ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Master Thesis
at the Department of Information Technology
and Electrical Engineering

Executing Process Networks on Heterogeneous Platforms using OpenCL

AS 2012

Tobias Scherer

Advisors: Lars Schor
Andreas Tretter
Professor: Prof. Dr. Lothar Thiele

Zurich
31st May 2013

Abstract

Upcoming heterogeneous systems ask for new programming paradigms. Abstracting the underlying hardware architecture is desirable in order to support productive software development.

This thesis proposes a design flow and runtime-system for executing process networks on heterogeneous systems using OpenCL. Process networks are a popular model of computation for deterministic parallel programming and OpenCL is a royalty-free standardised programming interface with a broad support in industry. The proposed design flow consists of a program code synthesis framework for building applications from a generic high-level process network specification. The synthesised application is targeted to a certain OpenCL architecture that was predefined by a high-level specification. The target code is built for this architecture specification integrating reusable building block primitives into it. Those primitives are location-based FIFO channels minimising the number of memory copy operations, a process wrapper that is interconnectable to channels and mirrors the process network functionality, and an extensible task activation framework responsible for interprocess synchronisation. Heterogeneous systems, being inherently parallel computer architectures, demand scalable and parallel applications. To simplify this, a notion of shadow copies was introduced to transparently abstract data-parallelism.

Extensive evaluations on two heterogeneous systems have shown that the proposed design flow and runtime-system support a wide range of heterogeneous platforms and parallel applications. Furthermore, the evaluations have proved that the proposed framework is suitable for efficient and productive software development for heterogeneous systems and that it provides enough flexibility so that the programmer can efficiently exploit the parallelism offered by multicore CPUs and GPUs.

Acknowledgements

First of all I would like to express my sincere gratitude to Prof. Dr. Lothar Thiele for giving me the opportunity to write this master thesis in his research group.

I wish to express my warm and sincere thanks to my advisors Andreas Tretter and Lars Schor for their many enriching discussions and their extensive support during the thesis. The many hours of discussing results and proceedings were really helpful and motivating. I also appreciated that your door was always open for me to discuss ideas and problems. It was a pleasure to work with you and also to contribute to your future research.

Furthermore I would like to thank my family, my friends and also my girlfriend Tanja for their constructive motivation and patience during this thesis.

Contents

List of Figures

1
Introduction

-

1.1. Motivation

Today, we are in the middle of a fundamental transition concerning computer architectures. It is driven by an ever-increasing demand for computing power in all fields, be it High Performance Computing, personal computers, or even embedded systems. All of them asks for power-efficient, but still powerful computer architectures. Heterogeneous system are one way to address both objectives, namely being power-efficient and offering an increased performance at the same time. The clue of a heterogeneous system is that it combines different types of computing units to a complete system. Computing units can, for example, be special purpose hardware suitable only for one particular class of problems. Specialisation allows computing units to be optimised in terms of power-efficiency or performance. Typically, these different types of computing units have to be programmed separately, which makes writing software for such a system tedious and complex. Applications have to be manually adjusted and optimised for a particular system and cannot be easily ported to another one.

Traditional programming models are not convenient to program these new heterogeneous systems because they lack in flexibility and lead to laborious, time-consuming, and error-prone software development. OpenCL — a standardised interface for cross-platform, parallel programming with a broad support in industry — is one step towards portability. With OpenCL, a piece of code can be ported to a broad range of different computer architectures. This simplifies software development and makes the code reusable for different platforms, which is a persuasive argument for using OpenCL. Unfortunately, OpenCL is a complex framework by itself, and needs a lot of knowledge and understanding of low-level details of the underlying hardware. There are various difficulties that are unsolved in OpenCL. For example, communication is a tedious task, especially when devices of different vendors are combined to a heterogeneous system. Also synchronisation is left to the user, even though it leads to indeterministic software if it is done improperly.

The problems arising with OpenCL are not new and have already been solved for traditional programming models with the model of process networks. Process networks offer a data-driven notion to write inherently parallel software. Synchronisation is achieved implicitly and the behaviour of an application will be correct for any deadlock-free schedule. For process networks it is possible to calculate correctness tests offline. It can be guaranteed that an application will never block and therefore behaves correctly. Furthermore, an implementation of process networks always consists of the same elementary parts, which can therefore be reused for multiple applications. The programmer can solely focus on the functionality of the application and need not care about low-level details, such as communication and synchronisation.

This master thesis proposes a design flow and a runtime-system that implements process networks by means of OpenCL. In this framework, applications are specified in a high-level language following the ease of the process network syntax, whereas the underlying architecture is specified separately and independently from the one of the application. This is the baseline for a program synthesis framework that creates architecture-dependent code according to the specification and builds the final executable by merging the created code with the the application-specific code. This approach offers a flexible and productive way of portable software development. The program synthesis framework combines reusable building blocks that were implemented and tested carefully to be flexible and adaptive for different process to processor mappings. An extensive evaluation has been carried out, comparing different OpenCL devices for their parallel processing performance on a realistic video-processing application.

1.2. Contributions

The contributions of this master thesis are as follows:

- An application programming interface (API) for specifying an application based on the notion of a process network and a separate high-level specification for heterogeneous architectures.
- A notion of parallelism that distinguishes between coarse-grained and fine-grained data-parallelism. The former is denoted as shadow copy, whereas the latter is denoted as intra-process parallelism.
- A runtime-system implementing the proposed API for executing process networks on top of OpenCL. This includes location-based FIFO channels that minimise the number of copy operations considering the underlying memory architecture and an extensible task activation framework that predictably controls the process invocations and synchronisation among the processes.
- A program code synthesis framework, which can create executable binaries from a high-level application and architecture specification. The proposed framework can build the same application for all the specified architectures by merging the architecture specific code with the application specific code.
- An extensive evaluation of the framework benchmarking the channel throughput, process overhead, performance of the task activation framework, and real-world performance of various devices for different levels of parallelism.

1.3. Outline

This thesis is organised as follows. After investigating related work in Chapter 2, an overview of existing parallel programming models is given in Chapter 3 and a problem statement is formulated. The proposed runtime environment is detailed in Chapter 4, followed by the design flow specification in Chapter 5. Chapter 6 presents the results of the evaluation of the proposed framework. The thesis finally concludes in Chapter 7 presenting an outlook.

2
Related Work

The recent arise of high-performance graphic processing units (GPUs) motivated many research projects to exploit heterogeneous computer systems consisting of central processing units (CPUs) and GPUs. To support this evolution, programming languages have been developed that natively support GPUs. Examples of such programming languages include Cg [2], Brook [3], and Accelerator [4]. Today, Nvidia's CUDA [5] and OpenCL [6] are mainly dominating general-purpose GPU programming. While CUDA is a proprietary framework targeting the GPUs Nvidia produces, OpenCL is maintained by a non-profit technology consortium and adapted by many software and hardware vendors. Besides GPUs, programs written in OpenCL can be executed on CPUs, various accelerators, or DSPs. Even more, OpenCL can be used to program the STHorm platform [7], a cluster-based on-chip many-core system.

Even though OpenCL provides a standard interface to program heterogenous systems, the programmer must manage many low-level details including mapping of tasks onto devices, data transfer between host and device, and synchronisation between the different devices. Thus, many projects recently developed abstraction layers for OpenCL. Maestro [8] is an extension of OpenCL providing automatic data transfers between host and device as well as task decomposition across multiple devices. While tremendously sim-

plifying the task of the programmer, Maestro introduces new restrictions as, for instance, that the individual tasks have to be independent of each other. The task-level scheduling framework detailed in [9] extends OpenCL by a task queue enabling a task to be executed on any device in the system. Furthermore, dependencies between tasks are resolved by manually specifying a list of tasks that have to be completed before a new task is executed. Nonetheless, the burden task of data exchange is still left to the programmer and no automatic design-flow is provided to efficiently design applications in a high-level programming language.

dOpenCL (Distributed OpenCL) [10] is an extension of OpenCL to program distributed heterogenous systems. Even though the approach abstracts the different nodes of a distributed system into a single node, the programmer is still responsible for managing many low-level details of OpenCL.

A widely used approach for multi-core programming at the process-level are process networks. In particular, GPUs have recently been considered as an option to improve the execution of process networks on ordinary computing systems [11, 12, 13]. For instance, the multi-threaded framework proposed in [11] integrates both POSIX threads and CUDA into a single application. KPN2GPU, a tool to produce fine-grain data parallel CUDA kernels from a process network specification, is described in [12]. An automatic code synthesis framework taking process networks as input and generating multi-threaded CUDA code is described in [13]. Sponge [14] is a compiler to generate CUDA code from the StreamIt [15] programming model. All of them have in common that they map streaming applications specified as process networks onto heterogenous systems. In contrast, the approach in this thesis generates OpenCL code enabling the same framework to be used for a wider range of heterogeneous platforms.

The high-level compiler described in [16] generates OpenCL code for applications specified in Lime [17], a high-level Java compatible language to describe streaming applications. The Lime task programming model is similar to SDF graphs, but provides support for non-determinism by introducing special operators. The work in [16] mainly focuses on optimizing the individual OpenCL code. In contrast, this thesis follows the Y-chart design approach [18] enabling a platform-independent specification of the application and an automatic high-level optimization of the process-to-device mapping to efficiently utilize the system. Furthermore, by extending the DAL programming model [19], the programmer is able to specify dynamic interactions between multiple streaming applications.

3

Programming Model and Problem Statement

This chapter presents parallel programming concepts. To start with, the difficulties of writing parallel software are highlighted in the introductory section. Afterwards, the high-level parallel programming model of process networks is introduced in Section 3.1.2. Process networks have some nice characteristics that simplify writing parallel applications. Then, motivated by upcoming heterogenous systems, existing programming frameworks for general purpose GPU-computing (GPGPU) are summarised in Section 3.2. The focus is afterwards set to OpenCL, a framework that allows to program various types of computing units in a portable way. Finally, a problem statement is formulated in Section 3.4 to build the baseline of this thesis.

3.1. High-Level Parallel Programming

Parallel computing is a fundamental concept of programming that stands in contrast to sequential programming. It describes a concept where multiple calculations are executed simultaneously. The roots of parallel programming are found in High Performance Computing, but it has recently experienced a new wave of interest in computer engineering research. The reason for this

is on one the hand the physical limit for frequency scaling in digital circuits, which led to multicore CPUs in personal computers. On the other hand is an increasing demand for efficient low-power devices with a huge computation performance in the fields of embedded systems. Thus, embedded systems are also evolving towards highly parallel computer architectures with heterogeneous components.

This evolution concerning parallel computer architectures asks for new appropriate parallel programming models, and is a major change in software development. As the evolution of such mainstream parallel systems is still young, experts consider the currently existing high-level programming models as still not mature. For example Dr. Clay Breshears — Technical Advisor to the Intel Software Network Parallel Programming Community, and author of the Book *The Art of Concurrency* [20] — compares the evolution of programming models with the for loop that took decades to be developed in its modern syntax. He also states:

> *"I believe that we're still in the infancy of multicore processors and widespread parallel programming models. [...] If we consider MPI and threads as the assembly languages of parallelism, then OpenMP and TPL are the first passes of implementing parallelism in a much easier, higher-level way. Having parallelism as an integral part of the definition of a programming language is the next step in the evolution that will make things easier.* [21] *"*

There are quite a few difficulties with parallel programming that make writing software all but straight-forward. Deterministic behaviour is not easily reached as there are race conditions between concurrent processes that share the same data. Synchronisation of critical sections is therefore indispensable. However, if synchronisation is done improperly, it will unnecessarily serialise the application, or even worse, it will lead to dead-locks. Synchronisation errors can usually not be detected by compilers, and remain undetected until the application behaves incorrectly. Debugging is a prominent way to track errors, but a debugger usually also influences the timing behaviour, and it might happen that the same program runs glitch-free while debugging. But even without such special problems, it is a challenging task to debug multi-threaded programs, because it is not exactly easy to understand the behaviour of simultaneously running threads.

Another important question is how to exploit the full performance of an underlying hardware architecture. It is dependent on the correct schedule of the

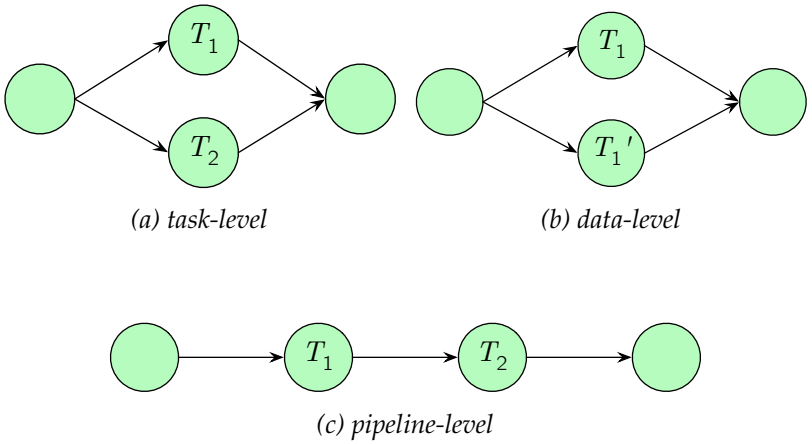
concurrent processes, whereas the schedule itself is dependent on the process-to-processor mapping. Usually, processes are not completely autonomous, because they communicate with each other. In a real system, communication cost cannot be neglected, as the resources are sparse (e.g., limited bandwidth on a bus), and also because communication introduces latencies. In practice, communication costs do not scale proportionally with the amount of data that has to be transferred (it is often cheaper to transfer a bigger amount of data once in comparison to transferring the same amount of data in small pieces). For all these reasons, finding an optimal schedule and an optimal mapping for such a system is non-trivial.

3.1.1. Classification of Parallelism

The parallelism of an application can be classified into three major groups, namely task-level, data-level, and pipeline-level parallelism. They are depicted in Figure 3.1. Task-level parallelism is shown in Figure 3.1-(a) by a dependency graph of four tasks. Tasks T_1 and T_2 are independent of each others outputs, and can therefore be executed simultaneously. In contrast, there is data-level parallelism depicted in Figure 3.1-(b). The topology of the dependency graph is equal to the one of task-parallelism, but the difference is that both tasks T_1 and T_1' consist of the same instructions, but are applied to different data. The third class is pipeline-parallelism shown in Figure 3.1-(c) where T_2 is directly dependent on T_1. In this case, the tasks can be seen as an assembly-line, where each stage produces data to be forwarded to the next stage. Optimally, these processes can run concurrently.

Another classification is the parallelism granularity of an application. It describes the ratio of computation to communication according to the definition in [1, p. 10]. An application that is split into many small processes with a lot of communication is called fine-grained, whereas the same application is called coarse-grained, if it is split into fewer long-running processes with less communication overhead.

This section provided an overview of parallel programming. The next section will detail a formalised distributed parallel programming model called *process network*.



(a) task-level *(b) data-level*

(c) pipeline-level

Figure 3.1.: The three different classifications of parallelism depicted as dependency graphs between tasks.

3.1.2. Process Networks

Process networks are a well-known theoretical concept for describing dataflow applications. In principle, an application is split into several autonomous processes. Each process solves exactly one specific subtask and runs concurrently with all other processes. The only relation between them is the data they exchange using point-to-point first-in first-out (FIFO) channels; this is also the only way to communicate between processes. A process network can be denoted as a directed graph (Fig. 3.2), where the nodes represent processes and the arcs represent channels. Channels are theoretically unlimited in size, and a process can read from its input channels or write to its output channels. The read operation will block the process in case that the channel is empty. The execution time of a process can be arbitrarily long, but eventually when it finishes, it will be automatically invoked again. A process might be stateful, i.e., it can store data between to invocations.

An application specified as a process network shows the amount of concurrency between its processes. Especially task-level and pipeline-level parallelism are naturally represented by the process network itself. A very neat feature of process networks is that, no matter what scheduling is applied to them, the functionality will be equivalent. This is usually not the case when writing parallel software without proper synchronisation. Process networks however have a really simple notion of synchronisation that will always guarantee correct behaviour.

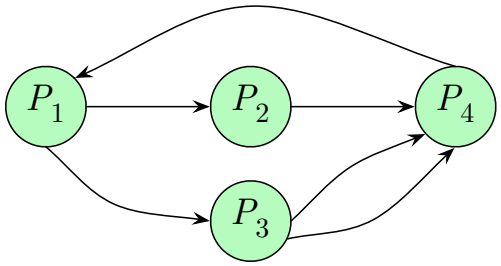


Figure 3.2.: A process network consisting of four processes denoted as a directed graph. The nodes represent the processes, and the arcs the interconnecting channels.

The next section introduces a particular notion of process networks, namely *synchronous data flow*.

Synchronous Data Flow

The notion of *Synchronous Data Flow (SDF)* was introduced in 1987 in [22] by Lee et. al. and specifies a subclass of process networks. Their work was largely influenced by the inherent data-centric view of signal processing applications. Signal processing applications are defined as independent processes interconnected through channels, where each process is solving a subtask of the complete application. This way, the application can also be depicted as a data flow graph (Fig. 3.2), where the processes are represented by nodes, whereas the channels correspond to the interconnecting arcs. The processes are invoked continuously and infinitely often in a self-loop.

In contrast to general process networks, a process in SDF is stateless. This means, that the behaviour of a process only depends on the input tokens and not on the past invocations. A state can however be emulated for SDF by using a self-channel which transfers the state into an input token. Another difference is that every process writes a constant amount of data to all its output channels in each invocation and reads a constant amount of data from its input channels. The amount of data can be accessed in terms of tokens, describing a logically indivisible amount of data. The size of such a token is fixed per channel. So, the rate of tokens for every channel is constant for all invocations of a process and specified before execution. However, the input token rate and the output token rate of a channel can be different on every channel.

In the original paper, each channel could be assigned a delay, which is equivalently treated as a certain number of initial tokens lying on the delayed channels.

Properties of SDF graphs

One of the main reasons for introducing the SDF model of computation was exposing task-level and pipeline parallelism in an application. This is non-trivial in imperative coding, and is therefore usually not exhibited. SDF graphs, on the other hand, do naturally represent an application such that task-level and pipeline-level parallelism can easily be achieved. Moreover, since the token rates are constant for the whole runtime, it is possible to calculate a schedule offline, which is not possible for general-purpose process networks. Another property is that SDF applications can be analysed for correctness, i.e. it is possible to calculate the needed buffer sizes for a given schedule such that the application will never dead-lock. Yet another neat property is that the behaviour of an application is always correct for any schedule that is not dead-lock free. SDF graphs do therefore not suffer from race-conditions and dead-locks, which are serious problems of other common parallel programming models. Furthermore, synchronisation is an inherent property of SDF, and is not left to the programmer. Synchronisation and indeterminism are what makes parallel programming error-prone and time-consuming. Both are eliminated for process networks, which is a big advantage.

On the other hand, the SDF model of computation does also have some disadvantages. First, the static token rate restricts the space of applications, because all processes have to process the same amount of tokens in each invocation. This is not the case for more general process networks. Second, there is a tradeoff to chose between fine-grained and coarse-grained parallelism of an application. Fine-grained parallelism will introduce a considerable communication overhead and finally limit the application speedup. More coarse-grained processes however, will unnecessarily serialise the application and will therefore decrease the flexibility of mapping processes to other computing elements. Third, the mapping and the schedule are tied to one specific platform and cannot be simply ported to another one.

SDF is a theoretical formalism, which defines rules how an application should behave. It describes *what* an application should do with a completely data-driven model. Therefore SDF can be classified as declarative coding, because SDF does only define the behaviour of an application and not *how* this be-

haviour can be achieved. A concrete implementation of the SDF model of computation is proposed in Chapter 5.

The next section provides a bottom-up view on parallel programming by introducing heterogeneous platforms and their programming models.

3.2. Heterogeneous Platforms

Real-world applications usually consist of both, a parallel part, and a sequential part that cannot be further divided. Future hardware systems should reflect that structure to support a broad range of applications to run quick and power-efficient. Systems integrating different types of computing units are called heterogeneous. Their components, so-called computing units, have different capabilities that can be used to accelerate some special tasks. Since heterogeneous platforms are offering multiple computing units that can work concurrently and programmed independently, they are classified as a subset of parallel systems.

Recently, the chip vendors were investing a lot of their research resources in developing new heterogeneous architectures. For example, Intel — specialised in selling general-purpose CPUs — released its new massively parallel accelerator processor *Xeon Phi*[1] targeting the market of High-Performance Computing. It is a coprocessor that can accelerate special tasks and therefore relieve the CPU. Such a system consisting of one or more CPUs and a Xeon Phi is heterogeneous, as both computing units can be programmed independently and operate simultaneously. Another example of a heterogenous system is NVIDIA's Tegra[2] processor, an ARM system-on-chip which integrates a multicore CPU, mobile GPU, and video accelerators on the same chip. Its ultra-low-power design is designed to be used in mobile phones and tablet computers, while still offering a good performance. AMD is focusing on integrating GPUs and CPUs to a compound chip called *Accelerated Processing Unit* (APU) mainly targeting the market for power-efficient devices, such as Laptops, Netbooks, but also budget PCs. All of these examples show that the classical fields of CPU and GPU circuitry are gradually merging. Today, shared-memory multi-core processors have been established very well, but the number of cores cannot be scaled endlessly. The new era of heterogeneous computer architectures has yet just begun.

