Universität Heidelberg
Interdisciplinary Center for Scientific Computing
Engineering Mathematics and Computing Lab

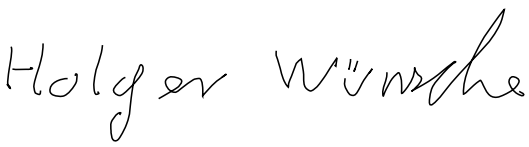
Bachelor Thesis

SYCL 2020 work group parallel primitives:

Optimized algorithms for GPUs and CPUs in hipSYCL

Name:	Holger Oliver Wünsche
Matriculation number:	3476005
Advisors:	Aksel Alpay, Prof. Dr. Vincent Heuveline
Date of submission:	March 15, 2021

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht.
Die Arbeit ist in gleicher oder vergleichbarer Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.



Abgabedatum: March 15, 2021

Zusammenfassung

In dieser Arbeit wurden die *group algorithms*, die in der vorläufigen SYCL 2020 Spezifikation beschrieben sind, implementiert und optimiert. Die Algorithmen werden kollektiv auf CPUs und auf NVIDIA und AMD GPUs ausgeführt. Die implementierten Funktionen umfassen *reductions*, inklusive und exklusive *scans*, Barrieren und *broadcasts*, die innerhalb sogenannten *work groups* oder *subgroups* ausgeführt werden. Der Quellcode ist in C++17, HIP und CUDA geschrieben, um die Funktionen der verschiedenen Architekturen vollständig ausnutzen zu können. Für die Optimierung wurden unter anderem Vektor-, *shuffle*- und *DPP*-Instruktionen verwendet. Um die Effektivität von Änderungen festzustellen, wurde sycl-bench um Benchmarks für die neuen Funktionen erweitert und die Ergebnisse mit den Bibliotheken CUB und rocPRIM verglichen. Die Laufzeit fast aller Funktionen konnte um einen Faktor von 1.5 bis fast 9 gegenüber der initialen Implementation verkürzt werden. Auf AMD GPUs konnte sogar ein Speedup von 33x erreicht werden. DPP Instruktionen auf AMD GPUs haben bereits gute Beschleunigungen erreicht und zeigen noch mehr Potenzial die Laufzeit weiter zu verkürzen.

Abstract

In this thesis group algorithms, as specified in the SYCL 2020 provisional specification, were implemented and optimized. The algorithms run cooperatively on CPUs as well as NVIDIA and AMD GPUs. The implemented functions consist of reductions, inclusive and exclusive scans, barriers and broadcasts executed on so called *work groups* or *subgroups*. The source code is written in C++17, HIP and CUDA to be able to fully utilize the functionalities of each architecture. As optimizations, vectorization as well as *shuffle*- and *DPP*-instructions are used. To ascertain the effectiveness of our changes we extended sycl-bench to also contain benchmarks for the new functions. Further we compared our implementation to the libraries CUB and rocPRIM. Overall, the runtime could be reduced for almost all functions by a factor between 1.5 and almost 9 with respect to our initial implementations. On AMD GPUs a speedup of 33x could be obtained. DPP instructions in particular enabled significant speedups and have some potential for further runtime reductions.

Contents

1. Introduction

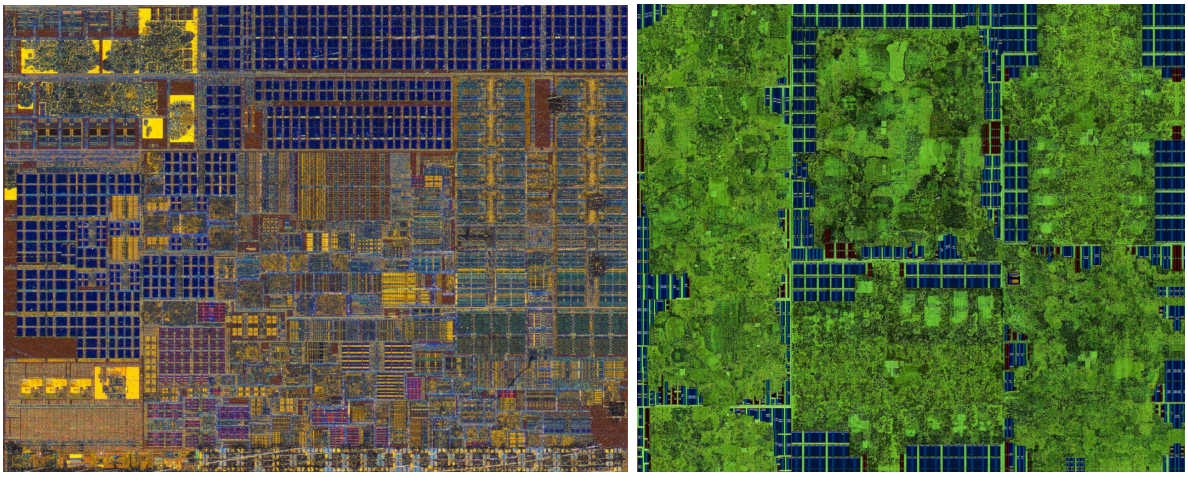
In 1974, a paper by Dennard et al. proposes a scaling law for MOSFET, stating that increasing the transistor density of integrated circuits will increase speed proportionally, while the power density stays constant [17]. This observation was later called Dennard scaling. Classic Dennard scaling stopped at a feature size of approximately 130 nm, further reduction in transistor size lead to degraded performance. Through the use of strained silicon layers, in which the distance between silicon atoms is increased, and metals with high dielectric constant, performance could be achieved at even lower feature sizes [31]. Around 2006 Dennard scaling could not be achieved anymore, because the power density did not remain constant due to increased leakage currents for smaller transistors, resulting in higher power consumption limiting the possible speedups [10]. Further performance improvements were achieved using processors with multiple, more energy-efficient cores. To take advantage of multiple cores, applications had to be redesigned. The work has to be split up into multiple sections, that can run in parallel [20]. Like single core processors, multicore designs are also power limited [18].

To improve energy efficiency, specialized processors can be used [36]. As a result almost every second High-Performance Computing (HPC) cluster in the first 100 of the November 2020 TOP500 list[1] contains accelerators today, most of which are Graphics Processing Units (GPUs) [59].

GPUs were created to speed up graphics. To this end, they contain many lower frequency cores in comparison to CPUs. The architecture is optimized for high throughput, executing massively parallel programs [46]. Consequently GPUs contain fast memory, many simple cores and little cache. This is in contrast to CPUs, which contain few complex cores with large caches to achieve low latency and high speed for a single task [33].

The theoretical speedup in parallel computing can be calculated using Amdahl's law. According to Amdahl the speedup is less or equal to $(r_s + \frac{r_p}{n})^{-1}$ with $r_s + r_p = 1$ and r_s representing the ratio of sequential portions, and r_p the ratio of parallel portions of the program [6]. This shows: The sequential portion of the program needs to be small to get good speedups using many cores. GPUs were designed to be latency tolerant and are able to switch to different computations, for example other pixels, when long latencies

[1] a list ranking the fastest super computers



(a) single core of an Intel I7-7820X Central Processing Unit (CPU) [19][2]

(b) single compute unit of an NVIDIA RTX 2080 GPU [19][3]

Figure 1.1.: The CPU core contains many more different structures, than the compute unit on the GPU.

are encountered. This reduces the sequential portions of programs through overlap [33].

Using GPUs for general purpose computing started around 2007, when GPU performance increased [46] and General Purpose GPU (GPGPU) programming models and SDKs where released (for example NVIDIA CUDA [41] and ATI Stream). In 2009 the open standard OpenCL [38] maintained by the Khronos Group was released providing a framework for programs executing on heterogeneous platforms including CPUs and GPUs [38]. Over the years multiple other accelerators where created, like the Intel Xeon Phi [13], Field Programmable Gate Arrays (FPGAs) or the NEC SX-Aurora TSUBASA vector engines [63].

Many HPC applications, like GROMACS [22], VASP [54] or Ansys Fluent [7], provide CUDA backends to enable the use of NVIDIA GPUs. Since CUDA is the proprietary language from NVIDIA, the code utilizing CUDA can only be executed on NVIDIA GPUs. In contrast, OpenCL and SYCL are both open standards developed by the Khronos Group. Multiple implementations exist, for example Codeplay's ComputeCpp [14] or hipSYCL developed at Heidelberg University [2]. Together the different implementations support architectures, like NVIDIA, AMD and in future also Intel GPUs, as well as CPUs. To a limited degree compilation is also possible for FPGAs [21].

Writing code for heterogeneous architectures with varying performance characteristics is complex [47]. A single programming model has the additional advantage of providing an interface for a set of core functionality, supported on all hardware. When supporting

[2]section of `https://live.staticflickr.com/4610/39024590565_8172128767_o_d.jpg`
[3]section of `https://live.staticflickr.com/65535/50867163908_1832695846_o_d.jpg`

different architectures, the complexity can be reduced by using the same language for all devices, because the already correctly functioning code for one architecture can be used as a starting point [56].

The difficulty of development poses a significant barrier to the usage of new architectures [34]. Since SYCL code is written in pure C++, which means without any language extensions, it is easy to integrate into existing projects [1, 49]. Additionally, C++ is often used in HPC applications [4]. This lowers the development effort and can increase efficiency [37].

Before explaining the group algorithms introduced in the SYCL 2020 provisional specification [49], an overview over all relevant topics is given. It is structured in a way, that sections are mostly self-contained and can be skipped in case the topic or concept is already known to the reader. The sections 1.2, 1.3 and 1.5 provide further motivation and explain the most fundamental concepts for this work. In section 1.7 the implemented algorithms are presented.

1.1. Heterogeneous Computing

Heterogeneous computing refers to computations using multiple different architectures [12]. In the context of HPC, the processors are commonly CPUs and GPUs. Accelerators, like GPUs, can deliver significantly improved performance and energy efficiency, if the hardware is suited for certain applications. This comes at the cost of increased complexity when designing the software [9]. This trade-off is what makes heterogeneous computing interesting in HPC applications, because energy efficiency and performance are important design constraints [44].

The different architectures of accelerators can require specialized programming languages. Some approaches exist, like *OpenMP offloading*, which allow to run existing code on GPUs. There is also research for offloading to FPGAs [55]. To use OpenMP offloading, OpenMP directives are used inside normal C++ or Fortran code. While OpenMP offloading can increase performance, programs written, for example, in CUDA can be significantly faster. The increased performance is possible, because the programmer has more control and can employ low-level optimizations, resulting in more complex code [15].

OpenCL and SYCL attempt to allow developers to write code for CPUs and accelerators using a single language. While the target device still needs to be considered to get high-performance code [30], the advantage of using one language for all architectures is the portability between different systems. In general both languages, OpenCL and SYLC, can leverage close to peak hardware performance, despite their portability [16].

1.2. GPU Execution Model

This section is based on architecture descriptions and programming guides from AMD and NVIDIA [5, 26, 40, 43, 51].

GPUs implement multiple compute units performing computations using Single Instruction Mulitple Data (SIMD) units. *Local memory*[4], which is low-latency high-bandwidth memory[5], can be used for communication inside a compute unit. The capacity of local memory is usually less than 128 kB. *Global memory*[6] is the main memory of the GPU. Modern server GPUs are usually equipped with between 32 and 64 GB. Accesses to memory are coalesced into transactions of a fixed size. This makes continuous accesses more efficient, since fewer transactions are required.

The memory consistency and coherency guarantees on GPUs are weak compared to CPUs [3], as a result communication between compute units is generally not possible and communication inside a compute unit requires synchronization, to guarantee all change to the local memory are visible. On AMD and NVIDIA GPUs, communication between SIMD lanes can be done using so-called shuffle instructions, which allow exchanging data similar to SIMD cross-lane instructions.

To make the best use of the hardware, it can make sense, to launch more work groups than available processors on the GPU. This allows to overlap memory accesses and computation, by switching between different subgroups running on the same compute unit. In contrast to CPUs, context switching has low overhead on GPUs [40]. The occupancy, which is the ratio of active computations to the maximum number of active computations on a compute unit, is limited by resource usage like registers and local memory, and determines the maximum amount of possible overlap.

CPUs are optimized for a wide variety of tasks with low latency. Many improvements to the architecture, such as branch prediction and out-of-order execution, reduce latency even more. In contrast, GPUs are designed to be highly latency tolerant by quickly switching between tasks to allow better utilization of the given hardware. Hiding execution stalls enables the high throughput on GPUs.

1.3. SYCL

SYCL is a system-wide programming model developed by the Khronos Group. In this section, an overview is given, based on the SYCL standard [58]. The terminology used

[4] on CUDA this is called shared memory
[5] each bank of the usual 32 or 64 banks of local memory can usually supply 32B per clock cycle
[6] with a bandwidth of approximately 900 GB s^{-1} or higher for *HBM2*

by the SYCL execution model will be used throughout the thesis to refer to the different components and concepts.

SYCL 1.2.1 was released in 2018 based on C++11, with the aim to allow parallel heterogeneous programming through the use of modern C++ and open standards. It used OpenCL for target support and included template and lambda-functions. The SYCL runtime implicitly manages data movement and kernel launches for the developer using a dependency graph.

In early 2021, SYCL 2020 was release now based on C++17 introducing Unified Shared Memory (USM) and build in parallel reduction operations. It also adds work group and subgroup algorithms[7]. Since SYCL 2020, it is also allowed to use other frameworks than OpenCL for target support, while still being compliant with the standard.

SYCL follows a Single Instruction Mulitple Threads (SIMT) model, allows multiple target architectures and is single-source. Single-source means the code for the host[8] and the device is parsed together by the compiler. As a consequence the compiler can ensure matching data types on the interface between host and device code. This could not be done automatically, if the code would be parsed separately.

Because it is an open standard, there are multiple implementations like Codeplay's ComputeCpp [14], Intel's Data Parallel C++ [24], Xilinx triSYCL [61] and hipSYCL developed at Heidelberg University [2]. There is also an additional implementation, that is not yet public at the time of writing called neoSYCL [28].

1.3.1. Execution Model

The SYCL execution model identifies six managed resources:

- Platforms: Multiple platforms can be present and represent the devices accessible through a vendor's runtime.
- Contexts: A context can contain multiple devices. All devices inside the same context must be able to access the global memory of the other devices and belong to the same platform.
- Devices: Hardware that can execute SYCL kernels.
- Kernels: SYCL functions that are executed on SYCL devices. They are defined as C++ function objects.

[7] implemented in this work
[8] the system executing C++ applications including the SYCL API

- Modules: Modules represent a set of kernels, which can be executed by devices inside a context.
- Queues: A queue references a context, platform and device. It executes kernels on a single device, but can also have dependencies from any other device.

Kernels are launched based on data dependencies specified by the developer. Using the dependencies, a directed acyclic dependency graph is generated at runtime. The SYCL runtime guarantees, that kernels are launched in an order satisfying the dependencies. The runtime can also use the dependency graph to overlap memory movement and computation.

According to the SYCL execution model, once a kernel is launched, an index space is defined. For each point in the index space an instance, called *work item*, of the kernel code is executed. All work items execute the same code, but are able to follow different branches and act on different data based on the id of the work item.

The work items are organized into *work groups* executing on a single compute unit. Each work group has a unique id and each work item is assigned a unique local id inside the work group. Work items inside a work group executed concurrently are called *subgroup* and can make independent progress with respect to other subgroups.

