynamic rupture Flop/s used at TPV5 are not included in the measurements, but the time spent on the respective compute kernels is. Therefore, the Flop/s count is lower overall for TPV5.

Once again, the characteristic difference between all platforms is observable, which is the highest for LOH1. Here, oneAPI (open-source) and hipSYCL reach 72.07 % or 58.94 % of 4467.99 GFlop/s obtained with native CUDA. The discrepancies mitigate for TPV5 (85.82 % and 73.79 % of 2605.39 GFlop/s) due to its dependency on CPUs but are still notable. Therefore, as in the previous subsection, the created portability can currently not be considered as a replacement of native CUDA. However, a continuation of these explorative studies is desirable for future work.

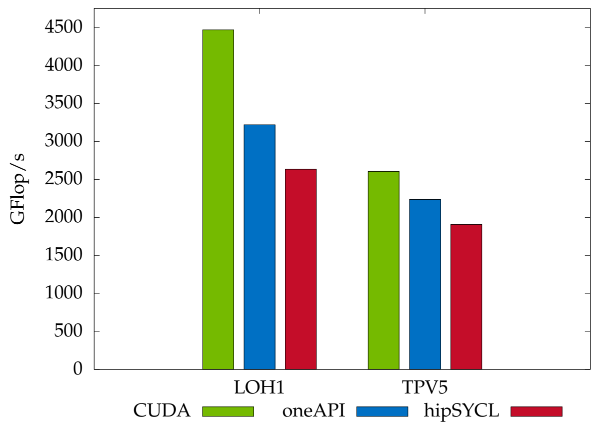


Figure 4.15.: Flop/s achieved by LOH1 and TPV5 benchmark with disabled output on an AMD EPYC 7662 using 24 cores and two Nvidia RTX 3090. The higher dependency of TPV5 on the CPU reduces the efficiency of the deployment of GPUs compared to LOH1.

[8]`https://seissol.readthedocs.io/en/latest/tpv5.html`, bottom

5. Conclusion

SeiSol is an open-source software for numerical simulations of earthquakes and seismic waves, implemented as an MPI+X model, where X is replaceable with OpenMP, CUDA, and HIP. Due to the announcement that the next upgrade of SuperMUC-NG is going to include Intel's new generation of HPC GPUs, it becomes necessary to adapt SeisSol's heterogeneous components for the Sycl standard. Additionally, a broad variety of modern GPUs and their programming models could require the need for cross-platform development in the future.

This thesis presented the extended versions of Device and GemmForge based on the Sycl standard that allows executing SeisSol simulations on both Intel and other GPUs. Investigating Performance Portability using a discrete prototype GPU Intel Iris Xe Max with the oneAPI platform revealed that the portability achieves 85.60 % of its effective peak performance using single GEMM benchmarks, which is superior to Intel's integrated SoC design. However, it was found that Intel MPI was not yet ready for use in HPC data centers due to its low efficiency in collective operations with GPU pointers in direct comparison to CUDA-aware OpenMPI without UCX layer.

As an exploratory study, similar experiments with Intel Arria 10 FPGA and Nvidia RTX 3090 were conducted, showing that Sycl's cross-platform approach works, but its efficiency highly depends on a specific implementation of the standard. hipSYCL from the University of Heidelberg, for example, could not exploit the underlying hardware as much as oneAPI compiled as open-source variant compared to native CUDA. Yet, none of both investigated platforms exceeded or caught up with the performance of the native CUDA implementation. When running SeiSol proxy application, even a small difference between oneAPI and CUDA in terms of performance added up to over 13 %, justifying the fine-tuned kernels employed by SeisSol. Therefore, this direct portability is not a replacement for native programming models in its current state.

The limited performance of Iris Xe Max did not allow to execute an earthquake simulation on this GPU. Instead, RTX 3090 was used and the experiments showed correct results with Sycl, even exact ones with hipSYCL in comparison to CUDA. Therefore, it is assumable that SeiSol is working correctly once the new Intel GPUs are available.

