A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at the

Department of Computer Engineering and Microelectronics
Embedded Systems Architectures (AES)
Technische Universität Berlin

Author

Daniel Schürmann

Supervisor

Dipl.-Ing. Jan Lucas

Referees

Prof. Dr. Ben H. H. Juurlink,

Prof. Dr. rer. nat. Sabine Glesner

Declaration

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help.
Only the sources and resources listed were used.

Berlin, January 31, 2018

Daniel Schürmann

Abstract

For many applications modern GPUs could potentially offer a high efficieny and performance. However, due to the requirement to use specialized languages like CUDA or OpenCL, it is complex and error-prone to convert existing applications to target GPUs.

OpenMP is a well known API to ease parallel programming in C, C++ and Fortran, mainly by using compiler directives. With the introduction of new features in OpenMP 4.0 to support offloading to accelerator devices, OpenMP became a potential programming model to utilize GPUs and easily convert existing code. Different implementations for OpenMP accelerator offloading have been developed for Nvidia GPUs and Intel devices. However, these implementations are vendor-specific and lack portability.

OpenCL, on the other hand, is a framework for writing parallel applications in a portable way. In this work, we design and implement an extension for the Clang compiler and a runtime to offload OpenMP programs onto GPUs using a SPIR-V enabled OpenCL driver. The OpenMP execution model was successfully mapped to the OpenCL execution model providing an efficient technique to parallelize applications while maintaining portability. According to our experiment results, we were able to obtain a speed-up of 327.8 for a simple kernel and a speed-up of 1.9 for a real-world benchmark application compared to serial code.

Zusammenfassung

Für viele Anwendungen wären moderne Grafikkarten aufgrund ihrer Effizienz und Leistung gut geeignet. Aufgrund der Notwendigkeit, spezialisierte Programmiersprachen wie CUDA oder OpenCL zu verwenden, ist es jedoch aufwändig und fehleranfällig, bereits bestehende Anwendungen für die Ausführung auf Grafikkarten zu konvertieren.

OpenMP ist eine verbreitete Programmierschnittstelle mit dem Ziel durch die Verwendung von Compiler-Direktiven die parallele Programmierung in C/C++ und Fortran zu vereinfachen. Mit der Einführung neuer Funktionen in OpenMP 4.0, wie der Unterstützung von Beschleunigerkarten bei der Ausführung, wurde OpenMP ein mögliches Programmiermodell, um Grafikkarten zu nutzen und auf einfache Art existierenden Code zu portieren. Verschiedene Implementierungen mit Unterstützung für Nvidia Grafikkarten und Intel Prozessoren wurden entwickelt. Diese Implementierungen sind jedoch herstellerspezifisch und nicht portierbar.

OpenCL hingegen ist ein Framework, mit dem parallele Anwendungen portabel geschrieben werden können. In dieser Arbeit wird eine Erweiterung für den Clang Compiler sowie eine Laufzeitumgebung entworfen und implementiert, um OpenMP-Programme über einen SPIR-V-fähigen OpenCL-Treiber auf Grafikkarten auszuführen. Das OpenMP Ausführungsmodell wurde erfolgreich auf das OpenCL Ausführungsmodell abgebildet und bietet damit eine effiziente Technik, um Anwendungen zu parallelisieren und dabei die Portabilität zu erhalten. Im Ergebnis konnte eine Beschleunigung von 327.8 für einen einfachen Kernel und eine Beschleunigung von 1.9 für eine realistische Benchmark-Anwendung im Vergleich zu seriellem Code erreicht werden.

MASTER THESIS

OpenMP Accelerator Offloading with OpenCL using SPIR-V

Daniel Schürmann

Table of Contents

I. Introduction

In recent years, the speed of CPU performance enhancements has slowed down to small improvements, while the performance of GPUs continues to increase sharply. This led to the situation that typical desktop GPUs are able to outperform CPUs in terms of raw performance and efficiency. GPUs were originally developed for the purpose of graphics applications only and consisted of fixed-function units. The demand for increasing flexibility regarding the programmability of graphics shaders led to today's GPU architectures with massively parallel processors consisting mainly of vector units. This made it possible to leverage the performance and efficiency of the GPU for general purpose GPU programming (GPGPU). Due to the inherent parallelism of the GPU's architectures, different programming models for GPGPU programming were developed, defining new programming languages or introducing specialized library functions and data types to be used.

This property makes these programming models suitable for new high performance applications running on GPUs. Existing legacy programs, however, would have to be rewritten. While rewriting existing programs to make use of the GPU's compute capabilities can be very beneficial in terms of performance, and thus lowers future hardware expenses, in general, it involves a number of complications to be taken care of. The re-implementation might introduce new bugs to the program and make unfixed bugs harder to keep track of. The rewritten programs have to be semantically equivalent, a property which is difficult to proof.

A different approach to overcome this problem and still make use of the GPU's performance and efficiency are programming models based on compiler directives. Compiler directives are code annotations which can be applied on already existing source code. They alter the runtime behavior and control flow of the program to make use of multiple processors or accelerators and speed up the execution time. Ideally, these directives don't modify the semantic meaning of a program and thus, could potentially be ignored by a compiler without affecting the output of the program. On the other hand, compilers with support of the directives can distribute the program execution over different threads or make use of vector units to increase performance. OpenMP is a programming model based on compiler directives and aims to ease the parallelization of code by providing a simple and flexible interface for programming parallel applications in a portable and scalable manner. With Version 4.0, OpenMP introduced accelerator offloading, and thus, became a potential programming model for GPGPU programming. However, current implementations are available for NVIDIA GPUs only. OpenCL, on the other hand, is a cross-vendor effort to provide a framework for the parallel programming of heterogeneous platforms. In this work, a portable implementation of OpenMP accelerator offloading using OpenCL with SPIR-V kernels is presented. The proposed implementation is twofold and consists of a compiler extension for Clang [1] to generate SPIR-V kernels and an OpenMP runtime plugin using OpenCL to offload the kernels to an accelerator.

The next chapter contains a detailed description of the development and history of programming models for GPGPU programming. Existing compilers with support for compiler directives and related work are described in Chapter 3. The OpenMP programming model is presented more detailed in Chapter 4. A description of the idea and the circumstances leading to the design of this work can be found in Chapter 5 and Chapter 6 covers the actual implementation. Two benchmarks are presented in Chapter 7 to show the potential benefits of this work. Chapter 8 concludes the work and discusses open problems and Future Work.

II. History of GPGPU

Since the architectural improvements of GPUs made it possible to use them as accelerators for parallel processing, various efforts have been made to ease GPGPU programming. This chapter describes the history of GPGPU programming models beginning with the use of graphics shaders in the early years up to the development of frameworks specifically designed towards GPGPU programming to template libraries for C++. Finally, programming models based on compiler directives are presented.

2.1 Early Years

With the advent of programmable shaders and floating point support of modern GPUs around 2001, general purpose computing became practical on GPUs. In these days, it was required to reformulate computational problems with graphics primitives supported by the shading languages of OpenGL and DirectX [2]. Despite these limitations, the high increase in performance and efficiency showed the possibility to use GPUs as parallel processors and established the new use-case: General Purpose GPU Programming. Today's graphics APIs also support the execution of compute shaders, which can do almost arbitrary calculations embedded in a graphics program with the restriction that control flow has to be structured.

2.2 Domain specific languages

General purpose programming languages specifically designed towards the programming of GPUs, like BrookGPU [3] and Sh [4], marked the beginning of modern high-performance computing on GPUs around 2004. These languages aimed to hide graphics-related details from the programmer and provided a data-parallel programming model. In the Single Program Multiple Data (SPMD) programming model, sequentially written programs run simultaneously on multiple threads but on different data [5].

Close to Metal (CTM) was a short-lived programming interface developed by ATI in 2006 and based on the Brook language [6]. It exposed the native instruction set and memory and thus, granted previously unavailable low-level functionality over the underlying parallel processing architecture of graphics hardware to the programmer. The disadvantage of this approach, however, is a lack of portability as the instruction set and architecture of GPU hardware may change from generation to generation.

In 2007, NVIDIA released the CUDA platform and framework [7]. CUDA had increasing success by combining a subset of C++ with extensions for GPGPU programming where host and device code can be written in a single source file to program NVIDIA GPUs. By targeting an intermediate representation, NVPTX, rather than the GPU's native instruction set, more languages, like Fortran or OpenCL C, can be used to work with the CUDA runtime. Constant iterations, which introduced new features like unified memory and the support of shared libraries, as well as the availability on many NVIDIA GPUs made CUDA today's most popular framework for GPGPU programming.

In response to the proprietary CUDA framework, which is only available for NVIDIA hardware, the Khronos Group announced OpenCL in 2008 [8]. OpenCL is an open standard for parallel programming of heterogeneous platforms and aims to support cross-vendor portability on different platforms consisting of CPUs, GPUs, DSPs, FPGAs and other processors. OpenCL defines an API callable from the host to manage memory and kernel execution on the device as well as a kernel language. The kernel language, OpenCL C, is derived from ISO C99 and extended with additional functionality to support parallel programming of accelerators [9]. OpenCL specifies the kernels to be compiled at runtime. Although this guarantees portability between architectures, the implied overhead and concerns regarding the exposition of the kernel's source code led to the Standard Portable Intermediate Representation (SPIR) as an additional, optional technique to provide precompiled kernels. This way, as for CUDA, it became possible for OpenCL devices to be targeted by other programming languages than OpenCL C. With OpenCL 2.1, the successor of SPIR, SPIR-V, became mandatory for all compliant OpenCL implementations [10].

The recently released OpenCL 2.2 specification introduced the new OpenCL C++ kernel language with support of classes, templates, lambda expressions and more. This development closes the gap between CUDA and OpenCL in terms of programmability.

OpenCL is announced to converge with Vulkan, a modern graphics API from the Khronos Group [11]. Consequently, in the future, Vulkan compute shader could do fully general purpose computations without any restrictions.

2.3 Higher Order Functions

The C++ AMP library, developed by Microsoft in 2012, is intended to further ease GPU programming, mainly by passing lambdas to a new introduced parallel for-loop function. C++ AMP is an open standard with the initial implementation being based on DirectX 11 [12].

In 2014, the Khronos group announced SYCL, a single-source programming model for OpenCL. SYCL is based on pure C++ with template libraries and runs on all OpenCL devices with support of SPIR [13]. ComputeCpp from Codeplay and the open-source project, triSYCL, are currently available implementations of SYCL.

As part of the GPUOpen software suite, released in 2016, AMD developed the Heterogeneous Compute Compiler (HCC) [14]. HCC provides different accelerator modes to support heterogeneous offload to AMD GPUs via Heterogeneous System Architecture (HSA) enabled runtimes.

First, the Heterogeneous-compute Interface for Portability (HIP) is a C++ runtime API and kernel language to easily convert existing CUDA applications to AMD GPU hardware. For this purpose, HIP shares much of the syntax and language features with CUDA. Additionally, HCC supports C++ AMP and HC, a C++ template library which is inspired by C++ AMP but provides more low-level control over the memory and removes some restrictions. However, HCC can be seen as the conceptual successor of CTM and compiles device code directly to the native ISA of supported AMD GPUs. This means that the compiled binaries are not portable between different hardware architectures.

