Performance Evaluation and Improvements of the *PoCL* Open-Source OpenCL Implementation on Intel CPUs

Tobias Baumann
Zuse Institute Berlin
Berlin, Germany
tobias.baumann@zib.de

Matthias Noack
Zuse Institute Berlin
Berlin, Germany
noack@zib.de

Thomas Steinke
Zuse Institute Berlin
Berlin, Germany
steinke@zib.de

ABSTRACT

The Portable Computing Language (PoCL) is a vendor independent open-source OpenCL implementation that aims to support a variety of compute devices in a single platform. Evaluating PoCL versus the Intel OpenCL implementation reveals significant performance drawbacks of PoCL on Intel CPUs – which run 92 % of the TOP500 list. Using a selection of benchmarks, we identify and analyse performance issues in PoCL with a focus on scheduling and vectorisation. We propose a new CPU device-driver based on Intel Threading Building Blocks (TBB), and evaluate LLVM with respect to automatic compiler vectorisation across work-items in PoCL. Using the TBB driver, it is possible to narrow the gap to Intel OpenCL and even outperform it by a factor of up to 1.3× in our proxy application benchmark with a manual vectorisation strategy.

CCS CONCEPTS

• **General and reference** → **General conference proceedings**; • **Software and its engineering** → **Parallel programming languages**; **Multithreading**; **Scheduling**; **Compilers**.

KEYWORDS

Parallel Programming, HPC, Performance, Optimization, Scheduling, Vectorization, SIMD, OpenCL, PoCL, LLVM, TBB

ACM Reference Format:
Tobias Baumann, Matthias Noack, and Thomas Steinke. 2021. Performance Evaluation and Improvements of the *PoCL* Open-Source OpenCL Implementation on Intel CPUs. In *International Workshop on OpenCL (IWOCL'21), April 27–29, 2021, Munich, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3456669.3456698

1 INTRODUCTION

Of all the TOP500 list [4] systems, 91.8 % feature Intel CPUs. Current and foreseeable trends in the TOP500 include i) an increasing number of accelerators and co-processors - 27.2 % of all systems have accelerators and co-processors installed, ii) an increasing number of systems with ARM CPUs - starting with the Fujitsu A64FX processors of the current number one system, and iii) a raise in x86 CPUs from AMD.

IWOCL'21, April 27–29, 2021, Munich, Germany

ACM ISBN 978-1-4503-9033-0/21/04…$15.00
https://doi.org/10.1145/3456669.3456698

Given this increasing diversity of HPC architectures, code portability is more important than ever. In this context, OpenCL [11] with its wide practical vendor support and recent upgrade to version 3.0 is an important framework for both, direct application development as well as serving as a backend for higher level abstraction like SYCL or Intel oneAPI. Its portability-first design supports running the same code on CPUs, GPUs, and other accelerators like FPGAs, allowing programmers to work with a single code base for a multitude of architectures. However, achieving performance portability still takes some effort.

OpenCL implementations are typically provided by the hardware vendors, but are not always maintained with the same enthusiasm as other, less portable solutions. Hence, the specific level of OpenCL support varies across different platforms, and users depend on the hardware vendor's policies and decisions. The Portable Computing Language (PoCL) has been introduced by Jääskeläinen et al. [8]. As vendor independent open-source OpenCL implementation, PoCL aims to be easily portable to OpenCL-capable devices, offering an alternative to vendor provided solutions or a lack thereof. The core runtime of PoCL is a CPU backend which has been used as an OpenCL implementation on ARM in previous studies [5, 7, 21, 23, 24]. PoCL also supports certain GPUs and application-specific instruction set processors.

With these characteristics, PoCL fills a gap in the OpenCL ecosystem. For instance, using LLVM as backend, PoCL provided AVX-512 on Intel x86 targets in December 2017, almost a year before the Intel OpenCL implementation gained support for AVX-512 in October 2018, which did not include its Xeon Phi (KNL) CPUs. This demonstrates that PoCL as an open-source project has the potential to benefit from the community and other projects to adapt quickly, while vendors might have their priorities and resources elsewhere. However, the adaptation of PoCL, especially in HPC, depends on its performance as compared to vendor solutions and other frameworks.

In this paper, we evaluate PoCL and the Intel OpenCL implementation regarding performance on Intel CPUs. We employ three benchmarks of different characteristics to analyse performance differences: i) a synthetic micro benchmark, ii) a proxy application benchmark, and iii) a real world application benchmark. We implement and evaluate, as well as outline, solutions for improving the performance of PoCL and narrowing the gap to the Intel OpenCL implementation.

Specifically, we provide an in-depth discussion of different scheduling strategies and propose a new device for PoCL using the *Threading Building Blocks* (TBB) open source library. In order to enable data-parallel execution of work-items as a generic vectorisation strategy, rather than opportunistic vectorisation inside a kernel

function, we analyse the state of vectorisation capabilities available through LLVM. Additionally, we describe our solutions for some performance bugs we encountered along the way.

This paper is structured as follows: Following related work in Section 2, we introduce OpenCL and the utilised OpenCL implementations in Section 3, and the benchmark setup in Section 4. In Section 5, we analyse the performance differences between PoCL and the Intel OpenCL implementation. After a deeper analysis of the scheduling in Section 6, we propose a TBB device for PoCL in Section 7. We analyse LLVM's vectorisers with respect to work-item vectorisation in Section 8, followed by a description of further contributions to the PoCL project in Section 9. We provide a performance evaluation in Section 10 and conclude with a summary in Section 11.

2 RELATED WORK

Performance aspects of OpenCL on CPUs are addressed in the following studies: Karrenberg et al. showed how to improve the performance of OpenCL on CPUs inside their own OpenCL driver derived from LLVM and AMD SDK [10]. Lee et al. evaluated the performance of Intel OpenCL on Westmere EP CPUs giving several hints on how to improve performance [12].

In [9], Jääskeläinen et al. investigated how well task-level parallelism is supported in AMD's and Intel's OpenCL implementations and compare them to PoCL on x86 CPUs.

Performance comparisons between PoCL and other non-vendor OpenCL implementations, or similar programming models, can be found in the following studies: Zhang et al. demonstrated a lower scheduling overhead in their OpenCL implementation for the Matrix-2000 architecture compared to PoCL [27]. Haidl et al. compared the performance of their framework to PoCL on ARM, IBM Power8 CPUs and to Intel, PoCL on Sandy Bridge EP, KNL [6].

PoCL has been used by a number of projects on ARM CPUs, e.g. in combination with FPGAs [7, 24] and in combination with ARM Mali GPUs [5, 23]. Rafique and Schneider describe PoCL as single-threaded but it remains unclear why PoCL failed to exploit task-level parallelism in this particular study [21].

In this work we focus on two major performance drawbacks discovered in recent PoCL versions on x86 CPUs, i.e., limitations in the work-group scheduling and code generation phase, both not addressed sufficiently in previous studies.

3 OPENCL CONCEPT AND USED IMPLEMENTATIONS

OpenCL (Open Computing Language) is an open standard for cross-platform, parallel programming introduced in 2009, and maintained by the Khronos Group. Before looking at the Intel and PoCL implementation of OpenCL, we provide a quick summary of its main concepts and terminology.

The OpenCL platform model (displayed in Figure 1) is designed to present a uniform abstraction that can be mapped to different kinds of parallel processors.

An OpenCL application contains *host code* and *device code*. Host code (written in C/C++) is executed on the *host* CPU to set-up the OpenCL environment and to submit work to OpenCL *devices* (usually accelerators or the host CPU itself). The device code (also called *kernel code*) is written in OpenCL C (a C99 dialect), compiled at runtime for a specific device and then executed on that device.

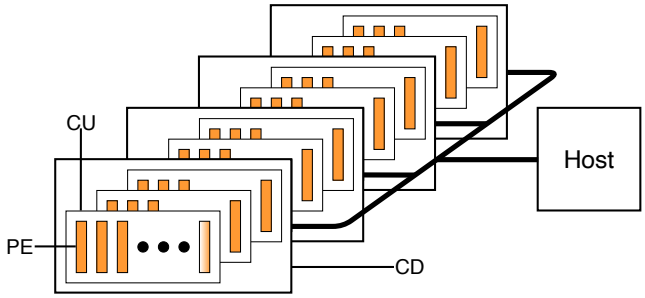


Figure 1: The OpenCL platform model: One host plus one or more compute devices (CD) each with one or more compute units (CU) composed of one or more processing elements (PE) [11].

