Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

OpenCL-HPX Integration

Michael Schupikov

Course of Study: Informatik

Examiner: Prof. Dr. Dirk Pflüger

Supervisor: Gregor Daiss, M.Sc.

Commenced: December 18, 2020

Completed: June 18, 2021

Abstract

Distributed applications combine the computational capabilities of heterogeneous nodes. As such, they offer challenges regarding data transfer and synchronization. *HPX* is a library for concurrent, parallel applications. It strives not only to address challenges regarding distributed systems, but also to conform to current and upcoming C++ standards. One of the solutions found in heterogeneous systems is provided in form of the *OpenCL* standard. It enables the cooperation between hardware resources through a unified interface. In this work, we combine *HPX* and *OpenCL* in form of an executor. The *OpenCL* executor enables *HPX* users to benefit from more resources on heterogeneous nodes. We describe the executor's design and its implementation. Furthermore, we present the testing methods to ensure the correctness of the executor. Finally, we provide benchmarks on NVIDIA and AMD GPUs.

Contents

1 Introduction **15**
1.1 HPX . . . 15
1.2 OpenCL . . . 16
1.3 Executor . . . 17
1.4 Goals . . . 17
1.5 Related Work . . . 17
1.6 Outline . . . 18
1.7 Terminology . . . 18

2 Setup **19**
2.1 Environment . . . 19
2.2 Structure . . . 20

3 Implementation **21**
3.1 Error Checking . . . 21
3.2 Event Pool . . . 22
3.3 Future Data . . . 23
3.4 Futures . . . 24
3.5 Operation Modes . . . 27
3.6 Executor . . . 34

4 Testing **41**
4.1 Usage . . . 41
4.2 Test Setup . . . 41
4.3 Test Cases . . . 42
4.4 Results . . . 46
4.5 Analysis . . . 47

5 Benchmarking **49**
5.1 Matrix Multiplication . . . 49
5.2 Streaming . . . 53
5.3 Block Size . . . 56

6 Conclusion **65**
6.1 Reproducibility . . . 65
6.2 Outlook . . . 65

A Benchmarking Results **67**
A.1 Matrix Multiplication . . . 67
A.2 Data Stream . . . 67

A.3 Blocks 68

B Design Decisions **71**
B.1 Protected Destructor in Base Classes 71
B.2 Avoiding Singletons 71
B.3 Using Exceptions for Error Propagation 72
B.4 Ordering of Fields 72
B.5 Visibility of Declarations 73

C Machine Characteristics **75**
C.1 GeForce RTX 3080 GPU 75
C.2 AMD EPYC 7551P GPU 75
C.3 GeForce GTX 1080 Ti GPU 75

Bibliography **79**

List of Figures

1.1 Basic visualization of distributed functionality supported by *HPX* 15
1.2 High level architecture of *HPX* 16
1.3 *OpenCL* portability. 16

2.1 Source code structure of the *OpenCL* executor. 20

3.1 Interface of `future_data`. 23
3.2 Mechanism of the callback mode. 28
3.3 Mechanism of the polling mode. 30
3.4 Interface of the *OpenCL* executor. 34

5.1 Runtime of matrix multiplication depending on block count for callback mode. . 52
5.2 Runtime of matrix multiplication depending on block count for polling mode. . . 53
5.3 Runtime of matrix multiplication between 8 and 128 blocks using callback mode. 54
5.4 Runtime of matrix multiplication between 8 and 128 blocks using polling mode. . 55
5.5 Runtime of matrix multiplication for all modes and small matrices. 56
5.6 Runtime of allocating and populating input buffers on device. 58
5.7 Runtime of retrieving result from device. 59
5.8 Runtime of allocating and populating input buffers on device focusing between 8 and 128 blocks. 60
5.9 Runtime of retrieving buffers from device focusing between 8 and 128 blocks. . . 61
5.10 Allocation and population of buffers on the device (input) and transferring the result from the device (output) for small data volumes. 62
5.11 Runtime depending on work group size (block size) for callback mode. 63
5.12 Runtime depending on work group size (block size) for polling mode. 64

List of Tables

1.1 Terminology equivalents for *CUDA* and *OpenCL* 18

2.1 Versions of used development tools, libraries and the Linux distribution. 19

5.1 Relation of block count, matrix dimensions and number of participating `float` elements. 52

A.1 Matrix multiplication runtime for increasing block sizes (callback mode).. 67
A.2 Matrix multiplication runtime for increasing block sizes (callback mode).. 68
A.3 Runtime for allocating and populating buffers on the device. The data volume is measured in block counts. 68
A.4 Runtime for retrieving data from the device. The data volume is measured in block counts. 69
A.5 Runtime of matrix multiplication using various block sizes (callback mode). . . . 69
A.6 Runtime of matrix multiplication using various block sizes (polling mode). . . . 69

C.1 *OpenCL* device used on machine `pcsgs05`. 76
C.2 *OpenCL* device used on machine `argon-epyc`. 77
C.3 Device used on machine `argon-gtx`. 78

List of Listings

3.1 Core error checking functionality for *OpenCL* functions. 22
3.2 Constructor of `future_data`, overloaded for polling mode. 24
3.3 Constructor of `future_data`, overloaded for callback mode. 24
3.4 Definition of operation modes for the executor. 25
3.5 Explicit template specifications for `create_future()`. 26
3.6 Construction of `future_data` and corresponding `hpx::future`. 26
3.7 Construction of a `future_data` from an *OpenCL* event. 27
3.8 Setting the callback on an event. 28
3.9 Callback function setting the computation's status in `future_data`. 29
3.10 Set and clear *OpenCL* polling function in *HPX* scheduler. 31
3.11 Calling the polling function in the *HPX* scheduler. 31
3.12 Registration of the polling function and its removal on destruction. 32
3.13 *OpenCL* executor's polling function. 33
3.14 Initialization of the *OpenCL* executor. 35
3.15 Creating *OpenCL* platform by index. 36
3.16 Creating *OpenCL* context on device. 37
3.17 One-way execution of provided *OpenCL* function. Error checking is omitted for brevity. 38
3.18 Two-way execution of provided *OpenCL* function. A corresponding `hpx::future` is returned. Error checking is omitted for brevity. 38
3.19 Enabling one-way and two-way execution for the *OpenCL* executor. 39
3.20 Short, memorable names for the *OpenCL* executor providing different operation modes. 39
4.1 Aligning return value with test results. 42
4.2 Test results of all unit tests. 47
5.1 *OpenCL* kernel used for matrix multiplication benchmark. 51
B.1 Destructor of type `B` is not invoked, because `A:: A()` is not virtual. 72

List of Algorithms

5.1 Runtime benchmark of matrix multiplication. 50
5.2 Runtime benchmark of data input-output. 57

1 Introduction

1.1 HPX

HPX is an acronym for High Performance ParalleX.

HPX is a library for concurrency and parallelism [KDL+20]. It is written in C++ and is developed by the *STE||AR Group*. The goal of *HPX* consists of two major aspects. Firstly, it provides capabilities in alignment with the C++ standard for local usage. Secondly, it facilitates the development of scalable, distributed and concurrent systems. Figure 1.1 visualizes the idea. Using *HPX* , the application is not limited by the resources on the local machine. Instead, *HPX* enables its user to distribute computationally expensive tasks to distant nodes. Therefore, it provides scalability and enables the use of resources.

HPX is dominantly used in scientific computing. Additionally, it is used in gaming, finances, data mining and cyber security and other fields. Furthermore, *HPX* is the foundation of libraries for shared programming, performance portability programming and distributed array processing kits.

The high level architecture of *HPX* is shown in in Figure 1.2. We highlight some parts of its design. The users of the library benefit from the C++ API. The interface is designed based on the C++ standard. Therefore, most developers are already familiar with some parts of the API. Additionally, *HPX* provides extensions, which currently are not part of the C++ standard. One example is `std::tag_invoke()`, which is proposed to the language's standard. However, `std::tag_invoke()` has not been included in the standard, yet. Another example are features of later standards like `std::optional` and `std::filesystem`, which are part of C++17.

Therefore, using *HPX* with C++14 provides access to library additions of C++17 and C++20. Furthermore, *HPX* provides access to upcoming library features, which are proposed for the upcoming C++23 standard.

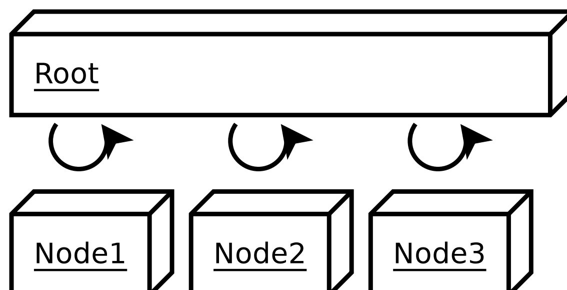

Figure 1.1: Basic visualization of distributed functionality supported by *HPX* .

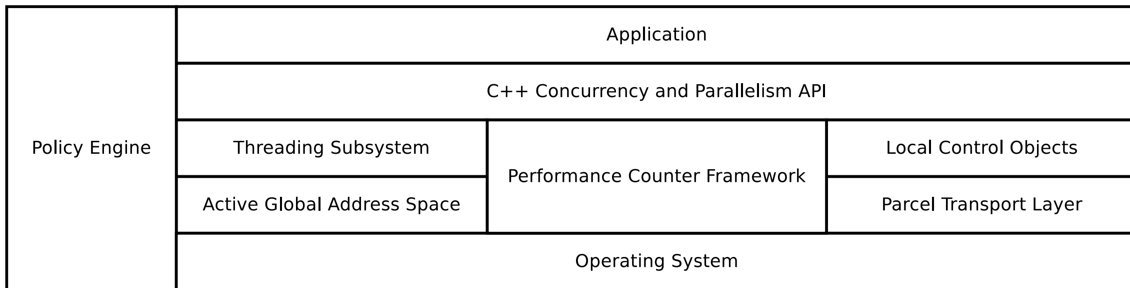

Figure 1.2: High level architecture of *HPX* .

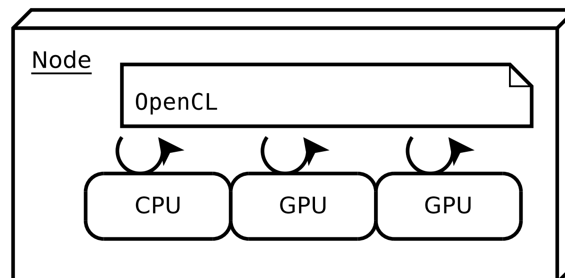

Figure 1.3: *OpenCL* portability.

Moreover, it *HPX* provides extensions to the interface of the standard library. Firstly, it provides enhanced capabilities for threading and scheduling through the threading subsystem and policy engine, respectively. Additionally, it provides support for communication between instances on distant nodes via the parcel transport layer. The implementation of the *OpenCL* executor relates to local control objects.

1.2 OpenCL

OpenCL is a standard for cross-platform, parallel computing. It enables the usage of all supported resources on the machine through a unified interface. It especially does not rely on the hardware provided by any particular vendor.

The main goal of the *OpenCL* standard is visualized in Figure 1.3. We consider a machine featuring a powerful CPU and two GPUs from different vendors. The usage of *CUDA* provides access to GPUs from the supported vendor only. Therefore, *CUDA*'s user would have two options. One option consists of adjusting the application to another interface, which provides desired functionality. The second option is to disregard all of the additional resources. Contrary to *OpenCL* , the developer has access to all supported resources through a unified interface.

OpenCL is used for professional creative tools, scientific software, medical software, vision processing, neural network training and interfacing.

1.3 Executor

An *executor* in *HPX* is an object that invokes a function [Dan15]. By using executors as abstraction to a direct function call, the developer is relieved from internal details. In particular, executors in *HPX* allow to choose the mode which defines how the state of execution is verified. Additionally, the user is relieved from retrieving the state of the execution manually. Finally, further improvements are achieved using executors as abstractions [KTK+17].

The *OpenCL* executor supports two modes of execution. Firstly, the executor supports a fire-and-forget approach. Using this mode, the provided function is called. However, no future is returned. This mode is intended for blocking operations Alternatively, this mode covers the use case which does not require the function execution's processing state in general.

Secondly, a future can be retrieved from the function's invocation.

1.4 Goals

In this work, we implement an *OpenCL* executor for the *HPX* project. Using *HPX* features, we intend it to allow direct usage of the *OpenCL* interface. *HPX* futures are to be supported. The verification of invoked *OpenCL* functions' state is to be determined using the callback and polling mechanism.

Solutions for loose coupling between the executor and other parts of the implementation are to be maximized. Additionally, the overhead caused by the executor's provided abstraction is also to be minimized. Finally, the provided interface is to be as direct and lightweight as possible.

1.5 Related Work

One alternative to *HPX* consists of the *MPI* standard [GGL+99]. *MPI* is an acronym for Message Passing Interface. One implementation of the *MPI* standard is *OpenMPI* [GFB+04]. However, *HPX* still provides better performance compared to *OpenMPI* by a factor of approximately 1.2 to 128 nodes [BKK+19]. Additionally, many *MPI* implementations provide solutions for specific research problems disregarding compatibility to other implementations [GFB+04].

Another alternative is *Cpp-Taskflow* [HLGW19]. *Cpp-Taskflow* also provides better performance to *OpenMPI* , but not to the extent of *HPX* . Additionally, *Cpp-Taskflow* focuses on minimizing the code complexity over supporting an interface compatible to the C++ standard.

Furthermore, the *OpenCL* executor itself is derived from the implementation of the already existing *CUDA* executor in *HPX* . We design the *OpenCL* executor to maintain the same API and characteristics users of the *CUDA* executor might expect.

CUDA	*OpenCL*
streaming multiprocessor	compute unit
thread	work item
thread block	work group

Table 1.1: Terminology equivalents for *CUDA* and *OpenCL* .

1.6 Outline

The structure of this work is as following.

In Chapter 2 on the facing page, we describe the environment used for development. This chapter provides an overview over the directory structure of the *HPX* project. Additionally, we provide the information regarding the libraries we develop against.

In Chapter 3 on page 21, we provide the detailed description regarding the executor's implementation. Additionally, we provide local design decisions in this chapter.

In Chapter 4 on page 41, we prove the correctness of our implementation. We describe performed tests and their results.

In Chapter 5 on page 49, we provide information regarding runtime characteristics of the *OpenCL* executor. We especially focus different execution modes of the executor on various hardware.

In Chapter 6 on page 65, we provide a summary of our results. Furthermore, we provide an outlook regarding future work.

Due to extensive tests, we provide more detailed runtime information in Appendix A on page 67.

Design decisions that affect the multiple parts of the implementation are elaborated in Appendix B on page 71. In particular, we provide context to our decisions and the reasons for taken solutions.

Due to the variety of used hardware, we provide detailed information for every used machine in Appendix C on page 75. This especially facilitates runtime predictions on hardware not participating in our tests.

1.7 Terminology

At the date of writing, *CUDA* is primary used for GPU's utilization. Because the *OpenCL* terminology for functionally equivalent entities differs from the one used for *CUDA*, we provide a short table of equivalent terminology in Table 1.1.

2 Setup

In this chapter, we describe the environment characteristics used for development. Additionally, we provide the necessary steps in order to add a new module to *HPX* .

The goal of this section consists of two parts. Firstly, we want to provide the necessary conditions to reproduce our results. Secondly, we provide a guidance for further extensions to *HPX* .

2.1 Environment

We use `gcc` as compiler and `cmake` as building system generator. As building system itself, we use `make`. Furthermore, we use the *OpenCL* libraries supported by the *CUDA* runtime or those explicitly installed on machines featuring an *AMD* GPU. Table 2.1 shows the corresponding versions.

Although *OpenCL* version 3.0 is available on some machines, we target version 1.2 as the greatest common divisor. This is especially justified by support for version 1.2 only on machine `pcsgs05`, featuring a *CUDA* device.