2.4 GPGPU Programming Based on Compiler Directives

Although the programmability of GPU kernels increased drastically from restricted graphics shaders to full-featured C++ template libraries, already existing programs still have their data parallel code sections to be rewritten to make use of the higher performance and efficiency of modern GPUs. Different programming models based on compiler directives were developed, aiming to reduce the effort to increase the performance by parallelizing a program. By using compiler directives, the programmer guarantees data-race freedom of the marked code section, which otherwise could not be determined statically.

In 2007, Hybrid Multi-Core Parallel Programming (HMPP) was presented by Romain Dolbeau et. al. [15]. It was the first directive-based programming model designed towards portable GPGPU programming. HMPP uses a preprocessor and code generator to transform annotated code sections according to the directive into target source code, e.g. CUDA. Then, the vendor programming tools are used to compile these code sections. Finally, the compiled binaries are linked with the HMPP runtime into a shared object file. The targets have to be specified as part of the directives, and thus, the code is not directly portable across different platforms. Despite becoming an open standard in 2011, OpenHMPP was not able to gain much recognition, with the only implementations today being the CAPS compiler and the PathScale ENZO Compiler Suite.

OpenACC (for Open Accelerators) is a programming standard from 2011 [16]. Its purpose is to simplify parallel programming of heterogeneous systems consisting of a CPU and one or more GPUs. Unlike HMPP, most compilers with support of OpenACC directly translate the marked code sections to the target specified via compiler flags and not as part of the directives. This way, the code becomes portable across different target platforms.

OpenMP (for Open Multi-Processing) predates the other two directive-based models and was first published in 1997. Originally, it was designed to support shared memory multiprocessing in C/C++ and Fortran on various platforms. In response to the arrival of OpenACC, OpenMP introduced accelerator offloading with Version 4.0 in 2014 [17].

3. Related Work

In this chapter, information on the offloading capabilities for compilers with support of directive-based GPGPU programming are provided. At the beginning, the compilers with support of OpenHMPP are presented, followed by OpenACC, and finally the available compilers with OpenMP accelerator offloading support are introduced.

Two compiler with support for OpenHMPP are available. The “Compiler and Architecture for Superscalar and embedded Processors” (CAPS) is a common project from INRIA, CNRS, the University of Rennes 1 and the INSA of Rennes, and was the initial implementation of HMPP. CAPS’ source-to-source translation of the annotated code sections can target either CUDA or OpenCL C. PathScale Inc. was a company specialized in the development of optimizing compilers. The PathScale ENZO Compiler Suite is the second compiler with support of OpenHMPP. Differently from CAPS, ENZO does a native compilation to NVPTX for CUDA. Both compilers, CAPS and ENZO, also support OpenACC for the same targets. However, CAPS enterprise and PathScale are no longer in business and their development is no longer available.

Commercial compilers with support of OpenACC are available from The Portland Group (PGI) [18] and Cray [19]. These commercial compilers also use CUDA as back end via NVPTX. The same applies for a number of open-source and academic compilers. There are three academic research compilers which use a source-to-source approach to translate C code with OpenACC directives to the kernel language of the corresponding target. The Omni OpenACC Compiler from the University of Tsukuba is implemented by using the Omni compiler infrastructure and translates the source code to CUDA [20]. The Open Accelerator Research Compiler (OpenARC) is built on top of the Cetus compiler infrastructure and can additionally target OpenCL C and the Intel Many Integrated Core Architecture (MIC) [21]. The RoseACC compiler, a collaboration between the University of

Delaware and Lawrence Livermore National Laboratory, is based on the ROSE source-to-source compiler and produces OpenCL C kernels from OpenACC directives [22].

Two open-source compilers use native compilation for the accelerator. The OpenUH OpenACC Compiler, based on Open64, targets NVIDIA GPUs via NVPTX and CUDA and also AMD GPUs via the HSA platform [23]. The GNU Compiler Collection (GCC) has an NVPTX backend, and additionally, experimental support for the HSA platform and Intel MIC [24]. In 2016, Hao-Wei Peng and Jean Jyh-Jiun Shann described the translation of OpenACC programs to LLVM IR with SPIR kernels using Clang [25].

Due to the long history of the OpenMP specification, there exist many compilers with support of the shared memory parallelism directives. However, only few compilers have support for the offloading capabilities introduced from Version 4.0 of the OpenMP specification, most of them being commercial. The PGI compiler, the Cray compiler and the IBM XL compilers for POWER [26] generate NVPTX and support accelerator offloading to CUDA devices. For the Intel C++ and Fortran Compilers accelerator offloading to Intel MIC devices is supported [27].

The academic ROSE Research Compiler translates OpenMP source code to CUDA source code [28]. For the open-source compilers, GCC and Clang, OpenMP accelerator offloading is still under development, but can already be tested. GCC and Clang are able to target Intel MIC devices and additionally will support CUDA [29 - 32].

With advent of the offloading capabilities of OpenMP 4, some efforts have been made to efficiently translate OpenACC programs to OpenMP [33][34].

4. OpenMP

The preliminary goal of OpenMP is to enable easier and more productive parallel programming of shared-memory multiprocessing architectures. OpenMP is jointly defined by a consortium consisting of a group of major computer hardware and software vendors, the OpenMP Architecture Review Board (OpenMP ARB).

Therefore, OpenMP is defined to be portable across multiple platforms and architectures and scalable from small desktop computers to many-core supercomputers. In this chapter, the advantages and disadvantages of OpenMP are evaluated. Then, the execution model is described and a brief introduction into the parallelization and offloading capabilities is given. The chapter ends with a short overview over techniques to specify the data sharing and runtime behavior of OpenMP.

4.1 Advantages and Disadvantages

The biggest advantage of OpenMP, however, is that the original code in general does not need to be modified in order to be parallelized using the OpenMP programming model. This way, in most cases the source code remains a valid serial program when a compiler without support of OpenMP is used. The inserted compiler directives are then treated as comments and, thus, ignored. The necessity of only adding compiler directives to the code to get an increased performance makes OpenMP a convenient way to easily parallelize legacy code. Regarding the different compiler directives, library routines and environment variables, OpenMP gives the programmer fine-grained control over the control flow and data sharing of the program. Since Version 4.0, OpenMP allows the programmer to offload code sections to accelerators, such as GPUs, and consequently, program heterogeneous systems with OpenMP.

On the downside, OpenMP has no protection against the introduction of race-conditions or deadlocks. These kinds of bugs are hard to debug and can cost the time saved by using OpenMP. It is therefore in the responsibility of the programmer to ensure the correctness of the program. Although OpenMP code in general is portable across platforms and architecture, this does not apply to the achievable performance of a particular implementation. Differences in the number of cores, vector units and memory architecture can make it necessary to have different OpenMP statements for different target architectures in order to maximize the performance. This process is called tuning. Also, the potential speed-up is inherently limited by the proportion of the program that can be parallelized as postulated in Amdahl's law.

Despite the disadvantages, OpenMP provides a convenient way to parallelize code in a single-source style. The necessity of only adding a few lines of code to a sequential program in order to gain a noticeable speedup helped to make OpenMP a popular standard for parallel high-performance applications.

4.2 Terminology

The following sections are intended to be used as brief introduction into the OpenMP programming model and therefore try to do without the use of too specific terminology. However, some terms will be necessary to understand the basic concepts.

An OpenMP directive (or pragma) specifies how the compiler processes the input source code and which parallel actions are to take place. OpenMP directives have the following syntax:

`#pragma omp` *`directive-name [clause, ...]`*

They can be followed by clauses which specify the additional behavior to occur and end with a new line. A construct is an OpenMP directive together with the following associated code statement. This can be a single statement, a loop, or structured code block. The region of a construct is all the code that is encountered during the execution of the construct. This includes called routines and code introduced by the OpenMP implementation.

4.3 Execution Model

Differently from the SPMD programming model of kernel languages like CUDA or OpenCL C, an OpenMP program starts its execution single-threaded, just as any C/C++ or Fortran program. This initial thread executes sequentially on the host device. For the parallel execution, OpenMP uses a fork-join model where the threads are created during runtime. Every thread has the ability to create a team of threads to execute a code region in parallel. The new team consists of the thread that created the team, the so-called master thread, and potential additional threads. Depending on the implementation, the parallelism can be nested. OpenMP defines worksharing constructs to divide the work inside the construct among the threads of a team. The threads of a team can be synchronized and can also share data. When accessing shared data, the programmer is responsible to prevent potential data-races by using synchronization constructs, e.g. a barrier, atomic or critical constructs, or library routines. Different teams, on the other hand, run independently and can only be synchronized by using lock routines or critical sections (available only on host execution) while executing the enclosing region. OpenMP also offers a Single Instruction Multiple Data (SIMD) construct to transform a for-loop into a SIMD loop which executes the loop iterations concurrently using SIMD instructions.

If supported and available, the host device can offload code and data to other target devices. When offloading, OpenMP enables the programmer to declare the creation of a possibly unspecified number of teams on the device. Each device, including the host, has its own data environment and threads. The threads cannot migrate between devices. They may, however, migrate between the cores of one device.

4.4 Parallel Directive

The `parallel` directive is the most important compiler directive of OpenMP. When an executing thread encounters the parallel construct, a team of threads is created to execute the following region in parallel. The encountering thread becomes the new master thread of the team. The number of created threads can further be specified by using a `num_threads` clause. It is guaranteed by the OpenMP specification that at most that many threads are created. However, the minimum number depends on the implementation and runtime.

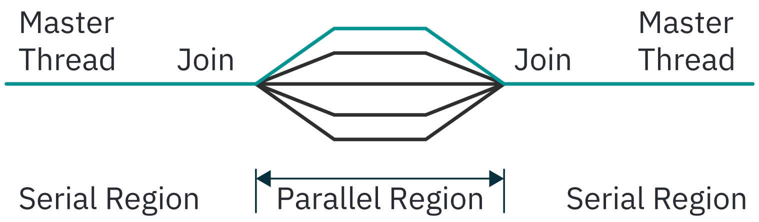

Figure 4-1: The master thread forks new threads at the beginning of a parallel region.

With data sharing clauses it can be specified which variables are shared between the threads or private to each thread. If not specified, by default variables declared outside of the region are shared and when declared inside they are private. The threads can be synchronized by using a `barrier` directive. A barrier must be encountered by all threads in the team before any thread is allowed to continue the execution beyond the barrier. Additionally, a `single` directive can be used to specify a code region to be only executed by one thread of the team. At the end of the region is an implicit barrier. This is not the case for the `master` directive which specifies a code section to be executed by the master thread only.

OpenMP allows for two different ways to divide work among the threads of a team. The loop construct, declared by using the `for` directive, may be used to exploit data level parallelism. The loop construct must be associated with at least one for-loop whose iterations are distributed across the threads that already exist in the team executing the parallel region. An optional `collapse` clause causes a number of loops to be collapsed into a larger iteration space that gets distributed across the threads. The distribution is done via partitioning of the iterations into chunks and assigning these chunks to the executing threads. A `schedule` clause can further specify the way how the iterations are divided into chunks and how these chunks are assigned to the threads.

```
void vector_add(int *A, int *B, int *C, int n) {
        #pragma omp parallel for
        for(int i = 0; i < n; i++) {
              C[i] = A[i] + B[i];
        }
}
```

Figure 4-2: A simple vector addition parallelized using OpenMP

The second way to divide work among the threads of a team is designed to exploit thread-level parallelism. A code region containing a set of structured code blocks that are to be distributed among the threads of the team can be marked with the `sections` directive. Then, the contained structured code blocks, preceded by a `section` directive, should be executed each by one thread of the team. Due to the SIMD design of modern GPU vector architectures the sections construct is executed sequentially and thus, disadvantageous.