In the next months, it is expectable that Intel will release its highly discussed Iris Xe HPG GPU "DG2". The author suggests re-executing SeiSol on this graphics card and perform a competitive performance analysis against related Nvidia and AMD GPUs and derive issues or bottlenecks of this early portability. Assessing GPU-aware Intel MPI implementation is also crucial for that. Furthermore, investigating Sycl combined with FPGAs could be an appealing task to future researches.

A. Figures

```
#pragma omp target enter data map(to: A [:size], B [:size])
#pragma omp target teams distribute parallel for
for (size_t i = 0; i < size; i++)
{
    A[i] = scalar * A[i] + B[i];
}
#pragma omp target exit data map(from: A [:size])
```

Figure A.1.: Shortened Saxpy example with OpenMP.

```
cl::sycl::queue q {cl::sycl::gpu_selector{}};
float *devA = (float *)malloc_device(size, q);
float *devB = (float *)malloc_device(size, q);
q.memcpy(devA, A, size);
q.memcpy(devB, B, size);
q.wait();

auto rng = cl::sycl::nd_range<1>{{N}, {1}};
q.submit([&](cl::sycl::handler &cgh) {
    cgh.parallel_for(rng, [=](cl::sycl::nd_item<1> item) {
            devA[item.get_global_id(0)] = scalar *
            devA[item.get_global_id(0)] + devB[item.get_global_id(0)];
    });
}).wait();

q.memcpy(A, devA, size);
q.wait();
```

Figure A.2.: Shortened Saxpy example with Sycl.

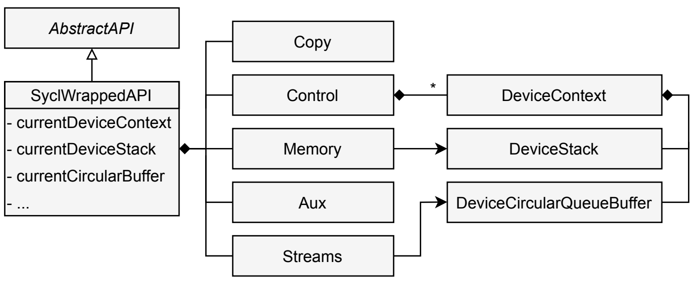


Figure A.3.: Final design of the Sycl component of the Device repository.