Note that `cmake` does not find the required *OpenCL* runtime on all machines using the default configuration. Therefore, we provide the location of corresponding header files via option `OpenCL_INCLUDE_DIR`. Additionally, we need to provide the location of the shared library in that case. We set `OpenCL_LIBRARY` via the tool `ccmake`. Alternatively, `cmake` accepts the modification as command line option.

We select commit

```
84cf823669 (Merge pull request #5279 from msimberg/more-gcc-10-deprecation-warnings)
```

in the *HPX* `git` repository for the final stage of the development.

`gcc`	10.2.0
`make`	4.2.1
`cmake`	3.18.2
OpenCL	1.2
CUDA	11.2.2
HPX	1.6.0-rc2
Linux	5.8.0-55-generic

Table 2.1: Versions of used development tools, libraries and the Linux distribution.

```
compute_opencl
├── CMakeLists.txt
├── docs
│   └── index.rst
├── examples
│   └── CMakeLists.txt
├── include
│   └── hpx
│       └── compute_opencl
├── README.rst
├── src
│   ├── opencl_executor.cpp
│   └── opencl_future.cpp
└── tests
    ├── CMakeLists.txt
    ├── performance
    │   └── CMakeLists.txt
    ├── regressions
    │   └── CMakeLists.txt
    └── unit
        ├── add.cl
        ├── CMakeLists.txt
        ├── opencl.cpp
        ├── opencl_executor.cpp
        ├── opencl_test_cases.hpp
        └── opencl_test_kernel.hpp
```

Figure 2.1: Source code structure of the *OpenCL* executor.

2.2 Structure

We perform the development of the *OpenCL* executor in `libs/full/compute_opencl`. All *HPX* modules follow a unified directory structure. We use the script `libs/create_module_skeleton.py` to create a new module.

The final structure is shown in Figure 2.1. We mention the most important directories.

`include/` contains the header files of the executor. See Appendix B.5 on page 73 for more detailed elaboration.

`src/` contains the source files of the executor.

`tests/performance/` contains the performance tests.

`tests/unit/` contains unit tests to verify the executor's correct implementation.

3 Implementation

In this chapter, we describe the executor's implementation. We use a bottom-up approach in order to cover potential dependencies before their usage. Because we intend to provide guidance for future *HPX* developers, we focus on some details and our design decisions.

Firstly, we describe our error checking mechanism. Secondly, we justify the absence of any event caching mechanism in the *OpenCL* executor. Then, we describe the design of the shared state. The shared state is used for futures to determine whether the associated computation has completed. Therefore, we describe the creation of futures as next step. The executor supports two operation modes. Therefore, we first describe both modes and finally the executor itself. As end result, the executor provides the interface intended to use outside of the module.

We implement most parts of the implementation in the namespace `hpx::opencl::experimental`. Where possible, we use unnamed namespaces in order to hide the executor's internals. We elaborate the usage of unnamed namespaces in Appendix B.5 on page 73.

3.1 Error Checking

HPX heavily relies on exceptions as error propagation mechanism. No alternatives such as the proposed `std::expected` to the C++ standard are used. Therefore, we align the implementation of the *OpenCL* executor accordingly.

Utility functionality for error checking is implemented in `opencl_error.hpp`. It provides the exception `opencl_exception`, which derives from `hpx::exception` and provides the error code returned from an *OpenCL* function.

Function `check_opencl_error()` provides the core mechanism of successful invocation of an *OpenCL* function. Listing 3.1 shows its implementation. It inspects the returned error code from an *OpenCL* function and throws `opencl_exception` on failure.

For facilitated checking, the function `opencl_checked()` expects a message and an *OpenCL* function to be called. The provided *OpenCL* function is expected to provide its result as return value. The returned error code is then checked with `check_opencl_error()`.

The benefit of dedicated error checking functions consists of consistent error checks across the executor's implementation.

Listing 3.1 Core error checking functionality for *OpenCL* functions.

```
inline auto check_opencl_error(char const * const message,
                               cl_int const errcode)
    -> void
{
    if (CL_SUCCESS != errcode)
    {
        throw opencl_exception{message, errcode};
    }
}
```

3.2 Event Pool

One challenge in using *CUDA* is its poor performance for creating of events. Therefore, *HPX* uses a dedicated event pool for its *CUDA* executor. The pool allocates a predefined number of events in advance using `cudaEventCreateWithFlags()`. The usage of an event pool shifts the performance penalty from the executor's execution to the first access to the event pool in `hpx::cuda::experimental::cuda_event_pool::get_event_pool()`. This is ensured by its static instance in the function.

Instead of creating events individually in *CUDA* executor's polling mode, the executor retrieves events from the static event pool. After usage, it returns the events to the event pool. Should the processing require more events than initially created, the event pool allocates one more event, pushes it on the lockless stack and retrieves it again for the user.

CUDA's events do not depend on the particular target device. Therefore, only one lockless stack is maintained as underlying data structure. The lockless stack avoids expensive locking mechanisms for concurrent access, while still providing necessary guarantees.

Contrary to *CUDA*, the *OpenCL* backend maintains events internally. Especially, it does not assume provided event pointers to be initialized prior to their usage. While *OpenCL* still provides `clCreateUserEvent()`, its purpose consists of waiting on an external event before the execution of a command in the command queue. It is not intended to allocate events prior to their usage.

We verify that no performance impact exists if uninitialized events are provided to the enqueued command on the command queue. Therefore, an event pool would not provide performance benefits. As consequence, we do not implement an event pool for the *OpenCL* executor.

Moreover, `clCreateUserEvent()` depends on the device's context in contrast to *CUDA*'s `cudaEventCreateWithFlags()`. Therefore, an implementation of the event pool would require one stack for each context. Together with the absence of a lockless map in *HPX* , this would either require more expensive locking or custom implementation of an lockless map. A possible implementation is provided in [AM04]. In both cases, the overhead is not justified given the design of the *OpenCL* backend and the intended usage of *OpenCL* events.

In conclusion, we decide to omit any event caching mechanism in the *OpenCL* executor.

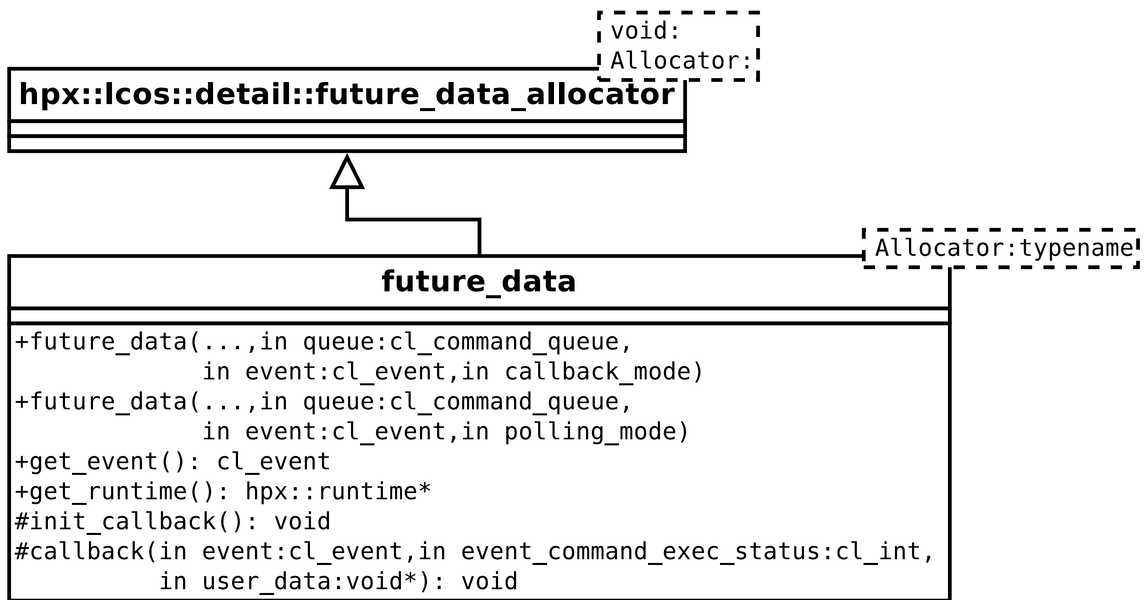

Figure 3.1: Interface of `future_data`.

3.3 Future Data

`future_data` is the pendant to `std::promise` in the C++ standard. It represents the shared state holding the asynchronously calculated result until its retrieval through the corresponding `hpx::future`. Because `future_data` is only used in the internal parts of the executor, we define the structure in an unnamed namespace. It is located in `opencl_future.cpp`. This way we prevent the user from accessing the executor's internals and therefore facilitate its correct usage.

The internally available interface of `future_data` is shown in Figure 3.1. It is derived from `hpx::lcos::detail::future_data_allocator`. The namespace `lcos` contains local control objects, which we mentioned in Chapter 1 on page 15.

The structure is initialized using one of its constructors. Because constructors cannot be specialized in a template using C++, we overload them by the executor's operation type. Both constructors take the allocators, which are required by the base type `hpx::lcos::detail::future_data_allocator`. Additionally, they take *OpenCL* 's command queue and the created event associated with the execution state.

- For the polling mode, no further initialisation is performed as shown in Listing 3.2. The `future_data` instance is simply added to the polling containers and the contained event is periodically checked for completion.
- Listing 3.3 shows the constructor for the callback mode. It registers the callback function `callback()` for the *OpenCL* event. This is performed in `init_callback()`, which utilizes `clSetEventCallback()` and provides the constructed `future_data` as user defined data to the callback function.

Listing 3.2 Constructor of `future_data`, overloaded for polling mode.

```
future_data(init_no_addref no_addref,
            other_allocator const& other_alloc,
            cl_command_queue const& queue,
            cl_event const event,
            cl_context const context,
            polling_mode const)
    : hpx::lcos::detail::future_data_allocator<void, Allocator>(no_addref, other_alloc)
    , event_{event}
    , context_{context}
    , runtime_{hpx::get_runtime_ptr()}
{
}
```

Listing 3.3 Constructor of `future_data`, overloaded for callback mode.

```
future_data(init_no_addref no_addref,
            other_allocator const& other_alloc,
            cl_command_queue const& queue,
            cl_event const event,
            cl_context const context,
            callback_mode const)
    : future_data{no_addref, other_alloc, queue, event, context, polling_mode{}}
{
    init_callback();
}
```

> The constructor for the callback mode delegates to the constructor for the polling mode. Because no mode-specific initialization is performed for the polling mode, this ensures the initialization of `future_data`'s fields without repetition in the callback constructor. Therefore, repetition of implementation is avoided.

While the referenced event is passed to `callback()` explicitly, `future_data` needs to provide it for the polling function via `get_event()`. We annotate the function with `HPX_NODISCARD` in order to warn in case of redundant usage.

After creating a `future_data` instance, *HPX* provides the associated `hpx::future`. On blocking waiting for the result, the `hpx::future` would wait for the `future_data` to become available. We invoke `future_data::set_data(hpx::util::unused)` to signal the completion of the corresponding computation.

3.4 Futures

The creation of futures is implemented in `opencl_future.{h,c}pp`. The interface consists of loosely coupled methods, which create `hpx::futures` according to the required operation mode. A registration class for the polling mode is provided additionally.

Listing 3.4 Definition of operation modes for the executor.

```
enum class future_mode : std::uint32_t
{
    callback,
    polling,
};

using callback_mode = std::integral_constant<future_mode, future_mode::callback>;
using polling_mode = std::integral_constant<future_mode, future_mode::polling>;
```

Firstly, we declare an `enum class` for the executor's operation modes. Listing 3.4 shows their definition. The usage of an `enum` enforces distinct indices among the modes, which helps to avoid potential mistakes during future changes. Moreover, the `enum class` prevents implicit conversions, further preventing another class of potential mistakes. We select an unsigned underlying type to use the available range to full extent.

For each operation mode, we declare a dedicated type. This is required to make use of overload resolution for the `shared_data`'s constructor.

After having declared supported operation modes, we provide a function template to create corresponding futures. The template is explicitly specialized for each supported operation mode as shown in Listing 3.5. This enables us to separate their declaration from definition.

The compiler selects the corresponding template specialization for `callback_mode` and `event_mode`. The default consists of the unspecialized template. Because it is only used for unsupported operation modes, it throws immediately. Consequently, we attach the `HPX_NORETURN` attribute to it. `HPX_NORETURN` corresponds to `[[noreturn]]` from standard C++. The compiler is then allowed to perform optimizations for the function, assuming that it never returns.

The goal of all `create_future()` specializations consists of creating a `future_data` instance. Consequently, it also provides the corresponding `hpx::future` for the user to retrieve the result. Additionally, mode specific operations are performed.

We first obtain a `future_data` instance through the module-internal function `construct_future_data()`. In the specialization for `polling_mode`, we additionally add the newly created instance to the polling queue via `add_to_futures_incoming()` as shown in Listing 3.6. Finally, we construct the corresponding `hpx::future` using `hpx::traits::future_access()` and return the result.

The construction of `future_data`'s instance is performed in the module-internal function `construct_future_data()`, which is shown in Listing 3.7. As prerequisite, we create the required allocators. Then, we use `traits::construct()` in order to obtain a new `future_data` instance. Besides the pointer to one of the allocators, the arguments provided to `traits::construct()` are forwarded to the constructor of `future_data`. According to the provided operation mode, one of the constructor overloads in Listing 3.3 or Listing 3.2 is used.

Listing 3.5 Explicit template specifications for `create_future()`.

```
template <typename mode>
HPX_EXPORT
HPX_NODISCARD
HPX_NORETURN
auto create_future(cl_command_queue const& queue,
                   cl_event         const  event,
                   cl_context       const  context)
    -> hpx::future<void>
{
    throw hpx::opencl::experimental::opencl_exception{"Executor mode not supported", -100};
}

template <>
HPX_EXPORT
HPX_NODISCARD
auto create_future<callback_mode>(cl_command_queue const& queue,
                                  cl_event         const  event,
                                  cl_context       const  context)
    -> hpx::future<void>;

template <>
HPX_EXPORT
HPX_NODISCARD
auto create_future<polling_mode>(cl_command_queue const& queue,
                                 cl_event         const event,
                                 cl_context       const context)
    -> hpx::future<void>;
```

Listing 3.6 Construction of `future_data` and corresponding `hpx::future`.

```
template<>
HPX_EXPORT
HPX_NODISCARD
auto create_future<polling_mode>(cl_event const event)
    -> hpx::future<void>
{
    auto ptr = construct_future_data<::polling_mode>(event);

    add_to_futures_incoming(ptr.get());

    return hpx::traits::future_access<hpx::future<void>>::create(
        ptr.release(), false);
}
```

Listing 3.7 Construction of a `future_data` from an *OpenCL* event.

```
template <typename mode>
HPX_NODISCARD
auto construct_future_data(cl_event const event)
    -> unique_ptr
{
    allocator alloc{};
    other_allocator other_alloc{alloc};

    unique_ptr ptr{traits::allocate(other_alloc, 1),
        hpx::util::allocator_deleter<other_allocator>{other_alloc}};

    traits::construct(other_alloc, ptr.get(), init_no_addref{}, other_alloc,
        event, mode{});

    return ptr;
}
```

3.5 Operation Modes

Currently supported operation modes are the callback mode and the polling mode. Common goal to all modes consists of determining whether the computation has finished. In the successful case, the according `future_data` is set to a ready state. For execution failures of the enqueued function, an exception is set instead[1].

We provide the implementation details for the callback mode. Then, we introduce the implementation of the polling mode.

3.5.1 Callback Mode

Firstly, we describe the general idea of the callback mode. On computation begin, we set the callback function `callback()` on the corresponding event as shown in Figure 3.2. Then, we rely on the *OpenCL* runtime to call `callback()` on completion. In `callback()`, we set the shared state as ready and the potentially waiting `hpx::future` obtains the result.

The implementation of the callback mode requires two steps. Firstly, we set the callback function to the newly created event. This is performed by `init_callback()`, which is executed in `future_data`'s constructor for the callback mode. We then need to process the event's state in the callback function.

As shown in Listing 3.8, we first increment the reference count to the `future_data` instance via `intrucive_ptr_add_ref()`.