[1] `www.intel.com/xeonphi`

[2] `www.nvidia.com/object/tegra.html`

The concept of a heterogeneous computer architecture is not completely new. The best example of such an existing architecture is the PC. It contains a bunch of CPU cores, and usually also one or more separate graphics cards. The GPU is a device that is designed for solving massively data-parallel fine-grained problems, basically graphics calculations. Originally, the GPU was designed as a fixed-function special purpose processor that was only capable of rendering computer graphics. However, the fixed-functionality turned out to be so much limiting that it did not even allow to change the lighting or shading model. This implicated that it was impossible to render more complex sceneries or to apply some special effects. As a consequence, the GPU architecture advanced from this fixed-function processor towards a powerful programmable processor. This evolution was the basis for GPGPU computing. GPGPU stands for *general purpose graphics processing unit*. It describes a whole class of programming models that allow the GPU to be used as a compute unit to calculate common algorithms that are heavily parallelisable.

A very good overview of his evolution can be found in [23] and is summarised in the following paragraph. One of the first considerable approaches of running normal code on GPUs was the high-level interface called Cg that was presented by NVIDIA [2]. But as a mismatch, the computations still had to be defined as a graphical problem which does not exactly help to write understandable code for general programs. There was no real abstraction of the hardware architecture of graphics cards. Another approach was taken by researchers at Stanford when they proposed Brook [3], a high-level language for defining streaming applications that can be executed on the GPU. One of the main problems in these days was the lack of the scatter operation for GPUs. Usually GPUs output the calculated images to the screen and it was not foreseen to store the data, i.e. scatter, as it is needed for general programs. Therefore the inventors of Brook had to find a way to copy data back to the main memory which was a really important contribution. The first commercial approaches were RapidMind, PeakStream, and Microsoft Accelerator. They offered just-in-time compilation for the functions running on the GPU, which was an essential step towards portability.

Today, there are mainly three big frameworks for GPGPU-computing, namely OpenCL, CUDA, and DirectCompute. All three of them build an abstraction layer on top of the hardware. However, in order to have a flexible and performant framework, the user must understand many low-level details of the hardware such as memory access patterns and register usage. These GPGPU frameworks shall be introduced in the following section.

3.2.1. Comparison of Existing Languages for GPGPU-Programming

OpenCL is a standardised programming interface that is managed by the Khronos Group [6]. It is a cross-platform framework for parallel programming of modern computer architectures. The interface is implemented by the hardware vendors and the resulting implementations are afterwards certified by Khronos, a non-profit technology consortium.

CUDA is a proprietary framework by NVIDIA. It can be used to program NVIDIA-branded graphics cards. It is well-accepted in industries, mainly because it also supplies optimised libraries for common algorithms such as FFT, linear algebra, or random number generators, among others[3].

Microsoft DirectCompute is an application programming interface that is integrated into DirectX 11 and can be used to program compatible graphics cards. However, it can only be used with Microsoft Windows.

In this work, OpenCL is used as a parallel programming framework. Its cross-platform portability allows to program a broad range of different devices, which is not the case for CUDA and DirectCompute. OpenCL, being a royalty-free standard, will hopefully become a widely accepted parallel programming framework in the future. But even today, all big vendors already provide OpenCL support for their chips. The next section will provide an overview of OpenCL and introduce the incorporated terminologies.

3.3. Computational Model OpenCL

As mentioned before, OpenCL is a standardised cross-platform programming interface for parallel programming. The reason for introducing OpenCL was mainly the usage of graphics cards (GPU) as general purpose processors. A GPU, being a device that has a tremendous amount of parallel processing elements, is perfectly suitable for data-parallel algorithms. However, OpenCL offers portability, allowing the support of other devices, such as CPUs and special purpose accelerators.

[3]CUDA toolkit: `https://developer.nvidia.com/cuda-toolkit`

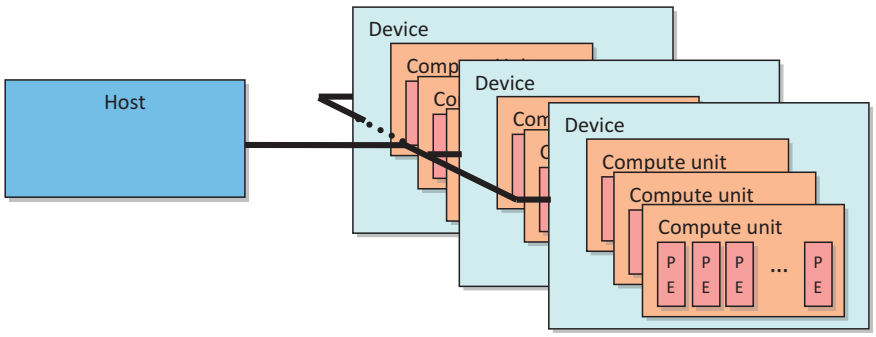


Figure 3.3.: OpenCL system, taken from [1].

3.3.1. Terminology

OpenCL introduces a number of terms that are needed to fully understand the concepts presented in this thesis. These terms shall be introduced in this section.

To begin with, an OpenCL application consists of two parts. The first one is the so-called **host** that is responsible to distribute work for the second part, the **device**. This relation is depicted in Figure 3.3. For every OpenCL system, there is exactly one host that controls all devices. The host code can be written in various different languages and describes the behaviour of the application[4]. The device code is written in a special C99-like language named *OpenCL C* and is called **kernel**. A device can be any kind of processor, be it a GPU, a CPU, or some other kind of accelerator. The kernel is portable, which means that it can run on any OpenCL compliant device. This is achieved by compiling the kernel source file at runtime. Kernels have a well-defined interface and implement a special function in a data-parallel fashion. A kernel is executed on a device and is a bounded function that will finish within a conceivable amount of time.

A **platform** is an implementation of the OpenCL interface by a vendor. OpenCL has an extensible interface to integrate multiple platforms on the same operating system. The different implementations are loaded as shared libraries and are compatible to implementations of other vendors. A platform can contain support for multiple devices of the same vendor. It is possible to create a **context** from any subset of devices of such a platform. A context is

[4]The original syntax of the OpenCL interface is in C, but there are bindings for many modern programming languages

an abstract model that transparently combines the different memory spaces of its devices to a single memory space. Data management is, in this case, handled by the OpenCL platform driver. However, it is also possible to create a single context per device and then manually handle data movements. The interactions between the host and a context are coordinated by **command queues**. Through command queues the user can indirectly launch kernels on devices, or can initiate memory transfers. Command queues allow to extract timestamps for all the commands that are processed by that queue, this allows to get the information when and how long the process has been queued and also how long the execution of a kernel took. This information can be used for profiling or load balancing of the application.

Devices are further divided into **compute units**, whereas those can again consist of one or multiple **processing elements**. The processing elements are the actors that calculate the operations. All processing elements can simultaneously operate on different pieces of data. In order to understand the reason for introducing such a hierarchy, the execution model has to be explained at this point. A kernel describes a function that is executed with an enormous amount of threads that are executed simultaneously in a data-parallel manner; they are called **workitems**. They are combined to **workgroups** of a distinct size. All workgroups together are denoted as work. When a kernel is sent to a device, it is specified how to split the work into workgroups and how many workitems are sent to the device. A workgroup is assigned to exactly one compute unit, i.e., it cannot be split to be processed by two compute units. A workgroup is executed according to the *single instruction multiple data* (SIMD) principle, i.e., the same instruction is simultaneously applied to every piece of data. In order to utilise all compute units, the work must have at least one workgroup per compute unit. An important point is that a workgroup must have the correct size that is usually dependent on the device itself. To find the optimal workgroup size is one of the trickiest things and requires a lot of understanding of the underlying hardware. All vendors have their own best practice guides [24, 25, 26], but even there, they cannot propose a recipe or a formula to calculate the optimal size of a workgroup or workitem. The user will normally end up doing extensive heuristic tests.

3.3.2. Hierarchies of Memory

The memory model of OpenCL is another abstraction of the hardware depicted in Figure 3.4. It is however strongly influenced by the GPU architecture.

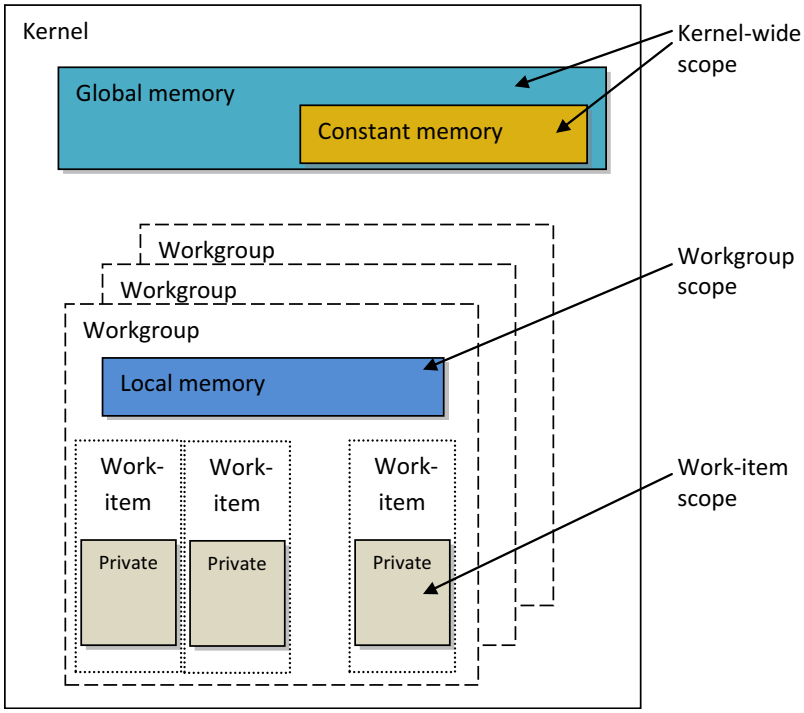


Figure 3.4.: OpenCL Memory Model, taken from [1].

OpenCL distinguishes between four types of memory. Those are global, constant, local, and private memory.

Global and constant memory can directly be accessed by the host. This is the only way to communicate between host and device. Both memory types are shared between all compute units of a device. Simultaneous access might create a high latency due to the bottleneck that is caused by the shared bus, even though the memory is usually n-way interleaved for GPUs. In contrast to global memory that can be read or written by the processing elements, constant memory is read-only for the processing elements.

Local memory is assigned to a compute unit and can therefore share data among a workgroup. Consistency is only guaranteed at synchronisation points in the code. There is no possibility to communicate with other workgroups from that memory. Local memory can be used as a programmable cache between global memory and registers, as local memory is usually much faster

than global memory. OpenCL offers ways to asynchronously copy data from global to local memory within a kernel.

Private memory is only available to each processing element itself. It cannot be used for workitems to communicate within their workgroup and is basically used to store intermediate results.

3.4. Problem Statement

This chapter has presented an overview of existing and upcoming heterogeneous platforms. Yet, there is no simple and easily understandable programming model available to program such systems. OpenCL allows to program CPUs, GPUs, and other accelerators. However, it is not suitable to program complete applications with it, but is rather designed to accelerate computationally intensive parts of an application. OpenCL offers a notion of programming parallel code in a SIMD fashion, which can be seen as an extension to imperative coding. Moreover, OpenCL also offers portability, because kernels can be executed on any OpenCL-enabled device, independent of its type. The price for these extensions is the additional complexity that is introduced by the OpenCL environment. The user must know a lot of low-level details just to setup an application. Furthermore, even though OpenCL provides an abstraction of memory and implicitly manages memory transfers, the user will not be relieved of initiating memory transfers between the host and the OpenCL context. It must also be mentioned, that this memory abstraction is only available within a platform, i.e., for devices of the same vendor. For different vendors, the user has to cope with data management on his own. However, since the main use-case of OpenCL is to accelerate parts of an application, there is often no need for inter-platform communication. Applications are still programmed using common models, and only the computationally intensive functions are outsourced as OpenCL kernels. For this reason, it can be said, that OpenCL does not simplify programming, but rather offers an extension to program additional devices. Portability of kernel code is the main strength of OpenCL, but still, finding the optimal workgroup size and the respective partition of work needs extensive heuristic tests, which hinders portability a lot. Parameters that are optimal for one setup might not be the right ones for another setup.

This chapter also introduced the high-level programming model SDF. It was shown, that the user is relieved of dealing with the usual problems of par-

allel programming, such as synchronisation, indeterminism caused by race-conditions, communication, and dead-locks. The user can focus on writing sequential code for all the processes, and separately specify the needed topology. Moreover, SDF offers ways to perform correctness tests for the topology. Furthermore, a schedule might be calculated offline. However, as it was mentioned, process networks have also disadvantages. One of them is the communication overhead introduced for fine-grained data-parallelism. SDF does basically not provide a notion of a lightweight data-parallelism as it is supported by OpenCL with SIMD. Another disadvantage is that SDF is generally not portable. For different computing units, processes have to be implemented exclusively using the languages and programming models of the respective device. When porting the code to another platform, it has to be adjusted and rewritten manually, which leads to unproductive software development.

Clearly, the advantages of SDF and OpenCL complement each other. SDF is a concept that simplifies parallel programming by hiding complexity. OpenCL, being a framework to program various accelerator devices, offers a way to write performant and portable parallel software. Is it possible to combine these models to get rid of their disadvantages and to combine their strengths? The next chapter analyses the difficulties that arise and elaborates the details of how this they can be combined.

4

Runtime Environment Combining OpenCL and SDF

SDF and OpenCL being based on two different paradigms, the one cannot naturally be expressed as the other. This was one of the major problems to be solved during this thesis. In the beginning it was unclear whether the two concepts were compatible or not, and if a process network could be expressed in OpenCL without heavy limitations. Essentially, the main objective was a combination of SDF and OpenCL to result in a much simpler programming model. It is however often a trade-off to decide between simplicity and bleeding-edge performance. The focus in this thesis was clearly the former.

The particular problems are explained in detail in the following section. Afterwards it is explained how OpenCL and SDF can be combined to a flexible runtime environment.

4.1. Differences of OpenCL and SDF

The two central parts of the SDF model of computation are the processes and their interconnection channels. Implementing processes and channels by means of OpenCL is needed in order to combine the two concepts. This sec-

tion highlights the conceptual issues that arise when combining those models. First an SDF process is compared to an OpenCL kernel. Afterwards the communication principles of both models are opposed to each other.

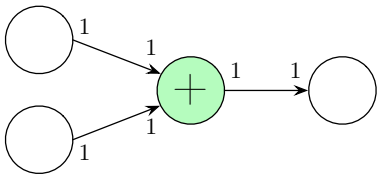
The SDF process describes a simple well-defined function that processes a certain amount of tokens in each invocation. Similar to that is the concept of an OpenCL kernel that describes a limited function, which can only access a specified amount of data in each invocation. While there is an obvious match between these two, they already differ in how data is passed. In OpenCL, the notion of a token does not exist. A token is a logically indivisible amount of data that can be either produced or consumed by a process. An OpenCL kernel has access to certain regions in memory that it can read from and write to. In contrast to a kernel, an SDF process is limited to process every token exactly once, whereas there is no such limitation in OpenCL. Another difference is that an SDF process will automatically invoke itself again after it has finished, whereas an OpenCL kernel must explicitly be put on a command queue from the host application in order to be executed once. Every single invocation must be triggered externally, which is a major difference to SDF. SDF processes will block when no input tokens are available. For an OpenCL kernel it is not allowed to be blocked for an undefined amount of time to wait for input data. Rather, an OpenCL kernel must finish promptly, as it does not offer possibilities for context switching or multi-tasking. Furthermore, an SDF process has ports that specify the interfaces to channels. A kernel in OpenCL has arguments that are passed to it when it is invoked. The arguments are defined as pointers to memory locations which are preallocated by the host application. Additionally, the kernel arguments need to be set manually for every single invocation, which is not the case for SDF. All these differences are one reason why it is not straight-forward to reflect an SDF process by an OpenCL kernel.

Another major difference between OpenCL and SDF is the kind of parallelism they offer. SDF naturally represents task-level and pipeline-level parallelism. Of course, also data-parallelism can be achieved, but especially for fine-grained data-parallelism it will lead to a serious amount of communication overhead. On the other hand, it is the strength of OpenCL to handle such fine-grained data-parallelism in an efficient way. It does so by supporting *single instruction multiple data (SIMD)* processing units. Hence, the OpenCL and SDF programming models complement each other in this. An OpenCL kernel specifies its function such that it can be split to run in parallel on thousands of processing elements simultaneously. An SDF process, on the other hand, is defined as a simple sequential operation, that is only executed on one

processing element. The parallelism exists only on a higher layer defined by the process network. It would be desirable to have a combination of those computation models which includes support for SIMD while still maintaining the ease of the SDF programming model. However, it was unclear if this could be done without a conflict clash of their paradigms.

The second integral part of SDF, the concept of a FIFO channel, does not exist in OpenCL and must therefore be emulated. This, as well, poses several problems that are again caused by the fundamentally different underlying mindsets. First of all, each OpenCL device has its own memory banks and cannot access any of the other devices' memories. As a consequence, data transfers between devices have to be performed manually by the host application using special functions. They arrange the necessary steps, such as the setup of the *direct memory access (DMA)* controller. The transfer times cannot be neglected and are usually a limiting factor for the application speedup. SDF, being a theoretical concept, abstracts how a channel must behave. One process can write to it and another can read from it. In OpenCL this is clearly not the case, as a kernel on one device cannot access data that was written by another process on another device. Another difficulty is that all the input data must have been copied to the respective device before a kernel can be started. After a kernel has successfully finished, data has to be transferred back again. The order of the data copy actions and kernel invocations must explicitly be programmed by using one of the various synchronisation primitives offered by the OpenCL interface. A kernel will only produce deterministic results if data management and synchronisation is handled properly. Moreover, the goal of implementing a performant FIFO channel can only be reached if the buffer location is chosen carefully and the memory transfers between those are handled transparently. As a channel interconnects two processes, the choice of the buffer location is dependent on the decision where the processes are mapped to for execution. Clearly, not all combinations are feasible or reasonable. All these issues need to be solved in order to implement a FIFO channel that is compliant to the SDF specification and at the same time is performant enough so that the execution of OpenCL kernels is not unnecessarily slowed down.

As it has been clearly shown, there is no one-to-one mapping from SDF to OpenCL. In fact, the issues listed above need a couple of sophisticated solutions which are detailed next.




```
typedef __global int token_t;
__kernel void add( token_t *IN1,
                   token_t *IN2,
                   token_t *OUT)
{
    *OUT = *IN1 + *IN2;
}
```

Figure 4.1.: Representation of the SDF process (+) on the left hand side as a functionally equivalent OpenCL kernel.

4.2. Overview of the Proposed Solutions

An overview of the basic concepts adopted in this work is given in this section. First, it is discussed how processes can be implemented. Afterwards, the very basic concepts of implementing a FIFO channel are explained. And finally, an important principle of a task activation framework is introduced. The details to all these concepts are presented later in the following sections.

A very basic SDF graph is depicted in Figure 4.1. The process in the middle adds two numbers from its input channels, and puts the result to the output channel. This example is mainly used to illustrate the basic concepts and is of course not realistic (a simple addition is not worth the parallelisation overhead, hence a process usually describes a much larger function with many instructions). The representing kernel for this example is depicted on the right hand side in Figure 4.1. The ports that are needed as interfaces to the channels are defined as kernel arguments `IN1`, `IN2`, and `OUT`. They are pointers to integers in the global memory of the respective device. Global memory is the only way to communicate with the host or with another kernel. Before the kernel can be started, the host must allocate the memory for this data, transfer the input data in that memory, and finally pass the references to this region to the kernel. In the body of the kernel function the two numbers are added and then stored in the memory location representing the output channel. For a general process with a number of input and output ports, the corresponding kernel will be given exactly one argument per port.

