Department of Physics and Astronomy

Heidelberg University

Master thesis

in Computer Engineering

submitted by

Joachim Mathias Meyer

born in Mannheim

2021

Compiler-assisted optimizations for data-parallel paradigms in hipSYCL

This Master thesis has been carried out by Joachim Meyer

at the

Institute of Computer Engineering

under the supervision of

Aksel Alpay, M.Sc. (EMCL)

Prof. Dr. Vincent Heuveline (EMCL)

Prof. Dr. Holger Fröning (CSG)

Acknowledgement

I want to thank Aksel Alpay for allowing me to pursue this compiler technology-focused thesis as part of the hipSYCL project, for his continued confidence in my work and valuable feedback.

Furthermore, I would like to express my appreciation for Prof. Dr. Holger Fröning for providing recurring feedback, incentive, and advice to focus the effort on a few main contributions.

Special thanks go to Simon McIntosh-Smith for enabling comprehensive access to the Isambard 2 HPC resources, allowing this thesis to be evaluated on a diverse range of modern hardware.

For all the LLVM related questions, I would like to thank everyone on the LLVM IRC channel for their time and answers, most notably Roman Lebedev and Johannes Doerfert.

I also want to profoundly thank my friends, especially Sarah Hell and David Prenninger, for their continuous emotional support and for providing confidence throughout the 5 years of my university studies.

Finally, I owe my family deep gratitude for giving me the freedom and support to pursue my studies and especially for the motivational support in the last months of writing this thesis.

Augsburg, Germany Joachim Meyer

Compiler-gestützte Optimierungen für Daten-parallele Paradigmen in hipSYCL:

Die wachsende Verbreitung heterogenen Programmierens macht die weitere Verbesserung der Performance Portabilität unerlässlich. SYCL bietet das nd-range parallel-for Paradigma, um Daten-parallele Kernel zu schreiben. Wie in CUDA und OpenCL Kerneln, erlaubt es die Verwendung von Barrieren für lokale Synchronisation. Auf GPUs kann dies effizient umgesetzt werden, während auf CPUs die erforderlichen Fortschrittsgarantien die Nutzung von vielen (leichtgewichtigen) Threads in Implementierungen, die als reine Bibliothek umgesetzt werden, nötig machen, was inakzeptabel ineffizient ist. Durch die Integration zweier Compiler-basierter Ansätze, die dies beheben, verbessert diese Arbeit die Performance des nd-range parallel-for Paradigmas in hipSYCL für CPUs um mehrere Größenordnungen auf verschiedenen CPU Architekturen. Das bedeutet, dass hipSYCL die erste SYCL Implementierung ist, die dabei gute Leistung auf CPUs liefern kann, auf denen kein OpenCL CPU Treiber zur Verfügung steht. Die beiden Alternativen werden bezüglich ihrer funktionalen Korrektheit und Performance verglichen. Zusätzlich werden Strategien vorgestellt, wie der Daten-Parallelismus effektiv auf SIMD Einheiten abgebildet werden kann. Hierbei wird festgestellt, dass explizite Verwendung von Daten-parallelen Typen und die Vektorisierung innerer Schleifen bereits gute Optimierungen sind, wohingegen die Vektorisierung äußerer Schleifen sub-optimal sein kann. Außerdem wird gezeigt, dass mithilfe erzwungener Vektorisierung mittels des Region Vectorizers, SYCL's sub-group auf die SIMD Ebene abgebildet werden kann.

Compiler-assisted optimizations for data-parallel paradigms in hipSYCL:

With heterogeneous programming continuously on the rise, performance portability is still to be improved. SYCL provides the nd-range parallel-for paradigm for writing data-parallel kernels. This model allows barriers for group-local synchronization, similar to CUDA and OpenCL kernels. GPUs provide efficient means to model this, but on CPUs the necessary forward-progress guarantees require the use of many (lightweight) threads in library-only SYCL implementations, rendering the nd-range parallel-for unacceptably inefficient. By adopting two compiler-based approaches solving this, the present thesis improves the performance of the nd-range parallel-for in hipSYCL for CPUs by up to multiple orders of magnitude on various CPU architectures. Thereby, hipSYCL is the first SYCL implementation to provide a well performing nd-range parallel-for on CPU, even if no OpenCL runtime is available. The two alternatives are compared with regard to their functional correctness and performance. Additionally, strategies for effective mapping of the data-parallelism to the SIMD paradigm are evaluated. It is found that explicitly using data-parallel types and inner-loop vectorization already provides good optimizations, whilst outer-loop vectorization can be sub-optimal. SYCL's sub-groups traditionally model the warps of GPUs. It is shown that the forced vectorization of the Region Vectorizer enables mapping the sub-groups to the SIMD level instead.

Contents

1 Introduction

Parallel and heterogeneous programming have brought the most notable advances in computing over the last decades. One of the major challenges with heterogeneous programming is the performance portability of code on the various computing resources. Open standards like OpenCL or SYCL provide means to program various classes of processing units using the same kernel. While CPUs expose up to 64 physical cores at this point in time, GPUs are usually programmed with thousands of threads. This difference makes it challenging to efficiently use CPUs as fallback or as additional computing resources using the same kernels as with GPUs.

The hipSYCL implementation of the SYCL standard currently provides unified access to CPUs and GPUs but the nd-range parallel-for paradigm is far from efficient on CPUs. This is due to the necessary forward-progress guarantees that have to be made so that barriers can be correctly implemented. This problem is a common issue with most heterogeneous programming models when also targeting CPUs.

This thesis will explore two approaches that were originally developed to solve this issue with OpenCL by applying them to the hipSYCL nd-range parallel-for implementation. Their implementation is presented and their feasibility for production usage is examined. The performance evaluation shows that with certain optimizations both approaches can reach multiple orders of magnitude performance improvements over the current implementation. Those benefits are shown to be independent of the CPU architecture used. With these improvements, hipSYCL is the first SYCL implementation achieving decent nd-range parallel-for performance on CPUs without requiring the availability of an OpenCL CPU runtime.

The programming models for GPUs are traditionally single-program multiple-threads (SPMT) focused. This is also the case for SYCL's nd-range parallel-for. CPUs do not provide efficient realizations of this paradigm but instead excel with single-instruction multiple-data (SIMD) programming. Using auto-vectorizers it is attempted to map the SPMT kernels onto the SIMD model, which is more favorable to CPUs. For this, the LLVM inner-loop vectorizer as well as the outer-loop vectorization capabilities of the Region Vectorizer from Moll and Hack [2018] are considered and experimentally evaluated.

The Region Vectorizer also provides intrinsics that, in conjunction with its forced vectorization, can be used to implement SYCL's sub-group hierarchical level on CPUs. This level is mapped to warps on GPUs. This thesis discusses the mapping of the sub-groups to SIMD units and the necessary improvements to enable performance-beneficial use of this extension.

Finally, SYCL is a single-source programming model, i.e. the host application code as well as the kernel code both reside in the same C++ compilation unit. This

opens the opportunity to utilize information from the surrounding host code inside a kernel. As an example, the propagation of host-side constant work-group sizes into the kernel is discussed. [1]

[1]This work used the Isambard 2 UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1)

2 Background and related work

SYCL is an industry-standard, hosted by the Khronos group. It specifies a DSL inside C++ to enable developing heterogeneous, data-parallel applications with a single-source model. The current version at the time of writing is SYCL 2020 [The Khronos® SYCL™ Working Group et al., 2021]. Multiple implementations of the standard provide access to various accelerators.

While Intel's DPC++, presented in Bader et al. [2019], provides access to their CPUs, GPUs, and FPGAs through OpenCL or Level0, Codeplay enabled the open-source variant of it with support for Nvidia [Reyes, 2020] and recently AMD GPUs [Macfarlane, 2021] through CUDA and HIP runtimes respectively. Codeplay also develops ComputeCpp, an embedded-systems focused SYCL implementation with support for specific OpenCL versions, including Intel and Arm GPUs as well as application-specific processors like Renesas R-Car. Support for RISC-V Vector accelerators is underway [Burns, 2020]. Xilinx works on an experimental SYCL implementation with support for its FPGAs, called triSYCL [Keryell and Yu, 2018].

Alpay and Heuveline [2020] present hipSYCL, a multi-backend SYCL implementation that supports Nvidia, AMD as well as Intel GPUs by reusing the respective Clang toolchains for CUDA, HIP, and DPC++. All CPUs are supported as long as a modern OpenMP capable compiler is available for it.

2.1 SYCL execution model

The SYCL standard defines an interface that enables accessing heterogeneous compute resources. The chosen abstractions are similar to those of the OpenCL and CUDA programming models. A `queue` is used to communicate tasks to the accelerator. `buffer`s are used to allocate host and device memory. By using `accessor`s a dependency on a buffer is created, the memory is moved to the device and kernels can access it. On the device, these buffers are then located in *global* memory. Accessors can be used to allocate *local* and *constant* memory as well.

Local memory is relevant when using hierarchical parallel execution models. These are nd-range and hierarchical parallel-for in the standard. Each id in the execution range, for which the kernel is executed in the parallel-for models, results in a work-item that is executed. Work-items are equivalent to CUDA threads. All variables and arrays defined in the kernel are implicitly in work-item *private* memory. This implies that only the one work-item has access to those values.

For the hierarchical models, the work-items are grouped into equally sized work-groups which in CUDA would be called blocks. On this hierarchical level the local memory can be shared between the contained work-items. Additionally, to make

these potentially parallel memory accesses to local memory well-defined, work-items of a work-group can synchronize with each other at work-group barriers. Note, as with CUDA or OpenCL, it is not possible to synchronize between the work-groups inside a kernel.

A main focus of SYCL is to provide an interface for data-parallel kernels. For this the three main execution models are:

basic parallel-for executes a kernel function in parallel across a plain n-dimensional iteration space, is non-hierarchical

`nd_range` parallel-for allows manually managing the partitioning of the global range into work-groups of parallel work-items. This kernel invocation mechanism is similar to models from other heterogeneous programming languages such as OpenCL or CUDA.

hierarchical parallel-for enables formulating kernels using a nested parallelism model. A first invocation of `parallel_for_work_group` launches parallel work-groups, while contained `parallel_for_work_item` calls launch parallel work-items within a group.

Additionally, `single_task` is available for task-parallelism and for example FPGA programming.

2.1.1 Semantics of parallel-for paradigms

Basic parallel-for

The basic parallel-for paradigm lets a user specify the dimensionality of their problem and the number of work-items to execute for each dimension. Basic parallel-for provides easy access to a single-program multiple-data (SPMD) like parallelism and is useful if a task has to be executed independently for multiple data elements. A simple example is the map shown in the listing 2.1.

```
cgh.parallel_for(sycl::range<1>(global_size),
[=](sycl::id<1> id) {
  out[id] = map_fn(in[id]);
});
```

Listing 2.1: Invocation of a basic `parallel_for`. It performs a map operation on `global_size` items.

Hierarchical parallel-for

Hierarchical parallel-for allows cooperative solving of problems. For this, the problem is divided hierarchically. The first level is the work-group level. As seen in the reduction sample in listing 2.2, this level is initiated by invoking the `parallel_for_work_group`

function with the number of groups to launch and the size of each group. The kernel function is then executed once per work-group, potentially in parallel. All variables declared in this scope are work-group local memory.

Inside the work-group function, `parallel_for_work_item` may be used to execute a work-item function in parallel, similar to the basic parallel-for paradigm. The main difference is that this is launched for every work-group with the earlier specified work-group size. `parallel_for_work_item` can be executed multiple times. Each call is synchronous with regard to the work-group. This means the work-item function is run for every work-item before the work-group function continues. Effectively this implies a work-group level barrier after the work-item function so that all work-items have to converge again.

The work-item function may reference the work-group local memory. Using this local memory, multiple `parallel_for_work_item` calls can iteratively solve a problem, like the reduction seen in listing 2.2. Work-item private values can be forwarded between `parallel_for_work_item` calls using the `sycl::private_memory`. On CPU this allocates a heap array with as many elements as the work-group has elements. Each work-item can then index this with its id, to set and load its value.

```
cgh.parallel_for_work_group(sycl::range<1>{num_groups}, sycl::range<1>{group_size},
[=](sycl::group<1> wg) {
  int scratch[group_size];
  wg.parallel_for_work_item([&](sycl::h_item<1> item) {
    scratch[item.get_local_id()[0]] = acc[item.get_global_id()];
  });
  for(size_t i = group_size/2; i > 0; i /= 2) {
    wg.parallel_for_work_item([&](sycl::h_item<1> item) {
      const size_t lid = item.get_local_id()[0];
      if(lid < i) scratch[lid] += scratch[lid + i];
    });
  }
  wg.parallel_for_work_item([&](sycl::h_item<1> item) {
    const size_t lid = item.get_local_id()[0];
    if(lid == 0) acc[item.get_global_id()] = scratch[0];
  });
});
```

Listing 2.2: Sample reduction kernel using the hierarchical parallel for paradigm.

nd-range parallel-for

The nd-range parallel-for paradigm can solve the same problems as hierarchical parallel-for but uses better-known semantics for it. Instead of first starting a work-group function and then nesting invocations to work-item functions, the nd-range paradigm allows specifying a global and a work-group size directly. This specification is then used to partition the global size into work-groups of the requested

size and execute the SPMD kernel function in parallel. Work-group local memory has to be allocated upfront using a SYCL accessor. Synchronization between work-items of a work-group, to safely use the local memory, is implemented via group-barriers, which can be invoked using either `sycl::nd_item::barrier()` or `sycl::group_barrier(work_group)`. The listing 2.3 shows this, implementing an equivalent reduction as with the hierarchical kernel earlier.

```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
[=](sycl::nd_item<1> item) noexcept {
  const auto lid = item.get_local_id(0);
  scratch[lid] = acc[item.get_global_id()];
  for(size_t i = group_size / 2; i > 0; i /= 2) {
    item.barrier();
    if(lid < i) scratch[lid] += scratch[lid + i];
  }

  if(lid == 0) acc[item.get_global_id()] = scratch[lid];
});
```

Listing 2.3: Sample reduction kernel using the `nd_range` parallel for paradigm.

2.1.2 parallel-for implementation in hipSYCL

Basic parallel-for implementation

The basic parallel-for paradigm is implemented in hipSYCL on CPU by iterating over the work-items with nested for-loops, which are annotated with OpenMP's `#pragma omp for`. This is done inside a `#pragma omp parallel` scope. On GPU a hardcoded work-group size is used while enqueuing the kernel function.

Hierarchical parallel-for implementation

For the hierarchical parallel-for paradigm, the implementation on CPU is relatively straightforward. The work-group function is run inside an OpenMP parallel for loop nest, iterating over the number of work-groups. Thus the work-group function is executed in parallel exactly once for each work-group. The nested `parallel_for_work_item` calls are directly implemented by a loop nest, depending on the dimensionality of the kernel.

The innermost loop of the loop nest is annotated with the `#pragma omp simd` directive to suggest that the iterations do not depend on each other and instruct the auto-vectorizer to vectorize the work-item function, if possible. These inner loop nests are called work-item loops (wi-loops). By executing all work-items before the loop nest is exited by iterating over them in a single thread, the implicit barrier at the end of the `parallel_for_work_item` call is satisfied by design.

The GPU implementation of hierarchical parallel-for is not completely standards conforming, as it executes the work-group function for each work-item as well. As

the code is completely uniform, it should not impact functional correctness. The work-item functions are executed directly by the very same work-items. On CUDA `__syncthreads`[1] is invoked to satisfy the barrier requirement.

nd-range parallel-for implementation

The nd-range parallel-for variant is very similarly implemented on GPU. The work-group size is directly mapped to the block size of CUDA or HIP and the kernel function is executed directly for all work-items. The work-group barriers are implemented by calls to `__syncthreads`. In the following the historic and current CPU implementation variants are introduced.

Naive implementation On CPUs an efficient implementation of the nd-range parallel-for is inherently more difficult. It cannot be implemented by using an OpenMP parallel for loop for the work-group and an inner loop nest for the work-items. Only one work-item would make forward-progress at a time, which makes implementing the group barrier impossible. For this reason, hipSYCL formerly implemented this pattern by switching the hierarchical levels. An outer-loop represented the work-items and was parallelized with OpenMP using as many threads as there are work-items. An inner loop nest iterated over the work-groups and executed the kernel. Group barriers were implemented using OpenMP's `#pragma omp barrier`.

As SYCL is usable for both GPUs and CPUs, it is likely that users develop a kernel focused on GPUs but aim to execute on CPUs as well, if no GPU is present. For good performance on GPUs, it is usually necessary to use high work-item counts per work-group. The typical maximum is 1024 with CUDA and HIP. If this is directly mapped to the CPU, it would imply that 1024 operating-system-managed threads are launched to execute the nd-range kernel. This comes with a high overhead due to the necessary context switching.

Boost.Fiber based implementation In 2020 an alternative nd-range implementation on CPU was added. It is based on Boost.Fiber which provides lightweight concurrency by means of so-called fibers. Instead of expensive OS thread context switches, the fibers are executed in a single thread. At certain points in the program, like reaching a barrier or finishing execution of the executed function, the stack context is switched and the next fiber is executed. At first, this was done by executing a `#pragma omp parallel` region, which usually only spawns as many threads as the CPU provides cores, and from there launching as many fibers as a work-group has work-items. Those fibers then again executed the work-groups in a loop internally. The main benefit, therefore, was the lower number of required threads.

[1] CUDA kernel synchronization documentation: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#synchronization-functions

An additional improvement has been made to greatly improve performance for nd-range kernels without any barrier. For this, the iteration space was inverted again. This means the work-groups are the outer iteration space again, which is statically partitioned for execution by a reasonable number of OpenMP threads. Initially, only a single fiber is spawned, which iterates over the assigned work-groups and for every work-group executes its work-items by using a for-loop nest. It only does this until a barrier is hit.

If a barrier is encountered, more fibers are spawned so that there are as many fibers as work-items in a work-group. These fibers then cooperatively execute the kernel, still using just a single thread. Effectively this means, if a kernel contains no barrier, it can achieve acceptable performance as no context switches of any kind are necessary. As soon as a barrier is reached, the penalty of fiber context switches is present again.

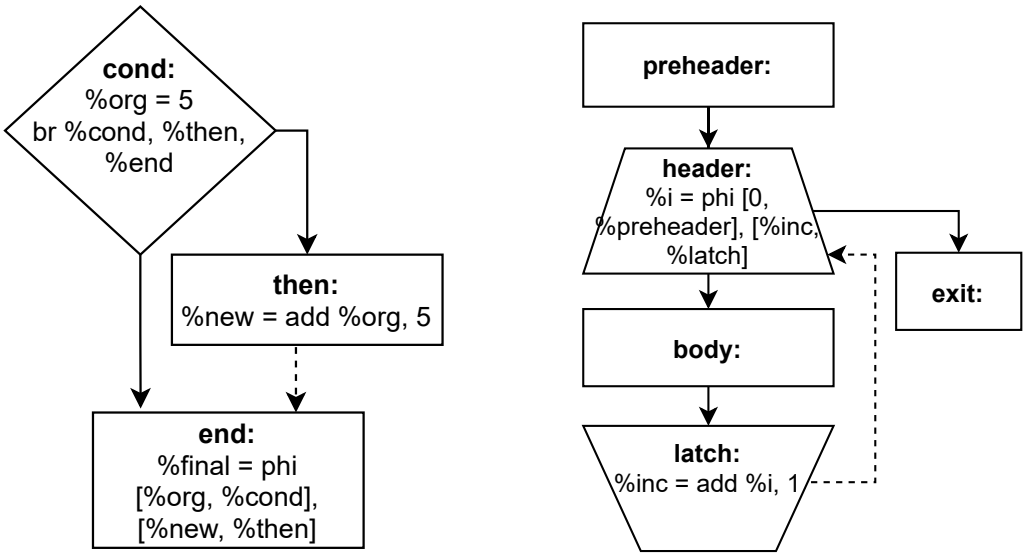
This optimization is discussed in Alpay and Heuveline [2021, slide 20]. The performance improvements are also observed by Lin et al. [2021]. This thesis aims to further improve on this by adding compiler support to hipSYCL to transform the nd-range parallel-for paradigm to be similar to the hierarchical one, in that it only executes barrier-free work-item loops inside a small number of threads.

2.2 LLVM

The LLVM infrastructure originally was presented in Lattner and Adve [2004]. It provides the Clang C++ front-end and a retargetable optimization and code generation infrastructure. In this thesis LLVM is used as the compiler framework of choice as it is easily extensible using front-end and optimizer plugins. It is also already relied on by hipSYCL for its GPU support.

LLVM's optimizer framework relies on the LLVM IR immediate representation of programs. The IR is organized hierarchically. The outermost level is the module, which in C++ terms is equivalent to the compilation unit. A module contains functions, the functions again contain basic blocks (BB). A basic block is an ordered list of instructions that always are executed sequentially without any conditionals or branching. The last instruction of a BB always has to be a terminator, which can be either a (conditional) branch to other basic blocks, a return or an unreachable instruction. Based on the branching terminators, the control flow between the BBs is modeled.