[1]No exceptions can propagate across thread boundaries in C++. Therefore, exceptions are set on the `hpx::future`'s shared state instead of being directly thrown.

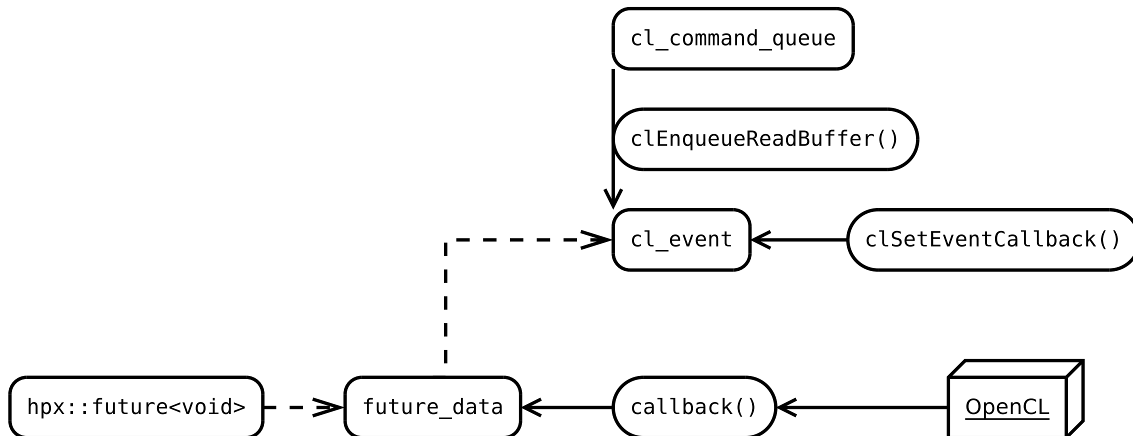

Figure 3.2: Mechanism of the callback mode.

Listing 3.8 Setting the callback on an event.

```
auto init_callback() -> void
{
    hpx::lcos::detail::intrusive_ptr_add_ref(this);

    auto const result = clSetEventCallback(event_.get(), CL_COMPLETE, callback, this);

    if (CL_SUCCESS != result)
    {
        using hpx::opencl::experimental::check_opencl_error;
        hpx::lcos::detail::intrusive_ptr_release(this);
        check_opencl_error("clSetEventCallback()", result);
    }
}
```

The callback function `callback()` is set via `clSetEventCallback()`. It expects the pointer to the event as first argument. The second argument consists of the event's state to trigger the callback. For *OpenCL* version 1.2, only `CL_COMPLETE` is supported. We then provide the callback function and the instance to the `future_data` as user defined data provided to the callback.

In this instance, we do not directly rely on the error checking functionality described in Section 3.1 on page 21. The reason is the adjusted path for the error case. Instead of throwing an exception via `check_opencl_error()` right away, we first decrement the reference count on the pointer to the `future_data` instance. Only then we throw the corresponding exception. This approach ensures proper cleanup of `future_data` in the error case.

With the callback set, the *OpenCL* implementation triggers `callback()` on completed events. Listing 3.9 shows its implementation. Special care is required for the implementation of `callback()`. Its execution is triggered by the *OpenCL* backend on a thread outside of *HPX* 's control. Therefore, we need to ensure minimal execution time for `callback()`. Blocking, locking and long-running tasks are highly discouraged.

Listing 3.9 Callback function setting the computation's status in `future_data`.

```
static auto CL_CALLBACK callback(cl_event,
                                 cl_int event_command_exec_status,
                                 void* user_data)
    -> void
{
    future_data* const data = static_cast<future_data*>(user_data);

    release_on_exit const on_exit{data};

    if (CL_COMPLETE == event_command_exec_status)
    {
        set_future_ready(data);
    }
    else if (0 > event_command_exec_status)
    {
        namespace hpxcl = hpx::opencl::experimental;

        data->set_exception(std::make_exception_ptr(
            hpxcl::opencl_exception{std::string{"OpenCL function failed ("}
                + std::to_string(event_command_exec_status) + ")",
                event_command_exec_status}));
    }
}
```

Firstly, we cast the provided user data to a `future_data` pointer. On creating the `future_data` instance, we increment the reference count to the pointer. As the execution associated with the event is finished, we need to ensure its release upon finishing the execution of `callback()`. This is performed using the module-internal `release_on_exit` type. As result, proper cleanup of `hpx::future`'s shared state is maintained for all paths.

The taken path depends on the event's error code, which is provided by the *OpenCL* backend via `event_command_exec_status`. It is `CL_COMPLETE` for successful computation. A negative integer indicates failure. In the successful case, we set the corresponding `future_data` as ready. In the failure case, we set `future_data`'s exception instead. To do so, we assemble a string indicating the error message.

Note that this operation mode will be removed in a future version of *HPX* . Therefore, we advice to regard it as deprecated. Instead, the users of the executor are encouraged to use its polling mode.

3.5.2 Polling Mode

Executor's polling mode avoids a callback on *OpenCL* 's event. Instead, the created `future_data` is added to designated containers. *HPX* 's scheduler then polls periodically the state of events in the containers. Figure 3.3 visualizes the polling mechanism. Note that polling functions are called in dedicated threads by the scheduler.

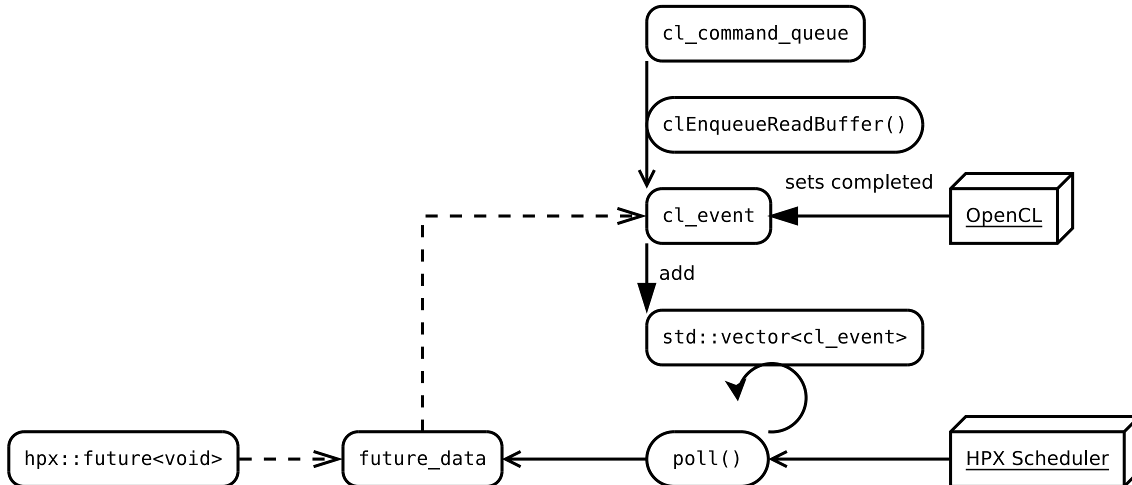

Figure 3.3: Mechanism of the polling mode.

We first describe the process of adding the events to the polling container. Then, we describe their polling mechanism.

The constructor of `future_data` does not perform polling-specific initialization. Instead, we add `future_data` instances to the lockless queue holding incoming instances. This avoids expensive locking between the adding thread and the polling threads. We achieve that by the invocation of `add_to_futures_incoming()` via `create_future()` for the polling mode. To do so, we first obtain the queue via `get_futures_incoming()`. Then, we add the newly created `future_data` instance to it.

Note that `get_futures_incoming()` yields a local, static instance of the queue. Because multiple instances of the queue are possible in the general case[2], the described mechanism is not the singleton anti-pattern. Instead, the static instance is used to delay its instantiation until its usage. On usage of the callback only, this avoids redundant instances. The same applies to the polling container, which we obtain via `get_futures_active()`.

In order to enable polling, we first add the polling function to *HPX* 's scheduler. First, we add the corresponding function pointer `polling_function_opencl_` to `hpx::threads::policies::scheduler_base`. We initialize the pointer with `null_polling_function` initially. Therefore, *OpenCL* polling is disabled by default. Then, we add the functions to set and remove the polling function from the scheduler in `set_opencl_polling_function()` and `clear_opencl_polling_function()`, respectively. Listing 3.10 shows their implementation. Note that both operations are performed atomically. The atomicity is enforced by usage of `std::atomic<polling_function_ptr>` in the scheduler.

Finally, we adjust `custom_polling_function()` in *HPX* 's scheduler. In order to call the newly added polling function, we load its pointer from the atomic container and call it. The process is shown in Listing 3.11 Note that the function pointer cannot be `nullptr`. During initialization, the polling function is set to the idling `null_polling_function()`. If polling is set, it is set to the polling function instead. Consequently, we can trade according checks for slightly increased performance.

[2] Note that `hpx::concurrency::ConcurrentQueue` is a move-only type. If moved, the state of the original instance is well-formed, but undefined according to the current C++ standard. Additionally, thread-safety during movement might be not guaranteed.

Listing 3.10 Set and clear *OpenCL* polling function in *HPX* scheduler.

```
void set_opencl_polling_function(polling_function_ptr opencl_func)
{
    polling_function_opencl_.store(opencl_func,
                                   std::memory_order_relaxed);
}

void clear_opencl_polling_function()
{
    polling_function_opencl_.store(&null_polling_function,
                                   std::memory_order_relaxed);
}
```

Listing 3.11 Calling the polling function in the *HPX* scheduler.

```
#if defined(HPX_HAVE_MODULE_COMPUTE_OPENCL)
    if ((*polling_function_opencl_.load(std::memory_order_relaxed))()
        == detail::polling_status::busy)
    {
        status = detaul::polling_status::busy;
    }
#endif
```

After the extension of *HPX* 's scheduler for the new polling function, we provide an interface to enable polling. For that purpose, we provide the `polling` structure in `opencl_future.hpp`. The structure is shown in Listing 3.12. Its purpose consists of enabling polling for a specific scope only.

We disable the copy and move semantics by disabling the corresponding constructors and operations via `HPX_NON_COPYABLE()`. This prevents certain classes of the structure's unintended use. In order to prevent repetition of implementation, we define `get_scheduler()`. It returns the *HPX* scheduler corresponding to the given thread pool, which is identified either by a string or an index. The default thread pool has index `0`.

The constructor of `polling` takes the thread pool's name in form of a string. In case the string is empty, `polling` uses the default thread pool. We declare `polling::polling()` as explicit to prevent accidental, implicit conversions from `std::string`. In the constructor, we retrieve the thread pool's scheduler and enable *OpenCL* specific polling via previously defined `scheduler_base::set_opencl_polling_function()`. Polling is disabled using `scheduler_base::clear_opencl_polling_function()`.

As result, the instantiation of a `polling` instance enables polling for all used *OpenCL* executors in the scope. Polling is disabled on leaving `polling` instance's scope.

We define `opencl_poll()` as our polling function. The execution time of the function is vital for the executor's performance. The function performs polling in two stages. Firstly, it verifies all currently executed *OpenCL* commands using their corresponding event's state. Then, it checks newly added `future_data` instances.

Listing 3.12 Registration of the polling function and its removal on destruction.

```
struct HPX_NODISCARD polling final
{
    HPX_NON_COPYABLE(polling);

    explicit polling(std::string const& pool_name = "")
        : pool_name_{pool_name}
    {
        auto* const scheduler = get_scheduler();
        scheduler->set_opencl_polling_function(&opencl_poll);
    }

    ~polling()
    {
        auto* const scheduler = get_scheduler();
        scheduler->clear_opencl_polling_function();
    }

private:

    HPX_NODISCARD
    auto get_scheduler() const
        -> hpx::threads::policies::scheduler_base*
    {
        auto const& pool = pool_name_.empty()
                         ? hpx::resource::get_thread_pool(0ull)
                         : hpx::resource::get_thread_pool(pool_name_);

        return pool.get_scheduler();
    }

    std::string pool_name_;
};
```

We use `check_future_ready()` for the first task. It checks the event's state and sets the shared state in `future_data` accordingly on completion. Moreover, it indicates whether to remove the `future_data` instance from the polling containers.

Listing Listing 3.13 shows the polling function. Note the usage of the erase-remove-idiom for the container. As the C++ compiler may assume the semantics of standard library functions, this provides extended optimization opportunities.

For the second task, we iterate through the dedicated concurrency queue. If the event is completed until it reaches the check, its `future_data` is set as ready. In case the event indicates an ongoing execution instead, it is added to the container of active `future_data` instances.

While we are using the lockless queue to insert newly created `future_data` instances into the polling mechanism, we use a `hpx::lcos::local::spinlock` for locking between scheduler's locking threads. We elaborate on our decision against using `std::mutex`.

Listing 3.13 *OpenCL* executor's polling function.

```
HPX_NODISCARD
auto opencl_poll()
    -> hpx::threads::policies::detail::polling_status
{
    using hpx::threads::policies::detail::polling_status;

    auto result = polling_status::idle;

    std::unique_lock<mutex_type> const lock{::get_mutex(), std::try_to_lock};
    if (lock.owns_lock())
    {
        auto& futures = get_futures_active();

        futures.erase(std::remove_if(std::begin(futures),
                                     std::end(futures),
                                     check_future_ready),
                      std::cend(futures));
        auto& incoming = get_futures_incoming();
        future_data_ptr data;

        while (incoming.try_dequeue(data))
        {
            if (!check_future_ready(data))
            {
                add_to_futures_active(std::move_if_noexcept(data));
            }
        }

        result = futures.empty() ? polling_status::idle
                                 : polling_status::busy;
    }

    return result;
}
```

The current implementation consists of multiple competing scheduler threads calling `opencl_poll()`. The number of cores on the target machine determines the number of polling threads. Because the data structures consist of pointers only, the execution of `opencl_opencl()` is fast compared to synchronisation overheads. Therefore, any single thread is enough to verify the state of all pending events.

The performance of `std::mutex` depends on the number of collisions between threads. In the best case, only few collisions occur between the threads. In such case, the performance of `std::mutex` depends only on setting a flag atomically. However, in case of many collisions between the threads, `std::mutex` would maintain an internal, synchronized state of all waiting threads. This would not only block the polling mode for the *OpenCL* executor, but might event block other executor's under certain conditions. Therefore, we use an improved approach.

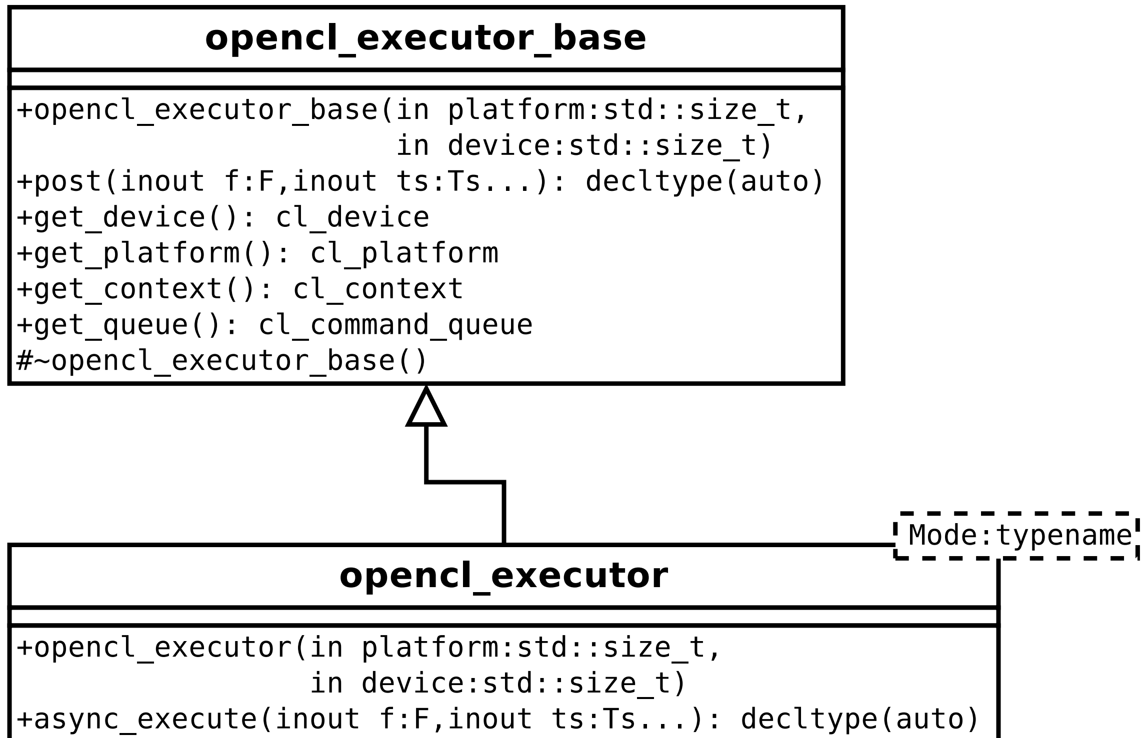

Figure 3.4: Interface of the *OpenCL* executor.