Although the example above is functionally correct, it does not exploit the possibilities offered by OpenCL. The kernel in Figure 4.1 is just a simple sequential code, that does not benefit from any granularity of data parallelism. Of course, the SDF can be extended in a way that the a process is duplicated many times, in order to achieve fine-grain parallelism. This procedure is de-

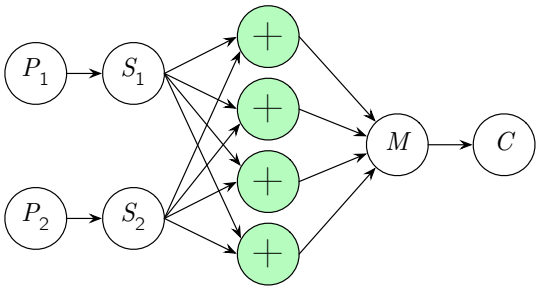
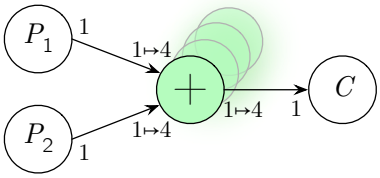


Figure 4.2.: Approach of achieving data-parallelism in an SDF by using splitters (S_1 and S_2) and a merger (M).

picted in Figure 4.2. The process network still calculates the same operation. The difference is that four adders can work in parallel on different pieces of input data now. The input channels of the input token producers P_1 and P_2 must be connected to all four adders. To achieve this, two data splitters S_1 and S_2 have to be inserted between the producers and the adders. They will simply take a token from the input channel and puts it to one of their output channels in a round-robin fashion. As the output token order must be maintained, a merger M needs to be placed between the adders and the consumer process C. It does the inverse of a splitter, i.e., it takes a token from one of its input channels and puts it to the output channel. The order must be strictly the same for the splitters and the merger. This procedure is documented in the original work on SDF in [22]. The disadvantage of this approach is that it becomes quite complex when adding even more parallel processes, and soon, the mergers and splitters will become the bottleneck of the application. It also requires more resources as it needs many more data channels, which all need some kind of memory to store the tokens.

This raises the question whether there is a more efficient way to enhance an SDF to use data-parallelism without the additional complexity of splitters and mergers. This thesis proposes to use a notion called *shadow copy*, which is depicted in Figure 4.3. A shadow copy is — as the name suggests — a copy of a process that performs the same operation as its original one, but on another piece of data. Instead of having an own channel for every shadow copy, the original channel is reused and the token rate is adjusted. In the simple adder example there are still two input and one output channel, as originally in Figure 4.1, but the token rate is adjusted to process four input tokens at the same time and put four tokens to the output channel. The corresponding OpenCL kernel that implements this functionality is shown in Figure 4.3




```
typedef __global int token_t;
__kernel void add( token_t *IN1,
                   token_t *IN2,
                   token_t *OUT)
{
    uint g_id = get_global_id(0);
    OUT[g_id] = IN1[g_id] + IN2[g_id];
}
```

Figure 4.3.: Approach of achieving data-parallelism in an SDF by using four shadow copies. The right hand side depicts the corresponding OpenCL kernel for the (+) process. The token rate of the channels is adjusted accordingly.

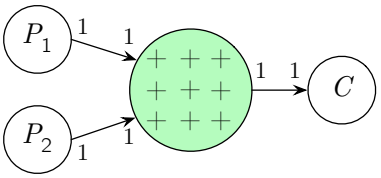


Figure 4.4.: Approach of achieving data-parallelism in an SDF by using inherently parallel operations within a process (in this case a vector addition).

on the right hand side. The arguments are now pointers to integer arrays. When the host application inserts a kernel to the command queue, it is specified how many workitems of this kernel will be launched. Every workitem has a distinct identifier which can be obtained by the OpenCL builtin function `get_global_id(0)` from within a kernel. This identifier is used as an array index for the kernel arguments. Each workitem will therefore only add two integers, but as there are multiple workitems executing concurrently, data-parallelism is achieved efficiently. In Section 4.5 the discussion is resumed and the notion of a shadow copy will be assigned to a complete workgroup instead of a single workitem. How the order of the tokens is guaranteed will be explained later, when the FIFO channel concepts are outlined.

There is yet another way to parallelise a process. Especially when processes get bigger, they might be inherently parallel in their function. The simple example can be refined to fulfil this property when the tokens on the channel are seen as vectors of multiple numbers as it is depicted in Figure 4.4. The adder performs a vector addition, which is something that can be parallelised to use as many workers as the vector has dimensions. In the figure the token rate on the channel is equal to the one in the original sequential example, but in contrast to that, the operation is performed concurrently by nine adders

at the same time, which is hidden in the process itself. This is exactly the concept of a SIMD operation, which is mainly offered by GPUs and can be used with OpenCL. This kind of parallelism can be used for fine-grained data-parallelism, whereas the shadow copies will offer rather coarse-grained data-parallelism on the layer of processes. More on this topic — especially how the two approaches can be combined and how to achieve a good performance — is detailed in Section 4.4.

Following the description of the elementary concepts of processes in OpenCL, the next paragraph introduces the basic ideas of the FIFO channel emulation. Channels are used to connect two processes and, as already mentioned, do not exist in that form in OpenCL. Concerning the implementation of the channel emulation, a couple of decisions were taken: Firstly, the channel is implemented as a ring buffer in linear memory. Secondly, each buffer has a fixed capacity. As it is possible to calculate the needed buffer size for both channels of a valid SDF network, this decision does not affect the functionality of the application.

The third decision was about the interface to a channel. Tokens can be read from a channel or written to it, these operations are called *pop* and *push* for a FIFO channel. As already mentioned, a kernel can only communicate with the host by its kernel arguments being provided by the host application. This means that global memory has to be allocated beforehand and is managed externally. That fact is nicely represented by the following interface. A channel can be asked to reserve a certain part of the memory to be exclusively available to one process, this is called *acquire*. Eventually when the process has finished, the same part of memory must be returned, which is named *release*. The interface is symmetric, which means it is used for both the *push* and the *pop* operation. Push describes the action of inserting data into the FIFO, whereas pop describes the action of reading and removing data from the buffer. Thus, there are four public interfaces for a channel: *acquire push*, *release push*, *acquire pop*, and *release pop*. This gives the flexibility that is needed to implement a channel in a few fundamentally different ways that will be described in Section 4.3. The interface is similar to the concept of a *windowed FIFO (WFIFO)* as it was described in [27]. For reasons of compatibility, the terms *acquire* and *release* were borrowed from there.

The difficulty with those channels is that an OpenCL setup might have many devices, as illustrated in Figure 4.5. Each device has its own memory that cannot be accessed by any other device. Processes can be mapped to any of those, and the data has to be transferred between them. The vital question is where

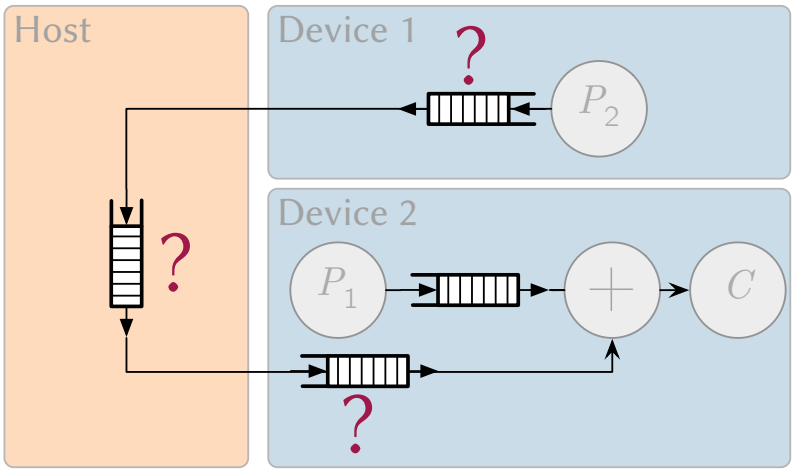


Figure 4.5.: Buffer location.

this buffer needs to be placed. In the example in Figure 4.5, one process P_1 and the adder are connected through a channel. Both are mapped onto *device 2*. In this case, it makes sense to place the buffer directly in the memory of *device 2*, as both processes are able to access the same memory location. How such a ring buffer can be implemented in a device memory is detailed in Section 4.3.3. It becomes much more complicated when two connected processes are placed on different devices. Where should the buffer be placed to get the best performance? Process P_2 is placed on *device 1* whereas its successor, the adder, is placed on *device 2*. The data transfer is controlled by the host, so the memories of all three are somehow involved in the transaction. It is clear that a simple ring buffer will not suffice in this case. There are multiple different solutions, some of which perform better than others. Section 4.3.2 lists the possibilities and compares them.

A related topic to the buffer placement is the timing of the respective data transfer actions. OpenCL cannot generally transfer data automatically[1]. A schedule is needed to transparently initiate memory transfers and guarantee that the data is available at the respective device when it is needed by a kernel. Figure 4.6 depicts a directed graph that illustrates the dependency of the channel interface and the invocation of the two connected processes, which is also

[1]OpenCL can only manage data transfer within the same context. Moving memory from one device to another only works if both devices are in the same context. This is usually not the case, especially not when the devices belong to a different platform.

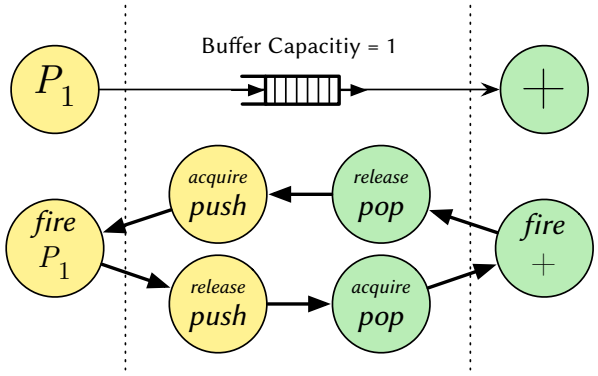


Figure 4.6.: Dependencies between process invocation and channel interfaces. The nodes represent functions calls. For this simple network with two processes and one channel with capacity of one token, the call order has to be a counterclockwise circle, as illustrated.

known as *fire*. For this simple example consisting of two processes connected through a channel with capacity one, the function call order will be strictly in a counterclockwise circe. In a more general case it has to adhere to the following guideline: Before a kernel can be started, it must be guaranteed that the data is resident in the memory of the device where the kernel is mapped onto. Thus, the host application must *acquire* the corresponding memory from all input and output channels of the process. For *pop*, the channel has to make sure that the data is transferred to the device, whereas *push* will make sure that the location can be exclusively used by this process. When the *acquire* operations have successfully completed, the process can be *fired*. Eventually, when it has finished, the host must guarantee that all channels are *released*. For *pop*, it will allow another process to overwrite the memory location, and for *push*, it must fetch and store the results.

However, as the channel size is fixed, a channel can become full at some point in time. It is certainly not valid to push data in a full channel. On the other hand, it is also invalid to pop data from an empty channel. This leads to the next important aspect: All kernel invocations and buffer copy actions need to be coordinated. This is done by the host application. The example in Figure 4.6 is really simple, but adding a few more processes and channels will introduce many more dependencies and also offer multiple valid solutions. Therefore a generic concept is needed to deal with the additional complexity introduced by growing SDFs. In Section 4.4, two different approaches of a *task activation policy* are introduced and discussed. The basic concept for both

of them is that a kernel can only be executed when its input channels contain enough data and all output channels have enough free space. The host application maintains a list of all processes and channels. All channels are monitored, and depending on the state of those, the host decides to initiate memory transfers and kernel invocations. However, the details of how this is achieved, differ according to the implementation.

4.3. Communication Channels in OpenCL

The different possibilities of implementing SDF channels by the means of OpenCL are elaborated in this section. First, a simple approach with a ring buffer in the host memory is explained. Then, an improved version with fewer memory copy actions is introduced. Afterwards, it is explained how a channel can be implemented with device memory only. And finally the concepts are merged to a general channel that combines the advantages of all the previous concepts.

4.3.1. Ring Buffer in Host Memory

This section describes a simple approach of implementing a channel between two OpenCL devices. As the host coordinates all memory transfers between those, it stands to reason to just place the buffer in the host memory. This means that a ring buffer is placed in the host memory and it can be programmed with any common programming model, such as an STL container for C++ (i.e. the circular buffer implementation of the boost library). A channel, in this case, will only implement the interfaces between the host and the devices. Figure 4.7 depicts this concept with an example SDF consisting of two processes. Process P produces three tokens per invocation and process C consumes the tokens one by one. The ring buffer of the channel is completely stored in the host memory. The kernel arguments (of type `cl_mem`) point to the locations storing the tokens for one invocation of the process. In order to be accessible from the host, a `cl_mem` buffer object has two representations: one in the device memory and one in *host unified memory*, which is in practice equal to host memory[2]. Both memories need to be synchronised which is only guaranteed at certain synchronisation points. One way to request the device memory

[2]It might be a special region in memory known as pinned memory which is not pageable [25, Section 3.1.1]. Another restriction is that it usually has to be aligned to a certain boundary[6, Sections 5.2.1, and C.3].

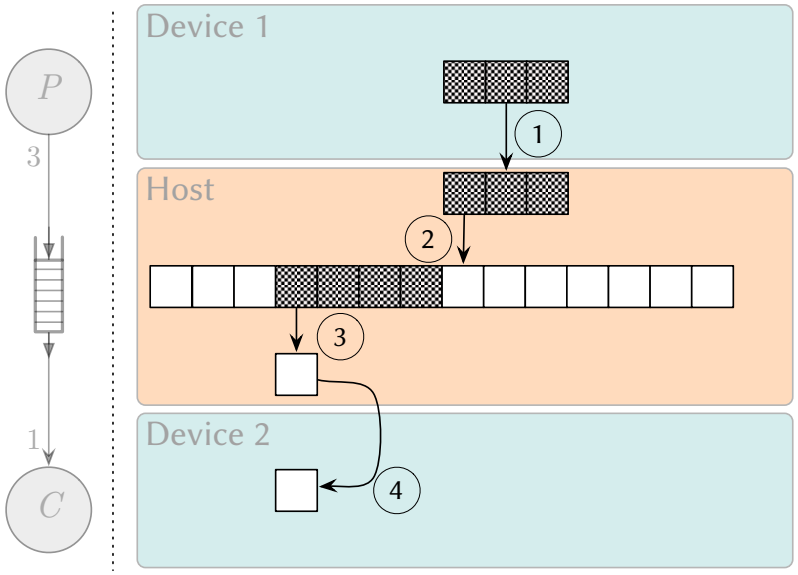


Figure 4.7.: Approach of implementing a channel based on a ring buffer in host memory. The process P on device 1 sends data to C on device 2 through this channel. The tokens of one invocation are stored in each of the devices' memories, whereas the complete buffer is only stored in the host memory. The push action will first transfer the data from the device to the host (1), and from there copy it to the correct location in the FIFO (2). The pop operation will afterwards copy the data within the host (3) and finally transfer it to the device (4).

to be copied to its counterpart on the host is when `clEnqueueReadBuffer()` is called by the host application. As depicted in Figure 4.7, the OpenCL driver will then typically in step one initiate a DMA transfer from the device memory to the host memory. Eventually when it has finished, the driver might copy the data to the region that it was asked to put the data in step two[3]. A similar procedure happens when `clEnqueueWriteBuffer()` is called in steps three and four, where the memory transfer is again divided in two copy operations.

Another property that needs to be guaranteed is that the tokens for a shadow copy are ordered correctly. As it was mentioned in Section 4.2, the concept of shadow copies adjusts the token rate on one side of a channel. The order of the tokens has to be maintained, which must be guaranteed implicitly by the channel. A ring buffer preserves the order of the inserted data and will afterwards use the same order for the pop operation. The tokens are accessed

[3]This behaviour could be observed by all drivers (Intel, AMD, and NVIDIA) in their current version.

in the same order for the input and the output by the kernel. This property guarantees that the order of tokens is always correct for shadow copies and that it is therefore possible to get rid of the splitters and mergers denoted in Figure 4.2. The advantage of the shadow copy approach is that data will be copied in larger blocks, which is usually faster than copying data in smaller pieces.

The basic requirements for implementing a channel are therefore fulfilled by the proposed concept. However, this approach has the disadvantage that a process is actually blocked during the memory copy operations. Even if there is still space in the FIFO, the producer process cannot continue with its next invocation. The same is valid for the consumer process. Even if there is still data available, it has to wait for its only buffer to be refilled again. To circumvent this, one could use multiple `cl_mem` buffers in the device memory and use them interchangeably. A reasonable number of alternating buffer sets might be three, one for reading input data, one for execution, and one for writing output data back. Before this problem is addressed, there is however yet another disadvantage that is more severe, which is the lack of efficiency of this implementation. The goal of having as few memory copy operations as needed is clearly not fulfilled as there are two such copy actions per push and pop operation.

4.3.2. Triple Buffering

The best implementation of a channel would have zero copy actions. However, as different devices often have different physical memories, only a DMA controller can transfer data between them. Therefore it is not possible to have zero copies in a general OpenCL setup. Each token has to be copied at least twice when devices do not have host unified memory. The token is produced at the first device, then transferred into host memory, and finally copied to the second device. Two implementations that fulfil this property are detailed in this section.

Both approaches can be illustrated with Figure 4.8. They have in common that the buffer is completely available in all three memories, but they differ in how the buffers are kept synchronous.

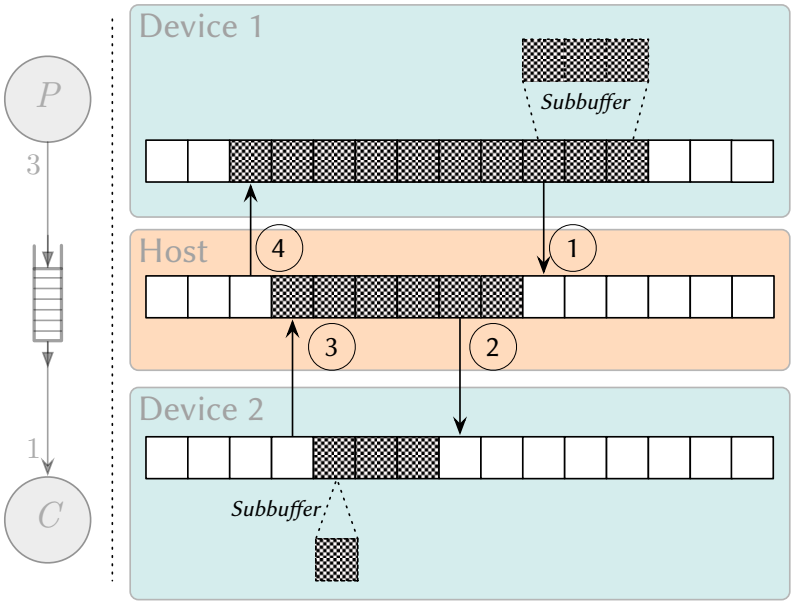


Figure 4.8.: Approach of implementing a channel with triple buffering. Process P on device 1 sends data to process C on device 2 using this channel. Tokens are denoted as subbuffers which is a specified region of a buffer of variable size. The complete buffer exists in the memories of both devices and in the host memory. Data transfers (1 and 2) and data invalidation actions (3 and 4) are coordinated by the host memory and are executed on the level of subbuffers.

Overlapping Buffers in Host Memory

Both approaches use two buffers of type cl_mem for the whole channel — one that is bound to the context of the first device and one to the context of the second device — instead of having a buffer for only one token as in Section 4.3.1. The size of the buffer is therefore equal to the capacity of the channel. Since a buffer usually consists of multiple tokens, it has to be divided. For the token emulation on the channel, *subbuffers* are used that can be created by the `clCreateSubBuffer()` function. In every instant of time a buffer can only be valid at one location. Luckily it is possible to coordinate the buffer transfers on the level of *subbuffers*. The synchronisation of the *subbuffers* must be handled manually from the host application which is described later.

So far, this concept has only introduced more complexity and does not solve the problem of too many memory copy operations. To understand the problem better, some details must be mentioned here. Calling `clEnqueueReadBuffer` needs a pointer to host memory to be passed as an argument. It was observed that this operation took notifiably shorter in some rare cases. The problem was analysed and it turned out that the pointer was aligned in this case (a certain number of zeros for the least significant bytes of the address). OpenCL does not explicitly mention that a `clEnqueueReadBuffer` must be aligned. But in the case when it is not aligned, the driver implementation will typically copy the data twice: first to a temporary host memory region that is aligned using a DMA transfer, and second, to the region that was provided by the user. This behaviour is not optimal, but is a simplification for the user. For the channel implementation that is proposed in this section, there is a nice workaround of the described problem. Instead of using a generic FIFO implementation, it is proposed to allocate the FIFO buffer that points to aligned memory[4]. This speeds up data transfers and limits the number of data copy operations. However, the alignment condition must also be fulfilled for every token in the buffer (because this is the smallest unit of data that is transferred). This limits the token size to a multiple of the alignment condition[5], which is typically in the range of 128...256 bytes. For the alignment condition a_{DEV1} and a_{DEV2}, that correspond to the to devices that are involved with a channel, the token size restriction s_T can be specified as:

$$s_T = n \cdot \max(a_{DEV1}\ ,\ a_{DEV2}) \qquad \forall n \in \mathbb{N}. \tag{4.1}$$

[4] Aligned memory can be allocated with the function `posix_memalign()` for POSIX-compatible systems.

