universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

HybChive - Experimental programming framework for heterogeneous architectures

verfasst von / submitted by:

Jan Clemens Stoffregen

Angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of: Diplom-Ingenieur (Dipl.-Ing.)

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet: A 066 940

Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet: Masterstudium Scientific Computing / Master's degree programme Scientific Computing

Betreut von / Supervisor:
Prof. Dr. Siegfried Benkner

Wien, 2018 / Vienna, 2018

Contents

Chapter 1

Introduction

1.1 Motivation

The demand of computing power in diverse fields of science and economy research subjects is indefinite. A large need for supercomputers can not only be found in the branches weather prediction, biochemistry and astrophysics but also in large companies that deal with large amounts of data. Powerful computing devices are generally needed to solve complex mathematical problems, to be able to deal with large amounts of data or to deal with a certain amount of data in a limited time period. Since many computational tasks can be processed in parallel, there will be always the attempt to use existing hardware in parallel, no matter how fast the hardware is. When expanding the existing hardware, financial aspects have to always be considered as well. Since there was a huge demand for well designed GPUs, because the gaming industry and the market for TVs and good computer screens is extremely large, GPUs with a large computing power became fairly cheap and since the started to offer general purpose functions beginning from 2000, GPGPUs became fairly attractive for scientists who wanted to execute certain mathematical tasks in parallel and while the budget for investments in parallel hardware was limited.[11] Since new hardware does not often replace

but extend existing hardware, hybrid architectures developed widely, also because individual hardware is always more suitable for only a limited field of computational tasks. Considering that the hardware is produced by different vendors and that the wide spread of diverse parallel hardware, such as GPGPUs by NVIDIA and AMD and Xeon Phis by Intel, one must consider that many devices have to be addressed with a wide range of parallel computer languages. NVIDIA, for example, has created an own language to address their GPGPUs, which is named CUDA[1]. This language is mainly designed to work with GPUs that are designed by NVIDIA. The counterpart of CUDA is OpenCl[2], designed by the Khronos Group[3]. Unlike CUDA, OpenCl can address GPGPUs by many vendors as well as other kinds of devices. Since CUDA is especially designed for the use of NVIDIA GPGPUs it may often be slightly faster than programs for the same functionality written in OpenCl though the performance of programs of both languages is often comparable.[7] Next to those two languages, the language MPI[4] has to be mentioned as well. This language can program a computercluster consisting of many nodes. The GPGPUs and other devices on those nodes can be also addressed by for example including OpenCl kernels in the MPI program. Another language is OpenMP[5] OpenMP is used to exploit the use of shared memory devices while MPI is preferred to use distributed memory architectures. Lastly, Intel TBB[6] is a language designed by Intel to address multi-core processors, for example the Xeon Phi.

Since local, physical servers might be necessary for some companies, due to data security regulations, other companies have the choice between many

[1] CUDA is the acronym for: "Compute Unified Device Architecture"

[2] OpenCl is the acronym for: "Open Computing Language"

[3] The Khronos Group focuses on the creation of open standard, royalty-free application programming interfaces (APIs) for authoring and accelerated playback of dynamic media on a wide variety of platforms and devices[13]

[4] MPI is the acronym for "Message Passing Interface"

[5] OpenMP is the acronym for "Open Multi-Processing"

[6] Inte TBB is the acronym for "Intel Threading Building Blocks"

virtual servers, consisting of the diverse kinds of devices mentioned above. Amazon Web Services is one of the biggest service in the cloud at the moment that offers the use of supercomputing devices online. Nevertheless, vendors like Aliyun, Microsoft Azure, Outscale, Peer 1 hosting, Rapidswitch and more cloud services are listed as possible cloud services for the use of GPGPUs online by NVIDIA as well.[14] Since the number of cloud services that offer the online use of GPGPUs is rising, it can be also assumed that the use of this kind of hardware is getting more popular. After having made the decision that a scientific institution or a company is in need of a high performance computing architecture, the following answers have to be answered:

- Which kind of architecture is needed and which kind of devices are needed? How fast will they be? How expensive will the usage be?
- Which computer language should be used? Is it possible for the company to create programs that exploit the given architecture? Do they have the specialists to use the chosen programming language?
- Is it possible to integrate the use of the new high performance architecture into the existing programs and internal processes?

To make an investment in high performance computing architecture that will be beneficial, all of those questions have to be answered.

1.2 Existing needs for the usage of high performance computing architectures

1.2.1 Easy Usability

The user of a high performance architecture is usually a researcher working in a research institution, for example in the weather prediction branch, or a specialised employee working at a company, for instance at the accountancy department. Since all specialists need further education on how to use high

performance computing architecture, one essential need of every specialist is to be able to learn fast how to use the architecture provided. Usually, specialists would prefer to invest more working hours to investigate the field that they are specialised in than learning how to program in a new computer language. Before a new architecture is introduced to the work environment of the user, the user often already performs the given tasks, only in another way. Naturally, it happens less often that a new architecture is introduced to perform a whole new task in a company, for example a new research procedure. More often, the hardware is being purchased to accelerate existing processes. In this case, the user would like to integrate the usage of the new architecture into his existing environment, for example into the code that he is already working with. Also, if the user has already a working code that he worked with before the introduction of the new hardware, he or she would like a certain portability of the code or of certain function calls that he already works with. The ideal scenario would be that the user can take the existing code and just keep on working with this code without any changes needed while the code uses the additional hardware in the background. This scenario would involve changing parts of the code in the background, by other programmers who have specialised in a programming language that can address the new hardware[7], without having to change the function call itself that is being used by the user. In addition, it would be also an advantage for the user if new features of the function call could be implemented without the necessity for the user to change anything. The creation of such a framework that provides all given desired features requires the participation of implementation experts who add implementations to a certain existing framework or structure. In addition, the framework needs to be able to be changed at certain parts without the necessity to change the implementations.

[7]In the following content of this thesis, those specialists will be called implementation experts.

1.2.2 Optimisation of high performance architectures

The question of how a computing environment, for example of a university or a big company, should be optimised includes many trade offs. While a company wants to execute the sum of all programs as fast as possible, a low energy consumption is demanded as well which includes not using the highest computing power level because this would cause a lower electricity efficiency level. It is always beneficial for a company if both measures, the electricity consumption and the system level performance of the computing architecture are easy to measure. While the electricity consumption can be measured easily it is already quite complicated to define the system level performance of a computing architecture. The throughput[8] for all running programs of a company for example is a weak measure for the performance of the hardware architecture since all programs contain different amounts of workloads. It would be ideal for the company if it would know exactly which program is executed by which user and how much value does the execution of this program contribute to the company value. Gaining this information is of course utopic since even the single programs of the users are not evenly valuable. In this thesis the assumption is being made that the company wants to optimise the utilisation of the hardware architecture. The company wants that all parts of the hardware are used as much as possible and with the highest computing power possible. A possible framework has to provide a possibility that addresses this need of a company to adjust which kind of trade off the company wants to go for.

A user, being an employee of the company, normally wants his or her functions to be executed as fast as possible. This is another trade off that exists for the company. While the company wants to optimise the utilisation of the cluster, some programs of the users might block the whole architecture while the program of the user is not using all parts fully. While the utilisation of the cluster might be optimised by users sharing the existing hardware,

[8] The throughput is defined as the sum of instructions per cycle of an architecture[2]

it might be faster for the user to use all parts of the hardware. Focusing more on the user, she or he also would like an automatic optimisation of their programs. While this is partly addressed by compilers, for example by loop unrolling, other optimisations of a single code can't be done without additional information provided by the user. A possible framework that addresses the optimisation of the execution time of one program has to include the possibility to execute asynchronous function calls. This cannot be fully automatically optimised by a compiler since not all data dependencies can be identified automatically. For the optimal execution time of a program, the user has to have the possibility to mark data dependencies so that functions can be executed in parallel. To satisfy the users, the company and the designer of a possible framework one has to also consider that the time until the code of the user is executed can't be too long. While the utilisation of the cluster might be perfect it can happen that some programs are never executed because they take predictably too much time on the whole cluster. This scenario must be addressed to verify that all programs have been executed after a certain amount of time. This thesis focuses on the optimisation of the execution time of the program of a single user while certain structures to consider the other needs are already being established.

1.3 Implementation Experts

1.3.1 Definition of Implementation Experts

Considering the fact that there are already many different programming languages that have to be used to use different kinds of hardware, many owners and users of such a computing architecture hire specialised programmers who can program in the respective language. They are also able to optimise the respective program. Since the optimal parameters of such a program depend on the single individual program and also sometimes on the mathematical problem, a company needs to optimise the parameters for each code individ-

ually. The specialised programmers are called implementation experts in this thesis. Looking daily on the online hiring freelancer platforms Upwork.com and Freelancer.com one can observe that jobs for programming parallel devices are in growing demand.

1.3.2 The needs of implementation experts

The needs of implementation experts and the end user are similar. Both want to invest as little work as possible to produce a perfect outcome. Since the user normally already works in a certain environment, the implementation expert wants to add on to this environment easily. The implementation expert does not want to fully change or reprogram the whole program of the user. Furthermore, the implementation expert wants to test the new program after it has been finished. To be able to test their newly developed program, it would be comfortable to take the existing program or function of the user and compare the output of both programs or functions with different input parameters. Since most existing functions address existing hardware, it might be even comfortable to add the new function to the existing one and execute both functions in parallel. The existing function of the user could be for a CPU for example while the new function is for a GPU. In this scenario, the user could execute his functionality even faster if he uses both the CPU and the GPU in parallel. A framework that addresses those needs must allow to integrate the existing function of the user while adding a new function as well. Since there is a growing market for parallel programming code and since the development of some programs might be very time intensive, it might be also comfortable for the implementation expert if he could offer his implementations for multiple users, for example as a part of a closed source library.

1.4 The objectives of this thesis

This thesis aims at designing a framework that can provide all desired properties identified in the sections above. The framework should be easy to use for the user who is not a programming expert for high performance computing architectures. Ideally, the user should define in a single function call only the functionality of his fields related properties. The rest of function call that addresses the architecture should be optimised automatically. Furthermore, the function call of the framework should be able to be easily integrated into an existing program. The provided function call that calls a part of the framework should be able to use the whole architecture that the user can use. If the user can use one node with GPUs and Xeon Phis, the framework should provide the possibility for the user to use all those devices in addition to the CPU at the same time. The framework should also be able to generate and store performance data so it can be reused in next function call. In this thesis, it is not considered that devices could be blocked by other users and it does not include any kind of dynamic scheduling in the design of the framework. Performance data has to be stored for all devices to calculate an optimal way to split the data between all devices so that the runtime is optimised at the end. In this thesis there are the following restrictions: There is only one user[9] using one node[10] for the execution of one function call[11] for an embarrassingly parallel program.[12] The framework has to include the possibility for implementation experts to easily add new alternative implementations.

[9] An environment with multiple users wanting to use the same devices is not considered in this thesis. This environment would require hybrid scheduling routines where the availability of all devices is checked and then blocked. In addition, this environment would benefit from considering the possibility to split the devies between users to optimise the utilisation of the whole cluster. Other aspects, for example the latest time until execution measure for each user, would have to be implemented as well.

[10] A cluster consisting of multiple nodes with multiple attached GPUs is not considered in this thesis

[11] Asynchronous functions and the markup of data dependencies are not being considered in this thesis

[12] More complicated data splitting patterns are not being considered in this thesis.

Chapter 2

Existing approaches

Numerous approaches exist to improve the programmability of heterogeneous high performance computing architectures. Those projects take different approaches which are shortly discussed in this chapter.

An wholistic approach to program high performance computing architectures takes Peppher which addresses the programmability and efficiency on given heterogeneous systems[1]. Peppher is an EU funded project with several participating parties consisting of universities, like the University of Vienna, and small and medium sized companies like Codeplay (from the United Kingdom). Peppher provides code - critical parts in multiple variants for different kinds of hardware. Those variants are statically preselected and then statically or dynamically scheduled. Those variants are annotated to performed an optimised scheduling process for given user objectives. A graph of tasks is defined and several criteria including data dependency is analysed by Peppher so that it can provide task parallelism. A runtime system executes a performance - and resource aware Peppher application and makes Peppher a wholistic approach which "targets the performance portability and efficient use of heterogeneous systems at several levels"[1].

The project Autotune analyzes a program during a single execution and returns recommended parameters for tuning the program.[9] It is developed

as an extension to the Periscope framework. Autotune is designed to be integrated in existing production programs and the tuning of the exising parameters. "The resulting Periscope Tuning Framework will be able to tune serial and parallel codes for multicore and manycore architectures and return tuning recommendations that can be integrated into the production version of the code."[9]

HyVM is providing a hybrid virtual machine including a runtime - system and an own programming model[6]. It addresses the challenges of non-uniform memory access and complex software stacks.[10] It offers a resource management support and it aims to provide a uniform program execution model. "The HyVM project is creating new hypervisor- and system-level abstractions in support of providing a uniform program execution model for future hybrid computing platforms. Rather than treating accelerators as external devices, the model anticipates future integrated systems by providing sets of virtual processing units for use by both accelerator and commodity programs, offering the resource management support needed to efficiently execute such parallel multi-core applications, and supplying the tool chains needed, at hypervisor level, to permit applications to freely use arbitrary combinations of accelerator and commodity cores."[6]

Petabricks provides a parallel language and a compiler[5]. It also aims to autotune algorithms for different convergence criteria. As an introduction, Petabricks is described as follows: "PetaBricks is a new implicitly parallel language and compiler where having multiple implementations of multiple algorithms to solve a problem is the natural way of programming. We make algorithmic choice a fundamental construct of the language. Choices are provided in a way that also allows our compiler to tune at a finer granularity. The PetaBricks compiler autotunes programs by making both fine-grained as well as algorithmic choices. Choices also include different automatic parallelization techniques, data distributions, algorithmic parameters, transformations, and blocking.