The data elements (or *work-items*) to process are organised in *work-groups* and *NDRanges* as shown in Figure 2. NDRanges represent the problem's dimension and size. Each work-item corresponds to one instance of the kernel function, all of which are executed in parallel. Work-items within a work-group are typically processed together and can be synchronised. Both, work-groups and NDRanges, can have up to three dimensions with certain OpenCL implementation or hardware-specific size-limits. The size of a work-group, i.e. the number of work-groups for a given NDRange, can be chosen implicitly by the OpenCL implementation or explicitly by the programmer.

The scope of this work are CPUs as compute devices. The relationship between the data-parallelism entities from Figure 2 and the platform model entities from Figure 1 mapped to hardware components is shown in Table 1. An OpenCL implementation for CPUs has to exploit the available task (threading) and data parallelism (SIMD) of the architecture.

Table 2 provides an overview of the main properties of the OpenCL implementations used in this study. Threads are handled by an underlying threading library. Typically, there is one software thread running on each compute unit, i.e. hardware thread. The

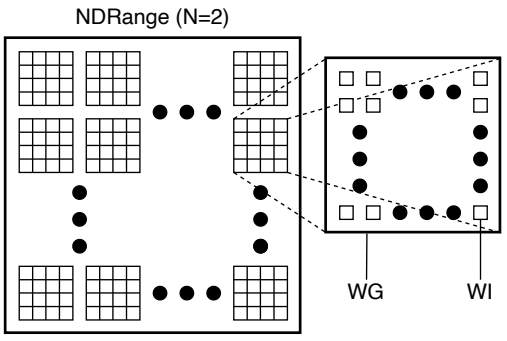


Figure 2: Data-Parallelism in OpenCL: Work-items (WI) are grouped together in work-groups (WG), which are arranged in N-Dimensional Ranges (NDRange).

Table 1: Relationship between the data-parallelism entities from Figure 2 and the platform model entities from Figure 1 mapped to hardware when using a CPU as compute device, e.g. work-groups are processed by compute units in the OpenCL model which map to hardware threads.

data model entity	platform model entity	hardware unit
NDRange	compute device	CPU
work-group	compute unit	hardware thread
work-item	processing element	execution unit

Table 2: OpenCL implementations overview.

name	threading lib	math lib	compiler
Intel 2021.1	TBB	SVML	Intel LLVM 12.0.0git*
PoCL 1.4	pthread	SLEEF/libclc	LLVM 9
PoCL 1.6	pthread	SLEEF/libclc	LLVM 11

(*as indicated by version string from binary)

math library provides vectorised implementations of the OpenCL built-in function to properly utilise the SIMD execution units. An underlying compiler framework is used for the online compilation of the kernel code. Clang/LLVM, which provide an OpenCL C frontend and a large variety of target architectures are the defacto standard here. The following two subsections introduce Intel OpenCL and PoCL in more detail.

3.1 Portable Computing Language

The Portable Computing Language (PoCL) is a vendor independent open-source implementation of the OpenCL standard aiming to be easily portable and to improve performance portability of OpenCL kernels [8]. PoCL uses Clang as an OpenCL C front-end and LLVM as its kernel compiler implementation, as well as portability layer. Using the advantages of open source, this makes it easy to implement OpenCL support for a specific target architecture (device), if LLVM has a backend for it.

Each PoCL release is compiled against a specific LLVM version, and a new PoCL version is usually released after each LLVM major release supporting its latest changes. PoCL 1.4, which is the baseline we started this work with, uses LLVM 9, while the most recent PoCL 1.6 uses LLVM 11.

PoCL uses a combination of *SIMD Library for Evaluating Elementary Functions* (SLEEF) [22] and LLVM's *libclc* [1] for vectorised math functions. The kernel compiler is first invoked when calling the usual *clBuildProgram()* and *clCompileProgram()* API entries. The final assembly code is generated when the number of work-items is known at the time the kernel is executed. PoCL transforms the kernel function into a work-group function, but applies no custom vectorisation steps. Depending on the kernel at hand, LLVM's loop vectoriser and SLP vectoriser might or might not find opportunities to vectorise parts of its code. The alternative is a manual vectorisation scheme using OpenCL vector data types as described in Section 8.

PoCL's default driver for CPUs running Linux is called the *pthread device* which employs the pthread library and creates a thread for each compute unit (i.e. hardware thread) on initialisation in addition to any application threads.

If no work-group size is specified by the application, PoCL first tries to maximise the work-group size to a predefined limit of 4096 work-items taking a preferred work-group multiple of 8 work-items for vectorisation into account. In a second step, the work-groups size is reduced to ensure that there is at least one work-group for each compute unit. However, the work-group size will not be reduced below a predefined minimum work-group size of 32 work-items. This results in a parallelisation threshold of $32 \times numberOfComputeUnits$ work-items, below which the PoCL pthread driver will not run in parallel. This can be omitted by specifying a work-group size manually and not leaving the choice to PoCL.

3.2 Intel OpenCL for CPUs

Intel distributes a number of different OpenCL implementations with open-source (GPU) and proprietary (CPU) licences. At the time of writing, the latest CPU-only OpenCL implementation is the proprietary *Intel CPU Runtime for OpenCL Applications*. We use version 2021.1.2-266 of this runtime to conduct all our measurements and always refer to this runtime when using the term *Intel OpenCL implementation*. This OpenCL implementation reports a `CL_DRIVER_VERSION` of 2020.11.12.0.14_160000 and is available as part of the Intel oneAPI framework.

The Intel OpenCL implementation uses an LLVM-based OpenCL compiler. The implicit vectorisation module of this compiler first generates LLVM-IR and then widens each instruction to make use of vector instructions. For this purpose, the vectoriser performs a sequence of LLVM transformations and analysis passes proprietary to Intel. As mentioned in [18], a number of constrains must be met to generate the best result when using an automatic vectorisation strategy, e.g. using a work-group size that is a multiple of the architecture's vector width in dimension zero. The same vectorisation scheme can be implemented manually by using OpenCL vector types as described in Section 8.

The Intel OpenCL implementation makes use of the Intel *Short Vector Math Library* (SVML) to implement the OpenCL built-in functions and the Intel *Threading Building Blocks* (TBB) [26] library to address task parallelism across CPU cores, including the scheduling of work-groups to threads. The Intel OpenCL implementation never generates more software threads than hardware threads including the application thread that is calling the TBB library.

4 OPENCL BENCHMARK SETUP

In this section, we describe the three benchmarks that we adopt to analyse and evaluate PoCL and the Intel OpenCL implementation. As shown in Table 3, the benchmark selection contains one micro benchmark and two application benchmarks. One of the application benchmarks has a rather regular and balanced workload, while the other one is imbalanced (cf. Section 5). All benchmarks are open source and use a common OpenCL helper library and benchmark timer [16].

From the application's point of view there are two strategies to achieve vectorisation: i) *automatic vectorisation* using standard scalar data types and relying on the vectorisation capabilities of the OpenCL implementation or ii) *manual vectorisation* using OpenCL's vector data types with a specific vector length e.g. `double8`.

Table 3: Benchmark overview.

name	type	workload	instruction mix	licence
op	synthetic	balanced	per operation	Boost
hexciton	proxy app	balanced	FMA	Boost
raytracing	real world	imbalanced	various	MIT

Table 4: Vectorisation modes available with the *op* and *hexciton* benchmarks and their properties for an example problem size of 16 data elements. Vectorisation can be performed automatically by the compiler or manually by substituting scalar data types with OpenCL's vector data types, where each work-item then processes multiple data items.

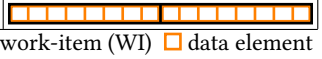
vec. mode	vec. length	#WIs	logical data layout
automatic	1	16	
manual4	4	4	
manual8	8	2	

Table 4 shows the resulting properties when applying these vectorisation strategies to a problem size of 16 data elements. Note that the number of work-items is calculated by dividing the number of data elements by the vector length.