All work items can be uniquely identified using their global id, corresponding to a point in the index space, or a combination of the work group id and the local id.

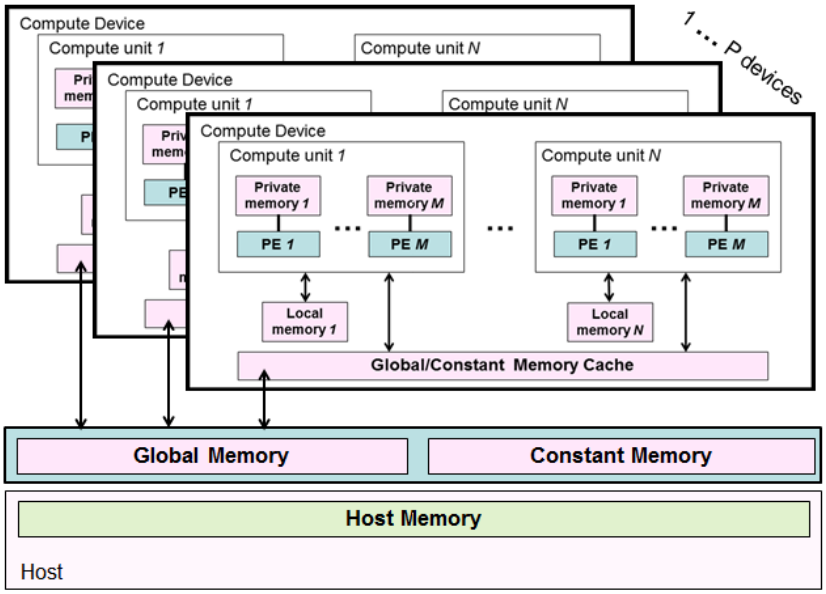


Figure 1.2.: An overview over the memory hierarchy [23][9].

[9]image from https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#_fundamental_memory_regions

The N-dimensional index space containing all work items of a kernel launch is called *nd-range*.

Work items have access to the following memory regions, also shown in Figure 1.2:

- *Global memory* is persistent across kernel invocations and accessible by all work items. However, there is no guarantee that two concurrent writes result in correct results.
- *Constant memory* is a region of global memory, that remains unchanged during kernel execution and is initialized by the host.
- *Local memory* is accessible by all work items in the same work group.
- *Private memory* is private to each work item and not accessible by other work items.
- *Generic memory* is a virtual address space across global, local and private memory.

1.3.2. Programming Model

A SYCL application has three different scopes:

- The *kernel scope* defines a single kernel function and will be executed on the device. In the example in Figure 1.3, this corresponds to lines 21-22.
- The *command group scope* specifies a kernel function and accessors. It is used to define a unit of work. Lines 17-24 inside the example correspond to the command group scope.
- The *application scope* is everything outside of command group scopes

Data accessed in a kernel needs to be inside a buffer and is accessed using accessors. Alternatively USM allocations can be used as an alternative to buffers. USM is based on pointers and does not need accessor, but in some cases explicit memory copies are needed. A buffer defines managed data regions, while the accessor is used to track the dependencies between kernels and access data inside a buffer. The dependencies are derived, by tracking all accessors and the access mode[10].

In the example in Figure 1.3 the accessor has the arguments `write_only` and `no_init`, which tells the runtime, the kernel only writes to the buffer `buff_a` and does not need the data to be initialized. Once the buffer is destroyed,the data will be copied back to the

[10]important modes are: read, write, read and write

```
1  #include <sycl/sycl.hpp>
2
3  int main()
4  {
5    sycl::queue q;
6
7    size_t n = 100000;
8
9    double *h_a;
10
11   size_t bytes = n*sizeof(double);
12   h_a = (double*)malloc(bytes);
13
14   sycl::range<1> work_items{n};
15
16   {
17     sycl::buffer<double> buff_a(h_a, n);
18     q.submit([&](sycl::handler& cgh){
19         sycl::accessor access_a { buff_a, cgh, write_only, no_init };
20         cgh.parallel_for(work_items, [=] (sycl::id<1> tid) {
21             if (tid.get(0) < n)
22               access_a[tid] = 0;
23         });
24     });
25   }
26
27   free(h_a);
28   return 0;
29 }
```

Figure 1.3.: A simple SYCL program for filling an array with zeros. The kernel is implemented as a lambda function in line 20-23 and there is no explicit memory movement. Instead of specifying the copy operation from the device to the host, this is done automatically using the buffer and accessor definitions.

host. The runtime can overlap kernel and copy of data, if the dependencies allow this. In other presented GPGPU languages, the order of copies and kernel launches must be managed by the developer. This is one of the big advantages of SYCL.

There are three execution models for kernels in SYCL.

The first is used for basic kernels launched using `sycl::parallel_for` with `sycl::range` as parameter. For these only the range of the nd-range is specified. Inside these kernels, synchronizations are not valid and the work group size is decided by the runtime.

The second one is for nd-range kernels, launched using `sycl::parallel_for` with `sycl ::nd_range` as argument. In this case the developer defines the dimensions of the work groups. In contrast to basic kernels, inside nd-range kernels synchronization is allowed and local memory can be accessed.

Kernels can also be started using the temporarily discouraged hierarchical interface with `sycl::parallel_for_work_group`. This way code is executed once per work group. Inside work groups, code for work items is written using `sycl::parallel_for_work_item`. Group algorithms are not allowed inside hierarchical kernels.

1.3.3. Implementation Options

The specification proposes three different possible implementation approaches.

The first is *Single-source Multiple Compiler Passes (SCMP)*, this technique uses a separate host and device compiler. First the host compiler parses the code and identifies SYCL kernels and all functions executed on the device. The identification of kernel code is called *kernel outlining*. The device compiler only needs to generate the code required by the device. The communication between the compilers can be achieved using generated header files. These header files can be used as interface between the host compiler and the SYCL runtime. To identify the kernels in the code that invokes it, the kernel name can be used. If the implementation supports it, a kernel name can be omitted.

The second possible implementation is *single-source single compiler pass*. It behaves similar to the SCMP approach, but instead of using multiple compilers, only one compiler parses the code and generates the host and device code.

The third option is to implement SYCL purely as a library. This would allow to compile SYCL code with off-the-shelf compiler.

1.4. Other Programming Languages

There are many programming languages with varying levels of hardware support. In this section, the most relevant languages, for this thesis, are presented and compared to provide further motivation.

Code for the CPU is sometimes referred to as host code, while code for the accelerator is often called device code.

1.4.1. OpenMP

OpenMP extends C++, C and Fortran using compiler directives. OpenMP 4.0 includes directives to offload code to another devices [45]. This allows to run legacy code on a GPU using only OpenMP directives and a compiler supporting OpenMP and offloading to the desired target. The supported targets include GPUs. There is also experimental support for FPGAs [15, 55]. To achieve the best results, in most cases, the code needs to be altered [15, 29]. Although explicit memory management is possible, it does not allow the level of control other languages in this section provide. Nevertheless, OpenMP offloading is able to provide significant speedups, compared to execution on CPU for some problems [35].

1.4.2. CUDA

CUDA is the platform and Application Programming Interface (API) created by NVIDIA to write GPGPU code targeting their CUDA-enabled GPUs. It can be used inside C, C++ and Fortran.

CUDA is single-source and introduces syntax extensions, for example to launch a kernel. A simple example showing how an array could be filled with zeros on the GPU can be found in Figure A.1.

The device code follows the SIMT model. This means the code is written for a single work item and multiple work items are executed in lockstep on SIMD units. To execute code on the device, it needs to be marked using attributes like `__global__` or `__device__`, because CUDA does not support automatic kernel outlining. For launching the kernel[11], launch parameters need to be provided using the syntax `kernel_name<<<...>>>` not present in C++. In addition, some built-ins are available inside the device code, for example to determine the thread id. These are often used to assign the data to the work items.

Features like queues[12], access to local memory and low level device instructions exist. CUDA also offers an established and mature ecosystem.

1.4.3. HIP

HIP (Heterogeneous-compute Interface for Portability) is developed by AMD and part of Radeon Open Compute (ROCm) [50]. HIP is a single-source C++ dialect like CUDA and follows the same programming model. The difference between HIP and CUDA is the name of runtime functions and some hardware specific features. HIP code can be compiled for AMD GPUs and CUDA enabled devices [8].

1.4.4. OpenCL

OpenCL is an open standard developed by the Khronos Group, targeting CPU, GPU, FPGA and Digital Signal Processors (DSP) based on C99 and C++11. There are multiple implementations and a wide range of supported devices from different vendors. There is for example the Intel SDK for OpenCL Applications [25], the NVIDIA OpenCL SDK [42] or AMD ROCm OpenCL [50].

In contrast to the other languages presented here, OpenCL is not single-source. Simple kernels can be written by writing them into separate files or inside a string, as can be seen

[11] This means submitting the code for execution on the GPU.
[12] to be able to have memory movement and computation in parallel

in the OpenCL example in Figure A.2. Because OpenCL is single-source, the compiler cannot validate the interface between host and device code.

The low-level nature of OpenCL results in accusations of being overly verbose [60]. In contrast to the other languages, the setup and tear down code cannot be omitted, which is a major factor for the verbosity of the model. It should be noted that steps like queue creation and device selection are also needed in larger CUDA or HIP programs. Wrappers exist which make developing in OpenCL easier, by handling most of the setup and cleanup [57].

1.5. HipSYCL

HipSYCL is a SYCL implementation developed at Heidelberg University [2]. It targets CPUs, as well as NVIDIA and AMD GPUs, leveraging existing toolchains. For the different targets OpenMP, CUDA and HIP/ROCm are used. HipSYCL also has highly experimental support for Intel GPUs utilizing SPIR-V code, since the backend is still in very early stages of development, it is not further discussed in this thesis.

1.5.1. Compiler Model

HipSYCL supports all three proposed implementation methods described in subsection 1.3.3, depending on the combination of targeted hardware.

HipSYCL can be used as pure library, when compiling only for CPUs. In this case any compiler supporting C++17 and OpenMP can be used.

The compilation for GPUs requires clang [48] and the hipSYCL clang plugin. This plugin augments the HIP and CUDA frontend to understand SYCL code. To this end, the plugin identifies kernels and functions executed on the device and attaches the required HIP and CUDA attributes.

To compile SYCL code using hipSYCL, a python script called `syclcc` provides a uniform interface for all supported backends.

There are two compilation modes for hipSYCL:

Integrated multipass, in this mode host and device compilation is handled by clang's CUDA and HIP drivers and corresponds to single-source single compiler pass. This mode provides the most interoperability, because backend-specific language extensions, like CUDA's syntax to launch kernels, are also available in the host pass. However this mode is affected by limitations in clang's compilation drivers. It is for example not possible to compile for both AMD and CUDA GPUs at the same time using this mode.

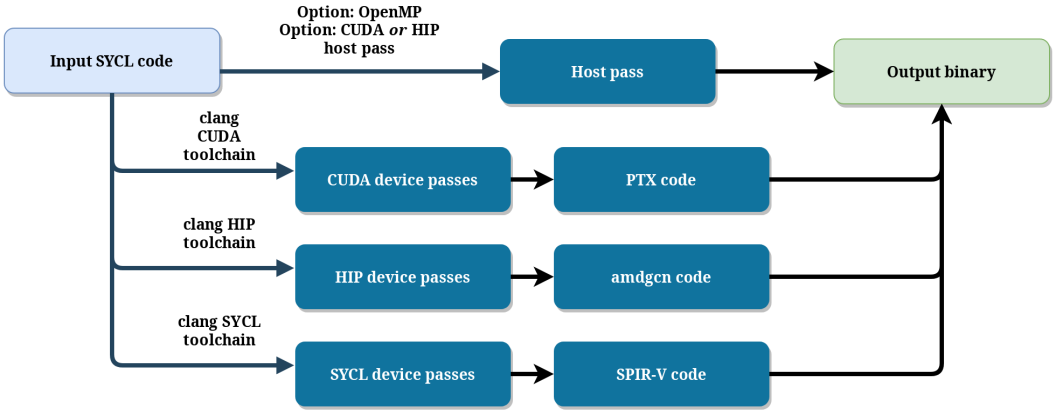


Figure 1.4.: Illustration of the hipSYCL compilation process [2][13].

In *Explicit multipass*, the compilation passes are handled by `syclcc`. This approach corresponds to SCMP. `syclcc` invokes the required device passes and embeds the compiled kernel images into the host binary. Because the compilation passes can be executed independently, hipSYCL can target multiple backends at the same time.

To mitigate the limitations of integrated multipass, CUDA can also be compiled using *explicit multipass*. Using both multipass methods hipSYCL can target arbitrary supported backends at the same time.

When compiling a program using hipSYCL, it is possible to write code only for accelerators or to write code directly in CUDA and HIP by checking if certain macros exist. This works, because in each compiler pass a set of macros is defined. Figure A.3 shows how to write specific code for the host or device, as well as adding CUDA or HIP sections. This ability to write code directly in the target language, allows the use of language specific low-level intrinsics and macros and even the ability to write assembler code for the target architecture. This feature was heavily used to implement the group functions. In this way most functions targeting subgroups where implemented in CUDA, HIP and C++, while most functions on work groups were written in such a way, that the same code works for NVIDIA and AMD GPUs.

1.5.2. Execution Model

Mapping of work items, subgroups and work groups, as well as memory, is done according to Table 1.1. On GPU the model matches the given hardware, because the hardware is designed to work with models, similar to the SYCL model. On CPU some concepts do

[13] `https://raw.githubusercontent.com/illuhad/hipSYCL/6d5976661087ac77427af0b23c4b56100278dd81/doc/img/syclcc.png`

	CPU	GPU
work item	fiber	SIMD lane
subgroup	fiber[14]	SIMD unit
work group	thread	compute unit
local memory	system memory	*shared* memory[15]
global memory	system memory	device memory

Table 1.1.: Mapping of work item, work group and subgroup to hardware.

not exist or are hard to implement efficiently.

Work items are implemented using fibers on CPU. Fibers are userland threads, scheduled cooperatively on single hardware thread. In contrast to fibers, regular thread are scheduled by the OS and can be interrupted at any time. Fibers are managed by the program and have to yield to other fibers in the same thread to run. This difference makes switching between fibers faster, because only a subset of the processor context needs to be saved and restored [27]. Since no fibers in the same thread can run concurrently, no locks are needed for multiple fibers accessing the same memory. Additionally *false sharing* is reduced, because all work items, inside a work group, run on a single thread.

Because no guarantee for a fixed vector width can be made, regarding vectorization inside hipSYCL, subgroups are of size 1 on CPU. On GPU this is not a problem, as the hardware and compiler create executables that can be executed using the SIMD units of the GPU.

1.5.3. Extensions

HipSYCL implements several extensions enabling different features. This way an extension exist, to allow custom device operations[16] and provides shortcuts to prefetch data to the host from shared USM allocations.

Relevant for this thesis is the scoped parallelism extension. It tries to combine the advantages of nd-range kernels and hierarchical kernels. The aim is to provide a performance portable execution model, that can also be implemented efficiently on the host. Scoped parallelism is designed to be clean, well-defined and to provide predictable performance.