4.5 Target and Teams Directive

With Version 4.0 of the specification, new offloading capabilities were introduced to OpenMP. As previously mentioned, the offloading capabilities were made in response to the OpenACC directive-based programming model. OpenMP specifies the offloading capabilities in addition to the existing parallelization capabilities to make use of the performance and efficiency of parallel high performance accelerators, like GPUs, in a single programming model.

The `target` directive can be used by the programmer to declare a code region to be offloaded to an accelerator. The code region gets compiled to one or more targets specified via compiler flags. For this purpose, the code region is transformed into a function or, when targeting a GPU, a kernel. A fallback version of this function is created to be executed on the host if the offloading fails.

At runtime, when a thread encounters the target construct variables get mapped to the device data environment and the target region gets executed on the device. The host waits for the execution to be finished before continuing its execution. However, a `nowait` clause can specify the host to start the target execution asynchronously. After the target execution has finished, the variables are copied back from the device to the host data environment. The variable mapping can further be specified to avoid unnecessary copies by using the map clause.

Differently from host execution which always starts with one team and one thread, the target execution can be started with multiple teams at the same time by using the `teams` directive. The teams directive, if specified, must be immediately succeeding the target directive. The number of created teams is implementation defined but can be bounded by using the `num_teams` clause. The `thread_limit` clause can be used to limit the total number of threads executing on the target device.

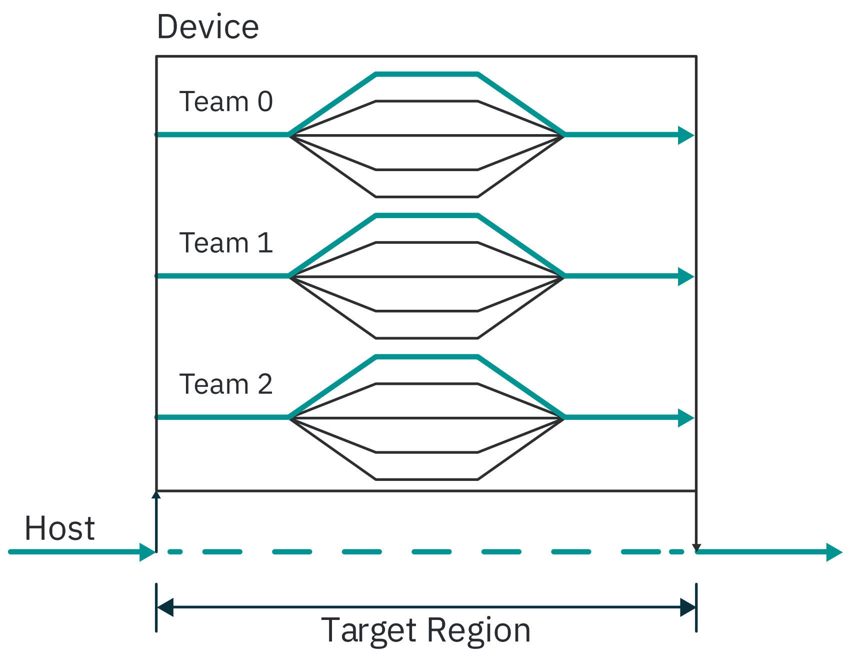

Figure 4-3: The number of teams is fixed across the whole target execution.

The target code region is executed by each team independently. The teams consist of the master thread only unless a parallel construct is encountered which creates a number of threads for each running team. In addition to the worksharing constructs for parallel regions, the `distribute` directive can be used to divide work among the teams executing the target region.

```
void saxpy(float* x, float* y, float a, int n) {
    #pragma omp target teams distribute parallel for map(to:x) map(tofrom:y)
    for (int i = 0; i < n; ++i){
          y[i] = a*x[i] + y[i];
    }
}
```

Figure 4-4: A small saxpy kernel gets offloaded to an accelerator.

The distribute construct works in the same way as the loop construct, and equally, exploits data level parallelism by the application to a for-loop. Furthermore, OpenMP allows for a `distribute parallel for` composite directive to distribute loop iterations among all the threads of all the teams executing the target region. This directive can also be combined with the previously mentioned `schedule` and `collapse` clauses to further specify the distribution of iterations among all running threads. This way, the highly parallel architecture of GPUs can be utilized appropriately with OpenMP.

4.6 Data-Sharing Clauses

OpenMP is a very explicit programming model such as it allows the programmer to have a detailed control over the data environment of a program. With every teams or parallel construct the programmer can define variables to be `shared` between the threads or `private` to each thread. Furthermore, variables can be `firstprivate` or `lastprivate`, meaning that a private variable is initialized by the master thread or the value of the last iteration from a worksharing construct is assigned to the original variable, respectively. The `reduction` clause can be used to define parallel reductions in OpenMP without the necessity of atomics or synchronization objects.

4.7 Runtime Library Routines and Environment Variables

An OpenMP implementation defines the default behavior with internal control variables (ICV) for things such as the number of threads created from a parallel region, the schedule used for worksharing loops or if nested parallelism is allowed. By changing these ICVs, the environment variables can affect the default behavior of the execution of OpenMP programs. Some of these ICVs, e.g. the ones mentioned above, can also be changed or retreived at runtime by using runtime routines. Other runtime routines serve the purpose of getting information like the number of threads in the teams or the team- and thread number of the current thread. Additionally, OpenMP defines runtime routines for synchronization between threads by using locks, for a wall clock timer and for device memory management.

5. Idea and Design

OpenMP is designed to be a portable multi-platform standard. However, portability is in practice only given if more than one implementation can be targeted. Although this is true for implementations regarding parallelization capabilities on CPUs, on GPUs OpenMP accelerator offloading is only usable on CUDA enabled devices from NVIDIA. OpenCL, on the other hand, is an open standard with conformant implementations from many different vendors.

The general idea of this work is a framework to offload the code which is annotated by OpenMP target directives to OpenCL devices. For this purpose, we generate SPIR-V kernels which get compiled at runtime and executed on the device.

In this chapter, we discuss the different circumstances leading to our design decisions. Therefore, the OpenCL landscape and the different OpenCL intermediate representations are evaluated. Afterwards, we consider the existing compiler frameworks and OpenMP runtimes to base our implementation on. In the second part of this chapter, an overview over the OpenCL execution and memory model, and a possible mapping from OpenMP to the OpenCL model is presented. In the final section, the compilation and runtime framework of this implementation is proposed.

5.1 OpenCL Landscape and Intermediate Representations

OpenCL is a framework consisting of an API and a kernel language. A conformant implementation of the OpenCL specification is called an Installable Client Driver (ICD). The ICD is a library implementation of the OpenCL API with the purpose of compiling and executing OpenCL kernels. Additionally, the ICD handles memory management and kernel parameter setting.

OpenCL offers two different ways to provide the kernel which is to be executed. The kernel can either be provided as source file consisting of OpenCL C code or in a binary format which contains the intermediate representation of the kernel. In both cases, the provided kernel is compiled by the OpenCL runtime into an executable targeting the device. One possible solution to target OpenCL devices with OpenMP code is a source-to-source transformation of the target code regions to OpenCL C code. As OpenCL C is based on C99, such code transformation would be feasible under some restrictions of OpenCL C, like the unavailability of recursion. The second possible solution is to target an OpenCL intermediate representation. The generated kernels could be used on any device with support of this particular intermediate representation. While this solution would allow to also support C++, the qualifications of the existing compilers (see below) led to the decision to use an intermediate representation as target format.

Since the initial release of OpenCL at the end of 2008, we saw a high adaption rate with conformant implementations being released from 15 different companies [35]. The OpenCL 1.2 specification is implemented by most vendors and already specified an extension for the intermediate language SPIR 1.2. The OpenCL specification 2.0 updated the extension for the intermediate language to SPIR 2.0. However, both intermediate representations have only to be supported optionally. At the end of 2015, the OpenCL 2.1 specification was released and includes the support of the SPIR-V intermediate language which is now mandatory for all conformant implementations. Two years later, official support is given by one single driver only, the Intel OpenCL SDK. Moreover, the Intel OpenCL SDK provides an OpenCL implementation which only targets CPUs. Despite making hardware that is feature-wise capable of OpenCL 2.1, the remaining hardware vendors did not yet implement the specification. However, Intel, Qualcomm and AMD provide OpenCL 2.0 drivers for their newer hardware with implementations of OpenCL 2.1 and 2.2 being expected to be released soon. The slow adoption of new OpenCL specifications make OpenCL 1.2, as common denominator, still the de-facto standard when targeting OpenCL devices. Regarding the intermediate representations, SPIR 1.2, and later SPIR 2.0, were based on LLVM IR while SPIR-V is fully defined by the Khronos Group and independent from the development of LLVM. Some facts led to the decision of using SPIR-V and OpenCL 2.1 as the targeted platform:

First, the development of LLVM brought compatibility issues between the dated SPIR versions and LLVM. While initially, LLVM could be transformed to SPIR by using a provided translator tool, for the current version of LLVM such a tool does not exist. Second, SPIR was only an optional extension of OpenCL. Thus, it depends on the vendor implementation if support is given, and it is highly uncertain if in future additional implementations are to come. SPIR-V, on the other hand, is mandatory since OpenCL 2.1 and every compliant implementation has to provide support for it. SPIR-V supports all of the features provided by OpenCL 2 on the device-side by exposing the OpenCL machine model. SPIR-V is also the intermediate representation of the Vulkan graphics API, which has seen a high adaption rate and what decreases the necessary effort for vendors due to shared code paths. For these reasons, the expectation is that in future there will be a broad support of SPIR-V by different vendors. Finally, although officially still on OpenCL 2.0, the AMDGPU-Pro OpenCL Driver [36] already supports the compilation and execution of SPIR-V kernels. Therefore, the implementation of this work can be tested on real GPU hardware.

5.2 Compiler and Runtime

One prerequisite the compiler must fulfill is to be available open-source. Other desirable qualifications are the ability to directly or indirectly generate SPIR-V IR code and to provide an existing OpenMP implementation. The existing OpenMP implementation is especially important as it allows code sharing between implementations and to have a combined compilation of host and target code from a single compiler invocation. Finally, a good approachability of the code base should be factored in as it can improve the quality and completeness of the implementation. The two major open-source compilers, GCC and Clang, both provide an existing OpenMP implementation targeting the Intel architecture. Internally, both compilers use intermediate representations for the code compilation, which later get lowered to machine instructions. The GCC internal IR is called GIMPLE while Clang uses the LLVM (formerly: Low Level Virtual Machine) IR. LLVM is comparably well documented and used by many academic publications [37]. While the SPIR versions 1.2 and 2.0 have been directly based on the LLVM intermediate representation, for SPIR-V the Khronos Group created the LLVM/SPIR-V Bi-Directional Translator to convert LLVM IR to and from SPIR-V. Nicholas Wilson accomplished to integrate this tool into LLVM to make SPIR-V available as target architecture [38]. The consideration of these different qualifications led to the decision to use Clang together with LLVM for the compilation process of the implementation.