In addition, we introduce novel techniques to autotune algorithms for different convergence criteria. When choosing between various direct and iterative methods, the PetaBricks compiler is able to tune a program in such a way that delivers near-optimal efficiency for any desired level of accuracy. The compiler has the flexibility of utilizing different convergence criteria for the various components within a single algorithm, providing the user with accuracy choice alongside algorithmic choice."[5]

The Open Community Runtime framework focuses on cooperative resource control for exascale systems. As an introduction OCR is described as follows: "The Open Community Runtime project is creating a runtime framework that explores new programming methods for machines with high core count. The initial focus is on HPC applications. Its goal is to create a framework and reference implementation to help developers explore programming methods to improve the power efficiency, programmability, and reliability of HPC applications while maintaining app performance. OCR will help app developers with the complex process of writing multi-core apps by masking the effort to manage event-driven tasks, events (which embody dataflow and code flow dependencies), memory data blocks (with semantic annotations for runtime use), machine description facilities, and more."[15]

The work "Modular Infrastructure for Automation of Architectural Adaptation and Performance Optimization"[3] emphasises a good user experience as well as a automated approach to optimise the performance aplication - wise on a hybrid platform. This paper takes existing tools and combines them for a comprehensive approach to optimise the execution of an application on a hybrid system.

Hadoop is also discussed in this section since an approaches exist to combine Hadoop with other languages for hybrid computer architectures.[4] Since MapReduce is only feasible for certain algorithms, this approach won't become a comprehensive solution for the usage of hybrid systems.[8] Nevertheless, many approaches exist that attempt to optimise the usage of MapReduce

for as many algorithms as possible.[12]

Properties of the different approaches Technologies that aim at optimising the usage of heterogeneous clusters can be characterized with different properties. Some provide a common programming language model and are designed in a way to be computer language agnostic so that the user can choose the desired computer programming language to use the respective framework. Some approache provide portability by a virtual - machine - like layer and some provide component based portability. The tuning of the different parts is in some approaches performed automatically and others perform the tuning procedure through component annotation. Some provide runtime - system analysing tools which can react during runtime, and some stay with a static assumption about the behaviour of the architecture where the software is executed on. Within the conclusion of this thesis, the framework developed in this thesis will be characterised by this properties.

Chapter 3

HybChive

The introducing chapter describes how a first motivation originated to start this project. The following chapter describes the iterations of an experimental software development approach during which new insights are gained which lead to the respective following iteration. The third chapter describes which purpose the final framework targets and how it is meant to be used.

3.1 Introduction

The idea for this project originated by looking at a group of scientists who have access to a cluster where they can run their analysis. The cluster is continuously expanded with new devices but the scientists who are not programming experts have only one code that runs only on one device.[1] As a result, new devices that the existing code has to be changed for are not used while other devices are used extensively. The attempt to create a comprehensive framework that addresses this problem and optimises the usage of the cluster was taken into consideration.
The main goal of this project therefore was to define necessary components,

[1] To only use one additional GPU in parallel, broad changes are needed in the code which are likely to be too complicated for a large number of scientists.

bottlenecks and software architecture aspects that appear while creating a solution attempt for the problem discussed. The framework should not only contain a possibility to be easily extended with routines while the scientist do not have to change anything. An optimal framework should also schedule routine calls from multiple users so that all parts of the cluster can be used in an optimal way.[2] It is obvious, for example, that a cluster is used more efficiently if two users use two GPUs in parallel compared to the situation that both users only use one GPU after one another. It is also a valid question to ask if, while using a GPU, a CPU that the GPU is attached to can be used as well since it remains unused while the GPU completes its assinged task. Thus, the main features of this kind of a framework were defined to be programmability, so that it is easy for the scientist to add code to existing code to address a new device, portability, so that accessible devices can be used in different constellations and the possibility to use different devices in parallel in a heterogeneous environment.

3.2 Software development iterations

During the software development four iterations took place during which necessary software architecture components and programming aspects were defined.

3.2.1 First iteration

In this iteration, a simple C - program was created that runs two additional stand alone programs in parallel which have the same functionality and which both run on a CPU at the same time each computing part of the solution in parallel. The main program is called scheduler and the two stand alone

[2] Defining an optimal way is not distinct since it has to be defined individually

programs are called variants. A stand alone program is a program that can be compiled and executed individually without any additional components needed. The design of the variants as stand alone programs has some advantages. If it is kept in mind that, later on, the adding of variants for other devices, for example for GPUs or Xeon Phis, must be possible and that different compiler flags for each device are needed and that, depending on the architecture, some variants might not be able to be executed, simply because the device does not exist on that node or because it failed, a stand alone program seems a suitable way to keep different variants isolated and manage their compiler flags and their individual performance data individually.
At first, the scheduler checks if any performance data exists. If no performance data exists, the scheduler compiles and executes both variants over a pipe with an input parameter that indicates for the variants to execute their test procedure.[3] The performance data is written into txt - files so that they can be reused later on. An insight of this iteration is that every variant needs its own performance data. It is suitable to save the performance data so that the test procedure, which generates performance data, has to be only executed once per device.[4] The performance data is a two - dimensional array, one containing the problem size and the other one containing the execution time needed for this problem size.[5] After finding all performance files for both variants, the scheduler reads in the performance data from the text files. Now the scheduler has to decide how to distribute the data to both variants. Since

[3] In this iteration, the scheduler exits after starting the test procedure for one variant and is restarted by the variant itself. During later iterations the scheduler sleeps and does not exit. This is more suitable because the scheduler does not repeat all its previous tasks but remains in a loop which is assumed to save time in the end since the scheduler can continue immediatly after all variants have produced their performance data.

[4] The possibility that one variant can be executed on several devices has not been considerd in this thesis.

[5] Performance data does not always only consist of the problem size and the respective execution time. While optimising code for a GPU for example, different parameters for the global and local work group size are optimal for different problem sizes which needs to be specified in the performance data as well.

both variants are identical, the scheduler is programmed in this iteration in a way that it splits up the data evenly without analysing the performance data further.[6] Both of the variants are thus executed with an even input size, they perform their algorithm and send the result back to the scheduler which outputs the final result.[7]
The first iteration has been a very simple approach to execute two programs in parallel. First insights have been gained while refinements of dealing with performance data, with the scheduler and with the variants as well as the improvement of the target quality attributes portability, programmability, and heterogeneous usage will be addressed in the following iterations.

3.2.2 Second iteration

After the previous simple approach to execute to programs in parallel, this iteration starts from scratch again to refine all components. The approach assumes that a user that did not program the variants himself wants to include them as a function call in his code.[8] The library that the user includes in order to call the function consists of groups of function calls. Each group of function calls consists of an own scheduling procedure and an own optimizing procedure which has to define the optimal data splitting point.[9] When a

[6]Finding an optimal data splitting point for two variants is trivial. For this reason, this iteration does not include this aspect. In later iterations, the data splitting algorithm becomes more sophisticated.

[7]The algorithm in this example performs a simple arithmetic operation on each element of a data vector and calculates the sizes of the vector afterwards. The algorithm is random and has been designed in a way that it contains a part where data is being processed in parallel and reduced afterwards.

[8]In our example, a scientist has existing code and wants normalise a vector within this code. The scientist only includes a library (in this iteration the library is called ppa.h) and executes the function call

[9]During this iteration, it has been considered that many algorithms can be executed in parallel but are not embarrassingly parallel. Thus, every function and its variants have an own scheduling and data splitting function. During the next iteration, this feature has been changed by introducing a standard optimizing procedure since the programming experts for certain devices might not be experts in scheduling and data splitting concerns.

user uses a function call of the framework of this iteration, the scheduler creates a shared memory segment by using the libraries sys/types.h, sys/ipc.h and sys/shm.h. Those shared segments can be accessed by individual processes which will be the variants in this case. The scheduler copies the input data of the user in the shared memory segment and executes the variants afterwards.[10] While the variants process their part of the data individually in parallel, the scheduler sleeps. After both variants are done, the scheduler outputs the result to the function that the user called.
During this iteration, the possibility to create a whole collection of functions that execute their variants in parallel has been created. Any stand alone program[11] can be implemented as a function of this framework. A function could be for instance a mathematical program that normalizes vectors or a search program. Further information about functions, later named Hybchive - routines, phases and variants can be found in the chapter "HybChive routines". While this iteration has brought additional functionality being the possibility to create a whole collection of routines that the user can choose from it has to be considered that those routines are only function calls contained in a library and not stand alone programs. This fact prevents the framework from including functions that can use heterogeneous devices. An important improvement in this iteration is that the user does not have to be an expert for programming a certain device. He just needs to include a library and to use the routine as a function call. Another additional feature of this iteration is that each routine has its own scheduler. This feature came to live because many algorithms can be executed in parallel but are not embarrassingly parallel and require more complex data splitting algorithms. On the one hand this feature makes it possible for the framework to include more algorithms,

[10] Like during the first iteration, only two equal variants for a CPU are being executed. Thus, the data is split up evenly without an additional data splitting algorithm. In further iterations this will be changed in this thesis.

[11] The framework developed in this thesis can only deal with C - programs. However, since a shared memory segment in Linux can be accessed by many programming languages, the framework can include variants that are written in other languages as well.

on the other hand it is necessary for the implementation expert to not only be an expert for the implementation but also for scheduling. In the following iteration this has been addressed by creating a default scheduler so that variants can be added without adding an individual scheduler every time. It has to also be mentioned that there is no template yet that the implementation expert can include his variant in. This feature has also been addressed by the next iteration.

3.2.3 Third iteration

During the third iteration a first prototype has been developed that includes all essential functionality to use all kinds of heterogeneous hardware in parallel for algorithms with arbitrary functionality. Since each variant that is executed with other variants in parallel is a stand alone program, the variant can use all devices that are programmable in C or C++. Devices could be for example GPUs, Xeon Phis but a variant could also address a cluster with MPI and another variant could be programmed with OpenMP. Furthermore, the tasks of the different stakeholders, the user, the implementation expert and the scheduling expert, are clearly separated. All stakeholders do not have to deal with any unrelated tasks that they do not have knowledge for which enables the possibility of a growing system of functions that use heterogeneous hardware in parallel and can be extended with additional functionality while maintaining all routines, functions and variants reusable. The architecture and its usage will be described in the following sections.The framework that has been developed has been given the name HybChive, a portmanteau of hybrid and archive. The word hybrid is included because the framework is designed to work with hybrid parallel architectures.

3.3 The purpose of HybChive

When it is mentioned that the framework HybChive can be used to use the devices of a heterogeneous architecture in parallel it has to be considered that the desire to use different parts of heterogeneous architecture in parallel can have different reasons. First of all, it is possible that one user would like to use all devices that he has access to to get a speedup. During the experimental evaluation it is shown that HybChive is able to provide a speedup for certain types of algorithms.[12] A second reason to use devices of heterogeneous architecture in parallel would likely originate from a party that is responsible to optimise the system level performance in some way[13]. It is more convenient for example if many users are able to use many devices at the same time because if all users can only take advantage of one device many users would have to wait quite some time until they can execute their routine after the previous user is done. The architecture of HybChive is designed in a way that this feature of a scheduler that schedules the routines of many users can be implemented.[14] Although it has not been implemented it is shown that this feature can be implemented in the chapter "future work". A third reason to use HybChive could be to execute different routines at the same time on different devices. An example for this scenario is the weighted expert algorithm during which different prediction algorithms are executed to predict the same event and are weighted respectively to the quality of their prediction. During the experimental evaluation, a weighted experts algorithm is implemented with HybChive and evaluated afterwards. A fourth reason to use heteroge-

[12] At the moment, HybChive only supports embarassingly parallel algorithms. However, it is possible to extend HybChive with a possible "opt-out" algorithm so that it does not use the standart data splitting algorithm but his own instead.

[13] This would be for example a cluster adminstration organisation.

[14] This however affects performance analysis, since the performance of a routine will be different depending on what else is running at the same time on a specific execution unit. The performance of a variant written for a CPU might be very different on different kinds of CPUs and depends as well on the amount of processes running on that CPU at the same time.

neous architecture is portability. A user would like to use only the devices that are available. It is desirable to have code that runs even if parts of the architecture have failed. It is also desirable that code can be used on other architecture where there is only an intersection between the new and the old devices. During the architectural overview it is explained how HybChive is designed to make this feature possible. During the experimental evaluation this feature is evaluated. A fifth reason to use heterogeneous architecture in parallel is the possibility to use all devices of the same kind with only one code whose parameters are automatically adapted according to the differences of each device. If the user has access to two different GPUs it would be useful to have one piece of code that is automatically adapted and used not only for those two GPUS but also for another arbitrary number of GPUs if they would be added to the node. During the architectural overview it is explained how this feature can be added to HybChive without extensive effort.

3.4 Fourth iteration: Refactoring and improving the usability of the code

After defining the purpose of HybChive and before defining the architecture in depth, one more iteration of refactoring and improving the usability of the code if performed. The whole code is migrated to Github and can be found here. A vision is defined as follows: "Hybchive facilitates the use of a set of routines with the same functionality but with different implementations." This vision is consciously not only focussed on heterogeneous hardware. Although the purpose of HybChive remains, as described, the same and focuses on the usage of heterogeneous hardware, HybChive offers the possibility to execute different implementations with the same purpose in parallel as well as the execution of a chosen implementation out of the set of implementations. Hence, the term "HybChive - set" is introduced and described on Github as

follows: "A HybChive set is a set of implementations with the same functionality. HybChive calls those implementations variants. A HybChive set could consist of one or more variants." Thus, HybChive enables a user to access a HybChive set in his code. He or she can decide, if the fastest or most accurate variant in the set should be used or if more than one variant of the HybChive set should be used in parallel. The API of HybChive is introduced in the architectural overview. A variant is defined as a stand alone program which is part of the HybChive set. Stand alone means that it is not a library or a function belonging to another code but software that can be compiled[15] and executed individually. Furthermore, in a variant other HybChive - sets can be accessed. The definition of a set is performed by assigning and documenting a purpose. For example, one HybChive - set has the purpose to invert a matrix. One variant of this set could be written in Python using the numpy routine. Another variant could use a BLAS/LAPACKE routine. HybChive has the purpose to measure and document the accuracy and performance of each variant so that the user is enabled witht he help of a HybChive function call in his code to use either the fastest or the most accurate one or both in parallel.

3.5 Github Integration

As mentioned in the fourth software iteration description, this software has been migrated to Github since it offers major advantages for the software development process and especially for the development of collaborative software frameworks. It can be found und the following URL:

https://github.com/janCstoffregen/HybChive

During the experimental evaluation, a Github hash is always indicated so

[15]Since variants belonging to a HybChive set can be written in different programming languages, the necessity of the compilation progress depends on the respective variant. If one variant is written in a scripting language like Python the compilation process is obviously not part of the execution process.

that the version of the software can be recreated and the experiments can be reproduced.