[14] subgroups are of size 1 on CPU and do not behave different from work items
[15] on-chip memory as described in section 1.2
[16] more information at `https://github.com/illuhad/hipSYCL/blob/33a4a7fcfc2ee13442d27b61fe6c970fa6a6af9f/doc/enqueue-custom-operation.md`

This is achieved by using an implementation defined kernel configuration and distributing the work using a user defined index space. The implementation defined kernel configuration allows the runtime to take the hardware capabilities and implementation details into consideration.

1.6. Subgroup Communication

Algorithm 1: Algorithm to generalize shuffle instructions.

Input: One value v of a given data type
an id specifying from which work item to read data
Output: One value r of the given data type
Data: t is a memory region at least as big as v, a multiple of 32-bit in size

```
1  s ← size of data type;
2  if native shuffle operation exists then
3  |  r ← shuffle(v, id);
4  else
5  |  copy s byte from v into t;
6  |  foreach 32-bit section of t → tmp do
7  |  |  tmp ← shuffle(tmp, id);
8  |  end
9  |  copy s byte from t into r;
10 end
11 return r
```

For subgroups special instructions can be used for communication between work items. One of these instructions is the shuffle instruction, which allows to read the value of another work item, specified by an id. Because the group algorithms have to work on a variety of data types, the shuffle instructions needed to be generalized, to support for data types not directly implemented in hardware.

The general approach if no shuffle instruction exists, as shown in algorithm 1, is to first copy the data into a local variable, which is a multiple of 32 bit[17]. After distributing, the exchanged data is copied into the element which is returned.

The approach for AMD and NVIDIA GPUs is almost identical. The only difference is the number of element sizes the shuffle instructions support and the choice of copy instruction to copy the data.

In addition to the shuffle operations, AMD GPUs also have DPP instructions. These instructions allow the work items to use data from other work items in their computations directly, instead of first loading them. The communication is not as flexible as

[17] This is the maximum size for shuffle instructions on AMD and NVIDIA GPUs

	bank 0			bank 1			bank 2			bank 3		
row 0	0	...	3	4	...	7	8	...	11	12	...	15
row 1	16	...	19	20	...	23	24	...	27	28	...	31
row 2	32	...	35	36	...	39	40	...	43	44	...	47
row 3	48	...	51	52	...	55	56	...	59	60	...	63

Figure 1.5.: Partitioning of 64 work items into 4 rows and banks. This visualizes, which of the 64 work items in a subgroup is affected, when using row and bank masking for Data Parallel Processing (DPP) instructions.

a shuffle, because the work items, with which data can be exchanged, is limited. The limitations are the result of the hardware implementing 16 element wide SIMD units. As a result each subgroup is split into four parts. 16 work items form a *row*, each pair of 4 threads inside a row is called a *bank*. This is shown in Figure 1.5.

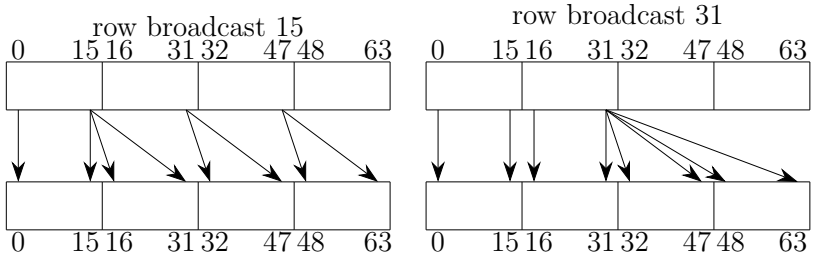


Figure 1.6.: Example for both DPP broadcasts. The numbers above and below the boxes show the ids at the beginning and end of a row. *row broadcast 15* broadcasts the values of the last work item in a row to the next row. *row broadcast 31* broadcasts from the 31 work item to all following work items.

Only work items inside the same row and bank can freely use the values of other work items. Some strategies, like shifting and rotating, are available inside rows. Broadcasting only works from the last thread of a row to the next row, or the 31th work item to row 2 and 3. This is visualized in Figure 1.6. The behavior of the DPP instructions can further be controlled using a row and bank mask, as well as a flag controlling the behavior for work items without data to read from. A work item has no data to read if the targeted work item is inactive or if the operation, for example when shifting to the right, targeting a work item outside the row.

1.7. Group Algorithms

The SYCL 2020 provisional specification defines the following group algorithms on nd-range kernels:

- *Barrier* provides a synchronization instruction between work items.
- *Broadcast* allows to distribute one value from a selected work item to all other work items
- *Boolean tests* are a group of three functions checking if any, all or none of the given values are true.
- A *Reduction* reduces values by applying a binary operation. This can for example be used to calculate the sum using *plus* as binary operation.
- *Scans* calculate $o_i = \bigoplus_{j=0}^{i} d_j$ with $\bigoplus$ representing the binary operation and d_j the jth value in d. If the binary operation is plus, then the result would be the prefix sum.

Each group algorithm can be executed on CPU, AMD and NVIDIA GPU and inside a subgroup or work group. For all algorithms, except barrier and broadcast, two versions exist:

- *Local value based* algorithm take exactly one value for each work item as input.
- *Pointer based* algorithms take two pointers marking the beginning and end of memory region accessible by all elements.

The pointer based algorithms are also implemented for hierarchical and scoped kernels.

All algorithms must work using native types, like integers and floating point values of up to 64-bit, and `sycl::vec` vectors. `sycl::vec` is a vector containing 1, 2, 3, 4, 8 or 16 elements of a native type.

1.8. Structure

In chapter 2 it is explained, how the correctness was checked and the benchmarks implemented and setup. It will also give an overview over the benchmarks using CUB and rocPRIM. In chapter 3 the implementation and optimization of the group algorithms on CPU is shown. In chapter 4 the group algorithms on AMD and NVIDIA GPUs are explained and the optimization results presented. In chapter 5 the results are summarized and further possible optimizations and research is shown. In Appendix A some additional code examples can be found.

2. Evaluation Setup

The code was optimized for speed. On GPU the amount of required local memory is also reduced, because it is a limited resource and regions used inside the group functions are unavailable to the developer. The correctness is tested using unit tests written for this thesis.

All benchmarks were compiled using clang 10.0.1.

2.1. Tests

The unit tests for hipSYCL were extended, to check the result of all group algorithms using one and two dimensional kernels of different sizes. The tests for group algorithm are implemented using a test-function, which takes three functions as parameter:

- A data generator creating the data used for the test.
- A function used to execute the algorithms with correct inputs.
- A validation function, which tests the output.

This approach reduces the amount of code needed for new tests. This setup also allows to reuse the data generator for many tests, reducing the verbosity even further.

2.2. Benchmarks

To measure the runtime sycl-bench [32] was extended by a new benchmark for all implemented group algorithms. The measured function is called 512 times per benchmark run and each benchmark is executed 5 to 10 times. The measurements given are always the median, as returned by sycl-bench. In plots the standard deviation is added as error bars.

For the measurements, the hardware shown in Table 2.1 was used.

The problem size and number of executions for each benchmark, listed in Table 2.3 was chosen to be large enough to fully utilize the hardware and to ensure sufficient runtime to get a small standard deviation.

Intel Xeon Gold 6230	AMD Radeon VII	NVIDIA V100
20 cores @2.1 GHz Cascade Lake CPU with AVX-512	60 compute units 13.8 TFLOPS@32-bit 3.46 TFLOPS@64-bit	84 streaming multiprocessors 14 TFLOPS@32-bit 7 TFLOPS@64-bit

Table 2.1.: Hardware used to run the benchmarks.

data type	size (in Byte)
int32	4
vec<int32:1>	4
vec<uint8:4>	4
fp32	4
vec<fp32:1>	4
int64	8
fp64	8
vec<int32:4>	16
vec<fp32:4>	16
vec<fp64:2>	16
vec<int32:8>	32
vec<fp32:8>	32

Table 2.2.: Tested data types and corresponding size. The *vec* data type is the vector class defined in the SYCL specification. The data types *int32* and *fp32* represent 32-bit large integer or floating point values.

The difference between median runtimes, when repeating a benchmark, is small. This is even the case, if the standard deviation is larger than 10% of the median on CPU. Which does not decrease, if the problem size or the number of runs is increased to the point, were the time required to run all the benchmarks exceeded one hour. We suspect, the scheduling of the fibers lead to a high variability in execution time.

The group size, controlled using the *local* argument, is tested for all power of twos between 32 and 1024. For pointer based functions on CPU, the size and number of work groups is limited to 1, 4, 8, 32 and 64, because each group runs on a single thread. Since the Intel CPUs have 40 cores, only when using 40 or more groups all threads are fully utilized. For local value based algorithms the number of work groups is determined by dividing the problem size with the work group size.

If nothing else is noted the mentioned speedups are the median over all data types and all work group sizes. In case the speedup of a data type is significantly higher or lower, it will be mentioned.

Benchmark	Hardware	size	num_runs
group reduce	AMD GPU	2 097 152	7
	NVIDIA GPU	131 072	10
	Intel CPU	32 768	10
group reduce pointer	AMD GPU	8192	7
	NVIDIA GPU	8192	10
	Intel CPU	51 200	10
scoped group reduce pointer	Intel CPU	204 800	5
group inclusive scan	AMD GPU	2 097 152	7
	NVIDIA GPU	131 072	10
	Intel CPU	32 768	10
group inclusive scan pointer	AMD GPU	2048	7
	NVIDIA GPU	2048	10
	Intel CPU	51 200	10
scoped group inclusive scan pointer	Intel CPU	204 800	5
group any of	AMD GPU	2 097 152	7
	NVIDIA GPU	131 072	10
	Intel CPU	32 768	10
group any of pointer	AMD GPU	81 920	7
	NVIDIA GPU	8192	10
	Intel CPU	51 200	10
scoped group any of pointer	Intel CPU	204 800	5

Table 2.3.: Benchmark parameters. On GPU the scoped parallelism benchmarks are not run, because they use the same code as for nd-range.

2.3. Library comparison

For the reduction and inclusive scan benchmarks using CUB [39] and rocPRIM [52] were implemented. The benchmarks also use sycl-bench as framework, but are launched using CUDA and HIP respectively. This was required, because the libraries would require attributes to be added to various parts of hipSYCL. The measurements contain both computation and data movements. Since the kernel launch and copying of the results is done manually and not by the SYCL runtime, some differences are possible.

For the comparisons the *plus* operation was used. Only algorithms using local values are implemented, because both CUB and rocPRIM do not offer algorithms comparable with our pointer based versions.

The reduction of both libraries only returns the result in the first work item, so a broadcast was added to obtain equivalent outputs. The local memory is allocated using

a special data type provided by the libraries. The size of the data type is dependent on the launch configuration. The amount of the required local memory for rocmPRIM is the same, as for the optimized group algorithms implemented in this thesis with 16 elements for reduction and scan. For CUB the local memory requirements are not as easy to predict, because padding is used to prevent access conflicts. The local memory allocated by CUB contains around 32 elements, plus an additional prefix element.

For rocPRIM and CUB the launch configuration, like the size and number work groups, has to be known at compile time. This is a significant difference to the algorithms specified in the provisional SYCL specification. Because we do not know the configuration at compile time, more runtime checks are needed in our implementations.

Both libraries offer multiple algorithms. For the comparison algorithms using shuffle instructions were chosen. For rocPRIM the shuffle based algorithm did not return correct results for work groups with over 256 work items. The code was checked and is in accordance with the documentation. For this reason the local memory based algorithm is shown as well for the reduction. This reduction algorithm uses 1024 elements of local memory.

Although rocPRIM also contains subgroup algorithms using DPP instructions, these can not be chosen, when using work group algorithms.

3. CPU

On CPU only runtime is considered during optimization, because the local memory is allocated only for group algorithms and as such does not impact the developer. Only sequential algorithms were used, because work groups run on a single thread.

In general the approach for optimizations is to reduce the overhead for local memory allocations. The other approach is vectorization. The performance, when using nd-range kernels, is bound by context switches, limiting the possible speedups.

The algorithms for subgroup are not shown, as they are trivial, because the subgroups contain only one work item in hipSYCL.

In Table 3.1 all code versions for CPU can be found. All speedups are presented in Table 3.2.

The version *removed vector code* did not change the speed, but improved code quality. Previous to this version the code handled vectors element wise, this change replaces element wise copies with a pointer based approach using aligned memory regions. The correct alignment needs to be used, because some intrinsics might depend on it. This is done by using `std::aligned_storage_t` from the C++ standard library.

name	change
baseline	first implementation of all functions
stack memory	replace dynamically allocated memory with statically allocated memory
removed vector code	removed vector specific code and allocate local memory using `std::aligned_storage_t`
OpenMP SIMD	use OpenMP for vectorization
explicit SIMD	use xsimd for vectorization

Table 3.1.: All benchmarked code versions for CPU in order.

Function	Version	Speedup
group reduce	baseline	1.0
	stack memory	1.7
	removed vector code	1.6
	OpenMP SIMD	1.5
	explicit SIMD	1.6
group reduce pointer	baseline	1.0
	stack memory	1.1
	removed vector code	1.1
	OpenMP SIMD	1.1
	explicit SIMD	1.1
scoped group reduce pointer	baseline	1.0
	removed vector code	1.0
	OpenMP SIMD	2.4
	explicit SIMD	1.9
group inclusive scan	baseline	1.0
	stack memory	2.4
	removed vector code	2.4
	OpenMP SIMD	2.4
group inclusive scan pointer	baseline	1.0
	stack memory	1.0
	removed vector code	1.0
	OpenMP SIMD	1.0
scoped group inclusive scan pointer	baseline	1.0
	removed vector code	1.1
	OpenMP SIMD	1.1
group any of	baseline	1.0
	stack memory	1.0
group any of pointer	baseline	1.0
	stack memory	1.1
scoped group any of pointer	baseline	1.0

Table 3.2.: Speedups on CPU

3.1. Barrier

The logic for the barrier is already present in the implementation of the boost fiber library used in hipSYCL. The only required change, was to make the barrier function available through the group object, passed as argument to the function.

A barrier requires the program to cycle through all work items, so they can reach the barrier. It also provides a memory fence functionality. The memory consistency guarantees on CPU are strong enough, so nothing has to be done.

The barrier implementation was not further optimized, because no reasonable improvements were found.

3.2. Broadcast

Algorithm 2: Broadcast

Input: value v, id i
Output: value r

```
1 if the local work item was chosen using i then
2 |   local_memory ← v;
3 end
4 barrier;
5 r ← local_memory;
6 barrier;
7 return r
```

The broadcast uses the value from one work item, indicated by an id, and makes it available to all work items in the work group. The id can either be an integer or an id object of the same dimension as the work group.

In algorithm 2 the selected work item writes its value into local memory. Then the work group needs to be synchronized, before reading the value. An additional barrier is required to ensure all work items have read the value, otherwise it is possible, that a fiber will override the value in another group function.

No further optimization potential was found, as the boost fiber library [11] does not offer a broadcast function.

3.3. Boolean Tests

3.3.1. Local Value Based

Algorithm 3: Check if any value is True

Input: predicate

Output: boolean stating if at least one given value was true

```
1 local_memory ← False;
2 barrier;
3 if predicate is true then
4 |   local_memory ← True;
5 end
6 barrier;
7 r ← local_memory;
8 barrier;
9 return r
```

This group of algorithms checks if any, all or none of the given values are true. The pseudo code in algorithm 3 shows the chosen approach. First the local memory is set to false[1]. After a barrier all work items with *true*[2] as predicate update the value.