4.1 Micro Benchmark: op

The *op_bechmark* [15] is a micro, low-level benchmark designed to measure the performance of individual operations (e.g. +, ÷) and OpenCL built-in functions (e.g. `sqrt`, `sin`). Each operation or function is applied to a global number of data elements for a specified number of iterations inside each individual kernel, which is repeated for multiple *runs*.

The number of data elements is chosen in such a way that all hardware parallelism available can be saturated by the number of work-items available without any under- or uneven over-subscription of the execution units. The goal is to measure the individual operation's performance as good as possible, while the relatively small work-load of each kernel can reveal runtime effects otherwise hidden by larger problem sizes.

4.2 Proxy Application Benchmark: hexciton

The *hexciton_benchmark* [14] is a proxy benchmark for the hexciton computation of the DM-HEOM [17] real-world code, developed by Noack et al. during a case study on optimising the hexciton kernel code for multicore and many-core processors [18]. It features the same kernel in different variants and stages of optimisation for different target devices. Each kernel is available as a scalar version to be automatically vectorised by the compiler, and a manually vectorised version that uses vector data types. Work-group sizes can be either enforced, or left for the OpenCL implementation to decide. Beside the OpenCL version, an OpenMP version resembling the OpenCL runtime as described in Section 8, is available. It uses the KART library [19] for compiling kernels at runtime, which is crucial for comparable performance as it enables the same level of code optimisation as OpenCL provides, e.g. using runtime data as compile-time constants. The automatic vectorisation kernels are fine-tuned to meet the requirements of the Intel OpenCL implementation for automatic vectorisation (cf. Section 3.2). The CPU optimised kernels of the benchmark use an AoSoA data layout of interleaved matrices, which allows for contiguous vector load/store operations instead of costly gather/scatter instructions, if recognised by the compiler. We run the manual vectorisation kernels with a vector sizes of 8 doubles aiming for AVX-512 vectorisation. The *hexciton_benchmark* uses a fixed number of 512×1024 small 7×7 matrices (data elements), which is derived from real-world problem sizes. The instruction mix mainly consists of streamlined FMA instructions.

4.3 Real World Application Benchmark: raytracing

The *raytracing_benchmark* [25] is a benchmark derived from a raytracing code published on Github. To use it as a benchmark, we modified it by removing interactivity, porting it from Windows to Linux and adding time measurements. The code has not been further optimised from the source and is supposed to reflect an average real-world application code, not optimised towards a specific hardware.

The benchmark renders a static scene (see Figure 3) which consists of a Stanford dragon placed on a face with a chequerboard pattern in front of a background picture. Benchmark parameters are *width*, *height* of the image and the number of frames (*iterations*) to render. The quality of the image increases with each iteration. In each iteration, the benchmark calculates a slightly shifted ray per pixel. The amount of work that has to be performed per ray depends on how many times the ray is reflected (up to 5), if the ray hits the sky or not, as well as the material of the intersected face, if the ray does not hit the background. Thus, the benchmark is imbalanced with respect to the work per work-item. This benchmark has no manual vectorisation mode. As the kernel uses complex data structures and work-items can take different code paths, we do not expect that the kernel can be vectorised properly. The kernel contains trigonometric and various other built-in functions.

4.4 Methodology

All measurements are obtained on a dual socket Intel Xeon Gold 6138 (Skylake) system comprising 20 cores per socket with hyper-threading enabled, resulting in 80 hardware threads. This Skylake system has two AVX-512 units per core, i.e. one AVX-512 unit per hardware thread.

Each benchmark performs multiple iterations of the measured kernels and is executed 10 times in total. Runtimes shown reflect the arithmetic mean over all iterations and benchmark runs.

The employed vectorisation modes and number of work-items for each benchmark are displayed in Table 5. The number of work-items for the `manual8` mode of the *op_benchmark* equals PoCL's parallelisation threshold ($32 \times 80 = 2560$, cf. Section 3.1).

For the *hexciton_benchmark*, we selected the *aosoa_naive_constants_perm* and *manual_aosoa_constants_perm* kernels for comparison. For the OpenMP version of the *hexciton_benchmark*, we add *aosoa_naive_constants_direct_perm*, *manual_aosoa_constants_direct_perm* and *aosoa_constants_direct_perm2to5*.

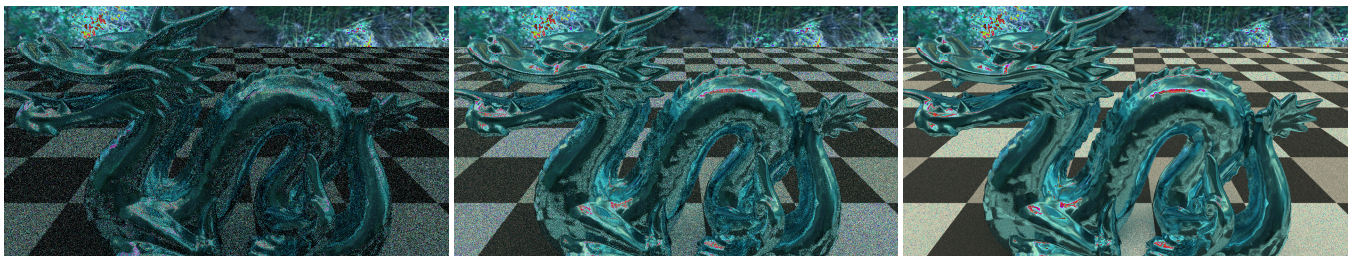


Figure 3: The images rendered by the *raytracing_benchmark* after 1, 10 and 100 iterations (left to right).

Table 5: Employed benchmark parameters.

benchmark	vec. mode	#WIs	data (read/written)
op	automatic	20480	320 KiB/160 KiB
	manual4	5120	
	manual8	2560	
hexciton	automatic	524288	392 MiB/392 MiB
	manual8	65536	
raytracing	automatic	≈8.3M	≈92 MiB/≈127 MiB

5 PERFORMANCE ANALYSIS: POCL VS. INTEL OPENCL

In this section, we analyse the performance of PoCL compared to the Intel OpenCL implementation. Because our measurements show that PoCL lacks significant performance, we aim to identify potential bottlenecks and strategies to improve the performance of PoCL.

5.1 Vectorisation of Built-in Functions

As PoCL and Intel use different libraries for the vectorisation of the OpenCL built-in functions, we start by measuring the performance differences of individual operations and built-in functions with the *op_benchmark*.

Initial measurements reveal that PoCL 1.4 has a lower performance than Intel across all scenarios. Investigating the PoCL source code, we discover an integer division bug in the scheduler code that causes an uneven distribution of work-groups to threads, especially when there are very few work-groups per thread (cf. Section 9.2). Along the way, we discovered a minor off-by-one issue related to the PoCL parallelisation threshold (cf. Section 9.1) that caused a sequential execution in case of manual vectorisation with a vector length of 8. Both adjustments are included in the PoCL 1.5 release.

Table 6 shows the average kernel execution time of selected operations of the *op_benchmark* across different vectorisation modes with PoCL 1.6 LLVM 11 compared to Intel. For most operations, Intel performs best with automatic vectorisation and worst with manual vectorisation with a vector size of 4 double elements while PoCL performs best with manual vectorisation with a vector size of 8 double elements and worst with automatic vectorisation.

When comparing PoCL to Intel, the performance gap is the biggest with automatic vectorisation and the lowest with manual vectorisation with a vector size of 8 double elements. Investigating the assembly code generated by PoCL, we found that PoCL fails to properly vectorise in automatic vectorisation mode while generating the expected vector instructions in manual vectorisation mode.

Table 6: *op_benchmark*: Average kernel execution time [ms] of selected operations after fixing the scheduler's integer-division bug. Coloured cells denote the minimum value across all vectorisation modes used to calculate the ratio.

implementation	vec. mode	`fma`	`sqrt`	`sin`
Intel	automatic	**0.59**	5.26	**5.66**
	manual4	1.41	**4.79**	19.02
	manual8	0.96	5.92	10.70
PoCL 1.6 LLVM 11	automatic	7.54	17.70	145.34
	manual4	2.72	6.74	45.19
	manual8	**2.08**	**6.66**	**28.57**
PoCL/Intel performance ratio:		**3.53×**	**1.39×**	**5.04×**

For both, automatic and manual vectorisation, vectorised versions of OpenCL's built-in functions are required. An alternative to the SLEEF/libclc libraries used by PoCL for vectorised math functions on CPUs would be using Intel's SVML, which can be enabled by passing `fvec-lib=SVML` to Clang. LLVM 12 will add support for the GNU libmvec `fvec-lib=libmvec` as well, which should be available on most Linux systems as part of the *libc*. However, this would probably require a larger code change in PoCL.