The IR is in single-static assignment (SSA) form, i.e. a value can only be assigned to once and the value does not change afterwards. Also values can only be used if the definition of the value dominates the use. Domination means that all paths to reach the use contain the definition of the value. This might not always be possible, e.g. if a variable is updated in a conditional and the original or the new value have to be used afterwards. For these cases, there are the so-called PHI nodes. A PHI node has a list of values it selects from, depending on the incoming branch to the BB. The value either has to be constant or dominate the terminator of the BB the



(a) CFG of a conditional with just a **then** block. The **then** block does not dominate the **end** block, as there is a path to **end** without **then**. **cond** dominates **end**, as it always comes before. As values cannot be updated, the **then** block creates a %new value for the `add`. The PHI node then selects the %org or %new value, depending on the incoming branch being from **cond** or **then** respectively.

(b) CFG of a loop. Illustrates the naming conventions. All edges are dominating except the **latch** to **header** edge. The **header** has two incoming branches. For each branch it has to select a value for the %i value. 0 is selected when coming from the **preheader**, the %inc value from the **latch**. The %i value definition dominates its *use*: the `add` instruction in the **latch**.

Figure 2.1: Simplified control flow graphs illustrating domination and the use of PHI nodes. Dominating edges are solid lines, dotted lines do not dominate.

branch is coming from. In this way, the SSA form can be kept even for loops, where the iteration (induction) variable depends on the value of the last iteration. This is visualized in 2.1.

Loops in LLVM have a BB called the header. This header dominates all other BBs in the loop. The loop body consists of the header and potentially more BBs. BBs from which a branch back to the header originates are called latches. The header therefore usually contains a PHI node that selects either the initial value for the induction variable if the block before the loop is the predecessor or the updated value if a latch is the source of the branch to the header. If this value starts with 0 and is incremented by 1 every iteration, it is called a canonical induction variable. Blocks that have a branch that leaves the loop are called exiting blocks. The block targeted by the leaving branch is called exit block, i.e. the first block after the loop.

Instructions in LLVM can have metadata attached. This can be used to propagate hints about optimization strategies through the optimization pipeline. An example for this is the `llvm.loop` metadata, that can be used to store user directed hints

about loop transformations. The `#pragma unroll` annotation in C++ code would lead to the `llvm.loop.unroll.enable` metadata being added to the operands of `llvm.loop`. This is then used as a hint by the optimization passes that perform the loop unrolling.

The LLVM optimizer is composed of many passes that perform certain transformations on the IR. There are module passes that operate on the whole module, function and loop passes only operate on their respective part of the module. A pass manager is used to order the execution of the passes. Plugins can add IR passes at predefined extension points.

2.3 Related work

Earlier work in the context of CUDA and OpenCL CPU implementations had to solve the same barrier issue. In Kaeli et al. [2015] the author describes the implementation of the AMD APP SDK's OpenCL CPU runtime. They use custom lightweight threads that allowed them to optimize stack location and alignment. Support for vectorization is only present when using OpenCL's explicit `floatN` vector data types which have overloaded mathematical operations that are mapped directly to vector instructions.

MCUDA by Stratton et al. [2008] is a CUDA CPU implementation using source-to-source transformations on an AST to perform deep loop fission, moving both loops and conditionals out of the work-item loops.

Patel et al. [2021] introduces an emulated GPU target for OpenMP offloading, achieving respectable performance despite mapping the work-items to single threads. The work-group size is usually auto-selected, so a reasonable thread count is achievable.

In Jäskeläinen et al. [2010] an OpenCL implementation is presented that aims to enable using OpenCL on application-specific processors (ASP). As OpenCL's kernel language includes work-group barriers as well, the work-group barrier issue had to be solved. The target processors were mostly FPGAs and not so much general-purpose CPUs, hence the kernel compiler tried to increase instruction-level parallelism (ILP) as far as possible. In Jääskeläinen et al. [2015] a continuation of this project, called portable OpenCL (POCL), is presented, aiming for high-performance execution of OpenCL kernels on CPUs. A CUDA backend also allows using POCL with Nvidia GPUs, where potentially no OpenCL driver is present. The POCL device compiler is discussed in more detail in 2.3.1.

POCL currently uses manually managed pthreads for parallelizing kernel execution on CPU. Baumann et al. [2021] analyzes whether using TBB as runtime instead could improve performance. They also find that LLVM's new VPlan and outer-loop vectorizer show promising performance uplifts.

A different approach to the work-group barrier problem was presented as continuation-based synchronization (CBS) in Karrenberg and Hack [2012]. The kernel is divided in barrier free sub-CFGs that are surrounded by a while loop with a switch statement

that selects the next to be executed sub-CFG. More on this in 2.3.2.

Vectorization of the data-parallel kernels plays a big role in CPU performance. Karrenberg and Hack [2011] presents a whole function vectorization approach for the kernel function which is more extensively discussed in Karrenberg [2015]. Tian et al. [2017] introduces implementation techniques for OpenMP's parallel and vectorization directives.

Currently an effort to refactor the LLVM vectorization infrastructure is underway. This new VPlan infrastructure has been introduced in Rapaport and Zaks [2017]. In the long run this should enable adding an outer-loop vectorizer in LLVM. Until the outer-loop vectorizer is ready, the Region Vectorizer (RV), implemented with Moll and Hack [2018], can add outer-loop vectorization to an LLVM based build. The inner-loop vectorizer and RV are discussed in more detail in 4.2.

2.3.1 POCL's device compiler

POCL implements an OpenCL kernel compiler that uses Clang's OpenCL C support as front-end and implements additional LLVM IR passes to transform the kernel functions, enabling well-performing execution on various targets.

The preferred way of handling kernels in POCL is:

1. create a `program`, which mostly represents the source
2. from this, create a `kernel`, a specific function from the source
3. compile the kernel
4. enqueue the kernel for execution

Step (3) in POCL is just an invocation of something similar to `clang -x cl kernel.cl -emit-llvm` to parse the OpenCL C(++) kernel language into LLVM IR and only in (4) does the actual target-specific optimization happen. For CPU targets step (4) involves transforming the scalar kernel function to a so-called work-group function. Hereby the scalar function works only on a single work-item and the work-group function contains work-item loops (wi-loop) that execute the kernel for each work-item in a work-group. As this target-dependent compilation happens at kernel enqueue time, the work-group size is also known, thus allowing the wi-loop to have compile-time constant bounds.

Transformations

As mentioned above, the target-dependent compilation step for CPUs involves generating wi-loops around the kernel to execute it for each work-item. This loop is then executed by a single thread, thus the work-items cannot guarantee forward-progress, as they are executed one after another. As OpenCL allows work-group barriers inside the kernel function, this leads to the same issues as in hipSYCL. To solve this, POCL implements a set of LLVM IR passes that transform the kernel

function into separate parallel regions, where each region starts and ends either at the kernel entry, exits, or at a barrier inside the kernel.

Each region eventually gets a wi-loop generated around it. The wi-loops are then chained after each other, leading to each parallel region being executed for every work-item of a work-group before any instruction of the next parallel region is executed. That way the synchronization guarantees required by the group barrier are met.

```
for(f : functions)
    if(isKernel(f))
        flattenKernel(f)
        O3(f)
        uniAnalysis = phisToAlloca(f)
        isolateRegions(f)

        for(l : f.loops())
            addImplicitBarriers(l)
        replicateBarrierTails(f)
        uniAnalysis += phisToAlloca(f)
        canonicalizeBarriers(f)
        isolateRegions(f)

        for(r : dfsParallelRegions(f))
            fixMultiRegionValues(r, uniAnalysis)
            if(containsConditionalBarriers(r)
                peelFirstWorkItem(r)
            createWorkItemLoopAround(r)
        removeBarrierCalls(f)
        O3(f)
```

Listing 2.4: Simplified summary of POCL kernel transformations.

In the following the transformation steps required are summarized. An outline gives the pseudo-code in listing 2.4. At first the kernel function is simplified and all call trees are inlined to make the barriers directly visible inside the kernel function. After a first round of the standard LLVM `-O3` pipeline, most PHI nodes are demoted to stack allocations, and dummy basic blocks are added at region entries to make the CFG transformations simpler.

Then, to handle loops inside the kernel that contain barriers, additional barriers are added to the loops' headers and latches. Hereby the loop control flow is handled as any other control flow without requiring special handling later. To retain the semantics of barriers in conditionals, tail replication is performed for the possible paths through conditionals to a barrier. In doing so every combination of visited barriers has its own kernel exit. Afterwards the barriers are split into barrier blocks, i.e. basic blocks with only a barrier and a terminator, which are then used as the beginning and end-points of the so-called parallel regions.

These regions are formed by conducting a DFS starting at the kernel's exits. At

every barrier found, all basic blocks visited since the last barrier, are pronounced a parallel region. Each parallel region could then be executed in parallel without having to consider any further synchronization.

In the final transformation the work-item loops are generated around the parallel regions. If a parallel region contains a conditional leading to a barrier, the region is replicated to execute it once for the first (0,0,0) work-item. This selects the uniform control flow to follow. After a barrier tail is selected, the parallel region is stripped of the conditional branches that led to selecting the parallel region. The tail is then wrapped in a wi-loop, starting with the second to be executed work-item. This retains the control flow and thus also the semantics of a kernel with barriers, given the requirement that every work-item has to execute a barrier if any one does so. For full optimization of the new control flow, another LLVM `-O3` pipeline is run by the OpenCL compiler.

One caveat is the additional, not OpenCL 3.0 C conforming, assumption that a loop containing a barrier (b-loop) is always executed the exact same number of iterations for every work-item. A loop with a barrier inside that is guarded by a conditional might not fulfill this requirement but could be allowed in OpenCL 3.0 C. More on this limitation in 3.3.

Using a simple variable uniformity analysis, uniform and varying values are identified. After the parallel region formation, the varying values that span multiple regions are stored in stack arrays. Existing varying `alloca`s are widened. The stack arrays are indexed with the work-item id so that the work-items' state can be retained over the region boundaries. The arrays are 64byte aligned for optimal vectorization opportunities.

As most PHI nodes are demoted to stack values before any transformations, all ex-PHI `alloca`s would be subject to being widened, as determining the uniformity of an `alloca` is difficult and often leads to pessimized results. To prevent this, the PHI demoting pass stores the uniformity value of the demoted PHI node for the corresponding `alloca` to the uniformity analysis. This can be looked up later before unnecessarily widening the `alloca`.

2.3.2 Continuation-based synchronization

A different approach to the work-group barrier problem was presented as continuation-based synchronization (CBS) in Karrenberg and Hack [2012]. Karrenberg proposes to define the entry and exits of the kernel as barriers and to split all barriers out of their basic blocks, so that only a barrier and a terminator form a so-called barrier block. Thereafter, a depth-first search (DFS) from the entry barrier through the CFG is then performed until all paths again reach a barrier. Every discovered barrier is given a unique ID. All blocks reached during the traversal outgoing from one entry form a sub-CFG. This process is then repeated starting with all newly discovered barriers.

The blocks of the sub-CFGs are replicated so that each sub-CFG forms a kernel on its own, which is completely barrier-free. The kernels are identified by the ID

of their respective entry barrier. At the exit points of the kernels, the exit barrier's ID is stored. This matches the ID of the sub-CFG that has to be executed next to resemble the original CFG.

Around each sub-CFG a wi-loop is generated to iterate over the work-items. The sub-CFG to be executed is selected by a switch statement in a while loop, that executes until a kernel exiting barrier is encountered. This handling of the control flow works well with the OpenCL and SYCL barrier definitions: if a barrier is hit by any work-item of a work-group, it has to be encountered by all work-items of that work-group. The while loop is executed for each work-group, which can be done in parallel.

To accommodate for private variables that are used in multiple sub-CFGs, a similar approach as in POCL is chosen: they are stored in an array, which, in the case of Karrenberg's implementation, is allocated by the driver, not directly on the stack.

Karrenberg expands this with an extended uniformity analysis, linearization of control flow, and explicit whole function vectorization of the sub-CFGs.

3 Comparison of loop fission approaches for synchronization

In the following the goal is to present compiler-based transformations that make the nd-range parallel-for paradigm execute more like the hierarchical paradigm on CPUs and thus more efficient. For this a work-group loop nest is parallelized using OpenMP. A non-parallelized loop nest for the work-items is inside the work-group loop nest. By performing deep loop fission on this inner loop nest, the required work-group barrier semantics can be realized.

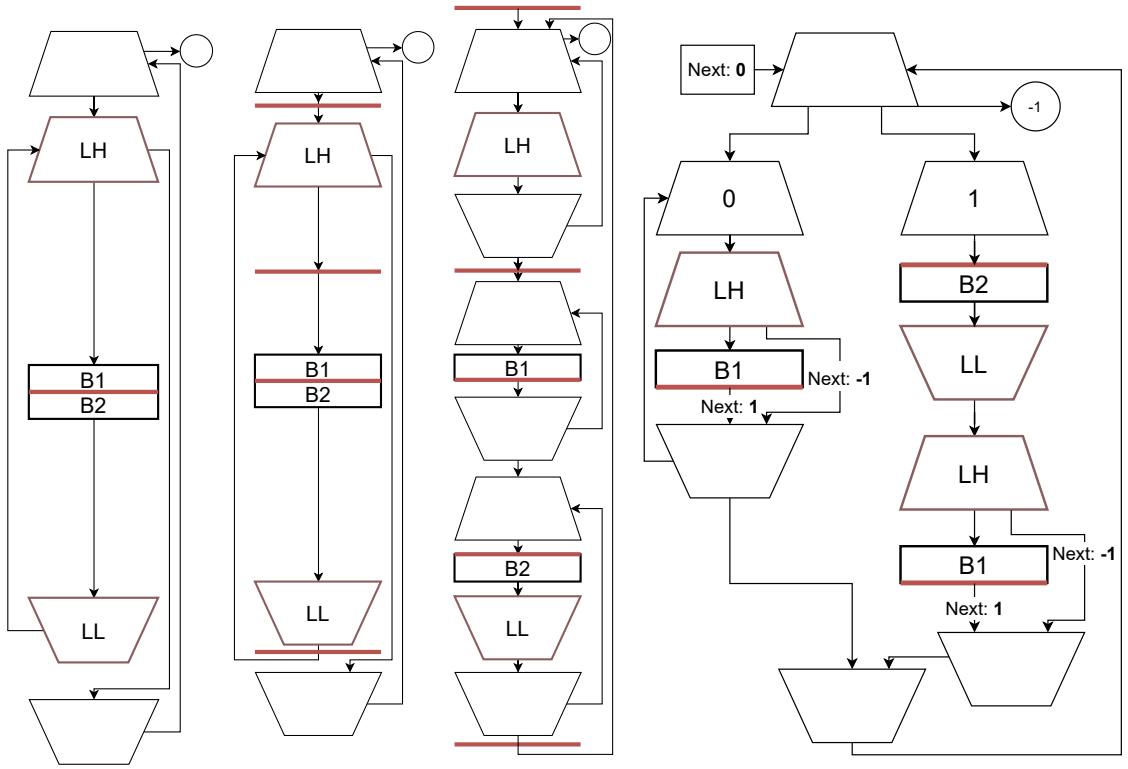
For the deep loop fission, the two approaches taken by the POCL device compiler and the continuation-based synchronization (CBS) method are presented. Figure 3.1 gives an initial example of the differences between the two techniques for a simple kernel containing a loop with a barrier. At the end of this chapter, the two loop fission implementations are compared, while the performance evaluation is done in chapter 6.

3.1 Integration of the POCL device compiler into hipSYCL

The open-source OpenCL implementation POCL, presented for example in Jääskeläinen et al. [2015] and briefly introduced in 2.3.1, employs an OpenCL kernel compiler based on the LLVM infrastructure. As the to-be-supported capabilities for OpenCL and SYCL kernels are very similar, this open-source compiler has been adopted into hipSYCL. Figure 3.2 shows the transformations done by the kernel compiler for two additional example kernels. There are some changes that had to be made for the integration, due to differences in the compilation flow and programming model. Listing 3.1 on page 27 therefore shows the steps that were adopted to hipSYCL in pseudo code.

In OpenCL the kernel code is traditionally written as a scalar function, completely separate from the host and driver code. The OpenCL driver usually compiles the kernel function and its dependencies on the fly as the host application prepares to use the kernel. Alternatively, the kernel can be pre-compiled to either an IR, like SPIR-V, or target specific binaries. Intel's ahead-of-time compiler, for example, supports offline binary compilation for Intel GPUs, FPGAs, and CPUs.

In contrast, the SYCL code is single-source, which means the kernel source is part of the host program's compilation flow. For accelerator targets like GPUs or FPGAs, the compilers detect and extract the kernel code and compile it to either IR or pre-compiled binaries for use on the targets. On CPUs, there are two options for

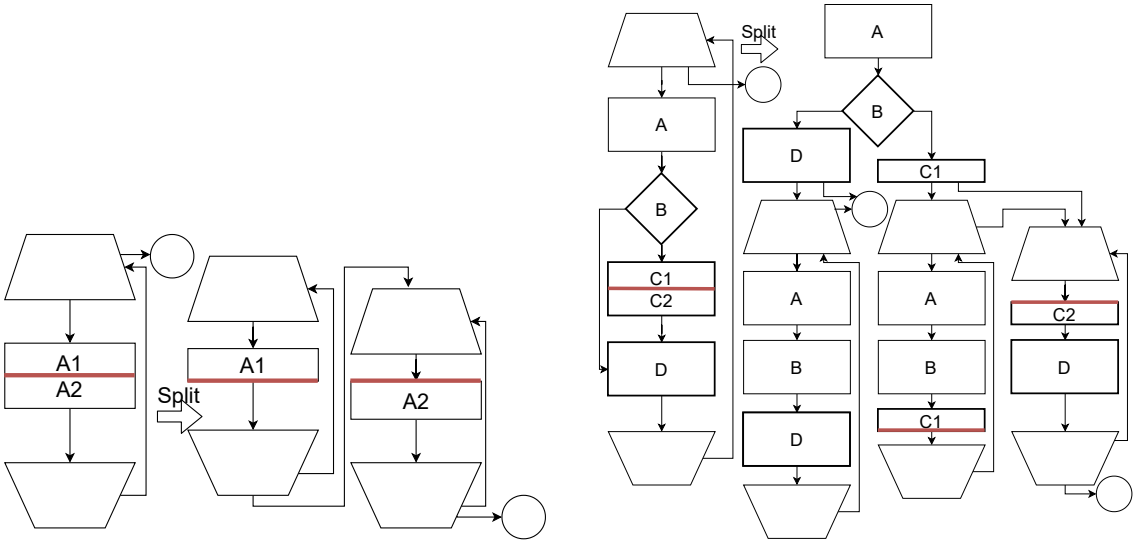


(a) CFG of a kernel with a barrier (red line) inside a kernel loop. LH/LL mark the inner loop's header/latch, which is enclosed in the wi-loop.

(b) The POCL compiler inserts three implicit barriers, as seen on the left: before the loop header, after the loop header in the body, and after the loop latch. Splitting the regions at these barriers and creating a wi-loop around each region leads to the CFG on the right. The kernel loop's back-edge is taken outside the wi-loop and links back to the first wi-loop.

(c) For CBS, the kernel has two sub-CFGs. One from the wi-loop body to the barrier or exit and one from the barrier to the barrier or exit. Around each sub-CFG, a wi-loop is generated. The next sub-CFG to execute is determined by the edge taken to the wi-loop latch. The outer while header switches to the target sub-CFG.

Figure 3.1: The same kernel with a loop containing a barrier, split once using POCL and once using CBS.



(a) Single barrier in simple kernel. The original body is split at the barrier and the parts are wrapped in a rotated wi-loop each, which are chained after each other.

(b) Barrier in conditional. The two paths that hit different barriers (in C or none), are replicated. The first parallel region is peeled to select the path, the wi-loops are peeled to select the tail to execute.

Figure 3.2: Two example kernels inside the wi-loop with a barrier each, showing the CFG after applying the transformations of the POCL device compiler.

a SYCL stack design: (1) Using an external runtime, like OpenCL, that implements the required execution paradigms and can work with an IR or native kernel binary. For this a compiler again has to extract the kernel to a format supported by the runtime. (2) Integrating the runtime directly into the SYCL framework, e.g. by using OpenMP or TBB to execute the inline kernels.

Due to SYCL's kernel submission approach, the latter variant can be mostly implemented in a library-only SYCL implementation without any compiler support and a compiler could perform combined host and kernel code optimization. In this case the kernel code is fully compiled to machine-specific code in a single pass. This is different from POCL's implementation, where a first simplifying pass for the IR generation is done, followed by another optimizing pass for the machine code generation at enqueue time. The optimizing pass also performs the `-O3` pipeline of LLVM twice: once before the work-item loop creation and once after.

POCL benefits from the already optimized IR, as the kernel function is already simplified and optimized before the CFG transformations are done. Another downside of the offline compilation in variant 2 is that the number of work-items is generally not known at kernel compile-time, except if compile-time constants are explicitly used at the kernel submission site, which could be propagated by a specialized compiler, or special attributes (commonly `reqd_work_group_size`) are used to indicate a fixed local size. Knowing the work-item bounds may improve perfor-

```
SAA = identifyKernelAndBarrierFunctions(module)
for(f : module.functions)
    if(SAA.isKernel(f))
        wiloop = findWiLoop(f)
        inlineBarriers(wiloop, SAA)
        uniAnalysis = phisToAlloca(wiloop)
        isolateRegions(wiloop)

        for(l : wiloop.inner_loops())
            addImplicitBarriers(l)
        replicateBarrierTails(wiloop)
        uniAnalysis += phisToAlloca(wiloop)
        canonicalizeBarriers(wiloop)
        isolateRegions(wiloop)

        for(r : dfsParallelRegions(wiloop))
            fixMultiRegionValues(r, uniAnalysis)
            if(containsConditionalBarriers(r)
                peelFirstWorkItem(r)
            copyWorkItemLoopAround(r, wiloop)
        removeOldWiLoop(wiloop)
        removeBarrierCalls(f)
        if(isO3)
            flattenKernel(f)
        O3(f)
```

Listing 3.1: Simplified summary of kernel transformations in hipSYCL using the POCL pipeline. hipSYCL adapted variant of 2.4 from p. 21.

mance and code size, as unnecessary bound checks and scalar epilogue loops, in case of successful vectorization, can be eliminated.