For this algorithms no locks are needed, because all fibers in a work group run sequentially.

The local value based boolean test algorithms were not further optimized, because the optimization potential was assumed to be too small, because of the synchronization overhead.

[1] or true for checks for all and none true values

[2] *false* for the *all* check

3.3.2. Pointer Based

Algorithm 4: Check if any value is True based on pointers

Input: pointer to first value *first*, pointer after last value *last*

Output: boolean stating if at least one given value was true

```
1 if work item is the first in work group then
2     while first < last do
3         if the value at first is True then
4             return True
5         end
6     end
7 end
8 return False
```

The pointer based version uses a while loop to iterate over all values. This algorithm stops as soon as the result is known. This is the case, if a true value is encountered when checking for *any* or *none* or a false value when checking for *all*. The effect can be seen in Figure 3.1. The plot shows the difference between exiting after the first element and checking every element.

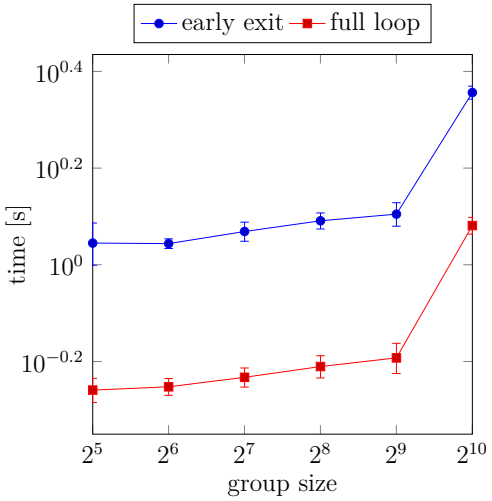


Figure 3.1.: Measurement for the boolean test algorithm based on pointers checking for *any*. For *early exit* the algorithm stops after the first element, while for *full loop* all items are checked.

The slowdown for work groups with size 1024 could be caused by cache misses, because each context switch for a fiber has to load some memory. For smaller work groups the cache could be large enough to not impact performance negatively. This slowdown can be observed in all benchmarks on CPU using nd-range kernels.

Even though local memory is used, it did not need to be dynamically allocated. The boolean value that is shared is so small, that it was saved in the four bytes used to share the pointer to local memory.

3.4. Reduction

3.4.1. Local Value Based

Algorithm 5: Algorithm to compute reduction.

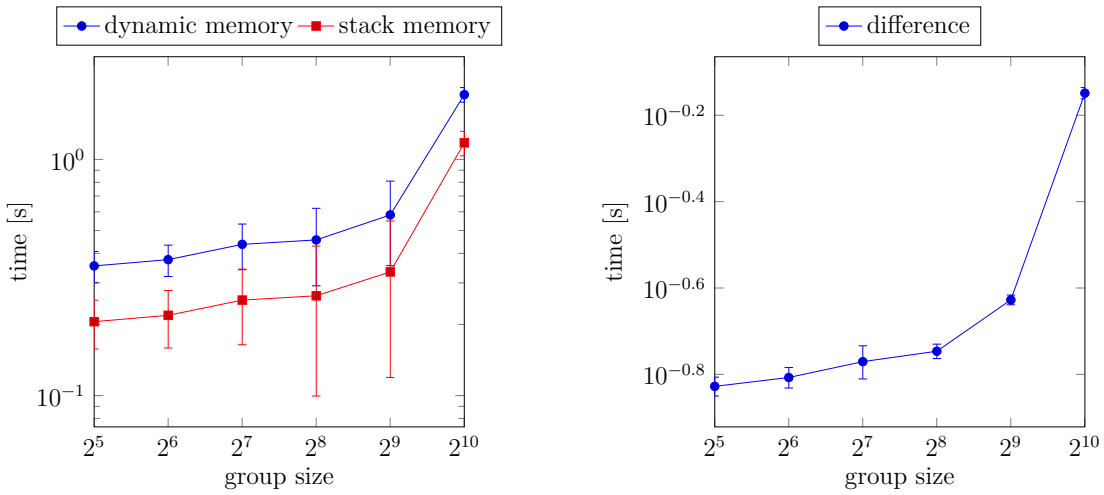
Input: value v, binary operation *op*

Output: value r

```
1  local_memory[local work item id] ← v;
2  barrier;
3  if work item is the first in the work group then
4  |  result ← v;
5  |  for i = 1; i < work group size; i++ do
6  |  |  result ← op(result, local_memory[i]);
7  |  end
8  |  local_memory[0] ← result;
9  end
10 barrier;
11 r ← local_memory[0];
12 barrier;
13 return r
```

The reduction uses a binary operation to reduce all values to a single one. To enable one work item to perform the computation, all work items store their value in local memory. Once all values are stored one work item performs the reduction.

This algorithm is the first to benefit from statically allocating the local memory, instead of dynamic allocation. The reason is, only the reduction an scan algorithms need to allocate local memory for all data types. The baseline implementation would allocate and deallocate the local memory each time it is called. This overhead can be completely removed, by allocating a fixed amount of memory on the stack. The results can be seen in Figure 3.2.



(a) reduction of 32-bit integer values

(b) time difference

Figure 3.2.: Comparison of dynamically allocated memory and stack memory using the reduction as an example.

Again the slowdown at 1024 work items per work group can be observed. The time difference between dynamic memory allocation and stack memory is also increased at a work group size of 1024. The speedup is likely because of the cache, because the data is on the stack it is less likely to be evicted, as other data close to the local memory region is accessed. The overall speedup resulting from the usage of stack memory achieved is 1.7x.

3.4.2. Pointer Based

Algorithm 6: Algorithm to compute reduction based on pointers.

Input: pointer to first value *first*, pointer after last value *last*, binary operation *op*

Output: value *r*

```
1  if first >= last then                          // no elements to reduce
2  |  return 0
3  end
4  if work item is the first in the work group then
5  |  result ← value at first;
6  |  foreach element tmp in the given region do
7  |  |  result ← op(result, tmp);
8  |  end
9  |  local_memory ← result;
10 end
11 barrier;
12 r ← local_memory;
13 barrier;
14 return r
```

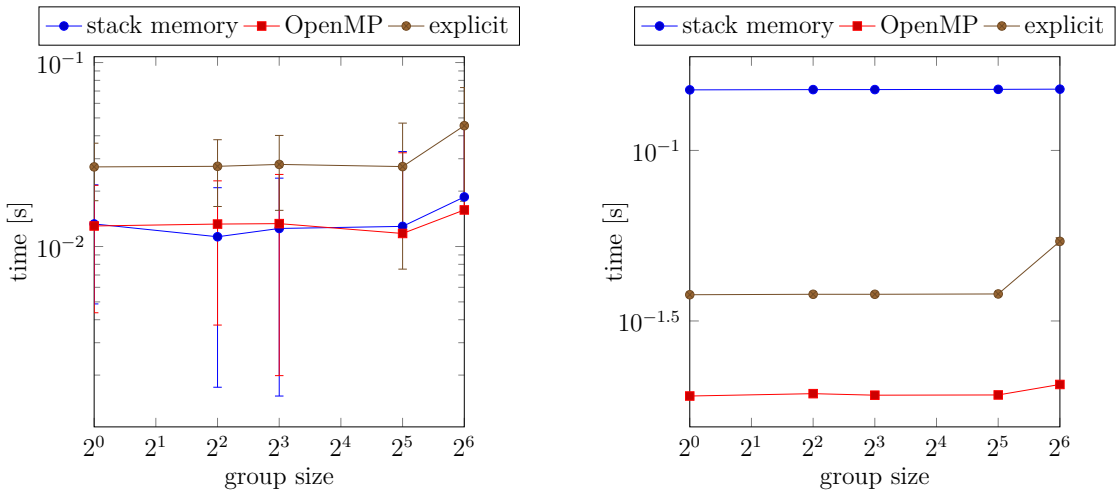
The pointer based reduction uses a for loop with the pointer p to access all values. This can be vectorized using OpenMP by adding `#pragma omp simd` in front of the loop. This instructs the compiler to vectorize the loop. Automatic vectorization with OpenMP is only possible if the loop does not exit early and the loop is a *for* loop.

We also tried vectorizing the *plus* operation manually using xsimd [62]. This is achieved by replacing the memory loads and stores, as well as the computation with wrappers from xsimd. We used aligned memory operations and horizontal addition instructions[3] for our reduction. During compilation the library determines the largest supported vector instruction and uses it in the code.

The results of both optimizations, manual and automatic vectorization, are plotted in Figure 3.3. For integer values no speedup was achieved and the manual vectorization was slower than the not vectorized code, likely caused by overhead. For floating point values significant speedups could be obtained. OpenMP is about 8 times faster, than the not vectorized code, the explicit vectorization was at least 4 times faster.

The reason for this difference can be seen in the generated code. In Figure 3.4 the section used for reducing most values is shown.

[3] A horizontal addition is when all values in the vector registers are summed up.



(a) 32-bit integer (b) 32-bit float

Figure 3.3.: Reduction measurements on CPUs for pointer based reduction using scoped kernel to reduce the overhead to a minimum.

The vectorized code by OpenMP and the code obtained using xsimd use different AVX2 instructions. OpenMP generates mostly `vpaddd` instructions. This adds up two 256bit wide vectors and sums all the values in the vector register in the end. In addition OpenMP utilizes the available hardware by calculating the sum using four AVX-256 registers and summing up 128 values in each iteration in the main loop.

The four registers `ymm0-ymm3` can be seen. The `480(%rdi,%r8,4)` is a shorthand for accessing the location at `%rdi`+4·`%r8`+480. This means in in each of the lines 2-17 four integers are read and added on one of the vector registers. In line 18 the register used for computing the address is updated and in line 19 and 20 the loop is repeated if there are values left.

The vectorization using xsimd[4] uses the horizontal add instruction `vphaddd`. The main problem with the manual coded version is that for each vectorized addition multiple further instructions like `vextracti128`, which is used to extract a 128bit value from the 256bit register, are needed, to reduce the result of the horizontal addition to a single non-vector element.

[4]The generated assembly can be seen in Figure A.4.

```
1  .LBB275_25:
2          vpaddd  (%rdi,%r8,4), %ymm0, %ymm0
3          vpaddd  32(%rdi,%r8,4), %ymm1, %ymm1
4          vpaddd  64(%rdi,%r8,4), %ymm2, %ymm2
5          vpaddd  96(%rdi,%r8,4), %ymm3, %ymm3
6          vpaddd  128(%rdi,%r8,4), %ymm0, %ymm0
7          vpaddd  160(%rdi,%r8,4), %ymm1, %ymm1
8          vpaddd  192(%rdi,%r8,4), %ymm2, %ymm2
9          vpaddd  224(%rdi,%r8,4), %ymm3, %ymm3
10         vpaddd  256(%rdi,%r8,4), %ymm0, %ymm0
11         vpaddd  288(%rdi,%r8,4), %ymm1, %ymm1
12         vpaddd  320(%rdi,%r8,4), %ymm2, %ymm2
13         vpaddd  352(%rdi,%r8,4), %ymm3, %ymm3
14         vpaddd  384(%rdi,%r8,4), %ymm0, %ymm0
15         vpaddd  416(%rdi,%r8,4), %ymm1, %ymm1
16         vpaddd  448(%rdi,%r8,4), %ymm2, %ymm2
17         vpaddd  480(%rdi,%r8,4), %ymm3, %ymm3
18         subq    $-128, %r8
19         addq    $4, %r11
20         jne     .LBB275_25
```

Figure 3.4.: Main summation loop as generated by OpenMP.

3.5. Scans

3.5.1. Local Value Based

Algorithm 7: Algorithm to compute an inclusive scan.

Input: value v, binary operation op

Output: value r

```
1  local_memory[local work item id] ← v;
2  barrier;
3  if work item is the first in the work group then
4  |   foreach position i an element was stored do
5  |   |   local_memory[i] ← op(local_memory[i], local_memory[i − 1]);
6  |   end
7  end
8  barrier;
9  r ← local_memory[local work item id];
10 barrier;
11 return r
```

There are two scan algorithms in the SYCL 2020 provisional specification, the inclusive and exclusive scan. The exclusive scan is an inclusive scan with all elements moved one to the right. The inclusive scan is computed as $o_i = \bigoplus_{j=0}^{i} d_j$, while the exclusive scan is computed as

$$o_i = \begin{cases} 0 & \text{if } i = 0 \\ \bigoplus_{j=0}^{i-1} d_j & \text{if } i > 0 \end{cases}$$

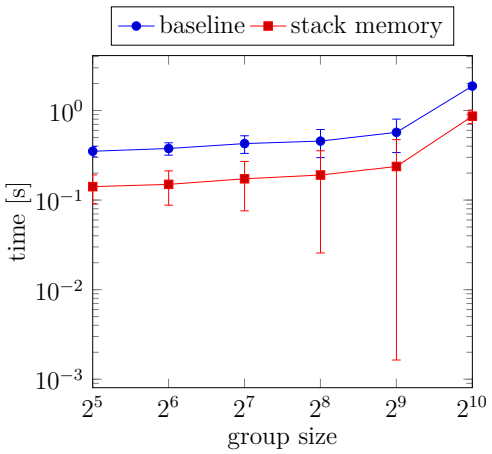


Figure 3.5.: The speeup achieved by using stack memory instead of dynamically allocated memory.

Using stack memory for the local memory provides a speedup of 2.4x. The measurements can be seen in Figure 3.5.

3.5.2. Pointer Based

Algorithm 8: Algorithm to compute an inclusive scan based on pointers.

Input: pointer to first value *first*, pointer after last value *last*, pointer to store result *result*, offset *init*, binary operation *op*

Output: pointer behind last element in result *result*

```
1 if work item is the first in the work group then
2 |   result[0] ← op(init, value at first);
3 |   foreach i less than the number of passed elements do
4 |   |   result[i] ← op(result[i − 1]), first[i]);
5 |   end
6 end
7 return result + number of elements
```

The pointer based scan algorithms are implemented using the same approach as the local value based algorithm. Instead of returning the values of the scan, the result are written into another memory region and the pointer behind the last element of the result is returned.

An attempt at vectorizing the scan algorithm using OpenMP was done, but OpenMP was not able to automatically generate vectorized code. It was not attempted to manually vectorize the algorithm, because this would need to be done for each operation separately. Possible algorithms for SIMD based scans are explained in the paper *Parallel Prefix Sum with SIMD* by Zang et. al [64].

4. GPU

On GPU, we try to reduce the local memory usage and the runtime. The optimizations consist mostly of improving memory accesses and using shuffle instruction. On AMD GPUs additional improvements can be achieved using DPP instructions.

The resulting speedups can be found in Table 4.1 and Table 4.2 and the benchmarked code versions are listed in Table 4.3.

None of the subgroup algorithms uses any local memory, because all communication is done using shuffle instructions.

Function	Version	Speedup	local memory usage
group reduce	baseline	1.0	1024
	removed vector code	1.0	1024
	subgroup	2.1	32
group reduce pointer	baseline	1.0	1024
	removed vector code	1.0	1024
	subgroup	1.7	32
group inclusive scan	baseline	1.0	1024
	removed vector code	1.0	1024
	subgroup	2.0	32
group inclusive scan pointer	baseline	1.0	1024
	removed vector code	1.0	1024
	subgroup	8.9	32
group any of	baseline	1.0	0
group any of pointer	baseline	1.0	0
	interleaved	1.4	0

Table 4.1.: Speedups and local memory usage in elements on NVIDIA GPU. The speedup is the median over all benchmarked data types and group sizes.