5.2 Observed OpenCL Runtime Issues

Based on the first assessment with the *op_benchmark*, we further check PoCL for potential issues that arise from the runtime behaviour of the OpenCL implementations.

The *op_benchmark* includes a nop kernel and the *hexciton_benchmark* includes an empty kernel. These two scenarios are ideal to reveal runtime overhead as they do not perform any computation on top of the kernel execution cost of the OpenCL implementation. Both benchmarks show that PoCL has a higher kernel execution overhead than Intel.

To confirm this observation, we scale the iterations of the multiplication operation of the *op_benchmark* from 0 to 100000 and apply a linear regression model (see Figure 4). This analysis reveals that PoCL has set-up costs of about 1.5 ms that do not exist for Intel. Also note that PoCL has a higher slope than Intel which indicates that Intel has a better code generation.

As described in Section 3.1, PoCL performs a final compilation step when launching a kernel. For this final compilation step, PoCL

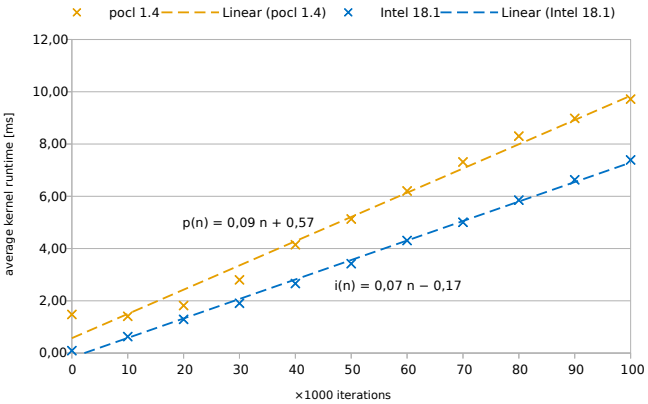


Figure 4: Linear regression model for the multiply operation of the *op_benchmark* using automatic vectorisation and an explicit work-group size of 1 (2560 data elements, 45 runs excluding 5 warmups).

takes runtime parameters specified on kernel launch into account, e.g. work-group size. We expect Intel to perform a similar approach as these runtime parameters could affect vectorisation. We do not measure overheads that occur from last-minute compilation or due to (kernel-)caching effects, because we exclude warm-up measurements as described in Section 4.

As PoCL and Intel use different libraries for threading, we further analyse the scheduling behaviour in the following section.

6 SCHEDULING

Mapping work-groups to compute units poses a typical scheduling problem that each OpenCL implementation has to solve. While each work-group typically consists of the same number of work-items, the time needed to execute each work-item can vary, both, inside a work-group and across work-groups. Depending on the distribution of these runtime variations and the work-group size, the work-group runtimes can also vary, or be rather uniform.

We call a scenario in which all work-groups need the same time to execute a *balanced workload*. Otherwise, we call it an *imbalanced workload*. Since this behaviour depends on the application, an OpenCL implementation has to provide a solution that works well for both scenarios by default. Additional tuning parameters can be provided for more advanced users to control the scheduling behaviour of the OpenCL Implementation.

In the following subsections, we discuss how PoCL and comparable frameworks (cf. Table 7) solve this problem of application-dependent scheduling. Typically, the time needed to execute a scheduling item is orders of magnitude higher than the time needed for scheduling. Thus, we omit scheduling overhead in the following theoretical considerations except when stated explicitly.

6.1 OpenMP

OpenMP [20] is an open standard that introduces annotations to parallelise loops amongst other things. The scheduling process of OpenCL can be mapped to the scheduling process of OpenMP as demonstrated in [18]. The following example shows an annotated

Table 7: Scheduling across frameworks.

framework	scheduling strategies	scheduling item
PoCL	default	work-groups
OpenMP	static, dynamic, guided	loop iterations
TBB partitioner	static, simple, auto, affinity	loop iterations
TBB task scheduler	work stealing	tasks

for-loop applying a function to each element of a data range, e.g. an array:

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < num_work_groups; ++i) {
    run_work_group(work_group_array[i]);
}
```

The entity names in the above example indicate how OpenCL can be mapped to OpenMP, i.e. in OpenCL, the for-loop and its iterations would become a kernel-launch over an NDRange representing the global and the local data ranges to process. The local size within the NDRange defines the work per work-group and is the equivalent to an element of the `work_group_array`. The OpenMP scheduler schedules loop iterations to threads, while an OpenCL implementation schedules work-groups to threads (cf. Table 7). The number of iterations or work-groups scheduled to a thread in one scheduling step is called *chunk size*.

The OpenMP specification [20] defines three scheduling strategies of interest that can be selected with the `schedule` clause: `static`, `dynamic` and `guided`. For `static`, "iterations are divided into chunks [...] and the chunks are assigned to the threads [...] in a round-robin fashion in the order of the thread number". For `dynamic`, "each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed", resulting in each thread completing multiple passes through the scheduler. `guided` is similar to `dynamic`, except that "the size of each chunk is *proportional* to the number of unassigned iterations divided by the number of threads". Note that threads pass through the scheduler sequentially and the number of iterations left to assign decreases after each pass of a thread. This leads to different numbers of iterations being assigned to each thread on the first pass.

The above definition of `guided` has certain degrees of freedom for its implementation. First, the temporary value of iterations to assign given as:

$$\text{proportionality constant} \times \frac{\text{unassigned iterations}}{\text{number of threads}}$$

is likely to be a floating point value. Since only whole work-groups can be scheduled, a rounding function has to be chosen. Second, the proportionality constant can have any value of range $(0, 1]$. Values above one decrease the performance of the schedule for balanced workloads, while for imbalanced workloads the average execution time of the iterations scheduled to the first thread must be below the average execution time of all iterations to not decrease the performance of the schedule. A value of zero can also occur (depending on the rounding function) when the proportionality constant is low or the number of unassigned iterations is low. This would imply that no work-groups are ever scheduled, which is prevented by always rounding up or defining a minimum amount of iterations

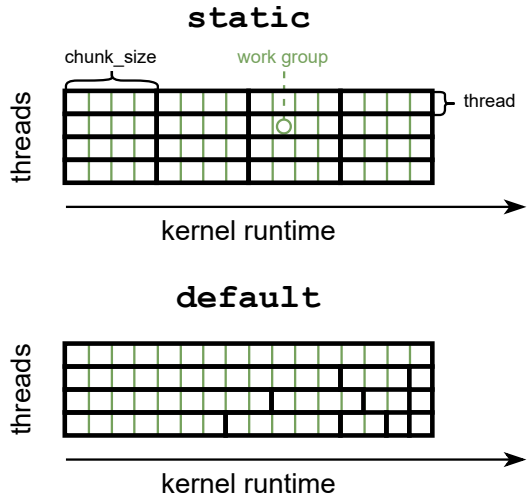


Figure 5: Schedules generated by different strategies for a balanced workload (64 work-groups, 4 threads). Both strategies result in evenly distributed workloads across threads. The *chunk size* chosen for `static` is 4. Higher values (8, 16) are possible to reduce potential scheduling overhead (neglected in the figure).

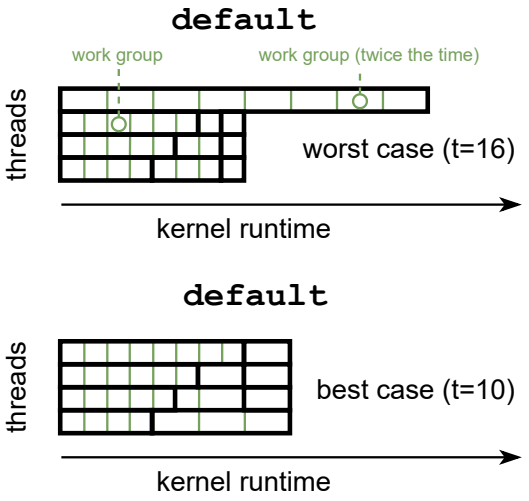


Figure 6: Worst case and best case of scheduling 32 imbalanced work-groups (8 of them need twice the time) with `default`, 4 threads, no scheduling overhead.

to assign to a thread in a scheduling step. The optimal value for the proportional constant depends on the degree of imbalance of the workload. However, OpenMP does not allow to control this constant.

The Intel OpenMP implementation describes `guided` as "each thread gets a big chunk on the first pass, and an increasingly small chunk on the next pass" [13]. This wording suggests that the number of iterations scheduled on the first pass is the same for all threads. It is reasonable to do so, as all threads are idle on the first pass.

`static` is better for balanced workloads while `dynamic` and `guided` are better for imbalanced workloads. `guided` tries to reduce scheduling overhead of less imbalanced workloads by starting with a huge chunk size followed by subsequent reductions. For each strategy, the chunk size can be specified to concretise the behaviour of the respective strategy.

6.2 PoCL

OpenCL schedules work-groups instead of iterations. The chunk size is the number of work-groups scheduled in one scheduling step.

PoCL's only scheduling strategy `default` corresponds OpenMP's `guided` with a proportionality constant of one and *ceil()* as rounding function, i.e. the number of work-groups assigned to a thread equals the number of unassigned work-groups divided by the number of threads, with one thread running per compute unit (CU).

Without scheduling overhead, this strategy performs well for balanced workloads (see Figure 5), but depending on the work distribution of imbalanced workloads, the performance of `default` can vary (see Figure 6).

Considering imbalanced workloads, the performance of `default` highly depends on the average execution time of the work-groups scheduled to the first thread in its first pass. As displayed in Figure 6, the worst case occurs if the work-groups that need the most time to execute are scheduled to the first thread in its first pass. There are certain work distributions that can result in a best case scheduling. They all have in common that the average execution time of the work-groups scheduled to the first thread in its first pass is equal to or less than the average execution time of all work-groups.

If the scheduling overhead is not negligible, performance drawbacks can occur for balanced workloads using `default`, because the number of scheduler passes is not the same for all threads in contrast to `static` (cf. Figure 5).

6.3 Intel TBB

The Intel Threading Building Blocks (TBB) [26] open source library is a high-level alternative to pthreads on CPU architectures, used by the Intel OpenCL implementation internally. With TBB it is possible to use the `parallel_for()` construct to execute a function over a data range that is expressed with a `blocked_range` data structure that can have multiple dimensions (similar to OpenCL's NDRange concept).

The necessary scheduling process consists of two steps: First, a *partitioner* [2] divides the *blocked range* into tasks, which are scheduled with a work stealing strategy in the second step.

The `auto` partitioner (default) "performs sufficient splitting to balance load". The `affinity` partitioner is similar to the auto partitioner "but improves cache affinity by its choice of mapping subranges to worker threads." The `static` partitioner "distributes range iterations among worker threads as uniformly as possible, without a possibility for further load balancing" and maps subranges to worker threads similar to the `affinity` partitioner. The `simple` partitioner "recursively splits a range until it is no longer divisible."

To control the granularity by which a *blocked range* is split, a *grain size* can be specified for all partitioners which works similar to the chunk size concept.

6.4 Summary - Scheduling Strategies

The optimal scheduling strategy depends on the work load. Thus, OpenMP and TBB provide multiple scheduling strategies, which allow for the specification of the chunk size and have a `static` mode that allows deterministic mapping. While Intel OpenCL uses TBB internally, it does not expose any means to influence its scheduling strategies. PoCL has only one scheduling strategy that may perform poorly for certain workloads. It does not allow to define a deterministic mapping from work-group to compute unit. When compared with the OpenMP standard, OpenCL lacks means to influence work-group scheduling, which is especially important on CPU targets. In the next section, we introduce a configurable TBB-based CPU-driver for PoCL.

7 TBB DEVICE DRIVER FOR POCL

In this section, we describe the main contribution of this work to the PoCL project: a device driver that allows the usage of the Intel Threading Building Blocks library for scheduling.

The implementation of the PoCL pthread device does not support multiple scheduling strategies and has no tuning parameters as comparable solutions to application depended scheduling as described in Section 6. PoCL lacks performance compared to the Intel OpenCL SDK, which uses the Intel TBB library. Thus, we decided to write a device driver for PoCL that allows the usage of the TBB library for scheduling.

The TBB device for PoCL is derived from the pthread device with the relevant code sections replaced with calls to the TBB library. This triggered refactoring and code structure changes. Due to a lack of scheduling options in the OpenCL standard, we added the possibility for the application developer to choose the partitioner and grain size to control the TBB behaviour as described in Section 6.3 with the `POCL_TBB_PARTITIONER` and `POCL_TBB_GRAIN_SIZE` environment variables.

Our solution creates a so called *meta thread* on device initialisation. The meta thread polls the queue for commands and kernels to execute, and in the case of kernels executes them in parallel with a blocking call to the TBB library. While the main application threads continues running the host code, TBB uses one worker thread (including the calling thread) per hardware thread to process the partitioned problem. The meta thread has the same function as the worker threads of the pthread device.

By using the TBB library just like Intel OpenCL, we could partly close the performance gap, and provide users with more control over the scheduling which Intel OpenCL does not expose. For the remaining performance differences, we have to assess the quality of generated kernel code, mainly the compiler vectorisation, and also the different vector math libraries (SLEEF/libclc vs. SVML). Since Intel OpenCL is not open source, we cannot examine possible custom TBB scheduling strategies, the vectoriser Intel implemented, its SVML library, or other code that might be relevant for performance.

8 VECTORISATION

Another critical performance aspect is the compiler optimisation performed on the OpenCL kernel code. PoCL uses LLVM/Clang to compile the kernel code, and hence depends on its optimisation capabilities. Vectorisation is especially important here, as it is necessary to make use of the data parallel SIMD instruction set extensions of CPUs, e.g. AVX2/AVX-512 on x86-64.

```
/* automatic vectorisation scheme */
#pragma omp parallel for
for (int group_id = 0;
     group_id < (num / VEC_LENGTH);
     ++group_id)
{
    /* vectorised work-item loop */
    #pragma omp simd
    for (int local_id = 0;
         local_id < VEC_LENGTH;
         ++local_id)
    {
        // scalar typed kernel code
        // double
    }
}