Issues and enhancemets that are documented in this thesis are documented on Github as well so that the reader of this thesis can look up the status of each enhancement.

Since every HybChive - set resides in an own directory and each variant in a HybChive - set resides in an own directory as well, submodules can be used on Github to include Hybchive - sets and / or variants from the repositories of other software developers.

One enhancement that can be mentioned here is the integration of continous integration, with Travis CI and / or Jenkins. Since the variants of a HybChive - set have in common that they should produce similar outputs with given identical inputs, it is likely an effective enhancement to implement those kind of testing procedures. Since Travis CI can not test variants that are created for high performance computing devices such as GPGPUs, local servers of the respective user have to be used to test those concerned variants.

3.6 Summary of used technologies

In this thesis, the client side API is programmed in C and compiled with gcc. Furthermore, the data distribution component is written in C as an own process thus it is reuseable by future client side APIs written in other computer programming languages.

Processes communicate with one another through the shared memory segments, trough the change of data in files of the framework and through bash commands. The overhead of this communication is inteded to be amortised by the speedup through the use of more computing devices in parallel.

The Variants created in this thesis are written in C and OpenCL. In general, variants and client side APIs can be written in every programming -

language that can access the content of shared memory segments. The shared memory segments enable inter process communication and are created with the C - libraries sys/types.h, sys/ipc.h and sys/shm.h.

Data is assigned statically to the different variants which process the data in the shared memory segments in parallel.

Several enhancements to include other technologies in this framework are documented on Github and within this thesis. Those technologies include Docker, OpenML and client - side APIs for Python as well as third party libraries like LAPACK and GPGPU libraries.

3.7 Code analysis in detail

3.7.1 API - Function Call

To analyse the code structure in detail, the HybChive function call is introduced. A HybChive function call is the function in the code of a user of HybChive. HybChive sets are supposed to be able to be called from all computer programming languages that are able to allocate a shared memory segment in Linux. In order to call a HybChive function, a library has to exist for the respective programming language. Currently, a library exist for the language C.[16] Every HybChive function has the type void(). Its input and output variables are defined by the parameters of the HybChive functions as explained below. In every programming language where a HybChive library implementation exist a HybChive function looks like the following:

Listing 3.1: HybChive function call

```
1 void hybchive(
2     char *set="name",                    //Name of HybChive Set
3     char *variants="all/specific",       //Chosen variant(s)
```

[16]The computer programming language C was chosen since many high performance programming frameworks like OpenCl and CUDA can be used in this language.

```
4     char *optimise="performance/accuracy",  // Optimised parameter
5     int numberOfParameters,                 //C specific input
6     ...);                                   // Set specific input
```

This function shows all aspects of a HybChive function call:

- The first parameter "name" defines the name of the HybChive set that the user wants to access. This could be for example a set with the name "searchForStringInArray" which includes a set of variants that are different implementations of a search for a string in an array.[17]
- The second parameter "variants" defines the set of variants that the user wants to execute. If this parameter is "all", all variants are executed in parallel.[18] This array also offers the possibility to only execute one or a limited set of variants if the user does not want to use all devices in parallel.
- The third parameter "optimise" gives the user the opportunity to decide if he wants the HybChive set to return the fastest or the most accurate combination of variants. The most accurate result would cause that only the most accurate variant would be executed.[19]
- The fourth parameter "numberOfParameters" is a programming lan-

[17] Another option would have been to assign a function call with different names to each routine. For example, a search routine of Hybchive could have been called "hybChiveSearch()" instead of calling hybchive(.. , char [] = "search", ..) with a string as an input. The first option has not been taken because those individual function calls would have needed to be registered in the library hybchive.h by the implementation expert. This would have caused a big and confusing file which would have been extend by many different parties. In contradiction to the first option, the second option makes it possible that the implementation expert does not have to change the library "hybchive.h" at all. Because the input value is a string, the scheduler takes this string and concatenates it with additional information to change the directory to the respective routine and executes it.

[18] At this stage of the software, HybChive would execute all variants in parallel without differentiating if they are written for the same hardware. Thus, two variants for a CPU would be executed in this case as well. An enhancement entry has been created for this feature which can be found here.

[19] The performance and accuracy measurements are performed and documented by the HybChive test procedure.

guage specific parameter which means that the HybChive function for different computer programming languages include general input parameters, which are the same independently of the programming language, and specific ones, which are only necessary if the programming languages requires it. In the case of the programming language C, this parameter is required in order to make it possible for the function to contain an arbitrary amount of input parameters after the HybChive - set independent input parameters described above. In order to make this possible, the library "stdarg.h" is used. This library needs the information though, how many input parameters are given each time the function is executed. In the programming language Python, as a comparison, this information is not needed and so the HybChive function call for this programming language would not include this paramter.

- The three dots is a C specific way to enable a variable amount of function input parameters. This part of the input parameter is passed to the variants by HybChive, it includes input parameters and output parameters to be filled by the variants with the results of the set.

As a result, the user can access a HybChive set throught the API which is the HybChive function call. To use the API, the user needs to include the library hybchive.h. The API consists of HybChive specific parts that define the set and the parameters that the user wants to access, programming language specific parts that enable the function to possess a variable amount of input parameters and HybChive - set specific parts where the user defines the input parameters for the HybChive set. [20] It has to be mentioned that this API can be easily adapted so that unchanged HybChive - sets can be called by other programming languages as well. Since the variants are stand alone programs they do not depend on the API that they are called from. The only work that has to be done to create an API for another programming language

[20] The meaning of all parameters has to be defined in the documentation. How HybChive routines are documented is defined in the according chapter "Documentation of the phases"

is to create a shared memory segment between the additional programming language and the scheduler. Since the shared memory segment is created via a library that communicates the the OS[21], it is sufficient for the additional API to create a shared memory segment and communicate the key to the scheduler so that the scheduler can access it. This would work for example with Python. As a summary, an API for HybChive is possible for every programming languages that allows IPC[22] with C.

3.7.2 Logging function and variant parameter documentation

Having in mind that the user can choose as a parameter of the HybChive function a certain parameter to be optimized by HybChive it becomes clear that HybChive is in need of an ability to measure and document the performance of the variants. Currently, HybChive is able to measure and document the time consumed for a certain size of a certain input variable. This information is stored in a file and later used to distribute data between different variants to optimise the performance of the HybChive set. The documentation of the performance of each variant is done by the test procedure which invokes the function hybchiveLog. The measured time which was consumed is stored in the file performance.txt in the directory of the respective variant. Additionally, the time consumption of certain parts of the scheduler are measured as well and stored in the file log.txt in the main directory. This file is overwritten by each HybChive function call.[23] The HybChive template includes a separate testing routine on purpose since for different HybChive variants different optimising parameters might be relevant. An example could be an OpenCL - routine which, depending on the input data to be processed,

[21] Currently HybChive can be only executed on Linux and macOS.

[22] Inter-Process Communication

[23] This is not an ideal way as the code of the user could invoke several HybChive functions which would result in overwriting every log information but for the last invocation of a HybChive function call. This issue has been documented and can be found here

is faster with a different set of parameters for the global and local worksizes. Since HybChive currently only provides a very minimalist way of documenting input parameters and resulting performance measures, a task has been created to develop a performance measurement data model that can be persistently stored. This task also includes the implementation to be able to use this data model and use it to generate an optimised data distribution between the different variants respective to the chosen input parameter by the user to be optimised. [24] Since it is an effort intensive procedure to test and find out, which combination of parameters is best to optimise a certain parameter of a HybChive function, this is one of the benefinitial features of HybChive as well.

3.7.3 The Scheduler

Once the user invokes the HybChive function in his or her code, the scheduler is executed which is the body of the HybChive function call. The input string "set" is the string that the user has defined as the HybChive - set that has to be accessed. The scheduler analyses the directory structure with the help of the given string to acquire the information which variants exist. The list of variants is used to access the directory of every variant to compile and execute the variants and varianttests. The scheduler checks if performance data exists for each variant. If it does not exist, it executes the varianttest of the respective variant.[25] After all variants possess performance data, the scheduler executes the optimizing procedure and continues after this procedure has finisehd.[26] After the optimizing procedure has ended, it communicates the data splitting pattern back to the scheduler via a shared memory segment. The scheduler creates shared memory segments for each of the given input

[24] The task can be found here

[25] The scheduler compiles and executes the varianttests and variants through a pipe with a shell command

[26] The optimizing procedure generates the data splitting pattern between the different variants.

parameters of the HybChive function.[27]
After the data spliting pattern has been generated, the scheduler can execute each variant. The input parameters for each variant are the shared memory keys for the HybChive - set parameters, their respective sizes [28] and the begin and the end of the part of the shared memory segment that each variant has to process.
After the scheduler executes the variants, it starts sleeping. After each variant is done, it increments the value in a designated shared memory segment by one. The scheduler realises that all variants have ended if the value stored in this shared memory segment is equal to the number of variants it has executed. Before the scheduler returns the result, it is absolutely necessary to remove the shared memory segments because a possibly following HybChive function will use a shared memory segment with the same key again. This is done with the following script. It removes all shared memory parts and related messages as well as related meta-data.[29]

3.7.4 Input data distribution

The task of distributing the data between the variants is a compliated subject. Since HybChive aims to be used for any kind of set and since HybChive sets can provide any kind of funcitonality, the sets can have an arbitrary amount of input and output variables as well. Thus, a general data distribu-

[27] Since the variants are stand alone programs, shared memory segments need to be created for each input parameter that the scheduler needs to communicate to the variants. This includes input and output parameters of the HybChive set. As an example, if a user would like to access a HybChive set that inverses a matrix, the scheduler needs to create a shared memory segment for the input and the output matrix. If the user would like to override the input matrix in this example, he could create another HybChive - set.

[28] Each variant access the shared memory segment in parallel.

[29] It has already been mentioned that one desired enhancement is to signalise data dependency from between several HybChive functions. In the case that two consecutive HybChive functions use the same data, the following script would have to be dynamically adapted to not delete shared memory segments that are used after the current routine. This enhancement is documented on the Github project here as well.

tion component is not possible. It can be concluded that the implementation expert has to define which data distribution routine is suitable for the respective HybChive set. This information is stored in the meta.yml file in the directory of the HybChive - set. During this thesis, one data distribution routine has been developed as well as the opportunity for HybChive sets to use data splitting algorithms provided by HybChive. As an example for a possible data splitting algorithm, the implementation is described as follows. The input data distribution component is located in the file optimize.c. It is a data distribution algorithm for a HybChive set with one input parameter.[30] It reads in the performance data of all variants and attempts to find the best way to split the data to optimize the runtime. One restriction of this optimizing algorithm is that no performance file has more than a thousand rows. A restriction of the size of the performance data assures that the stand alone program optimize.c will not take an undefined amount of time. If a user would like to speed up this program he could configure the varianttest.c programs in a way that it produces test data for less sizes of input data. A varianttest where every input size is evaluated and written to the performance data file will cause a longer execution time of the optimize procedure than a program that increments the possible input size by n=+100 for the next test. The optimize procedure reads in the performance data reads in the performance data and assumes that the last entry of the performance data includes the information of the biggest possible input size of each variant. This information is needed to not assign an input size that is too big to any device. Currently the biggest possible input size has to be defined manually. It is possible to implement a way of testing the variants by continously and automatically incrementing the input size and test if it causes a segmentation

[30]It can be observed that an optimizing algorithm can only be applied to performance data that provides a format that can be processed by the optimizing procedure. This again shows the necissity of a standardized way of documenting performance data regarging a defined parameter, for instance performance data referring to the throughtput of a HybChive set.

fault. This feature has not been implemented yet.
After reading in all performance data, the optimizing procedure generates a splitting pattern. At this point it has to be mentioned that the optimizing algorithm was not designed in an optimal way. Since designing optimal data splitting algorithms is a complex task, an easier data splitting algorithm was designed in this case. It was more important while designing the HybChive architecture to define an isolated location which can be improved later on by an expert in this field. Nevertheless, this data splitting algorithm already gives a speedup as shown in the experimental evaluation chapter.
The optimizing procedure checks if the input data is bigger than the sum of the biggest possible inputs of all devices. If the input data is too big, the program aborts the HybChive function.
Then the algorithm checks if it can split up the data evenly. If not, it is obvious that the biggest possible input size of one variant is less than the input size divided by the number of variants. In this case, the device with the lower biggest possible input size is assigned its biggest possible input size.
Afterwards, the algorithm tries to assign the remaining, unassigned data, again evenly to the remaining devices. If one of the biggest possible input sizes is again to low, the process is repeated until all input data is assigned to one device. The following pseudo code illustrates this algorithm.
The inputs are the biggest possible input size for each variant: **bpis**[**i**], the number of variants **i** and the size of the input data **n**. The output is the data splitting pattern which defines the input size of each variant i **dsp**[**i**].

Algorithm 3.1: Generate data splitting pattern after verifying that n < sum of all bpis[i].

Initialization **while** *not all data assigned* **do**
  if *n / i < bpis[i] for each i* **then**
    dsp[i] = n / i for each i
  else
    dsp[i] = bpis[i] for smallest bpis[i]
    repeat with remaining data and variants
  end
end

Since generating this pattern can be changed with respect to the needs of the user, another alternative data splitting pattern could be implemented which attempts to optimize either the execution time over all variants or the energy consumption.
Right now it has to be mentioned that no performance data is used to generate the data splitting pattern although it is already generated by the performance testing procedure. Since the generation of the data splitting point with the help of optimizing algorithms are a complex task, the task of this project was to define a way for an expert to easily define one or more data splitting algorithms in the future.
Finally, it has to be mentioned that this data splitting algorithm does not consider dynamical scheduling. It also does not consider that the biggest possible input might not be static for all variants. It might vary for instance for a CPU where the biggest possible input size depends on the number of processes running at the same time as well as the already used storage on that device.

3.7.5 Sets, variants, function invocation and performance data

A HybChive function invocation is defined as the function that is called by the user in his or her code. The body of any HybChive function is the scheduler. The scheduler access the HybChive - set that is defined by the user as one parameter of the HybChive function. Each variant of a HybChive - set consists of one or more phases. This chapter describes the structure of a HybChive - set in detail.