Function	Version	Speedup	local memory usage
group reduce	baseline	1.0	1024
	removed vector code	1.0	1024
	subgroup	7.6	16
	dpp	4.5	16
	restricted dpp	8.4	16
	unchecked dpp	15	16
group reduce pointer	baseline	1.0	1024
	removed vector code	1.0	1024
	subgroup	1.0	64
	dpp	1.0	64
	restricted dpp	1.0	64
	unchecked dpp	1.2	64
group inclusive scan	baseline	1.0	1024
	removed vector code	1.0	1024
	subgroup	2.4	16
	dpp	2.4	16
	restricted dpp	2.5	16
	unchecked dpp	4.4	16
group inclusive scan pointer	baseline	1.0	1024
	removed vector code	1.0	1024
	subgroup	32	16
	dpp	32	16
	restricted dpp	33	16
	unchecked dpp	43	16
group any of	baseline	1.0	0
group any of pointer	baseline	1.0	0
	interleaved	7.3	0

Table 4.2.: Speedups and local memory usage in elements on AMD GPU. The speedup is the median over all benchmarked data types and group sizes.

name	platform	change
baseline	both	first implementation of all functions
removed vector code	both	removed vector specific code and allocate local memory using `std::aligned_storage_t`
interleaved	both	access memory in an interleaved fashion
subgroup	both	use subgroup functions inside work group functions
dpp	AMD GPU	use dpp instead of shuffle for reduce and scan
unchecked dpp	AMD GPU	use dpp but do not check if subgroup is complete[1]
restricted dpp	AMD GPU	Only use DPP for data types larger than 16 byte

Table 4.3.: All benchmarked code versions on GPU in order.

4.1. Barrier

The barrier synchronizes all work items inside a work group or subgroup. Additionally it can also be used to get certain memory guarantees using memory fences. After a memory fence all work items within the scope can read all changes done by other work items in the same scope.

The available memory scopes provide consistency for all work items inside the same subgroup, work group and device. There is also a memory scope to guarantee a work item can observe its own changes, but this is always the case in this thesis.

The synchronization intrinsics for subgroup and work group provide a matching memory fence on NVIDIA GPUs. On AMD GPUs no synchronization instruction exists for subgroups, but they are not needed, because subgroups are executed in lockstep and synchronized automatically. If a higher level of consistency is required, the appropriate memory fence is additionally executed.

The implementation of the work group and subgroup barrier is assumed to be optimal, because a single specialized instruction is called.

[1] only valid if the work group size is a multiple of the subgroup size

4.2. Broadcast

Algorithm 9: Broadcast on subgroups

Input: value v, id of broadcasting work item i

Output: value

1 **return** *shuffle(v, i)*

The broadcast for work groups is identical to the CPU version. On subgroups, shuffle instructions are used, as shown in algorithm 9.

The broadcast is believed to be implemented in a manner, that is not easily improved. As a result no optimization was implemented.

4.3. Boolean Tests

4.3.1. Local Values Based

For the boolean test algorithms, intrinsics are available for subgroups and work groups. As a result no local memory is used for work groups and subgroups.

For work groups, the functionality is provided by a special version of the synchronization instruction. There are only versions for *all* and *any*. For *none* the instruction for *any* is negated, this corresponds to $\forall x \in X : P(x) \Leftrightarrow \nexists x \in X : \neg P(x)$. The implementation for subgroups is done using matching instructions for subgroups.

On NVIDIA GPUs the instructions also allow using a mask, to only use some work items in these operations, but this is not required for the group algorithms.

The local value implementations are considered optimal, as only a single specialized instruction is called.

4.3.2. Pointer Based

Algorithm 10: Boolean test for any true value using pointers.

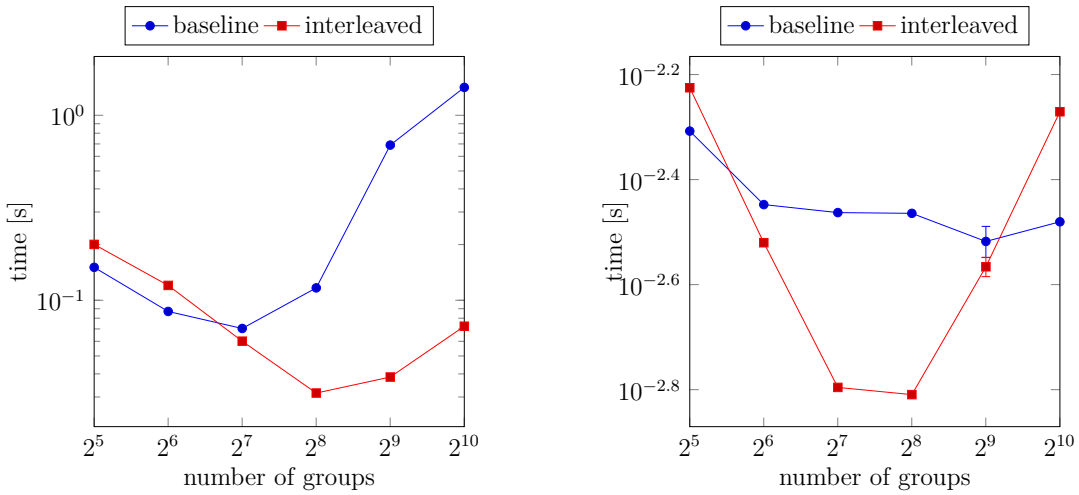
Input: pointer to first value $first$, pointer after last value $last$

Output: value r

1 $lid \leftarrow$ local id of work item;
2 $r \leftarrow$ True;
3 **for** p = *first+lid;* $p <$ *last;* p += *work group size* **do**
4 $\quad r \leftarrow r \vee *p$;
5 **end**
6 **return** *work group any of(r)*

In the pointer based boolean test functions, each work item checks the entries with a stride of the work group size and the local id as offset. The intermediate results are stored in a locale variable. In the end the work group algorithm using local values is used to get the final result.

In the baseline implementation each work item got assigned a continuous memory region to check, but this results in strided memory accesses from the view of the memory controller, because the accesses in each step are disjunct. The optimized version shown in algorithm 10 starts each work item offset by their local id and only check each work group sized element. As a result, all memory accesses in one step are to a continuous region.



(a) AMD GPU (b) NVIDIA GPU

Figure 4.1.: Pointer based boolean check on AMD an NVIDIA GPU.

The resulting speedup is 1.4x on NVIDIA GPUs and 7.3x for AMD GPUs. In Figure 4.1 can be seen, that interleaved memory access is less effective at very small and very large group sizes. No reason for this behavior was found. On AMD GPUs the runtime behaves as expected, as the speedup increases with larger work groups.

4.4. Reduction

4.4.1. Local Value Based

Algorithm 11: Algorithm to compute reduction.

Input: value v, binary operation op

Output: value $result$

```
1  lid ← local work item id;
2  range ← ⌈work group size/subgroup size⌉;
3  local_memory[lid] ← v;
4  v ← subgroup reduce(v, op) ;                          // first reduction
5  if work item is the first in the subgroup then
6  |  local_memory[subgroup id] ← v;
7  end
8  if only one item given to reduce then return local_memory[0] ;
9  if work group size == (subgroup size)^2 then
   |  // special case with two subgroup algorithms
10 |  if work item is part of first subgroup then
11 |  |  v ← subgroup reduce(local_memory[lid], op);
12 |  end
13 |  if work item is the first in the work group then local_memory[0] ← v ;
14 |  barrier;
15 else                                                  // default case
16 |  for i = ⌈range/2⌉; i > 1; i = ⌈i/2⌉ do
17 |  |  if work item belongs to the lower half and the target exists then
18 |  |  |  local_memory[lid] ← op(local_memory[lid], local_memory[lid + i]);
19 |  |  end
20 |  |  range ← range/2 + range%2;
21 |  |  barrier;
22 |  end
23 |  barrier;
24 |  return op(local_memory[0], local_memory[1])
25 end
26 return local_memory[0]
```

The reduction is computed using the subgroup algorithms shown in algorithm 11. This reduces the local memory requirement by a factor of the subgroup size, compared to the initial version.

If the work group consists of only one subgroup, then the algorithm can exit early, as the subgroup algorithm already reduced all elements. Another special case is, if the work group size is the square of the subgroup size. Then the result can be computed using a second subgroup reduction, because each subgroup reduction reduces the number of remaining results by a factor of the subgroup size. The only configuration, where this is the case, is on NVIDIA GPUs with a work group size of 1024.

The default case uses interleaved memory accesses. To achieve this the elements are split into two groups and the binary operation is applied to one element of each group and written back. After each of these steps the number of elements halves. In case the number of elements is odd, one extra element needs to be considered in the next step. The range variable is used to keep track how many elements are left and i determines the offset. Because i cannot reach 0, the loop is exited once two elements are left. These two remaining elements are reduced on return.

This version using subgroup algorithms achieved a speedup of 2.1x on NVIDIA GPUs and 7.6x on AMD GPU.

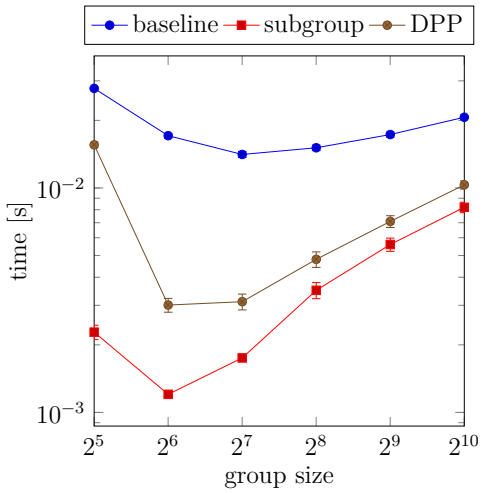
On AMD GPUs, the subgroup algorithms can be further optimized using DPP instructions. However, this is generally only possible if all work items inside a subgroup are active.

To get information about active threads the execution mask can be read from a special GPU register using in-line assembly. If not all work items are active, the shuffle based approach is used, otherwise the DPP instructions are used. Because communication is only possible inside a row[2] and from one row to the next, the final result will only be present in the last work item, before it is broadcast.

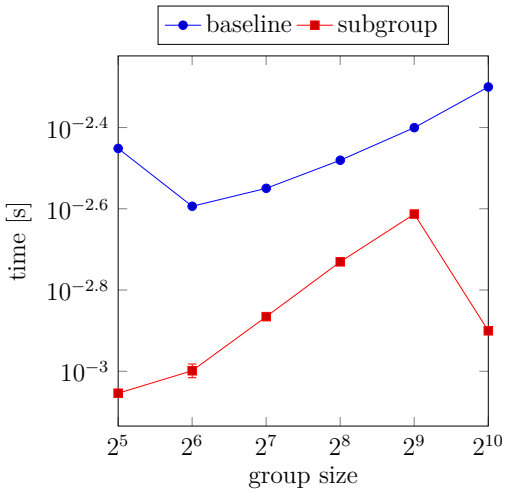
Using DPP instructions is slower than shuffle instructions for small data types with a speedup of 4.5x with respect to the baseline. To improve the performance the DPP based approach is only compiled into the executable, if the data type is larger than 16 B. This compile time condition increases the speedup to 8.4x, which is faster than the version using shuffle operations. DPP instructions should enable even better performance, as can be seen when the checks for a full subgroup are removed. This results in a speedup of 15x. However this unchecked version can return wrong results, which is why it cannot be used in a general setting.

In Figure 4.2 the measured times for 32-bit integer and vectors containing 4 32-bit floating point values is given. For the 32-bit integer case on AMD GPUs the *DPP unchecked* case is not shown has it has an almost identical runtime as *subgroup*.

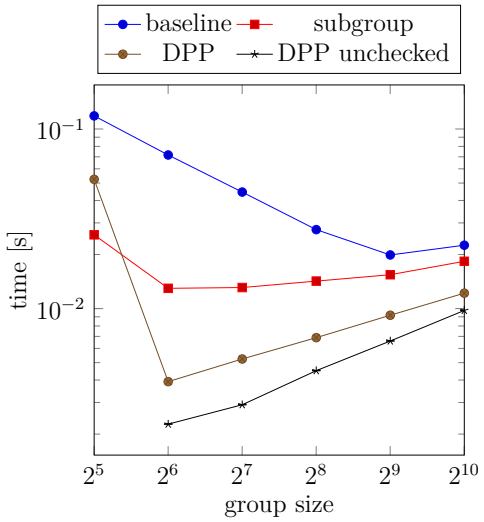
[2] 16 work items inside the same subgroup



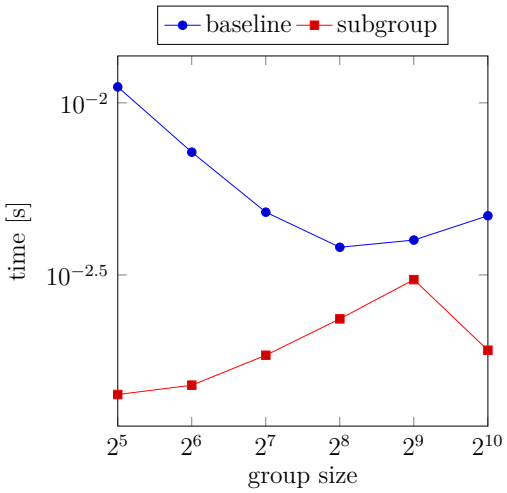
(a) AMD GPU using 32-bit integer



(b) NVIDIA GPU using 32-bit integer



(c) AMD GPU on 4 32-bit float



(d) NVIDIA GPU on 4 32-bit float

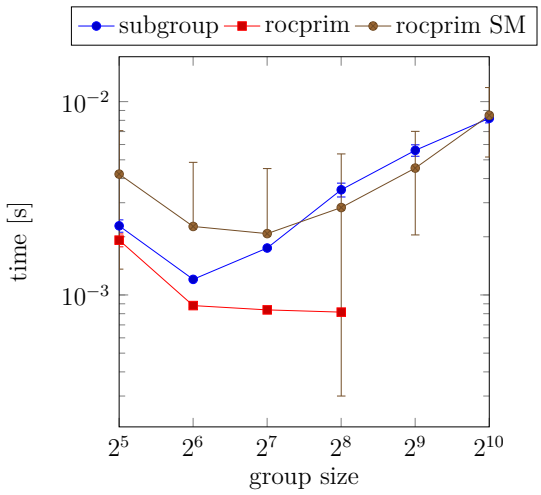
Figure 4.2.: Reduction measurements on GPU using local values. The *dpp* case is not shown for AMD GPUs with 32-bit integers, as the times are almost identical to the *subgroup* case.

For the NVIDIA GPU the difference when using subgroup algorithms, as well as the special case with a significant speedup at $1024 = 32^2$ work items, can be observed. Using two subgroup algorithms, results in a speedup of over 5x for 32-bit float and integer at work group sizes of 1024. As expected both 32-bit integer and the vector of 4 32-bit floating point values behave similarly.