As alternative to `std::mutex`, we use `hpx::lcos::local::spinlock`. Instead of maintaining an internal, synchronized state, the spin lock would usually wait for the lock by busy-waiting. However, we only *attempt* to secure the lock for the thread. On success, one single thread executes `opencl_poll()`. On failure, the thread immediately leaves `opencl_poll()` without wasting CPU time. This especially avoids the requirement for an internal, synchronized state in order to maintain waiting threads.

3.6 Executor

Following the naming convention in *HPX* , the *OpenCL* executor's interface is implemented in `opencl_executor.{h,c}pp`. It is split into the base class `opencl_executor_base` and main class `opencl_executor` as shown in Figure 3.4. The base class is designed to provide functionality common to all usage modes. On the other hand, derived `opencl_executor` provides functionality for its specific operation mode.

The public functionality of the *OpenCL* executer consists of `opencl_executor`.

The executor expects the indices of used *OpenCL* platform and device. The initialization is performed in `opencl_executor_base`'s constructor with the assistance of helper functions. Listing 3.14 shows the executor's initialization. Note that the order of initialization is important here, as initialization of fields depends on their requirements being initialized beforehand.

Listing 3.14 Initialization of the *OpenCL* executor.

```
opencl_executor_base::opencl_executor_base(
    std::size_t const platform, std::size_t const device)
    : platform_{::get_platform(platform)}
    , device_{::get_device(platform_.get(), device)}
    , context_{::get_context(device_.get())}
    , queue_{::get_queue(context_.get(), device_.get())}
{
}
```

Additionally, we need to consider what storage types to use for the executor internally. We evaluate two options here.

- The first option consists of using pointers directly provided by *OpenCL* . We would implement the cleanup mechanism in the executor's destructor using this option. As a pointer is eight bytes wide on a 64 bit platform, the size of the executor would be

  $$\underbrace{8 \text{ byte}}_{\texttt{platform_}} + \underbrace{8 \text{ byte}}_{\texttt{device_}} + \underbrace{8 \text{ byte}}_{\texttt{context_}} + \underbrace{8 \text{ byte}}_{\texttt{queue_}} = 32 \text{ byte}.$$

  The cache line size on target machines is 64 bytes. Therefore, two *OpenCL* executors would be loaded in one cache line, potentially improving performance if multiple executors are used.

- The alternative consists of using smart pointers instead of raw pointers. Every `std::unique_ptr` instance would store the original raw pointer with a pointer to the deleter. As both pointers are 16 bytes in total, the size of the executor is

  $$\underbrace{16 \text{ byte}}_{\texttt{platform_}} + \underbrace{16 \text{ byte}}_{\texttt{device_}} + \underbrace{16 \text{ byte}}_{\texttt{context_}} + \underbrace{16 \text{ byte}}_{\texttt{queue_}} = 64 \text{ byte}.$$

  With this option, only one executor would fit into a cache line. However, the usage of smart pointers facilitates cleanup of *OpenCL* objects and protects better against potential implementation regressions.

Considering the use cases of the executor, we prioritize the slightly more robust implementation over the size optimization. Therefore, we store *OpenCL* 's platform, device, context and queue using `std::unique_ptr` with custom deleting functionality for cleanup.

Note that the initialization order forces the ordering of the fields in `opencl_executor_base` to avoid compiler warnings. Therefore, we diverge from our design decisions in this instance, as access frequency and padding is of lower priority. Nevertheless, we analyse both criteria.

We introduce no padding in `opencl_executor_base`, as all fields are of size 16. Therefore, this aspect is satisfied. We determine that the queue is the most frequently accessed field. Although it is the last field, it would be loaded in the same cache line as others. We determine that the offset of the field in the cache line is of lesser priority.

We retrieve the pointer to the *OpenCL* platform in `get_platform()` via `clGetPlatformIDs()`. The helper function is shown in Listing 3.15. This requires two steps. Firstly, we determine the number of available platforms. In order to achieve that, we provide a pointer to `clGetPlatformIDs()` as last

Listing 3.15 Creating *OpenCL* platform by index.

```
HPX_NODISCARD
auto get_platform(std::size_t const platform)
    -> std::unique_ptr<_cl_platform_id, decltype(&hpxcl::cleanup_platform)>
{
    cl_uint number_platforms{};
    opencl_checked("clGetPlatformIDs()",
                   clGetPlatformIDs,
                   0u,
                   nullptr,
                   &number_platforms);

    if (number_platforms <= platform) {
        HPX_THROW_EXCEPTION(hpx::bad_parameter,
                            "hpx::opencl::experimental::get_platform()",
                            "invalid platform index");
    }

    auto const platforms =
        std::make_unique<cl_platform_id[]>(number_platforms);

    opencl_checked("clGetPlatformIDs",
            clGetPlatformIDs,
            number_platforms,
            platforms.get(),
            nullptr);

    auto const selected = platforms.get()[platform];
    return {selected, hpxcl::cleanup_platform};
}
```

parameter. The according variable then contains the number of available *OpenCL* platforms. Finally, we use the index to retrieve the pointer to the desired platform. Again, we use `clGetPlatformIDs()` with different parameters to retrieve the pointer to the platform. The result consists of the platform's pointer and the corresponding deleter, packed in a `std::unique_ptr`. This ensures proper cleanup on `opencl_executor_base`'s destruction.

We use the same approach to retrieve the platform's device, according to provided index. To achieve that, we utilize `clGetDeviceIDs()` in the same fashion.

As the next step, we create the device's context via `clCreateContext()`. Its usage is shown in Listing 3.16.

Finally, we create the *OpenCL* command queue. We utilize `clCreateCommandQueue` to achieve that, which expects previously created pointers to device and context.

After initialization, the created platform, device, context and queue are accessible through the corresponding getter functions. While `std::unique_ptrs` are stored internally, the getters provide raw pointers. The reason consists in their anticipated usage for *OpenCL*'s API. We avoid passing pointers to corresponding deleters that way, while maintaining ownership over the field's destruction.

Listing 3.16 Creating *OpenCL* context on device.

```
HPX_NODISCARD
auto get_context(cl_device_id const& device)
    -> std::unique_ptr<_cl_context, decltype(&hpxcl::cleanup_context)>
{
    cl_int errcode{0};
    auto const context = clCreateContext(nullptr, 1u, &device, nullptr, nullptr, &errcode);

    check_opencl_error("clCreateContext()", errcode);

    return {context, hpxcl::cleanup_context};
}
```

Note the protected destructor in `opencl_executor_base`. As base class, we cannot declare `opencl_executor_base` as *final*. This leaves us with three options, which we elaborate in appendix Appendix B.1 on page 71.

The executor is capable of one-way and two-way execution via `opencl_executor_base::post()` and `opencl_executor<>::async_execute()`, respectively. One-way execution is the fire-and-forget approach, which does not yield a future. On the other side, the two-way execution provides a future. Both functions use the executor's internal command queue and take a `clEnqueue*` function with their parameters as parameters. Listing 3.17 and Listing 3.18 show their implementation. Compared to the call to a raw *OpenCL* function, two modifications in the parameter list are expected.

The first parameter to `clEnqueue*` is the command queue. As the executor-internal queue is used, the first parameter is omitted in the call. The executor provides the pointer to its internal queue instead. Secondly, the last parameter is *OpenCL* 's event. The event is used to relay the execution status to the `future_data` instance, pointed to by `hpx::future`. The internal usage of *OpenCL* 's event prevents executor's user from providing a custom event for `async_execute()`. Therefore, the last parameter to the provided `clEnqueue*` function is also omitted.

Because `post()` does not yield a feature, it does not utilize the provided event. This prompts a decision whether to enable executor's user to use events for the one-way execution.

- If events are used, the user can use `hpx::apply()` with an arbitrary *OpenCL* event. Especially, the usage of custom events is possible to trigger the execution explicitly or depending on *OpenCL* commands outside of the *OpenCL* executor. However, this might be not expected because events are not supported by the two-way execution.
- The prevention of event's usage for the one-way execution aligns with the interface for the two-way execution. This option therefore aligns with the rule of least surprise while designing the executor's interface.

We decide for the second option. Therefore, we expect no event pointer to `post()` and set the according argument to `nullptr` unconditionally as shown in Listing 3.17.

Finally, we need to provide *HPX* with the information that `opencl_executor` is one-way and two-way executable. To do so, *HPX* cannot use concepts, which were introduced in C++20. The reason consists of C++14 as requirement. This leaves *HPX* with two options. It could have used SFINAE

Listing 3.17 One-way execution of provided *OpenCL* function. Error checking is omitted for brevity.

```
template <typename F, typename... Ts>
auto post(F&& f, Ts&&... ts) -> decltype(auto)
{
    return f(queue_.get(), std::forward<Ts>(ts)..., nullptr);
}
```

Listing 3.18 Two-way execution of provided *OpenCL* function. A corresponding `hpx::future` is returned. Error checking is omitted for brevity.

```
template <typename F, typename... Ts>
auto async_execute(F&& f, Ts&&... ts) -> decltype(auto)
{
    cl_event event{};
    auto const result = f(queue_.get(), std::forward<Ts>(ts)..., &event);

    return create_future<Mode>(queue_.get(), event, context_.get());
}
```

techniques to determine whether an executor provides the required methods. Instead, an explicit approach is taken. It requires us to provide explicit template specializations for each type as shown in Listing 3.19.

The benefit of this approach consists in its flexibility. For instance, this gives us the option to introduce another operation type for the executor, which would only support one-way execution.

We provide short type definitions for convenience as shown in Listing 3.20. Because we are operating in namespace `hpx::opencl::experimental`, the names are not prone to collisions with other types in the *HPX* project. Therefore, we provide shorthand declarations for implemented types.

Firstly, we determine the default for the executor's operation mode. It is used for `hpx::opencl::experimental::executor<>`. Because C++14 is supported, we can not benefit from class template deduction. As consequence, we currently cannot omit the angle brackets after the type's name. C++17 and later would allow us to shorten the executor's declaration.

Additionally, we provide explicit types for the callback and polling mode. Because those are easy to remember, this minimizes cognitive load for the executor's user.

Listing 3.19 Enabling one-way and two-way execution for the *OpenCL* executor.

```
namespace hpxcl = hpx::opencl::experimental;

template <>
struct is_one_way_executor<hpx::opencl::experimental::opencl_executor<
    hpxcl::callback_mode>> : std::true_type
{
};

template <>
struct is_two_way_executor<hpx::opencl::experimental::opencl_executor<
    hpxcl::callback_mode>> : std::true_type
{
};

template <>
struct is_one_way_executor<hpx::opencl::experimental::opencl_executor<
    hpxcl::polling_mode>> : std::true_type
{
};

template <>
struct is_two_way_executor<hpx::opencl::experimental::opencl_executor<
    hpxcl::polling_mode>> : std::true_type
{
};
```

Listing 3.20 Short, memorable names for the *OpenCL* executor providing different operation modes.

```
using default_mode = callback_mode;

template <typename mode = default_mode>
using executor = opencl_executor<mode>;

using executor_callback = opencl_executor<callback_mode>;
using executor_polling = opencl_executor<polling_mode>;
```

4 Testing

We verify the correctness of the *OpenCL* executor's implementation using unit tests. The tests are distributed in `libs/full/compute_opencl/tests/unit/` across the following directories.

`performance/` contains performance tests. While performance tests contain assertions to ensure their correctness, their main purpose is performance measurement of the *OpenCL* executor.

`regresions/` contains regression tests. As the implementation is too current to contain regressions, the directory currently contains no tests.

`unit/` contains unit tests. We focus unit tests in the sections of this chapter.

In case the executor's operation mode might influence the test's correctness, the executor is tested using the callback and polling mode.

4.1 Usage

The build system requires `HPX_WITH_TESTS` set to `ON` in order to build the test cases. Unit tests additionally require `HPX_WITH_TESTS_UNIT` set to `ON`.

The tests are compiled via `make tests.unit.modules.compute_opencl`.

Moreover, the user can enable performance benchmarks by setting `HPX_WITH_TESTS_BENCHMARKS` to `ON`. This enables the compilation of the performance benchmark. Id contains additional assertions for its use case.

One option to run tests consists of invoking the executable directly. In case of failure, this provides the user with the error message of thrown exception. Alternatively, `ctest -R 'opencl'` runs all tests related to the *OpenCL* executor.

4.2 Test Setup

The build setup for tests is configured in `CMakeLists.txt` of the corresponding directory. In order for `ctest` to pass the expected *OpenCL* source and kernel to the tests, `opencl_executor_PARAMETERS` is configured accordingly. The configuration especially ensures an absolute path passed to the test. If we run the executable directly issuing `bin/opencl_executor_test`, the test assumes the build directory as current working directory to determine the relative path to *OpenCL* 's source code. Moreover, it assumes the default directory structure of the *HPX* project. We have considered following options for our decision.

Listing 4.1 Aligning return value with test results.

```
auto const result = hpx::init(argc, argv, init_params);
return result or hpx::util::report_errors();
```

- One option consists of creating a dedicated configuration file using `configure_file()` provided by `cmake`. The advantage consists of absolute paths. That enables the user to issue run the testing executable from any directory without providing an explicit path if used from any other directory than the one used for compilation. However, this would introduce a dependency on the build system, which is not precedented by the *CUDA* executor.
- The alternative consists of providing a default path assuming the directory structure and working directory. In this case the user needs to provide an explicit path in some cases. However, we avoid an unprecedented dependency on the build system using this option. Furthermore, the assumption simplifies the test configuration and implementation.

As conclusion, we decide to implement default paths for the tests for the standalone test executable.

The tests are compiled issuing `make tests.unit.modules.compute_opencl` in the build directory of *HPX* 's source tree.

Note one detail regarding the return value of the tests. On the one hand, `ctest` considers the return type only in order to determine whether the tested executable passes all tests successfully. On the other hand, `HPX_TEST()` and `HPX_TEST_MSG()` only display error messages without affecting the return type directly. We align both testing methods in the return value of the application as shown in Listing 4.1.

On the one hand, we retrieve the executable's return value through the return value of `hpx::init()`. On the other hand, we retrieve potentially failing test assertions via `hpx::util::report_errors()`. The final return value consists if one of the values evaluates to `true`. Consequently, the tests in the executable fail for `ctest` if any of the *HPX* checks fail. This is the expected behavior for the test's user.

Note that we use `operator or` instead of the more common `operator ||` in order to slightly improve the code's maintainability.

4.3 Test Cases

4.3.1 *OpenCL*

The standalone *OpenCL* unit test is located in `opencl.cpp`.

By using the *OpenCL* backend, the *OpenCL* executor in *HPX* relies on its availability and correct results. To verify its correct execution, the *OpenCL* backend is tested directly in this test. It verifies the availability and correctness of results provided by the *OpenCL* backend. If failures are encountered in this test, the search for regressions can safely ignore *OpenCL* executor's implementation itself. Therefore, solutions are easier and more swiftly found.