HybChive - sets and variants

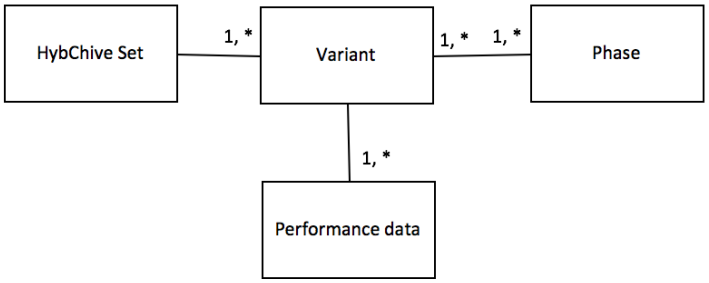

Figure 3.1: Structure of a HybChive - set

All files for one HybChive - set are stored in one directory[31]. Nothing for this HybChive - set is stored outside of this directory and nothing inside this directory belongs to another HybChive - set. The directory of this HybChive - set has to have a unique name, otherwise the scheduler might execute another routine than the user wants.[32]

Each HybChive - set consists of one or more variants. The variants provide

[31] Currently, all HybChive sets are stored in the same directory as the scheduler - code as well. An enhancement - issue has been created on Github that can be found here

[32] An enhancement is an error message created by the scheduler if the names of two HybChive - sets are identical. It can be found here.

the same functionality but differ implementation - wise. For example, a HybChive - set could be designed for a matrix multiplication. Then one variant could use an existing library, for example LAPACK, another variant could be an own implementation, for example for a GPGPU in OpenCL.
Each variant consists of one or more phases. To give the definition of a phase, imagine that you have an algorithm that you would like to include in HybChive. Either a HybChive - set exists already, then you would include you implementation as a variant in this set. Otherwise you would create a new HybChive - set for this functionality. If the implemnentation of the algorithm contains one part where data can be processed in parallel, the HybChive variant will have one phase. If the implementation of the algorithm contains two sequential parts where data can be processed in parallel and if the data has to be reduced in between those two parts[33], after the first part, then the HybChive - variant can be designed in a way that it has two phases. A phase is thus one encapsulated part of a variant that is suitable to process data in parallel and reduce it afterwards. If the implementation expert creates a second phase, the reason most probably is that the first phase has to be fully finished until the next phase can begin.[34] The next phase most likely has the output of the first phase as an input in some way. If the user executes two sequential HybChive functions in his or her code, it will be slower than one HybChive - set with two phases because the scheduler has to create another shared memory segment in the first case. Phases can consist of already implemented variants from other routines as well as newly implemented variants. Phases are likely to appear in more complex variants which contain several parts that can be computed in parallel but that have to be reduced in between.

[33] "reduced" is meant in a sense of creating a result after the parallel processing of data being necessary to continue the software, comparable to a map-reduce procedure.

[34] One example for a HybChive variant with more than one phase would be a stencil code with many numbers of iterations. A stencil code can be parallelized as well but the data is being split up, processed and retrieved more than once whereas the data to be searched in is only split up and processed once.

Examples for algorithms and their resulting phases:
1. An algorithm with one phase: An algorithm that would be implemented as a HybChive - variant with one phase would be a simplest search algorithm where the only task would be to find a string.
2. An algorithm with two phases: Imagine that a data scientist performs a machine learning algorithm and wants to train it with cross validation afterwards. Because it is more convenient, he creates one HybChive - variant with two phases because he wants to always perform those two steps after one another for his research. The reason to create a HybChive - set with more than one phase can be thus convenience as well. Since it can be re-used, this feature saves time for other scientists who want to perform the same two tasks after one another. Since a HybChive - variant with more than one phase is code that access HybChive - sets itself, one HybChive - set, being a phase of another HybChive - variant, can be called several times in a loop as well.

3.7.6 Variants

A variant consists of one or more phases. As mentioned above, phases are themselves variants of other HybChive - sets if there is more than one phase in the variant. If the variant does not have several phases, it does not access other HybChive sets and consists of code that does not include any HybChive - functions.
The variant has to have a unique name in comparison to the other variants of the respective HybChive - set. Each variant of one HybChive - set has the exact same functionality and the excact same input parameters. Precision levels, implementation, parameters and used computer languages can differ though. Many aspects resulting from those differences have to be considered, as an example, if the precision levels between two or more variants differ and if all of the variants are being executed by HybChive, the precision level of the output will be influenced by the precision levels from all variants. It has

to be assumed by the user that the output has the precision level of the variant with the lowest precision level. It is desireable that the implementation expert documents the precision level and all other aspects of the variants the he or she implements.
Every variant of one a HybChive - set is designed in a way that given a certain input, every variant produces the same output, although possibly including some small differences because of the rounding errors above. The implementations of the variants themselves are likely entirely different as well, since different implementation experts have implemented them.
At the moment, a HybCive - set needs to consist of excactly one variant per device. More implementations for one device, for example two variants for the same GPU, would cause an error.[35]
Variants consist of three files which themselves have in each variant - directory the exact same name.

The Makefile consists of all necessary flags for the respective variant.[36] Following the convention from Linux, this file is called Makefile. It compiles the variant and its test and is executed by the scheduler through a pipe. Considering that different devices need a large number of compiler flags, the implemetation expert can organize the compiler flags in an isolated way for each variant.[37] The varianttest file, which is called varianttest.* with an ending depending on the programming language that it is written in, consists of the test that produces performance data. The variant file, which is called variant.*, is the actual implementation that is being executed if performance data is already available because the test has already been executed. At this point it has to be mentioned that HybChive works currently only for

[35] An enhancement of HybChive would be that the scheduler realizes that there are two competing variants in a phase and tests them against one another. This enhancement is documented here

[36] The implementation expert has to indicate that the user needs to set for example the paths to the libraries there, for example for OpenCL or Cuda.

[37] One possible enhancement of HybChive is to organize compiler flags centralized for each parallel computing programming language. This enhancement is documented here

computer languages that are compiled and then executed as a binary. A possible enhancement would be to include a shell script in the directory of a variant so that the scheduler does not execute the binary but the shell script so that the scheduler does not have to know the ending of the code file that is executed instead of the binary.[38]
During the experimental evaluation chapter, the files of phases and variants are shown in detail and how they are implemented to create new phases and variants.

3.7.7 Documentation of the phases

When an implementation expert creates a new HybChive - set, he has to document the functionality of this set since he has to expect that other implementation experts will add variants to this new set as well. The documentation for each HybChive set can be very individual, it should describe the input and the output of the phase but additional information is optional and can be added by the implementation expert if desired.[39] There are for example different prediction algorithms that can produce a similar output. It is up to the implementation expert if an additional variant has to be a defined algorithm or can be a different algorithm that produces the same output in a different way. It is also optional for the implementation expert to define a desired precision level that each variant that is or will be part of this phase has to provide. In any case, the implementation expert has to define which type the input and output variables have to have.

[38] This enhancement is documented here

[39] One enhancement is the possibility that the implementation expert provides input and output data that can be used for automatic testing for additional variants.

3.8 The memory allocation of the input data

HybChive can execute different implementations of the same functionality at the same time. For example, if a user calls a HybChive - set on a node with two GPUs, HybChive would split up the data and assign it to one implementation for the first GPU, a second implementation for the second GPU and a third implementation for the CPU if all variant are included in the respective HybChive - set. To split up and to assign the data to variants that might be written in different programming languages, the data needs to be stored in a shared memory segment. For this purpose, three scenarios have to be considered:

1. **The user is able to originally allocate the memory:** It is possible, that the user has written the entirety of the code himself. In this case, he or she can allocate the input variables for a HybChive routine initially as a shared memory variable. HybChive uses the C libraries <sys/types.h>, <sys/ipc.h> and <sys/shm.h> to generate the shared memory segment.[40] If the user creates this shared memory segment in his code already, HybChive does not need to copy the input data into a shared memory segment which saves a lot of time. For example, if the user reads in the data from a csv into a variable, this variable can already be allocated as a shared memory segment by the user.[41]
2. **The user is not allowed to allocate the data:** This scenario can happen if the user is only programming a part of the whole software and the input data comes from a part of the software that he can't influence. In this case, HybChive automatically creates shared memory segments for each of the HybChive - set specific arguments.

[40] Currently, there is an API for the programming language C. Thus, a user can address all HybChive - sets from code written in C. It is possible however to create APIs with the same functionality for all programming languages that can access a shared memory segment.

[41] A possible enhancement is to signalize to HybChive that the memory is already allocated in a shared memory. This enhancement is documented here

Since copying the data into a shared memory segment is time intensive, a desired feature to be implemented in HybChive would be to decide if copying the data and using all devices would be more time intensive than not copying the data and using the fastest device available.[42]
If the user wants to achieve a speedup, using HybChive makes sense if the amount of time that the user saves by using more than one device at the same time with the help of HybChive is more than it takes to copy the input variables of the HybChive function into the shared memory segment.

3. **The user can't use the shared memory features because of hardware or operating system restrictions:** Since HybChive has been developed in a Linux environment for now, it might not be possible to use the named libraries on other operating systems. Furthermore, some memory protection mechanisms could prevent the use of shared memory. In this case, the fastest implementation should be executed without HybChive.

In the course of this thesis, HybChive assumes that the user has not allocated the input data as a shared memory segment. Since, in one of the experiments, 3 devices are used in parallel in this thesis and since a speedup is observed even in this slowest case[43] of memory allocation in which HybChive automatically copies the whole amount of data in a shared memory segment, it can be assumed that the speedup compared to the execution of a program on a single devide is even bigger if the data does not have to be copied into the shared memory.

[42] This feature is documented here

[43] It is the slowest case because HybChive has to copy all input data into a shared memory segment which is time intensive. If the user would allocate the shared memory segment himself and uses this segment for the input data already, HybChive would not have to copy it so it would be faster than the case that was implemented in this thesis.

Chapter 4

Experimental Evaluation

The experiments in this thesis aim at respectively exploring one or more aspects of the code. Since, between the experiments, the code might change, every experiment will document the hash code of the git version used for the experiment, so that it becomes reproducible. The output code snippet is reduced to the significant experiment purpose by taking away some debugging - ouput information.

4.1 Evaluating Functionality

4.1.1 Hello HybChive

Github Hash: ea21655 | **Specs:** macOS Sierra, 2.9 GHz Intel Core i5, 16 GB 2133 MHz LPDDR3 | **Goal of this experiment:** Explore the basic functionality of the code.
Execution of the code: The user invokes the following hybchive function:

Listing 4.1: HybChive function call of the first experiment

```
1    char set[]="hybChiveC_SetTemplate";
2    char variants[]="all";
```

```
3     char optimize[]="performance";
4     int numberOfParameters = 1;
5
6     hybchiveLog( "user | execute HybChive Function" );
7
8     hybchive(
9         set,
10        variants,
11        optimize,
12        numberOfParameters,
13        sizeof( A ),
14        "double",
15        A
16    );
```

Explanation of code snippet:

Line 1: The hybchive - set that the user wants to access is defined.

Line 2: The user would like to execute all variants of this HybChive - set in parallel. Within this thesis, the possability to call one variant instead has been created, by changing this argument to the name of the variant. An enhancement to give an array of variants as the input has been created here.

Line 3: The optimize procedure is chosen. HybChive aims at an architecture where optimising algorithms are designed in a way that they are tightly coupled to the HybChive - set since each set has specific parameters that can be altered for optimal execution behaviour. For simplification purposes, all experiments in this thesis use the same optimising procedure.

Line 4: The number of parameters is the number of HybChive specific parameters following after this input. In this case it will be one HybChive specific parameter, begin a 4 * 4 matrix being called A with all entries as zeros in this experiment.

Line 13 - 15: Each HybChive - set specific argument has to indicate the sizeof the input, the type of the input and the pointer to the input itself. The size of the input is needed since the HybChive has to know the exact size

for the shared memory segment that it creates for each parameter. Similarly, HybChive needs the type of the variable for the creation of the shared memory size. The pointer is needed for the copying process that copies the content of the HybChive - set specific argument into the shared memory segment.