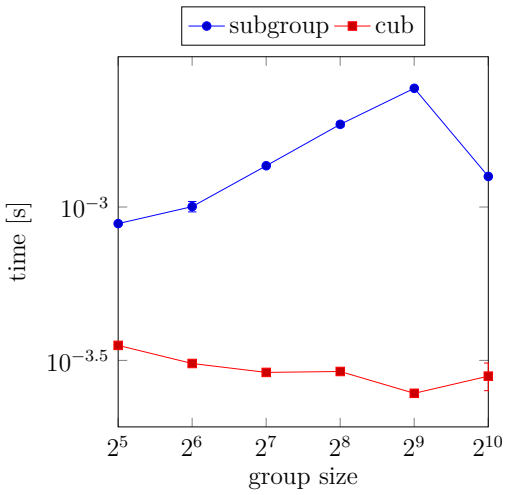
On the AMD GPU, the data type has a significant impact on the performance. For 32-bit integers, using DPP instructions slows the computation down, while for the vector of 4 32-bit floating point values the DPP case is significantly faster. For the vector type the unchecked DPP case shows further potential. The reason, for the better performance with vector values, is that DPP instructions are pipelined. As a result when using them for small values, some time is spend waiting for a fixed number of cycles to clear the pipeline, while the shuffle instructions allow switching to other subgroups increasing throughput.

In Figure 4.3 our implementation is compared to the implementation of rocPRIM [52] and CUB [39]. The shuffle based reduction is faster than our implementation in all cases, except on AMD GPU using 64-bit floating point and a work group of size 64. Our implementation on AMD has comparable performance with the shared memory version from rocPRIM. CUB has implementations for summation on subgroups, implemented directly in PTX[3] for many data types. This results in very good performance, but makes the implementation harder to understand and read.

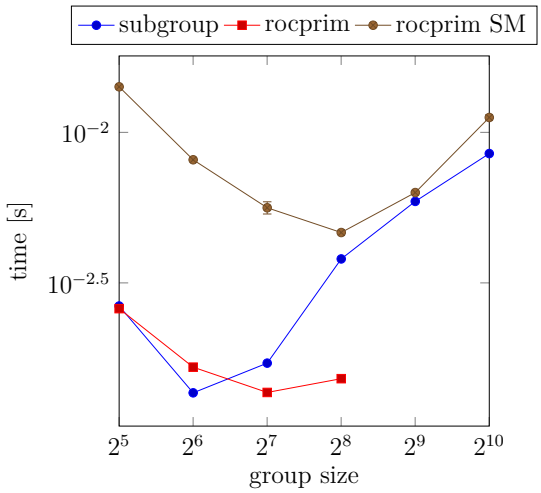
[3] a low level language comparable with assembly



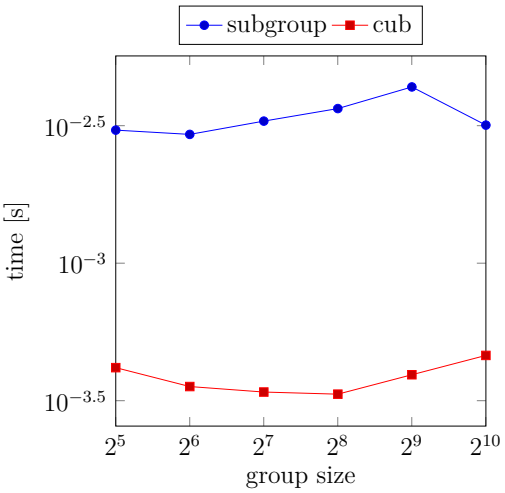
(a) AMD GPU, 32-bit integer



(b) NVIDIA GPU, 32-bit integer



(c) AMD GPU, 64-bit floating point value



(d) NVIDIA GPU, 64-bit floating point value

Figure 4.3.: Comparison of implemented algorithms for reduction with similar libraries. The shuffle based rocPRIM algorithm did not return correct results for work groups larger than 256, the shared memory algorithm is given in addition.

4.4.2. Pointer Based

Algorithm 12: Algorithm to compute reduction based on pointers.

Input: pointer to first value $first$, pointer after last value $last$, binary operation op

Output: value $result$

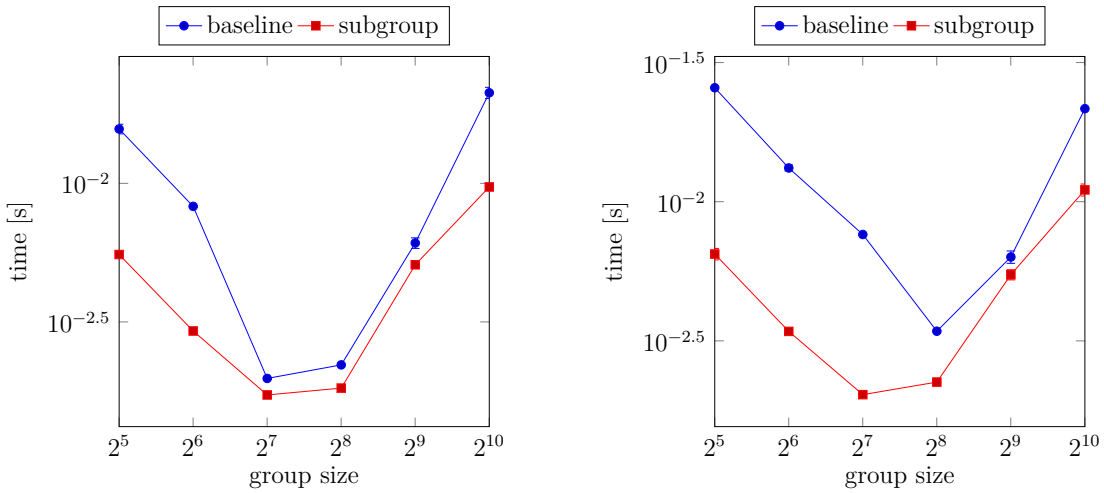
```
1  lid ← local id of the work item;
2  num_elements ← last − first;
3  range ← work group size;
4  start_ptr ← first + lid;
5  if num_elements <= 0 then return 0 ;
6  else if num_elements == 1 then return value at first ;
7  if num_elements >= work group size then
8      r ← *start_ptr;
9      for p = start_ptr+range; p < last; p += range do
10         r ← op(r, *p);
11     end
12     return work group reduce(r, op)
13 else
14     num_subgroups ← ⌊ num_elements / subgroup size⌋;
15     if only one subgroup in work group ∧ lid < num_elements then
16         copy all values to local_memory;
17     else if subgroup id < num_subgroups then
18         active_threads ← num_subgroups · subgroup size;
19         r← *start_ptr;
20         reduce only with active_threads until each work item has one value;
21         r ← subgroup reduce(r, op);
22         if first work item in subgroup then
23             local_memory[subgroup id] = r;
24         end
25         range← num_subgroups;
26     end
27     result ← interleaved reduction of all elements in local memory;
28     return result
29 end
```

To compute the reduction using pointers, two special cases have to be handled. The first one is, if no element needs to be reduced, in which case a default constructed

element is returned. The other case is, if exactly one element needs to be reduced, in which case it is just returned.

The reduction is performed similarly, whether we have less elements than work items or not. In each case the number of subgroups is used, which allows us to assign at least one element to each work item. In case we have more elements than work items, we can use the group based reduction to get the final result. Otherwise we use the subgroup based reduction and write the intermediate results into local memory to be reduced using a loop like to the one used in algorithm 11.

The baseline implementation is identical, but uses less optimized work group reductions for the final reduce step.



(a) 32-bit integer

(b) 64-bit integer

Figure 4.4.: Pointer based reduction measurements for NVIDIA GPU

In Figure 4.4 the difference between 32-bit and 64-bit values can be seen. For the baseline the 64-bit integers are significantly slower than the 32-bit integer. When using shuffle instruction, this difference between the data types is reduced.

4.5. Scans

4.5.1. Local Value Based

Algorithm 13: Algorithm to compute an inclusive scan.

Input: value v, binary operation *op*
Output: value *result*

```
1  v ← subgroup inclusive scan(v, op);
2  if last work item in subgroup then
3  |  local_memory ← v;
4  end
5  barrier;
6  if work item is part of first subgroup then
7  |  lm_index ← local work item id;
8  |  if local id > number of subgroups then
9  |  |  lm_index ← 0;
10 |  end
11 |  tmp ← subgroup inclusive scan(local_memory[lm_index], op);
12 |  if local id > number of subgroups then
13 |  |  local_memory[local work item id] = tmp;
14 |  end
15 end
16 barrier;
17 if subgroup id == 0 then
18 |  return v
19 end
20 return op(local_memory[subgroup id - 1], v)
```

The inclusive scan is computed, by first computing the inclusive scan inside the subgroups. Then, the value of the last work item of each subgroup is written to local memory. Using a single subgroup another inclusive scan is performed over the values inside local memory. These values can then be used as offset for each subgroup. The first subgroup does not need an offset, because it does not depend on a previous subgroup. All other subgroups apply the calculated offset before returning.

When an exclusive scan is computed, all values are shifted one to the right using shuffle instructions. This leaves the first work item in each subgroup without value. The missing values can be read from local memory, because they correspond with the offsets. This is shown in Figure 4.5.

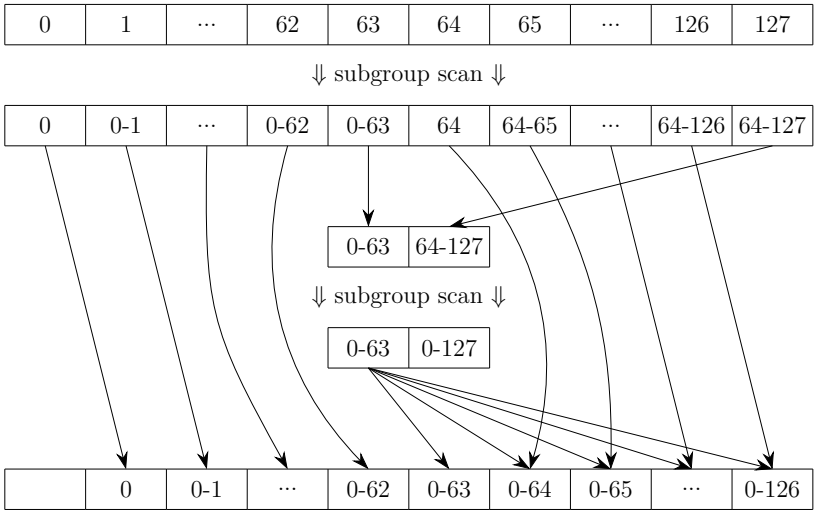


Figure 4.5.: Algorithm used to implement exclusive scan. The numbers in the boxes represent from which work item the values are aggregated from. The subgroup scan used is an inclusive scan. The box in the bottom left corner represents the first element, which is usually is initialized with zero.

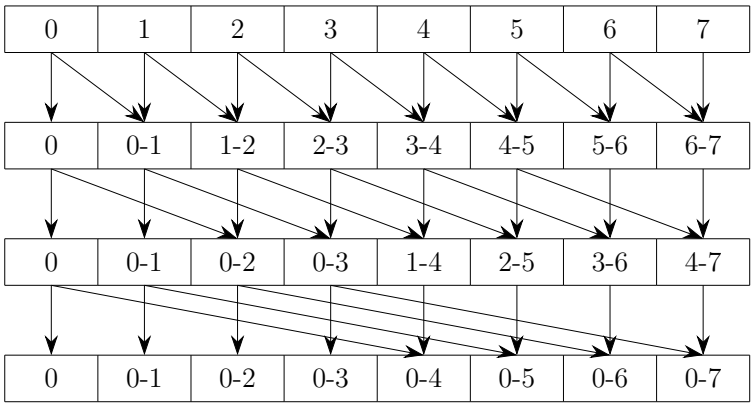
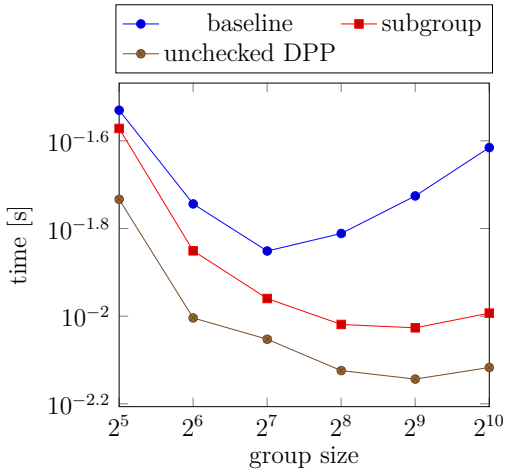
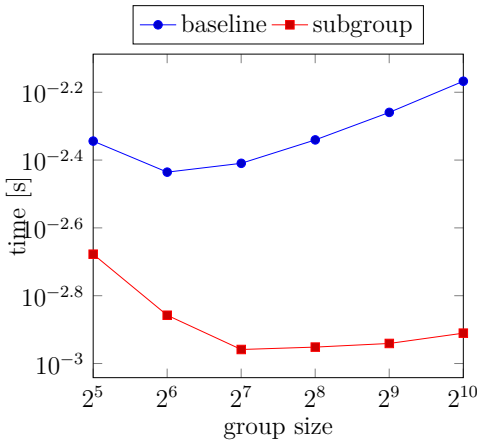


Figure 4.6.: Algorithm used to implement inclusive scan initially. The numbers in the boxes represent from which work item the values are aggregated from.

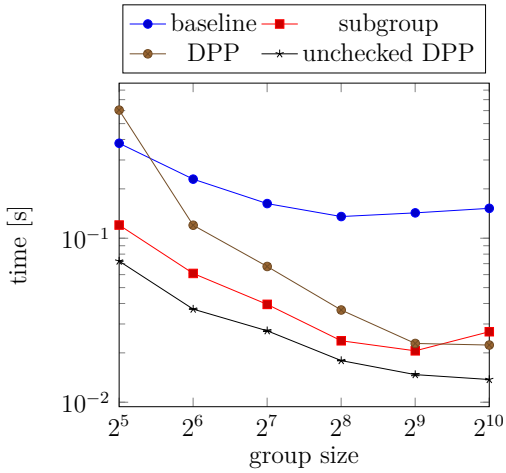
Initially the scan was performed by writing each value into local memory and applying the binary operation on an offset. This offset was then doubled every step, until the scan is completed. This is visualized in Figure 4.6.



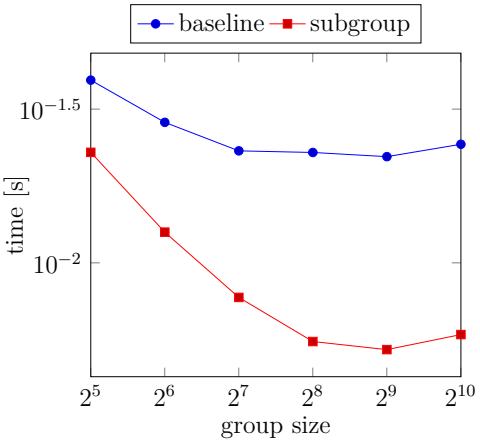
(a) AMD GPU, 32-bit integer



(b) NVIDIA GPU, 32-bit integer



(c) AMD GPU, 8 32-bit integers



(d) NVIDIA GPU, 8 32-bit integers

Figure 4.7.: Inclusive scan measurements for 32-bit integers and vectors of 8 32-bit integers. The measurements for *dpp* are not shown for AMD GPU using 32-bit integers, because it is almost identical to *subgroup*.