The hipSYCL project strives to provide a library-only implementation or otherwise rely on existing stacks. Consequently, it was decided to use the 2nd approach for this thesis. OpenMP is used to parallelize the loop nest over the work-groups. The work-item loop nest is located inside a dedicated kernel launcher function which is called directly from within the parallel work-group loop nest. For each dimension of the `nd_range`, a nested loop in both the work-group and work-item loop nests is present. Inside the innermost work-item loop, the kernel functor is called with the locally constructed `nd_item` as its parameter.

All transformations added to the hipSYCL kernel compiler pipeline are thus done directly on the work-item loop. To simplify these transformations, the work-item loop nest is annotated with `#pragma omp simd collapse(N)`, to merge the loop nest into a single loop.[1]

[1] In general using `#pragma omp ... collapse(N)` is considered a possible optimization, as the loop iteration count that OpenMP could parallelize on is bigger and branches are eliminated. Note, that for the `simd` variant, notable slowdowns have been observed, compared to just marking the innermost loop with `#pragma omp simd`. See 3.2.3 where the CBS pipeline gains

`[[clang::annotate("hipsycl_nd_kernel")]]` is used to mark the dedicated kernel launcher function. Therefore, by reading the `llvm.global.annotations` table of the LLVM IR module, all kernel function can be reliably identified in the compiler plugin. To ensure that the function is not inlined too early, it is also marked with `__attribute__((noinline))`. This attribute is removed after the strictly necessary transformations are completed. To easily identify the barrier functions, the same annotation mechanism is used, but with the `"hipsycl_splitter"` string value instead. Another benefit of using the annotations for the kernel launcher and splitter is that the compiler passes are not dependent on the runtime using OpenMP or the function names. Hence, a TBB backend or the new `sycl::group_barrier` functions could be added without modifications to the compiler.

The new kernel transformation passes in hipSYCL are run at the earliest possible extension point in the pass pipeline so that the full pass pipeline can optimize the transformed CFG afterwards. Compared to the POCL compiler, where the kernel function is optimized, transformed and optimized again, this implies that the IR input to the transformation passes is not as optimized (and simplified), yet. The first pass done by hipSYCL is the identification and marking of the wi-loops as such, so that they can be identified by the transformations later. To enable the required transformations for the loop fission, the kernel functor and recursively all functions inside the kernel, which transitively lead to a work-group barrier, are inlined into the wi-loop.

The difference between working on a dedicated kernel function and working inside the wi-loop required some, mostly technical, adjustments to the POCL LLVM passes. While being integrated into the hipSYCL LLVM plugin they were enabled to restrict their transformations to the work-item loop. Also, the variable uniformity analysis in the current version of hipSYCL is not as effective as in POCL, because the `nd_item` encapsulates both: uniform values and varying values, with respect to the wi-loop, like the work-group size or id and the work-item id respectively.

OpenCL on the other hand these values are encapsulated inside global functions like `get_group_id`. Uses of these functions are replaced in POCL by loads from variables with well-known names, which are then marked as uniform. As potentially more values can be marked uniform in POCL due to this difference, the OpenCL compiler might have fewer stack arrays. The stack arrays are a known source of missed optimizations, therefore less of them might benefit the OpenCL version's performance overall. The OpenCL approach could be partially re-used in hipSYCL as well, but given the possible alternative of using simplifying passes, as shown in section 3.2.2 for the CBS pipeline, the added complexity to the `nd_item` outweighs the short-term benefits.

Apart from that, the main changes are that the wi-loop structure is copied, instead of being created by the compiler. The POCL compiler also supports fully unrolling the wi-loop, which it does if requested via an option or if the work-group size is below

the ability to manually create the wi-loop in the compiler so that the innermost loop can be vectorized.

a certain threshold. This is not possible with hipSYCL, as the number of work-items is usually not compile-time known. Also, while creating the wi-loops, the OpenCL compiler performs partial unrolling to achieve higher instruction-level parallelism if an unroll factor greater than 1 can be found without leaving a remainder for the work-group size. This optimization is again dependent on knowing the work-group size at compile-time.

3.2 Integration of CBS into hipSYCL

In hipSYCL only the core transformations required for correctness as proposed in Karrenberg and Hack [2012] were implemented at first. The kernel launcher function is shared with the POCL pipeline. Therefore, the main differences between the paper and hipSYCL were, again, working with an already present work-item loop, although rotated to a do-while loop, as there has to be at least a single work-item for execution, missing on optimization opportunities due to the more complex nd_item, and using the stack allocation mechanism for work-item private values as in POCL.

3.2.1 Base implementation

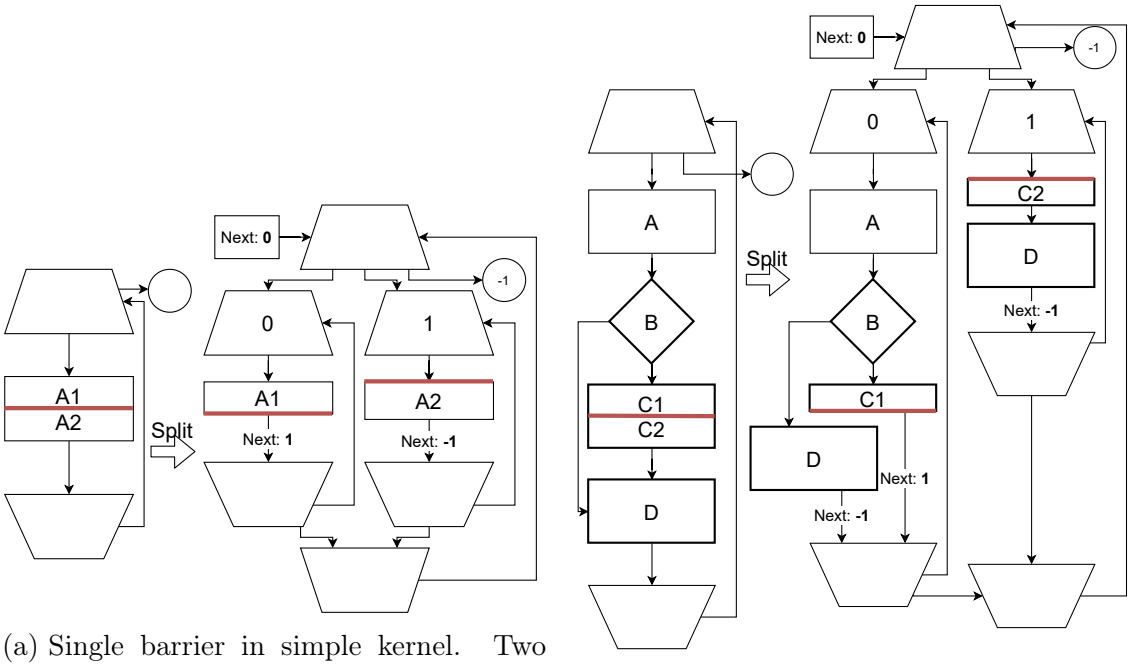
On the following pages, the specifics of the CBS implementation in hipSYCL are outlined. Figure 3.3 depicts the CFGs of two simple kernels and how the described transformations affect them. Listing 3.2 shows the CBS pipeline's steps in pseudo code.

Canonicalization

The CBS implementation in hipSYCL re-uses most transformations from the POCL pipeline. Thus, similarly, the CBS pipeline starts by identifying the kernel and barrier functions. An analysis pass called SplitterAnnotationAnalysis (SAA) collects the `"hipsycl_nd_kernel"` and `"hipsycl_splitter"` entries from the `llvm.global.annotations` table for this. Note, in hipSYCL terms the kernel function is the kernel launcher, containing the wi-loops and construction of the `nd_item` and not the user-provided kernel function.

The next step is marking the already present work-item loop. For this, custom metadata is appended to the so-called loop id, which is the `llvm.loop` metadata attached to the terminator of the loop's latch. The work-item loop is simply identified by being the outermost (and only) loop inside the kernel function.

With the wi-loop being identified, a recursive search for all work-group barriers inside the kernel is performed. The search makes use of the SYCL kernel function restriction that all functions called inside a kernel have to be visible in the compilation unit. All call instructions inside the wi-loop are followed to their function definitions, which then again are searched for call instructions. The called functions are checked with the SAA to identify barrier functions.



(a) Single barrier in simple kernel. Two sub-CFGs are formed. The first contains the contents of the loop body until the barrier and unconditionally stores that the 2nd sub-CFG should be executed. This sub-CFG is executed for each work-item before continuing, thus ensuring the semantics of the barrier. The 2nd sub-CFG then executes all instructions after the barrier before storing a -1 as next sub-CFG, indicating that the kernel is done after executing this sub-CFG.

(b) Barrier in conditional. Two sub-CFGs are formed: the first one executes A, B, and depending on the conditional B, C1 or D. If D was executed, the kernel is finished and thus the next case for the switch is -1, i.e. the exit. If C1 is selected, execution continues until the original barrier. Then it is stored, that the 2nd sub-CFG has to be executed next. The 2nd sub-CFG unconditionally executes C2 and D, as this is the only control flow left after hitting the barrier.

Figure 3.3: Two example kernels inside the wi-loop with a barrier each, showing the CFG after applying the transformations for the CBS.

```
SAA = identifyKernelAndBarrierFunctions(module)
for(f : module.functions)
    if(SAA.isKernel(f))
        wiloop = findWiLoop(f)
        inlineBarriers(wiloop, SAA)
        if(!isO0)
            flattenKernel(wiloop)
        phisToAlloca(wiloop)
        canonicalizeBarriers(wiloop)

        subCFGs = identifySubCFGs(wiloop)
        instAllocaMap = storeMultiSubCFGValuesToAllocas(subCFGs)

        for(subCFG : subCFGs)
            newWiLoop = copyWiLoop(wiloop)
            blocks = cloneBlocks(subCFG, newWiLoop)
            newInstAllocaMap = remap(blocks, instAllocaMap)
            loadMultiSubCFGValues(newWiLoop, instAllocaMap)

        createWhileSwitchAround(subCFGs)

        widenAllocas(f)
        fixSingleSubCFGValues(subCFGs, newInstAllocaMap)

        removeOldWiLoop(wiloop)
        O3(f)
```

Listing 3.2: Simplified summary of kernel transformations in hipSYCL using the CBS pipeline.

If a barrier function is identified, the call tree is stored while unwinding from the recursive search. Given the functions to be inlined, the pass performs a fixed-point iteration, inlining all relevant call instructions in the kernel function. This is done until no further calls to functions that should be inlined are left.

When a call to a barrier function is encountered, it is replaced by an undefined dummy function with the known name `__hipsycl_barrier()`. This simplifies later passes, as they can check for the function name only and do not have to check the function in the SAA. Another benefit is that uses of the `nd_item` parameter are eliminated, the effect of which will be discussed later.

As a next step, depending on the optimization level, the kernel is fully flattened, i.e. all functions are recursively inlined. Afterwards a pass is run, promoting `alloca`s to SSA values.

Again taken from the POCL pipeline, all PHI nodes, except the wi-loop induction variable, are demoted to `alloca`s. This greatly simplified the development of the following passes, as control flow changes can be performed without having to insert and update PHI nodes. After a loop simplify pass, the barrier canonicalization from POCL is performed. As a reminder, the main goal of this is to ensure that a call to

the `__hipsycl_barrier()` is the only instruction inside a basic block, apart from the terminator. This in return greatly simplifies later control flow changes, as the transformations can work on a basic block granularity.

A major difference between the POCL and CBS barrier canonicalization is that CBS requires the BB terminators to be simple branches or `ret` instructions, as the barrier blocks are discarded later. This would lead to invalid transformations if the branch was conditional. POCL uses barrier blocks with conditional branches to enable kernel loops. Apart from this in-block canonicalization, a barrier block is inserted to be the first basic block of the wi-loop body and right before the wi-loop latch, i.e. the last blocks of the kernel are ensured to be barrier blocks as well. These barriers are used as entry and exit barriers later, removing the need for custom handling of the entry and exit blocks.

Sub-CFG and dependency identification

With the barriers canonicalized, the sub-CFGs are formed. The first step is identifying all kernel exit barriers, which are equivalent to the predecessors of the wi-loop latch. Each of these barriers is inserted as a key to a map, with the exit barrier id `-1` as the value. The single entry barrier is also stored with the special id `0`. Afterwards all unknown barriers in the kernel are identified by iterating over all blocks in the wi-loop. Each new barrier is inserted in the map with a unique id.

For each of the non-exit barriers a sub-CFG is created. This is done by using each barrier as CFG entry once. From this, a simple DFS is performed, adding all visited non-barrier BBs to the sub-CFG. For every sub-CFG exit, i.e. the last BB before a barrier, the id of the following barrier is stored to a map. The barrier blocks are not included in the sub-CFG.

As the sub-CFGs are identified, it is now possible to determine which SSA values are defined in one sub-CFG and used in another. Still working on the original BBs, all users of every instruction in a sub-CFG are analyzed. If the instruction and the user do not reside in the same BB and the user is (also) contained in any other sub-CFG, the value has to be marked. If the instruction is a `load` or a `getelementptr` (GEP), it is checked, whether the pointer is already from a widened `alloca` instruction. In this case the `alloca` is re-used. Otherwise, a new `alloca` with an array length of `1024` is added to the function entry. The length is the arbitrarily defined maximum of work-items in a work-group for the hipSYCL CPU backend. It is chosen to match the number of allowed threads in a block in the CUDA programming model. As with the POCL implementation, the alignment is set to 64 bytes.

Immediately after the instruction that should be stored a GEP is inserted, which uses the wi-loop induction variable to index into the just created `alloca`. The GEP is followed by a `store`, writing the value of the instruction to the pointer returned by the GEP. The instruction to `alloca` relation is stored in a map for later use.

Sub-CFG replication

Only now the sub-CFGs are replicated. This is done by first cloning the wi-loop header and latch, which take care of the control flow and appropriate iteration count. Afterwards all blocks that were found by the DFS to belong to the sub-CFG are cloned. If a successor of an original BB is an exit barrier, a new block is created that writes the id of the exit barrier to a pre-created `alloca`. This `alloca` is called `LastBarrierId` and contains the last encountered barrier id, as the execution has to continue from that barrier later. After storing the barrier id, the new block branches to the wi-loop latch. The new block replaces the barrier as a successor in the replicated block. The original to new block and value mappings are stored in a value to value map. With the help of this, a new instruction to `alloca` map is filled, where the original instructions are looked up in the value to value map, and its clone is used as the new key to the same `alloca`.

Subsequently, an empty *load* block is created at the entry of the loop body. This load BB is then used to restore the values from the loop state `alloca`s. To do so, the original instruction to `alloca` map is iterated over. For every instruction that is not part of the sub-CFG that is operated on, it is checked whether any of the users are part of the sub-CFG. If this is the case, a GEP to the respective `alloca`, indexed with the new wi-loop induction variable, is created in the load BB, followed by a `load` from this GEP-returned pointer. The value of the `load` is then stored to the value to value map, with the key being the instruction value that had to be restored.

With the value to value map filled, the values in the wi-loop header, latch, and all the replicated sub-CFG blocks are remapped according to the value to value map. This means, that wherever an instruction that had to be loaded from a loop state `alloca` is used, the use is replaced by the `load` and so on. This also connects the BBs among each other in the sub-CFG, as the original BBs that are still used as targets inside the cloned branches are replaced by the new, cloned BBs.

Thereafter, the wi-loop is rotated to a do-while style loop as it is known that there will always be at least a single work-item, if executing the kernel at all. For this, the wi-loop induction variable is moved to be the first instruction of the earlier created load BB, which from now on serves as the wi-loop header. The cloned wi-loop header is now dominated by the latch. Therefore, the uses of the induction variable have to be replaced by the updated value that was used as an incoming value to the induction PHI node. The moved PHI node inside the load BB has to update its incoming block from the latch to the former header. For the LLVM `LoopInfo` the former header is now the latch. Thus the loop id, containing the loop annotations, like vectorization hints and so on, has to be copied to the header's terminator.

The last step of the replication is to store the load BB as the entry block (wi-loop header), the former wi-loop header as exit block, and the new induction variable for the while-switch generation.

After all sub-CFGs are replicated, the control structure is generated to re-instantiate the original control flow. The structure can be described as a while loop contain-

ing a switch expression. The switch selects the sub-CFG depending on the barrier id stored in `LastBarrierId`. For this a header block is created that contains a `load` from `LastBarrierId`, the value of which is then used in a following `switch` instruction.

The default target is a block which only contains an `unreachable` instruction, as this should never be selected. For every sub-CFG, a new case is added. The to-be-matched value is the id of the entry barrier of that sub-CFG. The target of the branch is the load BB of the sub-CFG, that was stored as the entry earlier. The wi-loop induction PHI node incoming block has to be rewritten from the original pre-header to the while header. The exiting target of the conditional branch in the exit of the sub-CFG, i.e. the wi-loop latch, aka the former wi-loop header, has to be changed to the while header.

Aside from the entry barrier id cases, a special case has to be added to the `switch`: the exit id. The target thereof is the original wi-loop's exit block. Finally, to get the while loop started, the initial entry barrier id is stored to the `LastBarrierId` in the original wi-loop pre-header. The branch is changed to point to the while header, instead of the original wi-loop's header.

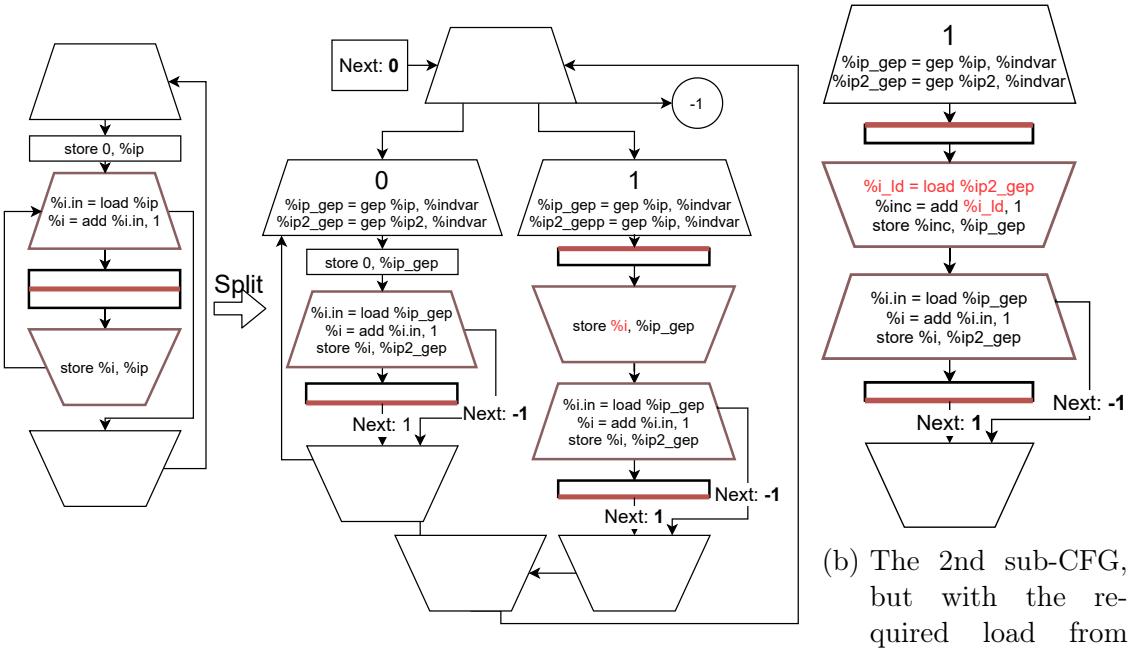
As the original wi-loop is now no longer needed and also not reachable anymore, it is cleaned up by a call to `llvm::removeUnreachableBlocks`.

Widening allocas and fixing sub-CFG internal dependencies

The last two steps to fix up the kernels are again related to correctly keeping track of work-item private values.

The first of the two fixes already existing `alloca`s. For this it is assumed all `alloca` instructions are in the function entry block. All `alloca` instructions are widened if they neither have metadata attached identifying them as loop state `alloca`s from earlier steps, nor have a user that is not inside any sub-CFG, nor have all transitive pointer users inside a single sub-CFG. Widening means a new `alloca` with the predefined 1024 elements and 64 byte alignment is created. For every sub-CFG, a `getelementptr` instruction is inserted that indexes the `alloca` with the induction variable. The sub-CFG local GEP then replaces all uses of the original `alloca` in its sub-CFG. The `alloca` must not have any uses left and can be erased.

The second value fix-up step is required for e.g. induction variables of kernel loops that contain barriers. As visualized in figure 3.4a, the second sub-CFG starts with the kernel loop's body and latch. Both of these potentially use the kernel loop's induction variable. As the header and thus the definition of the induction variable is only in the first sub-CFG or in the second sub-CFG after the body and the latch, the uses of the induction variable are not dominated there. The fact that the induction variable is defined in one sub-CFG (1) and used in another (2), leads to it being stored to a loop state `alloca` already. It is not restored yet, however, as the second sub-CFG contains both: a definition and uses of the induction variable. This is fixed by the next steps.