Listing 4.2: Output of the first experiment

```
1 1526197333.578247, user | execute HybChive Function
2
3 1526197333.578326, scheduler | start scheduler
4
5 1526197333.578988, scheduler | 4.4 All performance files found.
      Execute performance optimizing procedure
6
7 createSharedMemorySegmentandKeys | Size: 128, Type: double
8 createSharedMemorySegmentandKeys | Randomly generated key: 1253895963
9 createSharedMemorySegmentandKeys | Create shared memory segment for a
      double hybchive specific argument
10 createSharedMemorySegmentandKeys | Copy input to shared memory segment
11
12 attachSharedMemorySegment | ( variant 1234 ) | for key: 1253895963 and
       size: 128
13
14 1526197333.682565, optimize | 6. Reading performance files
15
16  6.3 Largest possible input for variant 0: n= 1000
17
18  6.3.1 Size of input data: n*n = 16
19
20  6.3 Largest possible input for variant 1: n= 1000
21
22  6.3.1 Size of input data: n*n = 16
23
24  6.3 Largest possible input for variant 2: n= 1000
25
26  6.3.1 Size of input data: n*n = 16
```

```
27
28 1526197333.699881, optimize | 7. Run Optimizing Algorithm
29
30 optimize | 7.0.1 Largest possible input for given problem and sum of
       devices: 3000000
31 optimize | 7.0.3 Check, if data can be split evenly
32 optimize | 7.0.5 Data evenly splitable
33 1526197333.699935, optimize | 7.1.2 End Optimizing Routine
34
35 optimise | 7.2 Create shared memory space for input.
36 optimise | 7.3 Put splittable in shared memory space
37 1526197333.705829, scheduler | 7.4 Get the splitting information
38 1526197333.706001, scheduler | 7.5 Splitting Data arrived in scheduler
39
40 scheduler | 7.6 Device: 0, n = 5
41 scheduler | 7.6 Device: 1, n = 5
42 scheduler | 7.6 Device: 2, n = 6
43
44 createSharedMemorySegmentandKeys | Size: 8, Type: double
45 createSharedMemorySegmentandKeys | Randomly generated key: 972422130
46 createSharedMemorySegmentandKeys | Create shared memory segment for a
       double hybchive specific argument
47 createSharedMemorySegmentandKeys | Copy input to shared memory segment
48
49 Size of shm: 8
50 Size of input: 8
51 createSharedMemorySegmentandKeys | shm[ 0 ]: 0.000000
52
53 attachSharedMemorySegment | ( variant 1234 ) | for key: 972422130 and
       size: 8
54 scheduler | sharedCommunication[ 0 ]: 0.000000
55  scheduler | sharedCommunicationKey: 9
56
57 scheduler | 8.6 Converting inputs for programs into strings
58 ------
59 cd hybChiveC_SetTemplate/variant1 && make final && ./variant 0 1
       972422130 3 1253895963 128 5
```

```
60 ------
61
62 scheduler | 8.6 Converting inputs for programs into strings
63 ------
64 cd hybChiveC_SetTemplate/variant2 && make final && ./variant 1 1
       972422130 3 1253895963 128 5
65 ------
66 scheduler | use concatenate to prepare shell command
67
68 scheduler | 8.6 Converting inputs for programs into strings
69 ------
70 cd hybChiveC_SetTemplate/variant3 && make final && ./variant 2 1
       972422130 3 1253895963 128 6
71 ------
72 scheduler | use concatenate to prepare shell command
73
74 scheduler | 16. start sleeping
75 gcc variant.c -o variant ../../attachSharedMemorySegment.o
76 gcc variant.c -o variant ../../attachSharedMemorySegment.o
77 gcc variant.c -o variant ../../attachSharedMemorySegment.o
78
79
80 variant 0 | Hello, I have 1 input(s):
81 variant 2 | Hello, I have 1 input(s):
82 variant 0 | I will show you the first input, in a set you design you
       should know how many inputs you need.
83 variant 2 | I will show you the first input, in a set you design you
       should know how many inputs you need.
84
85
86 attachSharedMemorySegment | ( variant 2 ) | for key: 1253895963 and
       size: 128
87 variant 2 | to finalize: printf part of the input!
88 attachSharedMemorySegment | ( variant 0 ) | for key: 1253895963 and
       size: 128
89 variant 0 | to finalize: printf part of the input!
90 variant 0 | A[ 0 ]: 4.000000
```

```
91 variant 2 | A[ 0 ]: 4.000000
92 variant 0 | sharedMemoryCommuncationKey: 972422130
93 variant 2 | sharedMemoryCommuncationKey: 972422130
94
95
96 attachSharedMemorySegment | ( variant 0 ) | for key: 972422130 and
       size: 8
97 attachSharedMemorySegment | ( variant 2 ) | for key: 972422130 and
       size: 8
98 variant 0 | dataSlice 5
99 variant 2 | dataSlice 6
100
101
102 variant 0 | problem size 16
103 variant 2 | problem size 16
104
105
106 variant 0 | number of variants: 3
107 variant 2 | number of variants: 3
108
109
110 variant 0 | begin: 0, end: 5
111 variant 2 | begin: 10, end: 16
112 ,
113 ,
114 variant 0 | sharedMemoryCommuncationContent before finish: 1.000000
115 variant 2 | sharedMemoryCommuncationContent before finish: 2.000000
116
117 variant 1 | Hello, I have 1 input(s):
118 variant 1 | I will show you the first input, in a set you design you
       should know how many inputs you need.
119
120 attachSharedMemorySegment | ( variant 1 ) | for key: 1253895963 and
       size: 128
121 variant 1 | to finalize: printf part of the input!
122 variant 1 | A[ 0 ]: 0.000000
123 variant 1 | sharedMemoryCommuncationKey: 972422130
```

```
124
125 attachSharedMemorySegment | ( variant 1 ) | for key: 972422130 and
        size: 8
126 variant 1 | dataSlice 5
127
128 variant 1 | problem size 16
129
130 variant 1 | number of variants: 3
131
132 variant 1 | begin: 5, end: 10
133 ,
134 variant 1 | sharedMemoryCommuncationContent before finish: 3.000000
135
136 scheduler | 17. copy shared memory segment back to input variable
137
138 user | matrix after hybchive function call
139 0.000000 0.000000 0.000000 0.000000
140 0.000000 1.000000 1.000000 1.000000
141 1.000000 1.000000 2.000000 2.000000
142 2.000000 2.000000 2.000000 2.000000
```

Explanation of the output:

Line 1: This output is generated the the function hybChivelog() which generates a time information as well as an entry in the hyChive.log file. This file is located in the hybChive - directory in a central way so that, if HybChive is extended and more than one user uses HybChive on the same nodes at the same time, the log will be able to reflect that. Each log entry should be written in a way that it starts with the location that this entry is located in, in the form that it can be observed in this case as well: <location of entry> | log entry. Since HybChive consists of many different parts, it is hard to locate the entry otherwise.

Line 5: This entry is generated after all performance files have been found. The scheduler checks, if every variant in the set owns a performance file with the name given as the performance procedure at the HybChive - function

invocation ("performance" in this case.) Thus, every variant can have many performance files in the form <nameOfThePerformanceProcedure>.txt. The output of the generation of the performance files will be documented in one of the following experiments.

Line 7 - 10: HybChive creates a shared memory segment and a belonging (int)key for every HybChive - specific argument and copies it to this segment.

Line 12: After the shared memory segment has been created, the scheduler attached this memory segment to its scope since every data processing of the variants on the shared memory segment has to be copied back to the input variable after the variants have finished their tasks.

Line 14: The optimising procedure is started and reads in the performance files of each variant.

Line 16 - 30: The optimising procedure assumes that the biggest possible input for one device is pointed out in the last entry of the performance.txt file. In this case, each variant, being written for the same device (a CPU in this case) points out the problem size n = 100 in the performance.txt file. This information comes from the test itself where the file is written and where the biggest possible input for the respective devices is defined. The performance procedure needs this information since it has to assign slices of data to each device and those slices should not be too big for the respective device. Since we have a very small problem size in this example, the size of the input data is not bigger than each possible input size of each variant so no further tests are performed.[1]

Line 31 - 33: The actual optimisation algorithm is performed. In this example, the optimisation algorithm just splits up evenly the input data between the devices for simplification purposes. Althoug it reads in all performance data, it does not use it to optimise the runtime. Since this is a task for an implementation expert, this thesis aims at creating the design that

[1] If the problem size is bigger than the sum of biggest possible input sizes of each variant, the user is informed via output and asked to abort his code.

enables those expert to work individually from one another which is possible for the performance routine since it lives as a stand - alone software in its own file.

Line 35 - 36: The optimise code allocates a shared memory segment to communicate the data splitting pattern back to the scheduler. After the optimise code finishes, the scheduler continues its following tasks.

Line 37 - 42: The data splitting pattern has been captured by the scheduler. In the case of the HybChive set in this experiment with the problem size 4 * 4, the device 0 gets the first 5 entries to process, the second device gets the second 5 entries and the third device gets the last 6 entries, so that everything adds up to the problem size and all entries of the matrix can be processed. The variants themselves figure out the slices that they have to process with the help of their id and the information displayed in the current lines. No calculating referring to data splitting pattern should be every performed in the scheduler since it is a HybChive - set specific problem.

Line 44 - 55: The scheduler creates a shared memory segment for the communication between the variants and the scheduler. After each variant has finished, it increments the first entry of this segment by one, so that the scheduler finalizes the HybChive function call when this number equals the number of variants that have been executed. [2].

Line 57 - 71: The scheduler prepares the shell command for each variant. It is able to compile each variant with possibly different commands and flags since this is respectively documented in the Makefile for each variant under the command "make final". Since each resulting binary or script has to be called "variant"[3] The input of every variant is the variantID, which is assigned

[2]This aspect does not consider the case that one variant fails during its execution. No process is implemented to deal with this szenario yet. An enhancement is documented here

[3]New HybChive - sets are created by copying and pasting the HybChive - set template under a name that reflects the purpose of the new HybChive set. The HybChive - set will include a template for a first variant which will include the necessary directory and file structure.

to each variant individually by the scheduler, then the number of parameters[4] which defines the number of HybChive specific input arguments. Afterwards the key for the shared memory segment that is used for the communication with the scheduler is given as an input argument, followed by the number of variants that were executed by the scheduler. This information is necessary for the variant to calculate its data slice in this example, which is given as the last parameter. The fifth and the sixth parameter is the shared memory key and its size for the input data that should be processed. It has to be mentioned that the scheduler dynamically creates those shell command lines and works already with HybChive - sets that have more than one input argument as well. A limitation that has to be mentioned is that HybChive only works with doubles at the moment, the enhancement is documented here and described in this thesis further to demonstrate that it is easily done.

Line 74: After the scheduler executes the shell commands through a pipe and starts the variants it starts sleeping until all variants have finished their processes.

Line 80 - 83: Those outputs mark the beginning of all variants. All variants only differ by their name in this experiment and thus their output only differs through the variandID being displayed in all of their outputs. They are designed to serve as a template for further variants and thus have a functionality that is focussed on demonstrating their functionality that can be further extended. One can already observe that the variants are executed simultaneously through the output since the second variant with the ID 1 is a little bit delayed in this example.

Line 96 - 111: Each variant allocates and attaches the shared memory segments created by the scheduler with the help of the HybChive - variant specific arguments which contain the memory key and the size of the respective segment.

[4]This parameter is only necessary for variants written in C with a variable amount of input arguments and is not generally obligatory for all HybChive - variants.

Line 134: Each variant communicates to the scheduler that it has finished by incrementing the shared memory segment for communication by 1. If the entry [0] of this segment equals the number of variants that have been executed, it continues its tasks.

Line 136: After all variants are done, the scheduler copies the shared memory segment for each HybChive - set specific argument back to its original buffer which was allocated by the user. This is necessary since the shared memory segment has not been allocated by the user and belongs to the scope of the scheduler.

Line 138 - 142: The input matrix is displayed in the code of the user after the HybChive - set has processed it. It marks, which entry has been processed by which variant. The entries being 0 have been processed by the variant 0 and so forth.

4.1.2 Generation of performance files

Github Hash: d6b77d8 | **Specs:** macOS Sierra, 2.9 GHz Intel Core i5, 16 GB 2133 MHz LPDDR3 | **Goal of this experiment:** Explore the performance file generation.

HybChive aims at storing the performance files in which parameters in the code are mapped to output measures. As an example, the problem size can be mapped to the time that the HybChive - function invocation needs to finish. The performance files are generated by the optimise.c code in this example. The HybChive method invocation is the same as above. The difference to this experiment is that the performance files of one of the three variants has been deleted.[5] This experiment displays how performance files are generated by the test of this variant. The output of the varianttest.c code, which is the test of the variant in this example, is shown below.

[5]Normally, the user should generate the performance files him - or herself on the respective device. The performance files had already been generated in the last experiment to focus on this aspect in this experiment

Listing 4.3: Output of the second experiment

```
1 varianttest | Executing  test procedure
2
3 varianttest | Biggest possible input for this device defined here:
      1000
4
5 varianttest | increment after each test by: 100
6
7  Test with n = 0: time: 0.000171
8
9  Test with n = 100: time: 1.557680
10
11  Test with n = 200: time: 1.129758
12
13  Test with n = 300: time: 2.677607
14
15  Test with n = 400: time: 2.242029
16
17  Test with n = 500: time: 3.820866
18
19  Test with n = 600: time: 3.338601
20
21  Test with n = 700: time: 4.875937
22
23  Test with n = 800: time: 5.479547
24
25  Test with n = 900: time: 5.029659
26
27  Test with n = 1000: time: 5.500197
```

Explanation of the output:

Line 3: The implementation expert, who is in charge of defining the test for this variant, defined the biggest possible input for the device that the variant is written for as n = 1000.[6].

[6] An enhancement for HybChive is that biggest possible input sizes are found automat-

Line 5: The implementation expert defines that the varianttest should be performed for every n += 100 problem sizes. Here a tradeoff can be observed. The bigger the increment is, the faster the test will be performed. The smaller the increment is, the better the optimisation procedure can assign slices to the according variants.

Line 7 - 27: The test is performed biggest possible input / increment times and the time needed for each test is stored in the performance.txt file which later will be read in by the optimisation procedure to optimise the assignment of the data slices to the respective variants.

One has to mention, that at the moment the variant - tests for each variant in a HybChive - set are executed sequentially. Thus, an enhancement would be to execute them in parallel, which is documented here. An enhancement is also to precisely document which device the performance data is generated for so that it can be resued by other users as well. This enhancement is documented here. Finally, the possibility to design multiple performance optimisation procedures per HybChive - set has to be implemented to ensure that the user can choose the optimisation procedure that suits his or her goal. This enhancement is documented here. The test of the variant and the variant itself are split up on purpose to cleanly separate the purposes of both codes.

4.1.3 Creating a new HybChive - set

Github Hash: 05a49c8 | **Specs:** macOS Sierra, 2.9 GHz Intel Core i5, 16 GB 2133 MHz LPDDR3 | **Goal of this experiment:** Explore the possibility to create a new HybChive set.

HybChive provides an easy way to add a new HybChive - set. An implementation should add a new HybChive - set if no set with the same functionality and the same inputs exists already. HybChive - templates are designed in a way that they can be copied and pasted in the same directory under a differ-

ically. The enhancement is documented here

ent name to start a new HybChive - set. In this example the new HybChive - set is generated in the following way:

Listing 4.4: Create a new HybChive - set by copying the existint template

```
cp -R hybChiveC_SetTemplate/ newSetForSecondExperiment
```

The new HybChive - set resides now in its own directory with the name "newSetForSecondExperiment" in the same directory as the former set that we explored. To demonstrate, that the user can already call this set now and thus that the set is ready to be adapted by the implementation expert, we call this set with the adapted HybChive - function call and see that the output is identical to the first experiment.

Listing 4.5: HybChive function call of the third experiment

```
    char set[]="newSetForSecondExperiment";
    char variants[]="all";
    char optimize[]="performance";
    int numberOfParameters = 1;

    hybchiveLog( "user | execute HybChive Function" );

    hybchive(
        set,
        variants,
        optimize,
        numberOfParameters,
        sizeof( A ),
        "double",
        A
    );
```

The output matrix of this experiment is the following:

Listing 4.6: Output after the HybChive function call of the third experiment. The variant with the id number zero has processed the first row and part of the second row whereas the variant with the id 1 has processed part of the second line and part of the third line since each variant writes its id in the matrix as defined in the implementation of the variant itself for demonstration purposes

```
1 user | matrix after hybchive function call
2 0.000000 0.000000 0.000000 0.000000
3 0.000000 1.000000 1.000000 1.000000
4 1.000000 1.000000 2.000000 2.000000
5 2.000000 2.000000 2.000000 2.000000
```

It can be concluded that the newly generated HybChive - set works as expected and that new HybChive - sets can be generated by only copying and pasting an existing HybChive - set under a new name.