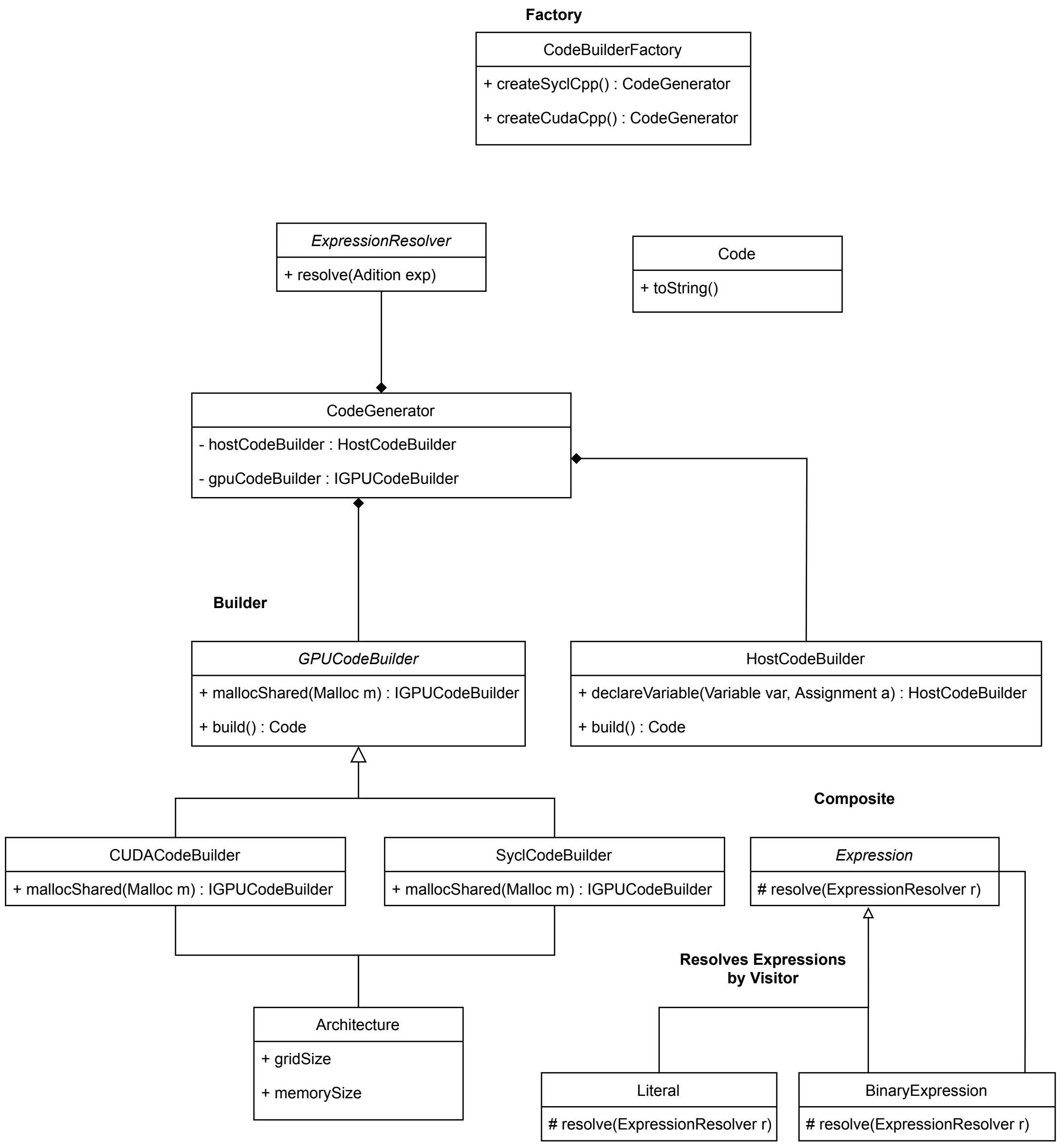

Factory
CodeBuilderFactory
+ createSyclCpp() : CodeGenerator
+ createCudaCpp() : CodeGenerator
ExpressionResolver
+ resolve(Adition exp)
Code
+ toString()
CodeGenerator
- hostCodeBuilder : HostCodeBuilder
- gpuCodeBuilder : IGPUCodeBuilder
Builder
GPUCodeBuilder
+ mallocShared(Malloc m) : IGPUCodeBuilder
+ build() : Code
HostCodeBuilder
+ declareVariable(Variable var, Assignment a) : HostCodeBuilder
+ build() : Code
Composite
CUDACodeBuilder
+ mallocShared(Malloc m) : IGPUCodeBuilder
SyclCodeBuilder
+ mallocShared(Malloc m) : IGPUCodeBuilder
Expression
# resolve(ExpressionResolver r)
Resolves Expressions by Visitor
Architecture
+ gridSize
+ memorySize
Literal
# resolve(ExpressionResolver r)
BinaryExpression
# resolve(ExpressionResolver r)

Figure A.4.: Initial design considerations for GemmForge.

List of Figures

List of Tables

Bibliography

[1] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A. A. Gabriel, C. Pelties, A. Bode, W. Barth, X. K. Liao, K. Vaidyanathan, M. Smelyanskiy, and P. Dubey. "Petascale High Order Dynamic Rupture Earthquake Simulations on Heterogeneous Supercomputers". In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* 2015-Janua.January (2014), pp. 3–14. ISSN: 21674337. DOI: 10.1109/SC.2014.6.

[2] R. Dorozhinskii and M. Bader. "SeisSol on Distributed Multi-GPU Systems: CUDA Code Generation for the Modal Discontinuous Galerkin Method". In: *ACM International Conference Proceeding Series* (2021), pp. 69–82. DOI: 10.1145/3432261.3436753.