For both compilers, GCC and Clang, there exist subprojects for the OpenMP runtime. GOMP, a part of the GCC infrastructure as well as the OpenMP subproject of LLVM contain the OpenMP runtime components necessary to execute OpenMP programs. Although the LLVM subproject aims for ABI compatibility with GOMP, and thus both runtimes should work interchangeably, the better integration with the chosen compiler gave the impact to base the runtime implementation of this work on the OpenMP subproject of LLVM.

5.3 OpenCL Execution and Memory Model

Just like OpenMP, OpenCL distinguishes between the host and one or more devices. The host is able to use the OpenCL API to compile the kernels and to interact with the devices through a command queue. The interacting can be the enqueueing of kernel execution or device memory management.

Before enqueueing, the host has to compile a program consisting of a set of kernels. Then a kernel instance is created from the compiled program. The kernel instance includes the kernel and the index space of workitems executing the kernel concurrently. The workgroups form a grid of up to three dimensions and are further divided into workitems. The composed index space is called an NDRange. Due to the SPMD programming model of OpenCL, every workitem executes the kernel function depending on the assigned ID from the iteration space.

On the device side, we have one or more compute units (CUs), each divided into one or more processing elements (PEs). A workgroup is being executed by a single CU with the PEs processing the workitems. While different workgroups run independently, i.e. there is no defined forward progress or ordering relation between the workgroups, the workitems within a workgroup can be synchronized by using barriers or other workgroup functions.

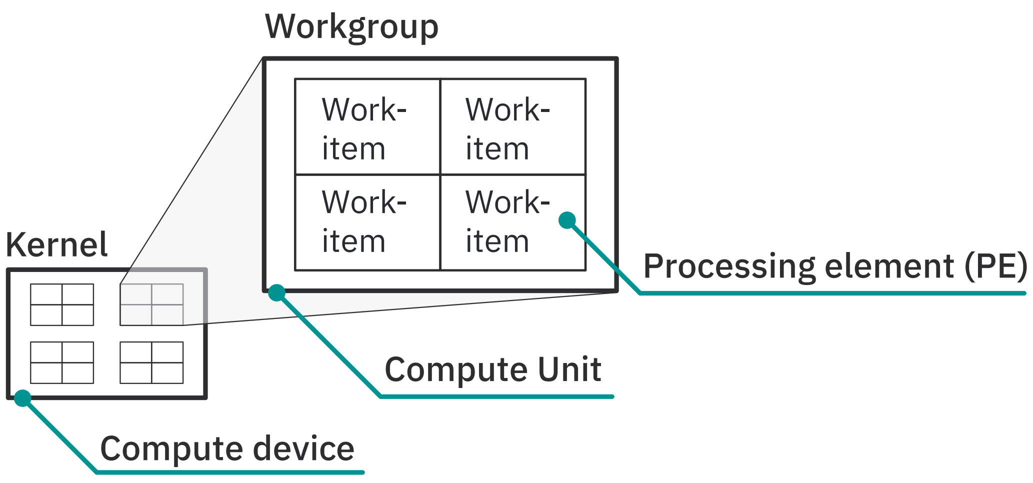

Figure 5-1: The OpenCL execution model.

OpenCL distinguishes between the host memory and the device memory. Memory objects can be moved between host and device through functions within the OpenCL API. The memory allocation on the device is either done by the host or statically, inferred for a particular kernel instance. The device memory consists of four address spaces which map to different memory regions. In OpenCL, every pointer to a memory object can only point to the memory region of its assigned address space. The global memory is accessible by all workitems in all workgroups. Workitems can read or write to any element of a memory object in the global memory. The constant memory is a region of global memory that remains constant during the execution of a kernel. This memory is initialized by the host. The local memory contains objects which are shared by all workitems among a workgroup. Each workgroup has it's own local memory region. The private memory contains variables which are private to a workitem, i.e. they are not visible to other workitems.

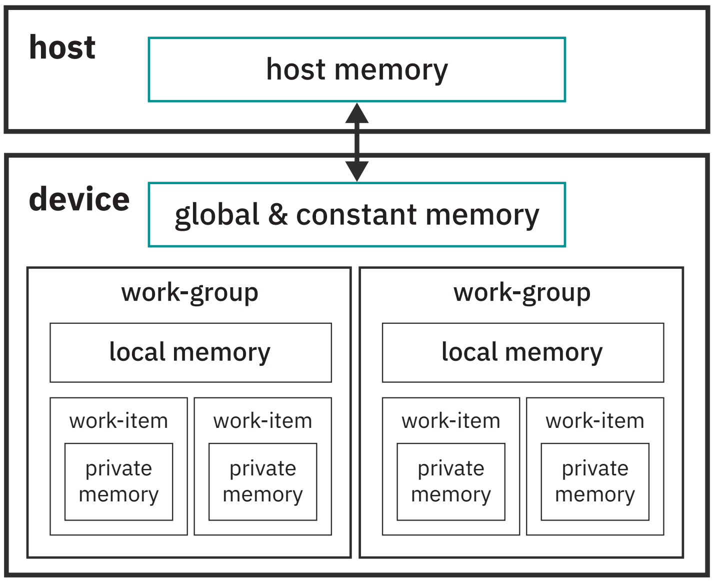

Figure 5-2: The OpenCL memory model.

With OpenCL 2.0, the generic address space was introduced, consisting of the global, local and private memory region. This way, it is possible to have pointers without knowing the memory region of the memory objects they are pointing to.

Other additional features of OpenCL 2.0 are the Shared Virtual Memory (SVM), device-side enqueueing and subgroups. SVM logically extends the global memory into the host address space, giving workitems access to the host address space. Device-side enqueueing is the support of nested parallelism by launching child kernels directly from inside a kernel running on the device. And subgroups are an additional decomposition of workgroups. The size of a subgroup is device-dependent and should fit naturally the vector size of the hardware implementation to allow for data sharing without the need to transfer the data to the local memory region.

5.4 Mapping between OpenMP and OpenCL

Due to the flexibility of the implementation of the OpenMP specification, there exist different possible ways to map the OpenMP execution model to the OpenCL execution model. Regarding the host execution, OpenCL doesn't provide an extra execution model but a library API. On the host side, we leave the OpenMP model out of consideration as this is handled by existing implementations and focus on the target offloading features. When a target construct is encountered, variables are to be mapped to the device data environment and the construct is to be executed on the device. The corresponding OpenCL components are the necessary function calls to initialize the device, manage the device memory and compile and enqueue the kernel.

Inside the target construct, OpenMP defines the teams, the parallel and the SIMD construct, offering different nuances of parallelism. For a feasible mapping from an OpenMP construct to OpenCL, we have to consider the amount of synchronization that is necessary to ensure the correctness of the program. The teams construct offers the highest degree of independence making the least amount of synchronization necessary. Exactly like the OpenCL workgroups, OpenMP teams run independently without any implied ordering and can only be synchronized by global synchronization objects like locks. A plausible mapping of a team is therefor a workgroup with different teams mapping to different workgroups. It would also be possible to map multiple teams to a single workgroup, but as this lowers the grade of parallelism possible within a team, we decided to stick to a one-to-one mapping.

The parallel construct offers a broader range of synchronization possibilities like barriers as well as the sharing of variables. This matches well with the parallelism of workitems within a workgroup. OpenMP lets the implementation specify a maximum number of threads per team, which in our case corresponds to the workgroup size. The OpenMP SIMD construct gives the programmer the ability to make use of SIMD units to parallelize for-loops. One apparent mapping of the SIMD construct seems to be the OpenCL subgroups. However, as OpenCL allows subgroups to make independent forward progress we do without support for the SIMD construct in our implementation because of the differences in the semantic meaning. Nested parallelism may or may not be supported by an implementation. Although it would be possible to support nested parallelism by device-side enqueueing of a kernel, this part is left out as future work.

Regarding the memory model, OpenCL differs a lot from OpenMP. As OpenMP is based upon the existing sequential programming languages C/C++ and Fortran, the memory model consists of a flat address space with additional data sharing rules. We try to map the OpenMP data sharing rules to the different OpenCL memory regions according to their visibility to each thread.

Variables, which are shared by different teams, and thus must be visible to all workitems of the different workgroups, shall be placed in the global memory. Variables shared by threads within a team must be visible to all workitems within the workgroup and shall therefor be placed in the local memory. Other variables are private to each thread and can be placed in the private memory of a workitem.

OpenMP specifies library runtime routines to be available in the target constructs, and thus on the device. OpenCL provides the possibility to implement library functions in OpenCL C and compile them into a SPIR-V library. The SPIR-V library can then be linked at runtime with the kernel program. However, as OpenMP programs rarely use runtime routines, they will not be within the scope of this initial implementation.

5.5 Design Conclusion

With the design decisions to use OpenCL 2.1 with SPIR-V kernels as the target architecture and Clang together with it's OpenMP runtime for the implementation, we are able to propose the final design of this framework. For the source code compilation of OpenMP programs, Clang is used as the compiler frontend where the compilation for the host-side code is done by the existing implementation and the device-side code generation has to be added. The LLVM backend is used to generate the SPIR-V kernels, which then have to be combined with the host code into a fat binary.

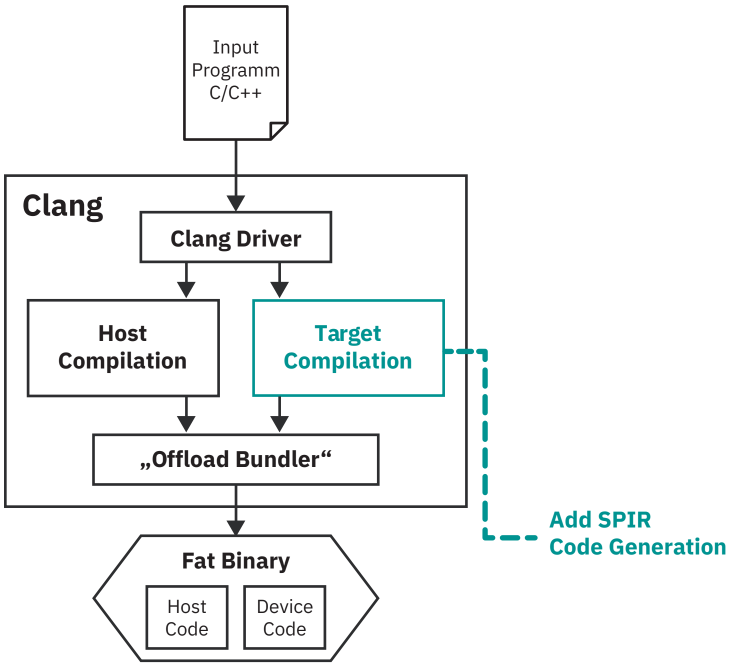

Figure 5-3: Proposed compiler framework extending Clang.

The host code contains the calls to the OpenMP runtime. The existing OpenMP runtime will take the necessary actions to handle the OpenMP execution on the host-side. For the encountering of a target construct, the runtime has to be extended to make the appropriate calls to the OpenCL API for target offloading and could potentially also link the OpenMP library routines with the kernel.