/* manual vectorisation scheme */
#pragma omp parallel for
for (int group_id = 0;
     group_id < (num / VEC_LENGTH);
     ++group_id)
{
    /* manually vectorised kernel */
    // vector typed kernel code
    // double_vec_t / double8
}
```

Figure 7: An OpenCL runtime model written in OpenMP with a task parallel for loop processing work-groups, and a data parallel, vectorised loop processing the work-items within each group. In a manual vectorisation scheme, the vectorisation result of the SIMD loop is replaced by a kernel that uses vector instead of scalar data types.

When looking at an OpenCL kernel and the surrounding OpenCL execution model, different vectorisation strategies are possible. Conceptually, the vectorisation can be applied on either a) a separated instance of a kernel, i.e. a single work-item, or b) across work-items, similar to the execution model known from GPUs where work-items within a work-group are mapped to SIMD lanes and executed in a data-parallel fashion.

If we extend the task parallelism code example from Section 6.1 by adding another, SIMD parallelised loop for processing the work-items within a work-group, we obtain a simple, one-dimensional OpenMP model of the OpenCL runtime: Two nested loops where the first one processes the work-groups and the second one processes the work-items within each work-group, as shown in Figure 7. The latter loop is a natural target for vectorisation across work-items which mirrors the GPU execution model on the CPU's SIMD units. The kernel inside this loop nest can be vectorised by replacing the scalar instruction with vector instructions matching the vector width of the architecture. As the kernel function might contain

further loops, a compiler vectoriser needs to be able to perform an outer-loop vectorisation here.

An alternative to automatic compiler vectorisation is explicitly using OpenCL vector data types, e.g. `double8` or `float16` for AVX-512, and their corresponding operations which can be directly mapped to SIMD vector registers and instructions. Like with automatic compiler vectorisation, built-in functions require a vectorised implementation here. Executing such a kernel via the OpenCL API then requires to divide the NDRange size by the vector width.

The Intel OpenCL CPU Runtime implements the automatic compiler vectorisation described above, and thus provides the programmer with a predictable vectorisation strategy. Predictability is important here, because even though the compiler might vectorise the code, it does not optimise the memory layout to match the vectorisation strategy [18]. A mismatch between these can result in costly, non-contiguous memory accesses via expensive gather/scatter instructions instead.

PoCL and LLVM/Clang do not provide the means to perform this work-item vectorisation scheme, yet. LLVM provides a loop vectoriser which can vectorise suitable loops, and the SLP (Superword-Level Parallelism) vectoriser which can group multiple suitable instructions into a single SIMD instruction. The loop vectoriser does not support the required outer loop vectorisation, thus only inner loops or groups of instructions inside a suitable kernel function can be vectorised. Intel OpenCL is also based on LLVM, but it uses its own proprietary optimiser passes for vectorisation.

However, LLVM contains an experimental *VPlan* vectoriser [3], which aims to provide the long missing outer loop vectorisation for LLVM and to eventually replace the LLVM loop vectoriser. The basic idea of implementing a work-item vectorisation strategy in PoCL would be to annotate the equivalent to the work-item loop as shown in Figure 7 to be vectorised by the compiler. Given the rather complex PoCL code base, we decided to first evaluate this idea using the OpenMP model of the OpenCL runtime, and the OpenMP version of the hexciton benchmark that uses the KART runtime compilation library for the kernel code.

Since the VPlan vectoriser recognises the `#pragam omp simd` directives of OpenMP, the benchmark itself can be used as is. We implemented a wrapper script for the Clang compiler that splits the compilation process into multiple steps, first generating LLVM IR with a disabled loop vectoriser, which is then transformed using the VPlan optimiser pass. This resulting IR is compiled into an object file. Additionally, the runtime configuration files specifying the toolset and command line options used to compile the OpenMP kernels were modified to use this wrapper.

In contrast to LLVM's regular loop vectoriser, the VPlan pass manages to vectorise the outer work-item loops of all kernel versions where the inner loops have a compile-time known loop count. However, the compiler fails to recognise the continuous memory access pattern and the vector registers are loaded and stored with expensive gather and scatter instructions instead. As opposed to the OpenCL version of the benchmark, the OpenMP benchmark allows to manually permute the loop order, effectively transforming the outer loop vectorisation scheme into an inner loop vectorisation by moving the work-item loop into the loop test. This is possible due to the specific loop nest of the benchmark kernel, but not in general. For this transformed version of the kernel, both the VPlan, and the loop vectoriser succeed and generate the expected contiguous vector load/store instructions, providing an estimate how the compiler vectorised code could perform.

At the current state, it looks like LLVM's vectorisation capabilities are not sufficient to result in a work-item vectorisation for PoCL that can compete with Intel OpenCL's vectoriser. The benchmark results are shown and discussed in section 10.2.

9 FURTHER PERFORMANCE IMPROVEMENTS

In this section, we briefly describe further performance improvements we contributed to the PoCL project during this work.

9.1 Parallelisation Threshold off-by-one Issue

As described in Section 3.1, PoCL has a parallelisation threshold. A source code comment suggests that using a number of work-items that equals the parallelisation threshold should result in parallel execution, which it did not. We developed a patch to fix this off-by-one issue. This patch was merged to the master branch ahead of the PoCL 1.5 release.

This patch only improves performance in an edge case when all of the following three requirements are met: i) the number of work-items equals the parallelisation threshold, ii) the work-group size has not been set manually by the application developer, iii) there is enough work to process per work-item (depending on the used hardware). The parameters that we chose for the manual8 vectorisation mode (cf. Table 4) of the *op_benchmark* correspond to this edge case, which results in performance improvements of multiple orders of magnitude in this particular case since the benchmark otherwise runs sequentially.

9.2 Scheduler Integer Division Bug

Investigating the pthread device source code revealed an integer division bug, which could cause an unfavourable assignment of work-groups to threads. Because of this bug, the scheduler switches to a smaller chunk size too late, which can result in insufficient or no work for the last thread. We developed a patch that was merged to the master branch ahead of the PoCL 1.5 release.

Because of the characteristics of the scheduling strategy (see Section 6), the consequences of this bug are partially self-healing. The bug had the highest performance impact in balanced scenarios where the number of work-groups is equal to the number of compute units or is a small multiple of it.

10 EVALUATION

In this section, we discuss the performance results of our contributions described in Section 7, 8, and 9.2.