[3] S. Dominick. *Performance portability and evaluation of heterogeneous components of SeisSol targeted to AMD GPUs*. 2021.

[4] C. Uphoff and M. Bader. "Yet Another Tensor Toolbox for Discontinuous Galerkin Methods and Other Applications". In: *ACM Transactions on Mathematical Software* 46.4 (2020). ISSN: 15577295. DOI: 10.1145/3406835. arXiv: 1903.11521.

[5] IntelCorporation. *Intel Processor Graphics Gen11 Architecture*. 2019.

[6] S. Junkins. *The Compute Architecture of Intel ® Processor Graphics Gen8*. 2014.

[7] D. Blythe. *The Xe GPU Architecture*.

[8] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C. K. C. Luk. "Performance Characterisation and Simulation of Intel's Integrated GPU Architecture". In: *Proceedings - 2018 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2018* (2018), pp. 139–148. DOI: 10.1109/ISPASS.2018.00027.

[9] T. Deakin and S. McIntosh-Smith. "Evaluating the performance of HPC-style SYCL applications". In: *ACM International Conference Proceeding Series* (2020). DOI: 10.1145/3388333.3388643.

[10] B. Li, H. C. Chang, S. Song, C. Y. Su, T. Meyer, J. Mooring, and K. W. Cameron. "The power-performance tradeoffs of the intel Xeon Phi on HPC applications". In: *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS* (2014), pp. 1448–1456. ISSN: 23321237. DOI: 10.1109/IPDPSW.2014.162.

[11] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz. "Comparative performance analysis of Intel Xeon Phi, GPU, and CPU: A case study from microscopy image analysis". In: *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS* (2014), pp. 1063–1072. ISSN: 23321237. DOI: 10.1109/IPDPS.2014.111.

[12] S. Christgau and T. Steinke. "Porting a Legacy CUDA Stencil Code to oneAPI". In: *Proceedings - 2020 IEEE 34th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020* (2020), pp. 359–367. DOI: `10.1109/IPDPSW50202.2020.00070`.

[13] T. Cramer, D. Schmidl, M. Klemm, and D. Mey. "OpenMP Programming on Intel Xeon Phi Coprocessors : An Early Performance Comparison". In: *European Conference on Parallel Processing* (2012), pp. 547–558.

[14] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler. "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA". In: Association for Computing Machinery (ACM) (2017), pp. 1–6. DOI: `10.1145/3110355.3110356`.

[15] M. Martineau, S. Mcintosh-smith, C. Bertolli, A. C. Jacob, S. F. Antao, A. Eichenberger, G.-t. Bercea, T. Chen, T. Jin, K. O. Brien, and G. Rokos. "Performance Analysis and Optimization of Clang ' s OpenMP 4 . 5 GPU Support". In: *th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (2016), pp. 54–64. DOI: `10.1109/PMBS.2016.11`.

[16] A. Munshi. "The OpenCL Specification - Version 1.0". In: *Khronos Group Specifications* (2009).

[17] B. Gaster. *OpenCL An Introduction for HPC programmers(SC 2010)*. 2010.

[18] R. Banger and K. Bhattacharyya. *OpenCL Programming by Example*. Packt Publishing LTD, 2013, p. 112. ISBN: 9781849692342.

[19] L. Howes and A. Munshi. "OpenCL Extension Specification". In: *Khronos Group Specifications* (2015).

[20] B. Gaster. *The OpenCL C ++ Wrapper API*. 2010.

[21] J. Fang, A. L. Varbanescu, and H. Sips. "A comprehensive performance comparison of CUDA and OpenCL". In: *Proceedings of the International Conference on Parallel Processing* (2011), pp. 216–225. ISSN: 01903918. DOI: `10.1109/ICPP.2011.45`.

[22] H. C. D. Silva, F. Pisani, and E. Borin. "A comparative study of SYCL, OpenCL, and OpenMP". In: *Proceedings - 28th IEEE International Symposium on Computer Architecture and High Performance Computing Workshops, SBAC-PADW 2016* (2017), pp. 61–66. DOI: `10.1109/SBAC-PADW.2016.19`.