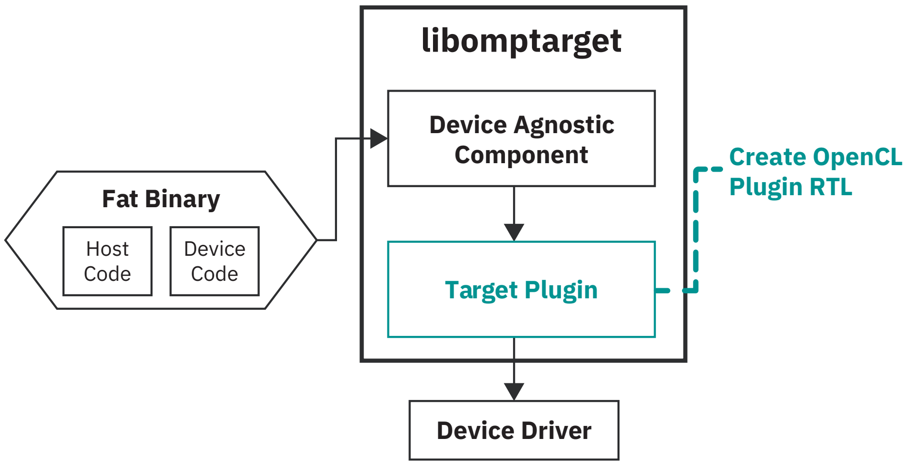

Figure 5-4: The extended OpenMP runtime interacts with the device driver to offload the SPIR-V kernels.

Although this implementation will not cover the whole OpenMP specification, the idea is to make it possible to compile and run existing, valid OpenMP programs on any device with support of OpenCL 2.1 without the necessity of changing the code while keeping the correctness of the program.

In the next chapter, the implementation is described in detail, covering the compiler and the runtime and the proposed implementation for the OpenMP device runtime routines.

6. Implementation

The LLVM Project is a compiler infrastructure which provides a collection of modular and reusable tools [39]. The provided technologies are designed for various optimizations throughout the compilation process. LLVM consists of different sub-projects and aims to offer a complete replacement of the GCC toolchain. The sub-projects used for the implementation of this work are Clang, the LLVM core libraries and the OpenMP runtime. Other sub-projects like the built-in linker lld might be used indirectly for the host compilation, but are not modified and could possibly be replaced by a different tool. In this chapter, the different stages of the compilation process and in each stage the necessary changes to support OpenMP accelerator offloading to OpenCL devices are presented.

The first section is about the compiler frontend and discusses all the steps from source code until the LLVM IR. The second section refers to the compiler backend, which takes the LLVM IR as input and outputs the binary object file. The final section covers the OpenMP shared library which is responsible for target offloading at runtime.

6.1 Compiler Frontend

Clang is a C/C++ frontend for the LLVM Project and is responsible for the invocation of the different tools required for the compilation and for the translation from source code to LLVM IR. For this purpose, Clang consists of two parts, the Clang Driver and the Clang Compiler Front End (Clang CFE) which uses the LLVM core libraries as its backend.

6.1.1 Clang Driver

The Clang Driver is a small program which gets invoked when Clang is executed. The driver controls the overall execution of all tools involved in the compilation process. For this purpose, the driver parses the command line arguments of the compiler invocation and executes the tools from one or more selected toolchains for the processing. Relevant compiler options, set by compiler flags via command line, are forwarded to the respective tools. In the Clang Driver, a toolchain consists in general of three parts. First, a compiler has to be chosen to translate the source code to target-specific assembly code. The assembler is responsible to convert the assembly code into target-specific machine code object files. Finally, the linker combines multiple object files into a binary object which can be either a shared library or an executable.

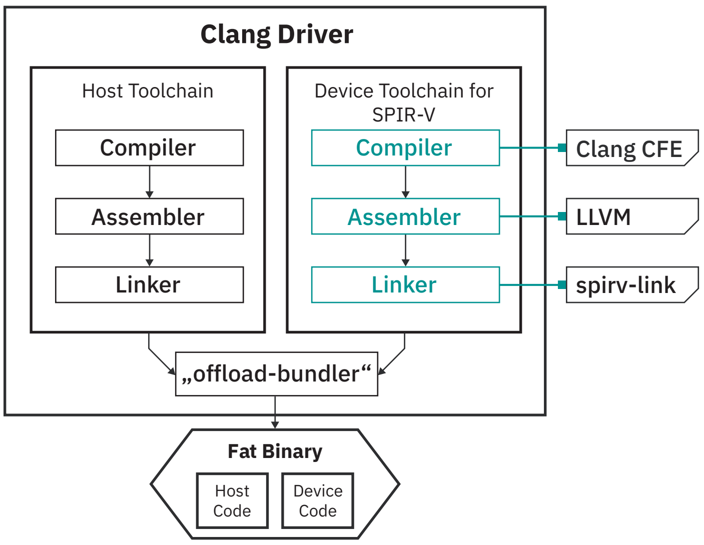

Figure 6-1: The Clang Driver selects a toolchain for the host and each target, and controls the overall execution.

The use of OpenMP can be indicated to the compiler with the “-fopenmp” flag and offloading target architectures can be specified by “-fopenmp-targets=<target, ..>”. With these options set, the driver selects a toolchain for the host code compilation and one toolchain for each specified target architecture. The input source files are then compiled for the host and for each target by their respective toolchain. Afterwards, an “offload-bundler” tool is used to combine the emitted binary objects into a fat binary code file.

In order to provide support for the SPIR-V target, we created a new toolchain to be selected if the specified target is “spir-unknown-unknown” or “spir64-unknown-unknown” (the “unknown” refer to the targeted vendor and operating system). The new SPIR-V toolchain uses Clang CFE as compiler with LLVM providing an integrated assembler. For the initial implementation, cp was chosen to imitate the linking of a single file by copying the file. Unfortunately, multiple files cannot be combined using cp what restricts the implementation to the compilation of single files of source code. In a later version, spirv-link from the Khronos Group’s SPIRV-Tools was selected as linker for the generated SPIR-V modules. This way, it is possible to compile projects consisting of multiple files containing source code.

6.1.2 Clang CFE: AST Generation

The compiler frontend is responsible for the translation from source code to an intermediate representation. To finally emit LLVM IR, Clang CFE processes the code through the main stages as shown in figure 6-2. The behavior of these stages may be modified by command line options given by the driver. This includes e.g. the enabling of OpenMP pragmas or target dependent code emission.

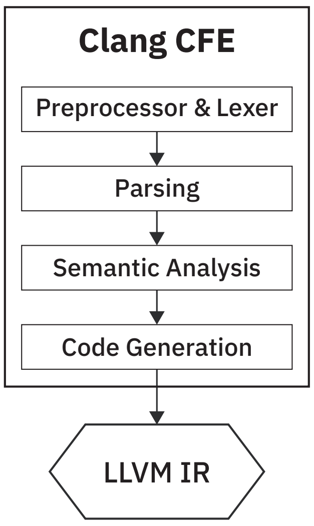

Figure 6-2: Stages of compilation in the Clang front-end.

Preprocessing and lexing stage handle the tokenization of the input source file. The preprocessor expands macros and `#include` statements and subsequently, the lexer reads the character sequence of the source file and generates a sequence of tokens. Preprocessor and lexer don't have to be aware of the OpenMP language feature but only of pragmas in general.

In the parsing stage, the sequence of tokens gets read in and transformed into structured data objects which are then handled by the semantic analysis stage. The parser also performs a syntactic analysis of the input sequence with respect to a given formal grammar of the source code language. This means that the parser has to have knowledge about the syntax of the language. For OpenMP, this includes the OpenMP language extensions or pragmas. As Clang supports OpenMP accelerator offloading to the Intel MIC architecture, the parser already contains the necessary rules to read in OpenMP programs.

The abstract syntax tree (AST) is a tree-based representation of the program, which is produced from the next stage of the compiler pipeline. The semantic analysis stage controls the adherence of the program to the semantic rules of the source language and determines weather the code is well-formed. In this stage, most of the compiler warnings and errors are generated. The semantic analysis includes the checking of data typing rules and, with regard to OpenMP, also of the data sharing rules and structural restrictions of the OpenMP language extensions. For that purpose, the semantic analysis computes information like the data types of expressions or implicitly given attributes. Then, the AST is constructed, and during that process, annotated with this formentioned information. The AST is still completely target independent and had, therefore, not to be changed for this implementation. When compiling an OpenMP program, the AST contains a so-called AST-node for each occurring OpenMP construct. The AST-node contains information like the construct's kind, explicit and implicit attributes, and the associated code block.

6.1.3 Clang CFE: LLVM IR Generation

The code generation stage is the last stage of the frontend and transforms the AST into LLVM IR which eventually gets lowered to assembly or machine instructions from the backend. Although the LLVM IR is designed to be a target independent intermediate representation of the program, differences in the target architectures sometimes require changes to the control flow or the use of built-in functions. This may cause the necessity to have a target dependent code generation. The main part of this implementation is done in the code generation stage to emit appropriate LLVM IR code which afterwards gets transformed into SPIR-V kernels.

As already mentioned, the AST is a tree-based representation of the program while LLVM is based on single static assignment statements. For this transformation, the code generation stage contains functions matching the AST-nodes. The actual LLVM code generation is done via the IRBuilder library which is part of the LLVM core libraries and responsible to ensure a valid single static assignment and to connect the statements with each other. The code generation of the OpenMP language extensions is split into two parts. In the CGStmtOpenMP class, the target independent part is done while the code generation targeting the Intel MIC architecture is done in the CGOpenMPRuntime class and consists of different functions matching the different OpenMP pragmas. As the Intel MIC architecture is binary compatible to x86_64 architectures, its implementation is also referred to as the *generic implementation*. For the SPIR-V target, we created a subclass from the CGOpenMPRuntime class to reuse the existing code generation whenever suitable. In the CodeGenModule, the correct class for the code generation gets selected. The generic OpenMP implementation in Clang is a work-in-progress. For that reason, and also due to the complexity of this work, the scope of this implementation is limited to a subset of the OpenMP specification. Not included are, among other things, the `declare target` pragma to make functions available on target devices and the `reduction` clause.

Setting Metadata and Name Mangling

Although the source code of an OpenMP program is written either in C or C++, SPIR-V for OpenCL is specified to be generated from OpenCL C/C++ source code only. To comply with the specification, metadata about the used OpenCL C version being 2.0 is generated on class instantiation. This information from the metadata is later added to the SPIR-V header by the LLVM backend.

OpenCL provides a number of built-in functions. These are functions to obtain information like the thread or workgroup index, the total number of threads, to emit barriers, and math functions. Because many of these functions are overloaded, i.e. they can be applied to arguments of different types, OpenCL uses the Itanium ABI standard for C++ name mangling for all built-in functions. Name mangling means to incorporate the function type into the function's name to distinguish between overloaded functions. As C does not support overloading of functions but uses a global name space, name mangling is generally not performed. Many OpenMP programs, however, contain computationally intensive numeric procedures, and thus, make use of specialized math functions. In order to provide these functions to programmers of OpenMP programs written in C, we enabled name mangling on all external C-functions. This way, the backend recognizes the functions as built-in and emits the corresponding SPIR-V operation.