The test itself calculates the addition of vectors using two independent methods. Firstly, the vectors are added using the CPU and standard C++ functionality only. Because the standard library of C++ is well tested, we can rely on its results. Secondly, vectors are added using the *OpenCL* backend. Note that the C++ bindings of *OpenCL* are used to ensure absence of memory leaks. The test then consists of both results' comparisons. It fails, if the addition using the standard library does not match with the results provided by the *OpenCL* device.

To exclude other causes for failure, the dependencies to *HPX* are minimized. The functionality of *HPX* is only used for testing.

4.3.2 Executor

Unit tests regarding the core functionality of the *OpenCL* executor are located in `opencl_executor.cpp`.

Initialization

We verify the correct initialization of the executor in `test_initialization_from_valid_ids_is_successful()`. The goal is to ensure the correct initialization of the executor in all modes.

Note that the order of returned devices from the *OpenCL* backend cannot be verified to be constant for multiple invocations according to its documentation. Therefore, we need to consider precision of the test against its portability to different platforms.

- The first option consists of relying on the information provided in expected order on each request. This has the advantage of testing against the same platform and device, assuming their indices in the provided result are equal. However, this might lead to unexpected and unreliable tests if the order of provided devices changes.
- The second option consists of relying on less provided information, which is constant. This has the disadvantage of less precise tests. However, this improves portability, as no undocumented assumptions are necessary regarding any particular *OpenCL* implementation.

We decide for the second option. The purpose of the *OpenCL* executor consists of extending usable platforms for *HPX* . Therefore, portability is of priority.

As conclusion, we verify that the platform and the device have been initialized only. Additionally, we verify the initialization of the executor's *OpenCL* context and the command queue.

Destruction

After destruction of the executor its platform, device, context and command queue need to be released using *OpenCL* 's API functions. Especially, the corresponding pointers need to provide default values.

We verify both criteria by calling the executor's destructor and examining returned pointers. The test is successful, if at least one pointer is not in valid state. We use that as indication that the constructor is called. All all fields in `opencl_executor_base` and as extension in `opencl_executor` are of type `std::unique_ptr`. On invocation of the distructor, all fields are cleaned up.

One-Way-Execution

We verify the correctness of the one-way execution of the executor in `test_apply_executes()`. As prerequisite, we first generate the input in the array `buffer_in`. We are using a raw array instead of `std::array`, as the latter requires the type as template argument. Since `cl_int` has alignment attributes attached to it, this leads to warnings. By using a raw array, we avoid `cl_int`'s usage as template argument and therefore corresponding warnings.

As next step, we populate the input buffer on the *OpenCL* device using `clCreateBuffer()`. The flag `CL_MEM_READ_ONLY` guards against its modification by the executor. The flag `CL_MEM_COPY_HOST_PTR` populates the buffer on the device with the contents of the host buffer.

Finally, we use `hpx::apply()` in order to trigger the one-way execution of the *OpenCL* executor. Note that we pass the flag `CL_BLOCKING` to it. Therefore, the function is blocked until the buffer is read. After the execution of `hpx::apply()`, we verify the data retrieved from the device. The test is successful, if the contents of the input buffer and the output buffer match.

Shared State

Used `hpx::future` instances are required to point to a shared state, which consists of `future_data`. Invocation of `get()` throws an exception, if the `hpx::future` does not satisfy this condition.

The test verifying the validity of returned `hpx::futures` is implemented in `test_async_valid()`. We use the same setup an in `test_apply_executes()`. However, we verify the presence of the shared state by invoking `valid()` on the returned future from `hpx::async()`.

The test succeeds, if the executor returns futures that point to a valid shared state.

Future is Ready after Waiting

As soon as the result of the *OpenCL* backend is available, `hpx::future` instances are expected to reflect its presence. In `test_async_ready_after_waiting()`, we wait for the `hpx::future` to become ready after waiting on the result via `wait()`. The test is successful, if the returned `hpx::future` becomes ready eventually.

The execution of the test for both modes ensures the correct implementation of the callback and polling mode. Therefore, this test guards against classes of failures in both modes, which would fail to set the shared state as required.

As additional remark, note the usage of `wait()` instead of `wait_for()` or `get()`.

Note the usage of `wait()` instead of `wait_for()` or `get()`. We elaborate on our decision.

- The usage of `wait_for()` would provide better feedback in case of infinite loops. For instance, a timeout of two seconds can be set. Instead of blocking the test execution, the test would then fail eventually. However, `wait_for()` has a negative performance impact on the test. As the test already takes about two seconds to complete, we prioritize its performance over improved feedback.
- The usage of `get()` would block the execution until the result is available. Contrary to `wait()` and `wait_for()`, it is intended to fetch the result right away. The operation would modify the `hpx::future` instance. Therefore, the check for whether the future is ready would not be possible in some cases.
- The usage of `wait()` provides better performance than `wait_for()`. The waiting period is not bounded. However, the future is not modified. Because `wait()` does not modify the future, we can examine the future after waiting.

We prioritize performance and the ability to examine the future. Therefore, we decide for the latter option.

Correctness of Provided Result

We verify the correct execution of the *OpenCL* executor using the two-way mode in `test_async_provides_correct_result()`. The test setup is similar to the case for the one-way mode. Instead of using a blocking operation, we rely on the returned `hpx::future` instead.

Because the future is not further examined in the test, we use `get()` for synchronization. The test verifies the correct invocation of `clEnqueueReadBuffer()`. If the read contents match with the input, the test is successful.

Asynchronous Processing

The main goal of the executor consists of facilitated asynchronous invocation of *OpenCL* functionality. Therefore, we verify the executor's asynchronous execution in `test_async_is_processed_asynchronously()`.

In this test, we have some challenges to solve. Firstly, *OpenCL* does not provide any delaying functionality for the kernels. That is, we cannot execute a kernel for a predetermined period of time. Secondly, we need to invoke an actual *OpenCL* kernel, because we rely on *OpenCL* updating the state of the corresponding event. Therefore, we cannot use a custom function to mimic its behavior, which would allow a predetermined delay using `std::this_thread::sleep_for()`.

The execution of a native kernel would solve both challenges. Firstly, the *OpenCL* backend would manage the state of the corresponding event. Secondly, we would gain the ability to execute a kernel in form of a simple function using C. In such a native kernel, we potentially could use `sleep()` to force a predetermined execution time. However, native kernels are not supported on all devices. This can be verified by examination of the `CL_EXEC_NATIVE_KERNEL` capability in device's `CL_DEVICE_EXECUTION_CAPABILITIES`. Therefore, we prioritize the portability of the test and avoid the usage of native kernels.

Instead, we design an *OpenCL* kernel, which we attempt to run as long as necessary for the test. The kernel is located in `long_running.cl`. It performs basic, computationally expensive, arithmetic operations on the input.

By default, optimizations for *OpenCL* kernels are enabled. As we are using basic arithmetic operations, the compiler might optimize the computation. This would defeat the purpose of the kernel. In order to maintain the necessary runtime, we avoid optimizations on the kernel. For that purpose, we pass the option `-cl-opt-disable` to `clBuildProgram()` in `opencl_utils.cpp`.

The core functionality of the test consists of invoking the long running kernel. Then, we verify that the future is not ready via `hpx::future::is_ready()`. We wait for the future to complete and verify that the future is ready after completion.

The test setup verifies asynchronous execution of the provided *OpenCL* function. Only asynchronous operations can change the state of the future in a separate thread. Therefore, one of the checks would fail if the execution is synchronous.

4.3.3 Performance Benchmark

For the performance test, we lay the focus on the execution performance of the *OpenCL* executor. Nevertheless, the test's implementation contains some checks. Running all unit tests using `ctest -R` also includes the performance test by default.

However, the test result of the matrix multiplication is not verified by default. For the result's verification, the same matrix multiplication is additionally performed on the CPU. Moreover, we implement the multiplication on the CPU with simplicity in mind. This facilitates maintainability in the future with the disadvantage of decreased performance. Finally, the unit tests for the entire *HPX* project are computationally expensive. Therefore, we disable the result verification by default.

In order to verify the result, add `--verify-result=true` to the command line parameters of the benchmark test.

4.4 Results

All provided tests for the *OpenCL* executor are passing as shown in Listing 4.2. The additional execution time for all *HPX* unit tests is approximately 2.86 seconds using the *GeForce RTX 3080* GPU. Taking the setup time into account, the runtime impact is under a low number of seconds[1].

Note that the custom kernel in `test_async_is_processed_asynchronously()` fails occasionally. In some scheduling scenarios, it still completes its execution before the check for a busy feature. Execution of the tests in a busy loop yields one failure in approximately one hour. We performed the verification on a calm machine with no increased priority of execution.

[1] The exact runtime depends on used hardware.

Listing 4.2 Test results of all unit tests.

```
Test project /data/scratch/schupiml/pro/hpx/build
    Start 256: tests.unit.modules.compute_opencl.opencl
1/3 Test #256: tests.unit.modules.compute_opencl.opencl ..........................   Passed
 0.32 sec
    Start 257: tests.unit.modules.compute_opencl.opencl_executor
2/3 Test #257: tests.unit.modules.compute_opencl.opencl_executor ................   Passed
 1.93 sec
    Start 258: tests.performance.modules.compute_opencl.opencl_executor_stream
3/3 Test #258: tests.performance.modules.compute_opencl.opencl_executor_stream ...   Passed
 0.56 sec

100% tests passed, 0 tests failed out of 3

Total Test time (real) =   2.86 sec
```

One solution consists of increasing the kernel's execution time. This still would not give guarantees for successful test runs. As advantage, this would reduce the probability of occasional failures. However, this would also impact the execution time of tests for users with other hardware. Therefore, we hold that the current execution time with occasional failures is the correct solution.

Additionally, the tests have been conducted on an *Intel i5-4300U* in order to verify the correct execution on an CPU.

4.5 Analysis

Firstly, the tests show the correct implementation of the *OpenCL* executor given the tested conditions. The execution time of the tests is within the noise of the combined execution time for all tests.

However, the test implemented in `test_async_ready_after_waiting()` greatly contributes to the total execution time. We decide to include the test in the current version. If the impact on the execution time becomes of concern in the future, the removal of the test would reduce the execution time in a range from one to two seconds.

5 Benchmarking

We use three benchmarks to explore the runtime behavior of various devices using the implemented *OpenCL* executor. Firstly, we multiply matrices and measure the runtime of a generic kernel. Secondly, we copy data to the device and retrieve the result. This provides us with performance impact introduced by data transfer between host and device. Lastly, we determine the effect of the used work group's number to calculate the result on the device.

The general goal of the benchmarks consists of identifying potential bottlenecks rather than providing comparison with the *CUDA* implementation or evaluation of absolute processing times. Detailed tables of some results are provided in Appendix A on page 67.

5.1 Matrix Multiplication

This benchmark consists of matrix multiplication. The kernel features some optimizations. However, the optimizations are not exhaustive to allow better identification of performance bottlenecks. We use four work groups in order to calculate the results on different devices.

The goal of the benchmark consists of identifying the best performing hardware. Moreover, we identify bottlenecks for used devices.

5.1.1 Implementation

We use Algorithm 5.1 for runtime measurement. Firstly, we allocate and populate required buffers, create the kernel, set its arguments and measure its execution time. Then, we perform the execution of the kernel 1000 times. The performance result consists of the average runtime over all iterations.

In the benchmark, we are careful not to include the creation time of the future itself. Instead, the same future instance is reused for all iterations. Moreover, we populate the input buffers with random numbers using a constant seed. This provides us the advantage to test the general case, while maintaining the reproducibility of the benchmark.

The kernel itself is presented in Listing 5.1. It accepts the dimensions of the input matrices in `M`, `N` and `K`. Additionally, it expects the input matrices as arrays in `A` and `B`. We expect the matrices in column-major format. Array `C` is populated with the result.

For matrix multiplication, we use the `block_size` of 4. Consequently, the matrices are multiplied using blocks of 4×4 elements for each work group. *OpenCL* enforces the usage of blocks' multiples for input and output buffers. Therefore, we use square matrices and measure their dimensions in multiples of used blocks' sizes.

Algorithm 5.1 Runtime benchmark of matrix multiplication.

procedure OPENCL_EXECUTOR_MULTIPLICATION_TEST($number_iterations \in \mathbb{N}$)
  ALLOCATE_HOST_BUFFERS
  POPULATE_HOST_BUFFERS
  ALLOCATE_DEVICE_BUFFERS
  CREATE_KERNEL
  SET_KERNEL_ARGUMENTS
  TIMER_START
  for all $iteration \in \{1 \ldots number_iterations\}$ **do**
    EXECUTE_KERNEL
    WAIT_ON_FUTURE
  end for
  TIMER_END
  PRINT_ITERATION_TIME_AVERAGE
  CLEANUP
end procedure

5.1.2 Results

Figure 5.1 shows the results for the callback mode of the *OpenCL* executor. Similarly, Figure 5.2 shows the results for the polling mode. Moreover, we show the precise average runtime for both modes in Table A.1 and Table A.2, respectively. Both tables are presented in Appendix A.1 on page 67.

In both graphs, the abscissa corresponds to the number of matrix blocks in one direction. As we use blocks of 4×4 in size, a block count in one direction of 64 corresponds to matrix dimension of

$$4 \cdot 64 = 256$$

elements. Table 5.1 shows the relations between number of blocks, matrix dimensions used for multiplication and the total number of computed elements. The according numbers correspond to the size of one of three matrices used in calculation.

We identify two benchmarked sizes of interest.

- Firstly, the runtime for the callback mode increases abruptly between 16 and 64 blocks, which corresponds to 64×64 and 256×256 matrices. Focusing the callback and event mode separately, we show the relevant section in Figure 5.3 and Figure 5.4, respectively.
- Secondly, we find the runtime of small matrices notable. We display the runtime for matrices up to dimension 36×36 in Figure 5.5 for both modes.

5.1.3 Analysis

Overall, we measure increasing runtime for calculations involving more data.

Listing 5.1 *OpenCL* kernel used for matrix multiplication benchmark.

```
#define block_size 4u

void kernel multiply(int const M,
                     int const N,
                     int const K,
                     global float const * const A,
                     global float const * const B,
                     global float* const C)
{
    int const row = get_local_id(0);
    int const column = get_local_id(1);

    int const row_global = block_size * get_group_id(0) + row;
    int const column_global = block_size * get_group_id(1) + column;

    local float A_sub[block_size][block_size];
    local float B_sub[block_size][block_size];

    float accumulator = 0.0f;

    int const count_blocks = K / block_size;

    for (int block_index = 0; block_index < count_blocks; ++block_index)
    {
        int const row_tiled = block_size * block_index + row;
        int const column_tiled = block_size * block_index + column;

        A_sub[column][row] = A[column_tiled * M + row_global];
        B_sub[column][row] = B[column_global * K + row_tiled];

        barrier(CLK_LOCAL_MEM_FENCE);

        for (int index = 0; index < block_size; ++index)
        {
            accumulator += A_sub[index][row] * B_sub[column][index];
        }

        barrier(CLK_LOCAL_MEM_FENCE);
    }

    C[column_global * M + row_global] = accumulator;
}
```

Number of Blocks	Matrix Dimensions	Number of Elements
1	4×4	16
64	256×256	25 536
128	512×512	262 144
256	$1\,024 \times 1\,024$	1 048 576
512	$2\,048 \times 2\,048$	4 194 304
1024	$4\,096 \times 4\,096$	16 777 216

Table 5.1: Relation of block count, matrix dimensions and number of participating `float` elements.