[23] *Sycl 2020 Specification (Revision 3)*. 2020.

[24] Z. Jin and H. Finkel. "Evaluation of Medical Imaging Applications using SYCL". In: *Proceedings - 2019 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2019* (2019), pp. 2259–2264. DOI: `10.1109/BIBM47256.2019.8982983`.

[25] B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, and J. Sewall. *Data Parallel C++*. Springer Nature, 2020, pp. 1–2. DOI: `10.1145/3388333.3388653`.

[26] A. Alpay and V. Heuveline. "SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL". In: *ACM International Conference Proceeding Series* (2020), p. 3388658. DOI: `10.1145/3388333.3388658`.

[27] M. Käser and M. Dumbser. "An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes - I. The two-dimensional isotropic case with external source terms". In: *Geophysical Journal International* 166.2 (2006), pp. 855–877. ISSN: 0956540X. DOI: 10.1111/j.1365-246X.2006.03051.x.

[28] M. Dumbser and M. Käser. "An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes - II. The three-dimensional isotropic case". In: *Geophysical Journal International* 167.1 (2006), pp. 319–336. ISSN: 0956540X. DOI: 10.1111/j.1365-246X.2006.03120.x.

[29] C. Pelties, J. De La Puente, J. P. Ampuero, G. B. Brietzke, and M. Käser. "Three-dimensional dynamic rupture simulation with a high-order discontinuous Galerkin method on unstructured tetrahedral meshes". In: *Journal of Geophysical Research: Solid Earth* 117.2 (2012). ISSN: 21699356. DOI: 10.1029/2011JB008857.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software*. 1996, p. 395. ISBN: 020163361-2. arXiv: dd.

[31] C. Rivera, J. Chen, N. Xiong, J. Zhang, S. L. Song, and D. Tao. "TSM2X: High-performance tall-and-skinny matrix–matrix multiplication on GPUs". In: *Journal of Parallel and Distributed Computing* 151 (2021), pp. 70–85. ISSN: 07437315. DOI: 10.1016/j.jpdc.2021.02.013. arXiv: 2002.03258.

[32] S. J. Pennycook, J. D. Sewall, and V. W. Lee. "Implications of a metric for performance portability". In: *Future Generation Computer Systems* 92 (2019), pp. 947–958. ISSN: 0167739X. DOI: 10.1016/j.future.2017.08.007.

[33] S. Williams, A. Waterman, and D. Patterson. "Roofline: An insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76. ISSN: 00010782. DOI: 10.1145/1498765.1498785.

[34] M. Parker. "Understanding Peak Floating-Point Performance Claims". In: *Intel FPGA White Paper* (2016), p. 6. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf.

[35] S. Asano, T. Maruyama, and Y. Yamaguchi. "Performance comparison of FPGA, GPU and CPU in image processing". In: *FPL 09: 19th International Conference on Field Programmable Logic and Applications*. IEEE, 2009, pp. 126–131. ISBN: 9781424438921. DOI: 10.1109/FPL.2009.5272532.

[36] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang. "Understanding Performance Differences of FPGAs and GPUs". In: *Proceedings - 26th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018* (2018), pp. 93–96. DOI: 10.1109/FCCM.2018.00023.

[37] R. A. Harris, B. Aagaard, M. Barall, S. Ma, D. Roten, K. Olsen, B. Duan, D. Liu, B. Luo, K. Bai, J. P. Ampuero, Y. Kaneko, A. A. Gabriel, K. Duru, T. Ulrich, S. Wollherr, Z. Shi, E. Dunham, S. Bydlon, Z. Zhang, X. Chen, S. N. Somala, C. Pelties, J. Tago, V. M. Cruz-Atienza, J. Kozdon, E. Daub, K. Aslam, Y. Kase, K. Withers, and L. Dalguer. "A suite of exercises for verifying dynamic earthquake rupture codes". In: *Seismological Research Letters* 89.3 (2018), pp. 1146–1162. ISSN: 19382057. DOI: 10.1785/0220170222.