(a) On the left a kernel with a loop. The induction variable is stored and loaded using `%ip`, a pointer to an `alloca`. The loop induction variable `%i` already incremented in the header but used in the body and stored to `%ip` in the latch. After forming the sub-CFGs and creating the while-switch structure around them, the value `%i` does no longer dominate all its uses in the 2nd sub-CFG. No `load` from `%ip2_gep` is emitted yet, as both the user and definition are contained in one sub-CFG.

(b) The 2nd sub-CFG, but with the required load from `%ip2_gep` emitted. The existing `alloca` is re-used, ensuring that the correct value is already stored, when first entering the 2nd sub-CFG.

Figure 3.4: Handling of non-dominating values with definition and use in the same sub-CFG.

All instructions in a sub-CFG are checked whether the operands dominate the instruction. If one operand does not dominate the use, it has to be loaded from the loop state `alloca`. For this, a lookup into the earlier filled map that maps the new instructions to their loop state `alloca` is performed. If the operand is a `load`, the pointer is traced through the GEP to a loop state `alloca`, if possible. With the `alloca` already present, just a GEP and `load` instruction have to be emitted.

Note the position: the load should be added immediately before the user of the operand that on the one hand dominates the currently analyzed instruction and on the other hand is not dominated by any other user of the operand. This is relevant e.g. for loops that only conditionally enter a barrier. This leads to the loop being intact but rotated in the second sub-CFG. Thus the induction variable no longer dominates the latch, which is not only reachable from the load block but also from within the sub-CFG. Thus the load has to happen in the latch so that it uses the updated value of the induction variable.

The last strictly necessary step of this pipeline is removing the possibly leftover

barrier functions. As this pass is shared again with the POCL pipeline, it first searches any calls to the `__hipsycl_barrier` function and purges them. This should not trigger for the CBS pipeline as all barriers should have been eliminated by the sub-CFG formation already. Thereafter, it is checked whether the dummy barrier function still has any users. If not, the function is removed, as otherwise the linker would error due to the function missing a definition.

For optimization purposes another kernel flattening pass is invoked, and all memory accesses inside the wi-loops are marked parallel with respect to the corresponding wi-loop. The wi-loop id is extended with the `llvm.loop.vectorize.enable` metadata as well, to hint the LLVM loop vectorizer to vectorize the wi-loops. This also leads to LLVM emitting a warning, if it could not vectorize a loop, transparently informing the user of hipSYCL about the possible performance issue.

3.2.2 Variation: Invoking SROA before sub-CFG formation

As the wi-loops are parallel in nature, the best performance on CPUs is expected to be achieved by successful vectorization of the wi-loop. One issue that prevented successful vectorization of the wi-loops is that by inlining the kernel function, a call to `llvm.memcpy` is generated copying the `nd_item` from one stack allocation to another to satisfy the *by-value* semantics of a *trivially copyable* C++ type. The call to `llvm.memcpy` is currently not vectorizable by LLVM and thus led to the first wi-loop never being vectorized. The first wi-loop usually contains the index calculations and initial reads from global memory to e.g. shared memory which intuitively seem well vectorizable. In the default LLVM pipeline this would likely not occur for this certain type, as the `nd_item alloca` would be replaced with `alloca`s of its components by scalar replacement of aggregates (SROA). A memory to register promotion pass could easily replace those `alloca`s with SSA values as well.

The hipSYCL plugin has to hook into the LLVM pipeline at some point. Preferably this should be an extension point where CFG and loop transformations can be done with ease, LLVM optimizations later can optimize the added structures, and the assumptions on the input are manageable for all optimization levels. Therefore currently the `EP_EarlyAsPossible` or `PipelineStartEP` extension points are used for the legacy and new pass manager respectively. At this stage the SROA pass has not yet been run. Therefore the `nd_item` is still a single structure positioned in one or more `alloca`s. In the sub-CFG formation, the `nd_item alloca`s are widened as they are work-item private. With this change in place and the `nd_item` being used across multiple wi-loops, the standard `-O3` pipeline is no longer able to replace the `nd_item` with SSA values. Thus the `llvm.memcpy` stays inside the first wi-loop, preventing its vectorization.

To solve this, a minimal set of passes necessary to be run to eliminate the `nd_item alloca` were identified. These are run in the new pass manager pipeline after identifying the original wi-loop, inlining all function calls and replacing the calls to `nd_item::barrier` and `group_barrier` with `__hipsycl_barrier` inside the ker-

nel. The latter is required to eliminate all uses of the `nd_item` as such. The required passes are `IPSCCP` (interprocedural sparse conditional constant propagation and merging), `InstCombine` (combining of redundant instructions) and `SROA`. Afterwards a CFG simplify pass is run for clean-up of leftovers, but the three aforementioned passes lead to the `nd_item` stack allocations and `llvm.memcpy` call being eliminated for most kernels. Note, this is not done for the legacy pass manager, as the `IPSCCP` pass is a module pass, which cannot be added to the `EP_EarlyAsPossible` extension point used.

All in all, enabling the use of `SROA` before the CFG transformations significantly increased the chances for the first wi-loop to be vectorized by the LLVM loop vectorizer.

3.2.3 Variation: Scalar kernel function

During evaluation of another hipSYCL feature (scoped parallelism v2[2]), it was discovered that `#pragma omp simd collapse(N)` does not perform as well as just adding this hint without `collapse(N)` to the innermost loop. After some investigation it became apparent that the same loop nest, with the position of the OpenMP directive altered, would either not be vectorized at all or with a smaller vector width and / or interleave count if the outer loop was annotated with the `collapse(N)` version.[3]

To enable testing whether this had an effect for the nd-range kernels as well, the CBS pipeline in hipSYCL was altered to allow a kernel function without any wi-loop as input.

The kernel launcher work-group function was changed to create the local id and `nd_item`, and directly call the user kernel function with the latter as an argument, without any wi-loop around it. As the work-item induction variables are missing, the local id is constructed by using the three declaration-only globals with the well-known names `__hipsycl_local_id_x,y,z`, depending on the dimensionality.

In the LLVM plugin, the two modes are supported simultaneously: in the pass that usually marks the wi-loop as such, the user kernel is not yet inlined. Thus it is known that if there is any loop in the kernel function, it must be the wi-loop, which can then be marked as such. All later passes just have to check whether any loop is marked as wi-loop. If a marked loop is found, it is re-used and only the wi-loop body is considered the kernel. Otherwise the whole function is considered the kernel. This implies, for example, that the kernel exits are now the function exits and not the predecessors of the wi-loop latch, which has to be considered in the barrier canonicalization.

[2] https://github.com/illuhad/hipSYCL/pull/619

[3] Note the compiler remarks in the examples https://godbolt.org/z/3T5Mna3e4 and https://godbolt.org/z/coY97PGfd.

Creation of work-item loops for barrier free kernels

Nonetheless, the main changes are found in the sub-CFG formation. First of all, with an existing wi-loop, it is possible to check whether the wi-loop body contains any barrier at all and if not, the kernel is skipped. With the wi-loop missing, the sub-CFG formation pass has to create a loop structure around the kernel in all cases. If there is no barrier in the kernel, a new and empty function entry and a common function exit BB are inserted. All `alloca`s in the function are moved to the new function entry. Thereafter, the dimensionality of the kernel is detected by parsing the mangled name. The parsing currently only supports Itanium ABI.

With the dimensionality known, the local range values are searched. For this, knowledge about the kernel launcher function is required. The `local_size` argument of type `sycl::id<N>` is identified. An inter-procedural argument optimization might be performed, where the contained integers are passed on separately without the class hull. In this case, the arguments are called `local_size.coerce` with either `0`, `1`, or `1` as suffix. For the one dimensional ids the suffix is not present. In the optimized case, the argument values are directly used as the work-group size for their corresponding dimension. Otherwise the `local_size` argument is casted to an array of `N` integers of the largest size allowed for the target architecture, which should match the internally used `size_t` type. GEPs and `load`s are added to the function entry as required, to access the components of the array.

Having identified the necessary bounds for the wi-loop, it is created around the kernel body. Depending on the dimensionality, only the necessary loop nesting depth is created. To achieve this, the innermost loop is initially created by adding a new header, containing just a PHI node as the loop induction variable and a branch to the kernel body. Additionally a latch is added that increments the induction variable and conditionally branches to the header or to the earlier created function exit BB. This is repeated for each dimension, branching from the new headers to the last created header instead. Note that the innermost loop represents the last dimension $N-1$ of the SYCL range and id. After all headers and latches are created, the inner latches of depth D are updated to branch to the latch of the $D-1$ wi-loop instead of to the function exit block. The innermost loop is marked as wi-loop so that later passes, like the parallel marking and vectorization hint adding, work on this loop only, leading to the innermost loop being vectorized, similar to adding `#pragma omp simd` to the innermost loop.

All uses of the `__hipsycl_local_id_*` globals are replaced by the corresponding dimensions' loop induction variables. In the pass that usually cleans up the dummy barrier function, the globals are also removed if present, and all users are gone.

Adaptation to kernels with barriers

The case with barriers inside the kernel works rather similarly. The barrier canonicalization already took care of isolating the entry and exits. Thus, the function entry can immediately be used as the entry barrier, compared to first having to

identify the wi-loop body entry. In contrast to the wi-loop being present already, no contiguous induction variable exists at first. To enable the transformations detailed earlier that rely on the induction variable, a dummy `load` instruction is added to the function entry that serves as a placeholder and is removed again at the end of the pass.

Nonetheless, the main change to the wi-loop case is the sub-CFG replication. Here, the BBs of the sub-CFG are cloned immediately, without the wi-loop header and latch. The load BB is inserted before the other blocks and then the same loop creation routine from the non-barrier case is used to create the wi-loop nest. After creating the loops, a contiguous index calculation from the induction variables and the local size is inserted to the innermost header. This contiguous index replaces the dummy induction variable and is from now on used to create the loads and stores from the loop state `alloca`s.

The while-switch structure is created in almost the same way, as in the case of the already present wi-loop, the sole difference being the source of the BBs to be used as pre-header and exit. In the non-wi-loop case these BBs are the function entry and one of the exit barrier blocks. In the wi-loop case, the wi-loop pre-header and exit block are used.

3.2.4 Variation: Uniformity Analysis

Another identified issue with the code generated by the CBS pipeline is that contiguous values that might be used to index memory accesses are stored to loop state `alloca`s. As loop vectorizers mainly take the to be vectorized loop into account when analyzing uniformity and contiguity, the loop state `alloca`s hide this information as the vectorizer only sees a `load` of a potentially random value. To improve this, the uniformity analysis presented in Rosemann et al. [2021] and implemented by Moll in the open-source implementation of the Region Vectorizer[4] was pulled into hipSYCL.

The region that is taken into account is either the wi-loop or the whole kernel function, depending on the modus operandi, before sub-CFG replication. The analysis calculates the shape of a value with regard to a parallel region. A defined shape can be either uniform, strided (contiguous, if the stride equals the type size), or varying. For each value, it is analyzed whether it will have the same value for all work-items (i.e. uniform), or has a fixed increment between them, or may vary.

The uniformity analysis has to be seeded with basic initial information. For example, the kernel launcher function arguments can always be considered uniform. Additionally when a wi-loop is already present, all instructions that are not part of the wi-loop can be considered uniform with regard to the wi-loop. In case no wi-loop is present, the wi-loop induction variable or the dummy local id loads are pinned to a contiguous shape, as they are known to always increment by one. `alloca` instructions are initially considered uniform, until a non-uniform value is written to

[4] https://github.com/cdl-saarland/rv

a location inside an `alloca`'s frame.

With this information seeded, the uniformity analysis performs a fixed-point iteration to adjust the shapes of all non-pinned values. This, for example, leads to an `add` instruction being marked contiguous, if it has a uniform and a contiguous operand, like the work-group's offset to the global id and the wi-loop induction variable respectively.

For now, this information is used in hipSYCL, to restore the information required by the vectorizer to determine whether a memory access is contiguous or not. In theory, all uniform values that are put into loop state `alloca`s could be stored in single element `alloca`s as the value is equal for all work-items. This is currently only partially implemented requiring the changes from 3.2.5. Instead, for all contiguous values that are to be stored in a loop state `alloca`, the operands are traced recursively, until all discovered values are uniform or a loop induction variable. The visited contiguous instructions are collected and the uniform values are stored to a single element `alloca`.

While replication of the sub-CFGs, a new basic block is created as a pre-header to the outermost wi-loop. This is the uniform value load block. The uniform base values stored earlier are loaded here so that the load only happens once and the loaded value stays the same for all work-items throughout the sub-CFG. Thus work-items updating the `alloca` do not affect the value for the next iterations.

Inside the load BB, which is in the wi-loop body, the earlier collected contiguous instructions are restored. The clones are remapped to use the load values from the uniform load BB as well as the already replicated wi-loop induction variable. With this in place, a vectorizer should be able to determine more often that a certain memory access is contiguous and thus efficiently vectorizable.

3.2.5 Variation: Keeping PHI nodes

While evaluating the uniformity analysis from the last section, another problem became apparent: demoting the PHI nodes inside the kernel leads to a large number of `alloca`s which pessimize the uniformity analysis. One case where this happened, was the linear id calculations. For this, hipSYCL uses a for loop over the dimensions to multiply the accumulator with the local size of a dimension and add the dimensions' id component to the accumulator.

This would lead to an initial value of `0` being stored to the ex-phi `alloca`, marking it as uniform. In the loop body, a contiguous value would always be added before continuing past the loop. Thus the value after the loop would always be contiguous, but a pointer that has both a uniform and a contiguous shaped value stored to it, has to be marked varying. This causes contiguous values to not be correctly identified. A first mitigation was to loop-rotate the id and range calculations in the C++ code. As there is always at least one dimension, the first dimension's id is the initial value and the other values are then added in the for loop that starts at index one. With that change, this particular issue was solved.

For multi-dimensional kernels and other occasions, it would be useful to not demote the PHI nodes in the first place. To achieve this, the pass that performed the demotion is not added to the pipeline anymore. In the sub-CFG formation, there are mainly two places that required changes.

Firstly, the PHI nodes have to be considered whenever CFG transformations are done. For example in the case that a barrier divides the CFG, so that no back-edge for a loop is present in one sub-CFG, the PHI node still expects the incoming edge, which is invalid. Therefore, after replicating the sub-CFG and remapping the instructions, so that the control flow is intact again, all PHI nodes' incoming blocks are cross-checked with the actual predecessors of the block. As the predecessors will always be a subset of earlier existing incoming blocks, the old incoming block entries that no longer correspond with a predecessor can be purged.

The second place, where PHI nodes need special attention, is the fixing of single sub-CFG values. As the missing dominance of operands of an instruction is the main criterion for action here, this has to be refined. PHI nodes explicitly do not require incoming values to dominate the PHI node itself, but the terminator of the incoming block has to be dominated. This has to be checked to prevent storing all PHIs in `alloca`s again.

With PHI nodes being retained in the single sub-CFG handling, it is possible to move the `load` from the `alloca` to the load BB, instead of requiring it to be executed immediately before the using instruction, which would be unnecessarily often. This is possible because the PHI nodes were never demoted and for example, the initial value (the incoming value from the kernel loop pre-header) is the only operand that needs replacing and can be replaced by the `load` in the load BB. The value that comes from the latch still is handled as usual through the PHI node.

For further improvements on this end, a PHI node is manually inserted for the direct successor of the load BB, if it has another predecessor. This can be the case, for example, when the kernel contains a for loop with a barrier inside a conditional. Here, the increment of the induction variable in the latch was originally dominated by the loop header and therefore by the induction variable as well, and no PHI node was necessary. If the barrier was a direct predecessor of the latch, the latch might now be a direct successor of the load BB but other branches from the next loop iteration that might not hit the barrier also point to the latch. Therefore a selection has to be made whether the induction value of the last iteration, i.e. a load in the load BB, or a sub-CFG internal iteration, i.e. from the former loop header that no longer dominates the latch, is to be incremented. For this, a PHI node is added in the latch, which has multiple predecessors.

This last change is the missing piece to allow storing uniform values to a single element `alloca` instead of a loop state array. The value from that `alloca` is then loaded inside the wi-loop pre-header, alongside the uniform values required for restoring the contiguous values as discussed in section 3.2.4.

With all these variations combined, the number of loop state `alloca` arrays is reduced from 24 to a single one, for the blocked SYCL DGEMM implementation from Deakin et al. [2021]. The vectorizer is able to correctly identify all memory

accesses as contiguous. Not demoting PHI nodes is currently only fully supported, when the wi-loops are created by the compiler.

3.3 Direct comparison

The POCL device compiler and Karrenberg both present solutions to the work-group barrier problem, which were originally used for OpenCL C kernels. Both methods have been adapted into hipSYCL's LLVM plugin. As some analyses could not be directly adopted, the comparison will concentrate on their general solution to the barrier synchronization problem in data-parallel languages.

Both techniques follow a similar approach: they use a single thread to execute all work-items inside a work-group, eliminating the need for cross-thread synchronization. The work-item synchronization is implemented by splitting the wi-loop at the barriers instead. As the wi-loops are executed serially, this is an implicit form of synchronization between the work-items. POCL relies on barrier tail replication and work-item peeling to achieve correct execution of the control flow, whereas CBS splits the CFG into sub-CFGs, fully agnostic of conditionals or loops.

While CBS only requires that all work-items of a work-group reach the exact same barriers in the same order the same number of times, POCL adds another requirement: loops with barriers in them must execute the exact same number of iterations. Although the example given in listing 3.3 neither calculates something useful nor is it likely to be encountered in practice, it still is allowed by the OpenCL 3.0 and SYCL specifications.

```
[=](sycl::nd_item<1> item) noexcept {
  const auto lid = item.get_local_id(0);

  scratch[lid] = acc[item.get_global_id()];
  item.barrier();

  for(size_t i = 0; i < 2 + lid; i++) {
    scratch[lid] += i;
    // only call the barrier if all work-items still run the loop.
    if(i < 2) item.barrier();
  }
  acc[item.get_global_id()] = scratch[lid];
}
```

Listing 3.3: Example of a kernel that is correctly executed when using the CBS pipeline, but not with POCL.

Apart from this admittedly pathological case, the base pipelines both validate all SYCL code that has been tested so far. This includes benchmark suites like SYCL-Bench [Lal et al., 2020] and parallel research kernels [Hammond and Mattson, 2019],

as well as the hipSYCL adapted SYCL-BLAS[5] and oneDPL[6] libraries.

By performing the barrier tail replication and fully selecting the conditional paths to a barrier via peeling the first work-item, POCL's pipeline eliminates most branches in the hot work-item loops, exposing simpler control flow and more instruction-level parallelism. This benefits both, vectorization and targeting non-CPU targets like FPGAs, while yielding 14-45% higher instruction count before vectorization.

The CBS pipeline, on the other hand, is likely to produce harder to predict control flow as all conditional branches in the wi-loops are retained and the while-switch structure around the wi-loops adds additional complexity.

The uniformity analysis used in the CBS pipeline stems from an improved version of the uniformity analysis originally presented in Karrenberg and Hack [2012] and is comparatively precise in combination with keeping PHI nodes and executing SROA before the transformations. The theory behind the improved version has its correctness and runtime optimality proven in Rosemann et al. [2021] for reducible control flow. This uniformity analysis could be adopted for the POCL pipeline as well, but this is out of scope for this thesis.

The POCL pipeline could probably benefit from some of the variations implemented for the CBS pipeline, like performing more optimizations before the CFG transformations, keeping the PHI nodes, and supporting POCL's original wi-loop creation mechanism. Not demoting the PHI nodes would probably be a higher effort, as the pipeline performs more CFG transformations that would have to take the PHI nodes into account. Another issue with the POCL implementation is that the fixing of un-dominated values relies on the SSA value names being preserved, which usually is not the case when compiling without debug information for compilation performance reasons. Therefore, `-fno-discard-value-names` has to be added to all Clang invocations.

Due to the conditionals and loops needing explicit attention, the complexity and maintenance cost for the POCL implementation is probably higher than for CBS, where the core algorithm is very straightforward. For the CBS pipeline, the biggest code overhead is created by the uniformity analysis. The analysis is already integrated into upstream LLVM from version 12 on.[7] The main reason for pulling in the analysis from the Region Vectorizer is that currently LLVM 11 is still a supported target compiler for hipSYCL. Thus, as soon as LLVM 11 is removed as a supported compiler for hipSYCL, LLVM 12's divergence analysis might be usable with custom seeding to remove the additional maintenance overhead.

A performance evaluation will follow in chapter 6.

[5]https://github.com/codeplaysoftware/sycl-blas
[6]https://github.com/hipSYCL/oneDPL
[7]https://reviews.llvm.org/D84413

4 Finding the right vectorization strategy

To achieve peak throughput on CPUs, their vector units have to be used. These work in a single instruction, multiple data (SIMD) fashion. This requires either manually writing vector code with compiler intrinsics or relying on automatic vectorization by the compiler. Manually vectorized code tends to be hard to maintain, hence auto-vectorization is often relied on. At first, a vector type-based approach to auto-vectorization is briefly presented, and afterwards options for the wi-loop auto-vectorization are discussed.