The subgroup based algorithm achieved a speedup of 2.4x on AMD GPU and 2.0x on NVIDIA GPU. Further optimization is possible on AMD GPUs, because they provide DPP instructions. The median speedup with DPP instructions is the same, as using shuffle instructions, but if only used for data types larger than 16 B the overall speedup is 2.5x. Without any checks concerning the subgroup size a speedup of 4.4x is possible, this however cannot be done in a general setting.

In Figure 4.7 the advantage of DPP instructions for scans is observable. Data can only be shared inside of a row, consisting of 16 work items, and from the last work item

in a row to the next row with DPP instructions. This is a good fit for scans, because elements only depend on their predecessors.

```
1 tmp ← local + XX(local);
2 if work item not finished then
3 |   local ← tmp;
4 end
```

(a) pseudo code

```
1  v_mov_b32_dpp v2, v1  row_shr:1 row_mask:0xf bank_mask:0xf
2  v_cndmask_b32_e64 v10, v2, 0, s[10:11]
3  v_add_u32_e32 v1, s13, v10
4  v_mov_b32_e32 v2, v8
5  s_nop 1
6  v_mov_b32_dpp v2, v1  row_shr:2 row_mask:0xf bank_mask:0xf
```

(b) dpp

```
1  ds_bpermute_b32 v1, v3, v1
2  s_waitcnt lgkmcnt(0)
3  v_cndmask_b32_e64 v13, 0, v1, s[8:9]
4  v_add_u32_e32 v1, s13, v13
5  ds_bpermute_b32 v12, v7, v1
```

(c) shuffle

Figure 4.8.: Generated assembly for 32-bit integer scan. Only the code between two DPP instructions or shuffle instructions is shown. The code on top is the source code with *XX* instead of the shuffle or DPP instruction. The *XX* in the pseudo code is a shuffle or a DPP instruction.

The speedup can also be observed when taking a look at the generated code in Figure 4.8. Only the code between two DPP instruction or shuffle instructions is shown, as this is the smallest part that shows the difference. The pseudo code Figure 4.8a shows the code the assembly is generated from. The `XX` function is either a shuffle or DPP instruction.

In Figure 4.8b the `v_mov_b32_dpp` instruction in line 1 and 6 is used to share the data. `v_cndmask_b32_e64` and `v_mov_b32_e32` in lines 2 and 4 are used to only change values of elements not yet completed, this is the same check as in the pseudo code. In line 3 the actual addition and copy takes place.

In Figure 4.8c the `ds_permute_b32` instruction in line 1 and 5 are the shuffle instructions. In line 3 the condition is checked and in line 4 the addition is performed using `v_add_u32_e32`.

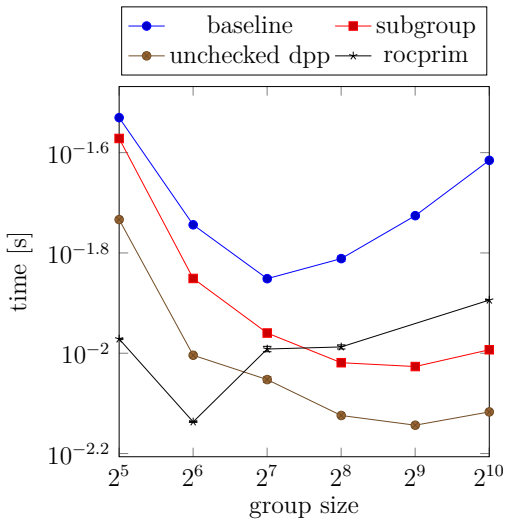
The important difference is the `s_nop 1` instruction in line 5 of the DPP assembly and the `s_waitcnt lgkmcnt(0)` instruction in line 2 of the shuffle assembly. Both instruction are used to wait, until the hardware is in the correct state. `s_nop 1` waits exactly one clock cycle, while `s_waitcnt lgkmcnt(0)` waits until the data from the shuffle operation

is present. This means on AMD GPUs the scan using shuffle instructions is limited by the speed of the local memory controller, while the variant using DPP instructions has one wait cycle. `s_waitcnt lgkmcnt(0)` allows the GPU to switch to a different thread to hide this latency.

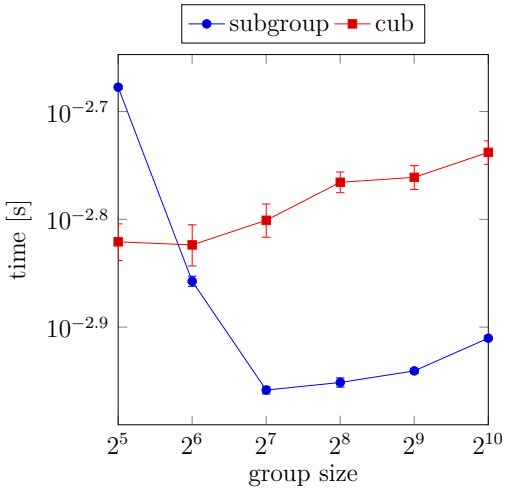
The DPP case can be even further optimized, because for some operations like addition the lines 1-4 in Figure 4.8b can be merged into one masked DPP addition instruction. To get speed improvements, other work needs to be available, as the latency is not reduced when using a DPP addition. For vector data types, it might be able to get close to full saturation of the SIMD unit.

To put this further into perspective: The AMD Radeon R9 Fury X, a high-end GPU from 2015, has a total peak bandwidth of $8.6\,\mathrm{TB\,s^{-1}}$ for shuffle instructions, but when using the registers, as is the case for DPP instructions, the peak bandwidth is $51.6\,\mathrm{TB\,s^{-1}}$ [53].

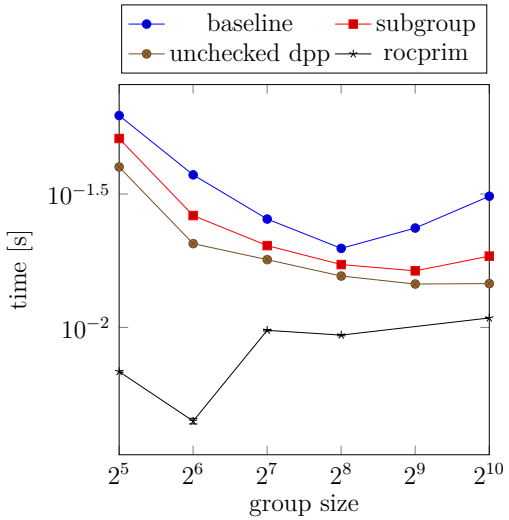
In Figure 4.9 the comparison between our implementation and rocPRIM and CUB is shown. Our implentation and AMD and NVIDIA GPU are faster than the library implementation for 32-bit integer and work groups with at least 64 work items for NVIDIA GPUs and 256 work items for AMD GPUs. This however is not the case for 64-bit floating point values, where the libraries are both faster.



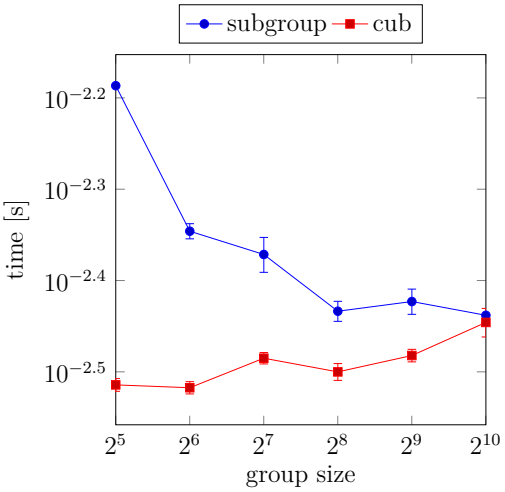
(a) AMD GPU, using 32-bit integer



(b) NVIDIA GPU, 32-bit integer



(c) AMD GPU, 64-bit floating point value



(d) NVIDIA GPU, 64-bit floating point value

Figure 4.9.: Comparison between our inclusive scan algorithms and rocPRIM and CUB.

4.5.2. Pointer Based

Algorithm 14: Algorithm to compute an inclusive scan based on pointers.

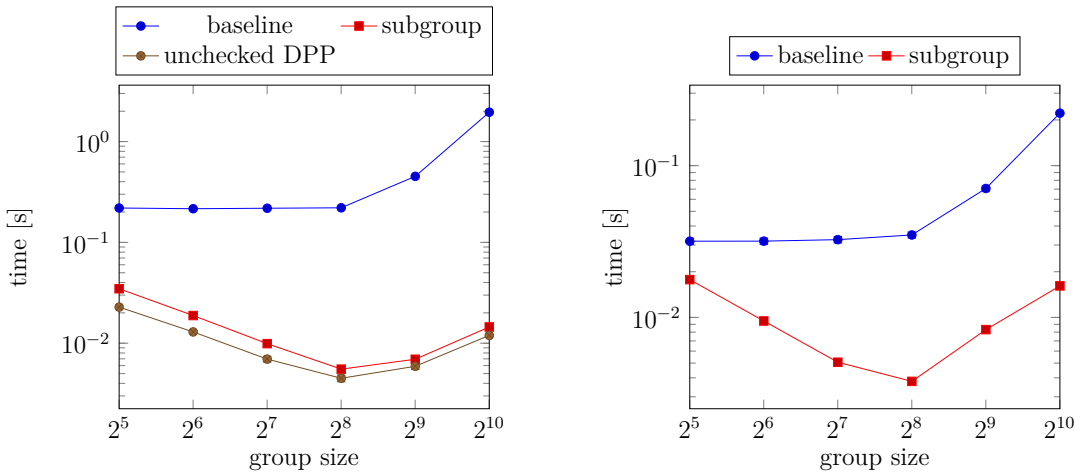
Input: pointer to first value *first*, pointer after last value *last*, pointer to store result *result*, offset *init*, binary operation *op*

Output: pointer behind last element in result *result*

```
1  lid ← local id of the work item;
2  range ← work group size;
3  i ← 0;
4  while scan not done do
5  |  offset ← lid + i · range;
6  |  if offset in range then
7  |  |  local ← first[offset];
8  |  else
9  |  |  local ← 0;
10 |  end
11 |  local = work group inclusive scan(local, op);
12 |  if not in first iteration then
13 |  |  local ← op(local, offset);
14 |  end
15 |  if offset in range then
16 |  |  result[offset] ← local;
17 |  end
18 |  carry_over ← group broadcast(local, id of last work item);
19 |  i← i+1;
20 end
21 return result + num_elemens
```

The pointer based algorithm iterates over all elements and computes the scan. In each iteration each work item contributes one element to the final result. The result of the last item is broadcast and applied in the next iteration. As a result each input value is read exactly once. This is important, because the data could reside in global memory, which is slow to access compared to local memory. Memory usage is identical to the work group based algorithm.

In Figure 4.10 the measurements are shown. When using subgroup algorithms inside the work group algorithms, a speedup of 8.9x on NVIDIA and 32x on AMD GPU can be achieved. The speedup on AMD GPU is over 100x if only 32-bit and 64-bit floating point and integer values with work group sizes of 1024 are considered. If DPP instructions



(a) AMD GPU (b) NVIDIA GPU

Figure 4.10.: Inclusive pointer based scan measurements for 32-bit integers. The measurements for *DPP* are not shown for AMD GPU, because it is almost identical to *subgroup*.

are used for 16 B or larger data types on AMD GPUs this can be increased to 33x. The unchecked DPP version reached a speedup of 43x. This speedup is increased to over 150x on work groups of size 1024 for 32-bit integer and floating point values. While this unchecked version cannot be used without the correct launch configuration, it can be used to approximate an upper bound.

5. Conclusion

In this work, all group algorithms and corresponding pointer based versions were implemented, as well as tests checking for correctness. Additionally, benchmarks for sycl-bench [32] where written and used to measure the effectiveness of our optimizations.

The runtimes could be reduced by a factor of up to 33x with respect to our initial implementation. This was achieved by employing low-level instructions specific to the architecture for CPUs, as well as NVIDIA and AMD GPUs.

On CPUs the execution of our optimized algorithms using nd-range kernels is bound by context switching. We still managed to achieve speedup of 1.7x for the reduction and 2.4x for the *inclusive scan* algorithm. These improvements were accomplished by using statically allocated memory on the stack for communication between work items instead of dynamically allocated memory. When using scoped or hierarchical kernels, no fibers are needed, as such the algorithms are no longer bound by context switches. Using OpenMP based vectorization, pointer based reduction could be sped up by a median factor of 2.4 over all values with single precision floating point values reducing the runtime by a factor of almost 8 and double precision floating point values by a factor of almost 4. These speedups match the width of the available vector instructions. In another approach the library xsimd [62] was used, but the speed of OpenMP was not reached with a speedup of 1.9x. For inclusive and exclusive scans, OpenMP was unable to generate vectorized code, but algorithms for vectorized inclusive scans exist [64] and could be implemented using xsimd. The drawback would be that these vectorization would be specific to a single operation.

On NVIDIA GPUs, speedups of 2.1x could be reached for the *reduce* algorithm using shuffle instructions. Using the same method the runtime for the *inclusive scan* algorithm was reduced by a factor of 2.4. The approach proved even more effective for the pointer based scan, as a speedup of 8.9x was measured. All these improvements are likely caused by the higher bandwidth shuffle instructions can deliver, compared to local memory.

The best speedups could be obtained on AMD GPUs. For the pointer based algorithm *any of* the execution time was reduced by 7.3x by accessing the memory in such a way, that the amount of transactions was reduced. Even better results were possible using low-level DPP instructions, which allow to directly access the registers of other work

items. In this work the reduction was sped up by a factor of 8.4 and the inclusive scan by a factor of 2.4. For the pointer based inclusive scan even a factor of 33 was achieved. We were also able to show, that DPP instructions can be used to further improve these functions. When measuring the performance with some checks removed speedups by 15x for the reduction, 4.4x for the inclusive scan and 43x for the pointer based inclusive scan were measured. Although this is not possible in most cases, some operations, like addition, can be implemented without those checks, if the identity element is known.

We also compared the reduction and scan using local values with CUB and rocPRIM. Both libraries provide optimized algorithms for work groups and subgroups, but need to know the kernel configuration at compile time. In general the libraries were faster. A notable exception is the scan using 32-bit integers.

In general the code complexity on GPU was significantly increased to guarantee correct results for all launch configurations and operations. Further speedups should be possible if kernels for some special cases[1] would be compiled and embedded into the executable. In addition to the general version, this would allow to choose the kernel with the best performance for the given launch configuration.

On CPU the performance is mostly bound by context switches between fibers. Maybe the number of context switches could be reduced using compiler support to identify sections, that can be executed without fibers or vectorized.

An interesting further investigation would be to compare the new algorithms in hipSYCL with other SYCL 2020 implementations. At the time of writing other implementations have not yet implemented the group algorithms or in case of DPC++ do not implement the correct signatures yet.

6. Acknowledgements

This thesis was supported by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1134-1 FUGG by providing access to the bwForCluster at Heidelberg University for the benchmarks on the Intel CPUs and NVIDIA GPU,

[1] for example those were DPP instructions can be used without checks

A. Appendix

```
1  #include <stdlib.h>
2  #include <math.h>
3
4  __global__ void zero(double *a, int n) {
5      int id = blockIdx.x*blockDim.x+threadIdx.x;
6
7      if (id < n)
8        a[id] = 0;
9  }
10
11 int main( int argc, char* argv[] )
12 {
13     int n = 100000;
14
15     double *h_a;
16     double *d_a;
17
18     size_t bytes = n*sizeof(double);
19     h_a = (double*)malloc(bytes);
20     cudaMalloc(&d_a, bytes);
21
22     int blockSize, gridSize;
23     blockSize = 1024;
24     gridSize = (int)ceil((float)n/blockSize);
25
26     zero<<<gridSize, blockSize>>>(d_a);
27     cudaMemcpy( h_a, d_a, bytes, cudaMemcpyDeviceToHost );
28
29     cudaFree(d_a);
30     free(h_a);
31     return 0;
32 }
```

Figure A.1.: A simple CUDA example setting all elements in an array to zero. In line 15 and 16 the pointer for the host and device memory is created and in line 20 memory is allocated on the GPU and the pointer stored in d_a. In line 26 the kernel is launched using the desired grid size (nd-range size in SYCL terms) and block size (work group size). In line 27 we copy the values in GPU memory to the host. In line 29 and 30 the memory on the host and the device is freed. The code run on the device is in line 4 to 9. In line 5 the id for each thread is calculated in line 7-8 it is used to set the value inside the array, if the calculated id is not out of bounds.

```
1  #include <stdlib.h>
2  #include <math.h>
3  #include <OpenCL/opencl.h>
4
5
6  const char *KernelSource = "\n" \
7  "__kernel␣void␣zero(__global␣double*␣a,␣const␣int␣n){␣␣␣␣␣␣\n" \
8  "␣␣␣int␣id␣=␣get_global_id(0);␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n" \
9  "␣␣␣if(i␣<␣n)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n" \
10 "␣␣␣␣␣␣␣a[i]␣=␣0;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n" \
11 "}␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n";
12
13 int main( int argc, char* argv[] )
14 {
15     int n = 100000;
16
17     double *h_a;
18     cl_mem d_a;
19
20     cl_device_id device_id;
21     cl_context context;
22     cl_command_queue commands;
23     cl_program program;
24     cl_kernel kernel;
25
26     size_t bytes = n*sizeof(double);
27
28     int gpu = 1;
29     clGetDeviceIDs(NULL, gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU, 1, &device_id,
           NULL);
30     context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
31     commands = clCreateCommandQueue(context, device_id, 0, &err);
32     program = clCreateProgramWithSource(context, 1, (const char **) & KernelSource,
           NULL, &err);
33     clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
34     kernel = clCreateKernel(program, "zero", &err);
35
36     h_a = (double*)malloc(bytes);
37     input = clCreateBuffer(context, CL_MEM_READ_ONLY,  sizeof(double) * n, NULL, NULL);
38
39     clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
40     clSetKernelArg(kernel, 1, sizeof(int), n);
41
42     int blockSize, gridSize;
43     blockSize = 1024;
44     gridSize = (int)ceil((float)n/blockSize)*blockSize;
45     clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &gridSize, &blockSize, 0, NULL,
           NULL);
46     clFinish(commands);
47
48     clEnqueueReadBuffer( commands, d_a, CL_TRUE, 0, sizeof(double) * n, h_a, 0, NULL,
           NULL );
49
50     clReleaseMemObject(d_a);
51     clReleaseProgram(program);
52     clReleaseKernel(kernel);
53     clReleaseCommandQueue(commands);
54     clReleaseContext(context);
55     free(h_a);
56     return 0;
57 }
```

Figure A.2.: A simple OpenCL program equivalent to Figure A.1. The code is a lot more verbose than for CUDA since everything has to be specified. Most of the steps performed explicitly can also be done in CUDA/HIP, but are not always needed. To write the kernel inside the same file as the host code, it is written in a string in lines 6-11. We have the setup and tear down code in lines 20-31 and 50-54, the kernel compilation in lines 32-33 and the kernel preparation and launch in lines 34,39-40 and 45. The explicit wait for the kernel to finish is in line 46.

```
1  void function() {
2    //SYCL code for CPU and GPU
3  #ifdef SYCL_DEVICE_ONLY
4
5    //SYCL code only for GPU
6    #ifdef HIPSYCL_PLATFORM_CUDA
7      //CUDA code
8    #elif defined(HIPSYCL_PLATFORM_ROCM)
9      //HIP code
10   #endif //HIPSYCL_PLATFORM
11
12 #else
13   //SYCL code only for CPU
14 #endif //SYCL_DEVICE_ONLY
15   //SYCL code for CPU and GPU
16 }
```

Figure A.3.: Example on how to write optimized code paths using hipSYCL.

```
1  .LBB275_41:
2          movq    %rdx, %rdi
3          vmovdqa (%rdx), %ymm0
4          vphaddd %ymm0, %ymm0, %ymm0
5          vphaddd %ymm0, %ymm0, %ymm0
6          vextracti128    $1, %ymm0, %xmm1
7          vpaddd  %xmm0, %xmm1, %xmm0
8          vmovd   %xmm0, %edx
9          addl    %edx, %ecx
10         leaq    32(%rdi), %rdx
11         addq    $64, %rdi
12         cmpq    %rbx, %rdi
13         jb      .LBB275_41
14         jmp     .LBB275_43
```

Figure A.4.: The loop used to reduce most values for the manual vectorized reduction. A lot of instructions are needed, like lines 6-7, to reduce the vector register down to a single value.

Bibliography

[1] Ayesha Afzal et al. “Solving Maxwell’s Equations with Modern C++ and SYCL: A Case Study”. In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2018, pp. 1–8.

[2] Axel Alpay. *hipSYCL*. 2021. URL: `https://github.com/illuhad/hipSYCL` (visited on 02/23/2021).

[3] Johnathan Alsop et al. “Lazy release consistency for GPUs”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–14.

[4] Vasco Amaral et al. “Programming languages for data-Intensive HPC applications: A systematic mapping study”. In: *Parallel Computing* 91 (2020), p. 102584.

[5] *AMD CDNA Architecture*. AMD. 2020. URL: `https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf`.

[6] Gene M Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[7] Ansys. *Ansys Fluent: Fluid Simulation Software*. 2021. URL: `https://www.ansys.com/products/fluids/ansys-fluent` (visited on 02/23/2021).

[8] Paul Bauman et al. *Introduction to AMD GPU programming with HIP*. Sept. 6, 2021. URL: `https://www.olcf.ornl.gov/wp-content/uploads/2019/09/AMD_GPU_HIP_training_20190906.pdf` (visited on 02/27/2021).

[9] Brahim Betkaoui, David B Thomas, and Wayne Luk. “Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing”. In: *2010 International Conference on Field-Programmable Technology*. IEEE. 2010, pp. 94–101.

[10] Mark Bohr. “A 30 year retrospective on Dennard’s MOSFET scaling paper”. In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007), pp. 11–13.

[11] boost. *Boost.Fiber*. 2020. URL: `https://www.boost.org/doc/libs/1_75_0/libs/fiber/doc/html/fiber/overview.html` (visited on 03/14/2021).

[12] Andre R Brodtkorb et al. “State-of-the-art in heterogeneous computing”. In: *Scientific Programming* 18.1 (2010), pp. 1–33.

[13] George Chrysos. "Intelő Xeon Phi coprocessor-the architecture". In: *Intel Whitepaper* 176 (2014), p. 43.

[14] Codeplay. *ComputeCpp CE*. 2021. URL: `https://developer.codeplay.com/products/computecpp/ce/home/` (visited on 02/23/2021).

[15] Christopher Daley et al. "A Case Study of Porting HPGMG from CUDA to OpenMP Target Offload". In: *International Workshop on OpenMP*. Springer. 2020, pp. 37–51.

[16] Tom Deakin and Simon McIntosh-Smith. "Evaluating the performance of HPC-style SYCL applications". In: *Proceedings of the International Workshop on OpenCL*. 2020, pp. 1–11.

[17] Robert H Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[18] Hadi Esmaeilzadeh et al. "Dark silicon and the end of multicore scaling". In: *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE. 2011, pp. 365–376.

[19] Fritzchens Fritz. *Fritzchens Fritz*. 2021. URL: `https://www.flickr.com/photos/130561288@N04/` (visited on 03/11/2021).

[20] David Geer. "Chip makers turn to multicore processors". In: *Computer* 38.5 (2005), pp. 11–13.

[21] Andrew Gozillon et al. "triSYCL for Xilinx FPGA". In: *The 2020 International Conference on High Performance Computing & Simulation*. IEEE. 2020.

[22] GROMACS. *GROMACS*. 2021. URL: `http://www.gromacs.org/` (visited on 02/23/2021).

[23] Khronos Group. *The OpenCL Specification*. 2020. URL: `https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#_host_environment_and_thread_safety` (visited on 03/14/2021).

[24] Intel. *Data Parallel C++ (DPC++)*. 2021. URL: `https://software.intel.com/content/www/us/en/develop/tools/oneapi/data-parallel-c-plus-plus.html` (visited on 02/23/2021).

[25] Intel. *Intel SDK For OpenCL Applications*. 2021. URL: `https://software.intel.com/content/www/us/en/develop/tools/opencl-sdk.html` (visited on 02/23/2021).

[26] *Introducing RDNA Architecture*. AMD. 2016. URL: `https://www.amd.com/system/files/documents/rdna-whitepaper.pdf`.

[27] Pekka Jääskeläinen et al. "Reducing context switch overhead with compiler-assisted threading". In: *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. Vol. 2. IEEE. 2008, pp. 461–466.

[28] Yinan Ke, Mulya Agung, and Hiroyuki Takizawa. "neoSYCL: a SYCL implementation for SX-Aurora TSUBASA". In: *The International Conference on High Performance Computing in Asia-Pacific Region*. 2021, pp. 50–57.

[29] Mikhail Khalilov and Alexey Timoveev. "Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU". In: *Journal of Physics: Conference Series*. Vol. 1740. 1. IOP Publishing. 2021, p. 012056.

[30] Kazuhiko Komatsu et al. "Evaluating performance and portability of OpenCL programs". In: *The fifth international workshop on automatic performance tuning*. Vol. 66. 2010, p. 1.

[31] Kelin J Kuhn. "Moore's Law Past 32nm: Future Challenges in Device Scaling". In: *2009 13th International Workshop on Computational Electronics*. IEEE. 2009, pp. 1–6.

[32] Sohan Lal et al. "SYCL-Bench: A Versatile Single-Source Benchmark Suite for Heterogeneous Computing". In: *Proceedings of the International Workshop on OpenCL*. IWOCL '20. Munich, Germany: Association for Computing Machinery, 2020. ISBN: 9781450375313. DOI: 10.1145/3388333.3388669. URL: https://doi.org/10.1145/3388333.3388669.

[33] Victor W Lee et al. "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU". In: *Proceedings of the 37th annual international symposium on Computer architecture*. 2010, pp. 451–460.

[34] Ewing Lusk and Katherine Yelick. "Languages for high-productivity computing: the DARPA HPCS language project". In: *Parallel Processing Letters* 17.01 (2007), pp. 89–102.

[35] Alok Mishra et al. "Benchmarking and evaluating unified memory for OpenMP GPU offloading". In: *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. 2017, pp. 1–10.

[36] Tulika Mitra. "Heterogeneous multi-core architectures". In: *Information and Media Technologies* 10.3 (2015), pp. 383–394.

[37] Ana Moreton–Fernandez, Hector Ortega–Arranz, and Arturo Gonzalez–Escribano. "Controllers: An abstraction to ease the use of hardware accelerators". In: *The International Journal of High Performance Computing Applications* 32.6 (2018), pp. 838–853.

[38] Aaftab Munshi. "The opencl specification". In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314.

[39] NVIDIA. *CUB*. 2021. URL: https://nvlabs.github.io/cub/ (visited on 02/27/2021).

[40] NVIDIA. *CUDA C++ Programming Guide*. 2021. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (visited on 02/27/2021).

[41] NVIDIA. *cuda-zone*. 2021. URL: https://developer.nvidia.com/cuda-zone (visited on 02/23/2021).

[42] NVIDIA. *OpenCL*. 2021. URL: https://developer.nvidia.com/opencl (visited on 02/23/2021).

[43] *NVIDIA A100 Tensor Core GPU Architecture*. NVIDIA. 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[44] Kenneth Obrien et al. "A survey of power and energy predictive models in HPC systems and applications". In: *ACM Computing Surveys (CSUR)* 50.3 (2017), pp. 1–38.

[45] *OpenMP Application Program Interface*. OpenMP Architecture Review Board. 2013. URL: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[46] John D Owens et al. "A survey of general-purpose computation on graphics hardware". In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library. 2007, pp. 80–113.

[47] Phitchaya Mangpo Phothilimthana et al. "Portable performance on heterogeneous architectures". In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 431–444.

[48] LLVM project. *clang*. 2021. URL: https://clang.llvm.org/ (visited on 03/07/2021).

[49] *Provisional SYCL Specification*. Khronos Group. 2020. URL: https://www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf.

[50] RadeonOpenCompute. *ROCm*. 2021. URL: https://github.com/RadeonOpenCompute/ROCm (visited on 02/23/2021).

[51] *Reference Guide, AMD Graphics Core Next Architecture, Generation 3*. AMD. 2016. URL: http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf.

[52] ROCmSoftwarePlatform. *rocPRIM*. 2021. URL: https://github.com/ROCmSoftwarePlatform/rocPRIM (visited on 03/07/2021).

[53] Ben Sander. *AMD GCN Assembly: Cross-Lane Operations*. 2016. URL: https://gpuopen.com/learn/amd-gcn-assembly-cross-lane-operations/ (visited on 03/07/2021).

[54] VASP Software. *Vienna Ab initio Simulation Package*. 2021. URL: https://www.vasp.at/ (visited on 02/23/2021).

[55] Lukas Sommer, Jens Korinth, and Andreas Koch. "OpenMP device offloading to FPGA accelerators". In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2017, pp. 201–205.

[56] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), p. 66.

[57] StreamHPC. *OpenCL Wrappers*. URL: https://streamhpc.com/knowledge/for-developers/opencl-wrappers/ (visited on 03/11/2021).

[58] *SYCL Specification*. Khronos Group. 2020. URL: https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[59] TOP500. *TOP500 - November 2020*. 2020. URL: https://top500.org/lists/top500/2020/11/ (visited on 02/16/2021).

[60] Angelos Trigkas. "Investigation of the OpenCL SYCL programming model". In: *Master's thesis. Edinburgh Parallel Computing Centre, University of Edinburgh* (2014).

[61] Xilinx. *triSYCL*. 2021. URL: https://github.com/Xilinx/triSYCL (visited on 02/23/2021).

[62] xtensor-stack. *xsimd*. 2021. URL: https://github.com/xtensor-stack/xsimd (visited on 03/05/2021).

[63] Yohei Yamada and Shintaro Momose. "Vector engine processor of NECs brand-new supercomputer SX-Aurora TSUBASA". In: *Proceedings of A Symposium on High Performance Chips, Hot Chips*. Vol. 30. 2018, pp. 19–21.

[64] Wangda Zhang, Yanbin Wang, and Kenneth A Ross. "Parallel Prefix Sum with SIMD". In: *Algorithms* 5 (), p. 31.