4.1.4 Creating a new variant in a HybChive - set

Github Hash: 49082b6 | **Specs:** macOS Sierra, 2.9 GHz Intel Core i5, 16 GB 2133 MHz LPDDR3 | **Goal of this experiment:** Explore the possibility to create a new HybChive variant.
Each variant resides in the directory of the respective HybChive - set in its own directory. A new variant can be created by copying and pasting an existing variant under a new name. This variant can be then adapted for the respective purpose. For example, a variant for a CPU could be copied and pasted under a new name and then be adapted for a variant for a GPGPU by adding all necessary flags and libraries to the Makefile and the varianttest and the variant. If we look at the example from the last experiment, we have 3 variants. During this experiment, we copy variant3 in the HybChive - set "hybChiveC_SetTemplate", copy it and paste it under the name "variant4". Now, the output looks as follows:

Listing 4.7: Output after the HybChive function call of the fourth experiment with four variants

```
1 user | matrix after hybchive function call
2 0.000000 0.000000 0.000000 0.000000
3 1.000000 1.000000 1.000000 1.000000
4 2.000000 2.000000 2.000000 2.000000
5 3.000000 3.000000 3.000000 3.000000
```

One can see that the fourth variant has been successfully integrated and processes its data slice as shown in the output snippet. Variant4 has processed the 5th row of the matrix.

After creating a fifth variant, the output looks as follows:

Listing 4.8: Output after the HybChive function call of the fourth experiment with five variants

```
1 user | matrix after hybchive function call
2 0.000000 0.000000 0.000000 1.000000
3 1.000000 1.000000 2.000000 2.000000
4 2.000000 3.000000 3.000000 3.000000
5 4.000000 4.000000 4.000000 4.000000
```

One can see that the input matrix is distributed to the variants as even as possible, as defined in the optimising procedure.

4.2 Evaluating performance

4.2.1 One Variant and a growing problem size

Github Hash: 37c9172 | **Specs:** macOS Sierra, 2.9 GHz Intel Core i5, 16 GB 2133 MHz LPDDR3 | **Goal of this experiment:** Explore the bottlenecks of HybChive that become visible when the problem size increases. During this experiment, the HybChive - set from the first experiment is taken,

with the name HybChiveC_SetTemplate, with one variant. Timemarks are taken at the following part of the procedure:

1. User invokes the HybChive - method. A HybChive log - point has been set immediately before the user starts the HybChive - method. The log - point can be found here.
2. Reading performance files. The optimising procedure reads in the respective performance files to perform the optimising procedure. The log - point can be found here. It has been chosen for this experiment since it marks the beginning of the optimising procedure.[7]
3. Start / End of copying HybChive - specific arguments into shared memory segments. This log - point can be found here.
4. Start of variants / End of all variants. Those log - points have been set to mark the beginning and the end of all variants. They can be found here.
5. Scheduler copies input shared memory segment back to input variable. This log point has been set here as well.
6. End of HybChive - method invocation. This log - point has been set right after the HybChive - method invocation in the code of the user.

Methodology: A sample HybChive - log snippet looks as follows:

Listing 4.9: Sample output of a HybChive - log - file

```
1 1527248339.254845, optimize | 6. Reading performance files
2 1527248339.267616, optimize | 7. Run Optimizing Algorithm
3 1527248339.267647, optimize | 7.1.2 End Optimizing Routine
4 1527248339.166239, user | execute HybChive Function
5 1527248339.166306, scheduler | start scheduler
6 1527248339.176394, scheduler | 4.4 All performance files found.
      Execute performance optimizing procedure
```

[7] It has to be metioned that the performance data exists already in this example. Thus, the performance of the varianttests has not been measured.

```
 7 1527248339.176420, scheduler | 4.5 Start copying HybChive - set
       specific parameters to shared memory segment
 8 1527248339.176535, scheduler | 4.5.1 End copying HybChive - set
       specific parameters to shared memory segment
 9 1527248339.273727, scheduler | 7.4 Get the splitting information
10 1527248339.273785, scheduler | 7.5 Splitting Data arrived in scheduler
11 1527248339.275780, scheduler | 16.01 All variants started
12 1527248340.279071, scheduler | 16.01 All variants ended
13 1527248340.279254, scheduler | 17. copy shared memory segment back to
       input variable
14 1527248340.280250, user | end HybChive Function - order timemarks
       yourself
```

The first part of this log is a time stamp, generated by the library time.h.[8] The second part of a log entry is the location of where this entry was generated. The last part is an eplanation which part of HybChive follows after the respective log entry. During this experiment, the differences between the different parts have been generated. The amount of time needed for different problem sizes are shown in the respective graph of this experiment.

One can observe in figure 4.1 that most of the time is consumed by the execution of the variants. This part includes the compilation and the execution of each variant.[9]

Another observation is that reading in the performance files by the optimising procedure takes an amount of time visibly larger on this figure than the other parts. As discussed this is due to number of measurements taken during the varianttest - procedure.[10]

[8]The number is the amount of seconds that the current time is past 1970. One enhancement is to generate the Year, date and time out of it so it becomes a standard log - file. This enhanvement is documented here.

[9]Each variant is compiled during a HybChive - method invocation since some parameters from the optimising procedure might only be usable if the variants are recompiled during runtime. One enhancement is to only compile the respective variant if this variant is executed on a new environment. This enhancement is documented here.

[10]The varianttests have not been included in the measurements of this experiment since it is obvious that this part of the software needs to be only executed once before the performance data is reusable.

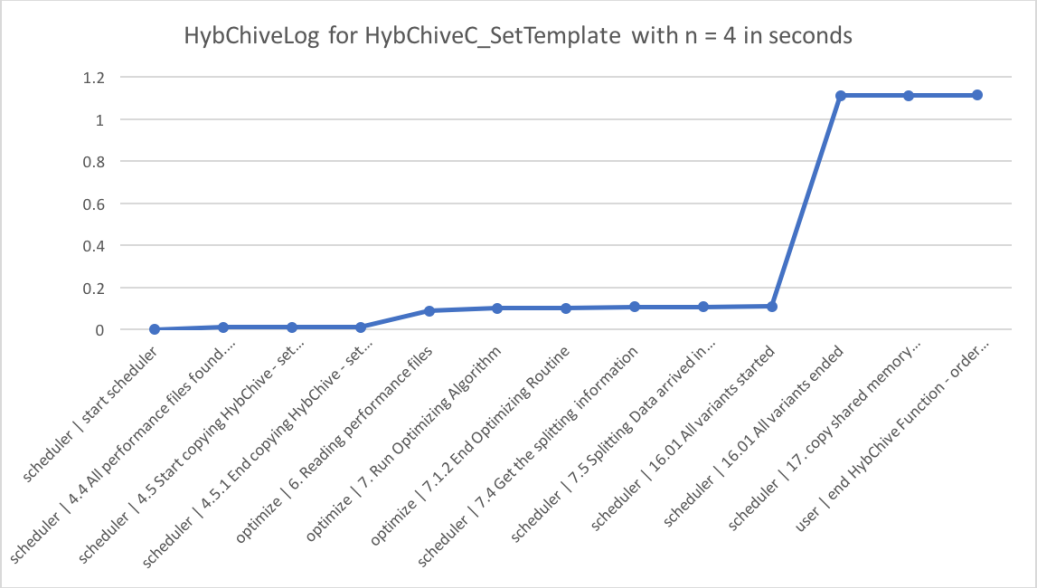

Figure 4.1: Time marks of a HybChiveC_Settemplate method invocation with the problem size n = 4 in seconds

The following figure illustrates the relation between the same HybChive - set method invocations for the problem sizes n = 4 and n = 700:

Figure 4.2 compares the timemarks created by the HybChive - log from the HybChive - set described above for the problem sizes n = 4 and n = 700. The graph aims at studying how the time measurements change with a growing problem size. An information missing in this figure that however has to be mentioned is that the execution time of the HybChive - method invocation with the problem size n = 700 was 2 percent higher. It can be observed through this information that the growing problem size does not dramatically increase the native HybChive - code without the actual variants.

One can observe that the process of reading the perfomance files in the optimising procedure takes the highest amount of time. Since this procedure has nothing to do with the problem size one has to search for another explanation. Since the process of copying the HybChive - set specific parameters

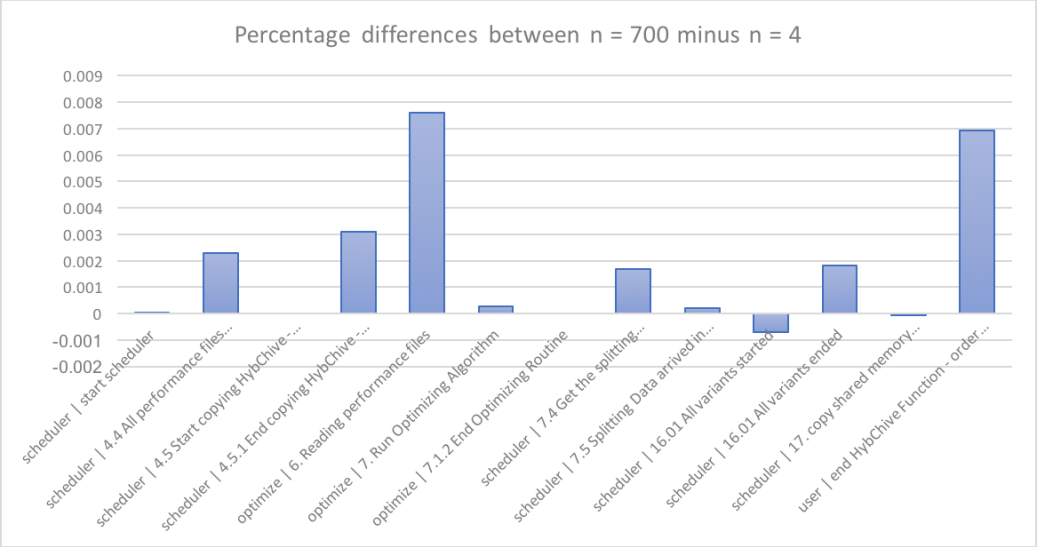

Figure 4.2: Comparison of the timemarks created by the HybChive - log from the HybChive - set described above for the problem sizes n = 4 and n = 700.

into the shared memory segment is likely the part of the code that takes more time with a growing problem size and since this happens directly before this part of the code, it can be assumed that this procedure influcences the following time measurement as well in some way.[11]

Another observation is the increase of the last part of the code, which signalises that the HybChive - method invocation has returned its outputs to the code of the user. This part of the code includes the process of copying the content of the shared memory segment back to the input parameter of the user so this variable can be further used in the code and is this explainable.

This experiment shows that HybChive is able to perform the shared memory creation and optimising procedures in a likely effective way without increasing its procedures much with a growing problem size of the HybChive -

[11] After copying the large amount of data in the shared memory segment, the processor of the machine might assign more ressource to other processes running at the same time for example.

set.

4.2.2 Varying amount of variants and large problem size

Github Hash: 5b4fdb7 | **Specs:** macOS Sierra, 2.9 GHz Intel Core i5, 16 GB 2133 MHz LPDDR3 | **Goal of this experiment:** Investigate the behaviour of HybChive with a growing number of variants.

This experiment aims at evaluating if the overhead of a HyChive - set gets larger if the number of variants grows. The HybChive - set HybChiveC_SetTemplate was taken and the runtime was measured with a problem size (double) n = 700 of this set. All variants are being executed by HybChive on the same CPU which results in the fact that no speedup can be expected with a growing number of variants.[12] Furthermore, the variants are identical in their implementation. With the help of the HybChive - log, different time markers were set and their measurements evaluated. Each different szenario (e.g. a HybChive set with a fixed number of variants) was measured four times after which the average of the HybChive - log entries were evaluated. The following graph illustrates the measurements taken for the given HybChive - set with different amounts of variants:

Figure 4.3 includes two graphs. The upper blue graph illustrates the time that HybChive needs from the start of the method invocation until HybChive - method returns a value. The lower orange graph illustrates the time that is needed from the beginning of the execution of the variants until all variants have finished their invocation. One can observe that the overhead that HybChive needs for the execution of the set of variants does not significantly increase with a growing number of variants since the overhead for the HybChive set with four variants is even slightly less than the the overhead of the HybChive - set with two variants. With a growing number of variants,

[12] The next experiment aims at investigating the possibility of HybChive to generate a speedup using two GPGPUs in addition to a CPU.

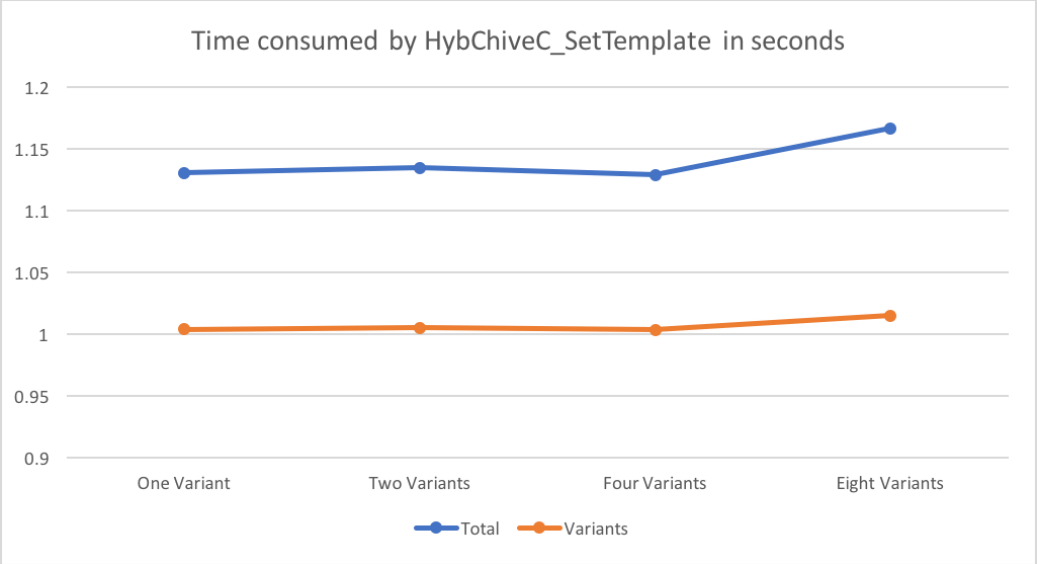

Figure 4.3: HybChive - log measured runtimes for the HybChive_CSetTemplate with a varying amount of variants

HybChive has to read in more performance data for the optimising procedure which is the only slightly time intensive, as shown in the last experiment and this experiment illustrates that this circumstance does not significantly increase the overhead of the HybChive - method invocation. Since HybChive is able to manage the overhead for a growing number of variants in an effective way, one can conclude after studying this experiment that a speedup can be achieved by adding a variant to the HybChive - set that is faster than the already existing variants in the HybChive - set.