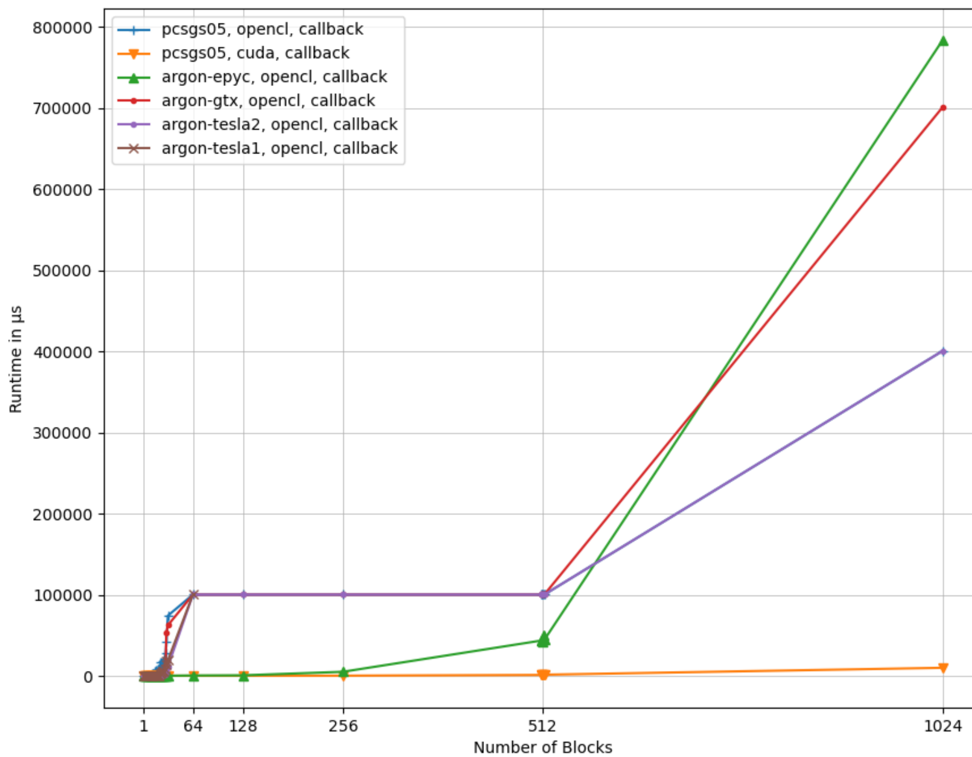

Figure 5.1: Runtime of matrix multiplication depending on block count for callback mode.

For the callback mode of the executor, we identify two major performance bottlenecks. While the average multiplication requires $158.0\mu s$ for 128×128 matrices, it raises to $454.88\mu s$ for 256×256 matrices. After that, the runtime stays constant for matrices up to dimensions of 2048×2048. We verify a similar behavior for the polling mode. However, the runtime is less prone to the bottleneck using the polling mode.

Both modes experience degraded performance for matrices larger than 2048×2048 in dimension. For larger calculation data, the performance drops drastically.

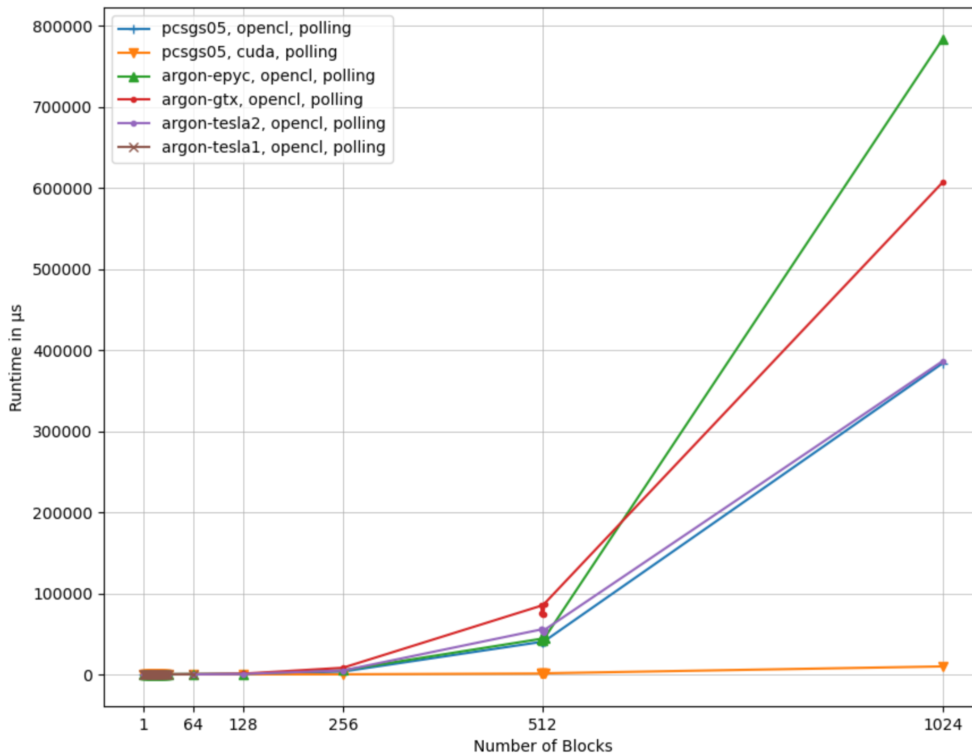

Figure 5.2: Runtime of matrix multiplication depending on block count for polling mode.

For small matrices, the polling mode exhibits better performance overall. Moreover, the performance for the polling mode is more stable. On the contrary, the performance measurements show greater variations for the callback mode. The average always exceeds the runtime using the polling mode.

5.2 Streaming

In this benchmark, we measure the required runtime to transfer data from the host to the device. The runtime includes the allocation of buffers on the target device. Moreover, we measure the time required to retrieve the result from the device.

The goal of the benchmark consists of identifying performance bottlenecks caused by memory migration between the host and device.

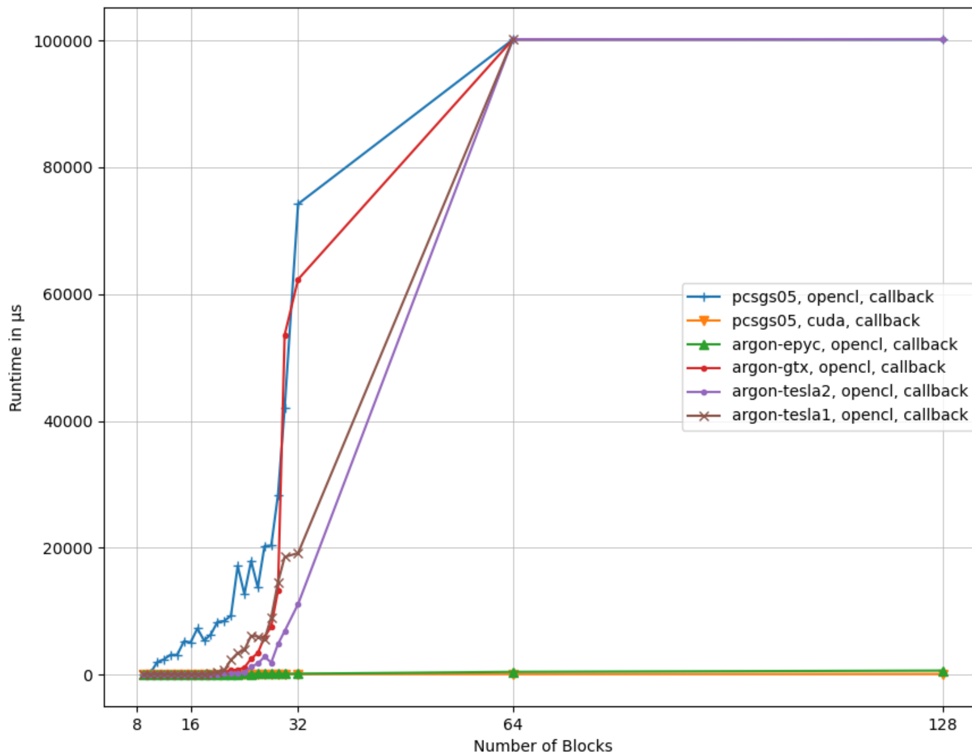

Figure 5.3: Runtime of matrix multiplication between 8 and 128 blocks using callback mode.

5.2.1 Implementation

The benchmark is outlined in Algorithm 5.2, which is based on the general use case. After setting up host buffers, we repeatedly allocate three buffers on the device. Two buffers are populated with input data from the host. After the execution of the kernel, we repeatedly retrieve the result from the device again. The number of iterations is 1000 for both operations.

The kernel consists of a simple vector addition as its result is not relevant for this benchmark. To facilitate comparison with other graphs, we use multiples of block size 4 in the mantissa. However, the kernel does not utilize work groups.

5.2.2 Results

We show the necessary runtime for the buffers' allocation in Figure 5.6. Figure 5.7 shows the average runtime to retrieve the result again. Consult Table A.3 and Table A.4 for device's input and output, respectively. Both tables are located in Appendix A.2 on page 67.

Again, we focus the suspected bottleneck around block size 64 in Figure 5.8 and for input and output Figure 5.9, respectively.

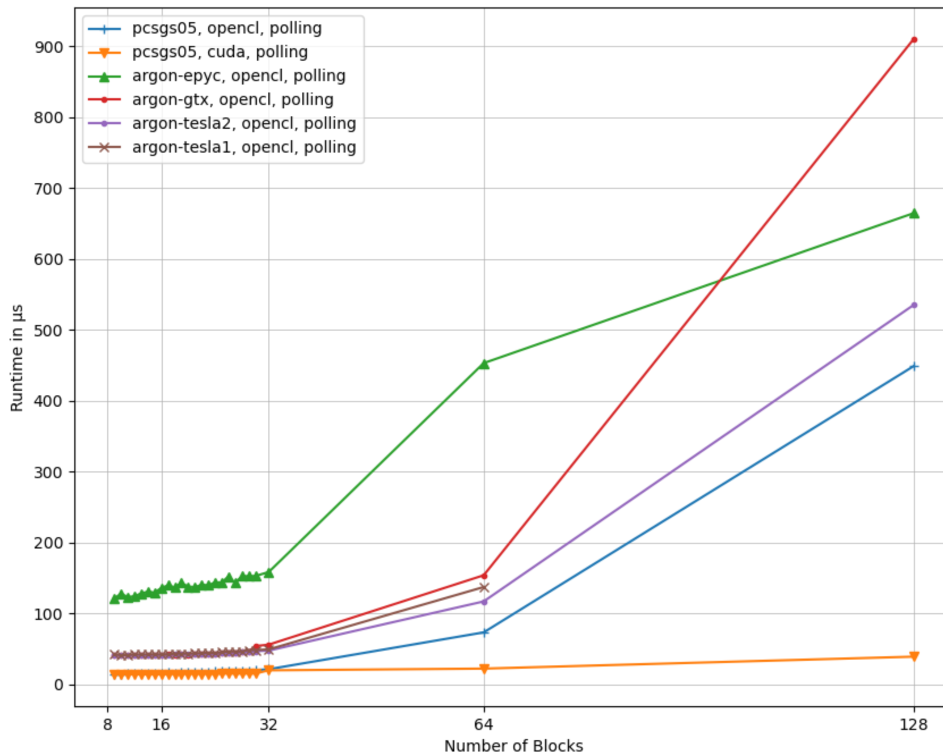

Figure 5.4: Runtime of matrix multiplication between 8 and 128 blocks using polling mode.

Finally, we provide focused visualisation of the results for small inputs in Figure 5.10.

5.2.3 Analysis

The runtime required for buffer allocation and population increases with memory requirements and the volume of transferred data to the device. Especially on machine `argon-epyc`, we hit a bottleneck using buffers with larger than 128 blocks in size. In this instance, the required runtime to allocate buffers increases drastically after this point. We observe a similar behavior for other machines, albeit to a lesser extent.

Retrieval of results performs better in our benchmark. The reason consists in less memory being transferred from the device back to the host. Additionally, no buffer allocation is required. In order to retrieve results of larger size, more runtime is generally required. A not linear performance regression can be observed for buffer sizes larger than 128 blocks.

Additionally, we record a performance spike for 256 blocks on the `argon-epyc` machine. We confirm the performance regression for this particular case over several benchmark runs.

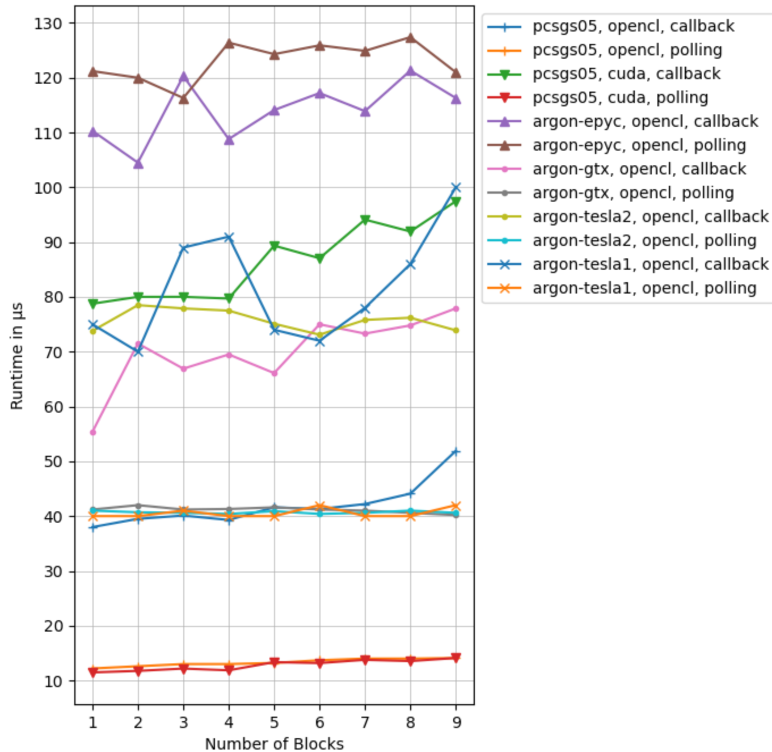

Figure 5.5: Runtime of matrix multiplication for all modes and small matrices.

5.3 Block Size

In the matrix multiplication test, we defined a constant block size for the matrices. Then, we benchmarked the runtime for various matrix sizes. In this benchmark, we set the matrix size to constant size of 256×256. Then, we use benchmark the runtime for various block sizes.

The goal consists of confirming the best work group size, which is reported by the individual devices.

5.3.1 Implementation

This benchmark focuses on used work group size. It follows the same principle as the matrix multiplication test. Especially, the same kernel is used. With constant matrix size, we divide the matrix in blocks of 2 to 64 elements in dimension.

Algorithm 5.2 Runtime benchmark of data input-output.

```
procedure OPENCL_EXECUTOR_MULTIPLICATION_TEST(number_iterations ∈ ℕ)
    ALLOCATE_HOST_BUFFERS
    POPULATE_HOST_BUFFERS
    TIMER_START
    for all iteration ∈ {1 . . . number_iterations} do
        ALLOCATE_DEVICE_BUFFERS
    end for
    TIMER_END
    PRINT_ITERATION_TIME_AVERAGE
    CREATE_KERNEL
    SET_KERNEL_ARGUMENTS
    EXECUTE_KERNEL
    WAIT_ON_FUTURE
    TIMER_START
    for all iteration ∈ {1 . . . number_iterations} do
        RETRIEVE_RESULT
    end for
    TIMER_END
    PRINT_ITERATION_TIME_AVERAGE
    CLEANUP
end procedure
```

5.3.2 Results

Figure 5.11 and Figure 5.12 display the runtime for the callback and polling mode, respectively.

Machine `argon-eypc` has not provided any results for more than 16 work groups.

5.3.3 Analysis

As expected from the matrix multiplication benchmark, the performance for the polling mode is superior to the callback mode's performance. For both modes, we notice two block sizes of interest. Using eight work groups on the device yields better performance for both modes. The effect is especially noticeable for the callback mode due to its already increased runtime. We achieve best performance with 64 work groups for all capable machines. This aligns with the information provided by their respective devices.

Overall, the best results are expected using the polling mode on machines `pcsgs05` and `argon-tesla2` using 64 work groups.