[5] The alignment condition is device specific and must be queried using `clGetDeviceInfo`.

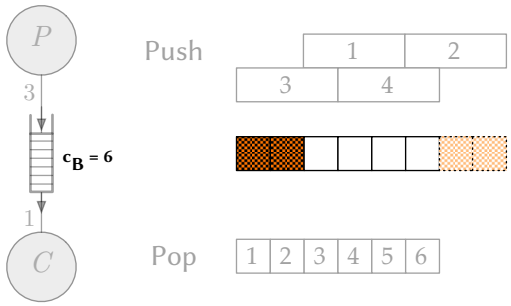


Figure 4.9.: Channel access order for the situation with two initial tokens (dark orange). Push operation 2 will write data over the boundary; this has to be managed by the implementation, which copies the data from this region to the beginning of the buffer after completion.

Another restriction concerns the buffer capacity for this implementation where the tokens are mirrored as subbuffers. A token must be a piece of data in linear memory. As the subbuffer does not copy any data, it must be guaranteed that the buffer is evenly dividable by the token size. The buffer capacity c_B can be expressed in terms of the token size s_T and the token rates at the input of the channel $r_{T,i}$ and the one at the output of the channel $r_{T,o}$ as:

$$c_B = n \cdot \text{lcm}(r_{T,i} \cdot s_T \ , \ r_{T,o} \cdot s_T) \qquad \forall n \in \mathbb{N}. \tag{4.2}$$

This restriction is valid for channels without initial tokens. The more complicated case of having initial tokens is illustrated in Figure 4.9 for a simple SDF with two initial tokens placed on the channel. The buffer size is chosen according to the restriction above. However, the second time when process P writes to the channel fails, because it writes data over the boundary of the channel. In order to deal with this situation, the buffer has to be enlarged by the amount of initial tokens. After the data has been pushed there, the *release push* function must copy the data from this region to the beginning of the buffer. This only needs to be done once for the whole runtime and will therefore not decrease the general performance of the implementation. For i initial tokens, the buffer has to be allocated according to:

$$c'_B = n \cdot \text{lcm}(r_{T,i} \cdot s_T \ , \ r_{T,o} \cdot s_T) + i \cdot s_T \qquad \forall n \in \mathbb{N}. \tag{4.3}$$

The additional space will only be written once and never used by the *pop* interfaces.

In this approach, it is not possible to use common libraries for the ring buffer implementation. It is clearly not a simple ring buffer anymore, and the host

application has to keep track of which regions contain unread data and which data are available to be overwritten. The functionality can be hidden behind the *acquire/release* interface that was introduced in Section 4.2.

So far, it has been described what the two approaches have in common. Next, it is explained where they are different.

Synchronisation Method Using clEnqueueReadBuffer and clEnqueueWriteBuffer

In this approach, the channel consists of a buffer with three representations, as it was described in Section 4.3.2. Each part of the buffer is only valid at one location per instant of time. This method uses `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` as synchronisation primitives. The basic approach is illustrated in Figure 4.8 for a token rate of three and one. For the example in the figure, the approach can be described as follows: First, a *subbuffer* with size three is created by the host in the context of *device 1*. Then, the kernel of process *P* is invoked with the *subbuffer* as kernel argument. Eventually, when the kernel execution has finished, the host will initiate a data transfer from the device to the host memory using `clEnqueueReadBuffer`. Meanwhile the process can already work on the next *subbuffer*. After the data has been successfully transferred to the host memory, the data can be transferred to *device 2*. For this, a *subbuffer* with size one is created. Finally, this *subbuffer* is copied to the memory of *device 2* using `clEnqueueWriteBuffer`. The created *subbuffer* is then used as a kernel argument for process *C*.

Of course, the information when a part of the buffer can be reused must also be traversed backwards. In this case this is rather simple, as the buffer region only has to be marked as invalid by the host application. So the host is simply waiting for process *C* to be finished and will then mark the region in the original buffer as free space.

In a more general case with many more processes and channels, it is not as simple as that example anymore. The basic principles are however exactly the same. How the schedule is done will be explained in Section 4.4.

Synchronisation Method Using clEnqueueMapBuffer

There is another method of synchronisation offered by OpenCL which allows to build the same functionality as above. The second approach uses

the `clEnqueueMapBuffer` and `clEnqueueUnmapMemObject` as synchronisation primitive. Mapping is the operation of explicitly synchronising the device memory to the host memory. As it can also be applied to *subbuffers*, it is possible to build a channel similar to the one in the first method. Again, the buffer is therefore only valid in one memory at the same time.

For the example in Figure 4.8, process P will write data into its assigned *subbuffer*. After finishing, the host will initiate the *subbuffer* to be mapped into its own memory (step 1). The important thing here is that the same region of the second buffer is also already mapped to the host memory. In step 2, the second buffer can be unmapped for this region, which will initiate the data transfer to the second device. So far the approach was quite similar to the first synchronisation method. The difference is how the data is invalidated. The initial state of the buffers has to be restored, i.e., the buffer of *device 2* has to be mapped to the host memory, and the buffer of *device 1* has to be unmapped in order to be valid in the device memory. So in step 3, the *subbuffer* will be mapped back to host memory after the data has been processed. In this case it is reasonable to set `CL_MAP_WRITE_INVALIDATE_REGION` as option, because it will tell the OpenCL driver that the data is only needed for writing data to it, and therefore that it does not have to transfer any data back. Finally, in step 4, the same region from the first buffer is unmapped back to the device memory.

The map and unmap operations are reported to sometimes produce a higher throughput for memory transfers[6]. Especially when the buffer is not changed, this operations will not initiate any DMA transfers. In the case of an SDF channel, the buffer is always completely overwritten (a new token is written to the buffer). For this reason, a performance increase could not be observed. Even worse, the invalidation did not turn out to be free of cost. Since this approach also adds more complexity, it was discarded. The synchronisation method with `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` is simpler and, from a performance perspective, not worse than the approach described here. Section 4.3.4 introduces a *hybrid channel* which can work on pointers for every case where this is possible. The next section will explain how a zero-copy channel can be achieved.

[6]The Intel OpenCL best practice guide suggests to use this [26, p. 17]. They argument that map/unmap is lightweight and comparable to passing pointers.

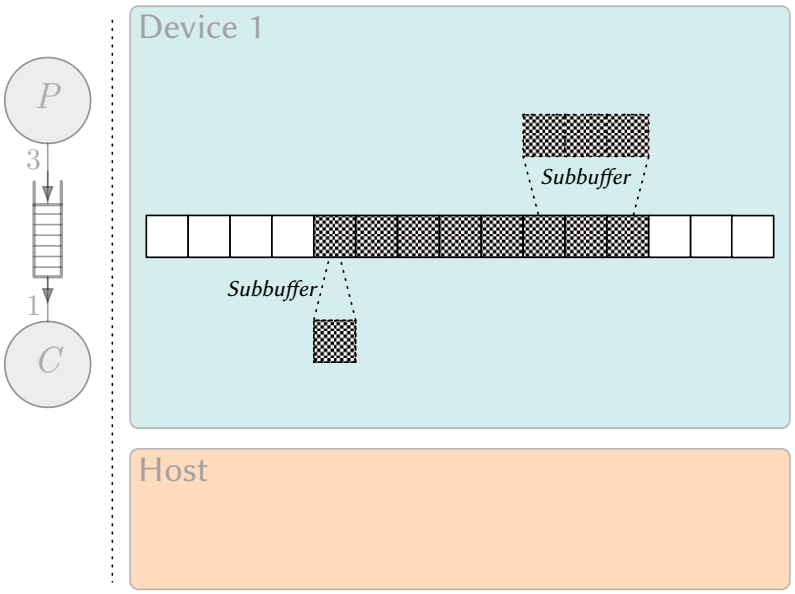


Figure 4.10.: Approach of implementing a channel with single buffering. The process P sends data to C using this channel. Both processes are mapped onto the same device. Tokens are denoted as subbuffers which is a specified region of a buffer of a variable size.

4.3.3. Channel Implementation Directly on Device Memory

The implementations that were discussed so far were all targeting channels between two devices or contexts, respectively. All of them can of course also be used for channels between processes that are mapped onto the same device. However, this is inefficient, because it does not make sense to transfer the data to the host memory and back in every invocation. For this case, it is possible to build a buffer that needs zero copies. This section explains how a zero-copy channel can be implemented by means of OpenCL. Figure 4.10 depicts the same SDF with producer *P* and consumer *C* as in the previous examples. But in contrast, this time, both processes are mapped to *device 1*.

The approach is quite similar to the one described in Section 4.3.2. There is also one big buffer of type `cl_mem` and tokens are emulated as *subbuffers*. The difference is how the buffer is allocated, as it only needs one buffer instead of two. And instead of allocating the buffer first in the main memory, OpenCL can simply create its own buffer. When the option `CL_MEM_HOST_NO_ACCESS` is set as a parameter to `clCreateBuffer`, the OpenCL driver is informed that

no copy of the buffer has to be allocated in the host memory. As the buffer is only used from the device memory, it need not be synchronised with the host memory and all memory transfer operations can be omitted.

For the simple example shown in Figure 4.10, the cycle of operations could be the following:

1. Create a *subbuffer* of size three for the output data of process P and set it as kernel argument,
2. invoke process P to write data into the *subbuffer*, and
3. release the *subbuffer* object after the process has finished.
4. Create a *subbuffer* for the input data of process C and set it as kernel argument,
5. invoke kernel C to process the token,
6. finally release the *subbuffer* object.

For simplicity, the *subbuffers* were always created and released again in the example above. From an implementation perspective, this is not efficient, and the *subbuffers* should be stored and reused instead of always creating and releasing them[7].

4.3.4. Combined Channel Implementation

The channels in Sections 4.3.2 and 4.3.3 were built to be equal in as many details as possible. This allows to combine the approaches to a general channel implementation. The differences can be hidden behind their common *acquire/release* interface. The implementation of a channel depends on the mapping of both processes that are interconnected through it. This mapping is assumed to be known at runtime (Later in Chapter 5 it is explained how the mapping problem could be solved a priori). In Figure 4.11, the processes of the adder example are mapped as illustrated. Channels c_1 and c_3 connect the processes that are mapped onto the same device. In this case, there is no need for the indirection over the host memory. Therefore, the implementation can directly access the same buffer without memory copy operations. A zero-

[7] It has been observed that for the drivers of AMD and Intel, releasing the buffer did not have any effect. However, the NVIDIA driver always transfers the complete buffer into host memory when releasing a *subbuffer* and reallocates and fills the memory again when it `clCreateSubbuffer` is called. This is really inefficient and it is therefore advisable to only create the *subbuffers* once.

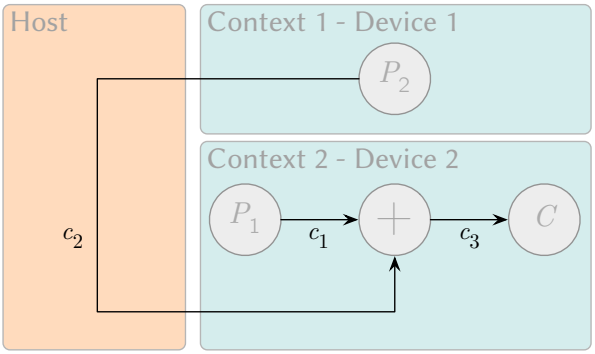


Figure 4.11.: Selection of the correct channel implementation based on the mapping of the processes that are interconnected through it.

copy channel can be used for both channels c_1 and c_3. For this, the allocation of the buffer can be done by the OpenCL driver using `clCreateBuffer` with `CL_MEM_HOST_NO_ACCESS` as parameter. For channel c_2 it is clearly needed to use the host memory as indirection. So, in this case, the channel has to be implemented as it was described in Section 4.3.2. Summarised, two buffers are created that point to the same region in the host memory, one of them for *device 1* and the other for *device 2*. The channel implementation takes care of transmitting the data from one device to the other.

An additional special case of a channel implementation is depicted in Figure 4.12. Device 1 has *host unified memory*, which means that it is possible to directly pass a reference of a buffer to the kernel without copying the data first. This corresponds to the implementation of directly accessing the buffer as if it was on the same device. For the other side however it is still needed to copy the data from the host memory to the other device that does not have host unified memory. Therefore this special case can be seen as a hybrid implementation of the two approaches. This led to the decision to implement a channel with interchangeable ports for both sides, namely an input and an output port. The input port implements the interfaces *acquire push* and *release push*, whereas the output port implements the interfaces *acquire pop* and *release pop*. The channel will allocate the memory depending on the mapping of the processes and the properties of the devices that are involved. This flexible approach allows to communicate to processes executed as native POSIX threads by providing a simple additional variant of a port, which is needed in Chapter 5 when the approaches of this thesis are integrated into a larger programming framework called *distributed application layer*.

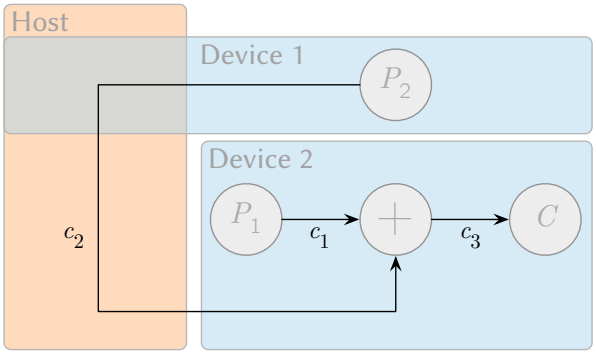


Figure 4.12.: Device 1 has host unified memory. The channel implementation can use this information to select a more performant variant of its channel implementations.

With these hybrid channels, it is possible to emulate the behaviour of pointer passing that was suggested by Intel with their map and unmap implementation [26, p. 17]. The proposed implementation does therefore successfully emulate an SDF channel in OpenCL in a performant and flexible way. The next section will elaborate another important topic — the task activation framework.

4.4. Task Activation Policy

The building blocks of an SDF were described in the sections before. The processes are described as kernels, the channels can be emulated, and finally those two components can be connected. However, there is still one important thing missing. All the concepts so far did not include the coordination among the different interfaces. This section shall introduce how this can be done.

As it was explained already, a kernel is not allowed to be blocked to wait for data. Therefore, in OpenCL, all input data, and also all output spaces must be available at kernel invocation. The dependencies between the interfaces of two processes and one channel were depicted in Figure 4.6 as a motivational example. The schedule for the actors was trivial in that case, as there was only one valid possibility. A marked graph representation for a real channel is depicted in Figure 4.13. Essentially, a marked graph is an SDF with all the input and output token rates equal to one. Clearly, in this example there are multiple valid ways to schedule the actions, e.g., it is valid to invoke all functions counterclockwise starting at *acquire pop*, and it is also valid to first invoke

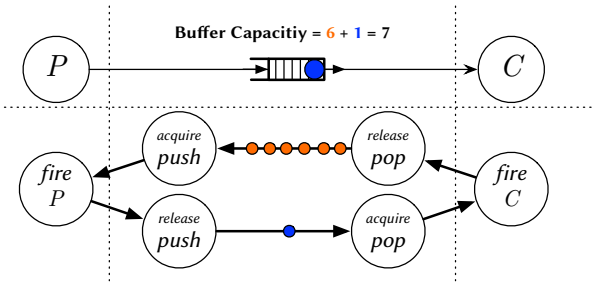


Figure 4.13.: A simple producer-consumer SDF can be modelled as a marked graph that describes the interface dependencies between two interconnected processes through its channels. The orange marks correspond to the free places in the channel, whereas the blue token corresponds to a mark currently lying on the channel.

the *acquire push* function six times and then to continue with *fire P* and so on. The first schedule allows both, *P* and *C*, to be executed simultaneously on different devices, whereas the second schedule will serialise the invocations. Figure 4.14 depicts the adder example of Figure 4.1 modelled as a marked graph. Even though there are only four processes and three channels, the marked graph is already rather large and there are various valid possibilities to schedule the actions. Clearly, the complexity will be even higher for a more general SDF. For larger complex process networks it becomes incomprehensible to find a good schedule. For this reason there must be a task activation framework to automatically achieve coordination among the channel interfaces and the process invocations. In this work, two different possibilities of implementing such a framework have been looked at. They are discussed in the following.

Direct Approach

In OpenCL, for almost every synchronous function, there is also an asynchronous version of it. For those functions, a callback function can be registered that is guaranteed to be executed somewhen after the function has completed[8]. The first approach to build a task activation framework was based on these callback functions. The approach is mainly following the model of the marked graph (e.g., Figure 4.13 on the bottom). It started from a simple SDF implementation in OpenCL and served as a proof of concept. The simple

[8]OpenCL uses the concept of callback functions that can be registered to an event object. The documentation thereof can be found in [6, Sec. 5.9, p. 185].

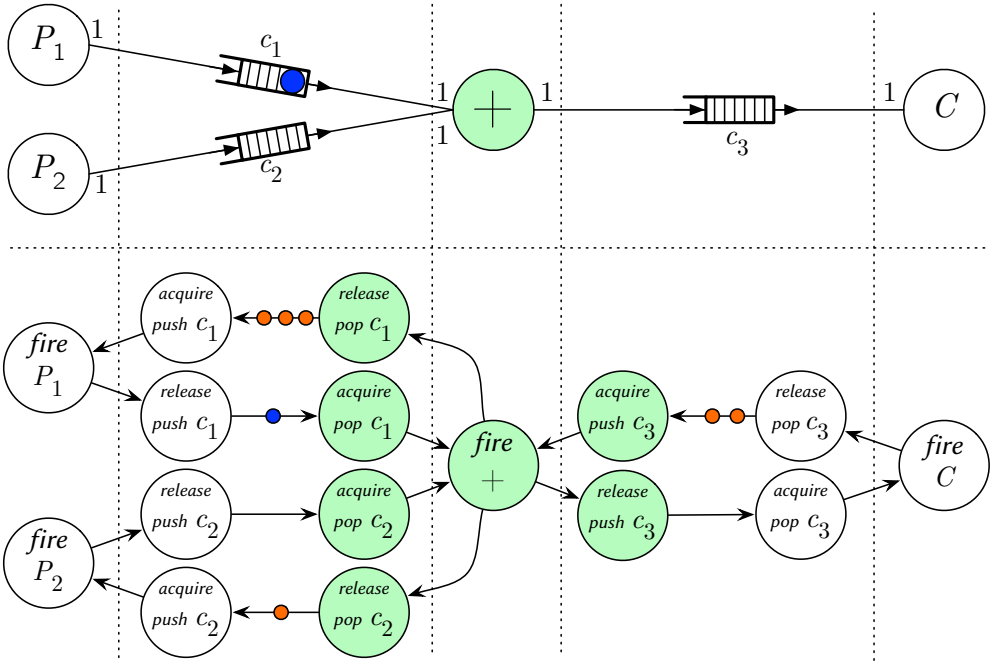


Figure 4.14.: The SDF from the initial example in Figure 4.1 can be modelled as a marked graph. The orange tokens stand for the number of free places in the FIFO. Tokens currently lying on the channel are coloured in blue.

application was then step by step extended to a more general framework. The original example is the producer-consumer example in Figure 4.13. The implementation is based on the simple marked graph representation in the bottom of the figure. This means that for every node all output arcs are implemented as callback functions. For example, in *release push*, a callback function is set to *acquire pop* of the successive process. The *fire* node is invoking the kernel and when the kernel has finished, the respective *release* function is called. The process was afterwards extended to work for more general SDF topologies. The extended version is therefore able to cope with multiple input and output channels per process.

However, this implementation has some limitations. The framework cannot exactly control when a function should be called. This implies that it is unable to properly schedule processes, because it cannot enforce the order in which they are invoked. Moreover, for a setup with multiple OpenCL drivers, each driver will decide when a callback function must be invoked (the specification is rather vague on that point, every implementation fulfils the specification if it will invoke all callback functions somewhen in the future after the event has been reached). Another issue might be that the stack could grow infinitely when the recursion level is not watched by the OpenCL driver (`clSetEventCallback` within a callback function[9]). However, in reality this behaviour could not be observed. The main reason for reimplementing the framework was a serious problem that could not be solved with this approach. During the application runtime it happened that all devices had been idle for some milliseconds even though there were processes that could be executed during that time. This effect could be observed mostly for bigger SDFs and processes with a short runtime. The reason for this behaviour is yet unclear. At this point it was hard to debug this phenomenon, because the OpenCL drivers are closed source.

Another restriction was, that the implementation could not make use of larger buffers, as the design did not offer an easy way to integrate this feature.

[9] In OpenCL it is possible to register a callback function within a callback function. It has been observed that some implementations will directly invoke those, if the event is already `CL_COMPLETE`. If the recursion level is not watched properly, this might lead to a stack overflow. For this reason one should check carefully whether the implementation is correct on that point.