For most workloads, the performance differences between the PoCL 1.4 (LLVM 9) version with the scheduler's integer-division bug fixed, PoCL 1.5 (LLVM 9) and PoCL 1.6 (LLVM 11) are negligible. Thus, we compare PoCL 1.4 (LLVM 9) to PoCL 1.6 (LLVM 11) in order to quantify the performance impact of the improvement described in Section 9.2, and specifically point out where this direct comparison is not feasible.

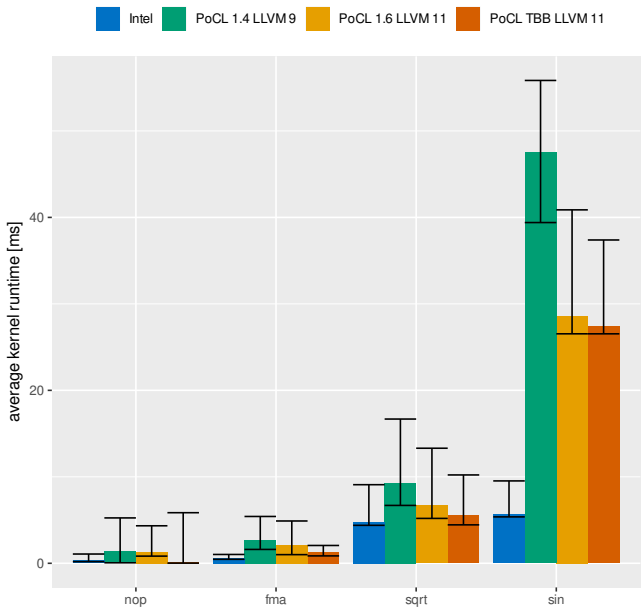


Figure 8: Selected operations from the *op_benchmark* (20480 data elements, 10000 iterations, 45 kernel runs excluding 5 warmups). For each operation and OpenCL implementation, only the fastest individual vectorisation mode is displayed (cf. Table 6). The error bars indicate minimum and maximum values observed.

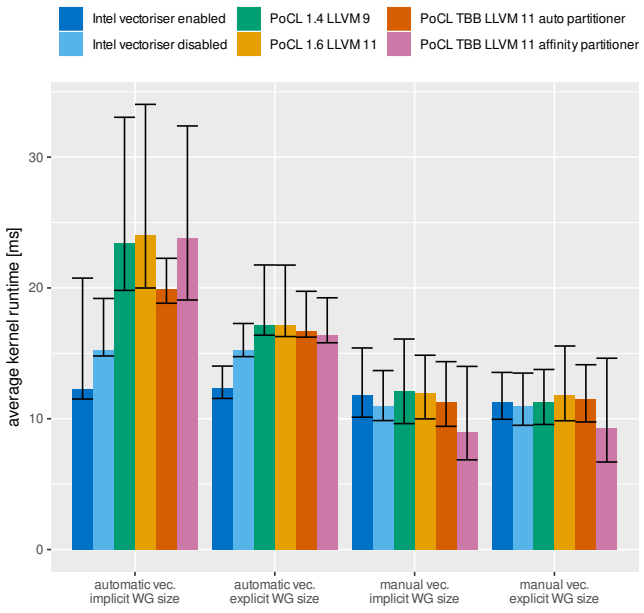


Figure 9: *hexciton_benchmark* vectorisation strategy comparison (25 iterations excluding one warmup iteration). The work-group (WG) size was chosen implicitly by the OpenCL implementation or explicitly by the application. The error bars indicate minimum and maximum values observed.

10.1 Synthetic Micro Benchmark: op

For selected operations of the *op_benchmark*, the outcome of our improvements are displayed in Figure 8. Because there is no single vectorisation strategy that performs best for all OpenCL implementations or all operations, we compare the highest performance value of an operation (or built-in function) across all vectorisation strategies. The results marked with PoCL 1.4 already include the off-by-one fix described in Section 9.1.

These initial measurements with PoCL 1.4 show a slowdown of 4.59× for `fma`, 1.95× for `sqrt` and 8.39× for `sin` compared to Intel. Especially trigonometric functions that are typically implemented in software and not in hardware significantly lack performance. This indicates that the libraries used by PoCL do not exploit hardware capabilities as good as Intel's SVML.

Almost all operations benefit from fixing the integer division bug in PoCL's scheduler (cf. Section 9.2), which can be observed by comparing PoCL 1.4 to PoCL 1.6. Basic arithmetic operations, e.g. `fma`, achieve speedups of up to 1.3×, while for more complex and trigonometric functions speedups of 1.4× to 1.5× are common (`sqrt`: 1.40×) and can reach up to 1.7× (`sin`: 1.66×).

Using the proposed TBB device in PoCL 1.6 with LLVM 11, basic arithmetic operations and simple functions achieve speedups of up to 1.8× (`fma`: 1.58×, `sqrt`: 1.20×) compared to the pthread device. Trigonometric functions do not change performance.

The total speedup for both, the fixed integer-division bug, and the TBB device combined is 2.06× for `fma`, 1.68× for `sqrt` and 1.74×, reducing the slowdown to Intel to 2.23× for `fma`, 1.16× for `sqrt` and 4.84× for `sin`.

10.2 Proxy Application Benchmark: hexciton

In addition to the synthetic *op_benchmark*, we use the *hexciton_benchmark* application proxy to evaluate the performance of a balanced real world applications. We compare two vectorisation strategies: automatic and manual, as well as two work-group (WG) sizes: the WG size chosen implicitly by the OpenCL implementation and a WG size specified explicitly by the application.

Figure 9 compares Intel OpenCL, with and without compiler vectorisation enabled, with the PoCL 1.4 baseline, the current PoCL 1.6 with the pthreads device, as well as our TBB device with the automatic and affinity partitioner. For kernels that depend on automatic vectorisation by the compiler, PoCL shows a much lower performance, as they do not get vectorised by LLVM's loop vectoriser. The gap of PoCL 1.6 to Intel is 0.51× with the implicit WG size and 0.72× with the explicitly specified WG size. The scheduler bug described in Section 9.2 as well as a newer LLVM version do not improve performance here. Explicitly specifying a WG size increases performance by a factor of 1.40× for PoCL 1.6. This indicates that the algorithm PoCL employs for determining the WG size, if none is specified, is sub-optimal here.

Disabling compiler vectorisation for Intel OpenCL for the kernel versions that depend on automatic vectorisation, makes a difference of 0.80× and 0.81×. Even with disabled vectorisation, Intel outperforms PoCL. For the implicit WG sizes, the gap is still 0.77×, the smaller work-groups size of 256 (25.6 WGs/CU) determined by Intel OpenCL vs. the 4096 (1.6 WGs/CU) from PoCL which leaves no room for a good schedule on the 80 hardware threads of the benchmark system. Explicitly specifying the WG size narrows the gap to 0.93×, and 0.75× with Intel's vectoriser enabled.

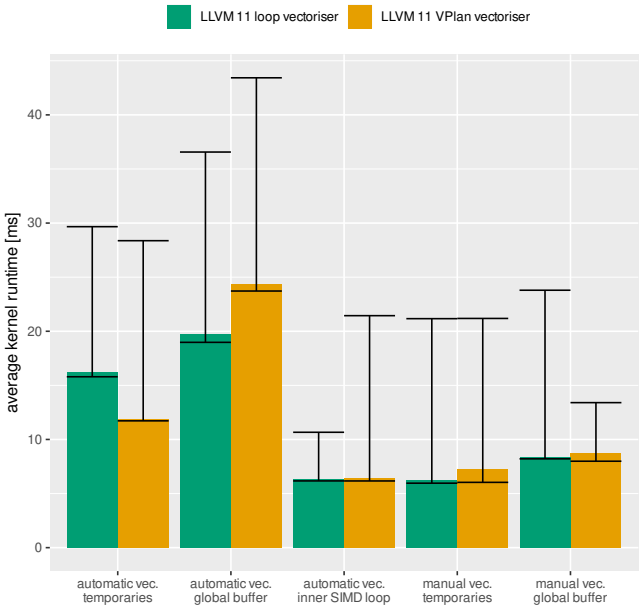


Figure 10: Results of the LLVM 11 vectoriser study using the OpenMP version of the *hexciton_benchamrk* as described in Section 8. The term *temporaries* means that computations were accumulated in a temporary local variable, while *global buffer* means the kernel works directly on the input buffer, potentially causing more memory accesses. For the *inner SIMD loop* version, the SIMD loop was manually permuted to become the innermost loop. The error bars indicate minimum and maximum values observed.

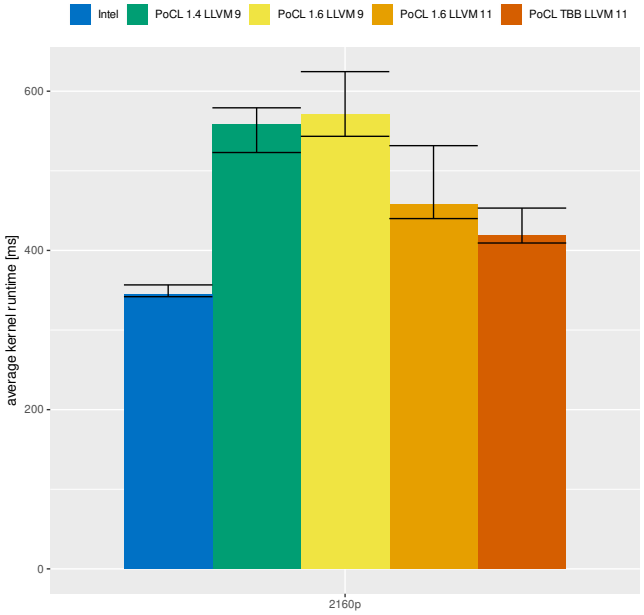


Figure 11: Average kernel execution time of the *raytracing_benchmark* with 50 iterations (width×height: 3840×2160 pixels). The error bars indicate minimum and maximum values observed.

For the manually vectorised kernels, the performance of PoCL matches the performance of the Intel OpenCL implementation, as the SIMD-hardware usage no longer depends on the compiler vectorisation. When using our proposed TBB device, the performance of the manually vectorised kernel with the implicit work-group size increases by a factor of 1.06×. When specifying the affinity partitioner, performance can be improved by a factor of 1.26× compared to the default auto partitioner of TBB. Using manual vectorisation and the TBB device with the affinity partitioner allows to outperform Intel by up to 1.31×, for the hexciton benchmark.

Figure 10 shows the result of the LLVM vector comparison, as described in Section 8. These numbers are based on the OpenMP version of the *hexciton_benchmark* using the OpenCL runtime model as described in Figure 7, and the KART library for runtime compilation. The first two kernels for automatic vectorisation show that depending on details in the code (here: the way the output data is accumulated and written) it can make a large difference on whether the loop vectoriser or the VPlan vectoriser generates the faster code. In both cases, the loop vectoriser does not actually vectorise the code, while the VPlan vectoriser performs the intended vectorisation. However, its failure to recognise the contiguous memory access pattern, through the kernel's AoSoA-layout index computations, results in expensive gather/scatter instructions. Working on the global buffer instead of writing temporary results into a local variable generates so many scatter instructions, that the non-vectorised version outperforms it by 1.23×. With temporaries, the VPlan vectoriser outperforms the loop vectoriser by 1.37×.

The manual permutation of the outer SIMD loop into the loop nest, which this kernel code allows, enables both compilers to generate efficient SIMD code, i.e. what we would like to see from a successful outer loop vectorisation as well. As expected, the manually vectorised kernels, which do not depend on the compiler vectorisation, show the same performance, with a slight slow-down for the global buffer access.

In conclusion, we can say that implementing work-item vectorisation in PoCL via the VPlan vectoriser would generate vectorised code which, at least in non-trivial cases like this benchmark, is not on par with both the Intel's OpenCL compiler, or a manual vectorisation scheme.

10.3 Imbalanced Workload with Heavy Use of Built-in Functions: raytracing