4.2.3 HybChive Case study with one GPGPU and one CPU

Github Hash: 611db4f | **Specs:** Device 1: Red Hat, x86_64, 4 cores per socket, 24020 MB Ram, Intel Xeon CPU, 2,67 GHz | Device 2: GPGPU Tesla C2050, OpenCL 1.1 CUDA, 304.54, Max Compute Units: 14, Max

Clock Frequency: 1147, Max Work Item Dimensions: 4198213 **Goal of this experiment:** Demonstrate that the usage of a HybChive - set can result in a speedup compared to an implementation without HybChive.

Functionality of the HybChive - set to be investigated

The algorithm that is implemented in this experiment has to be designed in a certain way to serve this experiment. First of all, it has to be easily understandable so that it can be easily demonstrated in which way a speedup is achieved. In addition, the results of this experiment have to be easily transferable to real - world applications so that it can be easily decided if the results of this experiment can be applied to the respective other real world application.

Due to this requirements, each entry of a matrix A [n * n] is incremented by 1 i times.

The results of this experiment can be thus applied to embarrassingly parallel algorithms with a large amount of floating point operations per entry. Examples for algorithms are

- An algorithm that checks for each entry of the matrix if it is a prime number
- Blocked algorithms where the blocks can be processed in parallel, for example a matrix matrix multiplication.
- Stencil code[13]
- Fourier transformations
- Compression algorithms

Since this thesis aims at creating an architecture for parallel computing of

[13]Since all variants of a HybChive - set have access to the shared memory segments of all input arguments, also more complicated data splitting patterns can be implemented for those kind of algorihms.

heterogeneous parallel computing rather than creating optimised implementations of specific algorithms named above, the described simple algorithm has been chosen to demonstrate which kind of speedup can be achieved by HybChive.

From a native user implementation to the working HybChive set

Before HybChive is used, the code of the user in our experiment looks as follows:[14]

Listing 4.10: Naive user implementation

```
1 for ( i = 0; i < n * n; i++ )
2 {
3   for( j = 0; j < numberOfFloatingPointOperationsToBePerformed; j++) {
4     A[ i ] += 1;
5   }
6 }
```

To create a HybChive - set with this functionality, this very same part of the code is included in the template of the HybChiveC-settemplate. In addition to this first variant for a CPU, a variant for a GPGPU is added. The kernel looks as follows:

Listing 4.11: GPGPU kernel of the second variant

```
1 __kernel void algorithm (
2     __global double* A_device,
3     __global int* problemSize_device )
4 {
5       int ii=get_global_id(0); int i = 0;
6     for(i = 0; i < numberOfFloatingPointsToBePerformed; i ++)
```

[14] It can be noted that the code of the user could be a much more effective implementation. This does not affect the fact that HybChive can achieve a speedup since an additional variant results in less necessary floating point operations for the respective user implementation anyway.

```
7     {
8         A_device[ ii ] += 1;
9     }
10 }
```

The following graph illustrates the runtime of a first run of the HybChive - set with the parameters n = 10 and numberOfFloatingPointOpertationsToBePerformed = 100.000.000.

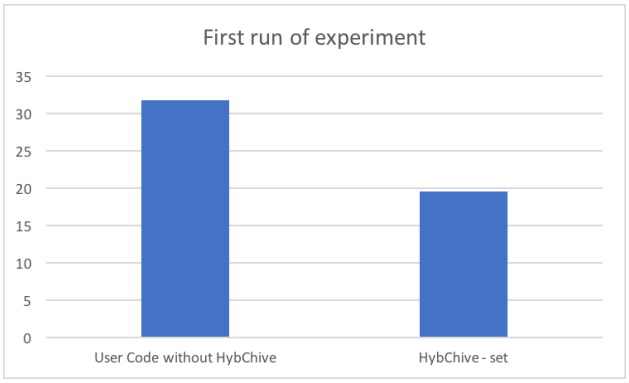

Figure 4.4: Comparison of the run-times in seconds needed consumed by the naive user implementation and the HybChive - set with two variants, one being for a CPU and one being for a GPGPU. In this experiment HybChive achieves a speeup of 1.6 compared to the naive user implementation

Figure 4.4 illustrates that the HybChive - set consumes 19 seconds whereas the naive user code implementation consumes about 31 seconds. Since the variant for the CPU of the HybChive - set is the same as the naive user - implementation it can be observed that the speedup is achieved by the additional GPGPU that processes some of the input matrix in parallel to the CPU. The speedup of this first run is thus round about 31 / 19 = 1.6. Until now it remains unclear how a different number of needed floating point operations and a different matrix size influence the speedup.

The second run of the experiment investigates the behaviour with a smaller number of needed floating point operations whereas the size of the input matrix remains the same. The experiment is performed thus with the parameters n = 10 and numberOfFloatingPointOpertationsToBePerformed = 50.000.000 to perform half the floating point operations compared to the first run of the experiment.

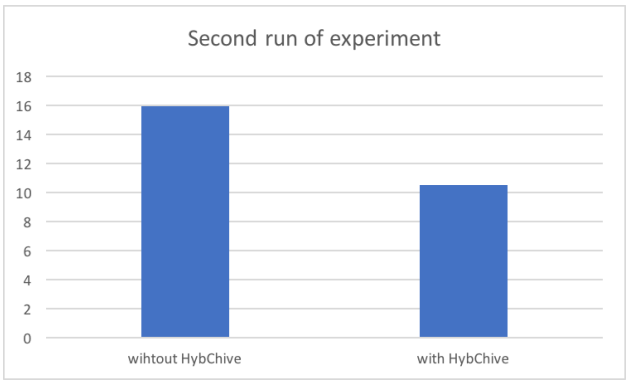

Figure 4.5: Comparison of the run-times in seconds needed consumed by the naive user implementation and the HybChive - set with two variants, one being for a CPU and one being for a GPGPU. In this experiment HybChive achieves a speeup of 1.6 compared to the naive user implementation, similarly to the first experiment.

Figure 4.5 illustrates that the HybChive - set consumes 10 seconds whereas the naive user code implementation consumes about 16 seconds. The speedup of this second run is thus round about 16 / 10 = 1.6 which is equal to the speedup of the first run of this experiment.

The next change of the number of floating point operations to be performed illustrates if the speedup decreases at a certain point. The number of the following run of the floating point operations to be performed is thus chosen to be numberOfFloatingPointOpertationsToBePerformed = 500.000.

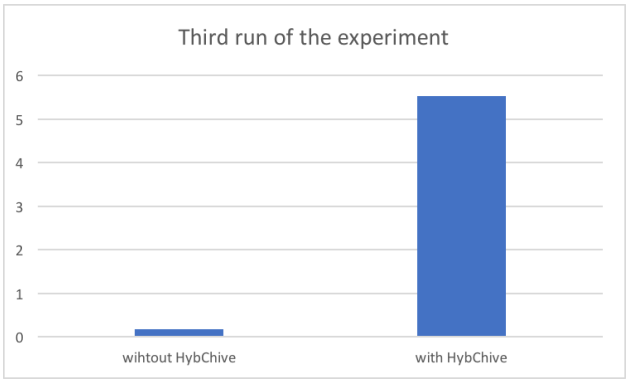

Figure 4.6: Comparison of the run-times in seconds needed consumed by the naive user implementation and the HybChive - set with two variants, one being for a CPU and one being for a GPGPU. In this experiment the naive user implementation is faster than HybChive since the number of floating point operations is not large enough to amortise the overhead of HybChive by parallel processing

Figure 4.6 illustrates that the HybChive - set consumes 5 seconds whereas the naive user code implementation consumes about 1 seconds. It can be thus observed that at a certain point, if the number of floating point operation per entry of the matrix is not sufficiently large anymore, the implementation of the user is faster than the HybChive - set due to the overhead produced by HybChive.

In a last experiment the size of the input matrix is increased. Thus the number of floating point operations is larger compared to the third experiment and the parallel processing of the matrix by HybChive amortises the HybChive headover accordingly.

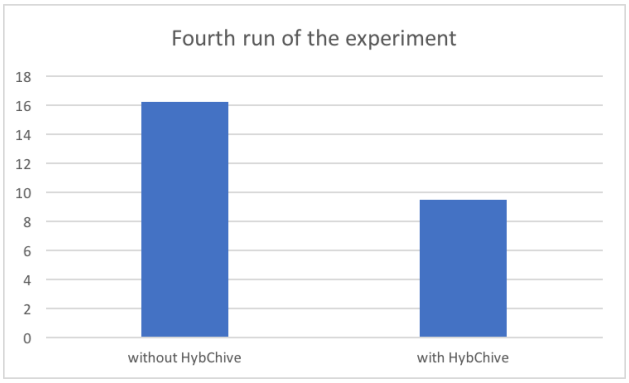

Figure 4.7: Comparison of the run-times in seconds needed consumed by the naive user implementation and the HybChive - set with two variants, one being for a CPU and one being for a GPGPU. The parameters in this experiment are n = 100 and numberOfFloatingPointOpertationsToBePerformed = 500.000

As illustrated in 4.7 It can be concluded that a certain number of floating point operations have to be performed for a HybChive - set to be more efficient than the related user implementation without HybChive.

Chapter 5

Resulting architecture

This chapter described the architecture that is a major result of the experiments and the software development iterations of this thesis. It aims at illustrating major aspects of the architecture of HybChive in a concise way.

5.1 Abstract Layer

Figure 5.1 illustrates the most abstract layer of HybChive. The user accesses HybChive via an API. The endpoint of the API is the scheduler. The scheduler reaches out to the HybChive - sets and the Data distribution component via an API since both Components are executed as own processes. The dots indicate that a connection exists between HybChive - sets and respective data distribution components which can be designed loosely coupled and tightly coupled to each HybChive - set. Loosely coupled data distribution routines can be applied to more than one HybChive - set whereas tightly coupled data distribution components only work with a specific HybChive - set.

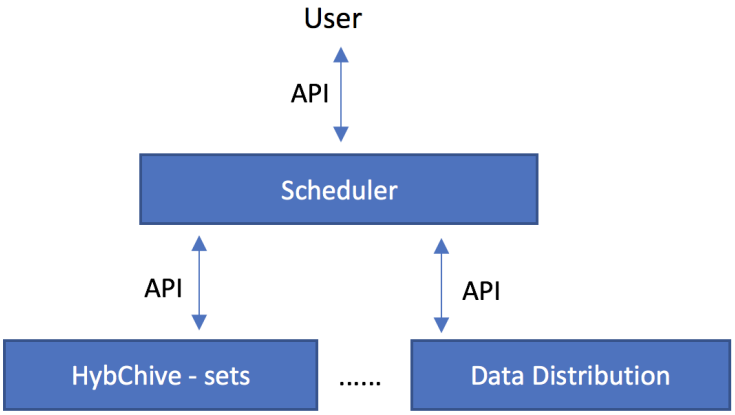

Figure 5.1: Abstract Architecture Layer

Enhancements of the abstract architecture layer

HybChive - Log Currently, the HybChive - log does not show the date and time but only the seconds gone by since 1970 due to the library default settings from C time.h. The enhancement would be to show date and time so that a hardware maintainer has a clear picture of the respective hardware unit.

Create possibilty to design multiple optimisation parameters per device Currently, the data distribution copmonent only provides the possibilty to optimise one single type of performance data and input parameter pattern. It is aimed at creating a possibilty to provide HybChive - set - specific and general optimising pattern so that the performance parameter can be optimised that is desired by the user.

Create mechanism to store performance data linked to a device Currently, the performance data does not store which device it belongs to. The enhancement aims at including the information which device the perfor-

mance data belongs to so that the performance data becomes reuseable for other users and tests have to be only run once per device.

Implement the possibilty for the user to choose which variant should be executed Currently, all variants are executed by HybChive after the invocation of the HybChive - method. The feature to be implemented for this enhancement is to create for the user the possibilty to choose which variant or which sets of variants the user wants to execute.

Refactoring the directory structure Currently, the directories of the HybChive - sets reside in the same directory as the core - routines. The directory structure of HybChive should be refactored to make the different files more clear.

Refine the clearance of the shared memory segments Currently, HybChive deletes all shared memory segments after a HybChive - function call. This process should be refined so that HybChive only deletes the shared memory segments that were created by the respective HybChive - function.

Only execute one variant per device This enhancement aims at automatically analysing which variant is written for which device or for which devices. This should lead to the functionality that only one variant is executed per device.

5.2 Scheduler Layer

Figure 5.2 illustrates the architecture of the scheduler. The scheduler is the endpoint of the API through which the user can use HybChive. The scheduler is written as a library for the respective computer - language. If one wants to extend the API for another programming language an according library

has to be written.[1]. In general, every computer language that can perform bash - commands and that can create shared memory segments can access the HybChive - sets. As a result, computer - languages that are compiled to a virtual engine are less likely to be able to create shared memory segments and are less suitable for the use with HybChive, as an example the possibility to create a HyChive - API for JAVA cannot be guaranteed.