Queue-Based Approach

At this point the decision was taken to refactor the implementation. The new implementation should be process-centric instead of channel-centric. This means that the processes should request the channels to *acquire* or *release* a token, instead of the channel requesting the process to *fire*. The control should completely be given to the task activation framework, i.e., it should coordinate all function calls and decide about their order. Moreover, the OpenCL command queues should be kept busy and only be flushed in batches of multiple commands. Additionally, the framework should support for concurrent DMA communication and kernel execution. Finally, callback functions should be used only where needed, and should be executed as quickly as possible.

The main part of this implementation is a *task queue*. In this queue, a task describes some function that must be executed, e.g., a process invocation. The single-threaded *task queue handler* decides when to execute a task and therefore establishes the order of task discharging. An important property of a task is that it must be defined to be non-blocking, otherwise it might cause deadlocks, because the task queue is single-threaded.

In order to better manage the graph structure of process networks, the processes and channels are mirrored by asynchronous representation objects viz. *process launchers* and *channel statuses*. The process launcher communicates with the OpenCL runtime environment. When the *fire* method is called, it will *acquire* data from all of its input channels and *acquire* space from all its output channels. Afterwards, the process launcher enqueues its kernel invocation. All the respective OpenCL operations are inserted to the same ordered command queue. Finally, the *release* interfaces are called and the necessary OpenCL operations are put to the command queue. Only after this has finished, the queue is flushed, which means that the batch of commands is transferred to the device and eventually, the operations will be executed. As the command queue is ordered, it is guaranteed that when the last inserted operation finishes, all previous operations have already finished before. In order to guarantee a non-blocking task queue, all OpenCL function calls have to be asynchronous and also the other necessary functions may never block.

Furthermore, the *release* function has to be split into two parts. The first part inserts the data transfer actions to the command queue, the second part adjusts the state of the FIFO in a callback function. The callback function will only do some pointer arithmetic and is therefore lightweight as it is suggested above.

Another point that must be guaranteed is that the tasks will never be blocked while accessing an empty or full channel. Thus, a task will only be inserted to the *task queue* when it is able to execute. The important question is when a task can be inserted into the task queue. For this, the host application might provide a *monitor* to observe all *channel statuses*. When the state of the FIFO has changed due to a finished *push* or *pop* operation, the *monitor* can decide whether another process can be put to the task queue. This can be implemented to be fully event-based and all changes are broadcast using signals.

This approach therefore fulfils the following properties: Firstly, commands are processed in batches instead of transferring every command alone. As it was described, a process launcher will enqueue all data transfers and its kernel invocation before the command queue is flushed. This amortises the setup overhead and will therefore be more performant [24, Sec. 4.5.6, p.4-21]. The command queues will also contain as many actions as possible. Secondly, all OpenCL function calls are executed from within the same POSIX thread sequentially, i.e. the *task queue* selects and discharges tasks one by one. This approach is less error-prone, reduces the synchronisation costs, and also the task switching overhead. Thirdly, the callback functions are only used when needed and are implemented such that they will finish promptly. As it was mentioned, they are only needed to adjust the FIFO state. It is important that those functions are lightweight, because they will usually take resources of the OpenCL driver. It has been observed that those callback functions were all executed from the same threads that also handle all other OpenCL related actions (e.g., kernel invocations, and data transfer initialisation). Therefore it makes sense not to overload callback functions with too much work. Fourthly, the task queue has full control to decide which process it will invoke next. In this work, a *first-come first-serve (FCFS)* scheduler was implemented. It is however implemented such that it can be modified to use any dynamic scheduler (for example priority-based scheduling, or deadline-based scheduling). The strategy when a task should be launched can also be based on the underlying process network. It might happen that some long running processes are waiting for the result of a previous task. In this case such a process should be prioritised instead of handling other less important tasks first. However, for many applications the FCFS scheduler might be a reasonable choice as it treats all tasks equally.

One property that is not fulfilled by this approach is the possibility of parallel communication and process execution. It can theoretically be achieved by using multiple command queues in a round robin fashion. In practice this has however caused more problems than it actually solved. One of them is that

multiple command queues will share their work and therefore, each queue might be shorter than if only one queue is used. In OpenCL however it is recommendable to have larger queues in order to maximise the throughput. The queue selection strategy is implemented by the OpenCL driver. So, multiple command queues will give away some of the control that was earned with this approach. Furthermore the queues must be synchronised to guarantee the order of how the tokens are pushed to the FIFOs. This might be achieved by using events, which might cause additional overhead and complexity. Moreover it might happen that a push operation with a high priority (e.g., all other processes are waiting for that piece of data) is unnecessarily deferred and a less important action is executed first (e.g., a pop operation). It has also turned out that not every device supports concurrent read, write, and execution. For all these reasons, the single-queue solution was finally retained.

In summary, one can say that the task queue based approach using a single queue represents the best compromise of simplicity, efficiency, and performance. Chapter 5 discusses how a runtime environment can be built based on that task queue model.

4.5. Fine-Grained and Coarse-Grained Data-Parallelism

The concept of *shadow copies* was introduced in Section 4.2. It describes how a sequential process of an SDF graph can be transferred to many simultaneously executed processes with the same functionality in a data-parallel manner. The concept is a compromise concerning process granularity. It was mentioned that a process should describe a larger function consisting of multiple instructions, as the overhead introduced by implementing a process as an OpenCL kernel is much bigger than just a C-function call. The OpenCL device has to do some preparation tasks before launching a kernel. Afterwards, the kernel has to be transferred to the device that it is mapped to. This introduces an overhead which can be compensated with longer running kernels.

A more computationally intensive function requires most probably also more input data and is producing output data tokens of a larger size. Shadow copies can only provide data-parallelism on the level of granularity that is given by the process. The input channel must contain enough tokens, before a process with shadow copies can be launched. The more shadow copies the process uses, the bigger the latency that is introduced for the whole application.

Furthermore, the buffer capacities have to scale with the number of shadow copies, and at some point, the available memory will be the limiting factor. Depending on the application it might make sense to have 10...100 shadow copies.

This number is mainly suitable for OpenCL devices of type CPU and *accelerator*. In order to efficiently utilise a GPU, thousands of workitems have to run simultaneously on the GPU. So, if a shadow copy was implemented as a workitem, the previous processes would have to produce thousands of tokens before the shadow copy process can be invoked. This might introduce a larger latency and might require huge FIFO capacities. At some point, the device memory of the GPU will be the limiting factor for the number of shadow copies. This leads to the conclusion, that the shadow copy approach does not suffice, and that it is not possible to fully exploit the potential of a GPU with this approach.

The shadow copy approach can be extended in two ways: Firstly, one could split the GPU into smaller subdevices that can be addressed separately by launching different kernels on them. This is however not yet possible for current GPUs with OpenCL. A milestone in this direction has been set by NVIDIA when releasing their *Kepler* GPUs supporting exactly that feature with CUDA 5.0[10]. It is merely a matter of time until the OpenCL driver from NVIDIA might support this feature, and as well until other vendors might follow this direction.

Secondly, the shadow copies can be extended with the approach of having inherently parallel processes as it was depicted in Figure 4.4. Instead of defining a shadow copy as a workitem, it is defined as a complete workgroup. A workgroup consists of a selectable number of workitems and is executed on exactly one compute unit. This extension allows a kernel to use the concepts of writing parallel code in OpenCL. One of the advantages gained by this, is that a process can use local memory and barriers for synchronisation of the workitems. The local memory allows efficient communication between the workitems within a workgroup. Shadow copies cannot make use of this, as intercommunication between them is strictly forbidden by definition. In this extension, shadow copies are used to provide enough workgroups to the compute units. When a workgroup is stalled in a global memory access, the next workgroup will be executed in the meantime.

[10]CUDA concurrent kernels: `http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html#managing-coarse-grained-parallelism`

For example AMD defines the minimum number of workgroups per compute unit to be four [24, p. 5-22]. For a device with eight compute units, a kernel should therefore be launched with at least 32 workgroups. This is in the range of shadow copies as it was proposed above. The number of workitems per workgroup should be a multiple of 64 for AMD GPUs and a multiple of 32 for NVIDIA devices. The maximum number of workitems in a workgroup can vary from device to device and has to be queried before it is set.

4.6. Summary

In this chapter, the main concepts of combining the paradigms of OpenCL and SDF were described. SDF processes have been mapped onto OpenCL kernels using a well-defined interface. It was discussed how data-parallelism can be achieved in a way that is suitable for all device types that OpenCL supports. To emulate FIFO channels in OpenCL, a channel implementation was proposed that selects the best data access strategy based on the process to device mapping. The interaction between the processes and the channels is controlled by a task activation framework that is based on a global task queue. The next chapter will discuss how the proposed approaches can be integrated into a high-level design flow targeting any platform supporting OpenCL.

5

High-Level Design-Flow for OpenCL-Accelerated Applications

This chapter presents a design flow and an automatic code generation framework for applications specified by the SDF model of computation. It allows to map SDF applications onto different architectures. The principles of Chapter 4 have been integrated into the *distributed application layer* (DAL) as an extension. The design goals of simplicity, portability, extensibility, and efficiency were followed during the implementation. *Simplicity*: the application developer should have a toolchain that simplifies software development. *Portability*: applications should be reusable for different architectures. *Extensibility*: the framework must be flexible in order to be extended in the future. *Efficiency*: The proposed framework should be lightweight and transparent.

The remainder of this chapter is organised as follows: First, an introduction to DAL is presented. Then, a short overview of the approach is given. After that, the specification of applications and architectures is detailed and the software synthesis framework is explained. Finally, the chapter concludes with an evaluation of the design goals.

5.1. Distributed Application Layer

The distributed application layer (DAL) was presented in [19, 28]. It describes a high-level design flow for mapping applications specified as process networks onto heterogeneous many-core systems. It is capable of handling multiple applications and start and stop applications at runtime as specified by a finite state machine. DAL is targeting upcoming heterogenous many-core systems, which are very likely OpenCL-enabled devices. However, so far, DAL does not support OpenCL. The extension of integrating the task queue OpenCL runtime environment into DAL is therefore reasonable.

The DAL runtime environment is a distributed application that is spread over the system. The workers of DAL are nodes that describe a shared memory system running one specific operating system. All nodes are connected through a network. The behaviour of an application is controlled by a master. A slave is running on each node to take commands from the master. The master controller knows the complete process network topology and can start and stop an application by sending the respective commands to its slaves.

5.2. Overview

The high-level design flow of DAL as considered in this thesis, is depicted in Figure 5.1. The application is specified as a process network in the format of an XML-file. It defines all processes, channels, and contains the topology information of the complete process network. On the other hand side there is a specification of the architecture. This is also defined in a separate XML-file. The architecture defines all nodes that are available, and for each node it is specified how many processors belong to it. The mapping XML file specifies the location where each process has to be executed, i.e., the process to processor assignment. For this work, it is assumed that the mapping is static and known a priori. For instance, mapping optimisation is tackled in [28]. These three XML files are the input to the next stage, the software synthesis, which produces a binary executable file that is targeted to the architecture from the specification. The functionality of the application is the same as it was specified in the process network specification. Further details are given in Section 5.4.

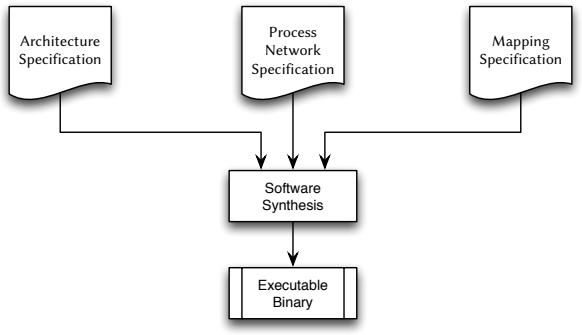


Figure 5.1.: Design flow of DAL.

5.3. Specification

This section describes how an application can be specified for DAL. The basic components are the process network, the architecture, and the mapping specification.

An application defined as a process network consists of two parts: The network topology and the functionality of all processes. The topology is specified in an XML-file such as the example that is depicted in Figure 5.2. It defines all the processes, channels, and the how they are connected.

Channels are defined with a tag called `sw_channel`. The channel capacity is fixed and must therefore be specified for all channels. For a given SDF it is possible to calculate the needed channel capacities in advance. A parameter `tokensize` is needed per channel in order to convert the notion of a token into the unit of bytes. Moreover, there are also additional optional arguments to set the token rate of both ends of the channel (otherwise they are assumed to be one).

Processes are specified with the `process` tag. The functionality of a process is defined by a source file that has to be provided by the developer. It can be specified in different languages, for example as a C-function or as an OpenCL kernel. A process can have multiple sources, in this case, the mapping will decide which one to use. For the special case of an OpenCL kernel, the inherent parallelism can be set as an optional argument `dataparallelism`. The coarse-grained parallelism can be set with an argument `shadowcopies`. This will then be used to automatically parallelise the processes as it was described.

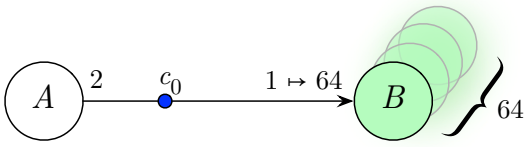
Ports are assigned to channels and processes in order to interconnect them using a `connection`. A channel has exactly one input port and one output port, each of them can only be linked once. Processes can have as many input and output ports as needed.

In the simple example in Figure 5.2, two processes are interconnected through a channel with an initial token placed on it. Process *B* is defined to either be an OpenCL kernel with 64 shadow copies, or a C-function. Process *A* writes tokens with the token rate two, whereas process *B* only reads them with a token rate one. The channel is given a capacity of 128 tokens; this allows to simultaneously execute process *A* and *B*.

The second specification is the one of the architecture. This, as well, is defined in an XML-file as in the example in Figure 5.3. It defines all the available processors of the heterogeneous platform. A processor has a certain type. For a heterogeneous platform there are multiple types of processors. In the example, there are two processors of type RISC and one processor of type GPU. The example architecture describes a PC setup with a CPU and a GPU, where two CPU cores are available to DAL. Each processor might have different capabilities. A capability describes a programming model that is implemented by DAL. In the example there are two capabilities: Executing POSIX threads, and executing OpenCL kernels. The CPU supports both of them, whereas the GPU is only able to handle OpenCL kernels. An identifier is used to distinguish one processor from other processors. For POSIX, this simply corresponds to the CPU affinity. For OpenCL the identifier is defined to be a unique string containing the vendor, the device name, the device type, and a counter (if multiple devices are equal).

The mapping is defined in a third specification depicted in Figure 5.4 and defines the location where the processes have to be executed. In the mapping XML-file, this is achieved with the `binding` tag. In the example, *process A* is mapped to *core_2* and *process B* to *gpu_0*. The binding also specifies which source file has to be executed with the `sourceid` argument. A mapping is only valid, if the processor is capable to execute the corresponding source type.

The three presented specification files are used as an input in the code synthesis stage, which is presented in the following section.




```
<?xml version="1.0" encoding="UTF-8"?>
<processnetwork name="APP">
  <process name="A" type="local">
    <port type="output" name="out1"/>
    <source id="0" type="c" location="process_A.c"/>
  </process>
  <process name="B" type="local" shadowcopies="64">
    <port type="input" name="in1"/>
    <source id="0" type="openclc" location="process_B.cl"
        dataparallelism="256"/>
    <source id="1" type="c" location="process_B.c"/>
  </process>

  <sw_channel type="fifo" tokenratein="2" initialtokens="1"
        size="128" tokensize="4"<?Bytes?> name="c0">
    <port type="input"  name="in"/>
    <port type="output" name="out"/>
  </sw_channel>

  <connection name="A_c0">
    <origin name="A">
     <port name="out1"/>
    </origin>
    <target name="c0">
      <port name="in"/>
    </target>
  </connection>

  <connection name="c0_B">
    <origin name="c0">
      <port name="out"/>
    </origin>
    <target name="B">
      <port name="in1"/>
    </target>
  </connection>
</processnetwork>
```

Figure 5.2.: A simple producer-consumer process network as XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<architecture name="ARCH">
<shared name="localhost">
    <port name="port1" />
    <port name="port2" />
    <port name="port3" />
</shared>

<processor name="core_0" type="RISC" id="1">
    <port name="port1" />
    <capability name="POSIX" identifier="1" />
    <capability name="OPENCL" identifier="CPU_32902_Intel(R)Core(TM)i7
        -2600KCPU@3.40GHz_DEV1_SUB1" />
</processor>
<processor name="core_1" type="RISC" id="2">
    <port name="port1" />
    <capability name="POSIX" identifier="2" />
    <capability name="OPENCL" identifier="CPU_32902_Intel(R)Core(TM)i7
        -2600KCPU@3.40GHz_DEV1_SUB2" />
</processor>
<processor name="gpu_0" type="GPU" id="3">
    <port name="port1" />
    <capability name="OPENCL" identifier="GPU_4098_Capeverde_DEV1" />
</processor>

<link name="link_1">
    <end_point_1 name="core_0"><port name="port1" /></end_point_1>
    <end_point_2 name="localhost"><port name="port1" /></end_point_2>
</link>
<link name="link_2">
    <end_point_1 name="core_1"><port name="port1" /></end_point_1>
    <end_point_2 name="localhost"><port name="port2" /></end_point_2>
</link>
<link name="link_3">
    <end_point_1 name="gpu_0"><port name="port1" /></end_point_1>
    <end_point_2 name="localhost"><port name="port3" /></end_point_2>
</link>
</architecture>
```

Figure 5.3.: Architecture as XML, describing a PC setup with a CPU and a GPU.

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping name="mapping" processnetwork="APP">

<binding name="producer">
    <process name="A" sourceid="0" />
    <processor name="core_2" />
</binding>
<binding name="consumer">
    <process name="B" sourceid="1" />
    <processor name="gpu_0" />
</binding>
</mapping>
```

Figure 5.4.: Mapping as XML, corresponding to a PC setup with a CPU and a GPU.

5.4. Software Synthesis

In the software synthesis stage, the application-specific code is merged with the platform-specific code based on the selected mapping. With this, it is possible to write applications independently from the architecture specific code. The architecture code can therefore be reused for various applications. In this section it is explained how this synthesis is done for the class of OpenCL-enabled architectures.

First of all, the controller threads for DAL have to be generated. This functionality is already provided by the original DAL framework. For a single node, this will generate the code to launch two threads, one for the master and one for the slave. For the OpenCL runtime environment, an additional thread is needed for the task queue that was introduced in Section 4.4. The code for all three of them is therefore generated. In case of an OpenCL enabled architecture, a unique context is generated per device for every unique identifier of the architecture XML file.

The steps so far were independent of the process network. Thus, the second step is to generate the code to setup the process network and to start it afterwards. In DAL, all the administrative operations are coordinated by the master controller. The master knows the topology of the process network and the corresponding mapping. It sends commands to its slaves commanding them to install processes and channels, and to start and stop the processes. In order to integrate the OpenCL runtime environment to DAL, the interface to install a process had to be extended to contain information about the type of the process and which source file belongs to it. This means, a process can be of type POSIX thread, or it can be an OpenCL kernel. Moreover, the inter-

face specifies to which location the process is mapped onto. For an OpenCL process this location denotes an OpenCL context. Furthermore, information about the number of shadow copies and the parallelism of the OpenCL kernel had to be inserted to that interface. When the slave receives a command to install an OpenCL process at runtime, it will load the source file of the process and compile the corresponding kernel for the corresponding device. When the slave receives the command to install a channel, it will create a primitive channel structure.

Finally, when all processes and channels are installed, the master will send the commands to link them. When a channel is linked to the second process, it allocates the memory dynamically in the suitable device memory as it was described in Section 4.3.4. There is yet another variant of a channel port which was not discussed in detail: A channel port connecting POSIX threads to the same channels as the OpenCL kernels which allows to create applications that integrate multiple types of processes. This opens the world to a much bigger class of applications. For example OpenCL kernels cannot read from or write to file streams, which is something that can easily be done within a POSIX thread.

The last step is to start the process network. After the slave has received the command to do so, it will insert the task to start a process into the task queue after checking if the process can be executed. Otherwise, the process will be directly triggered by the network at the point when it can be executed (but again through the task queue).

Somewhen, the application has to be stopped. Every POSIX process holds an event channel to the master and can ask the master to stop the application at some point in time. The master will then initiate the shutdown procedure. This comprises of stopping the processes and finally destroying the channels. The implementation of stopping is straight-forward, and thus omitted.

5.5. Design Goals

This section lists the design goals that were followed while integrating the OpenCL runtime environment into DAL. These are namely simplicity, portability, extensibility, and efficiency.

Simplicity

The application programmer should focus on the application and not spend its resources to tedious things as communication and synchronisation of the application. Simplicity is achieved by specifying the application in a simple model called SDF. The extension to DAL allows to specify an application in an XML file. The functionality of the processes has to be programmed as simple sequential processes. Each process can only read from and write to its channels which makes programming easy, as the programmer does not need to take care about how the data is transferred and is also relieved of the usual parallel programming difficulties, such as synchronisation, indeterminism, and dead-locks.