The evaluation results for the *raytracing_benchmark* are displayed in Figure 11. Initial measurements show that PoCL 1.4 is 1.62× slower than Intel. Fixing the scheduler integer division bug as described in Section 9.2 increases execution time by 2 %. Switching from LLVM 9 to LLVM 11, both with PoCL 1.6, achieves a speedup of 1.25×. Compared to PoCL 1.6 with LLVM 11, our proposed TBB device achieves a speedup of 1.09×. Varying partitioner and grain size does not achieve a higher speedup. Using the TBB device, the slowdown could be reduced from 1.32× to 1.21× compared to Intel.

11 SUMMARY

In this paper, we analysed the performance differences between the closed-source Intel OpenCL implementation and the open-source Portable Computing Language (PoCL) on Intel CPUs. Our analysis shows that PoCL is lacking performance compared to Intel OpenCL, and indicates that this gap is caused by deficits in work-group scheduling and code generation, especially compiler vectorisation. Following a deeper analysis of PoCL's scheduling strategy and a comparison to other frameworks, we propose an additional driver for PoCL using the Threading Building Blocks (TBB) library. We evaluated LLVM's compiler vectorisers with respect to work-item vectorisation of OpenCL kernels in PoCL. Additionally, we fixed a minor issue that improved edge case performance and a bug that decreased performance in balanced scenarios.

The TBB device enables speedups of up to 1.3× for the proxy application benchmark and up to 1.8× for the synthetic micro benchmark. For the proxy application benchmark, this also means a speedup of 1.3× over Intel OpenCL. With our contribution, PoCL's users now have the flexibility to choose between the default scheduling algorithm and the options of the TBB library to fine tune the scheduling behaviour to their needs. Providing OpenCL users with more control over scheduling, especially on CPUs, might be worth considering for integration into the OpenCL standard.

We found that LLVM currently lacks the necessary vectorisation capabilities to implement efficient work-item vectorisation in PoCL, but the VPlan vectoriser puts this goal in reach. For now, manual vectorisation using vector data types as described in [18] seems to be the best way to ensure a predictable vectorisation across work-item, as well as a performance which is on par with Intel OpenCL. For kernels making heavy use of built-in functions, using Intel SVML or GNU libmvec might be an alternative to the currently used SLEEF/libclc libraries.

Overall, PoCL with the TBB device can compete with Intel OpenCL in cases where automatic compiler vectorisation and built-in functions are not crucial for kernel performance. For the other cases, PoCL can build on LLVMs future improvements.

ACKNOWLEDGMENTS

We would like to thank the PoCL community, especially Pekka Jääskeläinen, Mauri Mustonen, and Michal Babej for helpful discussion on GitHub. This work was partially supported through funding from the German Federal Ministry of Education and Research (BMBF) within the Forschungscampus MODAL, project number 05M20ZBM.

REFERENCES

[1] [n. d.]. libclc website. https://libclc.llvm.org/ Accessed: 2021-03-18.
[2] [n. d.]. TBB Partitioner Summary. https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Partitioner_Summary.html Accessed: 2021-03-18.
[3] [n. d.]. Vectorization Plan. https://llvm.org/docs/Proposals/VectorizationPlan.html Accessed: 2021-01-13.
[4] 2020. TOP500 November 2020. https://top500.org/lists/top500/2020/11/ Accessed: 2021-03-18.
[5] Cristiana Bolchini, Stefano Cherubin, Gianluca C. Durelli, Simone Libutti, Antonio Miele, and Marco D. Santambrogio. 2018. A Runtime Controller for OpenCL Applications on Heterogeneous System Architectures. *SIGBED Rev.* 15, 1 (March 2018), 29–35. https://doi.org/10.1145/3199610.3199614
[6] Michael Haidl, Simon Moll, Lars Klein, Huihui Sun, Sebastian Hack, and Sergei Gorlatch. 2017. Pacxxv2+ RV: an LLVM-based portable high-performance programming model. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. 1–12.
[7] Graham Holland. 2019. Abstracting OpenCL for Multi-Application Workloads on CPU-FPGA Clusters.
[8] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming* 43, 5 (2015), 752–785. https://doi.org/10.1007/s10766-014-0320-y
[9] Pekka Jääskeläinen, Ville Korhonen, Matias Koskela, Jarmo Takala, Karen Egiazarian, Aram Danielyan, Cristóvão Cruz, James Price, and Simon McIntosh-Smith. 2019. Exploiting Task Parallelism with OpenCL: A Case Study. *Journal of Signal Processing Systems* 91, 1 (2019), 33–46.
[10] Ralf Karrenberg and Sebastian Hack. 2012. Improving Performance of OpenCL on CPUs. In *Compiler Construction*. http://www.cdl.uni-saarland.de/papers/karrenberg_opencl.pdf
[11] Khronos OpenCL Working Group. 2020. *The OpenCL Specification*. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf
[12] Joo Hwan Lee, Nimit Nigania, Hyesoon Kim, Kaushik Patel, and Hyojong Kim. 2015. OpenCL performance evaluation on modern multicore CPUs. *Scientific Programming* 2015 (2015).
[13] Paul Lindberg. 2009. Performance Obstacles for Threading: How do they affect OpenMP code? (January 2009). https://web.archive.org/web/20131126073803/https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code Accessed: 2021-03-18.
[14] Matthias Noack. 2015-2021. hexciton_benchmark source code on Github. https://github.com/noma/hexciton_benchmark
[15] Matthias Noack. 2016-2021. op_benchmark source code on Github. https://github.com/noma/op_benchmark
[16] Matthias Noack. 2017-2019. OpenCL Helper library on Github. https://github.com/noma/ocl
[17] Matthias Noack, Alexander Reinefeld, Tobias Kramer, and Thomas Steinke. 2018. DM-HEOM: A Portable and Scalable Solver-Framework for the Hierarchical Equations of Motion. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 947–956. https://doi.org/10.1109/IPDPSW.2018.00149
[18] Matthias Noack, Florian Wende, and Klaus-Dieter Oertel. 2015. OpenCL: There and Back Again. In *High Performance Parallelism Pearls*, James Reinders and Jim Jeffers (Eds.). Vol. 2. Morgan Kaufmann, Boston, 355–378. https://doi.org/10.1016/B978-0-12-803819-2.00001-X
[19] Matthias Noack, Florian Wende, Georg Zitzlsberger, Michael Klemm, and Thomas Steinke. 2017. KART – A Runtime Compilation Library for Improving HPC Application Performance. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf (Eds.). Springer International Publishing, Cham, 389–403.
[20] OpenMP Architecture Review Board. 2018. *OpenMP API Specification*. https://www.openmp.org/spec-html/5.0/openmp.html
[21] Omair Rafique and Klaus Schneider. 2020. Employing OpenCL as a Standard Hardware Abstraction in a Distributed Embedded System: A Case Study. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. 1–7. https://doi.org/10.1109/MECO49872.2020.9134270
[22] Naoki Shibata and Francesco Petrogalli. 2020. SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2020), 1316–1327. https://doi.org/10.1109/TPDS.2019.2960333
[23] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. 2017. Adaptive Optimization for OpenCL Programs on Embedded Heterogeneous Systems. *SIGPLAN Not.* 52, 5 (2017), 11–20. https://doi.org/10.1145/3140582.3081040
[24] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2019. Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2019)*. Association for Computing Machinery, New York, NY, USA, Article Article 1, 6 pages. https://doi.org/10.1145/3337801.3337819
[25] Alexander Veselov. 2016-2021. raytracing_benchmark source code on Github. https://github.com/new2f7/RayTracing
[26] Michael Voss, Rafael Asenjo, and James Reinders. 2019. *Pro TBB : C++ Parallel Programming with Threading Building Blocks*. Springer Nature. https://doi.org/10.1007/978-1-4842-4398-5
[27] Peng Zhang, Jianbin Fang, Canqun Yang, Tao Tang, Chun Huang, and Zheng Wang. 2018. MOCL: An Efficient OpenCL Implementation for the Matrix-2000 Architecture. In *Proceedings of the 15th ACM International Conference on Computing Frontiers (CF '18)*. Association for Computing Machinery, New York, NY, USA, 26–35. https://doi.org/10.1145/3203217.3203244