Target and Teams Constructs

When a target construct is encountered in the AST representation of the program, the generic implementation creates a kernel function, which later can be offloaded by the host to an accelerator. The teams construct is handled by the generic implementation in a similar way by creating an outlined function which is used to fork multiple threads. Because in OpenCL all threads and workgroups are created at the kernel launch, instead of forking the team's function we inline it. This allows for the reuse of the generic implementation. For both constructs, if not combined with, we do not know weather there is a closely nested parallel region. Therefore, it might be possible that the associated code region must be executed by one thread of the team only. To ensure the serial execution, an if-branch is emitted which gets executed by the thread with local thread index 0.

The variables which are mapped from the host to the device are visible to all threads in all teams on the device. Therefore, they are placed by the host in the global memory of the device. The initial idea was to change the target function's parameters to point to the global address space and use the private address space for all variables including pointers declared inside the target region. A problem arose when a pointer from the parameter list was assigned to a variable declared inside the target region as shown in figure 6-3.

```
void foo(float* x, int offset) {
    #pragma omp target teams map(tofrom:x)
    float* y;        // We don't know beforehand where y will point to.
    y = &x[offset];  // Now, y points to the global memory.
    ...
}
```

Figure 6-3: Pointer aliasing in C/C++.

As C/C++ allows for arbitrary pointer aliasing, i.e. any two pointers can point to the same memory region, it is in general not possible to determine the memory region a pointer will point to. Thus, the correct address space is unknown beforehand and could also possibly change during the variable's life time. The solution to this problem is to make use of the generic address space which was introduced with OpenCL 2.0. By setting the default address space to generic, any pointer may point to the global, local or private memory region.

Parallel Construct

As discussed in Section 5.4, variables declared inside the target region but outside of a parallel region should be allocated in the local memory region as they are shared by default. While we could change the default address space to generic, this has no influence on the memory region where variables are allocated but only onto which memory region a pointer must point to. In Clang, declared variables are allocated using the alloca instruction. With the transformation from LLVM to SPIR-V, all declarations using the alloca instruction are placed in the private memory. To allocate a variable in the local memory, a variable has to be declared as global variable in the local address space. Therefore, to ensure the correct behavior of shared variables, they must be copied to the local memory before entering a parallel region. The generic code generation for parallel regions creates an outlined function where referenced variables are provided via parameter. For the SPIR-V target, we create a global variable in the local address space for each implicitly or explicitly shared variable. We copy the variables' values to the new variables in the local memory and provide the addresses as arguments to the outlined function. The outlined function is inlined as there is no need to fork threads on OpenCL devices. When returning from the parallel region the values of the shared variables are copied back to the private memory.

Before the parallel function is called, any open branch for serial execution has to be closed. This way, the emitted function is executed in parallel by all threads of the workgroup on the OpenCL device.

Loop Distribution

In OpenMP, loops can be distributed among threads using the loop construct or distributed among teams using the distribute construct. In both cases, the generic implementation creates iteration variables which are then used in each thread to calculate the loop iterations to be worked on. While the default scheduling is implementation defined, the user can further specify a preferred scheduling of the loop iterations using the `schedule` clause. For CPU architectures, due to the data caching, a thread typically works on a chunk of neighboring loop iterations. On GPUs, on the other hand, it is preferably if neighboring loop iterations are worked on by neighboring threads, whereby loads and stores can be vectorized. Although our implementation defaults to the latter scheduling scheme, the other possibilities are supported as well.

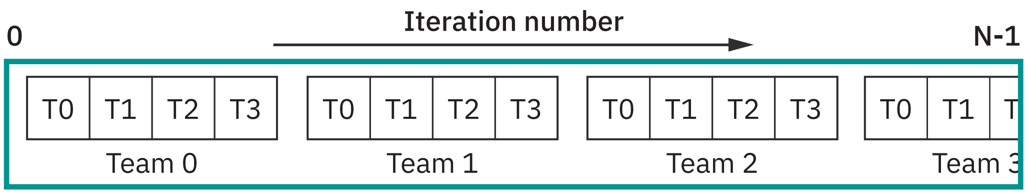

Figure 6-4: The used default scheduling scheme assigns one loop iteration to each thread.

For this purpose, the values of the different iteration variables are calculated using the OpenCL built-in functions to obtain the index and total number of threads or workgroups. If no scheduling scheme is specified, the chunk size for one thread is one while for one team it is the size of the workgroup. The stride, i.e. the number of iterations to be skipped before selecting the next one, is the number of threads per workgroup or the number of workgroups, respectively. Hence, all threads running on the OpenCL device can work concurrently on neighboring loops from the iteration space.

6.2 Compiler Backend

The compilation steps translating the LLVM IR to the final binary are referred to as the compiler backend. These involve the optimizer, assembler and linker for the host compilation and each specified target architecture as well as the “offloading-bundler”. The optimizer is an integrated part of LLVM and can perform various modifications to the LLVM representation of the program in order to increase the performance or decrease the size of the compiled program. Predefined optimization passes can be selected using the -Os (for smaller binary size) or -O0 to -O3 (for better performance) compiler flags. The LLVM IR provided in SSA form by the code generation step is very unoptimized and makes heavy use of load and store instructions. To promote memory references, i.e. alloca instructions which have only loads and stores as their uses, to register references using phi-nodes LLVM performs a transformation called mem2reg pass. However, regarding the SPIR-V target, any optimizations including the mem2reg pass could not yet be enabled. Thus, the emitted SPIR-V code is very cumbersome and inefficient and relies on optimizations performed by the OpenCL runtime.

Clang CFE uses the LLVM integrated assembler to generate target-specific object code. As SPIR-V is not yet officially supported by LLVM we use a fork of LLVM by Nicholas Wilson for our implementation. This fork includes the LLVM to SPIR-V transformation tool from the Khronos Group and makes SPIR-V available as target architecture. The SPIR-V backend is also responsible to replace the mangled built-in function calls by their corresponding SPIR-V operations.

After the object code has been emitted, the code from potentially multiple files gets combined by the linker. The linker is an external tool selected by the toolchain. For the host code and most other LLVM targets, LLD, the linker from the LLVM project, is used. As LLD doesn't support SPIR-V, we use *spirv-link* from the Khronos Group's SPIR-V Tools. To make *spirv-link* work in the expected way, two minor bugs had to be fixed and were successfully committed to the upstream project. When compiling OpenMP projects with accelerator offloading, the compiler backend performs an additional last step in the compilation process. The "offloading-bundler" is responsible to combine the binary files of the different architectures into a single fat binary, i.e. an executable or shared library consisting of the host code as well as the code for every specified target architecture.

6.3 OpenMP runtime

The OpenMP subproject of LLVM consists of the *libomp* runtime library against which code compiled with clang "-fopenmp" is linked as well as the *libomptarget* library which supports offloading to target devices [40]. The OpenMP runtime is responsible for the thread creation, task assignment, the evaluation of environment variables, and the offloading to target devices, among other things. For this purpose, the runtime implements functions which are inserted by the compiler as well as the runtime routines which are directly callable by user code, i.e. the OpenMP library routines.

The *libomptarget* library is further split into a target independent part and plugins for each target architecture. The target independent part is responsible to encounter and select appropriate target devices and the corresponding kernel images. It also provides a thin abstraction layer between the OpenMP action taking place and the low level functionality provided by the target plugin. In case of an offloading failure, fall-back code will be executed on the host. In order to support offloading to SPIR-V capable OpenCL devices, a plugin for the "spir64-unknown-unknown" target was implemented using the OpenCL API. The plugin consists of a set of functions for device initialization, memory management, kernel compilation and enqueueing. At class instantiation, the plugin searches for available OpenCL runtimes and devices. For the current implementation, only the AMDGPU-Pro OpenCL runtime is accepted. However, with the development of more compliant OpenCL 2.1 ICDs, more options might become available in the future. For the device initialization, an OpenCL context and a command queue is created. When loading a program binary object, it is compiled from SPIR-V to the vendor specific instruction set architecture (ISA) and the kernel objects are created. The memory management functions for data allocation, transfer, and release are implemented using the corresponding OpenCL API functions. Data transfers are currently implemented synchronously. However, it might be possible to increase the performance slightly with an asynchronous implementation. To run a kernel, the parameters are set and, depending on the given thread limit and number of teams, the global and local work sizes are calculated. The kernel gets enqueued and waited for to finish. An asynchronous kernel execution is still possible as OpenMP creates a helper thread in this case.

The plugin should also contain the OpenMP runtime routines callable from the user code. However, these runtime routines are not implemented as part of this work. As these runtime routines must be callable from the target code, they have to be available as SPIR-V functions and must be linked to the user code. One way to achieve this is to implement these routines in OpenCL C and compile them to SPIR-V. As part of the plugin, they could be linked at runtime when compiling a kernel. With the availability of the `declare target` pragma, an alternative way would be to write the runtime routines as an OpenMP program and link them at program compilation.

7. Experiment Results

In this chapter, two benchmarks are presented to evaluate our implementation using OpenMP accelerator offloading on OpenCL devices. The first benchmark is the calculation of the Mandelbrot set and suitable to show the potential benefits for workloads with high data parallelism. The second benchmark is a simplified but typical application to model hydrodynamics and can be seen as a real world example. The benchmarks are measured using the following system components:

- AMD Ryzen 1700
- Radeon RX 560
- Asrock B350-Pro4
- 16 GB DDR4-2400
- Ubuntu 16.04
- AMDGPU-Pro OpenCL Driver 17.10

7.1 Mandelbrot Set

The Mandelbrot set is a fractal named after the mathematician Benoit Mandelbrot and defined as the set of complex numbers c for which the sequence $z_{n+1} = z_n^2 + c$ does not diverge, i.e. when iterated from $z_0 = 0$, the value of zn remains bounded. Using the real and imaginary parts of c as image coordinates, the Mandelbrot set can be visualized as a colored image. Outside of mathematics, due to the aesthetic appeal when visualized, the Mandelbrot set has become popular in art and culture. For this benchmark, we use a modified version of the Mandelbrot implementation written to measure the Intel SPMD program compiler (ispc) [41]. The serial CPU implementation was inlined and provided with OpenMP directives to enable accelerator offloading.

```
void mandelbrot_omp(float x0, float y0, float x1, float y1,
                    int width, int height, int maxIterations,
                    int output[])
{
   float dx = (x1 x0) / width;
   float dy = (y1 y0) / height;
   #pragma omp target teams map(from:output[0:width*height]) thread_limit(width*height)
   #pragma omp distribute parallel for collapse(2)
   for (int j = 0; j < height; j++) {
        for (int i = 0; i < width; ++i) {
                float x = x0 + i * dx;
                float y = y0 + j * dy;
                int index = (j * width + i);
                float c_re, c_im;
                float z_re = c_re =x, z_im = c_im = y; int count = maxIterations;
                int k;
                for (k = 0; k < count; ++k) {
                        if (z_re * z_re + z_im * z_im > 4.f)
                                break;
                        float new_re = z_re*z_re z_im*z_im;
                        float new_im = 2.f * z_re * z_im;
                        z_re = c_re + new_re;
                        z_im = c_im + new_im;
                }
                output[index] = k;
        }
   }
}
```

Figure 7-1: The complete Mandelbrot kernel. Only two additional lines of code are necessary for accelerator offloading.