Portability

Applications can be created for various different architectures and setups. The code for the process network only has to be written once and can then be ported to all architectures supporting OpenCL. Following the Y-chart approach, a new architecture can be created independently of the application, and only a simple XML file has to be provided. The framework can then synthesise that new architecture and merge it with the code of the process network.

Extensibility

The proposed solution is extensible in multiple dimensions. First of all, it is possible to integrate additional capabilities for processors (e.g., CUDA for NVIDIA GPUs). A process can already contain multiple sources and a processor can have multiple capabilities. So the baseline is set to extend this even more. Second, the proposed hybrid channel implementation can also be used to implement other channel port types. Third, the task queue is really flexible. It can, on one the hand, be modified to any dynamic scheduler, and on the other hand, it is possible to implement other tasks that can be handled. It is also possible to use multiple task queues. All those properties make the proposed framework an extensible one.

Efficiency

A channel will only allocate its memory at runtime and decide where to store the FIFO. The design goal of efficiency led to zero-copy FIFOs wherever this is possible. In the other cases, the number of copies is minimised. Furthermore, the implementation will carefully reuse OpenCL buffer objects and kernels. The objective os to have a runtime-system that is transparent and only creates minimal overhead.

5.6. Summary

This chapter has presented a design flow to efficiently write parallel software. This software synthesis framework is afterwards used to evaluate the runtime environment in different dimension by creating various applications suitable to benchmark one aspect of the framework. The evaluation is described in the following chapter.

6 Evaluation

A number of concepts have been proposed in Chapters 4 and 5. This chapter evaluates the performance metrics that can be reached with them. First, the proposed channel implementations are compared by measuring the maximum data transfer rate. After that, in Section 6.3, the two different task activation frameworks are compared. Then, the overhead introduced by OpenCL is analysed in Section 6.4. Finally, a realistic application is used to compare different devices for their parallel computation performance. The following section lists the evaluation setups that are used for the benchmarks.

6.1. Evaluation Setup

The benchmarks have been conducted on two different heterogeneous systems. Setup 1 is a desktop PC with two graphics cards (Table 6.1), whereas setup 2 is a Laptop with a mobile CPU and graphics card (Table 6.2).

Setup 1	
CPU	Intel Core i7-2600K @ 3.40 GHz, 8MiB Cache
	Driver: Intel OpenCL RTE 2013 XE V.3.0.67279_x64
GPU1	AMD Radeon HD 7750
	Driver: catalyst-opencl 13.1-1
GPU2	NVIDIA Geforce GTX 670
	Driver: nvidia-opencl 313.18-3
Operating System	Arch Linux 3.7.6-1-ARCH x86_64

Table 6.1.: Evaluation setup 1.

Setup 2	
CPU	Intel Core i7-2720QM @ 2.20GHz, 6MiB Cache
	Driver: Intel OpenCL RTE 2013 XE V.3.0.67279_x64
GPU	NVIDIA Corporation GF106 [Quadro 2000M] (rev a1)
	nvidia-opencl 319.17
Operating System	Ubuntu 12.04.2 LTS 3.5.0-28-generic x86_64

Table 6.2.: Evaluation setup 2.

6.2. Intra- and Inter-Device Communication

The different channel implementations are evaluated with a synthetic benchmark. The process network in Figure 6.1 consists of two processes and a channel. Process A produces one token of t bytes in each invocation. This token is then transferred to process B using a channel with capacity c. Process B will eventually receive the token and access all bytes once to store it in a temporary variable. This guarantees that the data has to be transferred into a data register of the corresponding device, and reflects the behaviour of a realistic application. As there there are only minimal calculations performed on the data (a simple addition of a constant), this test will give an upper bound on the available data transfer rate. For the evaluation, the following parameter space is explored:

- The token size t is varied in the range of 512 bytes and 4 MB.
- The number of shadow copies (S) is set to the number of compute units of the corresponding device.
- Different values for intra-process parallelism (I) are compared.

- The channel capacity c is fixed to 32 MB.
- Process A and B are mapped to either the same or to different devices.
- Different channels have been used to interconnect processes A and B.

In a first setup, the data transfer rate between two processes mapped onto the same device is measured for two different FIFO implementations. The results for setup 1 are shown in Figures 6.2a and 6.2b for the NVIDIA and the AMD GPU, respectively. First, the optimised FIFO implementation is used, which keeps the data in the global memory of the device ("global buf."). Second, a naive FIFO implementation is used, which transfers the data through host memory ("HAM buf."). Having more workitems might lead to higher data transfer rates as more processing elements can concurrently read and write. The observed peak data rate is 20.30 GBytes/s when both processes are mapped onto the NVIDIA GPU and 7.96 GBytes/s when both processes are mapped onto the AMD GPU. As expected, the data transfer rate is considerably lower if the memory buffer is allocated in the host memory. In this case, the observed peak data rate is 1.09 GBytes/s when both processes are mapped onto the NVIDIA GPU and 1.67 GBytes/s when both processes are mapped onto the AMD GPU.

The data transfer rate for the case that one process is mapped onto the CPU and the other process is mapped onto the AMD GPU is illustrated in Figure 6.2c. "GPU to CPU" means that the producer process is mapped onto the GPU and the consumer process to the CPU. For "CPU to GPU", it is vice versa. The observed peak data rate is 1.91 GBytes/s when the producer process is mapped onto the GPU and 2.14 GBytes/s when the producer process is mapped onto the CPU.

Finally, Figure 6.2d shows a summary of the data transfer rates for setup 2. The observed peak data rate is 3.82 GBytes/s and measured when both processes are mapped onto the GPU.

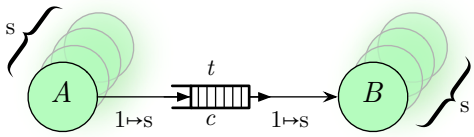
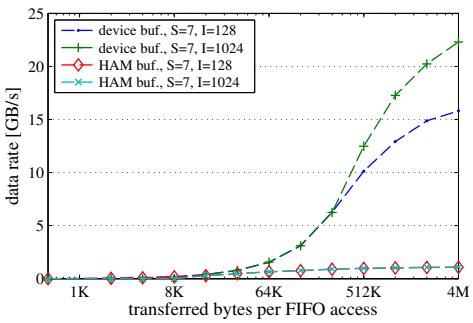
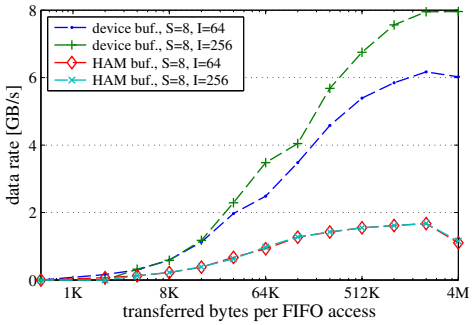


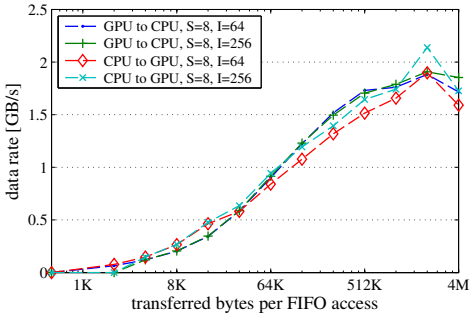
Figure 6.1.: A producer-consumer model for measuring the reachable throughput with parameterised shadow copies s, token size t, and channel capacity c.



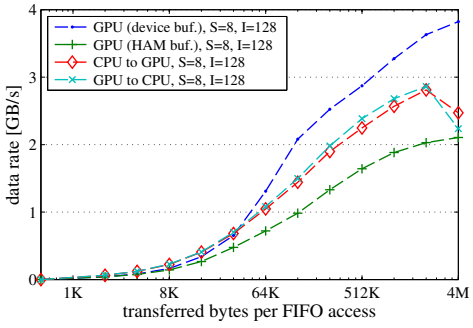
(a) Platform A, NVIDIA GPU.



(b) Platform A, AMD GPU.



(c) Platform A, AMD GPU to CPU.



(d) Platform B.

Figure 6.2.: Data transfer rate for different process mappings and channel implementations.

6.3. Comparison of Task Activation Frameworks

Two different implementations of a task activation framework were introduced in Section 4.4. Both of them, the direct approach and the task queue, are evaluated in the following. The example application in Figure 6.3 is used to benchmark their performance. Process A produces a token that is afterwards consumed by process B. Figure 6.4 shows the resulting speed-up when using the task queue mechanism relative to the execution time when using the direct callback mechanism. Both processes are mapped onto the CPU and the number of calculations in process A is varied. The x-axis represents the invocation frequency when using the direct callback mechanism. As expected, no speedup is achieved for low invocation frequencies. For higher invocation frequencies, the task queue mechanism is slower than the direct callback mechanism if the FIFO channel has a capacity of one token. In this case, both mechanisms can enqueue a new iteration only if the previous iteration is completed. However, the feedback loop is larger for the task queue mechanism as it has the additional component of the runtime-manager. If the FIFO channel has a capacity of more than one token, the task queue mechanism can enqueue a new iteration in parallel to the execution of the old one so that the task queue mechanism is faster than the direct callback mechanism. Finally, for very large invocation frequencies, the iteration completes earlier than the task queue mechanism can enqueue a new iteration so that the speedup declines again.

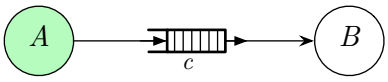
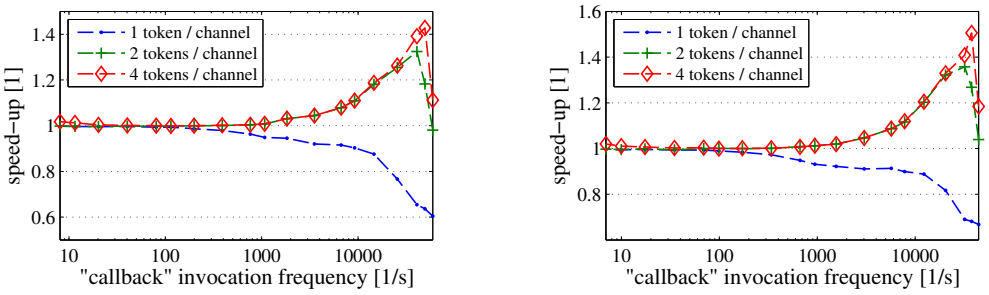


Figure 6.3.: Application for evaluating the performance of the proposed task activation framework.



(a) Setup 1. *(b) Setup 2.*

Figure 6.4.: Speed-up of the application in Fig. 6.3 when using the task queue mechanism relative to the execution time when using the direct callback mechanism. The x-axis denotes the invocation frequency when using the direct callback mechanism.

6.4. Overhead of OpenCL versus POSIX Threads

The next benchmark evaluates the overhead that is introduced by OpenCL. Figure 6.5 depicts the application that was used. Again, a simple producer-consumer application is used consisting of two processes A and B connected through a channel. Process A produces a token in each iteration, that is afterwards consumed by process B. The channel capacity was fixed to contain space for two tokens. This allows to execute neighbour processes simultaneously. The application was synthesised twice, the first resulting application is executing processes as POSIX threads, whereas the second is executing processes as OpenCL kernels. When synthesising the application for POSIX, either no optimisation, optimisation level O2, or optimisation level O3 with setting `march=native` is used. When `march=native` is set, G++ automatically optimises the code for the local architecture. When synthesising the application for OpenCL, the number of shadow copies is set to one so that each process is executed on exactly one core.

Figure 6.6 shows the speedup of the OpenCL and the optimised POSIX implementations versus the execution time of the unoptimised POSIX implementation, with the number of calculations in process A being varied. The x-axis represents the iteration period of the unoptimised POSIX implementation. As the kernel invocation overhead in OpenCL is virtually independent of the kernel's amount of work, the POSIX implementation performs better for small iteration periods. On the other hand, the overhead is less crucial for longer iteration periods and OpenCL implementations achieve even higher

speedups than the optimised POSIX implementations. This may be due to OpenCL's ability to utilise the CPU's vector extension so that four workitems are executed in SIMD fashion. That assumption is supported by the fact that the CPU of both setups is only able to execute four workitems in parallel and distributing the work on five workitems is counterproductive. Note that G++ also makes use of the AVX commands[1] when the corresponding option is enabled, which is why the O3 speedup is always higher than with OpenCL and I=1.

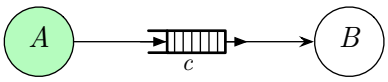
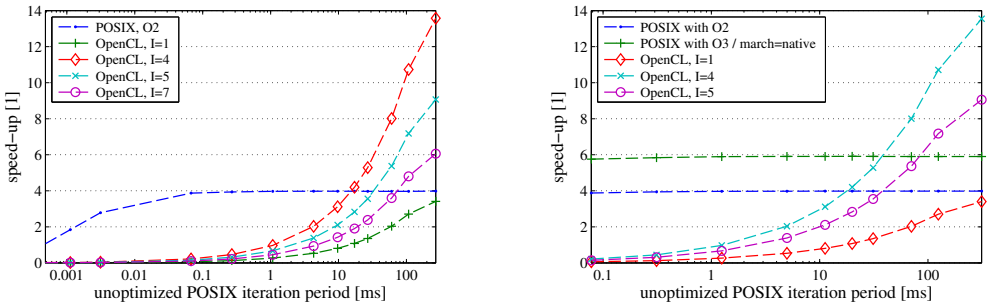


Figure 6.5.: Benchmark application that was synthesised as a POSIX implementation and as an OpenCL implementation.



(a) Setup 1. *(b) Setup 2.*

Figure 6.6.: Speedup of the OpenCL and the optimised POSIX implementations versus the unoptimised POSIX implementation. I denotes the level of intra-process parallelism.

6.5. Exploiting Task- and Data Parallelism

Different classifications of parallelism have been introduced during this thesis. This section evaluates how task-, pipeline-, and data parallelism can be used for accelerating an application. A special focus is set on the introduced notions of shadow copies and intra-process parallelism. The test consists of a realistic application, which describes a motion-JPEG decoder with a filter cascade that

[1] `http://software.intel.com/avx`

is applied to the raw image after decoding. The application is depicted in Figure 6.7. The *Split* process reads a video file stream from the main memory and splits the work to be forwarded to three decoders *Dec*. Those are decoding the JPEG frames to a raw image. The merger process *Merge* reassembles the raw images and forwards them to the next stages. Afterwards, a filter cascade of a 15x15 *Gauss*, and two 3x3 *Sobel* filters are applied to the image. The former blurs the image to get rid of noise, the latter highlights the edges that are found in the image by calculating the gradient magnitude of the image in x and y direction. Then, the *Sobel*-process forwards the same resulting image twice to the next stage. *Opticalflow* calculates the motion in both image directions and creates an output image that shows the magnitude of movement for every region in the original image. Finally, the *Screen* process will output the original and the processed image to the screen. For reasons of comparability, the *Screen* process was only consuming its input tokens without displaying it during the benchmarks. The *Dec*, *Split*, *Merge*, and *Screen* processes are written as C-functions and run as POSIX threads with a static mapping for all benchmarks. The *Gauss*, *Sobel*, and *Opticalflow* processes are provided as OpenCL kernels. Between the benchmarks, their mapping was changed in order to compare the performance of different devices. However, for all benchmarks, the three processes were always mapped to the same device. This allows to have zero-copy buffers between them, which turned out to be the fastest channel. This evaluation will therefore focus on varying numbers of shadow copies (S) and also on different numbers for intra-process parallelism (I). As a metric, it is measured how many frames per second (FPS) can be achieved in the *Screen* process. Every benchmark was executed for 5000 frames.

Figure 6.8 shows the frame rates achieved with different configurations on setup 1. Multiple configurations are shown on the x-axis. The baseline is given by configuration on the left, where only one CPU core is used for the whole application. In this configuration, there is essentially no possibility for exploiting any of the parallelism types. In this case 42 FPS are the maximum that can be achieved. Setting the intra-parallelism to a multiple of four will increase the frame rate to 62 FPS by benefitting of the SIMD vector processing units of the CPU. In the second configuration, the processes are distributed to all cores of the CPU and a frame rate of 156 FPS can be achieved. For this, the number of shadow copies (S) was set to three, as it has been observed, that the intel driver uses three operating system threads to emulate the compute units (using four shadow copies turned out to be slower as well). Compared to the single core configuration, this corresponds to a speedup of 2.5x. It must be said, that the load balancing of the cores was not perfect, because the OpenCL

worker threads were only utilising three of the cores, and unfortunately it was not possible to control their affinity during runtime. Given that only three cores were fully used (296% CPU usage measured with the `time` command), the speedup of 2.5x can be considered as a good result.

For the other configurations, the computationally most intensive processes are executed on one GPU, namely, *Gauss*, *Sobel*, and *Opticalflow*. With this configuration, it is possible to reach 2347 FPS, which is again limited by the CPU (in this case the decoder is the limiting factor). For this application, it is therefore possible to reach a speedup of 55x by fully leveraging the hardware. There is still room to achieve even better results, for example by using both GPUs. For the configuration using the AMD graphics card, increasing both, the number of shadow copies, and the intra-parallelism, is beneficial and results in a higher frame rate. In contrast, the NVIDIA GPU does mostly benefit from increasing the number of shadow copies, whereas increasing the intra-process parallelism is even counterproductive in certain cases.

Figure 6.9 shows the frame rates for different configurations on target platform B. The peak performance (931 FPS) is achieved when all cores of the CPU and the GPU device are available. It constitutes a speed-up of 25x compared to the case where all processes are mapped onto one CPU core (without using the SIMD units of the CPU). The plot also shows that the number of workgroups should be aligned with the available hardware. It has been observed that the Intel OpenCL SDK version that was used is distributing the OpenCL kernels to only three cores, which is why a higher frame rate is obtained when executing three shadow copies instead of four.

Overall, the results demonstrate that the proposed framework provides developers with the opportunity to exploit the parallelism provided by state-of-the-art GPU and CPU systems. In particular, speedups of up to 55x could be measured when outsourcing computation intensive code to the GPU.

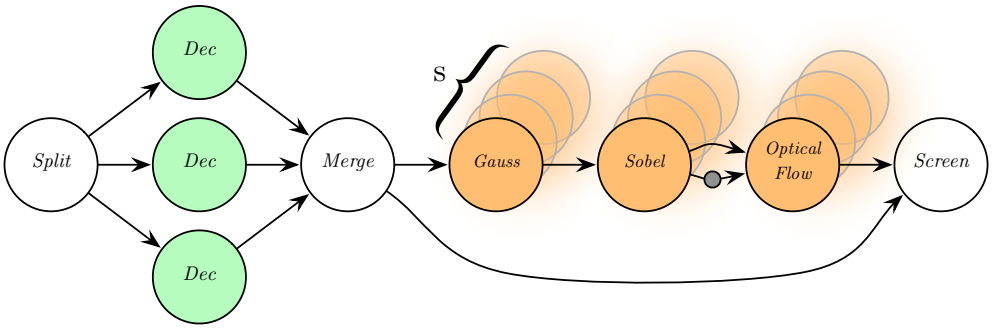


Figure 6.7.: Realistic benchmark application describing an MJPEG decoder and a filter cascade that is applied to raw images.

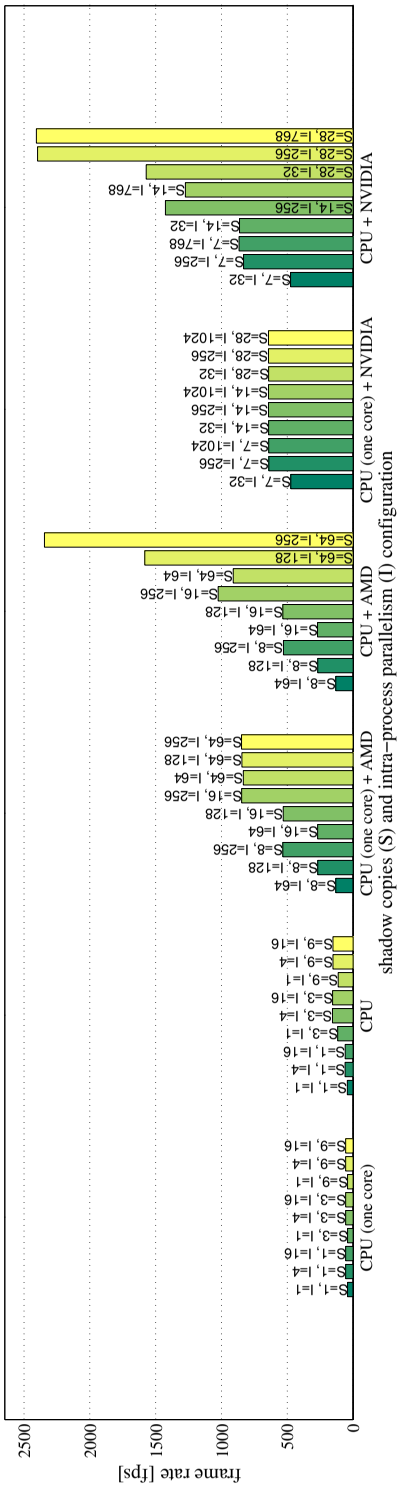


Figure 6.8.: MJPEG evaluation result on setup 1.