4.1 Vector type based vectorization

One level of data-parallelism that is exposed by the SYCL standard is the `sycl::vec<DT, N>` type, where `DT` is an arithmetic type and `N` one of $1, 2, 3, 4, 8, 16$. These vectors are required to be `sizeof(DT)*N` aligned. With this alignment requirement satisfied, the Clang frontend e.g. decomposes the struct `sycl::vec<float, 4>` into two vectors of `<2 x float>`. This seeds the IR with vector instruction, which will most likely end up in the generated code.

To further improve this and allow the same kind of optimization for other data types than floats, it might be worth exploring explicit vector type annotations as provided by Clang.[1] This would closely match the behavior of OpenCL's `float4` and similar types. Additionally, types annotated as vectors support common arithmetic instructions, which will be mapped natively to the vector ISA, if possible. `vec<float, 4> operator*(vec<float, 4>, vec<float, 4>)`, for example, could be directly mapped to a multiplication of the underlying vector types of `float4` which, on x86 with AVX enabled, is directly implemented by the `vmulps` instruction.

Kaeli et al. [2015] explains that this vector type-based vectorization was, for example, implemented in the AMD APP SDK OpenCL CPU driver, where a similar approach as with the fiber-based hipSYCL back-end was chosen for work-item parallelization. To make use of CPUs' vector resources, one had to use OpenCL's vector types, like `float4`.

Using the proposed vector type annotation would probably be beneficial to speed up the fiber implementation in hipSYCL, if the `sycl::vec` type is used. For all other kernels, this would not change anything. Also, if the compiler support with loop fission is enabled, this might interfere with loop vectorization. One prime example is that the Region Vectorizer crashes if any vector types are already present in the to

[1] See the documentation: https://clang.llvm.org/docs/LanguageExtensions.html#vectors-and-extended-vectors.

be vectorized region. Possible benefits of this technique are therefore left for future evaluation.

4.2 Loop vectorization

The other, and even more prominent, data-parallelism exposed through SYCL is the use of parallel-for, for kernel submission. Due to the data-parallel nature (SPMD) of this paradigm, it is usually well mappable to the vector units. For this, a small number of work-items is grouped and each work-item is interpreted as a vector lane, so the instructions of the kernels only have to be widened and operate on the respective work-items' data.

To achieve this, there are two main approaches: relying on the LLVM loop vectorizer to successfully optimize the wi-loop, or secondly manually performing whole-function-vectorization on the user kernel [Karrenberg and Hack, 2011] or outer-loop vectorization [Nuzman and Zaks, 2008] on the wi-loop. These two main alternatives will be considered in sections 4.2.3 and 4.2.2. The outer-loop vectorization implementation used is the Region Vectorizer [Moll and Hack, 2018]. Before diving into the actual vectorization techniques, SYCL's sub-group hierarchical level is considered, as it might be a good candidate to expose manual vectorization level optimizations.

4.2.1 Considerations regarding SYCL sub-groups

The SYCL programming model assumes a multi-level hierarchy of execution resources. When submitting an nd-range parallel-for kernel, these are from the outer to the innermost scope: the full work-item space is divided into work-groups, which then again can be divided into sub-groups.

On GPUs, this closely matches e.g. the CUDA model, where the full grid is divided into thread blocks (max. 1024 threads), i.e. work-groups, which are implicitly divided into warps (32 threads), mapped to sub-groups. Certain SYCL functions, like group barriers or reductions, scans and more, are defined for both the work-group and sub-group level. On the sub-group level, the barrier on a GPU does not have to perform any operation, as the warp is usually executed in lockstep.[2] The scan and reductions can be implemented using efficient warp shuffle operations.

On a CPU, the mapping of work-groups and work-items is not as clearly defined. In general, a work-item could be mapped either to a thread or a fiber, ensuring forward-progress, but especially for bigger work-group sizes this would add scheduling overhead. The barrier synchronization is not as efficient as on GPUs and the SPMD data-parallelism of work-items cannot be mapped as easily to the vector units, leading to non-optimal performance.

As opposed to the above, this thesis proposes to map full work-groups onto a single CPU thread and execute its work-items in a loop in the work-group's thread.

[2] On newer NVIDIA GPUs, `__syncwarp` should be called.

By vectorizing that wi-loop, the work-item data-parallelism is directly mapped onto the CPUs vector units, providing better utilization of the CPUs' resources.

In hipSYCL on CPU the sub-groups are traditionally limited to contain a single work-item, as there is no immediate equivalent to GPU warps. According to a discussion on the specification's requirements for sub-groups,[3] they are intentionally vaguely specified to allow implementations to pursue different ways to optimize the sub-group level.

Therefore, to give this level in the hierarchy a more useful meaning, it could be used for tiling memory accesses, possibly leading to better cache usage or, when employing a whole function vectorization approach, using the vector width as sub-group size. In the tiling case, sub-group barriers would have to be implemented the same way as work-group barriers are, by performing deep loop fission using one of the approaches from chapter 3. In the latter case, similar semantics would be available on the CPU as are on the GPU with the warps running in *lockstep*. Group functions like broadcast or reduction could possibly be implemented using vector intrinsics; sub-group barriers would be no-ops.

To achieve the latter a fixed sub-group size for a kernel could be selected, very similar to how warps on Nvidia GPUs work. The inactive elements are masked if the work-group item count is not a multiple of the sub-group size. Note that masking is not supported by all CPU vector instruction sets and proper support for vector predication in LLVM, as required by e.g. ARM's SVE extensions, is still in progress.[4]

Another option is to use multi-versioning of the kernel. For this, a kernel is compiled for multiple sub-group sizes and the size to use for all sub-groups is selected at run-time. The selection uses the size that is the largest divisor of the work-group item count, thus eliminating the need for masking, but possibly resulting in smaller than necessary sub-group sizes, most likely 1 for odd work-group sizes.

According to the specification, an implementation has to guarantee only that the sub-group has to consist of the same work-items throughout the execution of a kernel, but does not require all sub-groups in a kernel execution to have the same size. This allows using the common pattern with auto-vectorization: execute all work-items $0, .., N - N\%sgSize - 1$ grouped into sub-groups of size $sgSize$ and the remaining ones with a sub-group size of 1, also called epilogue. This approach will be explored in section 4.2.3 in conjunction with the Region Vectorizer.

4.2.2 LLVM-Autovectorizer

The LLVM project already contains multiple auto-vectorizers that are fit for different code patterns. The two main vectorizers are the loop and the super-word-level-parallelism (SLP) vectorizers.

The loop vectorizer searches for loops where the iterations either do not depend

[3] https://github.com/KhronosGroup/SYCL-Docs/issues/193#issuecomment-943588298
[4] https://reviews.llvm.org/D57504

on each other or only in a detectable reduction pattern, like sum, max, min, etc. If such a loop is found, in the ideal case the instructions inside the loop body are just widened by the vectorization factor (VF) and the iteration increment is multiplied by VF. Therefore, after loop vectorization, multiple iterations are executed at once using vector instructions, leading to increased efficiency due to the parallel nature of vector instructions and coming with additional benefits like less CPU front-end congestion, as fewer instructions have to be decoded, fewer branches are taken and so on.

The SLP vectorizer, on the other hand, is not limited to working on loops, but instead looks for any parallelism in memory accesses and arithmetic operations, combining these to vector instructions if possible. Partially unrolled loops, "structure accesses, such as RGBA in image processing [..], non-SIMD kind of parallelism, like subadd pattern, which is popular in complex computations [..], and numerical computations, like FFT, that require data reorganization" are common cases where the SLP vectorizer has optimization potential, according to [Rosen et al., 2007, p. 131].

Both vectorizers are part of the main optimization pipeline, almost at the end of the target-independent optimizations. Even though they are not part of backends, i.e. target-specific code generation, they account for information about the target. The available vector register widths and cost model, which is used to decide if vectorization and which vector width is profitable, are highly target-specific, for example.

SYCL nd-range kernels are inherently parallel and the work-items of those are executed using wi-loops, as of the implementation in this thesis. Thus, the loop vectorizer, exploiting the parallelism of loop iterations, is in theory the ideal vectorization strategy.

The first step in identifying a vectorizable loop is the legality analysis. Prominent parts of which are the uniformity and loop dependence analysis. The latter is to identify if one iteration writes to a memory location and the next iteration reads from the very same location. In this case, the vectorizer is likely not able to optimize the loop. Such accesses in wi-loops would only occur, if the kernel function reads and writes to the same location with the same index for multiple work-items, hence without using barriers, the result is undefined per the specification.

Therefore, to help the loop vectorizer understand that there are no legal in-wi-loop dependencies between work-items, the `llvm.access.group` metadata is added to all memory accesses inside the wi-loop. This access group metadata is then referenced by a `llvm.loop.parallel_accesses` metadata that is part of the wi-loop's loop id. This indicates to the vectorizer that all memory accesses part of the access group, are independent with regards to the wi-loop, and may be executed in parallel, i.e. being potentially vectorized.

After the vectorizer ensured it is legal to vectorize the loop, it tries to find an optimal strategy to do so. For this, it employs the inner-loop vectorizer stage to widen or replicate the instructions of the loop body or replace loop-carried dependencies with reduction patterns. This is done for all supported vector widths and

also some partial unrolling factors are explored. For each of these combinations, the cost model, which employs target-specific heuristics to estimate the cost of using the instructions, is calculated. The most promising variant is then selected and emitted.

While the vectorizer generally considers all loops, `llvm.loop.vectorize.enable` can be added to force vectorization, even though the vectorizer's cost model came to the conclusion that vectorization of a loop is not profitable.

Common issues with the LLVM loop vectorizer are instructions that it cannot vectorize, especially calls, but also complex control flow. Examples of problematic control flow are loops with multiple exits and nested loops. The nested loop issue is relatively likely to come up with SYCL kernels, while the wi-loops never have multiple exits. As described in 3.2.2, a major issue with vectorization was the presence of calls to the `llvm.memcpy` intrinsic.

While the control flow issues are slowly being solved, e.g. some multi-exit scenarios are supported from LLVM 13 on[5], the general vectorization infrastructure of LLVM is also revamped. Intel proposed introducing VPlan. The proposal targets to provide the means to cheaply attempt multiple optimization and vectorization strategies and select the most appropriate one. While introducing the new infrastructure, the loop vectorizer is also targeted to be extended to support outer-loop vectorization.

Only with the last change in place, LLVM's loop vectorizer is expected to be able to fully support the wi-loop vectorization. Enabling VPlan + outer-loop vectorizer with the `-mllvm -enable-vplan-native-path` option, led to clang crashing in the vectorizer for the tested trunk version (5352ea4a721e). The Region Vectorizer is considered as a drop-in alternative until the new outer-loop vectorizer is ready.

4.2.3 Region Vectorizer

As the LLVM loop vectorizer still has a number of constraints regarding supported control flow and the new VPlan infrastructure and outer-loop vectorizer are still highly experimental, the Region Vectorizer (RV) is used, to check whether forced wi-loop vectorization benefits the performance.

RV builds on top of Whole Function Vectorization [Karrenberg and Hack, 2011]. There SPMD kernels are implemented using the presented continuation-based synchronization, but the loop body is contained in its own function, which then is fully vectorized with a fixed vector width. This is equivalent to vectorizing the innermost wi-loop. Moll and Hack [2018] generalizes the function vectorization capability to vectorize regions, where a region is either a loop or a function. For loops, this explicitly includes outer-loops, which exactly fits the use-case for both, the POCL and CBS pipelines in hipSYCL. The original version of RV only supported the outermost loops of a function, which was adapted for this thesis.

RV can be built as an LLVM plugin that can be used along with the hipSYCL plugin, thus allowing to en- or disable the transformations at will. While recent

[5] A status report regarding the multi-exit cases can be found at https://github.com/preames/public-notes/blob/master/multiple-exit-vectorization.rst#open-topics.

development mostly happened on a fork targeting the NEC Aurora Vector Engine, it requires not yet upstreamed changes to LLVM. Henceforth the LLVM 12 compatible version from the Compiler Design Lab Saarland was used as the base. Some changes from the NEC fork were pulled in, as well as some necessary fixes were added. The used version can be found in the author's fork.[6]

Outer-loop vectorization

The basic outline of how the Region Vectorizer works for outer-loop vectorization is as follows: it starts with detecting the outermost loops that are marked with both `llvm.loop.vectorize.enable` and `llvm.loop.parallel_accesses` with appropriate groups, as done by the hipSYCL loop fission pipelines. These two annotations can also be added from C++ code by using `#pragma omp simd` and allow neglecting most legality checks. The used version also checks for `hipSYCL.loop.workitem` to only add the additional compile-time overhead for the nd-range kernels.

With the to be vectorized loop selected, RV checks whether a forced vector width is set, either by the environment variable `RV_FORCE_WIDTH` or the loop id metadata `llvm.loop.vectorize.width`. If so, this width is used for vectorization, otherwise, the default width for the selected target is used as maximum and all instructions in the region are checked for their maximum supported vector width to refine it. This may be due to using larger bit sizes or calls that cannot be vectorized.

All calls are run through multiple resolvers, trying to find a way to vectorize it for a certain width, starting from the current maximum, decreasing by a factor of 2 until 1 is reached: either the `llvm::TargetLibraryInfo` might know a vectorized equivalent of that function, there's a SLEEF [Shibata and Petrogalli, 2020] implementation, the function has already been fully vectorized or it is possible to recursively vectorize all functions from that call.

If a call is not vectorizable at all, the vector width is set to 1 and vectorization is aborted. This is e.g. the case for LLVM intrinsics without vectorized equivalents, like the `llvm.memcpy` intrinsic, but also for arithmetic intrinsics like `llvm.uadd.sat`, if the CPU does not natively support the required data type. Note, that this is not the case if a forced width is selected. In that case, the unvectorizable calls are replicated *VF* times, which the LLVM vectorizer calls scalarization.

After the vectorization factor is selected, a custom reduction analysis tries to identify the reduction patterns for each PHI node of the loop header. Supported reductions are addition, multiplication, logical and/or, min and max.

Following is the creation of the remainder loop control flow. This is relatively similar, to what the LLVM vectorizer does. Starting with adding a conditional branch, that depends on whether the iteration count is at least equal to the vector width, and selects either the newly cloned vector loop or the original scalar loop. At the exit of the vector loop, another check is added whether the iteration count was divisible by *VF*. If iterations are left, the scalar loop is executed as the epilogue.

[6] CDL: https://github.com/cdl-saarland/rv/tree/release/12.x, NEC: https://github.com/sx-aurora-dev/rv, author's used version: https://github.com/fodinabor/rv/tree/release/12.x.

Afterwards, the uniformity analysis from Rosemann et al. [2021] is run on the loop to mark values as uniform, strided or varying. The procedure of the analysis is already described in 3.2.4. However, the seeding is slightly different, as only the single parallel marked loop is considered and additional information as in the wi-loop case is missing, are all values outside this loop considered uniform. Only `alloca`s may be changed on write. The seeding for values inside the loop is only added to the PHI nodes of the loop header. The pinned shapes for those nodes are informed by the results of the reduction analysis.

With the knowledge about which values are uniform, the control flow is partially linearized according to the algorithm in Moll and Hack [2018]. After some small-scale optimizations to replace `llvm.memcpy` calls by vector loads and stores, as well as optimization of struct allocas, the main vectorization step happens. For this, all basic blocks are traversed in order of the dominator tree. For every instruction inside the BB, depending on its kind, a new equivalent one, with widened operands and result type, is inserted into new BBs. Instructions that cannot be vectorized are replicated. PHI nodes, for example, are handled specially as they allow loop carried dependencies and thus have to be reduced.

Using a during the vectorization filled scalar to vector map, the branches are updated to connect the new vector BBs up. Finally, the old region is made unreachable and all instructions are erased. If the value of a scalar instruction is used outside of the loop, it originally was just replaced by an `undef` value and erased, under the assumption that outside uses of values should be fixed by the reduction handling already.

This is not the case, though. For example, the `LastBarrierId alloca` used by the CBS pipeline is promoted to SSA values, leading to a PHI node inside the loop latch only and not the header. This PHI node then is used in CBS's while loop's latch. Therefore, it is not a reduction and must be handled differently. To keep the original semantics that the value which is stored by the last iteration to the `alloca` is the value used outside the loop, a fix-up step was added to enable using RV with hipSYCL. If a use outside the region is detected while deletion of the scalar loop, it is replaced by either the still scalar value inside the vector loop if it was uniform or the last element of the corresponding vector value.

Intrinsics

RV implements a number of special intrinsics. This includes boolean reductions like `rv_any`, `rv_all`, the broadcast `rv_extract` as well as providing information about the SIMD width and lane, i.e. `rv_num_lanes` and `rv_lane_id` respectively.

RV implements these intrinsics for the vectorized loop by replacing them directly with the corresponding values. For example `rv_num_lanes` can be replaced by a constant integer value set to VF; `rv_extract` is directly mapped to the LLVM `extract` instruction. If no vectorization happens due to either failing legality or cost model checks, RV lowers the intrinsics to the corresponding scalar value nonetheless. The `rv_num_lanes` intrinsic would be replaced by the constant integer value 1,

`rv_extract` by the scalar value of the single active lane.

The `rv_extract`, `rv_insert` and `rv_shuffle` intrinsics default to using single precision floating point values. This could be extended by adding explicitly typed functions, as done with LLVM intrinsics that contain a type suffix, as in the 32bit integer variant of the saturated add `llvm.uadd.sat.i32`.

These intrinsics can be used as building blocks for introducing a SYCL sub-group hierarchical level on the vector level. The following will present the attempts made at introducing an RV-based sub-group and corresponding algorithms.

The first thing to note is that to achieve any kind of reliable programmability and to actually implement the standard, a vectorization width has to be enforced for RV, either for each kernel or the whole program. Otherwise, RV might decide to vectorize the wi-loops inside a kernel with differing vector widths, which directly correspond to the sub-group widths. This would imply that the sub-group size could change before and after a barrier and thus also which work-items cooperate in a sub-group. The standard calls for a sub-group to be of the same size throughout the whole kernel. Not providing this guarantee makes it near impossible to develop algorithms using the sub-group level. The width can be fixed by either using the `RV_FORCE_WIDTH` environment variable or by adding the `llvm.loop.vectorize.width` metadata to the wi-loops' loop id.

Sub-group implementation using RV intrinsics

The first step to adopting RV's vectorization as sub-group level is to implement the `sub_group` API with RV intrinsics, which is straightforward. Whenever the sub-group id is required, `rv_lane_id()` is used and the range is implemented by querying `rv_num_lanes()`.

This is followed by the implementation of the sub-group algorithms, starting with the simple ones like `group_any_of(sub_group, bool pred)` which checks if the predicate argument is true for any work-item of the sub-group. This algorithm can be implemented in terms of `rv_any(pred)`. `group_all_of` is implementable with `rv_all(pred)`, as is `group_none_of` by negating the input. The sub-group `group_barrier` is currently a no-op, although according to the specification, it should invoke at least a release and acquire memory fence for the work-items in the sub-group.

For the implementation of `group_broadcast`, where actual values and not only booleans are used, more work is required. As mentioned, `rv_extract` has the semantics of a lane broadcast but only supports single-precision floats as values. Therefore, to implement a broadcast using the RV intrinsics, for, for example, a 64bit integer or even `sycl::vec` types as allowed by the templated nature of the SYCL function, the value to broadcast is divided into multiple parts on a byte level. The 32bit parts are reinterpreted to single-precision floats, which then are broadcasted using the `rv_extract` intrinsic.

It should be noted that this is likely less efficient on the CPU for non-32bit-sized values than just writing the value to either a well-known shared location or ex-

changing stack pointers using a specialized function for that case. Nonetheless this approach has been chosen, as these intrinsics mimic quite well what GPU programming models like CUDA and HIP offer, where this word-level splitting for broadcasts and shuffles has to be done as well. Therefore the similarity in the implementation could be beneficial for maintenance.

The core approach of shuffling only single precision floats around is also used to implement the `shift_group_left`, `shift_group_right`, `permute_group_by_xor` and `select_from_group` functions. The main difference between them is the indexing. While the broadcast only required extracting from a uniform lane of a vector value, the shift and permute functions have to move all values. To achieve this one `rv_extract` is used for each lane to request the value from the offset lane in the input vector value. This is followed by one `rv_insert` to move the value to the target lane inside a new vector value.

As the *VF* is not constexpr known, this has to be done inside a loop, which according to manual IR inspection is commonly optimized out by fully unrolling the loop. Note, that while both `rv_extract` and `rv_insert` require the lane index only to be a uniform value, the `rv_extract` should do the non-constant indexing and `rv_insert` constant indexing for better code generation.

Another special case exists for `select_from_group`, where all lanes can request the value from a random non-uniformly indexed lane. This does not work by just using an `rv_extract`, as it requires a uniform lane index and returns only a scalar value. Thus, to support this shuffling, the requested index is initially extracted from an index vector and then is used in an extraction from the value vector. The extracted value is inserted into the result vector, one lane at a time.

Sub-group reduction with RV intrinsics

On both a work- and sub-group level, SYCL standardized group reduction as well as inclusive and exclusive scans. Of these, only the reduction was adapted. Originally, as sub-groups in hipSYCL on CPU are just a single element wide, the one input value is returned for the sub-group case. For the RV variant with bigger sub-groups, the CUDA warp reduction was used as a base, as the warp intrinsics map well to the RV intrinsics.

Both versions first identify the active lanes in an integer bitmask. CUDA's `__activemask()` is replicated by `rv_ballot(rv_mask())`. Apart from that, the implementation is very similar: it consists of a for loop over $i = VF/2$ to inclusively 1, each step halving i, where the current local value is shuffled down by i. If $laneId + i$ is an active lane, the local value is combined using the binary operation with the value from $laneId + i$. After the loop, the result is broadcast using the typed wrapper around `rv_extract`.

In theory, this could be optimized by using vector reduction intrinsics directly. LLVM 14 will add `min`, `max`, `add`, `and`, `or` and `xor` builtins. To use these directly from C++, the vector type language extension mentioned in section 4.1 would have to be used. Therefore a dispatch based on the *VF* has to be added. The listing

4.1 shows an example implementation of how this could work. Note that the single-precision float is used here, as the `rv_extract` intrinsic currently only supports that type. This could be extended by adding more typed versions of the intrinsic. As mentioned in section 4.1 as well, the version of RV currently used is not compatible with input IR which contains vector instructions already. This would have to be fixed as well.

```
float group_reduce4(sub_group g, float x, sycl::plus<float> binary_op) {
  // create vector type of right size, so that the builtin can be mapped correctly
  float4 vec{rv_extract(x, 0), rv_extract(x, 1), rv_extract(x, 2), rv_extract(x, 3)};
  return __builtin_reduce_add(vec);
}
float group_reduce(sub_group g, float x, sycl::plus<float> binary_op) {
  switch(rv_num_lanes()) { // select implementation based on vector width
    case 1: return x;
    case 2: return group_reduce2(g, x, binary_op);
    case 4: return group_reduce4(g, x, binary_op);
    case 8: return group_reduce8(g, x, binary_op);
    default: return group_reduce(g, x, binary_op); // generic non-intrinsic version
  }
}
```

Listing 4.1: Example implementation of `group_reduce` using RV and Clang vector intrinsics for a VF of 4.

Alternatively, additional reduction intrinsics could be added to RV, which take a scalar input and are mapped to correspondingly widened LLVM reduction intrinsics during vectorization.

Work-group reduction using sub-group optimization

The sub-group reduction is then plugged into the work-group reduction. As presented in Wünsche [2021], the current work-group reduction has all work-items write their value to a common scratch memory and then uses only the very first work-item of the group to iterate over all those values and accumulate them. The result is then broadcast to all work-items. This is much more cache efficient than using a work-item cooperative approach, especially as both the fibers and the loop fission approaches do not have multiple threads working independently in a work-group.

To improve this using the sub-group reduction two main ideas were considered. The first idea was to again adapt the GPU variant of the work-group reduction. This turned out to be infeasible, as this implementation is based on the knowledge that the warp size (i.e. sub-group size) on supported CUDA and HIP GPUs is either 32 or 64, and the block size (i.e. work-group size) is at a maximum 1024. This means, there have to be at most two sub-group reductions, per sub-group. The first one reduces the sub-group's value, leading to a maximum of 16 or 32 values left for the work-group. On CUDA, if the work-group size was 1024, the remaining 32 values

are again reduced by a sub-group reduction. If the work-group size was different, or an AMD GPU is used, a cooperative group reduction with group barriers follows. As the sub-group size on CPUs is much lower, the first case with just two sub-group reductions is far less likely to occur. Therefore the other case has to be the optimized case.

It was then decided to implement a version that works with both forced and variable vectorization widths. This means that no assumptions regarding the sub-group size may be made across work-group barrier borders. Hence the current implementation stores all values to scratch memory, after which a work-group barrier is inserted. Thereafter, the first sub-group iterates over the scratch memory with a stride of *VF* and each work-item is offset from the index by the lane id. Every lane fills its own accumulator. This should lead to vectorized memory accesses and if the binary operation is trivial, also to vectorized arithmetic. As soon as the remaining items are less than *VF*, the sub-group reduction is executed on the *VF* accumulators once. The very first work-item then adds the remaining values and stores the final value back to scratch memory, where it is read from again by all work-items after another work-group barrier.

4.2.4 Comparison

LLVM's vectorizers are much more stable than RV. Using RV to compile the SYCL-Bench N-Body example [Lal et al., 2020], which uses `sycl::vec` which is partially mapped to vector values by the front-end, leads to a segmentation fault. In other sample applications, e.g. the k-nearest-neighbours from HeCBench [Jin, 2021], the use of the tested version of RV to vectorize the kernels, led to incorrect results.

For the default implementation of the group algorithms, such as group reduce, it was found that using a single work-item to perform the reduction was the most efficient approach. For RV this is a sub-optimal kernel design, as the hot loop is an inner-loop that does not benefit from RV's outer-loop vectorization, as only one lane ever enters the inner-loop. Combining RV and the LLVM inner-loop vectorizer leads to this case being vectorized very well: the general kernel is outer-loop vectorized by RV, whereas the inner accumulation loop is afterwards vectorized by LLVM's inner-loop vectorizer, making up for the shortcomings of this kernel design.

In general, when forcing a vectorization width, RV is able to vectorize all kernels that do not already contain vectorized instructions. However, this might lead to multiple scalarized instructions, potentially followed by pessimized performance. If RV deems vectorization of a marked loop possible, it performs the vectorization and does not check a cost model to verify whether vectorization is beneficial. In contrast, the LLVM loop vectorizer only transforms the kernels if the control flow is comparatively simple and bails without options to scalarize if e.g. a call to the `llvm.memcpy` intrinsic is encountered. Without the explicit forcing of vectorization using metadata, LLVM would check whether vectorization is actually beneficial, which for simplistic wi-loops or conditional memory accesses might not be the case.

A stable version of RV, i.e. an outer-loop vectorizer handling very diverse control

flows, integrated into LLVM would be worthwhile, especially in combination with VPlan, where the heuristically determined optimal vectorization strategy for each wi-loop will be selected.

The intrinsics provided by RV seem to be a good fit for the sub-group hierarchy level, as they offer low-level optimization opportunities with SPMD-like semantics. Although some extensions for better code generation might be necessary. By providing wrappers for reduction intrinsics or by offering a straightforward way to interact with the vector types and their newly exposed reduction intrinsics in Clang, sub-group algorithms could be highly optimized.

LLVM does not provide a way to force outer-loop vectorization nor related lane id queries, thus making a similar sub-group to vectorization mapping unfeasible. A flaw with the current RV intrinsics' implementation is that LLVM is not able to fully optimize code that is dependent on the intrinsics, as these are replaced only at vectorization time when most optimization passes have already run.

For the rather simplistic group_reduce example, the LLVM inner-loop vectorizer is most likely the better fit than using RV with intrinsics. The reduction loop that is executed by a single work-item is vectorized with an optimal vector width and is followed by the fitting add reduction intrinsics. Where the RV implementation had to manually take care of vector width vs work-group size checking, the LLVM vectorizer inserts these checks itself. RV interprets the check as a diverging branch, which is correct with regards to the wi-loop, but imprecise with regards to the actual vector width. Therefore, RV has to generate masked loads, which are much less efficient than contiguous loads.

The goal with using RV and its intrinsics is to allow manually writing of highly optimized kernels. In the case of the reduction, this would imply it should be possible to generate code equivalent to the work-group reduction vectorized by the LLVM loop vectorizer. As this is currently not possible, the RV intrinsics approach will require further optimization work.

5 Exploiting the single source nature of SYCL

SYCL is a single-source programming model. That is, the host application code and the kernel code, which is to be executed on the *device*, are in combined compilation units and can be built using a combined compilation driver. A specialized compiler might therefore optimize the kernels based on the surrounding host code, the invocation order and so on. Possible optimizations are propagation of constants, e.g. for the iteration space and thus potentially enabling full unrolling or removal of unnecessary bounds checks for loop rotation and vectorization. This optimization would analyze the construction of the nd-range specifying the iteration space for the kernel. If e.g. the work-group size is set to a constant value, this constant can be reused as a bound in the wi-loop and also to replace the query of the work-group size inside the user kernel, which should improve the quality of the generated code even further.

A major issue to take into account is that kernels are function objects that are instantiated at some point in the program and then usually stored in the runtime to be executed asynchronously from another thread. Therefore the compiler will have to match the kernel functor's point of submission, which there might be multiple of. For constant propagation, therefore either kernel versioning, i.e. effectively creating a kernel for each call site or parameter set, has to be used or optimizations to the kernel function are only done in case there is only one call site or parameter set.

As part of this thesis, only preliminary tests were conducted to evaluate the possible effect of constant iteration space propagation. In this case, the compiler did not have to identify the call sites and constants used there. Instead the proposed static kernel properties implementation by Aksel Alpay[1] was merged. The change allows wrapping a kernel functor `kernel` at submission site inside e.g. `sycl::attribute<sycl::reqd_work_group_size<16,16>>(kernel)`.

The type system of C++ then propagates the compile-time attribute, conveying the information that the kernel requires a work-group size of 16,16. Propagating this template argument to the kernel launcher function also effectively multi-versions the kernel function, which is inlined inside the kernel launcher function.

The compile-time information can be extracted at a C++ source level using template meta programming, to e.g. insert a check, that if the attribute is set, the run-time work-group size actually must equal the compile-time required size, otherwise the kernel is not executed and an error is emitted.

As the template arguments to the kernel launcher function are also retained in

[1] https://github.com/illuhad/hipSYCL/pull/543

its mangled name, the hipSYCL LLVM IR passes are able to extract the required work-group size for a kernel. This way, the effects of knowing the constant iteration space can be evaluated without actually conducting the much higher effort inter-procedural constant propagation.

For the hipSYCL nd-range kernels, there are three points of interest with a compile-time known work-group size. For the POCL and basic CBS pipelines, where a wi-loop is present and its structure is cloned, two of these optimizations can already be implemented effectively in the C++ kernel launcher function. These changes therefore are also applicable to the hierarchical and scoped parallelism execution models. If the work-group size is known at compile-time, the constexpr values are extracted and directly used as the bounds for the wi-loop, instead of the run-time work-group size parameter. This benefits the loop optimizations without further ado. The second point where this value is used, instead of the run-time parameter, is the construction of the local range, that is handed to the `nd_item`. After inlining all functions in the kernel and possibly running the SROA pass, the constant work-group size is directly used, wherever the local range is used in the user kernel, instead of loading a value from the `nd_item`.

The third point of optimization is specific to the loop fission approaches. Both POCL and CBS pipelines use stack arrays to store work-item private values over a barrier boundary. These stack arrays generally are comparatively large, 1024 elements, to allow execution of work-groups with that number of work-items. This might pessimize the caching of the stack, as the stack is rather large. To reduce this penalty for kernels, that are never executed with this large of a work-group size, the `reqd_work_group_size` attribute can be used. For this, the kernel function's mangled name is parsed to identify the constant values. The values for the multiple dimensions are multiplied, resulting in the total number of work-items and thus the required size of the stack arrays. This is then used whenever a new stack array is created in LLVM IR. The fallback still is 1024, if the attribute is not present.

In case the CBS pipeline is used in the mode that wi-loop has to be created, the constant values parsed from the kernel name, are also used as the wi-loop bounds. For this, it is first checked, whether the annotation can be found in the kernel name and if that is the case, three constant IR values are created and used from thereon. The identification of the local range function arguments is skipped in that case.

Using the attribute with work-group sizes that are a power of two, indeed eliminates the epilogue for all vectorized wi-loops. The performance impact of this is evaluated in chapter 6. Only with the performance impact evaluated, an informed decision about whether or not the effort for non-attribute based constant propagation would be justified.

6 Evaluation

The following pages present the experimental performance evaluation of the earlier outlined optimizations and variations. All benchmarks have been performed on systems provided by GW4 Isambard. The used systems are:

AMD Epyc 7442 "rome" 2x 2.25GHz 64-core, 128-threads Rome CPU, AVX2, 256GB DDR4-3200Mhz

Intel Xeon Gold 6338 "ilake" 2x 2.00GHz 32-core, 64-threads Icelake CPU, AVX512, 256GB

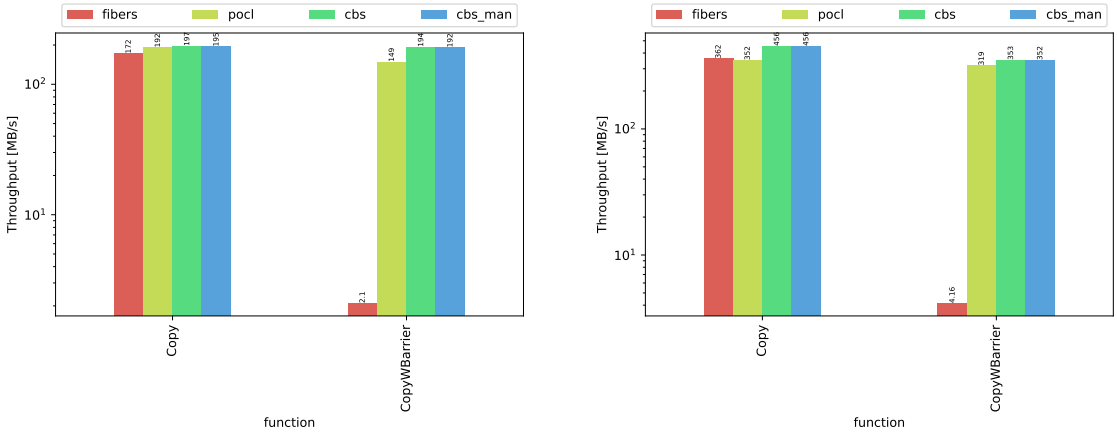
Marvell ThunderX2 "xci" ARM v8.1, 2x 2.1GHz 32-core, 128-threads CPU, NEON, 256GB DDR4-2666MHz

Fujitsu A64FX "a64fx" ARMv8.2, 1x 1.8GHz 48-core CPU, 512bit SVE, 32GB HBM2

For stable results, the physical core count is used as thread count `OMP_NUM_THREADS` and `OMP_PROC_BIND=true` is set. The only exception is Marvell ThunderX2, where using an SMT factor of 2, i.e. using 128 threads in total resulted in generally higher throughput. All benchmarks are repeated at least 10 times and the graphs show the maximum throughput achieved by any of those runs, if not otherwise noted. On the x86 systems, `-O3 -march=native` was used as default flags for Clang, whereas the AArch64 systems used `-O3 -mcpu=native`. On A64FX, additionally auto-vectorization using the scalable vector extension SVE is enabled by adding `-mllvm -scalable-vectorization=on`. Also, unless otherwise noted, a development version of LLVM 14[1] was used to take advantage of the SVE support, which still is in active development. Boost version 1.77 is used for the fiber support.

Although benchmark suites were used, like SYCL-Bench Lal et al. [2020] and HeCBench Jin [2021], only a small subset of both is relevant. Only those using nd-range parallel-for as execution paradigm have been considered, as this at the only paradigm, that changes have been made for. Even further reducing the number of benchmarks of interest, is the added requirement of using at least a single barrier in the kernels. The reason for this, can be seen in figure 6.1, which shows the results of a modified Copy kernel initially from BabelStream by Deakin et al. [2016]. The figure clearly shows that the performance of the fiber implementation is acceptable, as long as no barrier is present in the kernel. This is due to the at the beginning mentioned performance optimization in the fiber implementation, which loops over the work-items in a single thread until a barrier is encountered for the first time,

[1] https://github.com/llvm/llvm-project/commit/5352ea4a721ef252129994111b83dc350ecc71da



(a) BabelStream copy on Intel platform.

(b) BabelStream copy on AMD platform.

Figure 6.1: The Copy benchmark from BabelStream modified to use a nd-range kernel with a work-group size of 4. On the left, respectively, the plain copy kernel, on the right, with an added barrier after the actual copy.

then the additional fibers have to be spawned and synchronized at the barriers, which results in the initially mentioned performance issue.

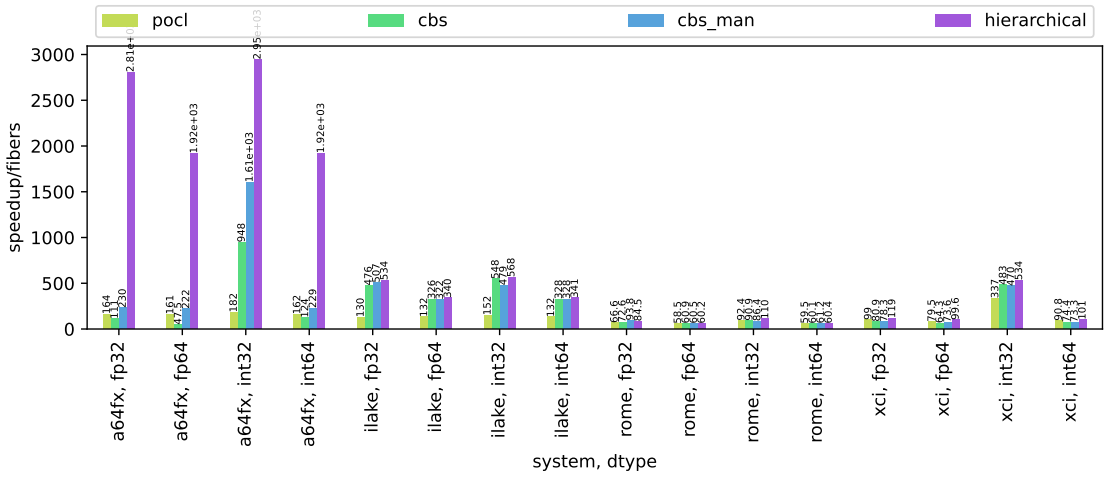
6.1 Pipelines

First of all, the performance of the currently used fiber implementation and the two loop fission pipelines, POCL and CBS, is evaluated using a number of existing benchmarks. Additionally, the CBS pipeline is evaluated once with already present wi-loop (cbs) and once with in the compiler created wi-loops (cbs_man). If an alternative implementation using SYCL's hierarchical parallelism exists, it is shown as a high-performance reference.

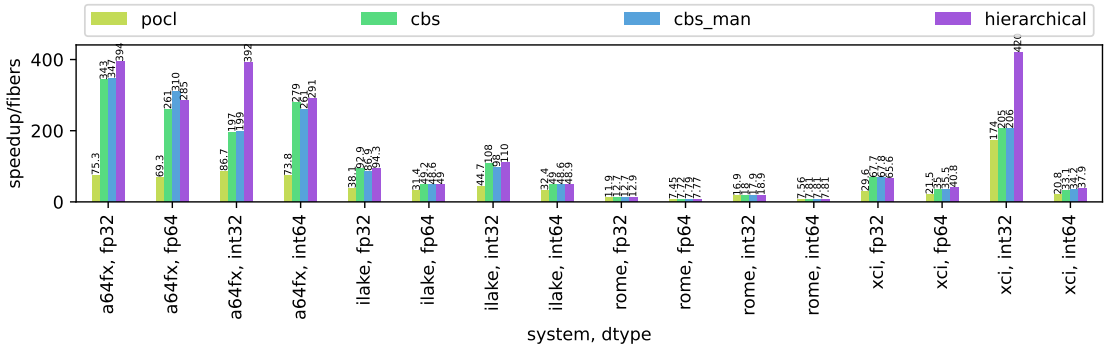
6.1.1 SYCL-Bench

The results of SYCL-Bench's reduction, scalar product, and N-Body benchmarks are presented in figure 6.2. For the reduction, all systems, data types, and implemented loop fission pipelines expose solid speedups of around 60-600x, i.e. at least one order of magnitude. On the A64FX platform, the throughput using fibers was notably at least an order of magnitude lower than on the other platforms. With this low base and the hierarchical reference implementation reaching similar throughput as the other platforms, a rather large speedup of up to 3000x is reached.

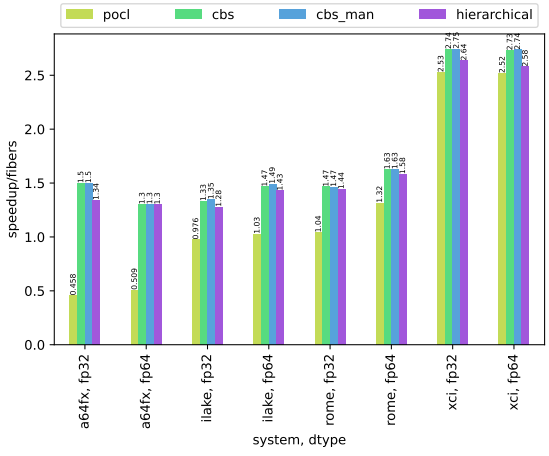
The order of magnitude gap between the loop fission approaches vs. hierarchical implementation on that platform can be explained by looking at the generated LLVM IR. The hierarchical version on the one hand has two large BBs that represent a vectorized wi-loop each and has an additional epilogue for work-group sizes that are



(a) Reduction kernel. 3072000 elements, work-group size 256.



(b) Scalar product kernel. 3072000 elements, work-group size 256.



(c) N-Body kernel. 30720 bodies, work-group size 256.

Figure 6.2: SYCL-Bench performance results. All graphs show the speedup of max throughput out of ten runs over the respective base fiber implementation.

not a multiple of the *VF*. On the other hand, the nd-range versions, except int32, do not use any vector instructions despite the presence of `vector.body` BBs and all loads and stores being in predicated BBs. The int32 version is partially vectorized, just exposing some more predication than the hierarchical version, which can also be seen from the speedup only being 2x lower for cbs_man than for hierarchical. Note that the lack of any vectorization for this kernel can only be observed on the A64FX platform. The LLVM vectorizer also emits vector instructions for the other AArch64 platform XCI. Therefore, it is likely that the immaturity of the scalable vectorization support in LLVM is the main reason for this.

The difference between the loop-fission pipelines is comparatively small, for most systems and combinations. Only the POCL pipeline falls short on the Icelake server system significantly, by at most a factor of 4x.

For the scalar product benchmark, the speedup is generally lower. The minimum is 7.5x and the maximum 425x. Similarly again, the A64FX fibers throughput is around one magnitude lower than the other systems', thus leading to more significant speedups. Note that the high speedup for int32 on XCI also is mainly caused by the fibers being an order of magnitude slower than for the other data types on that system. The hierarchical implementation again is either the fastest or close to the fastest variant, while the lack of performance with the POCL pipeline gains significance. For this benchmark, using the CBS variant with manually created loops seems beneficial, but not significantly so.

A different situation is found for the N-body simulation benchmark. Here the cbs and cbs_man variants constantly show the highest speedup, with the hierarchical implementation close by. This is only in the range of 1.3x to 2.75x faster than the fibers version. The POCL pipeline even falls short of the fibers implementation on both A64FX and Icelake but shows good performance on the ThunderX2.

The lower speedups are likely explainable by the use of `sycl::vec`, which, as discussed in section 4.1, already introduces some vectorized operations like load and store. This also applies to the fibers implementation, thus leading to vectorized code in the fibers version. The lack of which is part of the reasons for the usually bad fibers' performance. Another difference to the reduction and scalar product kernels from above is the position of the barriers. While the barrier in those other kernels is in the hot reduction loop, the N-body kernel has the barriers outside the innermost, compute-heavy loop. The possible partial vectorization of the innermost loop and due to the `sycl::vec` type as well as the comparatively fewer context switches in the kernel, therefore allow the fiber implementation to exhibit relatively good performance. As mentioned earlier, the performance of this could potentially be improved further for all variants by annotating the `sycl::vec` type with vector type hints.

It is notable that this is the only benchmark, where the hierarchical variant is consistently slower than the CBS variants. This is also the only benchmark using `sycl::private_memory`, which basically provides the equivalent to stack arrays that are required to keep work-item private state across wi-loops with the loop fission approaches. The hierarchical version allocates that memory un-aligned on the heap

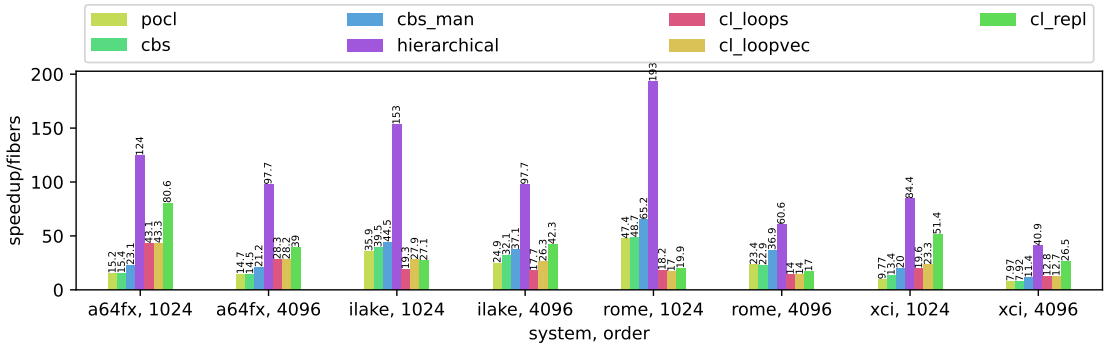


Figure 6.3: Blocked DGEMM implementation. Speedup of max throughput out of ten runs over base fiber version. *cl_** is an equivalent OpenCL version of the benchmark using POCL as driver.

instead of on the stack though, possibly pessimizing performance.

6.1.2 DGEMM

Another rather typical kernel in scientific computing using accelerators is DGEMM. In this case, the blocked SYCL DGEMM implementation from Deakin et al. [2021] is used as it also provides a hierarchical parallelism variant that is used as reference. Additionally, the SYCL code was ported directly to OpenCL. The *cl_** variants in figure 6.3 show the execution of that version using POCL. The latter part of the variants' name is the value of the `POCL_WORK_GROUP_METHOD` environment variable during execution. This variable directs POCL which wi-loop optimization mode to use. The effects of the values can be looked up in the documentation.[2] Additionally `POCL_MAX_PTHREAD_COUNT` is set to match the `OMP_NUM_THREADS` environment variable.

For this benchmark, it is rather notable that the hierarchical parallelism implementation performs significantly better than all other variants. This means the nd-range variants, as well as POCL, all leave some additional performance to be gained. While the general speedup factor for the nd-range variants is between almost 8x to 65x, the hierarchical version achieves up to 193x. Two major differences in the generated code can explain that performance gap. The first difference is that the initial load from global to scratch memory is converted to line-wise `llvm.memcpy` intrinsics, effectively eliminating the innermost work-item loop. Additionally, the innermost loop, calculating the actual multiplication, is vectorized with the hierarchical variant, while with cbs_man, the loop is fully unrolled.

Of the OpenCL implementation, the replication variant is consistently the fastest one, which stands for fully unrolling the wi-loops. This enables other optimization schemes, like the SLP-vectorizer. All hipSYCL pipelines fall short of the OpenCL

[2]http://portablecl.org/docs/html/env_variables.html

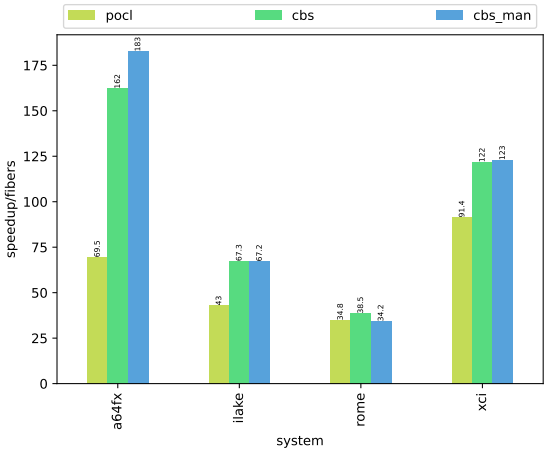


Figure 6.4: Wallace random number generator implementation. Generation of 33554432 random numbers. Speedup of max throughput out of 100 runs over base fiber version.

versions on the Arm CPUs. On the x86 platforms, only the repl variant is faster than the hipSYCL implementations, once. In addition to differences in the generated code, it is likely that the use of the more advanced OpenMP schedulers in comparison to manual usage of pthreads in POCL supports the performance uplift of SYCL over the POCL implementation on x86 systems. This is backed by Baumann et al. [2021], where it has been shown, that POCL's performance can be improved by using Intel's Threading Building Blocks (TBB) for scheduling.

Of the hipSYCL variants, the cbs_man is consistently the fastest, indicating that the initially scalar kernel function and requesting vectorization of the inner-loop only, improve the generated code.

6.1.3 Wallace's random number generator

The last benchmark is the implementation of Wallace's random number generator in HeCBench. The speedup over fibers ranges from 34x to 183x. The Arm platforms again have a relatively low baseline, thus exhibit higher speedups for all pipelines. The POCL pipeline is only able to match the CBS variants on AMD, but lacks some performance otherwise. Generally, the benchmark shows similar tendencies as the scalar product and reduction ones, thus confirming the general trend.

6.1.4 Summary

With these benchmarks, the trend is clear that the loop fission approaches are indeed effective at greatly increasing the performance of the nd-range parallel-for paradigm over the current fiber version. On the Arm systems, the fiber implementation generally performed even worse than on x86, allowing higher speedups using the compiler

extension. While the nd-range variants are not yet able to match the hierarchical implementation in most cases, they are in the same order of magnitude. The differences between the pipelines are expected, due to differences in applied optimizations. Part of the poor performance of the POCL pipeline most likely stems from not invoking SROA before the CFG transformation as discussed in 3.2.2 for the CBS pipeline.

The cbs and cbs_man variants in most cases have comparatively similar performance characteristics. As by earlier experimentation for Deakin et al. [2021] expected, the 2D DGEMM kernel performs consistently better with the cbs_man pipeline, which only forces vectorization of the innermost wi-loop, instead of collapsing the wi-loops. For the one-dimensional kernels, cbs_man also matched or outperformed the OpenMP SIMD loop version, thus making cbs_man the generally recommended configuration.

6.2 Vectorizer choice

As discussed in 4, multiple vectorization options were considered. The main question is, whether using a specialized vectorizer like the Region Vectorizer instead of the default LLVM vectorizers is necessary to achieve good performance.

The following evaluation relies on LLVM 12.0.1 with an additional bugfix cherry-picked[3] to enable using RV[4] as a second LLVM plugin in conjunction with hipSYCL. The CBS pipeline with manually created loops is used for all tests in this section. Despite not being the default pass manager in LLVM 12, the new pass manager is used for all benchmarks, as the CBS pipeline performs notably better in that case. This is due to the changes from section 3.2.2 only applying to the new pass manager.

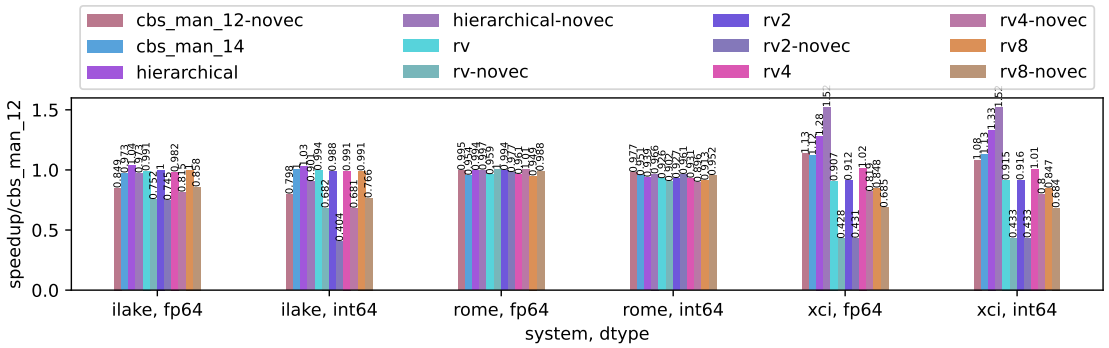
A64FX is not included here, as its main vector units are SVE units, which is not supported by the used RV version. The NEC fork specific to the SX-Aurora Vector Engine contains scalable vectorization support but relies on not yet upstreamed LLVM patches.

Whenever available, a hierarchical variant of the benchmark that is also compiled using LLVM 12 is added to the results. For benchmarks that were already presented in section 6.1, the cbs_man results are shown as cbs_man_14 for comparison purposes.

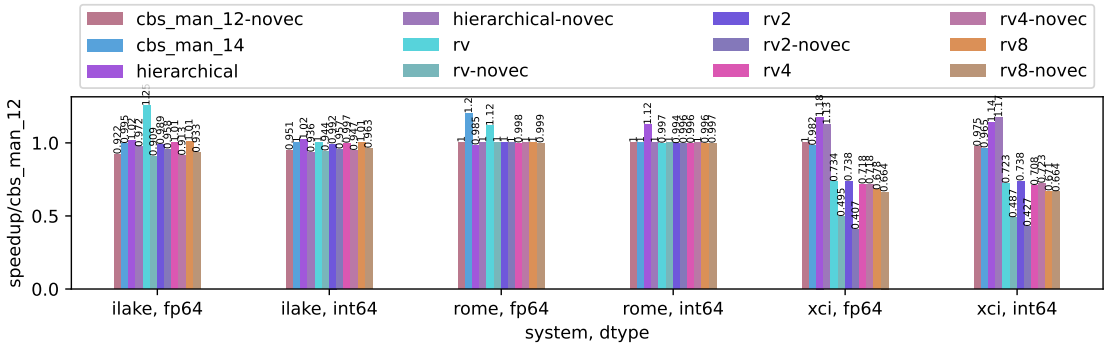
The variants using RV are named `rv`, if the vectorization factor is dynamically selected. Otherwise, the *VF* is appended to the name. To expose the individual contribution of the vectorizers, the benchmarks were run with both, LLVM's vectorizer enabled and disabled. The latter case is marked by appending `-novec`. The command-line options used to do so, are `-fno-slp-vectorize -fno-vectorize -mllvm -force-vector-width=1`. Note, these flags do not affect the behavior of RV.

[3] https://reviews.llvm.org/D104916

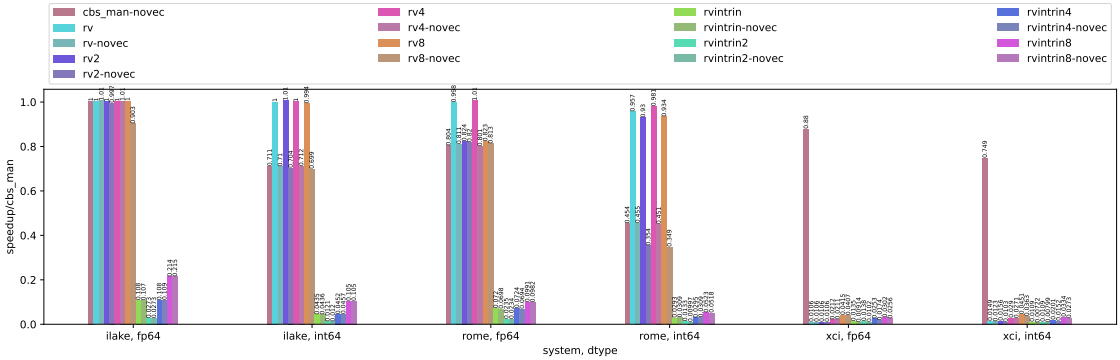
[4] https://github.com/fodinabor/rv/releases/tag/ma-thesis



(a) Reduction kernel. 3072000 elements, work-group size 256.



(b) Scalar product kernel. 3072000 elements, work-group size 256.



(c) Group reduce benchmark developed for Wünsche [2021]. 307200 elements, work-group size 256, 512 repetitions in the kernel. The rvintrin* variants use the `group_reduce` implementation with RV intrinsics detailed in section 4.2.3.

Figure 6.5: SYCL-Bench performance results using LLVM 12 + RV. All graphs show the speedup of max throughput out of ten runs over the respective base LLVM 12 -O3 CBS implementation with manually created loops.

6.2.1 SYCL-Bench

The SYCL-Bench results in figure 6.5 almost immediately make clear that just dropping in RV is not going to boost the performance by much, if at all.

The reduction results show that using RV alone, i.e. with the LLVM vectorizers disabled, leads to pessimized performance. Using RV in conjunction with the LLVM vectorizer, the performance stayed mostly within 10% of the LLVM 12 cbs_man reference. The notable exception is the ThunderX2 with a forced *VF* of 8. This is likely due to sub-optimal code generation as the ThunderX2 only supports vector widths up to 128bits.

Also interesting to note on the XCI platform is that the non-vectorized LLVM 12 CBS version is 8-13% faster than the vectorized version. Similar speedups are observable for the non-vectorized version of the hierarchical variant. This indicates that vectorization is not beneficial for this implementation of the pattern.

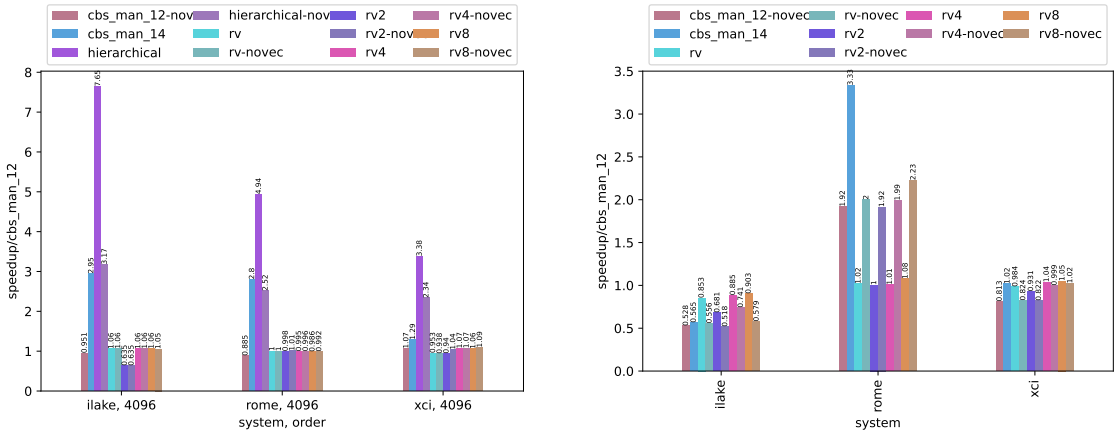
With the scalar product benchmark, the general trend is similar, that RV alone is not capable of matching LLVM's vectorizer performance. On the contrary, on the XCI system, it even pessimizes performance up to 60% without LLVM's vectorizer, and with both activated, the performance drops by a minimum of 26%. While disabling the LLVM vectorizer does not impact performance for the standard case at all, letting RV detect the *VF* on its own and leaving the LLVM vectorizer enabled improves the performance for the double-precision case on both AMD and Intel systems by 12-25%. Also notable is that the LLVM 14 version of the double-precision scalar product on AMD outperforms RV, with a 20% speedup over the LLVM 12 reference.

Group reduction

As detailed earlier, RV provides intrinsics which were used to construct a warp-reduce like group reduction. The results of which are shown in figure 6.5c as rv-intrin*. It is immediately visible that the RV intrinsics version is not at all able to compete with the already implemented `group_reduce`. In 4.2 it has already been discussed that the combination of RV's intrinsics would require more work to possibly provide viable optimization opportunities.

While there are multiple issues, e.g. the lane shuffle implementation having to rely on 32bit values instead of using possibly present 64bit instructions, the most pressing one for the intrinsics based group reduction is likely the misclassification of branches, that solely rely on the wi-loop induction variable and the `rv_lane_id` and `rv_num_lanes` intrinsics, as diverging branches. Even though the performance is on a low level for the intrinsics variants, they nicely show that the LLVM vectorizer does not interfere there at all and the effect of the forced *VF* is also well visible on Icelake and Rome.

Apart from the intrinsics version being performance-wise sub-optimal, the group reduction benchmark also shows how on Intel the floating-point case is not vectorized at all, as there the non-vectorized match the vectorized variants. For Intel and AMD



(a) Blocked DGEMM implementation. Speedup of max throughput out of ten runs over the respective base LLVM 12 -O3 CBS implementation with manually created loops.

(b) Wallace random number generator implementation. Generation of 33554432 random numbers. Speedup of max throughput out of 100 runs over the respective base LLVM 12 -O3 CBS implementation with manually created loops.

Figure 6.6: DGEMM and RNG Wallace performance results using LLVM 12 + RV.

otherwise, it's well visible that as predicted, the LLVM vectorizer performs the vectorization of the most important part of the group reduction: the accumulation loop, which is only reached by a single work-item. Using RV in conjunction with the LLVM vectorizer does not benefit performance, although the combination usually is able to match just using the LLVM vectorizer.

On the ThunderX2, RV does far more harm than help, leading to a performance regression of around 20-100x. In Moll and Hack [2018] a mobile Arm processor was used for evaluation. Plain RV did pessimize the performance there as well, therefore so-called BOSCC gadgets were used for runtime uniformity checks of conditions, which greatly helped the presented benchmark on that CPU. Using BOSCC with this benchmark did not impact performance at all. Given that it only checks whether branches are runtime uniform and thus can be skipped by all lanes and this benchmark not exposing control flow where this would be beneficial, this is expected.

6.2.2 DGEMM

DGEMM shows again that using LLVM 12 vs. LLVM 14 has far more impact than using RV with LLVM 12 or not. Despite this notable difference, RV also is able to consistently outperform the LLVM 12 vectorizer by a few percent on Icelake and XCI, while matching it on Rome. However, the not only vectorizer-related optimizations observed earlier that are done for hierarchical parallelism, seem to be much more effective at optimizing this pattern than a different vectorization strategy.

6.2.3 Wallace's random number generator

The HeCBench Wallace's random number generator results in figure 6.6b expose a diverse picture. On Icelake the LLVM 12 vectorization strategy is the best option, even the LLVM 14 version only achieves 57% of its performance. On the other extreme, is Rome, where the not LLVM-vectorized versions perform up to 2.23x better than with LLVM's vectorizer. Seemingly after LLVM 12 a performance issue with the vectorizer was fixed, as LLVM 14 performs 3.33x better than the LLVM 12 version. The XCI results for this are relatively favorable, as RV+LLVM achieved up to 5% better performance and only up to 7% performance drops for other *VF*s. With *VF* of 4 and 8, RV is able to match or even slightly outperform LLVM with its vectorizer disabled.

6.2.4 Summary

Despite the rather good match of RV for the SYCL kernel model, the experimental evaluation does not show stable enough performance benefits to generally enable it, at the moment. As a potential performance optimization left to the user, it might be mentionable. While the RV intrinsics resemble the CUDA warp intrinsics, the use of them currently does not benefit the tested reduction pattern due to missing optimizations in the analysis and intrinsic overload availability.

With the measured LLVM generational improvements, it is going to be interesting to see how the upcoming outer-loop vectorizer and the new VPlan infrastructure impact the kernels' performance. Given that for a number of kernels, the forced vectorization actually pessimized the performance, it should be considered to not add the `llvm.loop.vectorize.enable` metadata to all wi-loops. This allows the vectorizer to only vectorize the wi-loops if deemed beneficial. A static property could be added to allow adding this by the hipSYCL user.

6.3 Compile-time work-group size

The last point of evaluation is regarding whether propagating compile-time known work-group sizes is worth the effort. For this the benchmarks from 6.1 were re-run with the `sycl::reqd_work_group_size` attribute added to the kernel submission.

6.3.1 SYCL-Bench

In figure 6.7 the results are presented for the three standard SYCL-Bench benchmarks. Notably, A64FX is the only platform showing significant performance improvements for the reduction if the work-group size is compile-time known. For the N-Body simulation, the POCL pipeline on Icelake is the only variant that took advantage of knowing the wi-loop bounds at compile-time. The constant work-group size even deteriorates the performance or at least not change it significantly for the better, otherwise.

6.3.2 DGEMM

Figure 6.8 depicts the DGEMM results. In that plot, a similar situation as with the SYCL-Bench results is shown. While cbs with the annotation did not build successfully on A64FX, due to using the unstable PHI retaining variant, on the Arm systems, the cbs_man variant profits slightly from the known bound, but otherwise the performance is either not improved or even pessimized.

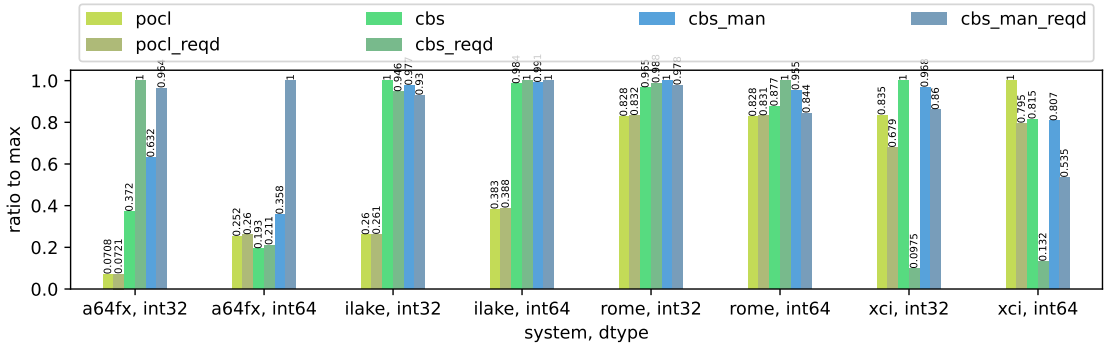
In addition to the base loop fission approaches missing several optimizations that are done for the hierarchical case on x86 as discussed above, using the constant iteration count makes the LLVM optimizer stop vectorizing the third wi-loop, which contains the backedge of the outer kernel loop and loads new values from global to scratch memory. On the other hand, a possible optimization can be observed there as well. The first wi-loop is fully unrolled and vectorized, but as it's executed only once, at the entry, it is not that performance-critical.

6.3.3 Wallace's random number generator

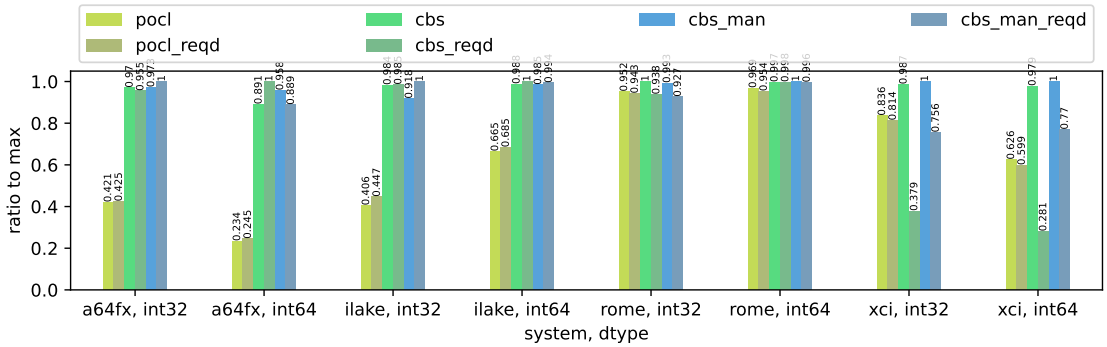
The random number generator results shown in figure 6.9 are even less conclusive. While the trend is that knowing the iteration count worsens the performance, the x86 platforms do not agree on that at all. The AMD results show that a constant work-group size might lead to a performance drop of up to 70%. On the other hand does the Intel system benefit from the constant bounds with all pipelines.

6.3.4 Summary

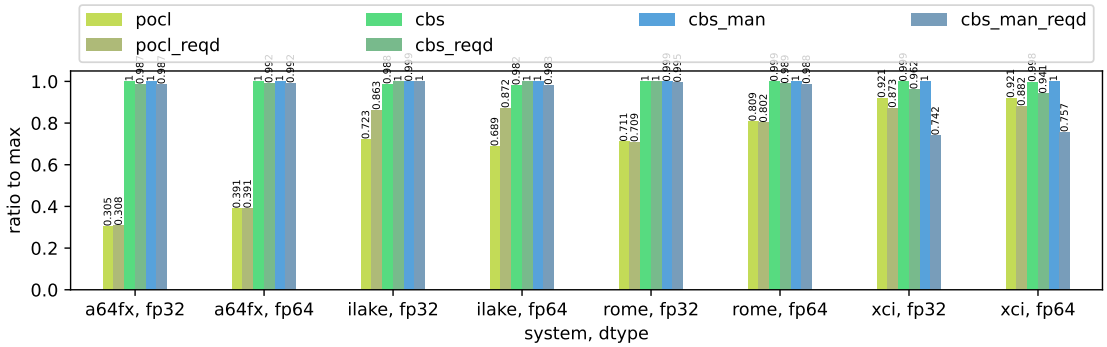
As the results differ greatly from great benefits to massive performance losses, with the same kernel performing unpredictably different across various targets, it seems safer to only expose the option to use compile-time known bounds. This is contrary to the initial guess that it would generally benefit performance to propagate constant work-group sizes. The effort required to implement an automated compiler-based version thus is currently not at all justifiable.



(a) Reduction kernel. 3072000 elements, work-group size 256.



(b) Scalar product kernel. 3072000 elements, work-group size 256.



(c) N-Body kernel. 30720 bodies, work-group size 256.

Figure 6.7: SYCL-Bench performance comparison between run-time vs compile-time known work-group size. All graphs show the ratio of max throughput out of ten runs scaled with respect to the maximum of that system and datatype combination.

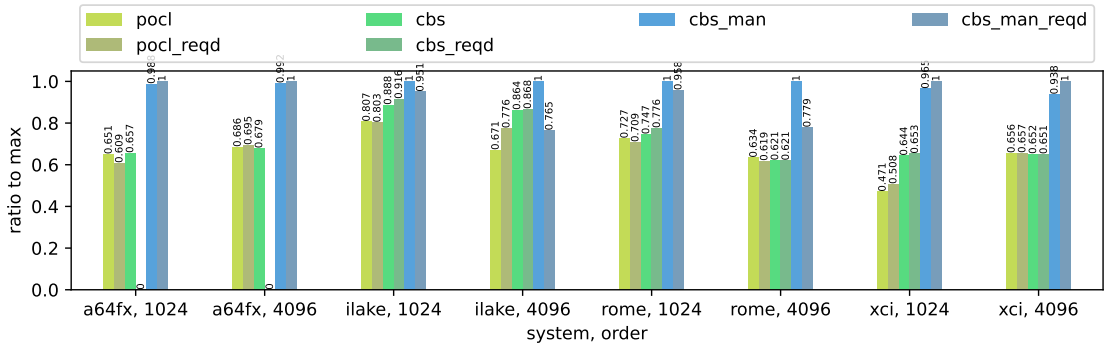


Figure 6.8: Blocked DGEMM performance comparison between run-time vs compile-time known work-group size. The graph shows the ratio of max throughput out of ten runs scaled with respect to the maximum of that system and order combination.

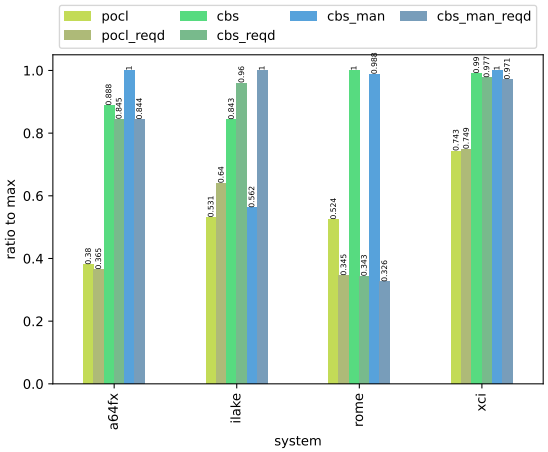


Figure 6.9: Wallace random number generator comparison between run-time vs compile-time known work-group size. Generation of 33554432 random numbers. The graph shows the ratio of max throughput out of ten runs scaled with respect to the maximum of that system and order combination.

7 Conclusion

This thesis aimed to improve the performance portability of the parallel SYCL execution models that were designed with GPUs in mind. To achieve this, two loop fission approaches were evaluated by implementing them in hipSYCL. The approaches were first presented in Jäskeläinen et al. [2010] and Karrenberg and Hack [2012]. The implementation details were presented and necessary optimizations were detailed. It was discussed that the approach POCL has chosen, allows for more hand-crafted compiler-based optimizations, whereas CBS is more general. One benefit of CBS is, it only requires that if a barrier is reached by any work-item, all work-items must reach it. This is how the barrier semantics are defined by the SYCL and newer OpenCL C standards. POCL additionally assumes that if a barrier is nested inside a loop, the loop has to be executed the exact same amount of iterations for all work-items, even if the barrier would not be reached. This can have a functional correctness impact.

Both approaches outperform hipSYCL's current fiber implementation in most cases by one or two orders of magnitude. While the CBS pipeline currently results in faster kernels in most cases, the POCL pipeline could close the gap by porting improvements made to the other one. By example of the DGEMM kernel, it was shown, that on the tested x86 systems, the SYCL implementation is able to outperform the POCL OpenCL implementation. This makes hipSYCL the first SYCL implementation that does not rely on an OpenCL runtime to achieve reasonable performance for the nd-range parallel-for paradigm on CPUs.

Two vectorization strategies were discussed. The first one relies on developers of data-parallel kernels to perform manual optimizations by using the `sycl::vec` class. The excessive use of this can even speed up the typically slow fiber implementation.

Apart from that, the SYCL parallel-for paradigm additionally exposes SPMD parallelism. Auto-vectorization of the work-item loops, which execute barrier-free sections of the kernel for each work-item, can thus be used to map the data-parallelism to the SIMD units of modern processors. Two auto-vectorizers were evaluated for this purpose: the standard LLVM loop vectorizer as well as the Region Vectorizer from Moll and Hack [2018]. Although RV is able to vectorize wi-loops with complex control flow, the performance with the current version may be heavily degraded or is only slightly better compared to LLVM's inner-loop vectorizer.

RV also supports intrinsics that can be used to map the SYCL sub-group hierarchical level to the SIMD units. This could enable writing manually optimized SIMD kernels written in a SPMD paradigm. While an exemplary working prototype implementation of the work-group `group_reduce` function was based on the intrinsics, it was not yet possible to achieve performance benefits over the regular variant.

Further extending the set of available intrinsics and integrating them deeper into the uniformity analysis might enable performance benefits in the long run.

Lastly, the impact of having a compile-time known work-group size was evaluated. It is used to override the size of the loop-state stack arrays required by the loop fission, compacting the stack size for potentially better caching effects. It also enables using constant loop bounds. Experimental evaluation shows that despite differing expectations, the performance was more often negatively impacted than positively. Therefore it can be considered a useful user-directed choice whether or not the work-group size should be compile-time known, as the user is able to measure the impact on their specific workload. For the cases with a negative impact, a deeper analysis of the applied optimizations could benefit the wider user base of the LLVM infrastructure.

For further improvements, the phase ordering problem should be considered. It might benefit the performance if the loop fission was executed later in the optimization pipeline so that the kernel is already further simplified before the inter wi-loop dependence analysis is performed. Additionally, the uniformity information that is used for deciding whether a value has to be stored in a stack array or whether a single value is enough, could be used for even more cases. Currently for example, recurring PHI nodes are not supported for the contiguous value restoration, for example. Manual optimization like full or partial unrolling of the wi-loop, as done in POCL, could be explored.

On the topic of vectorizers, the new VPlan infrastructure with its outer-loop vectorizer is worth evaluating, once the current compatibility issues are sorted out. Extending the Region Vectorizer with additional intrinsics and their integration into the uniformity analysis could improve its performance and open up opportunities for further experimentation with the sub-group to SIMD mapping.

Part I

Appendix

A Reproducibility

All development work done for this thesis is open-source. To improve reproducibility, the following describes the necessary steps to run the benchmarks.

First, a recent enough LLVM version is required. LLVM 12+ is encouraged. If LLVM 12 is chosen and RV shall be tested in conjunction with hipSYCL, the patch from https://reviews.llvm.org/D104916 has to be applied and LLVM built manually. When testing RV, clone https://github.com/fodinabor/rv/releases/tag/ma-thesis into the llvm-project folder and configure LLVM with RV as an external project.

The evaluated version of hipSYCL is https://github.com/fodinabor/hipSYCL/releases/tag/ma-thesis. Install it following the hipSYCL installation instructions. Note, the Clang plugin has to be built. Add `-DBUILD_CLANG_PLUGIN=ON` to the CMake configuration.

The benchmarks were run on the Isambard clusters. With all dependencies satisfied, the `run.sh` script from https://gitlab.com/fodinabor/ma-hipsycl-bench/-/tags/ma-thesis was used to run them. It also shows all flags used for the individual benchmarks. The repository additionally includes the raw results.

The versions of the used benchmarks are found at:

- SYCL-Bench https://github.com/fodinabor/sycl-bench/releases/tag/ma-thesis
- SYCL-Bench for RV https://github.com/fodinabor/sycl-bench/releases/tag/ma-thesis-rv
- SYCL DGEMM https://github.com/fodinabor/sycl_dgemm/releases/tag/ma-thesis
- HeCBench https://github.com/fodinabor/HeCBench/releases/tag/ma-thesis
- BabelStream https://github.com/fodinabor/BabelStream/releases/tag/ma-thesis

B Lists

B.1 List of Figures

B.2 List of Listings

C Bibliography

Aksel Alpay and Vincent Heuveline. SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL. In *Proceedings of the International Workshop on OpenCL*, pages 1–1, 2020.

Aksel Alpay and Vincent Heuveline. HipSYCL in 2021: Peculiarities, unique features and SYCL 2020. In *International Workshop on OpenCL*, IWOCL'21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390330. doi: 10.1145/3456669.3456691. URL https://doi.org/10.1145/3456669.3456691.

Alexey Bader, James Brodman, and Michael Kinsner. A sycl compiler and runtime architecture. In *Proceedings of the International Workshop on OpenCL*, pages 1–1, 2019.

Tobias Baumann, Matthias Noack, and Thomas Steinke. Performance evaluation and improvements of the pocl open-source opencl implementation on intel cpus. In *International Workshop on OpenCL*, IWOCL'21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390330. doi: 10.1145/3456669.3456698. URL https://doi.org/10.1145/3456669.3456698.

OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 2018. Version 5.0 November 2018.

Rod Burns. *Enabling OpenCL™ and SYCL™ for RISC-V® Processors*, 2020. https://www.codeplay.com/portal/videos/2021/06/17/enabling-opencl-and-sycl-for-risc-v-processors.html, last accessed on 17.11.21.

Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. Gpu-stream v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In Michela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, *High Performance Computing*, pages 489–507, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46079-6.

Tom Deakin, Simon N McIntosh-Smith, Aksel Alpay, and Vincent Heuveline. Benchmarking and extending sycl hierarchical parallelism. In *Workshop on Hierarchical Parallelism for Exascale Computing*, United States, November 2021. IEEE Computer Society.

The Khronos® OpenCL Working Group. *The OpenCL™ Specification*, 2021. Version 3.0.8, retrieved on 19.08.21 from https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

Jeff R Hammond and Timothy G Mattson. Evaluating data parallelism in C++ using the parallel research kernels. In *Proceedings of the International Workshop on OpenCL*, pages 1–6, 2019.

Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A performance-portable OpenCL implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.

Pekka O Jäskeläinen, S Carlos, Pablo Huerta, and Jarmo H Takala. OpenCL-based design methodology for application-specific processors. In *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 223–230. IEEE, 2010.

Zheming Jin. *HeCBench*, 2021. Version ba8310c1, https://github.com/zjin-lcf/HeCBench, last accessed on 07.11.21.

David Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. Chapter 8 - dissecting OpenCL on a heterogeneous system. In David Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang, editors, *Heterogeneous Computing with OpenCL 2.0*, pages 187–212. Morgan Kaufmann, Boston, 2015. ISBN 978-0-12-801414-1. doi: https://doi.org/10.1016/B978-0-12-801414-1.00008-9. URL https://www.sciencedirect.com/science/article/pii/B9780128014141000089.

Ralf Karrenberg. *Automatic SIMD vectorization of SSA-based control flow graphs*. Springer, 2015.

Ralf Karrenberg and Sebastian Hack. Whole Function Vectorization. In *International Symposium on Code Generation and Optimization*, CGO, 2011. doi: 10.1109/CGO.2011.5764682. URL http://www.cdl.uni-saarland.de/papers/karrenberg_wfv.pdf.

Ralf Karrenberg and Sebastian Hack. Improving performance of OpenCL on cpus. In *International Conference on Compiler Construction*, pages 1–20. Springer, 2012.

Ronan Keryell and Lin-Ya Yu. Early experiments using SYCL single-source modern C++ on Xilinx FPGA: Extended abstract of technical presentation. In *IWOCL '18: Proceedings of the International Workshop on OpenCL*, pages 1–8, 05 2018. ISBN 978-1-4503-6439-3. doi: 10.1145/3204919.3204937.

Sohan Lal, Aksel Alpay, Philip Salzmann, Biagio Cosenza, Nicolai Stawinoga, Peter Thoman, Thomas Fahringer, and Vincent Heuveline. SYCL-Bench: A versatile single-source benchmark suite for heterogeneous computing. In *Proceedings of the International Workshop on OpenCL*, IWOCL '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375313. doi: 10.1145/3388333.3388669. URL https://doi.org/10.1145/3388333.3388669.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society. ISBN 0769521029.

Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. On measuring the maturity of SYCL implementations by tracking historical performance improvements. In *International Workshop on OpenCL*, pages 1–13, 2021.

Charles Macfarlane. *Argonne and Oak Ridge National Laboratories Award Codeplay® Software to Further Strengthen SYCL™ Support Extending the Open Standard Software for AMD GPUs*, 2021. https://www.codeplay.com/portal/press-releases/2021/06/17/argonne-and-oak-ridge-national-laboratories-award-codeplay-software-to-further-strengthen-sycl.html, last accessed on 17.11.21.

Simon Moll and Sebastian Hack. Partial control-flow linearization. *SIGPLAN Not.*, 53(4):543–556, June 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192413. URL https://doi.org/10.1145/3296979.3192413.

Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: Revisited for short simd architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 2–11, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582825. doi: 10.1145/1454115.1454119. URL https://doi.org/10.1145/1454115.1454119.

Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara Chapman. A virtual GPU as developer-friendly OpenMP offload target. In *50th International Conference on Parallel Processing Workshop*, pages 1–7, 2021.

Chuck Pheatt. Intel® threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298, April 2008. ISSN 1937-4771.

Gil Rapaport and Ayal Zaks. Introducing vplan to the loop vectorizer. In *2017 European LLVM Developers Meeting*, 2017.

Ruyman Reyes. *Codeplay contribution to DPC++ brings SYCL support for NVIDIA GPUs*, 2020. https://www.codeplay.com/portal/news/2020/02/03/codeplay-contribution-to-dpcpp-brings-sycl-support-for-nvidia-gpus.html, last accessed on 17.11.21.

Julian Rosemann, Simon Moll, and Sebastian Hack. An abstract interpretation for spmd divergence on reducible control flow graphs. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434312. URL https://doi.org/10.1145/3434312.

Ira Rosen, Dorit Nuzman, and Ayal Zaks. Loop-aware slp in gcc. In *GCC Developers Summit*. Citeseer, 2007.

Naoki Shibata and Francesco Petrogalli. Sleef: A portable vectorized library of c standard mathematical functions. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1316–1327, 2020. doi: 10.1109/TPDS.2019.2960333.

John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core cpus. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 16–30, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-89740-8.

The Khronos® SYCL™ Working Group, Ronan Keryell, Lee Howes, and Maria Rovatsou. *SYCL™ 2020 Specification*, 2021. Version 2020 rev 3, retrieved on 16.05.21 from https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

Xinmin Tian, Hideki Saito, Ernesto Su, Jin Lin, Satish Guggilla, Diego Caballero, Matt Masten, Andrew Savonichev, Michael Rice, Elena Demikhovsky, et al. Llvm compiler implementation for explicit parallelization and simd vectorization. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–11, 2017.

Holger Wünsche. SYCL 2020 work group parallel primitives: Optimized algorithms for GPUs and CPUs in hipSYCL. Master's thesis, Heidelberg University, March 2021.

Erklärung:

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 29.11.2021 Joachim Meyer