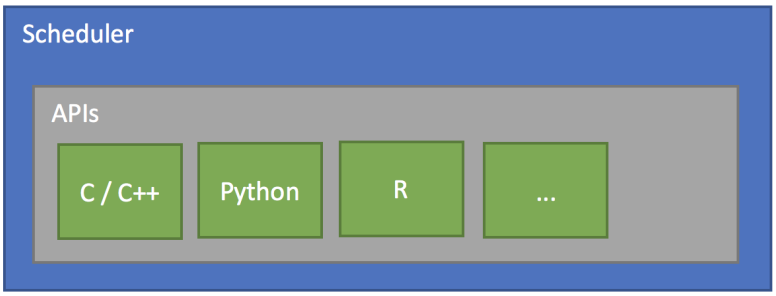

Figure 5.2: Scheduler Architecture

Enhancements for the scheduler

Ease the process of creating an API for an additional programming language Currently, all parts of the scheduler reside in function body of the library through which the user can access HybChive. The process of migrating all possible parts of the scheduler to a function that can be used as a stand alone process would ease the process of creating a client side library for the API of an additional programming language.

Compile variants during runtime only of necessary Currently, each variant is compiled in the course of every single HybChive function invo-

[1] One enhancement of HybChive is that all possible paths of the scheduler are migrated to a HybChive - set so that the library becomes as small as possible. This enhancement can be found here

cation. This process can be optimised since the variants do not have to be always compiled if the hardware environment or parameter don't change.[2]

Execute variant tests in parallel Currently, the variants are tested sequentially. This enhancement aims at executing the tests in parallel if the variants are not written for the same device.

Define process for what should happen if variant fails during runtime Currently, HybChive does not react if a variant fails during runtime. This feature can be implemented by a routine in the scheduler that continously checks if a variant is still working. If it fails, the scheduler can start a reassignment of the input - data to other variants.

5.3 HybChive - set Layer

Figure 5.3 illustrates the HybChive - set layer. HybChive consists out of n sets with different functionality. The set layer is designed in a way that additional sets can be easily added. To add a set, the user only has to copy the HybChive - set template and rename it. The HybChive - set template consists of variant - templates for different computer languages. If a user wants to create a HybChive - set for two GPGPUs for instance, he takes the Variant Template for C, copies it once and then add its functionality. Both variants will be executed in parallel if the user defines this in the API. In general, each user conists of a variant - file which contains the functionality that will be executed at last. In the varianttest - file, different parameters can be tested to investigate which set of parameters achieves the better performance measure that the user wants to optimize. The varianttest could for example test different kind of global and local workdimensions for a GPGPU to optimise the runtime of the respective variant.

[2] If parameters that influence the performance change due to the data distribution component variants might be compiled again.

At last, each HybChive - set consists of n .performance files that contain the information which set of parameters for the HybChive - variant result in which performance measure. The .performance - files are generated by the varianttest.

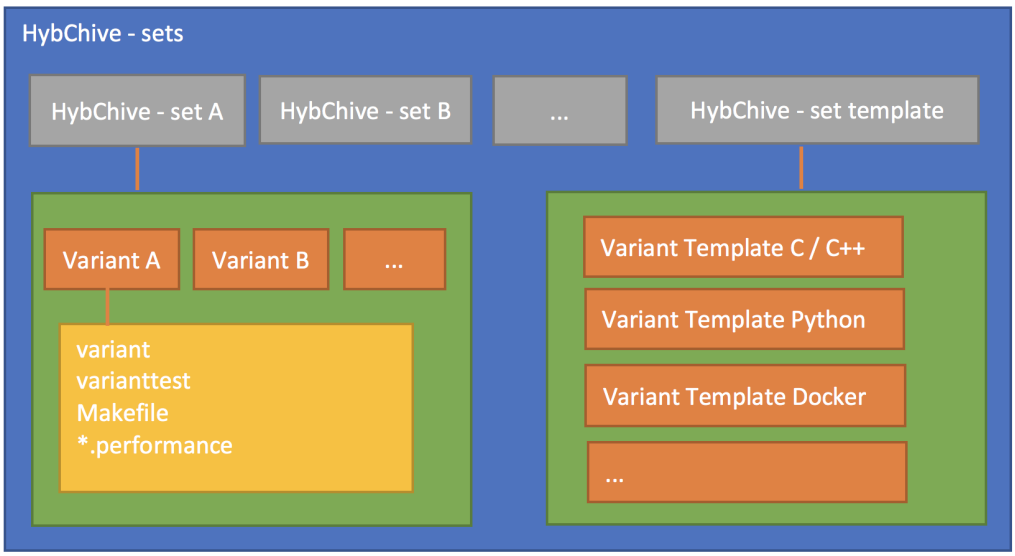

Figure 5.3: HybChive - set Architecture

Enhancements for the HybChive - set layer

Expand input possibilty for int, char,etc Currently, the biggest possible input size of the input arguments are defined manually by the implementation expert. This enhancement aims at creating a procedure that tests each device to find out which biggest input size for each device is possible.

The body of a HybChive - variant is a function call itself This enhancement aims at creating the possibilty for the user to only execute the native variants in a HybChive - set being those variants that are written in the same computer programming language that the user is calling the

HybChive - function from. This could be desired to avoid the botteleneck of copying the input arguments for the HybChive - set in the shared memory segments.

Organise the library paths for the Makefiles in a centralized way Currently, the user has to indicate the flags for each variant in the directory of each variant. If two variants are written in OpenCL, the implementation expert has to include the flags in each variant. This process can be centralised which this enhancement focusses on.

5.4 HybChive at run-time

Figure 5.4 illustrates how many processes are instantiated due to the call of one HybChive - function. The user code including the invocation of the HybChive - function is one process. The scheduler resides in this process as well and instatiate all other processes. The first additional process is the data distribution component. This component is designed as an own process since the desire exists for hardware being used by many users this component should be able to handle this szenario through extensions as well and as a result cannot be included in the process of one single user. Each variant of the HybChive - set is executed as an own process and accesses the same shared memory segment created by the scheduler. At last, all shared memory segments are freed by an additional process. This is designed as an own process to created the possibility to maintain created shared memory segments for future hybChive - sets of the user so that data dependencies could be exploited through future enhancement implementations.

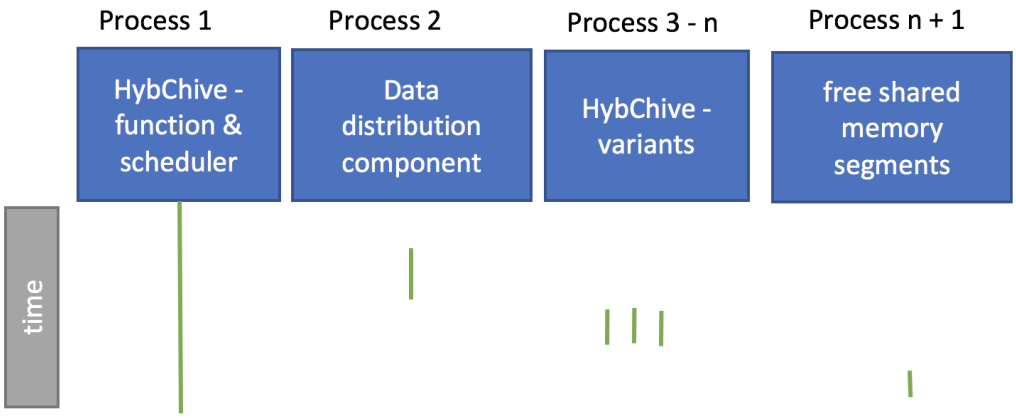

Figure 5.4: HybChive function invocation at run - time

Chapter 6

Conclusion

The framework "HybChive", developed within this thesis, provides a computer programming language agnostic and portable way to program heterogeneous high performance computing architectures.

It has been shown that compute-bound algorithms can achieve a speedup through the use of HybChive by using a variety of computing devices in parallel.

Performance tuning is achieved through component based annotation where tests analyse the influence of different component parameters.[1]

A runtime - system that analyses the behaviour of the software during runtime has not been implemented, it has been shown that it can be implemented through extensions in the developed architecture.

The possibility to add on to HybChive for third party users has been designed in an easy way and can be performed on Github.[2]

Lastly, numerous enhancements were defined to extend the functionality of HybChive and to investigate the combination of HybChive with other frameworks like OpenML, Docker, LAPACK and high performance computing GPGPU libraries.

[1] Components are called variants in HybChive

[2] HybChive - sets and variants can be included as submodules and thus HybChive becomes a collaborative framework

References

Literature

[1] S. Benkner et al. “PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems”. In: *IEEE Micro* 31.5 (Sept. 2011), pp. 28–41 (cit. on p. 12).

[2] Stijn Eyerman and Lieven Eeckhout. “System-level performance metrics for multiprogram workloads”. In: *IEEE micro* 28.3 (2008), pp. 42–53. URL: http://web.cse.ohio-state.edu/~zhang/2008-DS-CSE-788/MultiprogramWorkloads.pdf (cit. on p. 8).

[3] Leonardo Fialho and James Browne. “Framework and Modular Infrastructure for Automation of Architectural Adaptation and Performance Optimization for HPC Systems”. In: *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Cham: Springer International Publishing, 2014, pp. 261–277. URL: http://dx.doi.org/10.1007/978-3-319-07518-1_17 (cit. on p. 14).

[4] M. Grossman, M. Breternitz, and V. Sarkar. “HadoopCL: MapReduce on Distributed Heterogeneous Platforms through Seamless Integration of Hadoop and OpenCL”. In: *Parallel and Distributed Processing Sym-*

posium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International. May 2013, pp. 1918–1927 (cit. on p. 14).

[5] http://projects.csail.mit.edu/petabricks/. 2014 (cit. on pp. 13, 14).

[6] http://www.cercs.gatech.edu/projects/HyVM/Runtime. 2009 (cit. on p. 13).

[7] Kamran Karimi, Neil G. Dickson, and Firas Hamze. "A Performance Comparison of CUDA and OpenCL". In: *CoRR* abs/1005.2581 (2010). URL: http://arxiv.org/abs/1005.2581 (cit. on p. 5).

[8] J. Lin. "MapReduce is Good Enough? If All You Have is a Hammer, Throw Away Everything That's Not a Nail!" In: *ArXiv e-prints* (Sept. 2012). arXiv: 1209.2191 [cs.DC] (cit. on p. 14).

[9] Renato Miceli et al. "AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications". In: *Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers*. Ed. by Pekka Manninen and Per Öster. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 328–342. URL: http://dx.doi.org/10.1007/978-3-642-36803-5_24 (cit. on pp. 12, 13).

[10] Karsten Schwan, Ada Gavrilovska, and Sudha Yalamanchili. "HyVM - Hybrid Virtual Machines - Efficient Use of Future Heterogeneous Chip Multiprocessors". In: *Architecture of Computing Systems - ARCS 2010: 23rd International Conference, Hannover, Germany, February 22-25, 2010. Proceedings*. Ed. by Christian Müller-Schloer, Wolfgang Karl, and Sami Yehia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–1. URL: http://dx.doi.org/10.1007/978-3-642-11950-7_1 (cit. on p. 13).

[11] Stephan Soller. *GPGPU origins and GPU hardware architecture*. Tech. rep. FB 1: Druck und Medien, 2011 (cit. on p. 4).

[12] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1863103.1863113 (cit. on p. 15).

Online sources

[13] Khronos Group. *khronos.org/*. Jan. 2016. URL: https://www.khronos.org/ (cit. on p. 5).

[14] NVIDIA. *nvidia.com*. Nov. 2016. URL: http://www.nvidia.com/object/gpu-cloud-computing-services.html (cit. on p. 6).

[15] Open Community Runtime project. URL: https://01.org/open-community-runtime (cit. on p. 14).

Appendices

Appendix A

Abstract

High performance architectures become more and more heterogeneous. A user of this architecture has to know different programming languages to use all devices in parallel. Often specialised programmers are asked to develop routines for parts of the architecture.

This thesis provides a brief overview at first of technologies that address the programmability of heterogeneous architectures. Afterwards, this thesis defines needs of a holistic workflow and of a holistic software architecture with the help of which software is developed by a specialist for an end user which can be steadily extended for additional hardware in a heterogeneous high performance computing architecture.

As an experimental software approach, the developed framework aims at optimising embarrassingly parallel parts of the code referring to individually chosen parameters, for example the throughput of the code. Targeted architectures consist of a single node and attached devices which, by using the framework, process data stored in shared memory segments in parallel.

Furthermore, software architectural aspects and extensions will be discovered and described which satisfy major needs of the participating stakeholders in such an environment, such as the user of the code as well as the maintainer of the architecture. With the help of experiments the function-

ality of HybChive will be demonstrated and possible limiting aspects and future directions will be discussed.

The thesis will develop a prototype of the HybChive – framework.

Appendix B

Abstract in German

High Performance Architekturen werden immer heterogener. Der Benutzer einer solchen Architektur muss verschiedene Programmiersprachen beherrschen, um alle Computing - Devices parallel nutzen zu können. Oft werden spezialisierte Programmierer beauftragt, Routinen für Teile der Architektur zu entwickeln.

Die Thesis gibt zunächst einen kurzen Überblick über Technologien, die sich mit der Programmierbarkeit heterogener Architekturen beschäftigen. Anschließend werden die Anforderungen eines ganzheitlichen Workflows und einer ganzheitlichen Softwarearchitektur definiert, mit deren Hilfe die Software von einem Spezialisten für einen Endanwender entwickelt wird und die ebenfalls stetig für hinzukommende Hardware in einer heterogenen Software - Architektur ergänzt werden kann.

Das im Zuge einer experimentellen Herangehensweise entwickelte Framework zielt darauf ab, Teile des Codes, die "embarassingly parallel" sind, anhand individuell gewählter Parameter zu optimieren, zum Beispiel den Datendurchsatz eines Codeteiles. Architekturen, für die das Framework entwickelt wird, bestehen aus einem Single Node mit angehängten Devices, die durch das Framework Daten in Shared Memory Segments gleichzeitig verarbeiten.

Darüber hinaus werden Aspekte und Erweiterungen der Softwarearchitek-

tur beschrieben, die ermöglichen, wichtigen Bedürfnissen der beteiligten Akteure in einer solchen Umgebung gerecht zu werden. Akteure sind hier beispielsweise der Endanwender und der Betreuer der Architektur. Mit Hilfe von Experimenten wird die Funktionalität von HybChive demonstriert und mögliche limitierende Aspekte und zukünftige Perspektiven diskutiert.

Im Zuge der Arbeit wird ein Prototyp des HybChive - Frameworks entwickelt.