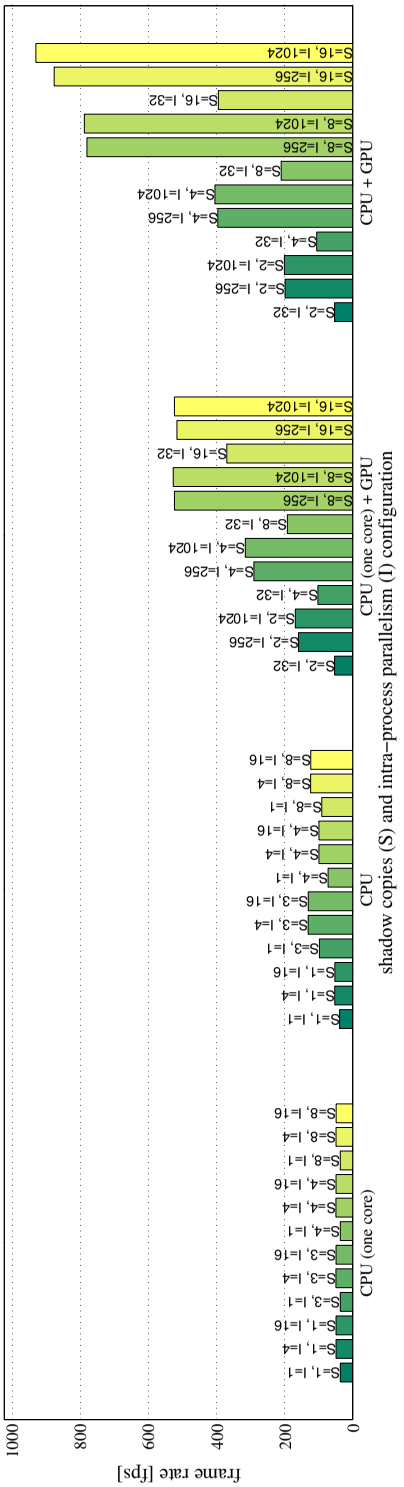


Figure 6.9.: MJPEG evaluation result on setup 2.

6.6. Summary

In this chapter, the core components of the proposed runtime-system were evaluated. First, the communication channels were benchmarked to give an upper bound on the achievable data transfer rates. The results have proved that the buffer placement strategy is remarkably better than the naive implementation of using the host memory for every transaction. It has also shown, that it makes sense to use shadow copies and then to transfer data in bigger blocks leading to a higher data transfer rate.

Second, the task activation frameworks have been compared, and, as expected, the direct callback approach is working better when the buffer capacity is limited to the size of one token. For a larger buffer, the task queue approach is faster, because it can keep the OpenCL command queue busy.

Third, the overhead introduced by the whole runtime-system has been compared to a native POSIX implementation. The results have shown that the OpenCL implementation can be faster than the POSIX implementation. For short-running processes the overhead introduced by OpenCL results in a much slower result, as expected.

Fourth, a realistic video processing application has been used to benchmark the overall performance gain. In comparison to only using one CPU core, a speedup factor of 55x was achieved by utilising all CPU cores and one GPU device. This proves, that the framework offers a lot of parallelisation opportunities to the programmer, which allows to efficiently use the underlying hardware. The complete video processing application was only written once, and was then synthesised for the different configurations by changing the mapping. This confirms that software development is simple and productive with the proposed framework.

7 Conclusion and Outlook

7.1. Conclusion

Motivated by upcoming heterogeneous systems, this master thesis proposed a design flow and runtime-system for executing process networks on on top of an OpenCL environment. It was elaborated how the notion of synchronous data flow graphs can be mirrored by means of OpenCL. A flexible runtime-system implementing the building blocks of a process network has been proposed. For that, an OpenCL kernel was abstracted to mimic the behaviour of a synchronous data flow process. Furthermore, location-based FIFO channels minimising the number of copy operations have been developed. Multiple channel implementations have been evaluated and finally been composed into a hybrid implementation combining all advantages. Those hybrid channels enable inter-communication between OpenCL kernels. Additionally, they can also be used to communicate with processes that are executed natively on the CPU within a POSIX thread. Moreover, an extensible task activation framework was introduced by the notion of a task queue. It can predictably manage process invocations and therefore be used as a scheduler.

Supporting productive software development for heterogeneous systems, a design flow consisting of two separate specifications has been proposed. The application is specified in a high-level language consisting of the process net-

work topology and the implementation of the processes. The second specification defines the hardware architecture in a high-level language and is independent of the application. A program synthesis framework builds specific code targeted for one of the defined architectures by combining the building blocks of the runtime system with program-specific code. This approach offers portability to develop applications once and deploy it on a wide range of OpenCL architectures.

Extensive evaluations have proved that software development is largely simplified with the proposed framework. Development of the benchmark applications has turned out to be an efficient task and that it was indeed effortless to program them. The evaluation has also compared the performance of a realistic video processing application. The achieved frame rates have been compared for different process to device mappings. In comparison to only using one CPU core, a speedup factor of 55x was achieved by distributing the application over all available CPU cores plus one GPU device.

Concluding, it can be said that the design goals of simplicity, portability, extensibility, and efficiency were truly met by the proposed framework supporting productive software development for heterogeneous systems.

7.2. Outlook

The proposed framework being extensible, there are multiple ways to modify it and therefore expand it. To continue with, the task queue can be extended to any dynamic scheduler to integrate load-balancing or to optimise the response time of applications. So far, the task queue does not use all of its power and does merely set on fairness to treat all processes equally.

Moreover, the runtime-system and the architecture specification could be modified to support a distributed OpenCL environment. The baseline for this extension is already set by the distributed application layer and should therefore be straight-forward.

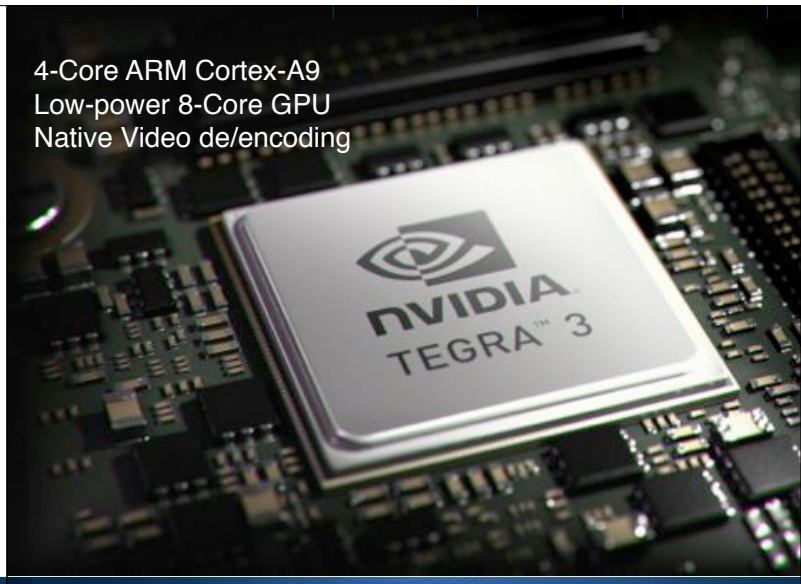
As this thesis was motivated by upcoming heterogeneous systems, it is reasonable to port and evaluate the framework on such multi-processor system-on-chips (although being a homogenous example, a reasonable candidate might be the STHORM platform [7] which natively supports OpenCL).

A

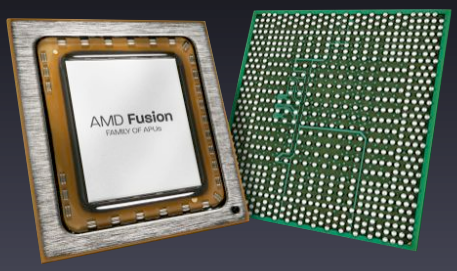
Presentation Slides

Executing Process Networks on Heterogeneous Platforms using OpenCL

4-Core ARM Cortex-A9
Low-power 8-Core GPU
Native Video de/encoding



AMD Fusion
Multicore CPU
combined with a GPU on the same die



Portability offered by OpenCL

OpenCL

AMD
ARM
NVIDIA
intel
IBM
life.augmented

OpenCL is a Complex Toolbox for Experts

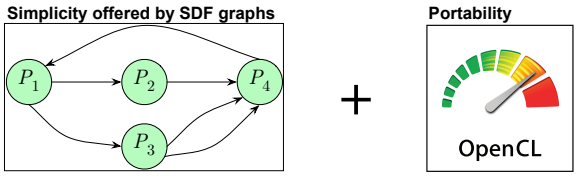
Outline

- Introduction
  - Motivation
  - **Related Work**
  - Problem Description
- Approach
- Evaluation

Related Work

- Simplification of OpenCL
  - Oak Ridge National Laboratory - Project Maestro
  - AMD and Northeastern University, Boston
- Process Networks on GPUs:
  - Seoul National University
  - University of Leiden
- High-level Languages:
  - IBM Research - Project Liquid Metal with Lime Programming Language

Problem Description



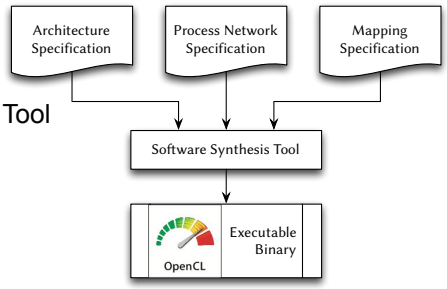
Design Goal ➔ **Efficiency**

Outline

- Introduction
- Approach
  - **Overview and Contributions**
  - OpenCL Terminology
  - Runtime-System
- Evaluation

Overview and Contributions

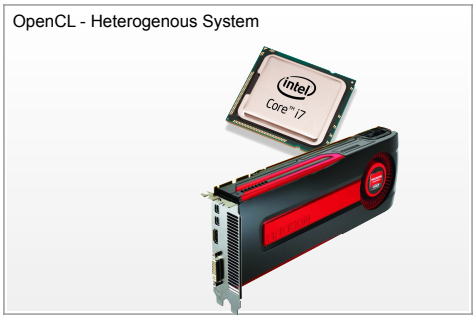
- Application Programming Interface
- Runtime-System
- Software Synthesis Tool



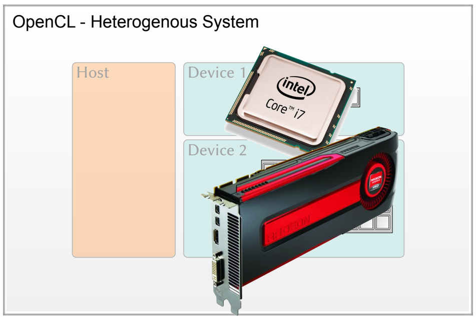
Outline

- Introduction
- Approach
  - Overview and Contributions
  - **OpenCL Terminology**
  - Runtime-System
- Evaluation

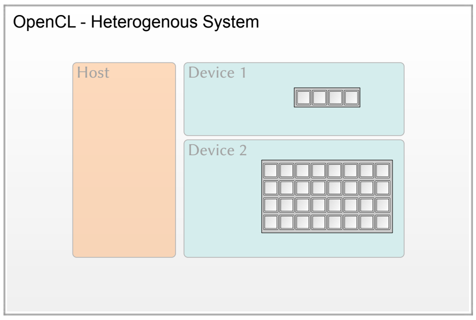
OpenCL Terminology - Platform Model



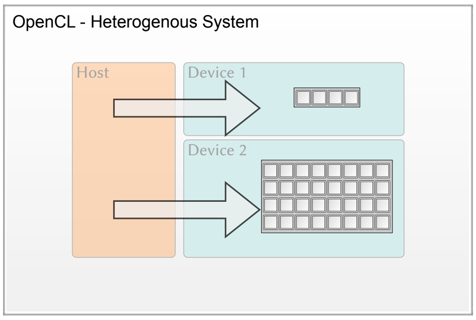
OpenCL Terminology - Platform Model



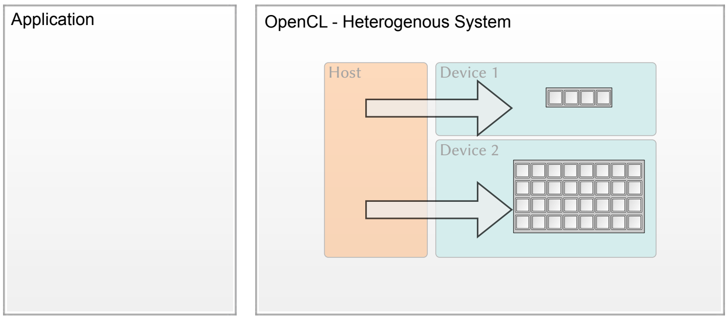
OpenCL Terminology - Platform Model



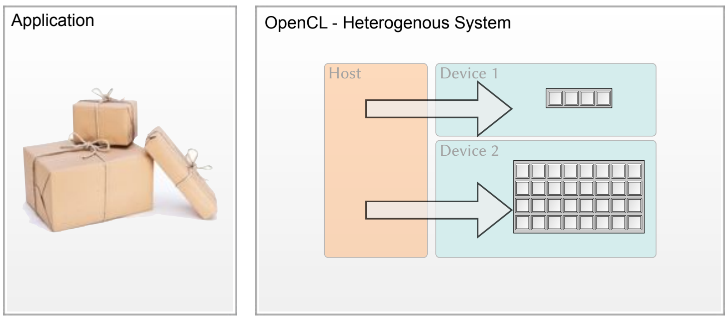
OpenCL Terminology - Platform Model



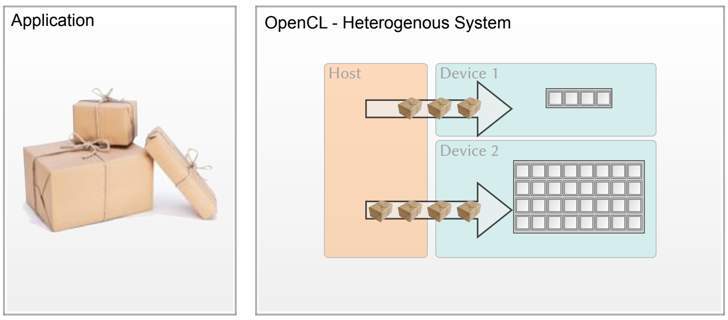
OpenCL Terminology - Execution Model



OpenCL Terminology - Execution Model



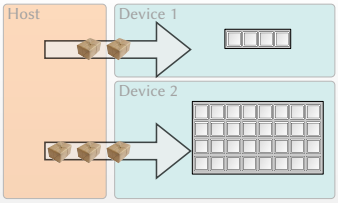
OpenCL Terminology - Execution Model



OpenCL Terminology - Execution Model

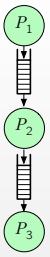
Application

OpenCL - Heterogenous System

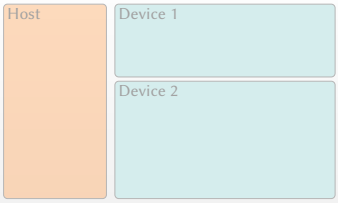


Runtime-System

Application



OpenCL - Heterogenous System



Runtime-System - Overview

Application

Heterogeneous System

Runtime-System - Overview

- SDF Process Emulation

Application

Heterogeneous System

Runtime-System - Overview

- SDF Process Emulation

- Communication Channels

Application

Heterogeneous System

Runtime-System - Overview

- SDF Process Emulation

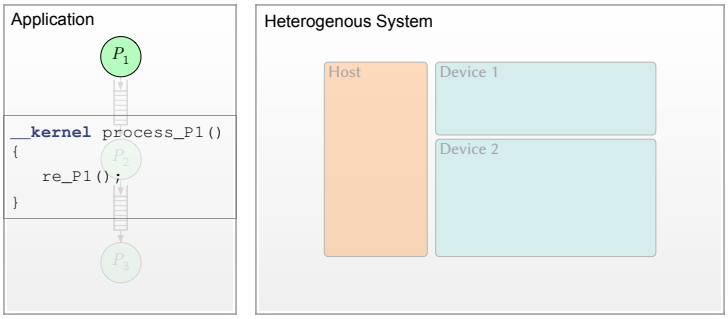
- Communication Channels

- Parallelism

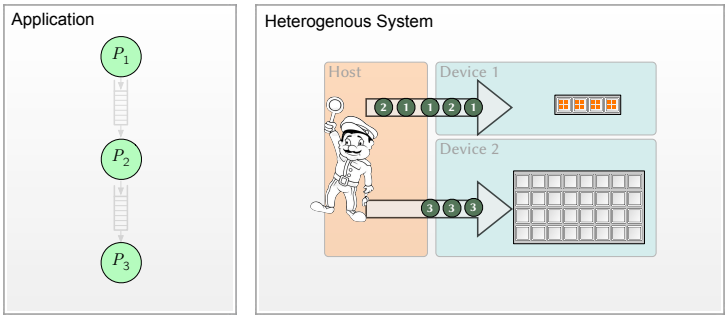
Application

Heterogeneous System

Runtime-System - SDF Process Emulation

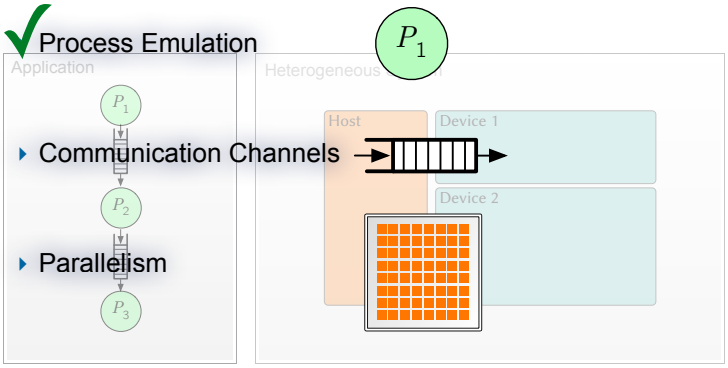


Runtime-System - Process Execution

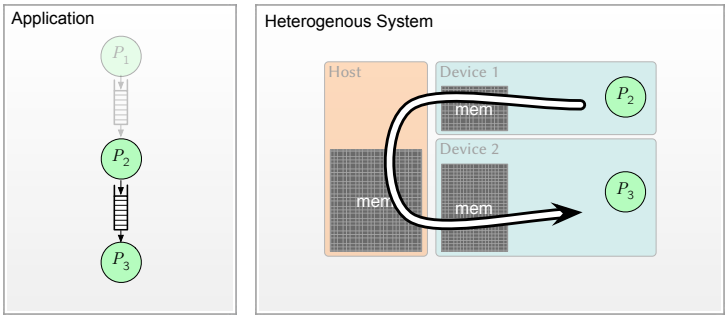


Runtime-System - Overview

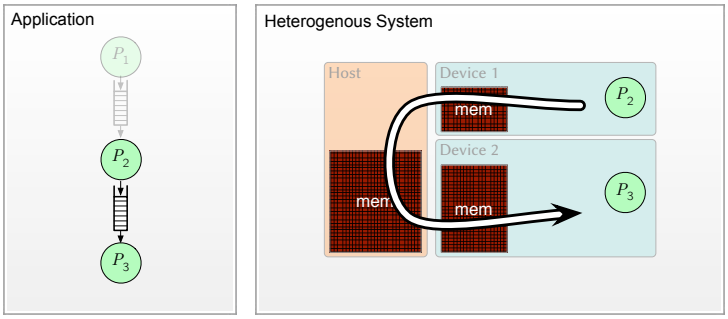
- ✓ Process Emulation
- Communication Channels
- Parallelism