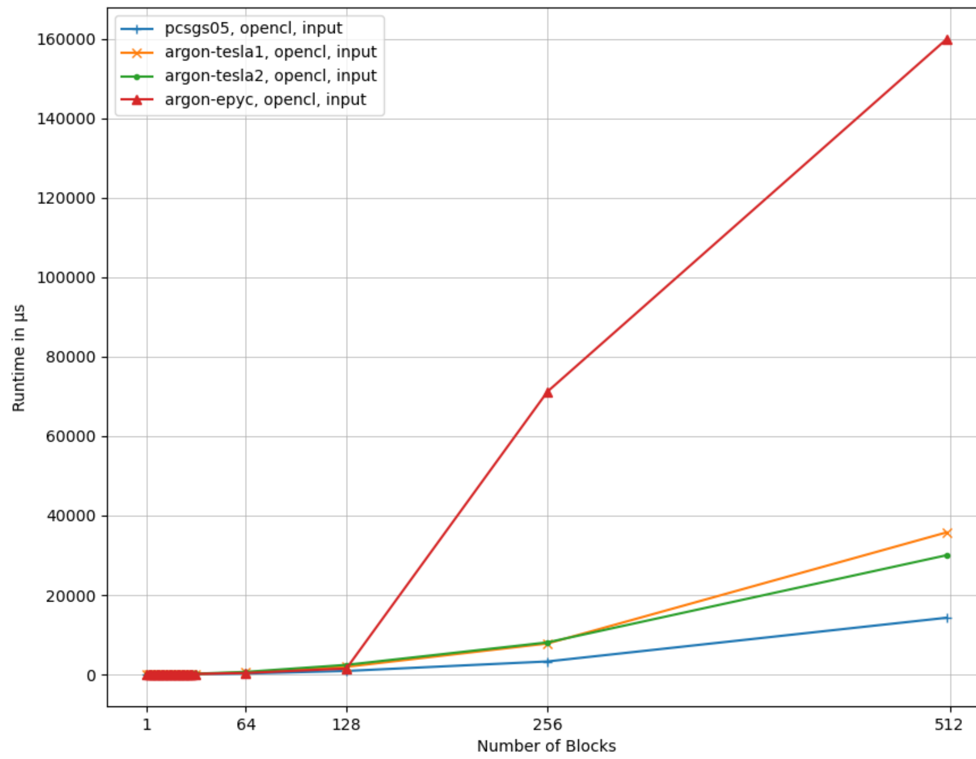

Figure 5.6: Runtime of allocating and populating input buffers on device.

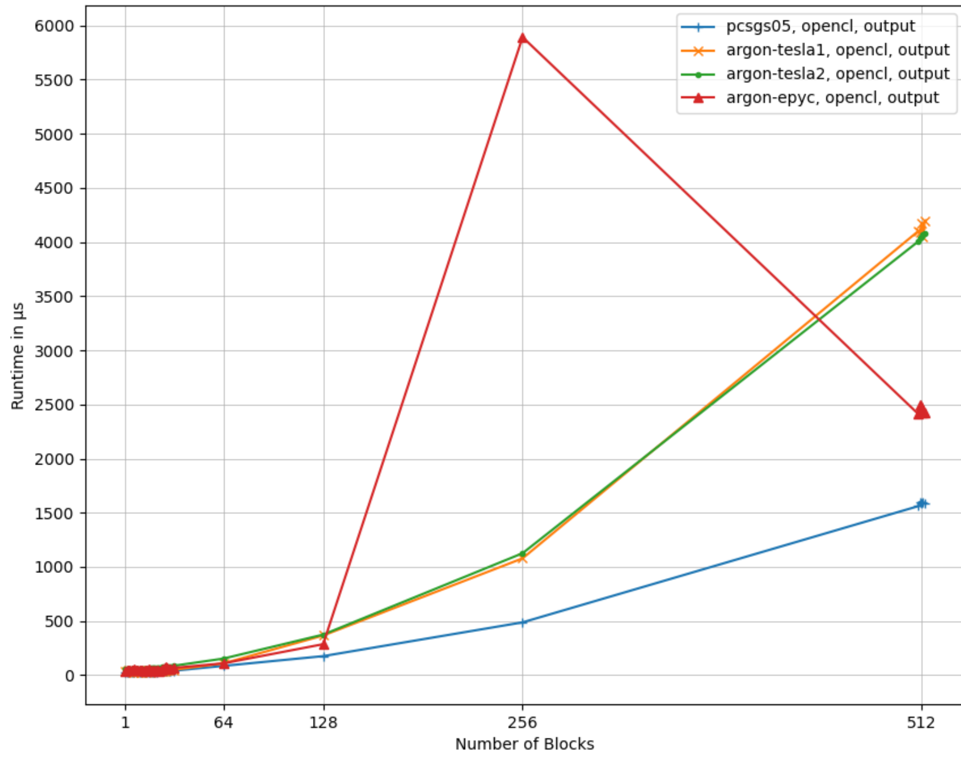

Figure 5.7: Runtime of retrieving result from device.

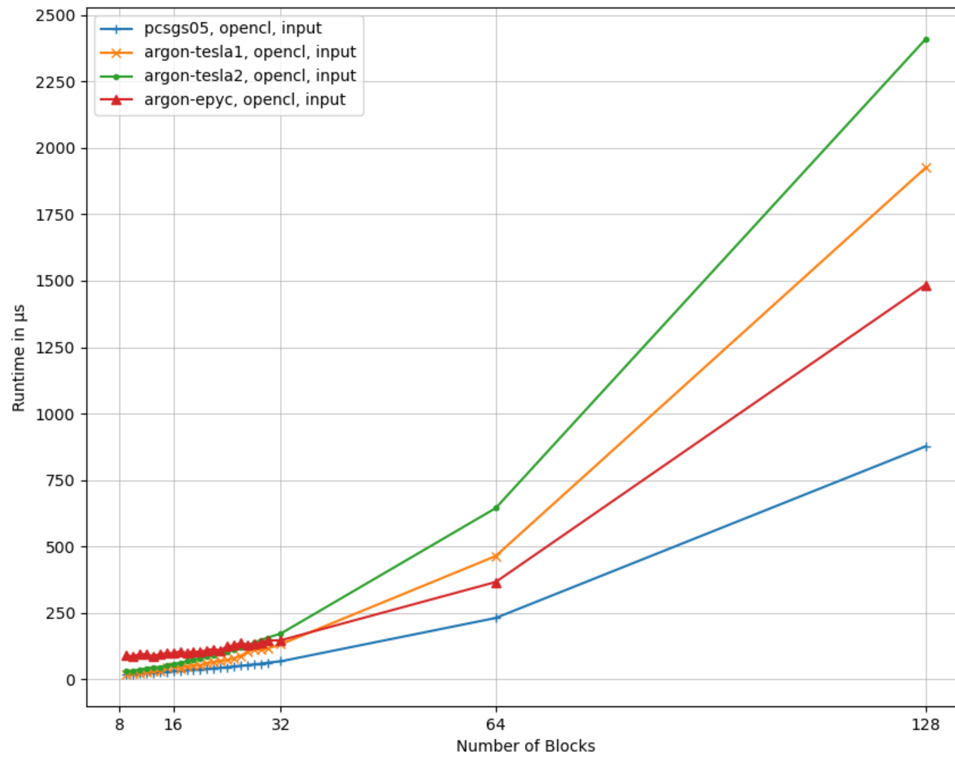

Figure 5.8: Runtime of allocating and populating input buffers on device focusing between 8 and 128 blocks.

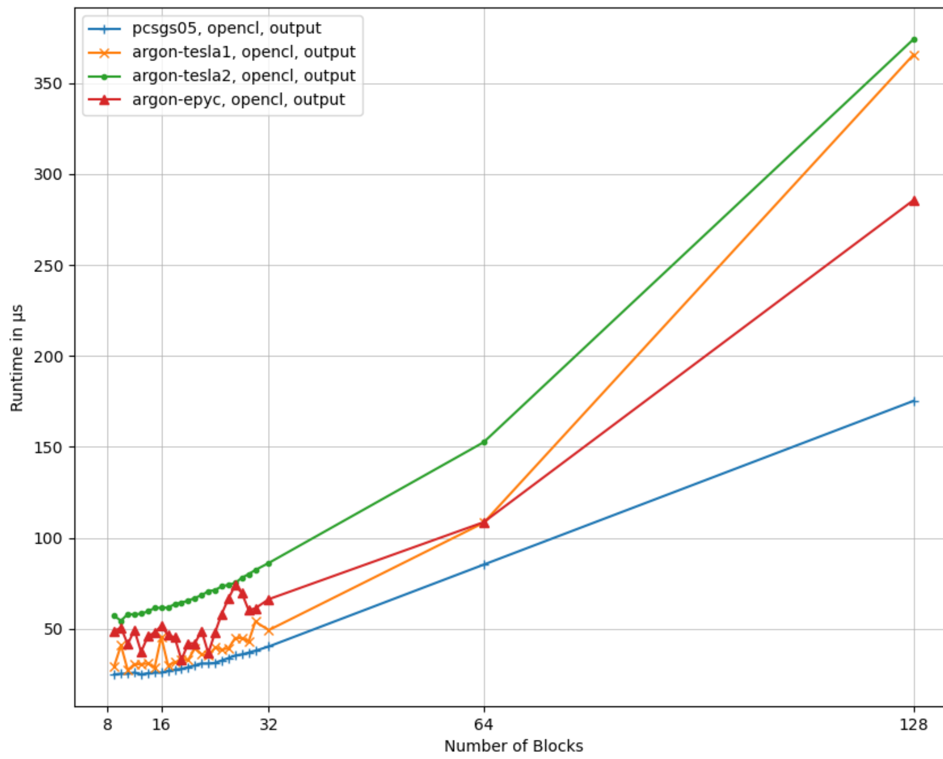

Figure 5.9: Runtime of retrieving buffers from device focusing between 8 and 128 blocks.

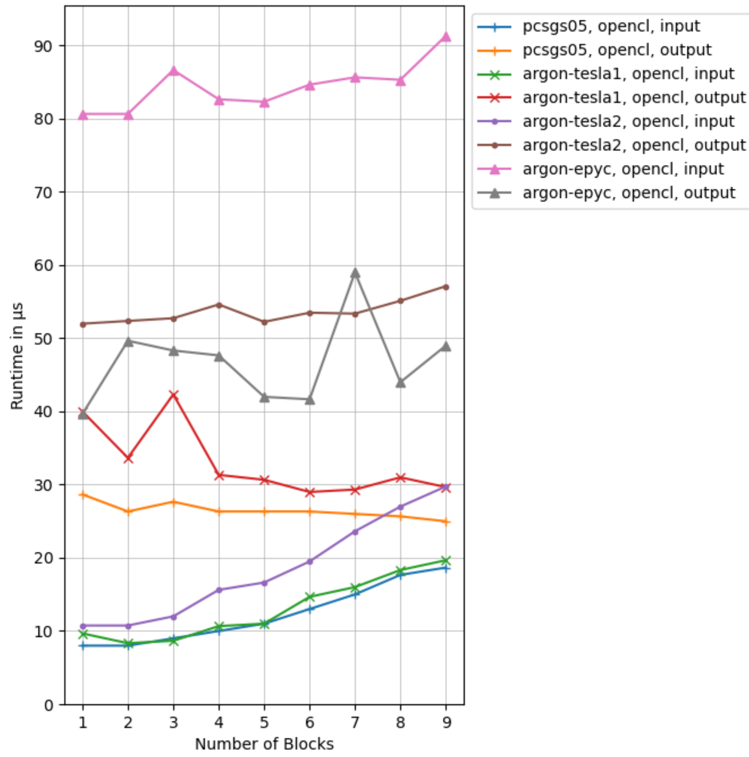

Figure 5.10: Allocation and population of buffers on the device (input) and transferring the result from the device (output) for small data volumes.

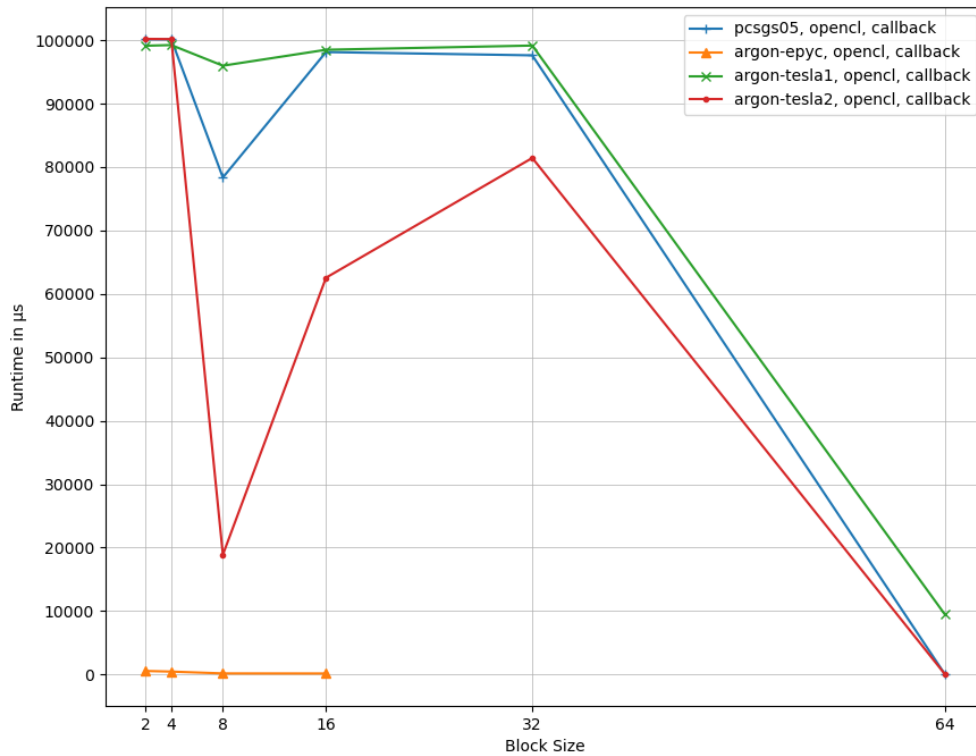

Figure 5.11: Runtime depending on work group size (block size) for callback mode.

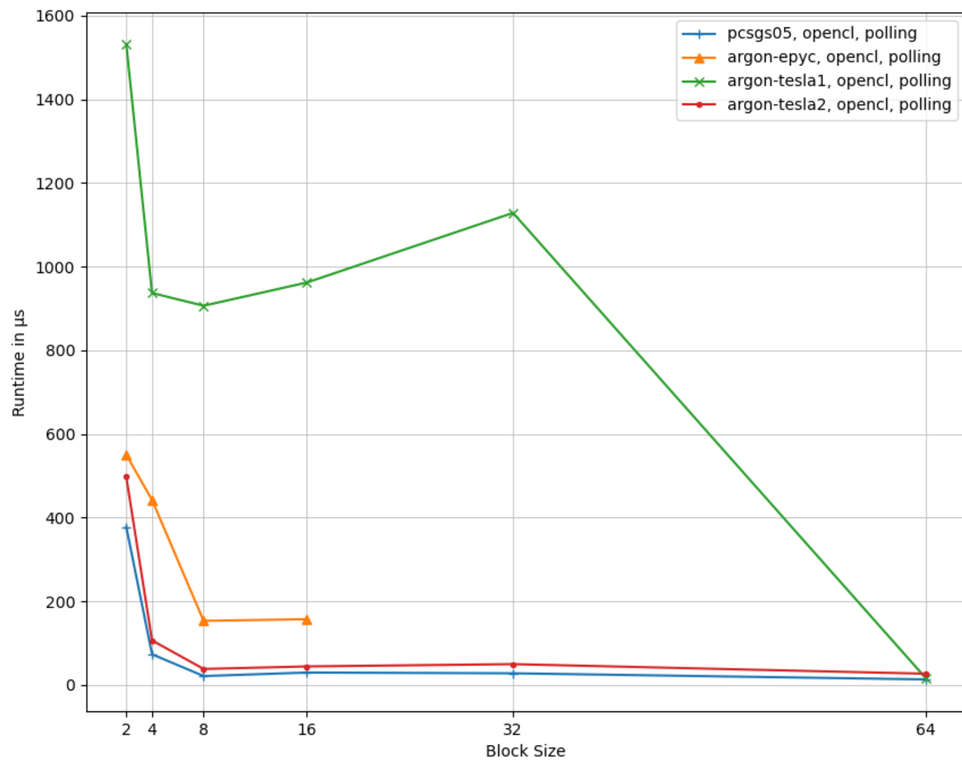

Figure 5.12: Runtime depending on work group size (block size) for polling mode.