The OpenCL version of this benchmark is written using the same algorithm as the serial implementation. The parameter space of the complex plane is divided into 3860 x 2160 pixels. This benchmark performs up to 100 iterations of the Mandelbrot function independently for each number in the iteration space to decide whether it is part of the Mandelbrot set. For each platform, the benchmark is run five times. We measure every run and choose the best execution time to complete this calculation. The result is drawn as black and white image.

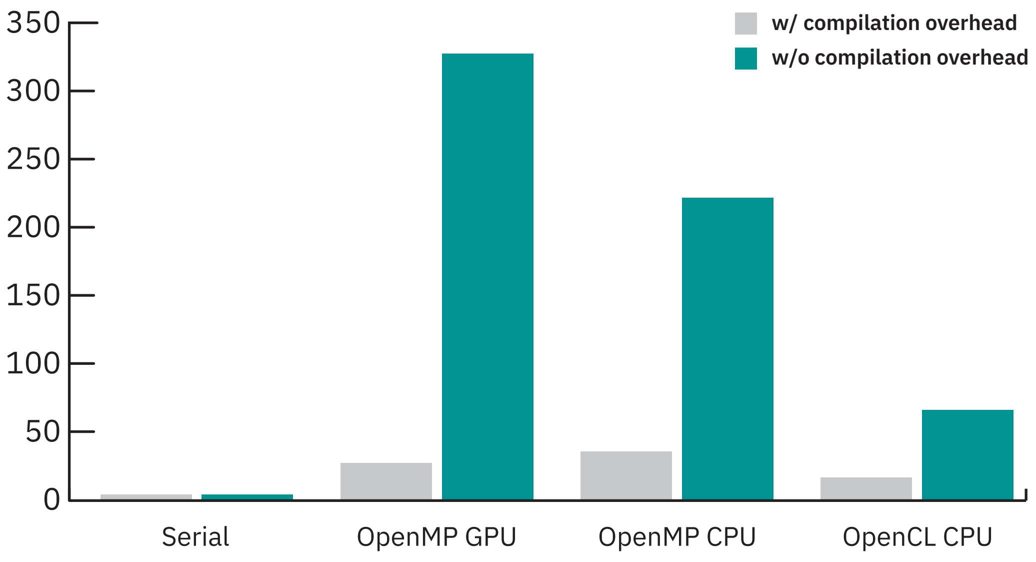

Figure 7-2: The speed-ups of the tested programs compared to the serial implementation, each with compilation overhead and without.

The compilation overhead refers to the setup of the OpenCL context, the command queue and kernel compilation. The Mandelbrot set is a well known example of data parallelism, and thus, shows a high speed-up of 327.8 when using OpenMP accelerator offloading to a GPU without compilation overhead compared to the single-threaded execution on a CPU. Surprisingly, the hand-written OpenCL benchmark with a speed-up of 222.3 performs worse than the OpenMP implementation. One possible reason for this behavior might be the OpenCL compiler used by the OpenCL runtime. When including the compilation overhead, e.g. if the benchmark is only run once, the speed-up is 27.6 compared to 36 when running the OpenCL GPU implementation. The longer compilation time of the SPIR-V kernel might be due to the unoptimized code generated from our compiler implementation. Compared to the serial version, the OpenCL CPU benchmark shows a speed-up of 17.1 and 66.7 when measured with or without compilation overhead, respectively. The high speed-up results from the multiple threads and vector units used by the OpenCL CPU runtime.

7.2 LULESH

The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) benchmark is developed at the Lawrence Livermore National Lab and models hydrodynamics to represent a typical scientific application [42]. Hydrodynamics describe the study of liquids in motion when subject to forces and has a wide range of applications including pipe flow, ship hull design and weather prediction, among others. LULESH is simplified to solve a Sedov blast problem with analytic answers and represents a typical hydrocode, i.e. it models the behavior of fluid flow systems by breaking down the system into a three-dimensional matrix of cells and calculating properties such as force for each of them. Implementations exist for a range of different programming models, including CUDA, OpenCL, OpenMP, MPI and serial implementations in different programming languages [43].

For benchmarking OpenMP accelerator offloading to OpenCL devices, the OpenMP version of LULESH was slightly modified by inlining all target functions. The reason for this is the absence of the declare target compiler directive and thus a necessity for successful compilation. Due to the low overhead of uniform function calls and the behavior of device compilers to aggressively inline functions anyway, the modification should not impact the expected performance. For comparison, we use the CPU implementation with and without OpenMP for multithreading and the OpenCL implementation.

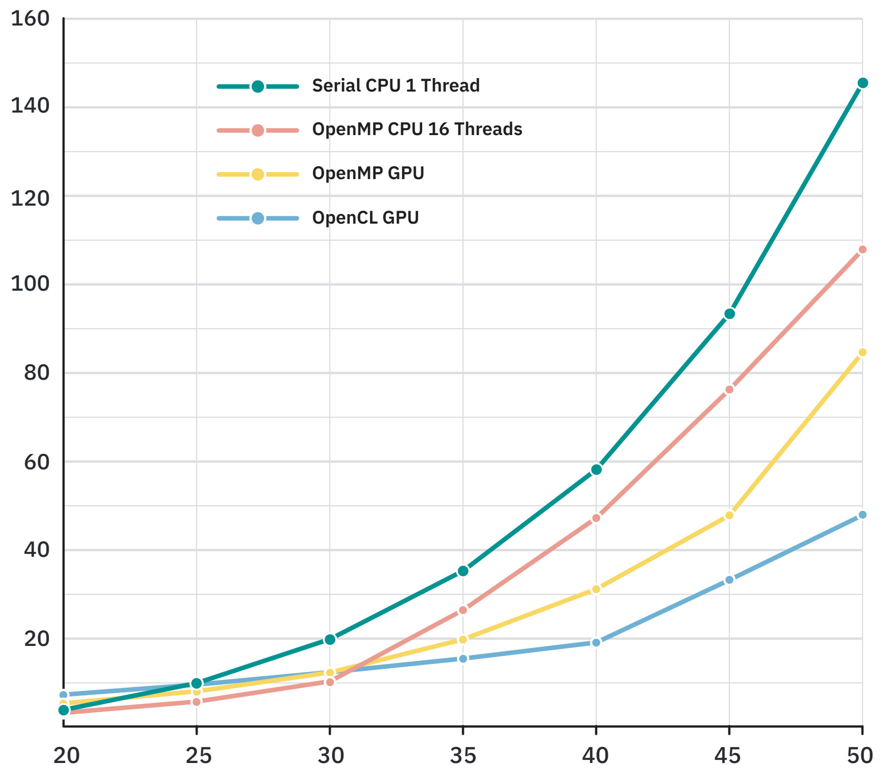

Figure 7-3: Execution times in seconds to complete the LULESH benchmark for different problem sizes.

We executed the benchmark five times for each target and problem size, and chose the best result. The results indicate that the OpenMP implementation of LULESH does not scale very well with the number of CPU threads as the execution times for the multi-threaded benchmark are hardly lower than the serial ones. The OpenMP benchmark using accelerator offloading to GPUs has shorter execution times for larger problem sizes than the CPU benchmarks. A similar behavior can be observed for the OpenCL benchmark which has the shortest execution times for larger problem sizes. The reason for the CPU benchmarks being faster for small problem sizes is that no compilation and data transfer is necessary.

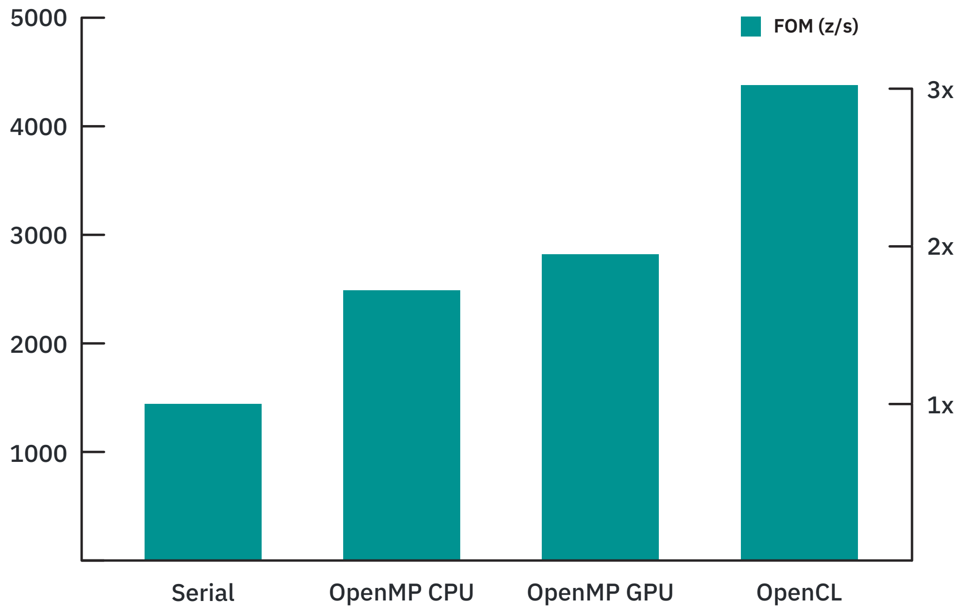

Figure 7-4: Maximum “Figure of Merit”, i.e. the number of elements solved per microsecond, reached by each benchmarked implementation, and the respective speed-ups.

LULESH uses double precision floating point data types for the calculations. As the Radeon RX 560 has a very low performance of only 1 / 16 for double precision versus single precision floating point operations the expected performance increase is lower to the same extend. The OpenCL results show potential improvements for accelerator offloading.

Nikolay Sakharnykh, Senior Developer Technology Engineer at NVIDIA, used the PGI Compiler with OpenACC to benchmark LULESH on an NVIDIA P100 and NVIDIA K40 GPU with Unified Memory and measured 30k FOM and 11k FOM, respectively [44]. However, the used graphics cards have a much higher theoretical peak performance for double precision floating point operations. The Nvidia P100 and K40 have 4.7 TFLOPS and 1.4 TFLOPS double precision peak performance, while the Radeon RX 560 is limited to 160 GFLOPS.

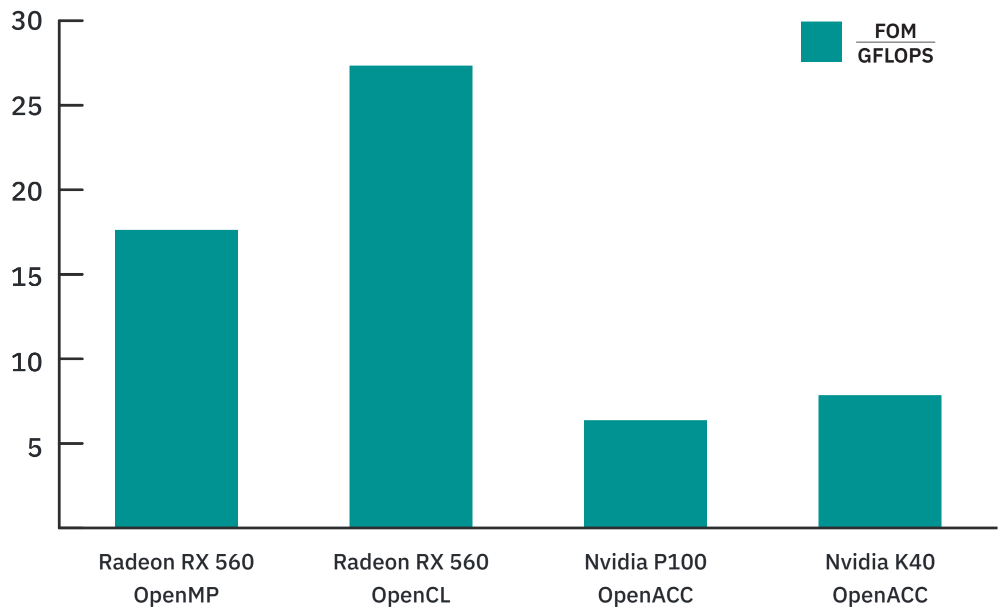

Figure 7-5: Comparison of the number of elements solved per GFLOP.

By comparing the measured application performance to the theoretical peak performance, we can observe a high efficiency of our OpenMP implementation using accelerator offloading. Note, however, that these results cannot be directly compared due to limitations described by Amdahl's law.

8. Conclusion

In this work, we proposed an extension for the Clang compiler and a runtime to offload OpenMP programs to OpenCL using SPIR-V kernels. This implementation may help to ease the parallel programming of GPUs by using OpenMP directives. We were able to generate SPIR-V kernels from OpenMP target regions and to offload these kernels at runtime using OpenCL. By benchmarking two parallelized programs, we could demonstrate the functionality and efficiency of this approach. While the Mandelbrot benchmark required only two additional lines of code to achieve a speed-up of 327.8 compared to the serial implementation, the LULESH benchmark showed results comparable to an OpenACC implementation. Moreover, the obtained performance is similar to the manually translated OpenCL programs.

Compared to other OpenMP implementations, binary portability was achieved by targeting OpenCL. As for now, the lack of conformant device drivers with support for OpenCL 2.1 still limits the adaptability. However, in future we hope for the development of more devices with OpenCL 2.1 support.

While the benchmarks have demonstrated the potential performance and efficiency of the approach of OpenMP accelerator offloading using OpenCL with SPIR-V kernels, some opportunities for extending the scope of this work remain. Many existing OpenMP programs make use of OpenMP features which are not implemented in this work, and therefore, could not be compiled and executed. With respect to the compiler directives, the main missing features are the `reduction` clause and the `declare target` pragma. While the `declare target` pragma is not yet supported by the Clang CFE, the `reduction` clause was omitted for this work for time reasons due to time constraints and the high expected effort necessary to implement this feature.

Other parts of the OpenMP specification missing in this work are the OpenMP runtime routines and evaluation of environment variables. In Section 6.3, we proposed two different ways for the implementation, either as OpenCL C functions compiled to SPIR-V or as OpenMP target functions. Other possible concepts to be considered for future work involve dynamic parallelism from OpenCL 2.0 to be used for OpenMP nested parallel constructs and the use of shared virtual memory to avoid unnecessary data copies on supporting devices.

One of the major deficiencies of this implementation compared to other implementations are the missing backend optimizations. The enabling of a set of optimizations, such as the previously mentioned mem2reg pass, but also control flow and loop optimizations, as well as constant folding and propagation, is highly desired.

Finally, in the event that SPIR-V support in the LLVM backend was to be included in the official LLVM project, a general availability of this work in the Clang compiler should be considered.

References

[1] The LLVM Foundation. “Clang: a C language family frontend for LLVM.” *The LLVM Compiler Infrastructure Project*, clang.llvm.org/.

[2] Owens, John D., et al. “A Survey of General-Purpose Computation on Graphics Hardware.” *Computer Graphics Forum*, vol. 26, no. 1, 2007, pp. 80–113.

[3] Buck, Ian, et al. “Brook for GPUs: stream computing on graphics hardware.” *ACM Transactions on Graphics (TOG)*. vol. 23. no. 3. ACM, 2004.

[4] Tarditi, David, et al. “Accelerator: using data parallelism to program GPUs for general-purpose uses.” *ACM SIGARCH Computer Architecture News*. vol. 34. no. 5. ACM, 2006.

[5] Darema, Frederica. “The SPMD model: Past, present and future.” *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer, Berlin, Heidelberg, 2001.

[6] ATI. “CTM Guide - CTI Technical Reference Manual.” AMD, 2006. *Internet Archive*, web.archive.org/web/20070222162035/http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf.

[7] NVIDIA. “CUDA Toolkit Documentation v9.1.85.” *NVIDIA Developer Documentation*, 19 Dec. 2017, docs.nvidia.com/cuda/.

[8] Khronos OpenCL Working Group. “The OpenCL Specification Version 2.2.” *Khronos OpenCL Registry*, The Khronos Group Inc., 12 May 2017, www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf.

[9] Khronos OpenCL Working Group. “The OpenCL C Specification Version 2.0.” *Khronos OpenCL Registry*, The Khronos Group Inc., 13 Apr. 2016, www.khronos.org/registry/OpenCL/specs/opencl-2.0-openclc.pdf.

[10] Kessenich, John, et al. “SPIR-V Specification Version 1.2, Revision 4, Unified.” *Khronos SPIR-V Registry*, The Khronos Group Inc., 17 Jan. 2018, www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf.

[11] “Khronos Releases OpenCL 2.2 With SPIR-V 1.2.” The Khronos Group, 16 May 2017, www.khronos.org/news/press/khronos-releases-opencl-2.2-with-spir-v-1.2.

[12] Microsoft. “Parallel Programming Using C++ AMP, PPL and Agents Libraries.” *Parallel Programming in Native Code*, blogs.msdn.microsoft.com/nativeconcurrency/.

[13] Richards, Andrew. “Update on the SYCL for OpenCL open standard to enable C++ meta programming on top of OpenCL.” *Proceedings of the 3rd International Workshop on OpenCL*. ACM, 2015.

[14] AMD. “HCC is an Open Source, Optimizing C Compiler for Heterogeneous Compute.” *GitHub*, github.com/RadeonOpenCompute/hcc/wiki.

[15] Dolbeau, Romain, et al. “HMPP: A hybrid multi-core parallel programming environment.” *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*. vol. 28. 2007.

[16] OpenACC. “The OpenACC Application Programming Interface Version 2.6.” *OpenACC-Standard.org*, Nov. 2017, www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf.

[17] OpenMP ARB. “OpenMP Application Programming Interface Version 4.5.” *The OpenMP API Specification for Parallel Programming*, OpenMP Architecture Review Board, Nov. 2015, www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[18] NVIDIA. “PGI Accelerator Compilers with OpenACC Directives.” *PGI Compilers & Tools*, www.pgroup.com/resources/accel.htm.

[19] Cray Inc. “Cray Fortran Reference Manual (8.5).” *Cray Technical Documentation*, June 2016, docs.cray.com/PDF/Cray_Fortran_Reference_Manual_85.pdf.

[20] Tabuchi, Akihiro, et al. “A source-to-source OpenACC compiler for CUDA.” *European Conference on Parallel Processing*. Springer, Berlin, Heidelberg, 2013.

[21] Lee, Seyong, and Jeffrey S. Vetter. “OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing.” *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014.

[22] Vanderbruggen, Tristan. “Open-Source OpenACC Compiler for OpenCL Compatible Accelerators.” *Rose ACC*, 11 June 2014, rose-acc.blogspot.de/.

[23] Tian, Xiaonan, et al. “Compiling a high-level directive-based programming model for GPGPUs.” *International Workshop on Languages and Compilers for Parallel Computing*. Springer, Cham, 2013.

[24] GCC. “OpenACC.” *GCC Wiki*, Free Software Foundation, Inc., gcc.gnu.org/wiki/OpenACC.

[25] Peng, Hao-Wei, and Jean Jyh-Jiun Shann. “Translating OpenACC to LLVM IR with SPIR kernels.” *Computer and Information Science (ICIS), 2016 IEEE/ACIS 15th International Conference on*. IEEE, 2016.

[26] IBM. “Offloading Computations to the NVIDIA GPUs.” *IBM Knowledge Center*, www.ibm.com/support/knowledgecenter/SSXVZZ_13.1.6/com.ibm.xlcpp1316.lelinux.doc/proguide/offloading.html.

[27] Newburn, Chris J., et al. “Offload compiler runtime for the Intel® Xeon Phi coprocessor.” *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International.* IEEE, 2013.

[28] Liao, Chunhua, et al. “Early experiences with the OpenMP accelerator model.” *International Workshop on OpenMP*. Springer, Berlin, Heidelberg, 2013.

[29] GCC. “Offloading Support in GCC.” *GCC Wiki*, Free Software Foundation, Inc., gcc.gnu.org/wiki/Offloading.

[30] Monakov, A.v., and V.a. Ivanishin. “Implementing OpenMP 4.0 for the NVIDIA PTX architecture in GCC compiler.” *Proceedings of the Institute for System Programming of the RAS*, vol. 28, no. 4, 2016, pp. 169–182.

[31] Bataev, A. B. A., et al. “Towards OpenMP Support in LLVM.” *2013 European LLVM Conference*. 2013.

[32] Bertolli, Carlo, et al. “Integrating GPU support for OpenMP offloading directives into Clang.” *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 2015.

[33] Sultana, Nawrin, et al. “From OpenACC to OpenMP 4: Toward automatic translation.” *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*. ACM, 2016.

[34] Juckeland, Guido, et al. “From Describing to Prescribing Parallelism: Translating the SPEC ACCEL OpenACC Suite to OpenMP Target Directives.” *International Conference on High Performance Computing. Springer International Publishing*, 2016.

[35] The Khronos Group. *Conformant Companies*. www.khronos.org/conformance/adopters/conformant-companies.

[36] AMD. “Radeon Software for Linux Release Notes.” *Drivers* + Support , 12 Dec. 2017, support.amd.com/en-us/kb-articles/Pages/Radeon-Software-for-Linux-Release-Notes.aspx.

[37] The LLVM Foundation. “LLVM Related Publications.” *The LLVM Compiler Infrastructure Project*, llvm.org/pubs.

[38] Wilson, Nicholas. “[Llvm-Dev] [SPIR-V] SPIR-V in LLVM.” *LLVM Developers Mailing List*, 1 May 2017, lists.llvm.org/pipermail/llvm-dev/2017-May/112538.html.

[39] The LLVM Foundation. “LLVM Overview.” *The LLVM Compiler Infrastructure Project*, llvm.org/.

[40] The LLVM Foundation. “OpenMP®: Support for the OpenMP language.” *The LLVM Compiler Infrastructure Project*, openmp.llvm.org/.

[41] Pharr, Matt, and William R. Mark. “ispc: A SPMD compiler for high-performance CPU programming.” *Innovative Parallel Computing (InPar)*, 2012. IEEE, 2012.

[42] Hornung, R. D., et al. *Hydrodynamics challenge problem*. No. LLNL-TR-490254. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2011.

[43] Karlin, Ian. *Lulesh programming model and performance ports overview*. No. LLNL-TR-608824. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2012.

[44] Sakharnykh, Nikolay. “Beyond GPU Memory Limits with Unified Memory on Pascal.” *NVIDIA Developer Blog*, NVIDIA, 14 Dec. 2016, devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/.