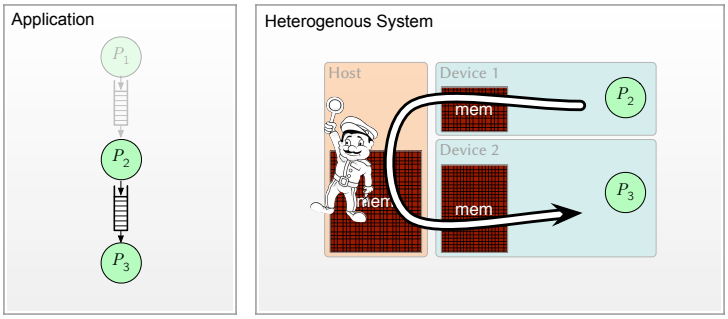
Runtime-System - Channels



Runtime-System - Channels

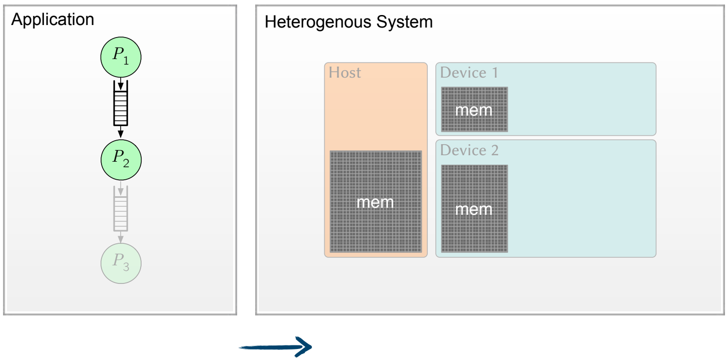


Runtime-System - Channels

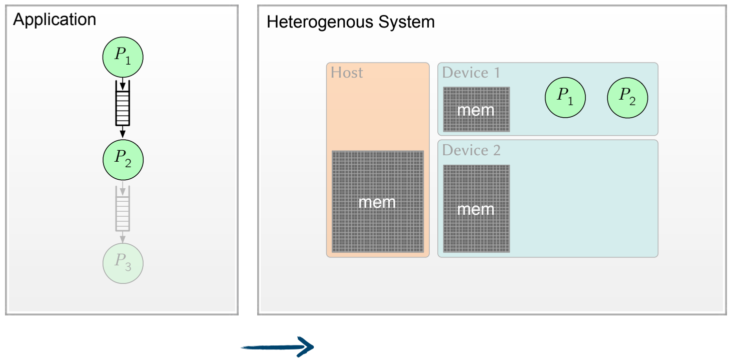


→ Triple Buffering

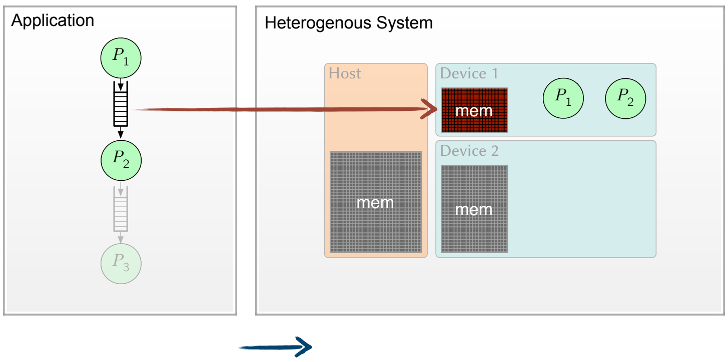
Runtime-System - Channels



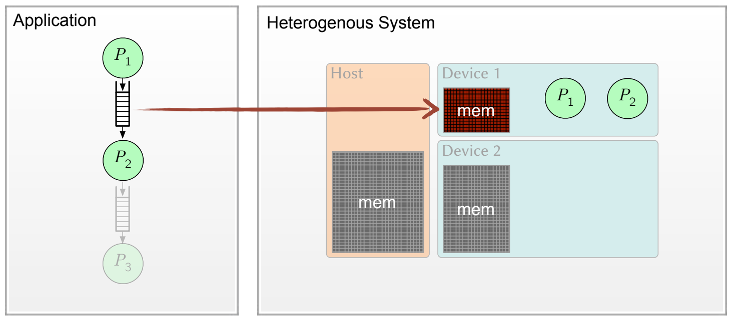
Runtime-System - Channels



Runtime-System - Channels



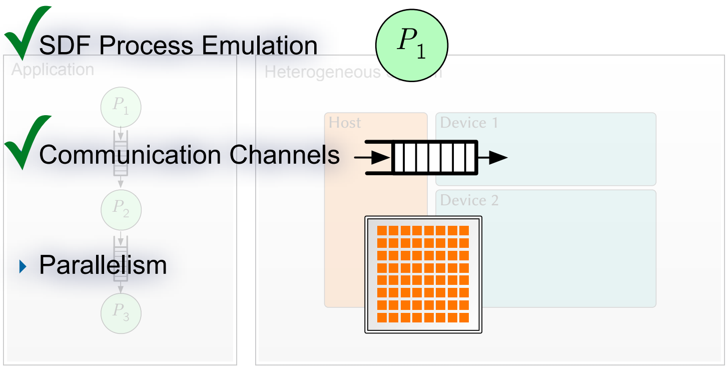
Runtime-System - Channels



→ **Zero-Copy-Buffer**

Runtime-System - Overview

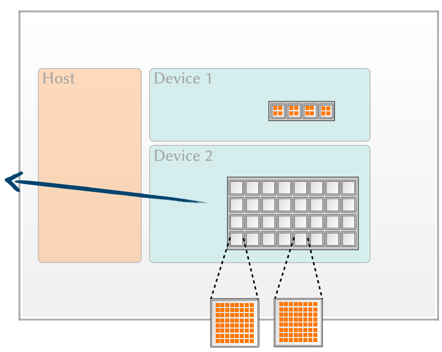
- ✓ SDF Process Emulation
- ✓ Communication Channels
- Parallelism



Data-Parallelism offered by OpenCL

2048 Processing Elements

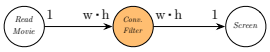
?!



Data-Parallelism - Convolution Filter

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

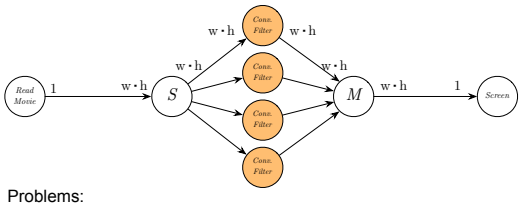
Data-Parallelism - Convolution Filter



Wednesday, June 12, 2013 | Tobias Scherer | 23

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

Data-Parallelism - Splitter / Merger

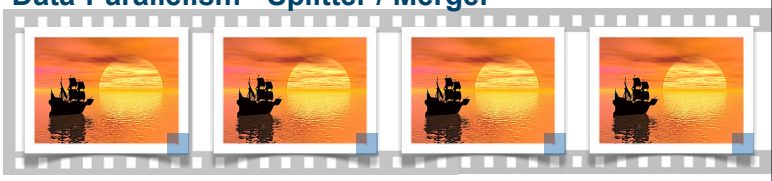


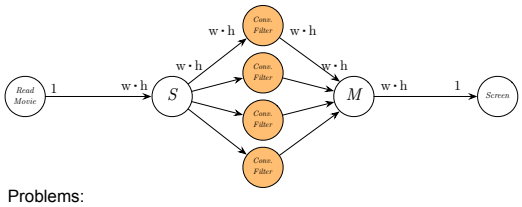
➔ Problems:

- **Network Topology** must be adapted: **Complexity**
- Needs more **Resources** (Splitter, Merger, Channels)

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

Data-Parallelism - Splitter / Merger





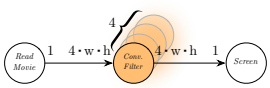
➔ Problems:

- **Network Topology** must be adapted: **Complexity**
- Needs more **Resources** (Splitter, Merger, Channels)

Wednesday, June 12, 2013 | Tobias Scherer | 24

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

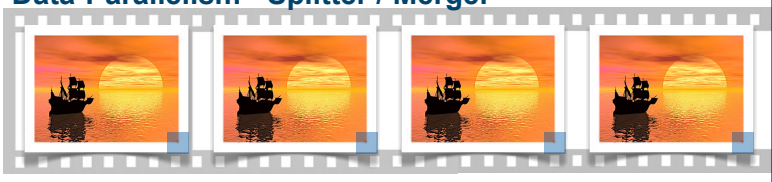
Data-Parallelism - Splitter / Merger

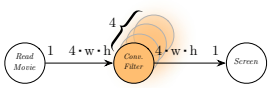


Wednesday, June 12, 2013 | Tobias Scherer | 24

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

Data-Parallelism - Splitter / Merger





⟶ Shadow Copies

Problems:

- **Latency**
- **Buffersize**

Wednesday, June 12, 2013 | Tobias Scherer | 24

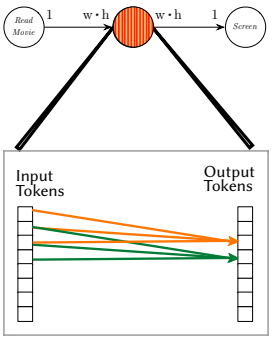
Data-Parallelism - Intra-Process Parallelism

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

Data-Parallelism - Intra-Process Parallelism

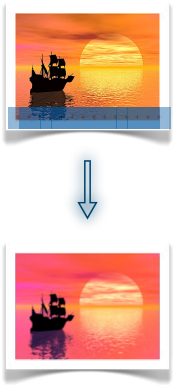
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

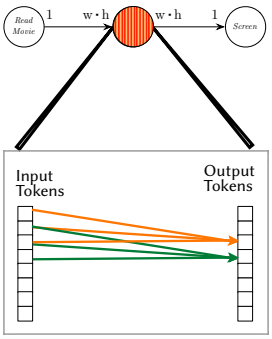
Data-Parallelism - Intra-Process Parallelism



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

Data-Parallelism - Intra-Process Parallelism

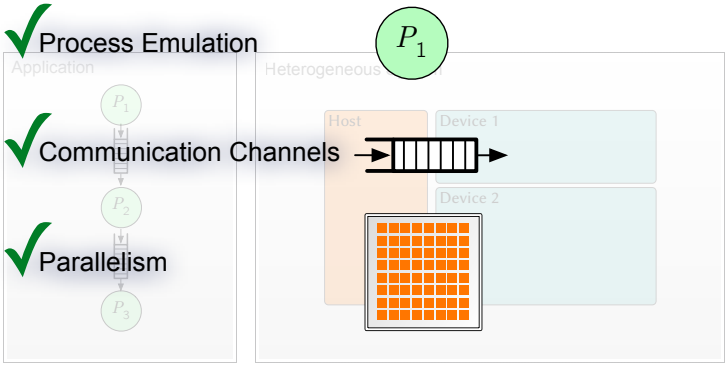




ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

Runtime-System - Overview

- ✓ Process Emulation
- ✓ Communication Channels
- ✓ Parallelism

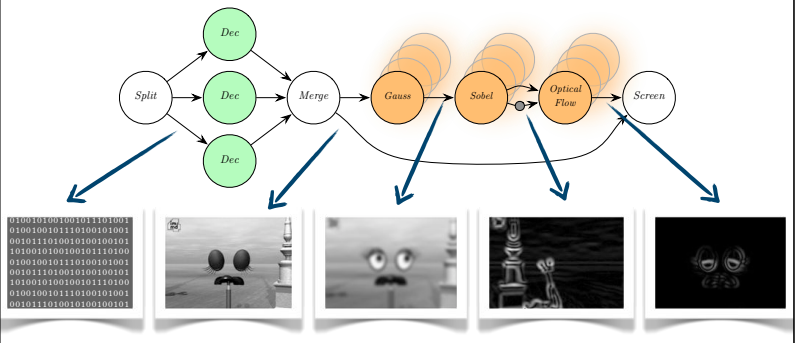


Outline

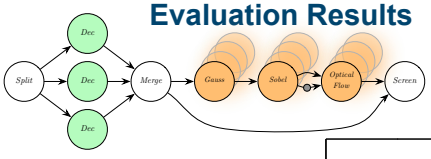
- Introduction
- Approach Runtime-System
- **Evaluation**

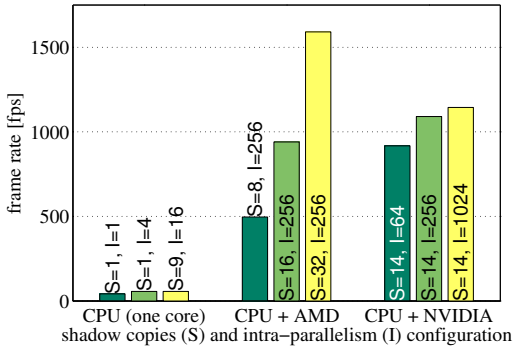
Evaluation - Setup

Evaluation - Video Decoder with Filters

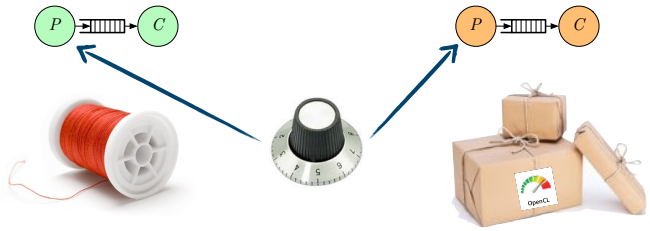


Evaluation Results

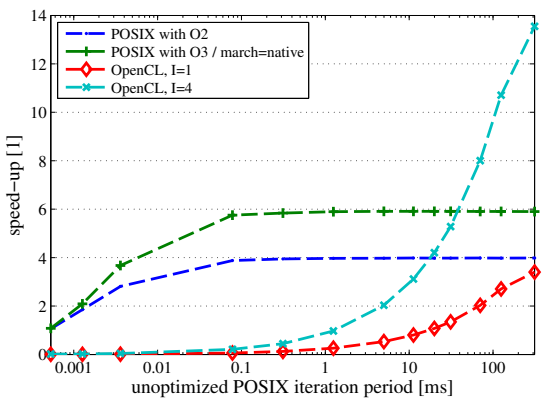




Evaluation - POSIX Thread vs. OpenCL Kernel

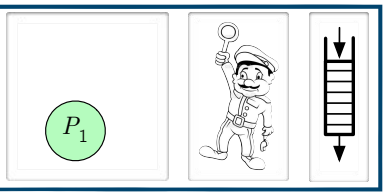


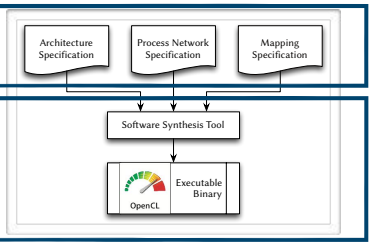
Evaluation - POSIX Thread vs. OpenCL Kernel



Conclusion

- Runtime-System
- Application Programming Interface
- Software Synthesis Tool





ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

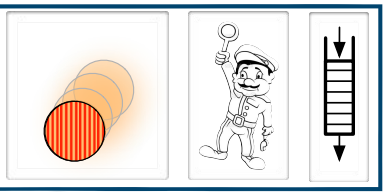
Conclusion

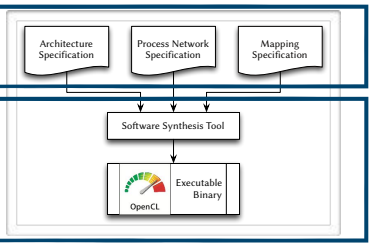
- Runtime-System
- Application Programming Interface
- Software Synthesis Tool

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

Conclusion

- Runtime-System
- Application Programming Interface
- Software Synthesis Tool





ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
TIK Institut für Technische Informatik und Kommunikationsnetze

B

Reading List for OpenCL

Besides the official OpenCL specification in [6], there are other recommendable sources that can be consulted for a deeper understanding of OpenCL. They are listed in the following.

- Book: *OpenCL Programming Guide* [29] – Describes the OpenCL interfaces, and is more detailed than the specification itself. There are also introductory examples.
- Book: *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition* [1] – A textbook for students highlighting all the aspects of parallel programming for heterogenous systems.
- BestPractice: *Writing Optimal OpenCL Code with Intel OpenCL SDK* [26], together with the informative online forum at `http://software.intel.com/de-de/forums/intel-opencl-sdk`, and the online optimisation guide at `http://software.intel.com/sites/landingpage/opencl/optimization-guide/index.htm` give very detailed information on how to tune OpenCL applications for Intel CPUs.
- BestPractice: *AMD Accelerated Parallel Programming OpenCL* [24] is very detailed and informative. It explains how OpenCL applications can be tuned for AMD graphics cards and also CPUs. Furthermore, the appendix lists a lot of useful information according to their products.

- BestPractice: *NVIDIA OpenCL Best Practice Guide v1.0* [25] – Rather old (OpenCL 1.0), not very detailed.
- Paper: GPU Programming [23, 30] – A very good summary of the GPU programming history.
- Paper: *OpenCL - A Parallel Standard for Heterogenous Computing Systems*
- Paper: Performance optimisations for OpenCL applications [31, 32, 33]

Bibliography

[1] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2*. Morgan Kaufmann, 2012.

[2] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A System for Programming Graphics Hardware in a C-like Language," in *Proc. SIGGRAPH*, 2003, pp. 896–907.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," in *Proc. SIGGRAPH*, 2004, pp. 777–786.

[4] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses," in *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 325–335.

[5] "CUDA Programming Guide," Nvidia, 2008.

[6] "The OpenCL Specification, Version 1.2, 2012," Khronos OpenCL Working Group, 2012.

[7] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an Ecosystem for a Scalable, Modular and High-Efficiency Embedded Computing Accelerator," in *Proc. Design, Automation Test in Europe Conf. (DATE)*, 2012, pp. 983–987.

[8] K. Spafford, J. Meredith, and J. Vetter, "Maestro: Data Orchestration and Tuning for OpenCL Devices," in *Euro-Par 2010 - Parallel Processing*, ser. LNCS, P. Dbra, M. Guarracino, and D. Talia, Eds. Springer, 2010, vol. 6272, pp. 275–286.

[9] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli, "Enabling Task-level Scheduling on Heterogeneous Platforms," in *Proc. Workshop on General Purpose Processing with Graphics Processing Units (GPGPU)*, 2012, pp. 84–93.

[10] P. Kegel, M. Steuwer, and S. Gorlatch, "dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems," in *Proc. Int'l Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2012, pp. 174–186.

[11] A. Balevic and B. Kienhuis, "An Efficient Stream Buffer Mechanism for Dataflow Execution on Heterogeneous Platforms with GPUs," in *Proc. Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*, 2011, pp. 53–57.

[12] ——, "KPN2GPU: An Approach for Discovery and Exploitation of Fine-grain Data Parallelism in Process Networks," *SIGARCH Comput. Archit. News*, vol. 39, no. 4, pp. 66–71, 2011.

[13] H. Jung, Y. Yi, and S. Ha, "Automatic CUDA Code Synthesis Framework for Multicore CPU and GPU Architectures," in *Parallel Processing and Applied Mathematics*, ser. LNCS, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Springer, 2012, vol. 7203, pp. 579–588.

[14] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: Portable Stream Programming on Graphics Engines," in *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 381–392.

[15] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Compiler Construction*, ser. LNCS, R. Horspool, Ed. Springer, 2002, vol. 2304, pp. 179–196.

[16] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a High-Level Language for GPUs: (via Language Support for Architectures and Compilers)," in *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, 2012, pp. 1–12.

[17] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures," in *Proc. Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010, pp. 89–108.

[18] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures," in *Proc. Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, 1997, pp. 338–349.

[19] L. Schor *et al.*, "Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems," in *Proc. CASES*, 2012, pp. 71–80.

[20] C. Breshears, *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, 2009.

[21] ——. (2012, Jul.) Is the Future Fine-Grained Or Coarse-Grained Parallelism? [Online]. Available: http://www.drdobbs.com/parallel/is-the-future-fine-grained-or-coarse-gra/240004067

[22] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[23] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[24] AMD. (2012) Accelerated Parallel Processing: OpenCL Programming Guide. [Online]. Available: http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

[25] NVIDIA. (2009) NVIDIA OpenCL Best Practices Guide. [Online]. Available: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[26] Intel. (2011) Writing Optimal OpenCLTM Code with Intel OpenCL SDK Performance Guide. [Online]. Available: http://software.intel.com/file/37171

[27] K. Huang, D. Grunert, and L. Thiele, "Windowed FIFOs for FPGA-based Multiprocessor Systems," in *Application-specific Systems, Architectures and Processors, Proc. Int'l Conf. on*, 2007, pp. 36–41.

[28] S.-H. Kang, H. Yang, L. Schor, I. Bacivarov, S. Ha, and L. Thiele, "Multi-objective mapping optimization via problem decomposition for many-core systems," in *Proc. IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. Tampere, Finland: IEEE, 2012.

[29] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL Programming Guide*. Addison-Wesley Professional, 2011.

[30] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation

on Graphics Hardware," in *Computer Graphics Forum*, vol. 26, no. 1. Wiley Online Library, 2007, pp. 80–113.

[31] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 73–82.

[32] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," *arXiv preprint arXiv:1005.2581*, 2010.

[33] T. Gunarathne, B. Salpitikorala, A. Chauhan, and G. Fox, "Optimizing OpenCL Kernels for Iterative Statistical Applications on GPUs," in *Proc. Int'l Workshop on GPUs and Scientific Applications (GPUScA)*, 2011, pp. 33–44.