6 Conclusion

In this work, we enhanced the capabilities of *HPX* by providing an *OpenCL* executor. Users of the executor can benefit from more available resources on their machines. With the capabilities of *HPX* regarding concurrency and parallelism, this ensures better utilization of resources on all available nodes.

With our detailed description of the implementation and our design decisions, we enable future *HPX* contributors to benefit from our experience. This facilitates the addition of further executors focusing additional libraries.

In particular, we described the process of creating futures using the *HPX* functionality. Moreover, we provided detailed descriptions of the callback and event mode. Finally, we provided guidance on extensions to the *HPX* scheduler in order to enable further polling mechanisms.

The correctness of the executor is ensured via the provided testing cases. All test cases are successful. Therefore, we are confident in the stability of our solution. In particular, we have verified successful execution on different GPUs from various vendors.

Our benchmarks provide results regarding the capabilities of tested GPUs. In particular, we have shown some suspected bottlenecks under suboptimal conditions. By comparing our benchmarks of our custom kernel, we have shown the importance of the computation's optimization.

6.1 Reproducibility

The test results can be verified using the unit tests accompanying the executor. Moreover, Chapter 2 on page 19 outlines the conditions in which the tests have been conducted.

The benchmarks can be verified using the accompanying benchmark test of the *OpenCL* executor. The corresponding data and its visualization can be reproduced using external scripts.

6.2 Outlook

As first step, the implementation will be rebased on the current development branch of the *HPX* repository and contributed to the *HPX* project.

Additionally, *OpenCL* might be not the only executor to enhance the capabilities of *HPX* . Further executors following a similar design are to be explored. Consequently, the possibility of a generic executor is worth exploration. Especially type erasure or variants would be beneficial in the far future in the *HPX* project.

Note that the described callback mode of the executor yields worse results than its polling counterpart. Therefore, the callback based functionality will likely be removed in the future.

A Benchmarking Results

This chapter accompanies the visual representations of the results.

A.1 Matrix Multiplication

A.1.1 Callback

The results of matrix multiplication for various machines and devices for the callback mode is provided in Table A.1.

A.1.2 Polling

The results of matrix multiplication for various machines and devices for the polling mode is provided in Table A.2.

A.2 Data Stream

A.2.1 Input

The results of the data transfer benchmark *to* various devices is provided in Table A.3.

block size	epyc	gtx	pcsgs05 (opencl)	pcsgs05 (cuda)
1	110.38	55.63	38.00	78.96
2	104.00	71.00	38.63	80.00
4	109.63	70.88	39.13	79.77
8	120.50	75.75	42.75	92.06
16	136.88	77.75	4720.88	93.66
32	158.00	62933.50	74655.00	107.89
64	454.88	100154.38	100143.00	111.36
128	657.38	100187.50	100143.75	126.95
256	5013.38	100188.25	100142.88	244.12
512	42089.88	100189.50	100142.88	1132.09
1024	783418.50	700886.14	400516.63	9943.91

Table A.1: Matrix multiplication runtime for increasing block sizes (callback mode)..

block size	epyc	gtx	pcsgs05 (opencl)	pcsgs05 (cuda)
1	120.875	41.00	12.13	11.47
2	120.50	41.88	12.63	11.75
4	125.88	41.38	13.00	11.85
8	126.50	40.50	14.00	13.54
16	135.13	40.88	16.00	13.68
32	156.50	55.75	21.00	19.45
64	452.75	153.88	73.00	21.87
128	664.38	908.50	448.75	38.72
256	5027.88	8179.50	3490.75	157.03
512	42198.13	74032.38	39500.75	1089.71
1024	783350.50	606707.57	383707.88	9853.82

Table A.2: Matrix multiplication runtime for increasing block sizes (callback mode)..

block size	epyc	testla1	tesla2
1	80.67	9.67	8.00
2	80.67	8.33	9.33
4	82.67	10.67	12.67
8	85.33	18.33	19.33
16	98.33	47.67	39.67
32	147.00	132.00	119.33
64	366.33	464.00	439.33
128	1484.67	1925.00	1721.67
256	71224.00	7790.00	6865.00
512	134969.00	36114.00	28287.00

Table A.3: Runtime for allocating and populating buffers on the device. The data volume is measured in block counts.

A.2.2 Output

The results of the data transfer benchmark *from* various devices is provided in Table A.4.

A.3 Blocks

A.3.1 Callback

The impact of block size for various machines and devices for the callback mode is provided in Table A.5.

block size	epyc	testla1	tesla2
1	39.67	40.00	32.67
2	49.67	33.67	33.00
4	47.67	31.33	34.00
8	44.00	31.0	35.67
32	66.33	49.33	72.0
64	108.67	108.33	137.0
128	285.67	365.67	363.0
256	5892.67	1078.67	1117.33
512	2477.00	4177.67	4023.67

Table A.4: Runtime for retrieving data from the device. The data volume is measured in block counts.

block size	epyc	tesla1	tesla2
2 545.00	99136.40	100225.40	
4 447.00	99242.80	100189.80	
8 159.00	95987.80	24877.40	
16 150.00	98499.80	76856.20	
32 —	99163.40	83843.40	
64 —	9467.20	38.80	

Table A.5: Runtime of matrix multiplication using various block sizes (callback mode).

A.3.2 Polling

The impact of block size for various machines and devices for the callback mode is provided in Table A.6.

block size	epyc	tesla1	tesla2
2	550.00	1531.8 490.00	
4	442.00	936.80 97.20	
8	153.00	906.40 30.60	
16	157.00	961.80 36.60	
32	—	1128.40 42.60	
64	—	16.60 18.60	

Table A.6: Runtime of matrix multiplication using various block sizes (polling mode).

B Design Decisions

B.1 Protected Destructor in Base Classes

Generally, type inheritance might introduce challenges for ABI compatibility. Nevertheless, the integration of the *OpenCL* executor encourages the usage of inheritance. Because inheritance is widely used in the project already, we decide to split the public implementation of the executor in `opencl_executor_base` and `opencl_executor`.

In Listing B.1, we have three options regarding the type's destructor.

- We can avoid declaring a destructor. This is the default design choice taken in the implementation of the *CUDA* executor. In this case, the compiler generates a public, default destructor for the type. However, this might lead to a failure to properly destruct the instance. A better design consists of excluding wrong usage of the type.
- We can declare a *virtual* destructor. This would resolve the issue presented in the preceding option. However, the compiler would generate a *vtable* on presence of the virtual destructor. If only the destructor is virtual, this would increase the type's size by `sizeof(void*)`. On a 64 bit platform, the size would be increased by 64 bits.

  If not used, the presence of the *vtable* alone has the same disadvantages as padding. Another disadvantage consists of the type's size. The location of the *vtable* depends on the implementation. Most common implementations place the *vtable* in front of the contained data. This would shift the contained fields by 64 bits. Without the *vtable*, `opencl_executor` already fills a whole cache line. With the *vtable*, the `queue_` would be carried over to the next cache line. The use case of the executor would not trigger other optimizations of the processor, which would compensate the distribution across cache lines. Therefore, lesser performance is expected if a virtual destructor is used.
- We can declare a *protected* destructor. If declared as protected, the destructor would prohibit the undesired use case [SA04]. If the type does not contain any virtual functions, no *vtable* is created. Especially the type's size would not change.

Considering all options, we decide to declare a defaulted, protected destructor in base types.

B.2 Avoiding Singletons

Singleton types are types that allow only one instance thereof. As such, they introduce a global state to the application and have the same disadvantages as globally declared variables. If initialized statically before the application's execution, singletons might lead to the "static initialization fiasco", which oftentimes leads to not deterministic errors.

Listing B.1 Destructor of type `B` is not invoked, because `A:: A()` is not virtual.

```
struct A
{
  ~A()
  {
    std::cout << "~A()\n";
  }
};

struct B final : A
{
  ~B()
  {
    std::cout << "~B()\n";
  }
};

auto main() -> int
{
  A* a = new B();
  delete a;

  return EXIT_SUCCESS;
}
```

Furthermore, the singleton instance is oftentimes accessed in the function directly. Therefore, it has a side effect, which is difficult to test.

Usage of singleton types would render the *OpenCL* executor less maintainable. Therefore, we do not introduce singleton types to the executor.

B.3 Using Exceptions for Error Propagation

HPX heavily relies on the usage of exceptions in the whole project. However, exceptions are one of the features that introduce performance penalties even if not thrown [Sch98]. An alternative consists of proposed implementations for `std::expected`.

However, we are required to interact with core functionality in *HPX* . As the core functionality expects error propagation via exceptions, we align the *OpenCL* executor with the rest of the project.

B.4 Ordering of Fields

Multiple options exists for the field's order in a type.

- If fields are ordered without further consideration, two problems might arise. Firstly, the fields might be loaded on different cache lines in the processor. Therefore, their access would be less optimized. Secondly, the developer can accidentally introduce padding. As padding reduces utilization of memory, this effect is to be avoided.
- Avoiding padding minimizes the type's size. By extension, padding might relate to slightly degraded performance. As conclusion, the developer should avoid padding in the types.
- The second consideration relates to the access frequency of the field. Placing most frequently accessed fields at the beginning of the space occupied by the type benefits the performance.

As first priority, we strive to avoid padding in our types. Secondly, we consider to order fields based on their access frequency, if possible.

B.5 Visibility of Declarations

The documentation included in the *HPX* repository suggests that header files can be put into the `src/` directory of the *OpenCL* executor. However, this rule is not used anywhere in the project. Therefore, we decide to place all header files in the `include/` directory. Therefore, even implementation meant to be used for the executor internally is accessible to the whole project.

Nevertheless, we limit the extent of accessible internals using unnamed namespaces where possible. This way we declare contained types and functionality effectively as `static`. Firstly, we limit the accessibility of such types as intended. Secondly, we provide the compiler with enhanced optimization options with this design decision.

C Machine Characteristics

In this chapter, we provide some detailed information regarding the used *OpenCL* devices for benchmarking.

C.1 GeForce RTX 3080 GPU

The device information related to the GeForce RTX 3080 GPU is provided in Table C.1.

C.2 AMD EPYC 7551P GPU

The device information related to the AMD EPYC 7551P GPU is provided in Table C.2.

C.3 GeForce GTX 1080 Ti GPU

The device information related to the GeForce GTX 1080 Ti GPU is provided in Table C.3.

Number of platforms	1
Platform Name	NVIDIA CUDA
Platform Vendor	NVIDIA Corporation
Platform Version	OpenCL 1.2 CUDA 11.2.162
Platform Profile	FULL_PROFILE
Platform Name	NVIDIA CUDA
Number of devices	1
Device Name	GeForce RTX 3080
Device Vendor	NVIDIA Corporation
Device Vendor ID	0x10de
Device Version	OpenCL 1.2 CUDA
Device OpenCL C Version	OpenCL C 1.2
Device Type	GPU
Device Profile	FULL_PROFILE
Max compute units	68
Max clock frequency	1800MHz
Compute Capability (NV)	8.6
Max work item sizes	1024x1024x64
Max work group size	1024
Preferred work group size multiple (kernel)	32
Global memory size	10501554176 (9.78GiB)
Max memory allocation	2625388544 (2.445GiB)
Global Memory cache line size	128 bytes
Local memory size	49152 (48KiB)
Max constant buffer size	65536 (64KiB)
Max size of kernel argument	4352 (4.25KiB)
Profiling timer resolution	1000ns
Execution capabilities	
Run OpenCL kernels	Yes
Run native kernels	No
Kernel execution timeout (NV)	Yes
Concurrent copy and kernel execution (NV)	Yes
Number of async copy engines	2

Table C.1: *OpenCL* device used on machine `pcsgs05`.

Number of platforms:	1
Platform Profile:	FULL_PROFILE
Platform Version:	OpenCL 2.0 AMD-APP (3186.0)
Platform Name:	AMD Accelerated Parallel Processing
Platform Vendor:	Advanced Micro Devices, Inc.
Platform Name:	AMD Accelerated Parallel Processing
Number of devices:	1
Device Type:	CL_DEVICE_TYPE_GPU
Max compute units:	60
Max work items dimensions:	3
Max work items[0]:	1024
Max work items[1]:	1024
Max work items[2]:	1024
Max work group size:	256
Max clock frequency:	1801Mhz
Max size of kernel argument:	1024
Cache line size:	64
Cache size:	16384
Global memory size:	17163091968
Constant buffer size:	14588628172
Max global variable size:	14588628172
Max global variable preferred total size:	17163091968
Max read/write image args:	64
Max on device events:	1024
Queue on device max size:	8388608
Kernel Preferred work group size multiple:	64

Table C.2: *OpenCL* device used on machine `argon-epyc`.

Number of platforms	1
Platform Name	NVIDIA CUDA
Platform Vendor	NVIDIA Corporation
Platform Version	OpenCL 1.2 CUDA 11.2.162
Platform Profile	FULL_PROFILE
Number of devices	8
Device Name	GeForce GTX 1080 Ti
Device Vendor	NVIDIA Corporation
Device Version	OpenCL 1.2 CUDA
Device OpenCL C Version	OpenCL C 1.2
Device Type	GPU
Device Profile	FULL_PROFILE
Max compute units	28
Max clock frequency	1582MHz
Max work item dimensions	3
Max work item sizes	1024x1024x64
Max work group size	1024
Preferred work group size multiple (kernel)	32
Global memory size	11721506816 (10.92GiB)
Max memory allocation	2930376704 (2.729GiB)
Global Memory cache size	1376256 (1.312MiB)
Global Memory cache line size	128 bytes
Max number of constant args	9
Max constant buffer size	65536 (64KiB)
Max size of kernel argument	4352 (4.25KiB)
Profiling timer resolution	1000ns
Execution capabilities	
Run OpenCL kernels	Yes
Run native kernels	No
Kernel execution timeout (NV)	No

Table C.3: Device used on machine `argon-gtx`.

Bibliography

[AM04] A. Alexandrescu, M. Michael. “Lock-free data structures with hazard pointers”. In: *C++ User Journal* (2004), pp. 17–20 (cit. on p. 22).

[BKK+19] M. Bremer, K. Kazhyken, H. Kaiser, C. Michoski, C. Dawson. “Performance comparison of HPX versus traditional parallelization strategies for the discontinuous Galerkin method”. In: *Journal of Scientific Computing* 80.2 (2019), pp. 878–902 (cit. on p. 17).

[Dan15] Daniel Bourgeois. *HPX and C++ Executors*. https://stellar-group.org/2015/05/hpx-and-cpp-executors/, last accessed on 17.06.2021. 2015 (cit. on p. 17).

[GFB+04] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. “Open MPI: Goals, concept, and design of a next generation MPI implementation”. In: *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer. 2004, pp. 97–104 (cit. on p. 17).

[GGL+99] W. Gropp, W. D. Gropp, E. Lusk, A. Skjellum, A. D. F. E. E. Lusk. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999 (cit. on p. 17).

[HLGW19] T.-W. Huang, C.-X. Lin, G. Guo, M. Wong. “Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 974–983. DOI: 10.1109/IPDPS.2019.00105 (cit. on p. 17).

[KDL+20] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, T. Zhang. “HPX - The C++ Standard Library for Parallelism and Concurrency”. In: *Journal of Open Source Software* 5.53 (2020), p. 2352. DOI: 10.21105/joss.02352. URL: https://doi.org/10.21105/joss.02352 (cit. on p. 15).

[KTK+17] Z. Khatami, L. Troska, H. Kaiser, J. Ramanujam, A. Serio. “Hpx smart executors”. In: *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*. 2017, pp. 1–8 (cit. on p. 17).

[SA04] H. Sutter, A. Alexandrescu. *C++ coding standards: 101 rules, guidelines, and best practices*. Pearson Education, 2004 (cit. on p. 71).

[Sch98] J. L. Schilling. “Optimizing away C++ exception handling”. In: *ACM SIGPLAN Notices* 33.8 (1998), pp. 40–47 (cit. on p. 72).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature