TAMPERE UNIVERSITY OF TECHNOLOGY

Ville Korhonen

PORTABLE OPENCL OUT-OF-ORDER EXECUTION FRAMEWORK FOR HETEROGENEOUS PLATFORMS

Master of Science Thesis

Examiners: Dr.Tech. Pekka Jääskeläinen and Prof. Jarmo Takala
Examiners and subject approved by the Faculty Council of the Faculty of Computing and Electrical Engineering
March 5th 2014

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's degree programme in Signal Processing and Communications Engineering
Ville Korhonen : Portable OpenCL Out-of-Order Execution Framework for Heterogeneous Platforms
Master of Science Thesis, 55 pages
December 2014
Major: Embedded Systems
Examiner: Dr.Tech. Pekka Jääskeläinen and Prof. Jarmo Takala
Keywords: OpenCL, task scheduling, out-of-order execution, heterogeneous platforms

Heterogeneous computing has become a viable option in seeking computing performance, to the side of conventional homogeneous multi-/single-processor approaches. The advantage of heterogeneity is the possibility to choose the best device on the platform for different distinct workloads in the application to gain performance and/or to lower power consumption. The drawback of heterogeneity is the increased complexity of applications all the way from the programming models to different instruction sets and architectures of the devices.

OpenCL is the first standard for programming heterogeneous platforms. OpenCL offers a uniform *Application Program Interface (API)* and device platform abstraction that allows all different types of devices to be programmed in the same platform portable way. OpenCL has been widely adopted by major software and chip manufacturers and is increasing in its popularity.

OpenCL requires an implementation for the standard in order to be used. One such implementation is the *POrtable Computing Language (pocl)* open source project launched in Tampere University of Technology. The aim for pocl is in easy portability on different devices. One goal of pocl is improved performance portability using a kernel compiler that is able to adopt to different parallel hardware resources on the devices.

This thesis describes an out-of-order execution framework for pocl. This work offers a flexible and simple API for efficient offloading of computation to the devices, and for synchronising computation between the main application and other devices on the platform. The focus in this thesis is set on the task level operation, with fast task launching and efficient exploitation of available task level parallelism.

An interest has emerged for using OpenCL as a middleware for other parallel programming models. The programming models might be highly task parallel and the size of the tasks might be much smaller than in nominal OpenCL use cases that tend to focus on data parallelism. In the runtime implementation of the proposed framework the focus was in minimising overheads in task scheduling in order to improve scalability for said programming models.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Signaalinkäsittelyn ja tietoliikennetekniikan koulutusohjelma
Ville Korhonen: Portable OpenCL Out-of-Order Execution Framework for Heterogeneous Platforms
Diplomityö, 55 sivua
Joulukuu 2014
Pääaine: Sulautetut järjestelmät
Tarkastajat: TkT Pekka Jääskeläinen ja prof. Jarmo Takala
Avainsanat: OpenCL, task scheduling, out-of-order execution, heterogeneous platforms

Heterogeenisesta laskennasta on muodostunut varteenotettava vaihtoehto perinteisten homogeenisten yhden tai useamman prosessorin ratkaisuiden rinnalle. Heterogeenisyyden etu on mahdollisuus valita alustalta kullekin laskentatehtävälle siitä parhaiten suoriutuva laite ja näin saavuttaa parempi suorituskyky tai pienempi tehonkulutus. Heterogeenisyyden varjopuolena on ohjelmien monimutkaisuus erilaisista ohjelmointimalleista eri käskykantoihin ja laitearkkitehtuureihin.

OpenCL on ensimmäinen ohjelmointistandardi heterogeenisille alustoille. OpenCL tarjoaa yhtenäisen ohjelmointirajapinnan ja laitealusta-abstraktion, joka mahdollistaa hyvinkin erilaisten laitteiden ohjelmoinnin samalla alustariippumattomalla tavalla. Monet merkittävät ohjelmisto- ja laitevalmistajat ovat lähteneet tukemaan OpenCL standardia omilla toteutuksillaan ja standardin suosio kasvaa jatkuvasti.

OpenCL standardi vaatii toteutuksen, jotta sitä voi käyttää. Eräs toteutus on avoimen lähdekoodin *POrtable Computing Language (pocl)*, joka on tehty Tampereen teknillisellä yliopistolla. Toteutuksen yhtenä päämääränä on tarjota helposti eri laitteille siirrettävä alusta. Toinen merkittävä päämäärä on suorituskyvyn siirrettävyys hyödyntämällä kernel-kääntäjää, joka pystyy adaptoitumaan laitteiden erinäisiin rinnakkaislaskentaresursseihin.

Tässä työssä kuvataan out-of-order laskentaan ohjelmistokehys pocl- projektiin. Tämä työ tarjoaa joustavan ja yksinkertaisen rajapinnan tehokkaaseen tehtävien jakamiseen laitteille ja tehtävien synkronointiin pääohjelman ja laitteiden, sekä alustalla olevien laitteiden välillä. Pääpaino työssä on tehokkaassa tehtävätason rinnakkaisuuden hyödyntämisessä ja nopeassa tehtävien jakamisessa.

Hiljattain mielenkiinto on herännyt OpenCL implementaation käyttämiseen välikerroksena muille ohjelmointimalleille. Nämä ohjelmointimallit voivat olla hyvinkin tehtävätasonrinnakkaisuuteen painottuneita ja tehtävien koko voi hyvinkin olla huomattavasti pienempi, kuin mihin yleisesti totuttu OpenCL toteutuksissa, jotka usein keskittyvät datatason rinnakkaisuuteen. Ohjelmistokehyksen runtime toteutuksessa on keskitytty tehtävien vuorontamisen kustannusten minimointiin, jotta toteutus skaalautuisi paremmin myös edellä kuvatuille ohjelmointimalleille.

PREFACE

The work in this M.Sc. thesis was carried out in the Department of Pervasive Computing at Tampere University of Technology as a part of the Parallel Acceleration Project (ParallaX) project.

I would like to thank my examiners Prof. Jarmo Takala for letting me work on this project and especially D.Sc. Pekka Jääskeläinen for his valuable guidance and advice on implementation. Furthermore, thanks go to my colleagues in the Customized Parallel Computing group at TUT for giving motivation and sharing insightful ideas. I would also like to thank Kalray and Clément Léger for his feedback on the implementation of this work. Last but not least, thanks to my family for giving me encouragement throughout the work.

Tampere, November 13, 2014

Ville Korhonen

CONTENTS

LIST OF ABBREVIATIONS

API	Application Programming Interface
ASIP	Application-Spesific Instruction-set Prosessor
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
GPGPU	General Purpose Graphic Processing Unit
GPP	General Purpose Processor
GPU	Graphic Processing Unit
HPC	High-Performance Computing
ISA	Instruction Set Architecture
ISP	Image Signal Processor
MIMD	Multiple Instruction Multiple Data
MPSoC	Multi Processor System-on-Chip
MPI	Message Passing Interface
P2P	Peer-to-Peer
SDI	Serial Digital Interface
SIMD	Single Instruction Multiple Data
SMP	Symmetric MultiProcessing
SPIR	Standard Portable Intermediate Representation
SPMD	Single Program Multiple Data
SSE	Streaming SIMD Extensions
TTA	Transport Triggered Architecture

1. INTRODUCTION

Requirements for computer performance have got continuously higher. The time when a single-core processor offered sufficient performance for applications available has passed. Performance of a single core has not developed significantly in several years because of heat dissipation and power consumption are becoming unbearable. Performance requirements for mobile devices are also increasing while battery technologies are leaving behind. In mobile context, it is important to minimize power consumption without compromising performance and this cannot be achieved by relying solely on *General Purpose Processors (GPP)*. GPP's are designed for general purpose computers and are usually optimised for computation speed. Usually there is a trade-off in speed and in energy efficiency.

Focus in seeking performance in GPP field has shifted from doing computation as fast as possible in a single serial stream to doing computation in multiple streams in parallel as fast as possible. Multi-core has become de facto feature in general purpose processors. Executing multiple programs and/or multiple threads of a program at the same time on several cores increases the overall performance to some extent. From a single program's point of view the performance gain from increasing core count saturates if the program cannot be reasonably further divided into more threads. On the other hand, if the work can be divided across all cores in the processor, the next problem is the shared memory which all the cores are using. In data intensive computation, memory bandwidth easily becomes the performance defining bottleneck on multi-core processors. Yet another problem with GPP's is the relatively high energy consumption due to generality. These properties encourage distributing part of the computation to other programmable computing devices.

High memory bandwidth and massively parallel computing abilities of graphic cards have been used for a long time to accelerate image processing for games and video rendering that would be impossible to achieve in real-time with a GPP. The concept of *General Purpose Graphic Processing Unit (GPGPU)* was brought up less than a decade ago. The idea of GPGPU was to enable GPU to be used in other than graphic processing. NVIDIA and AMD came up with their own vendor specific approaches opening their GPU architectures for the general use. Ever since GPUs have shifted from being just single purpose graphic accelerators into programmable parallel processors.

Even if devices offer a great deal of parallel computing resources, they cannot be exploited without proper tools. Programming heterogeneous platforms used to be heavily application-specific and dependent on tools provided by platform vendor. Designing and porting applications to new platforms was very cumbersome. To address this problem in year 2008 Apple together with other significant chip and software manufacturers formed an initial proposal concerning uniformed standard for parallel programming of heterogeneous platforms. Khronos Group accepted the proposal and eventually *Open Computing Language (OpenCL)* standard was released. OpenCL offers a portable standard and a framework for distributing execution across different heterogeneous platforms via unified runtime-API. Many vendors have released an OpenCL implementation for their platforms which adds to the value of the standard. One OpenCL implementation for OpenCL standard is an open source project *POrtable Computing Language (pocl)*, on which the proposed out-of-order framework is built on.

In this thesis, an out-of-order execution framework was implemented for pocl to allow heterogeneous multi-device platforms to be utilized as efficiently and as flexibly as possible, when there is task parallelism available in the application. The purpose of the framework is to be portable onto wide range of computational devices and to minimise coordination overhead and remove unnecessary dependencies to the host program. By this framework a programmer should be able to take performance of a heterogeneous platform to the full extent. For example one target is to offer a framework for researching parallel computing with custom *Application Specific Instruction-set Processor (ASIP)* devices and platforms developed in Tampere University of Technology.

One research aspect was to examine that could it be possible to break free from the usual use case of OpenCL applications launching a huge kernels at a time, towards more fine grained parallel tasks. A goal was to make the task scheduling runtime so lightweight that the suitability of the OpenCL could be evaluated as an implementation layer for other parallel programming models like OpenMP, Cilk or Halide.

This thesis is divided in following sections. Chapter 2 gives an overview on the field of compute platforms and gives an introduction to programming of heterogeneous platforms with OpenCL. Chapter also describes Open Computing Language standard, which is one approach to programming heterogeneous platforms, and Open Computing Language implementation Portable computing language to which the out-of-order framework is build. Task scheduling problem is discussed in Chapter 3. Chapter 4 describes the proposed framework, other framework supporting modifications to pocl and the example device interface implementations for the framework. Chapter 5 includes performance tests and Chapter 6 concludes the thesis.

2. HETEROGENEOUS PARALLEL COMPUTING

Heterogeneous computing is a method of computing, which utilises variety of different kind of programmable devices forming a heterogeneous platform. A heterogeneous platform can consist of multiple GPP, *Graphics Processing Units (GPU)*, *Digital Signal Processors (DSP)* and *Application-Specific Instruction-set Processors (ASIP)*. Motivations for heterogeneous computing are increased performance and/or lower power consumption compared to traditional CPU applications. Performance increase is gained not only from multiple devices working concurrently but from tasks being assigned to the device most capable of performing the task.

In this chapter, typical parallel compute platforms are presented including example devices/platform and the available parallelism they offer. After platform review the programming model of the OpenCL standard is reviewed and the target OpenCL implementation for this thesis is introduced.

2.1 Platforms

The field of compute platforms is large. Platform architectures ranges from simple embedded devices to computers with a CPU and multiple graphic cards or *Multi-Processor System-on-Chip (MPSoC)* with multiple different computing devices integrated in to the same chip. Also the properties of computational devices ranges from a simple heavily device driver controlled accelerator performing one task at the time, to a device able to independently execute a given set of tasks in parallel using multiple compute resources. For example, a single core in a processor is considered as a compute resource. The number of computational resources range from one to thousands depending on the device.

An interesting device feature is its parallel execution capabilities. Some devices are suitable for task parallel execution, which means that related or unrelated jobs can run concurrently and asynchronously on different computational resources. Commonly term *Multiple Instruction Multiple Data (MIMD)* is used. An example of MIMD is multiple different programs running on a PC. Another form of parallelism is data parallel computing, where the same operation is performed to different data items concurrently. Commonly used concepts are *Single Instruction Multiple Data (SIMD)* and *Single Program Multiple Data (SPMD)*. An example of data parallelism can be found from the field of image processing. The same operation is performed for every pixel in the image and every pixel can be processed in parallel with, e.g., SIMD-style vector operations.

2.1.1 Symmetric Multi Processor system

Symmetric MultiProcessing system (SMP) is a compute device which consist of two or more identical processor cores forming a single device. All processor cores have the same *Instruction Set Architecture (ISA)* and all cores operate in the shared memory independently from each other. Usual SMP implementation is a multi-core processor, which by itself can be thought as a special case of a heterogeneous platform where all computational resources happen to be identical. Multi-cores are usually used as the main processor or the host on the heterogeneous platform running the applications and controlling the execution on the top level. Multi-core processors are excellent in task parallel computing which is the reason they are the de facto CPU type used on platforms from hand held devices to servers. Multi-core processors are usually run with an operating system which handles running programs as processes consisting one or more threads. Thread is the smallest schedulable task for a single processor core, thus threads define the granularity of expressing task parallelism on multi-core CPUs. Due to homogeneity, threads can be executed on any core available on the device and cores can be executing their own threads simultaneously, without need for explicit taks allocation. Usually cores have also a limited capability to perform data parallel computation in a form of vector operations. For example x86 instruction set has *Streaming SIMD Extensions (SSE)* features which allow vector operations from 16x8 bit to 2x64 bit to be executed in parallel.

Multi-core processors have been popular in distributed systems like server farms, that consist of multiple distinct computers with their own multi-core processors. These homogeneous platforms are easier to program since all the compute resources have the same instruction set architecture and the same memory hierarchy. Work distribution is easy since once compiled tasks can be off loaded to any of the devices on the platform. Programming models for multi-processor platforms with shared memory communication have been long available. Maybe the most relevant tools are OpenMP and MPI. OpenMP is an API specification for parallel programming on multi-core and multi-processors platforms. MPI is a message passing interface, which allows devices to communicate over a multitude of interconnect mediums.

2.1.2 CPU with GPGPU Offloading

One common heterogeneous platform is based on a single CPU and one or more GPUs. GPUs offer great performance for data parallel computation with their vast number of computational resources and very high memory bandwidth. There are motherboards with a possibility to attach four, if not more, graphics cards. There are also graphics cards with two GPUs so it is possible to assemble a configuration with 8 GPUs which provides considerable computing performance for a personal computer. For example AMD R9 295X2 graphic card has two GPUs each having 2816 stream processors and setup having four of those has total of 22528 stream processors. On top of that each stream processor executes multiple instructions in parallel.

Graphic cards have their own local memory where the GPU operates. Offloading work for the graphics card requires also uploading the input data. After computation the results needs to be read back from the device. Data transfers in and out of the device introduces latency which needs to be taken account when using this type of device. An exception to this is graphics accelerators that share the memory address space with the CPU. This kind of setups have significantly lower overhead compared to separate graphic cards since the data can be shared with only passing pointers to the shared memory address space.

The memory transfers between the the local memory of the GPU and the system memory are performed with *Direct Memory Access (DMA)* transfers. If two graphic cards need to exchange data, it also happens via the system memory. In Fig. 2.1, is an example of a device pipeline where GPU 1 gets input data from the system memory and produces output for GPU 2 which calculates the final pipeline output back to the system memory. In Fig 2.1(a), costly data copying between memories happens twice. A better option would be the Fig. 2.1(b) where the system memory is bypassed with direct memory access from GPU 1 local memory to the other GPU's local memory.

AMD and NVIDIA have introduced *Peer-to-Peer (P2P)* feature on some of their products, enabling data transfers directly from a GPU memory to another GPU memory without CPU's intervention. NVIDIA introduced P2P implementation on their GPUDirect version 2 release in year 2011. AMD introduced their implementation DirectGMA in the same year.

Both AMD and NVIDIA make possible to transfer data from GPU to another kind of devices. In NVIDIA device the system memory can be used to relay data to InfiniBand devices, which are communication link devices often used in HPC computing. In AMD, data can be transferred directly to *Serial Digital Interface (SDI)* devices over the PCIe bus. SDI devices are for outputting video streams. At the moment communication capabilities of the GPUs are quite limited and it would require standardisation of communication methods to broaden GPUs connectivity to any kind of devices on the PCIe bus.

In the field of *High-Performance Computing (HPC)*, GPU accelerated clusters

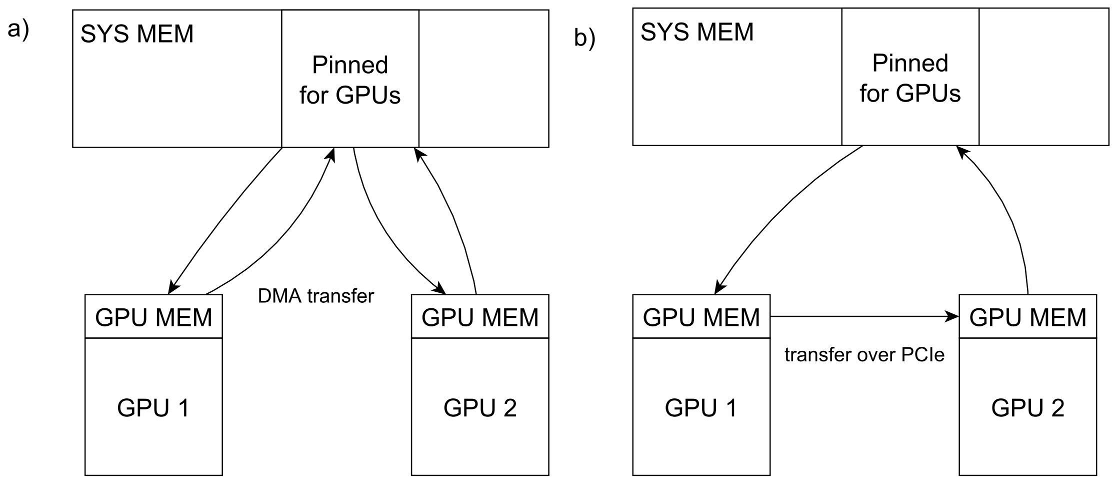

Figure 2.1: a) Devices cannot communicate directly and the host needs to transfer data from device to another via system memory. b) Devices transfer data between one another without host intervention.

have become viable competitor for the traditional CPU clusters. In performance/cost ratio comparison, CPUs are clearly out classed by GPUs. Using GPUs in HPC platform brings its own difficulties. The device interconnect topology needs to be designed in a way that it minimises the transfer latencies. Devices should be maximally utilised without interconnect congestion. Task mapping plays an important role in the overall performance [8]. Task mapping can exploit possible localisation of compute nodes. For example, pipelining devices connected to the same PCIe bus could exchange data without blocking the higher level interconnect. This is possible of course only if the devices have the previously mentioned P2P feature.

2.1.3 MPSoC Platforms

MultiProcessor System-on-Chip (MPSoC) platforms often used in mobile devices usually have several different kind of computational devices integrated in the same chip. For example Qualcomm's high end MPSoC Snapdragon 800 has a quad-core CPU, GPU, *Digital Signal Processor (DSP)* and *Image Signal Processor (ISP)* [13]. MPSoC platforms offer hardware for great variety of computation and selecting the right hardware for the for right task is important for boosting performance and/or to lowering the energy consumption [1].

In mobile MPSoC field, a popular topic has been the GPGPU computing. GPUs increase computing power on mobile platforms similarly to a PC platform. Most of the studies and applications have been image or video processing related, such as computer vision algorithms [16; 15] and face recognition applications [17]. Common conclusions in the studies are significantly increased performance and lowered energy consumption when comparing CPU only implementations to GPU or CPU+GPU implementations.

Other devices of the MPSoC platforms, such as DSP, have also been studied. For example in the study [18] included TI OMAP3530 chips DSP. In 2D FFT benchmark, The DSP was faster and more energy efficient compared the CPU and the GPU on the same chip. Best results in energy consumption was achieved by using DSP+GPU leaving the CPU idle. Slightly more energy consuming but better performing configuration was to use all of the devices concurrently.

The advantages of distributing computation across the MPSoC platform is evident. One major problem is that some devices are dependent on vendor specific libraries that make porting the applications difficult. For example in the mobile context, applications are sold for Android, iOS or Windows phone devices. For these operating systems there are multiple vendors offering different platforms with their own programming tools. In these circumstances, it is not possible to create applications accelerated by multiple computational devices that would work on all platforms supported by a specific operating system. GPUs are an exception, because operating systems requires a GPU supporting some graphic API. For example Android and iOS require OpenGL ES support and Windows requires DirectX and Direct3D.

2.1.4 Many-core Architectures

Many-core devices consist of multiple, usually uniform, processor cores. Many-cores differ from multi-core processors mainly in the core count. The definition is ambiguous since there is no limit in the core count when multi-core becomes many-core, or whether GPUs with hundreds of “cores” (without independent instruction streams) are considered many-cores or not. Devices identified as multi-cores usually have two to sixteen cores and many-cores have from tens to thousands of cores. One distinction between the two could be the field of use to which the device is intended for. Multi-core processor as a term is usually used for general purpose CPUs. Many-cores are usually aimed for more special computation intensive tasks.

Intel’s many-core brand Xeon Phi products are coprocessors to be used to accelerate Intel Xeon server processor platforms. It can used as a slave accelerator for main CPU or it can be configured as an independent compute node. The Xeon Phi product family provides devices with 57-61 cores, and each core is accelerated with vector operations. Xeon Phi uses bidirectional ring as an interconnect between the cores is illustrated in Fig 2.2. Each core has its own L2 cache and these caches are accessible by other cores. Xeon Phi offers higher throughput and better energy efficiency compared to sole Xeon processor platforms.

Another example of a many-core is the MPPA-256 from Kalray which can run

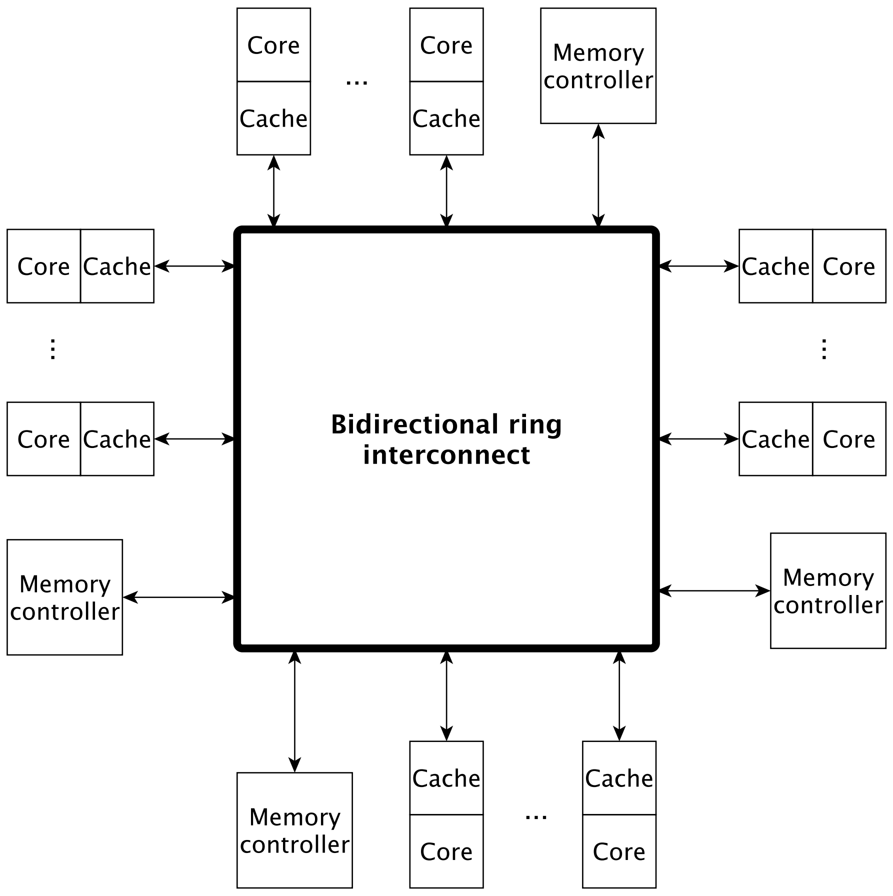

Figure 2.2: Xeon Phi interconnect topology. Each core has L2 cache and each core connects to the bidirectional ring interconnect.

in standalone or can be used as an accelerator. The MPPA-256 has 256 cores and multiple devices can be chained to form even larger processor entity. In the device architecture, 256 cores are divided to 16 clusters each containing 16 cores and for every cluster there is one system core to handle the cluster I/O. The MPPA-256 architecture is depicted in Fig 2.3. Kalray offers an SDK for traditional POSIX programming model with processes, threads and process interconnect tools. It also offers a C based parallel data flow programming model for describing task and data flow graphs.

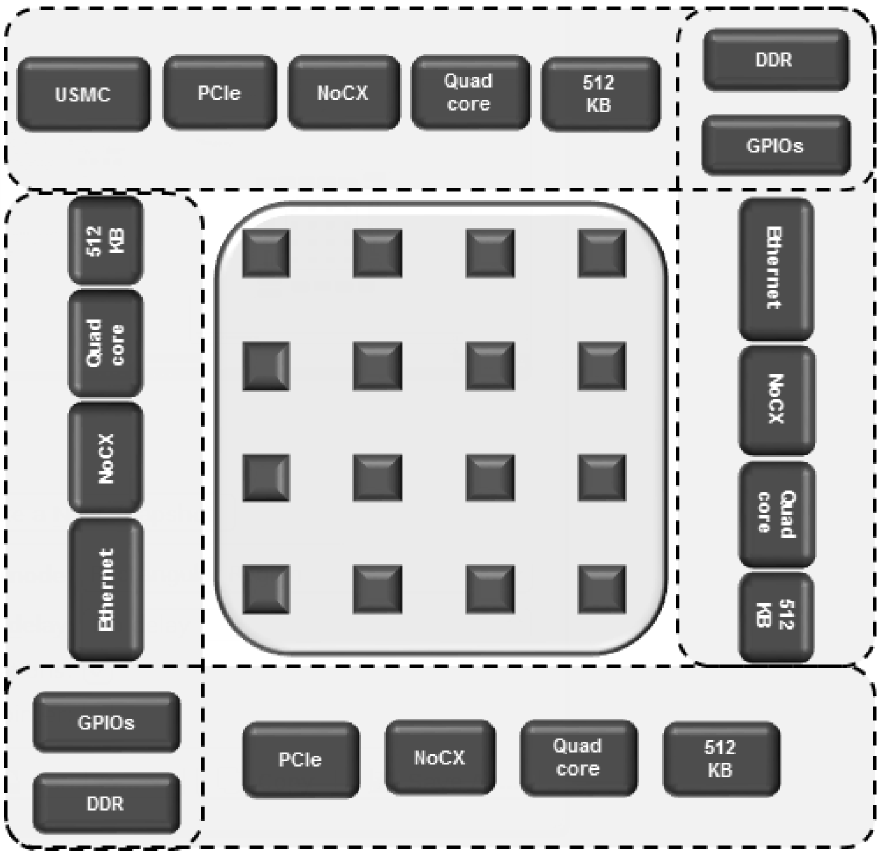

Figure 2.3: Kalray's MPPA-256 architecture. [11]

2.2 Programming

There are few parallel programming tools available for parallel compute platforms. The tools have different approaches for expressing task and data parallelism. Intel's Cilk Plus is a C/C++ language extension, which allows the programmer on language level to explicitly state the code regions that can be executed in parallel. The parallel execution is left for Cilk's runtime to handle. Cilk is aimed for SMP systems and is not suitable for heterogeneous platforms. OpenMP is another programming language extension, which offers an API for parallel programming. OpenMP is based on compiler directives that are used for marking parallel code regions and synchronisation points in the code. OpenMP was originally aimed for shared memory multiprocessing, but lately some advancements have been made towards the heterogeneous platforms. Intel's TBB is a C++ template library for parallel programming of multi-core processors and their Xeon Phi coprocessors. TBB offers a high abstraction level programming model for expressing parallel programs.

Tools targeted for heterogeneous platforms have been scarce. The usual approach has been vendor and device specific tools, libraries and programming models. In GPGPU field, both AMD and NVIDIA opened their GPU architectures for general use by publishing their own programming tools. AMD came up with their *Close To Metal (CTM)* interface in the year 2006, which was later renamed AMD Stream SDK for the official release. NVIDIA published *Compute Unified Device Architecture (CUDA)* platform in the year 2007. These were remarkable steps towards wider adoption of heterogeneous platforms.

The problem with these approaches is the platform portability. A CUDA accelerated program can only be executed on a platform with NVIDIA GPU, and a CTM application on AMD GPUs. It became evident that an uniformed standard for parallel programming of heterogeneous platforms was needed. In the year 2009, OpenCL 1.0 was released to address this problem. OpenCL offered a portable standard and a framework for distributing execution across different heterogeneous platforms via unified runtime-API. Many vendors have released an OpenCL implementation for their platforms which adds to the value of the standard.

This Chapter describes OpenCL key features focusing on details relevant to this thesis. Also an open source OpenCL implementation pocl is presented on which the proposed framework is constructed.

2.2.1 Open Computing Language

Open Computing Language, shortly OpenCL, is an open industry standard for general purpose programming of heterogeneous platforms. Standard includes programming language, compiler, API and a runtime system to support programming of the multi-device accelerated applications. OpenCL provides a low-level abstraction layer which relieves the programmer from delving into device specific details(if performance is not so crucial aspect) and enables creation of portable programs. OpenCL is suitable for a whole range of heterogeneous platforms from embedded and hand held devices to high performance compute servers. [12]

OpenCL is described by a hierarchy of models. The models are Platform model, Memory model, Execution model and Programming model. The Platform model describes the device platform abstraction used by the standard, a host running the host program and devices that execute kernel code. The Execution model describes how the computation is orchestrated on the devices, available commands for devices and synchronisation. The Memory model describes the supported memory hierarchies and ordering rules for memory accesses. The Programming model describes the higher level aspects, such as parallel programming models that are available, and the lower level platform portability aspects, such as memory layout and indexing of vector data types. OpenCL provides a framework for applications to use the available devices on the platform as a single heterogeneous parallel computer system. OpenCL framework's Platform layer allows host program to discover supported devices available on physical platform and query device properties. The framework's Runtime provides API for managing memory objects, carrying out computation on the devices and synchronising the program execution. The framework includes a compiler for C99 based OpenCL C programming language for creating programs running on the devices. [12]

Platform Model

OpenCL platform model is illustrated in Fig. 2.4. A platform consists of a *host* connected to a collection of compute *devices*. A device consists of compute units which are divided into one or more processing elements. Device executes computation on processing elements. OpenCL application consists of both the code running on the host and the code running on the devices.

The host is normally located in the CPU of the actual device platform. The purpose of the host is to run the host program. The host program describes the behaviour of the application on a higher level, defines tasks that needs to be done, gathers input information required by computation and offloads computation to the devices.

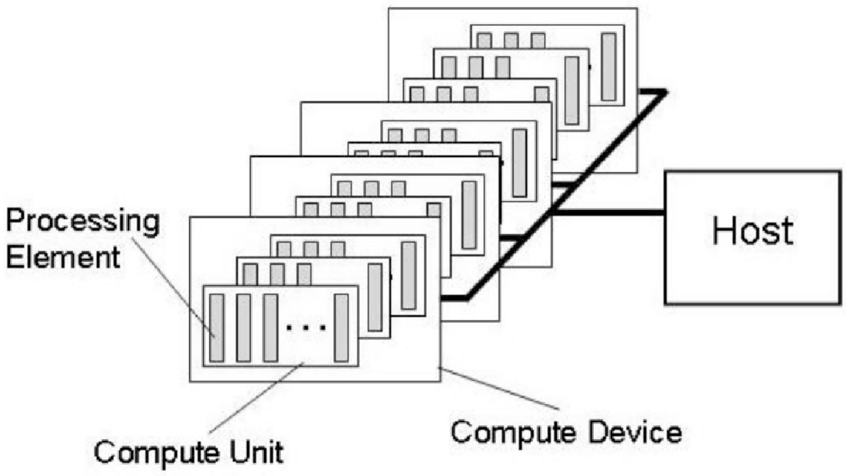

Figure 2.4: OpenCL platform model. [12]

Execution Model

Execution of OpenCL application is divided into two parts; the host program that runs on the host and *kernels* that run on devices. The kernels are executed in a well-defined context. The context includes the selected devices on the platform, kernels and memory objects that are used by the host and kernels. Through OpenCL API the host program creates and manages the context.

The host interacts with the devices through commands enqueued to command queues. A command queue can be assigned for a single device only but a single device may have multiple command queues. There are two modes of operation available for command queues. In-order execution ensures that the side effects of the enqueued commands appear as if they were executed in the order they were enqueued. Usually meaning that the memory operations on the memory objects occur in the correct order implied by the command enqueue order. The other option is out-of-order execution where the command execution is constrained by explicit synchronisation and explicitly defined event dependencies. Commands can be associated with an event object that encapsulates the commands execution status. The command dependencies are expressed with event wait lists. If a command needs to be executed after another, another command's event is placed to the command's event wait list. A command may not be executed until all events in its event wait list are marked as completed.

There are three types of commands in OpenCL. Memory commands transfer data between the host and device memory or between the memory objects. Synchronisation commands are used to explicitly control execution order of the commands.

Kernel commands are used for offloading computation to devices.

OpenCL introduces three types of kernels. OpenCL kernels are the most common approach and they must be supported by all OpenCL implementations. They are usually written in OpenCL C and compiled with OpenCL C compiler. OpenCL kernels can be provided as a source code and compiled at runtime or they can be provided as precompiled binaries. Also an optional extension *Standard Portable Intermediate Representation (SPIR)* enables support for external intermediate presentation of kernels. Through SPIR kernels can be defined with any kernel programming language that targets SPIR.

Native kernels are an optional feature and their semantics are implementation specific. Native kernels are either functions in the application or functions from libraries written in language other than OpenCL C. Native kernels can operate on the same memory objects as OpenCL kernels. The third option are the built-in kernels that are device specific precompiled functions. Built-in functions are always implementation specific. Only OpenCL kernels are described in detail in this chapter.

OpenCL kernels are launched by launching a kernel command on a device. Kernel commands include an index space called NDRange, meaning N-dimensional range. One kernel instance, also called as work-item, is executed for each point in the defined index space. For example, a generic matrix addition kernel can be created by defining kernel to execute a scalar addition in one NDRange coordinate. When addition is needed for *[n, m]* matrices the kernel is launched along with *[n, m]* NDRange. The device executes n times m kernel instances and when all work-items are finished, the result is ready in the output matrix.

Work-items are organised into work-groups, which are assigned to compute units. Work-items in a work-group are further assigned to the processing elements of the groups compute unit. NDRange index space is illustrated Fig. 2.5. Index space size is (G_x, G_y) and work-items are divided into work-groups of size (S_x, S_y). In kernel commands enqueue phase the work-group size may be left undefined when runtime or device driver implementation may choose an optimal size for the device.

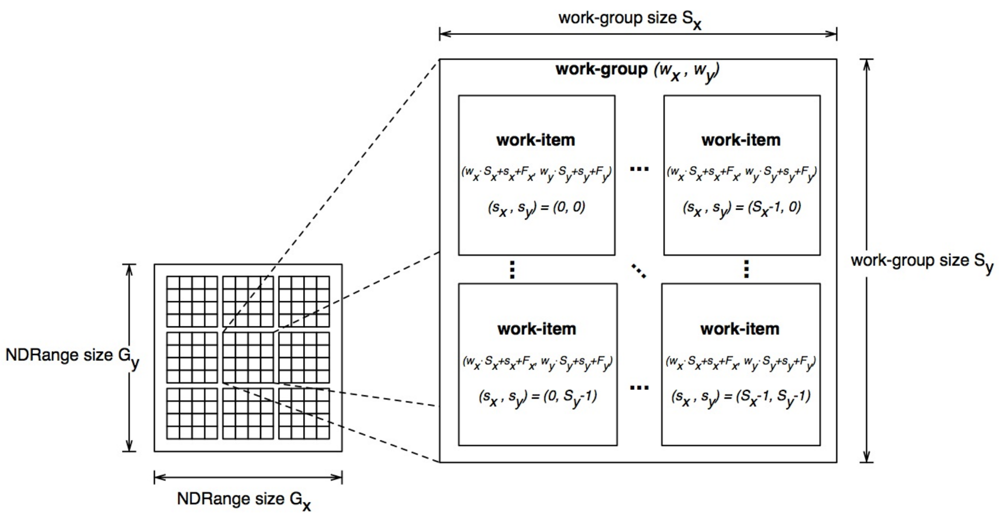

Figure 2.5: 2-dimensional NDRange. [12]

2.2.2 Portable Computing Language

Portable computing language aka. *pocl* is an open source OpenCL implementation. The design goal of *pocl* is to provide a modular platform and performance portable OpenCL implementation. At the time of writing *pocl* is not a complete standard conforming implementation. Some features are yet to be implemented.

Software architecture of pocl is illustrated in Fig. 2.6. Fundamental principle of pocl is to separate host and device specific parts to their own layers so they can be ported separately. The *Host layer* requires a target with an operating system and a C compiler. On the *Device layer* resides the operating system and device specific parts such as the code generation for target device and details concerning kernel execution on the device. OpenCL Runtime API is mostly implemented as generic C implementations that communicate with the *Device layer* through the *Host-device* interface whenever device specific details are needed. OpenCL Compiler implementation relies on LLVM and Clang OpenCL frontend on the *Host layer* and target specific LLVM backend on the *Device layer*. Performance portability features are managed by series of LLVM passes on the *Host layer*. [10]

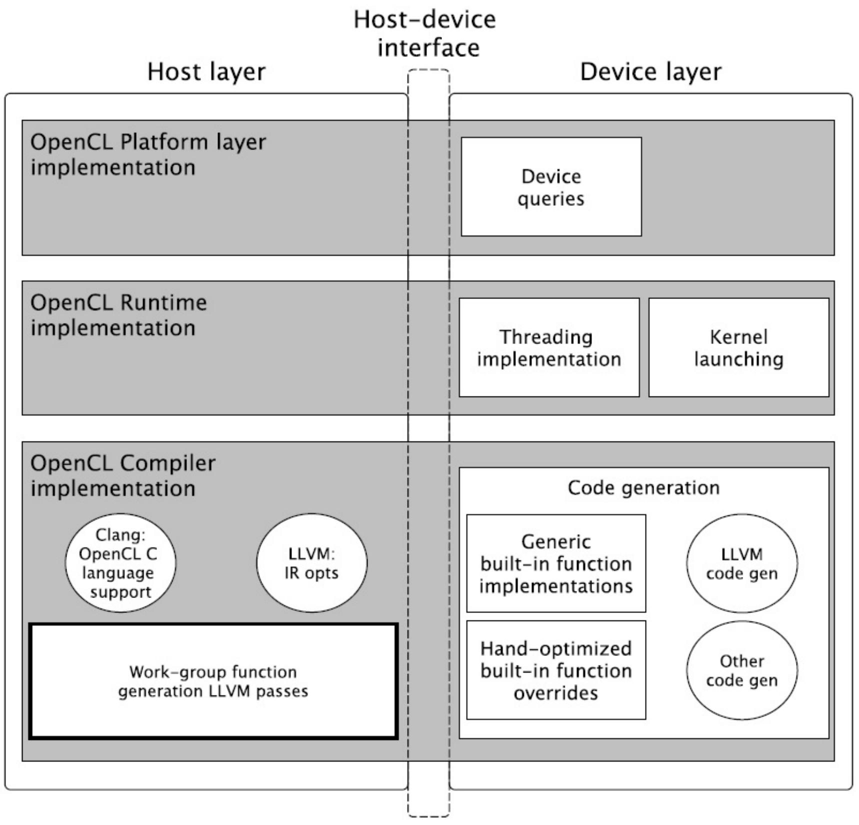
Host-device interface
Host layer
Device layer
OpenCL Platform layer implementation
Device queries
OpenCL Runtime implementation
Threading implementation
Kernel launching
OpenCL Compiler implementation
Clang: OpenCL C language support
LLVM: IR opts
Work-group function generation LLVM passes
Code generation
Generic built-in function implementations
LLVM code gen
Hand-optimized built-in function overrides
Other code gen

Figure 2.6: The software architecture of pocl. [10]

3. TASK SCHEDULING RUNTIMES

Creating applications for the parallel compute platforms with multiple independent compute units usually requires dividing the application into smaller independent jobs, that can be assigned to the compute resources on the platform. These independent operations are referred as tasks.

For executing these tasks the programming tools may offer a task scheduling runtime. Task scheduling runtime's purpose is to arrange the execution of given tasks in respect of ordering constraints to produce the expected output. Runtimes are available for single memory address space such as Cilk Plus and OpenMP and suitable for multiple memory spaces, such as TBB and OpenCL runtime. There are many aspects that needs to be addressed in the runtime implementation. In this chapter these aspects are introduced and their impact on performance is discussed.

3.1 Task Level Parallelism

When the application, or a part of it, is divided into tasks one must express the dependencies between them in order to get the intended output. Tasks and dependencies can be expressed as a task graph. A task graph consists of nodes and edges connecting them. Nodes are the tasks and edges are the dependencies. The information that the task graph encapsulates is the correct order of task execution to obtain correct results, and the task level parallelism available. For example the construction of a simple modular house can be divided into independent tasks and expressed as a task graph (Fig. 3.1). The ground must be levelled before building the foundations, and putting up the walls require foundations. Finally the roof is put on top of the wall elements. These are independent tasks that have a strict order of execution, thus edges defining the serial execution order. There is also task parallelism available in this graph. The construction of the walls and the roof are disjoint tasks that can be executed in any order or simultaneously and these tasks have no dependencies to the ground work. For lower construction time, these parallel task should be executed by different parties simultaneously.

OpenCL standard does not include a concept of a task graph but task graphs can be expressed with the included constructs. Task graphs in OpenCL are formed by enqueueing commands to command queues and by defining dependencies between the commands. A simple serial task graph can be formed by enqueueing commands

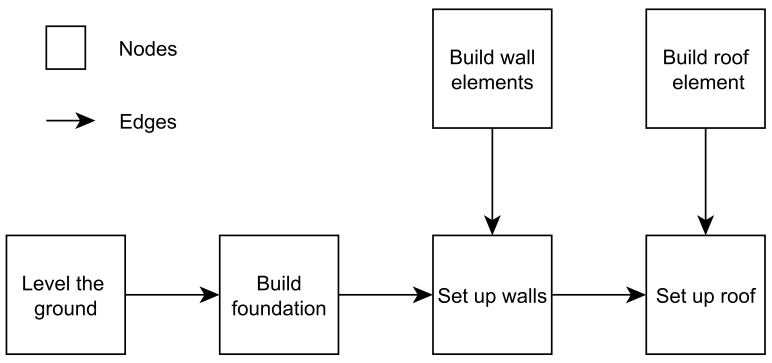

Figure 3.1: Task graph for building a modular house.

to an in-order command queue in the order which they are required to run. An example of a task graph built from an OpenCL command queue is illustrated in Fig. 3.2. The program writes kernel input data to the device then launches the kernel B which produces input for next kernel C and finally the host reads back the results from the device. Edges between the nodes are implicitly implied by the in-order semantics.

Out-of-order execution offers more flexibility to the task graph construction. In out-of-order command queues, every command has its own event wait list defining the execution ordering constraints. More complex task graph in Fig. 3.3 benefits from the out-of-order command queue because all buffer write operations can be executed in parallel on both devices. Device 1 can execute commands independently up to command G where it has to synchronise with the device 0 to get input data from command E to command G.

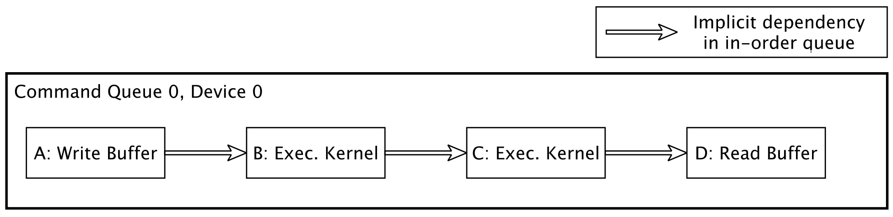
Implicit dependency in in-order queue
Command Queue 0, Device 0
A: Write Buffer
B: Exec. Kernel
C: Exec. Kernel
D: Read Buffer

Figure 3.2: Example of a simple task graph. All tasks need results from the previous task. No task level parallelism available

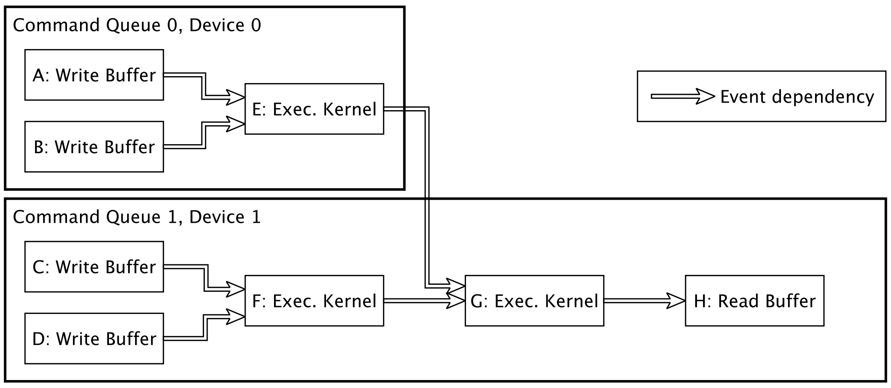

Figure 3.3: Example of program with a more complex task graph.

3.2 Task Scheduling

Available scheduling methods for task scheduling runtimes are static and dynamic scheduling or a combination of both. In static scheduling, given task graph is pre-analysed against the underlying device platform and based on that analysis tasks are scheduled to the available computational resources. Analysis consist of factors such as structure of the task graph, execution times of the tasks and communication costs between the tasks. Several methods, mainly heuristic, have been introduced for estimating said properties. Static scheduling is performed at compile time or at runtime before starting the task graph execution. The advantage of static scheduling is to be able to make sure that the most important task are scheduled first, which is crucial in communication-intensive, real-time and irregular problems. [7; 4; 14]

In dynamic scheduling, the schedule is formed at runtime. The execution order of parallel tasks and the exact place where a single task is executed is not necessarily known beforehand. Usually dynamic scheduling algorithms aim for high utilization of computational resources which makes it suitable for computation intensive applications with lots of parallelism available [7]. Dynamic scheduling can achieve better work load balance with dynamically varying compute loads [6].

Combining both the dynamic and static scheduling is also possible. Some of the scheduling is done beforehand at compile time or before launching the task graph and the rest of the scheduling is done at runtime. In OpenCL applications, the commands are explicitly assigned to certain devices by enqueueing commands to their queues which is a static decision. On the other hand, the OpenCL does not say anything about device's internal scheduling of the tasks. A device schedules the given tasks on the fly in respect of the dependencies and according to its own capabilities, in serial, in parallel, out-of-order etc.

Load balancing is a key factor in efficient task scheduling. The goal in load

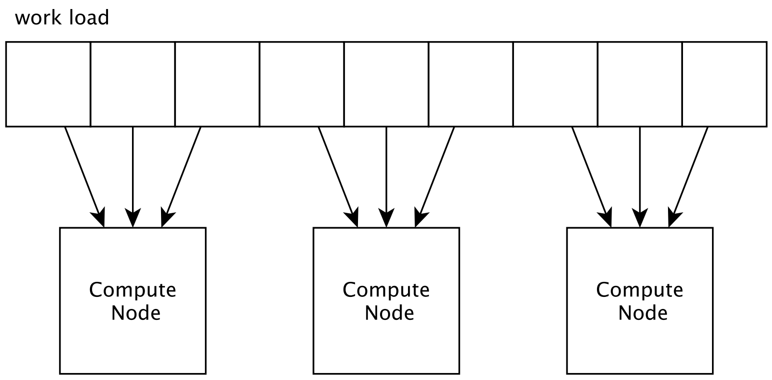

Figure 3.4: Simple static symmetric work distribution.

balancing is to maximise the utilisation of the compute nodes to achieve better performance. Work load can be balanced statically or dynamically. In static balancing, work load is partitioned and explicitly assigned for compute nodes. A simple partitioning method is to divide the jobs evenly between compute nodes as shown in Fig. 3.4. If the individual jobs perform the same static function and the compute nodes are identical, the result is a well balanced schedule with minimal scheduling overhead. On a heterogeneous set of compute nodes with significantly different performances the proposed simple balancing is far from optimal. The performance is defined by the slowest node because the higher level task cannot complete until all of its partial jobs are completed. This problem may be overcome by profiling the performance of the compute nodes and using that information in the task allocation. Node with a relative performance of 2 gets double amount of jobs compared to a node with a relative performance of 1 (Fig. 3.5). [9]

Static work balancing does not give a good load balance in every case. If the

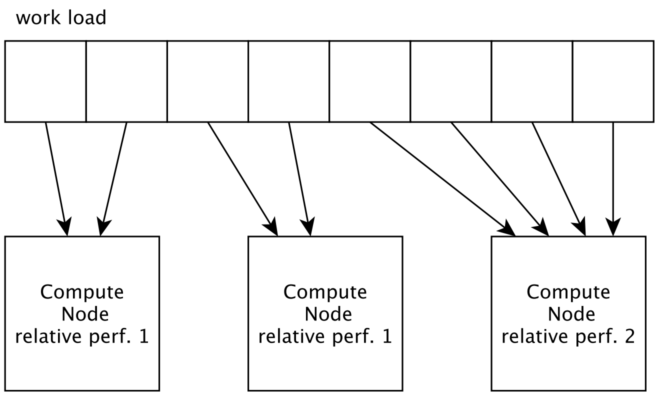
work load
Compute Node relative perf. 1
Compute Node relative perf. 1
Compute Node relative perf. 2

Figure 3.5: Static asymmetric work distribution by relative performances.

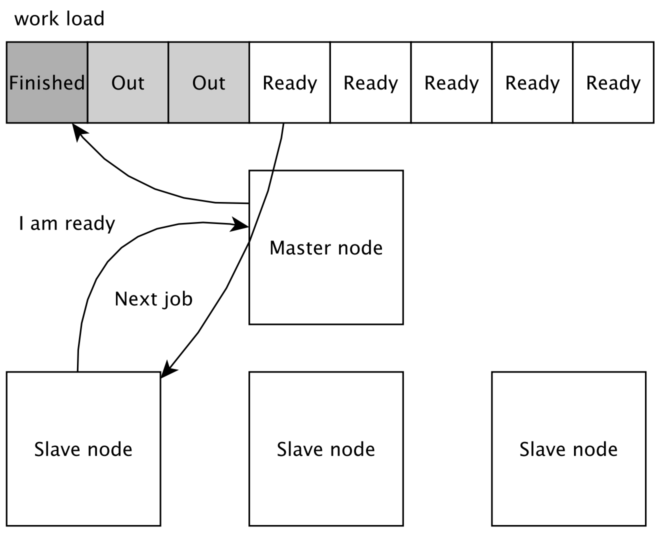

Figure 3.6: Dynamic work distribution using a master node. Finished is a completed job, Out is in execution on some node and Ready is ready for execution.

execution times of the individual partial jobs vary dynamically it is impossible to statically achieve optimal balance. The difference in execution time may be large between the best work distribution and the event where most of the worst case jobs are scheduled to a single compute unit. One possible solution to this problem is the dynamic work balancing techniques. In dynamic work balancing, jobs are not assigned to any particular compute node. One idea is that compute nodes are given a job when they have finished the previous job. One solution is to use a master node which pushes the jobs to the slave nodes when they are ready to take more work (Fig. 3.6). This way a job gets to execution as soon as there is a node available to compute it. The weaknesses in this configuration are increased synchronisation overhead and the master node which could bottleneck the work distribution when the amount of compute nodes is great. [9]

Other option is to use active nodes that independently pull more work from a shared work queue (Fig. 3.7). In this configuration, access to shared work queue must be synchronised, which introduces synchronisation overhead. The synchronisation overhead depends on the efficiency of the locking mechanism, on the number of consumers and on the work load of a single job. In highly suboptimal circumstances, considerable portion of total execution time could be wasted on lock contention where the compute nodes are waiting to get the lock. On the other hand it allows the work producing entity and compute nodes to work asynchronously from each other, and therefore minimize the scheduling overhead on the host.[9]

In multi-core platforms, a common approach on dynamic load balancing is the work stealing concept [2]. The basic idea is that the compute nodes have their own

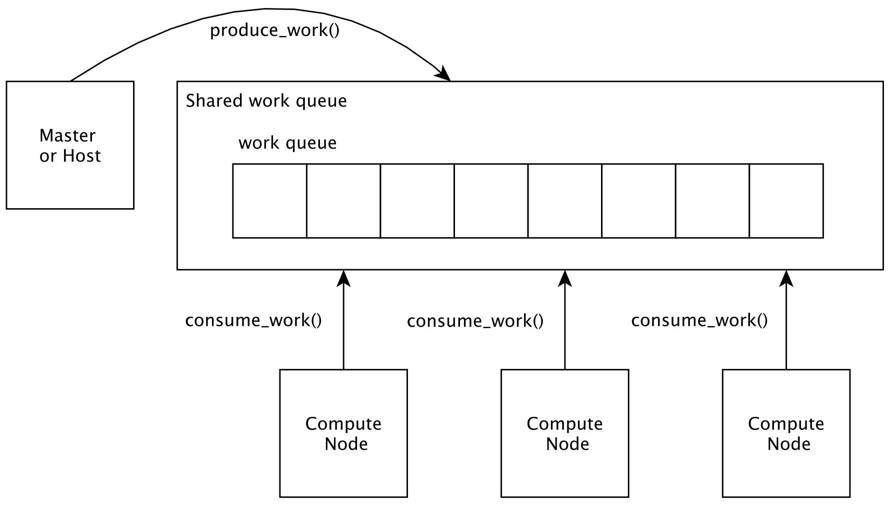

Figure 3.7: Dynamic work balancing by shared work queue where work producing and consuming is handled with access synchronising API functions.

work queues and when a node has consumed all the work from its own queue, it tries to steal work from the work queues of the other nodes. The method is able to obtain the good features of static and dynamic balancing. The work can be dispatched to compute nodes as in Fig. 3.4 with minimal scheduling overhead. The dynamic variations in jobs execution time can be overcome with the dynamic work stealing (Fig. 3.8).

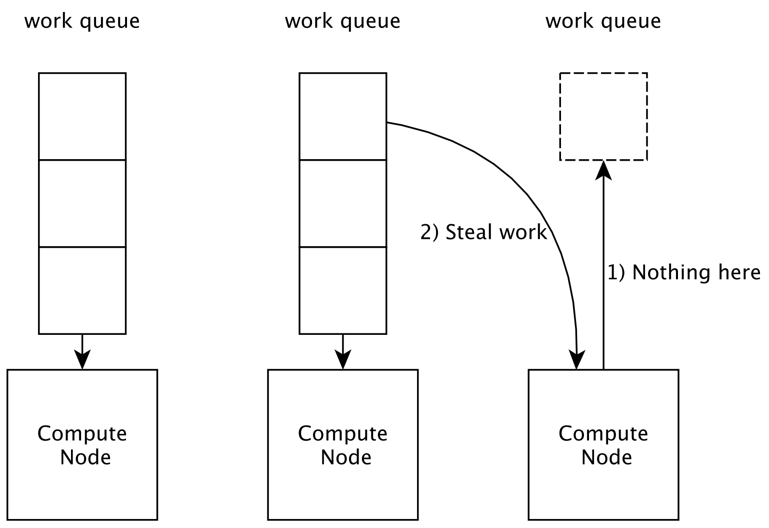

Figure 3.8: Dynamic load balancing by work stealing.

3.3 Synchronisation

Synchronisation is a crucial part of the task scheduling. Task synchronisation comprehends mutual exclusion and serialisation of tasks [5]. Data synchronisation is closely related to task synchronisation. Data synchronisation means that multiple copies of the same data are kept coherent.

Mutual exclusion is needed, for example, in the case presented in Fig. 3.7. Consuming an item from the shared work queue is considered a critical section where only one thread of execution is allowed at any time. Multiple simultaneous accesses to critical section would lead to an undefined state of work queues control structure. One common way to implement mutual exclusion is locks. When entering a critical section the thread tries to obtain the lock guarding the region. If the lock is obtained, the thread continues to the critical section, if not the thread is blocked until lock is released.

Locks are commonly implemented with atomic operations. Atomic operation is an operation that prevents any other party from concurrently accessing the same resources during the execution of an atomic operation. Common atomic operations are read-modify-write operations that read a memory location, alter its data, and write it back while blocking any other operations to the same location. For example compare-and-exchange instruction atomically compares the given memory location to given value and iff the values match, the contents of the memory location is exchanged with another given value. By this operation lock can be defined by trying to atomically write “1”(to lock) to a memory location iff contents of the location is “0”(is not locked), otherwise do nothing. [3]

Serialisation of tasks usually requires synchronisation. Serialisation is needed when a task produces a result that is used as another tasks input data. Task serialisation between compute nodes, that are not able to communicate with each other, is handled in the master node. On heterogeneous platforms where compute nodes are different devices this might be the case and the master node, usually the device running the application, has to take care of the synchronisation. When the master node gets acknowledgment from compute node about the completed task the host propagates this information to other compute nodes waiting for that task’s completion (Fig. 3.9).

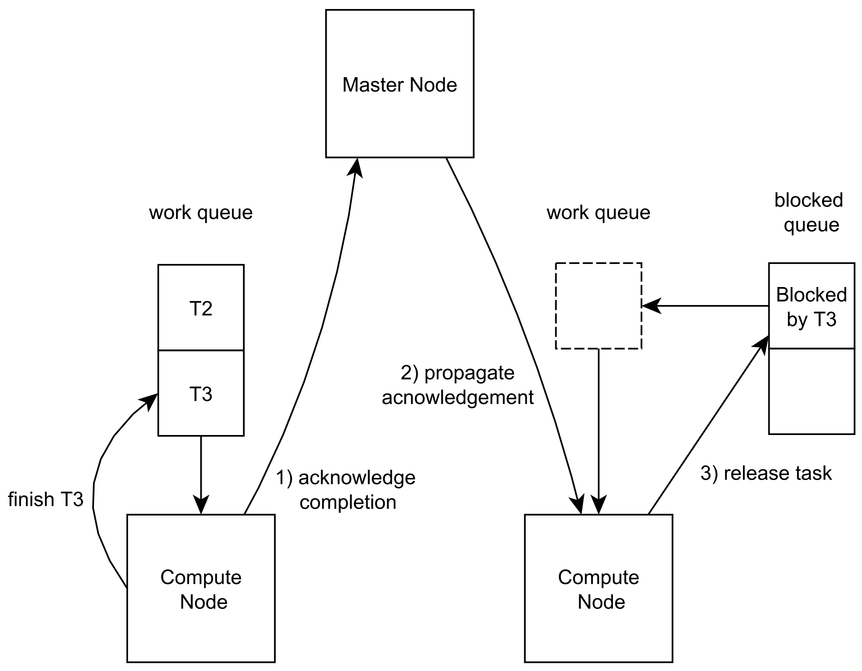

Figure 3.9: Task synchronisation handled by a master node.

Serialisation of tasks with more advanced devices can be implemented using a concept of message passing. There is, e.g., *Message passing interface (MPI)* which is a standardised message-passing system. MPI support is available for several programming languages and device platforms. Serialising task execution over multiple compute nodes can be arranged by sending messages between nodes containing information about completed tasks. A task on a compute node may be set to wait on some tasks completion. When the node receives message with information about desired tasks completion it may release the waiting task (Fig. 3.10). Message passing is viable option in cross device communication, if all parties are supporting it.

In task serialization, the possible shared memory address space could be used between compute nodes. A memory location is assigned to hold information about the status of the waited task. When the waited task is finished by another compute node the node updates the assigned memory location to indicate the completion. A compute node waiting a task to finish gets the acknowledgement by polling the contents of the assigned memory location as depicted in Fig. 3.11. Polling is a simple method to implement, but should be avoided if possible, fpr sake of power consumption and possible memory bus congestion.

On multi-core platforms active polling can be and should be avoided by using signalling between the threads of execution. Signalling is an application for locks where a thread waiting for a signal goes to sleep and the thread is woken up when it receives a signal. Worker thread running on a core may be suspended to sleep and

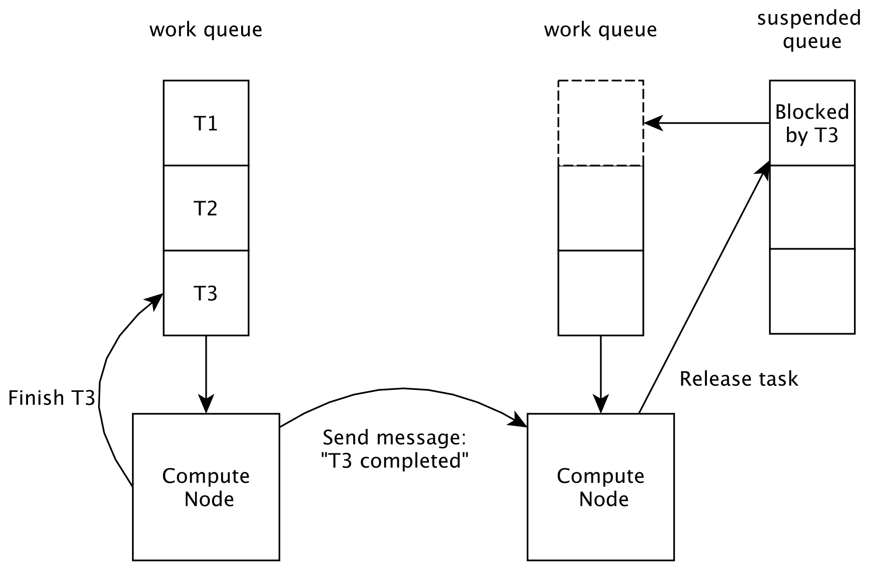

Figure 3.10: Task serialisation by message passing.

wait for a waking signal when it has no work to do or it cannot yet execute all the tasks due to precedence constraints. When the waited task is completed the waiting thread is signalled to wake up and to continue execution as illustrated in Fig. 3.12.

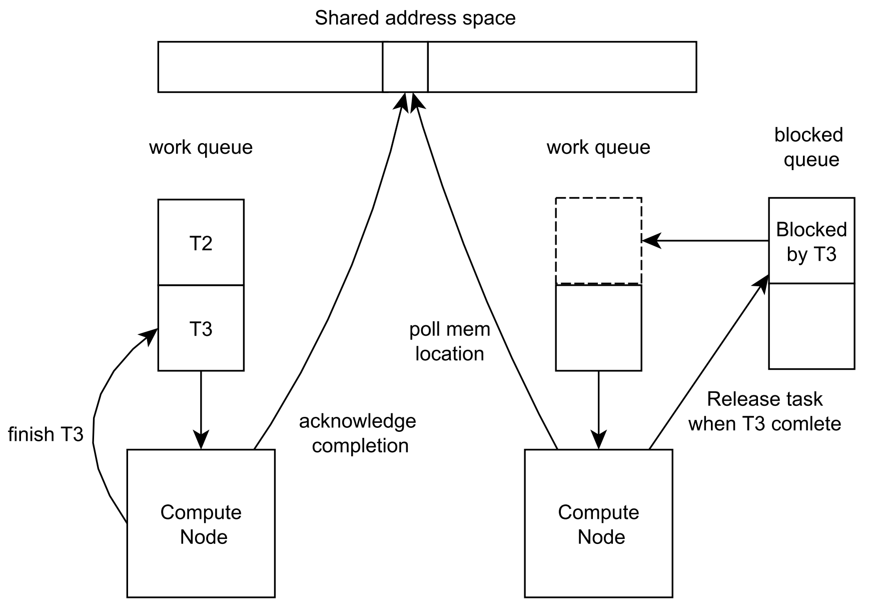

Figure 3.11: Task synchronisation with polling.

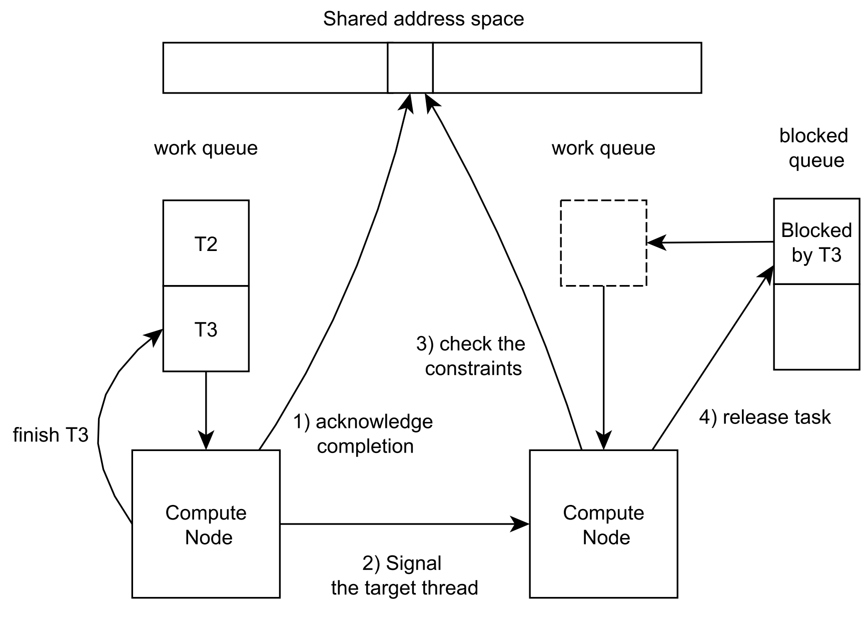

Figure 3.12: Task synchronisation with direct signaling.

Data synchronisation needs to be taken care of along with task synchronisation. Task launch on a compute node usually requires some input data to operate on and tasks usually produce output data which is needed somewhere. The purpose of the data synchronisation is to arrange data movements so that data is in intended state before the task execution begins. For example in OpenCL, a memory object can be used by all the devices in the context and it is the runtimes task to make sure that the data is in a right place at the right time. Of course it is also the application programmer's concern to define task dependencies in a manner that operations to data containers make sense (for example no concurrent writes to the same memory object).

Data synchronisation intertwines with the task synchronisation. When tasks are executed on the same compute node or on nodes in the same memory address space, no data synchronisation is needed since the data is immediately available when the previous task/tasks have produced it. In case of distinct devices with their own local memories, the data needs to be copied from memory to another. In host managed systems, the host can arrange the memory transfers in task synchronisation phase before releasing the task requiring the information as shown in Fig. 3.13.

Data synchronisation can also be handled with implicit tasks that take care of the data transfers. Where these implicit tasks are executed depends on the capabilities of the device platform, in general by the host or by the device. In Fig. 3.14, two implicit tasks are added between the two original tasks to copy data from a device to another via the system memory.

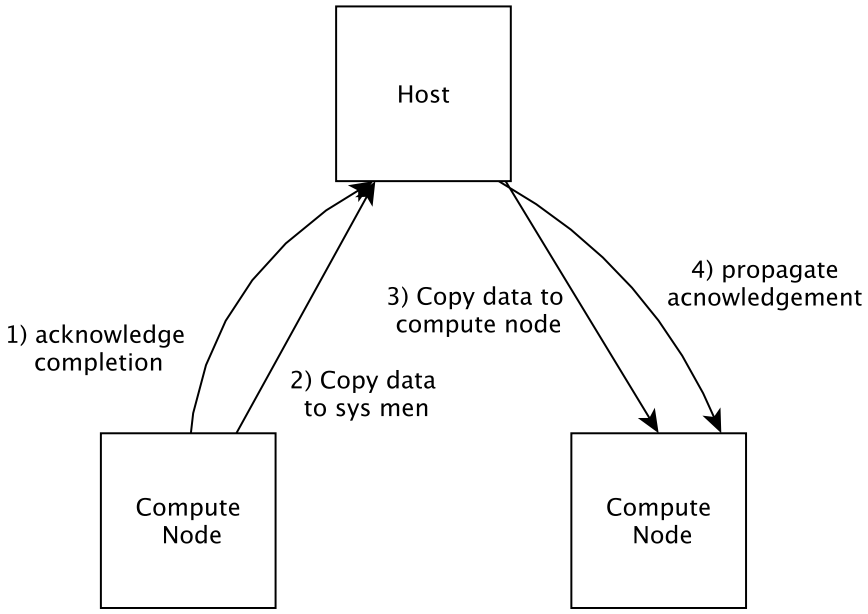
Host
1) acknowledge completion
2) Copy data to sys men
3) Copy data to compute node
4) propagate acnowledgement
Compute Node
Compute Node

Figure 3.13: Data synchronisation handled by the host.

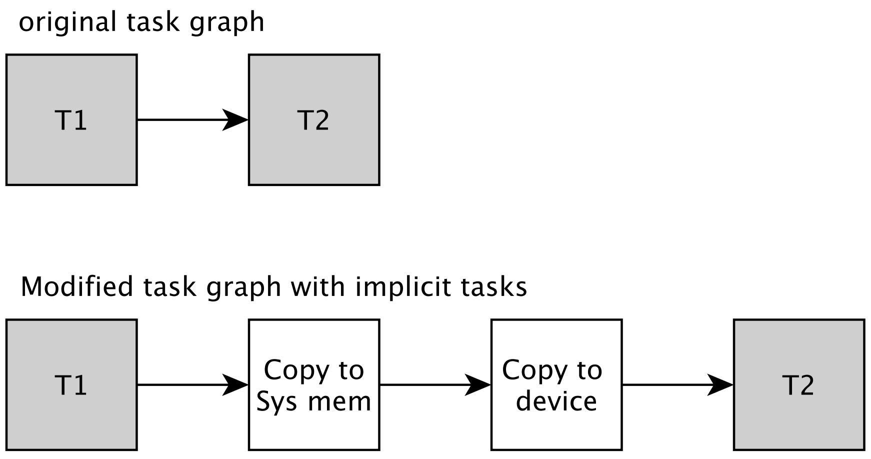

Figure 3.14: Data synchronisation arranged with implicit copy tasks. The device executing T1 could use DMA transfer to copy the data to system memory and the device with task T2 could use DMA transfer to obtain the data before launching T2.

3.4 Latencies

Latencies in distributed and heterogeneous systems are a major issue. The chain of events can be long with many phases when synchronising two tasks executing on different devices. In a host managed system the host might take care of most of the synchronisation as in example illustrated in Fig. 3.15. The host first dispatches the two tasks to the devices. The devices might be in their own memory address space behind an interconnect connecting the host and the devices. Transferring data over the off-chip interconnect always introduces latency. After the task dispatch, another device executes the given task and the other device remains waiting the given task to be released. Eventually the first device completes the task and informs the host by invoking an interrupt. The host device receives the interrupt and switches context to an interrupt handler, which also takes time. The interrupt handler asserts the host application that a device needs attention. Eventually host application gets back to execution and attends to devices needs by querying information from the device memory over the interconnect. The host notices the completion of a task and propagates the information to the waiting device, again over the interconnect, and now the other device may start executing the given task. In addition data synchronisation needs to be completed before releasing the other task. It is evident that latencies begin to accumulate in the host managed configuration. If the launched tasks include transferring lots of data in and out of the devices and the amount of tasks is great, the interconnect congestion might bottleneck the overall performance. Instead of doing something the devices are constantly waiting for something.

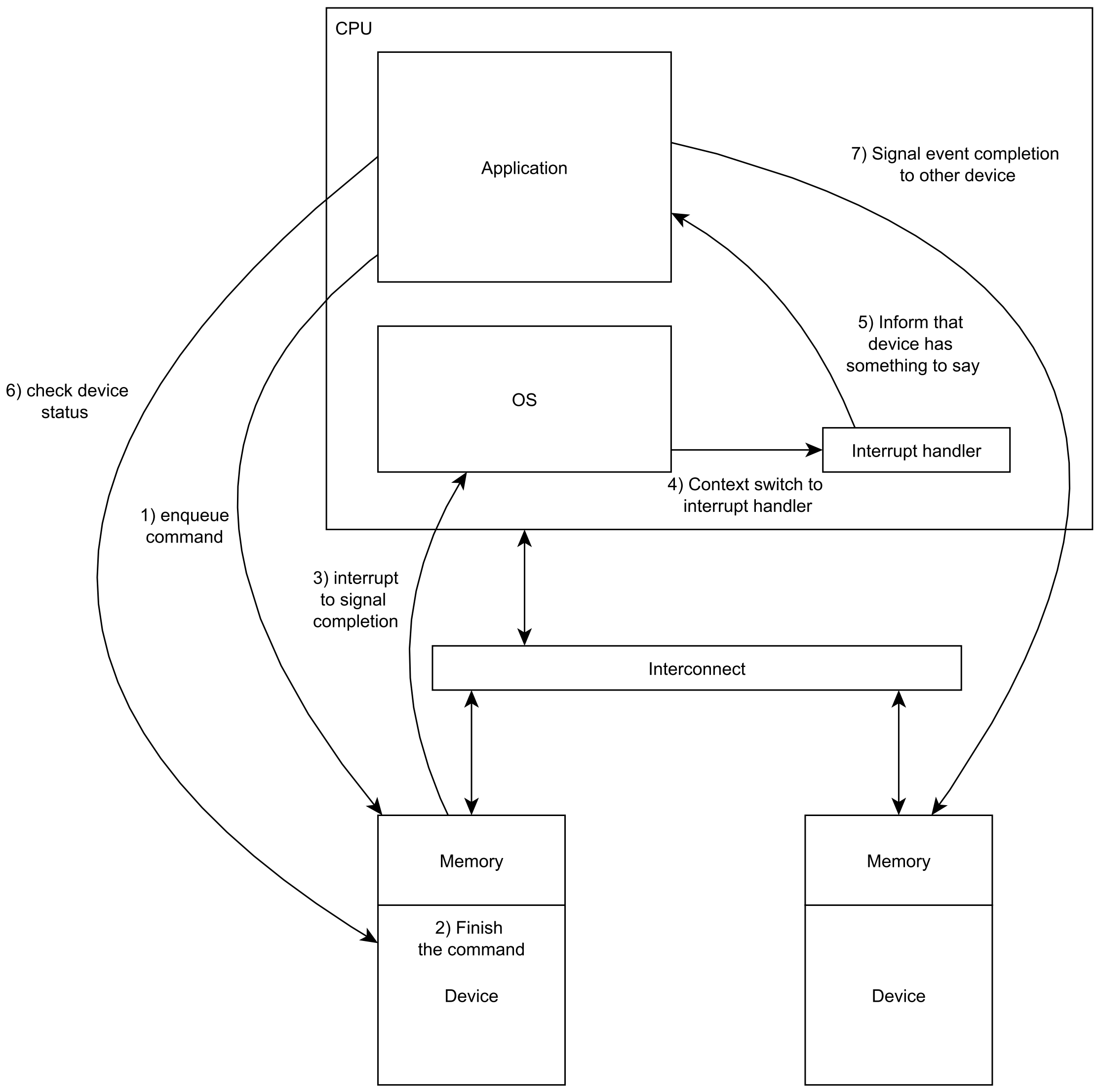
CPU
Application
7) Signal event completion to other device
5) Inform that device has something to say
OS
Interrupt handler
4) Context switch to interrupt handler
6) check device status
1) enqueue command
3) interrupt to signal completion
Interconnect
Memory
2) Finish the command
Device
Memory
Device

Figure 3.15: Possible chain of events in the case where two tasks are synchronised on two different devices.

To avoid latency and interconnect congestion some of the synchronisation overhead should be offloaded to devices themselves if possible. The previously mentioned P2P concept can be used to bypass host if two devices need to synchronise tasks. With P2P communication latencies caused by the actions on the host device and the number of transfers actions over the interconnect are reduced (Fig. 3.16).

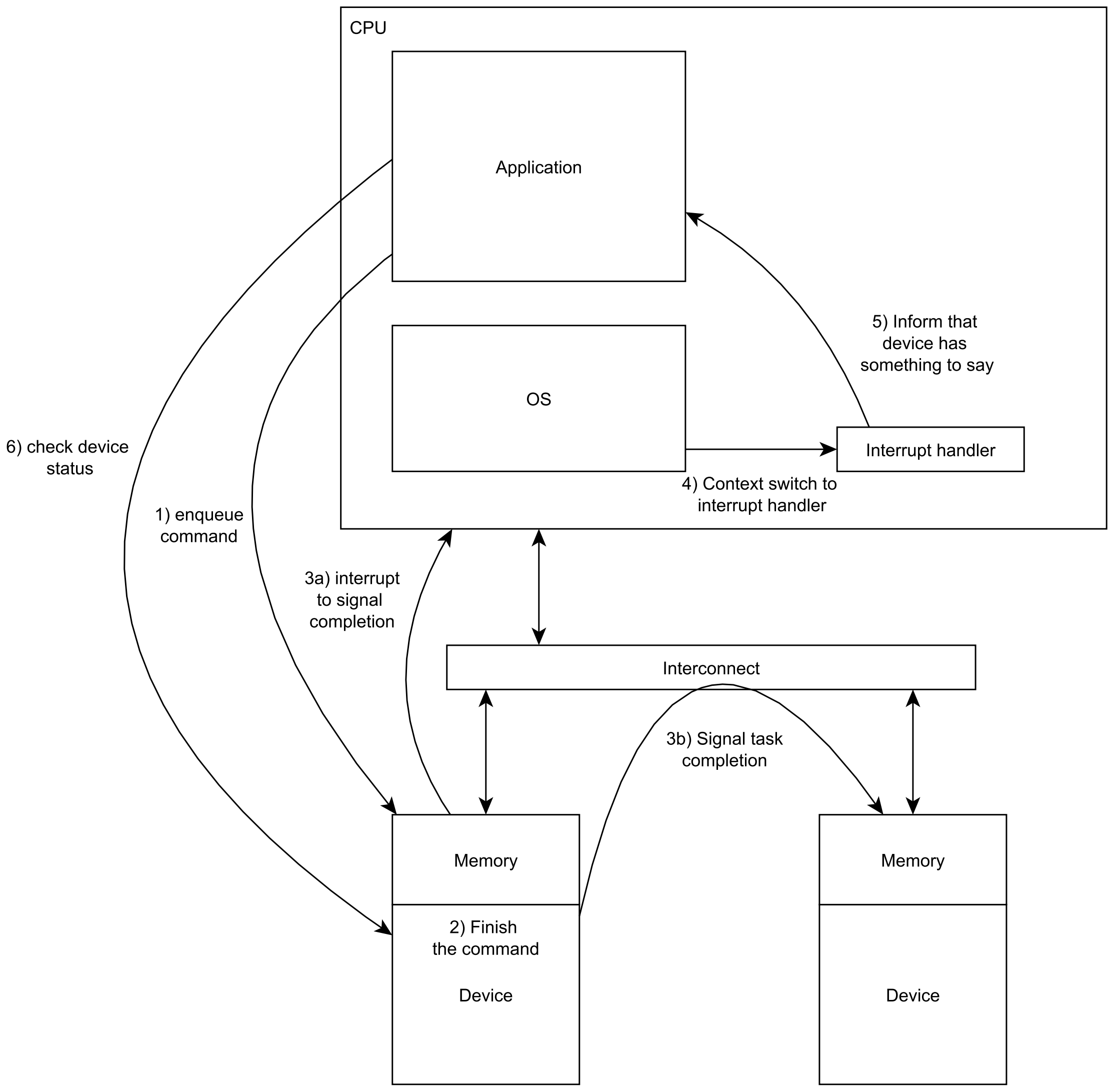
CPU
Application
OS
5) Inform that device has something to say
Interrupt handler
4) Context switch to interrupt handler
6) check device status
1) enqueue command
3a) interrupt to signal completion
Interconnect
3b) Signal task completion
Memory
2) Finish the command
Device
Memory
Device

Figure 3.16: Devices communicate directly with each other with P2P feature and avoid host related latencies.

4. OPENCL OUT-OF-ORDER EXECUTION FRAMEWORK

The goal of OpenCL standard is to be portable on vast variety of platforms. From that basis the design goals for the implemented out-of-order framework was set to support vast variety of heterogeneous platforms and synchronisation methods.

In order to support parallel computing as generically as possible, the properties of the presented parallel compute platforms were addressed in the frameworks design by using the existing example pocl device driver implementations as test platforms. The example implementations cover multi-core processors and separate devices with P2P feature, working in their own global memory address space. Also a very simple, strongly driver driven accelerator type of device was taken into account. A many-core company Kalray participated in the project by implementing a device driver for their MPPA-256 many-core processor and providing feedback about framework functionality.

The designed framework implementation consists of modifications to both the Host layer and the Device layer of the pocl. On the host side OpenCL API functions concerning enqueueing commands, flushing command queues and finishing execution were updated. Events and event handling was updated to serve out-of-order execution. Also a concept of shared global address spaces was added to pocl runtime. On the device side the device driver API was expanded with new functionality. The new functionality comprehend managing submitting commands to the device and synchronising command execution with the host and between devices.

In this Chapter, the framework API and required Host layer and the runtime modifications are presented. Example implementations are introduced in the end.

4.1 Device Driver API Functions to Enable Out-of-order Execution

The framework has an interface of five functions in the device driver API. These functions needs to be implemented to obtain out-of-order execution features. These functions have a higher level predefined semantics and it is up to driver implementation how the semantics are met. On the device side behaviour is target specific. Functions and their purpose are represented in Table 4.1.

Table 4.1: Device driver API extensions.

submit()	Submits enqueued command to the device driver.
flush()	Ensures that all previously submitted commands to a given command queue will eventually be executed.
join()	Used to ensure that enqueued commands in given command queue are completed before returning. Used by host to synchronise execution at the higher level.
notify()	Used for notifying device about completed event that it has been waiting.
broadcast()	When a command is completed the driver broadcasts a notifications to device drivers that are waiting for this commands completion.

The framework function *submit()* is called by clEnqueue-functions when handing the newly created commands to the device driver. *Flush()* is called by clFlush() to make sure that previously enqueued commands to given command queue are eventually finished. *Join()* is called by clFinish to wait for the device to complete all the commands in the given command queue. *Broadcast()* is called by the driver itself when command is completed. Broadcast informs the other device drivers that an event they are waiting for, has been completed. The broadcast has a generic default implementation which calls *notify()* for every device driver that has a command waiting the completed command.

4.2 Commands, Events and Command Queues

Out-of-order execution requires the event wait list feature for the commands. The wait list is implemented as a linked list of waited events in the struct that represents a command. When the commands in the wait list are completed they are removed from the list. A command is considered ready for execution when the wait list is empty.

Responsibility of storing dependencies needed to keep command flow running is assigned to the event which resides in a wait list. When an event is added to a wait list in the new commands creation phase, the event of the command being created is added to the waited event's notify list. If the event to be waited is already complete when forming event synchronisation the synchronisation is left undone. When a command is completed *broadcast()* is called for its event. The default implementation of broadcast goes through the events notify list and calls *notify()* for the driver owning the event in the notify list. With this method drivers that are waiting something can be suspended and woken up when their services are needed. the wait list and the notify list orchestration is shown in Fig. 4.1.

A way to support in-order and out-of-order command queues identically on the device layer is to utilise command's event wait list feature to implement the in-order

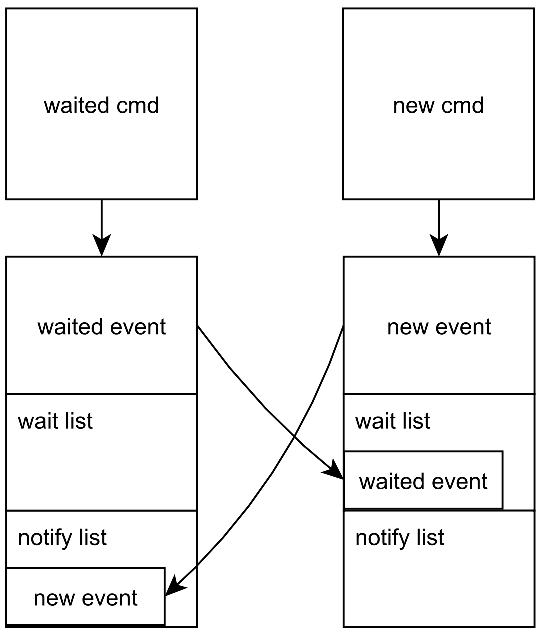

Figure 4.1: When creating new command waited commands event is added to new commands wait list and new commands event is added to waited events notify list.

command queue similarly to out-of-order command queue. This simplifies the device layer implementation since commands are handled in the same manner regardless of the command queue's execution mode. OpenCL runtime API does not require the use of events if the application programmer does not need them. For example events are not required when using in-order queue that is not synchronised to any other command queue. In consequence the concept of implicit events was introduced. If programmer does not request event to be returned when enqueueing a command an event is created anyway so that there is always an event to be used by the runtime in task synchronisation. This implicit event object is the same as normal events but it is not accessible by the programmer. It is only used by the runtime and the device drivers. OpenCL runtime uses the event in its internal synchronisation and frees the event when it is not used anymore.

In-order queue was enhanced with an automatic parallelisation feature. The idea is to extract task parallelism from in-order queue without violating the in-order semantics perceived by the OpenCL application. In-order queue may contain commands that have no common memory buffers they write or they read the same or different buffers. According to OpenCL 1.2 specification, in these situations commands have no effect on each other and they can be allowed to be executed in parallel. OpenCL 2.0 has a new feature program scope variables that can be defined in kernels. Program scope variables have the same lifetime as the whole OpenCL application and once initialized, they can be read and written by other kernels. For the OpenCL 2.0 compliance also the dependencies of these variables needs to be examined.

The parallelisation is based on events and on keeping track of the memory buffers that enqueued commands use. The memory buffer objects themselves store the information about their data synchronisation points. The event objects of read and write operations to a buffer, are placed to a synchronisation point. A single synchronisation point can hold either a single write operation or one to many read operations. In practice a write command always leads to creation of a new synchronisation point and for all subsequent read operations a new synchronisation point is created until the next write operation. Every event in a synchronisation point are synchronised with all the events in the previous synchronisation point to form needed command dependencies.

An example of synchronisation scheme in auto parallelisation is presented in Fig. 4.2. In this example, in-order command queues are used to describe a command flow where kernels are reading the same buffer and writing different buffers. In-order semantics suggests that each commands execution should start when previous command has completed. Although when data synchronisation data structure for buffer X is formed and event dependencies formed according to data synchronisation the resulting execution task graph is different. All kernel commands are waiting the initial write to buffer X, the kernels have no dependencies between them and the final rewrite to buffer X is waiting for all kernels to complete.

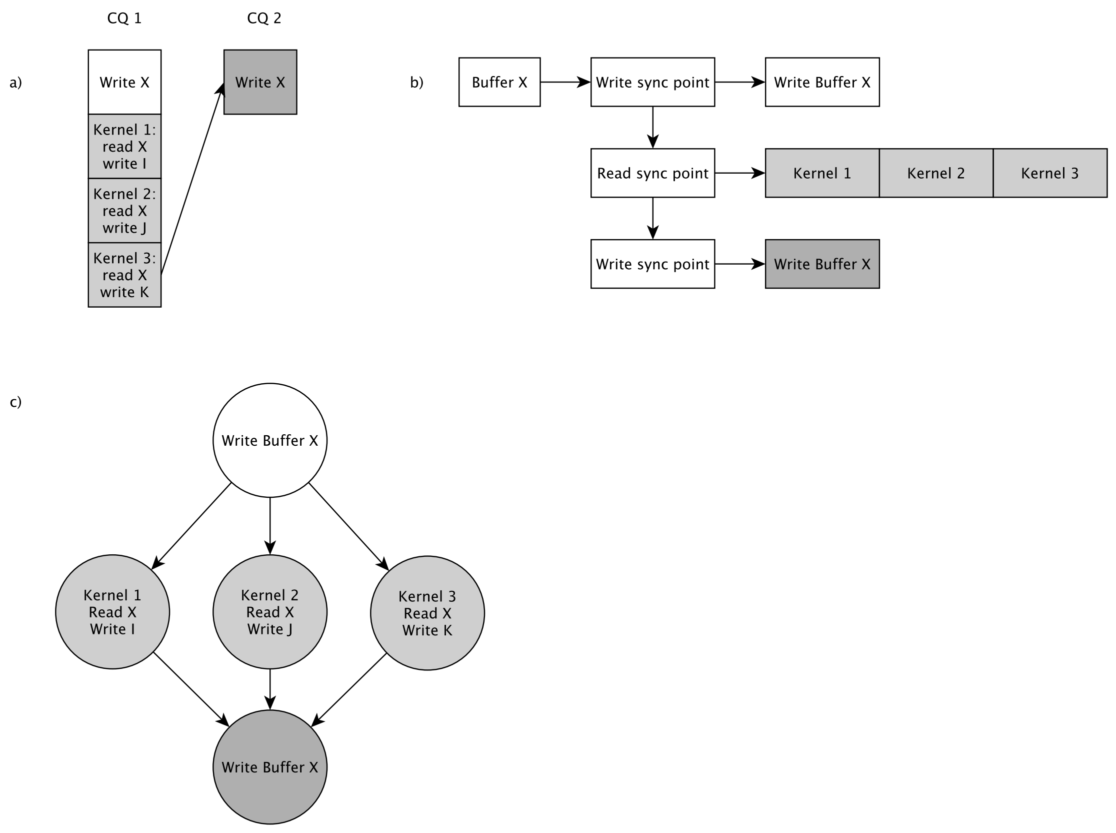

Figure 4.2: a) Commands enqueued to in-order command queues 1 and 2. b) The synchronisation data structure of buffer X when all commands are enqueued but none have completed yet. c) The actual command dependencies where available task parallelism is extracted while in-order semantics are preserved

4.3 Support for Shared Memory Address Spaces

A concept of shared global memory address space was introduced in pocl to support such platforms. Many MPSoC platforms and for example AMD APU platforms have a shared memory address space in which all the devices operate. Shared memory allows a zero-copy data passing method to be used where the data is passed as pointers to the shared memory address space.

On the Host layer the pocl runtime uses the global address spaces when allocating memory for OpenCL memory objects on the device side. Memory is not allocated per device, but per global address space. By this approach devices in the same address space may share the memory object allocations and avoid copying of data when they use the same memory objects. This complies with the OpenCL standard since the standard states that memory objects state is global in the context and all read and write operations to memory objects must be synchronised.

On the pocl Device layer side in device driver initialisation phase the drivers may declare a global address space for their devices or assign themselves in the same global address space with another device.

4.4 Example Implementations

Existing device interfaces *basic*, *pthread* and *ttasim* were updated to be compatible with the new driver-API extension for implementing out-of-order command queues. Also Kalray's MPPA-256 device interface is presented which is the first commercial many-core application for the framework. The designs of these interfaces are presented along with the mapping of the framework functions for out-of-order execution features.

4.4.1 Device Interface for pthread

Device interface *pthread* is an example implementation for multi-threaded homogeneous device platforms. POSIX threads are to execute multiple work-groups or multiple commands in parallel in their own threads. Currently pthread interface works with all x86 processors and with some ARM platforms with any core count.

The device layer layout for pthread interface is in Fig. 4.3. The implementation is based on a thread pool including one worker thread per hardware thread by default. The worker threads have their own work queues and each worker takes care of the command scheduling and execution independently. The commands are distributed evenly to the workers by round robin scheduling. The work-groups of the kernel are statically scheduled to one worker thread with the same round-robin scheduling. When there are multiple kernels to be executed, pushing all work-groups of a kernel to the same worker reduces cache misses when one core processes a limited amount of buffers at the time. Dynamic load balancing is applied in a form of work stealing. Using own work queues for workers reduces the lock contention in synchronisation when threads mainly operate with their own queue synchronised with a queue specific lock.

The submit() implementation bypasses the host side command queue and directly transfers commands from to the device driver. If the command has events to be waited it is stored into a list of pending commands. If the command is free to be executed it is pushed to some worker threads own work queue and that thread is woken up to execute the command. Since commands are executed as soon as possible, flush() is not used and the function has only an empty implementation.

When a worker thread completes a command it updates event status on the host side and calls the default broadcast() implementation to inform the drivers waiting this event by calling notify() for the required drivers. The driver also notifies itself if needed. By notifying itself, new released commands are easily got into execution.

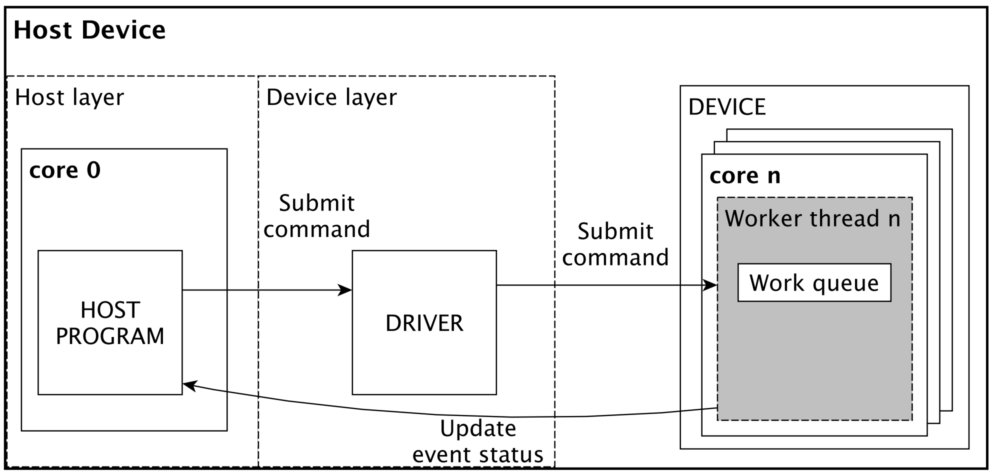

Figure 4.3: Device layer layout of pthread device interface.

The notify() checks if waiting command becomes ready to be executed. Commands ready for execution are pushed to the worker threads.

The join() implementation checks if the given command queue is empty and in case of an empty queue, join returns immediately. If there are unfinished commands in the queue host application is put to sleep. Whenever a worker thread finishes the last command from any command queue it signals host to wake up. Join() checks if the desired command queue was finished and goes back to sleep or return from execution. When the host application thread is sleeping more CPU time is available for worker threads.

Pocl enables multiple instances of pthread to be used as separate devices visible to OpenCL runtime. In case of multiple pthread devices the same thread pool is shared with all device instances because computational resources are the same regardless of the number of “virtual” devices. When multiple instances of pthread are used the devices inherently share the physical memory since all devices operate on the same processor. At driver initialisation phase pthread drivers assign themselves to the same address space, the first pthread instance declares a new address space and following instances join the address space. Motivation for using multiple pthread instances is mainly simulating a multiple multi-core processor platform for a OpenCL application. Using multiple pthread devices on a single processor does not grant any performance gain.

4.4.2 Device Interface for ttasim

Device interface *ttasim* is a driver for *Transport Triggered Architecture (TTA)* processor simulator. TTA processors are custom ASIP devices that are designed with

TCE toolset. The simulator is modelling a device that operates in its own global address space. Multiple ttasim devices may access the same shared global address space. The devices are capable of independent command queue execution and when there are multiple ttasim devices operating in the same address space they are able to synchronise events directly with each other. The device layer layout of ttasim devices is in Fig. 4.4.

The driver consist of two main components. There is a driver thread per ttasim driver instance working asynchronously from the host. The driver thread's tasks are to execute memory commands, to compile and upload kernel commands to the device and to handle synchronisation of events with the host. The second component is the TTA simulator thread for running the simulator itself. There is only one ttasimulator thread for all ttasim drivers advancing all simulations in a lock step mode. The lock step execution is used to ensure consistent view of the global memory for all devices.

The driver thread uploads submitted kernel commands to devices own command queue. TTA devices own main program controls the device and schedules execution of commands pushed to the device side command queue. When a command is completed device sets commands events status accordingly to the predefined location in the global memory. By polling the contents of the memory the driver thread receives information about completed commands.

The main program running on TTA does not support dynamic loading of kernels. Kernels must be compiled to the same image with the main program and then

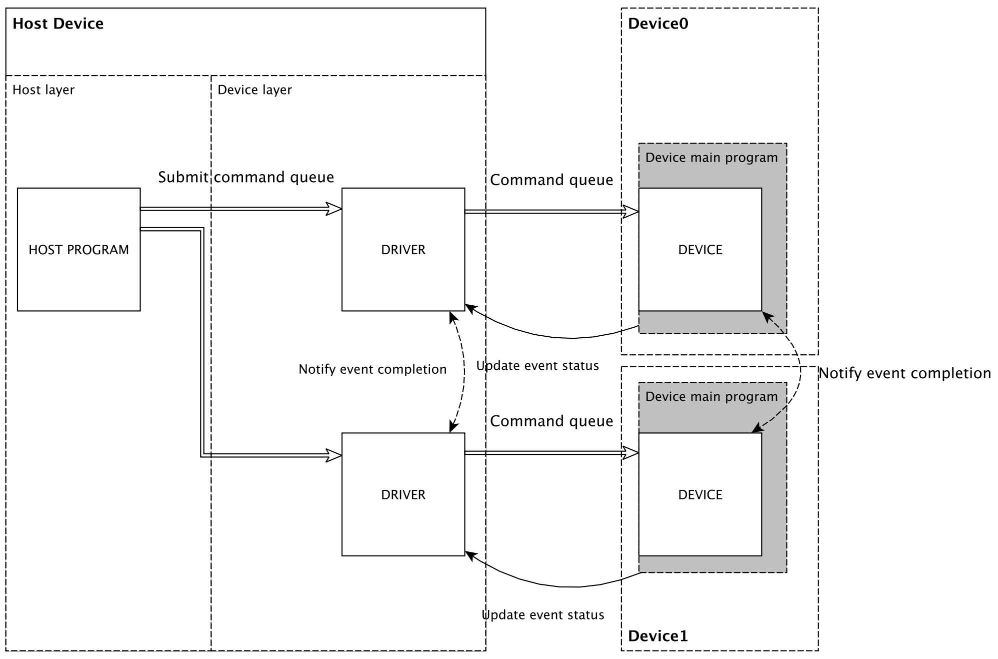

Figure 4.4: Device layer layout of ttasim devices.

uploaded to the device. This feature needs to taken into account in the driver to avoid compiling and uploading new image as much as possible. Thus, submit() handles kernel commands and other commands differently. All kernel commands are stored to the host side command queue to wait the command queue to be flushed. When command queue is flushed all kernel commands can be compiled to the same program image at once. Other commands that are not ready to run are stored to the drivers list of pending commands. Commands ready to run are pushed to the drivers ready queue and the driver thread is woken up. The driver thread then executes commands from the ready queue in first in, first out manner.

The flush() is a natural point in execution for kernels to be uploaded to the device. Implicit or explicit flush for a command queue usually marks a point in OpenCL application where application programmer wants the enqueued commands eventually to be executed. The flush() transfers all kernel commands from the given command queue to the drivers kernel command list and wakes up the driver thread. The driver thread waits for the device to complete all previously uploaded kernel commands from its command queue and then uploads new kernel commands to the device's command queue. Before uploading new commands a new main program image is compiled if the kernel command list contains kernels that are not present in the current device program image. Relaunching all or a subset of already uploaded kernels does not require recompilation and only new command structures are uploaded to the devices command queue. This reduces compilation overhead if the OpenCL application is not flushing command queues after every different kernel enqueue.

When the driver or the device completes a command the driver thread uses default broadcast() implementation to inform drivers about completion. The notify() checks commands event wait list if command is ready to run. Memory commands that become ready to run by notification are transferred from command list to the ready list and the driver thread is woken up. For kernel commands that reside in host command queue, only event wait list is updated. The kernels that are already uploaded to the device are notified by setting event status to complete in predefined location in global address space.

The join() wakes up the driver thread and puts host side into sleep if there are commands yet to be finished in the given command queue. The driver thread releases join() from sleep whenever all commands from any command queue are completed. When join() wakes up it checks the command queue and goes back to sleep or returns from execution.

When multiple TTA simulators are used they may share a host accessible global address space with each other. The global address space is different from the host address space. The address space is used for passing data between TTA devices

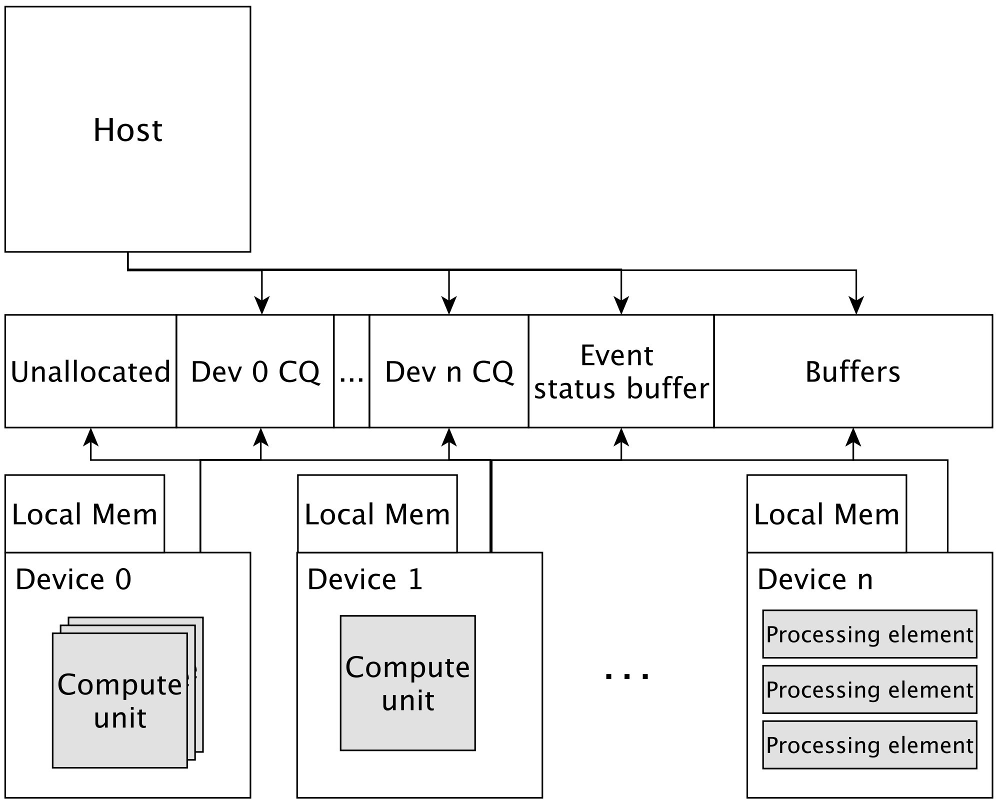

Figure 4.5: Platform layout of shared address space multi-device TTA.

and for synchronisation of events between devices themselves. Memory for OpenCL buffer and image objects are allocated collectively for all TTA devices thus copying data is only required when synchronising data between the host address space and the TTA global address space.

Shared memory platform layout is illustrated in Fig. 4.5. In the beginning of the global address space there is an unallocated memory region which is reserved for the TTA devices own use and may not be used by the driver. Each device driver allocates memory from the global address space for the device side command queue to be used by their own TTA.

The device side command queue is implemented as a static array of command structures. The device driver copies new kernel commands, including a status field, to the free slots of the queue. The device polls the status fields of the command slots in its command queue to determine if there are new commands to be executed. When a command is completed the device sets commands status field to indicate that this command is completed and this slot is free to be reused.

Device-device and host-device communication is managed via the global address space by reserving a memory region for storing event statuses. In that region an event owning device may update the events status and others may check the progress

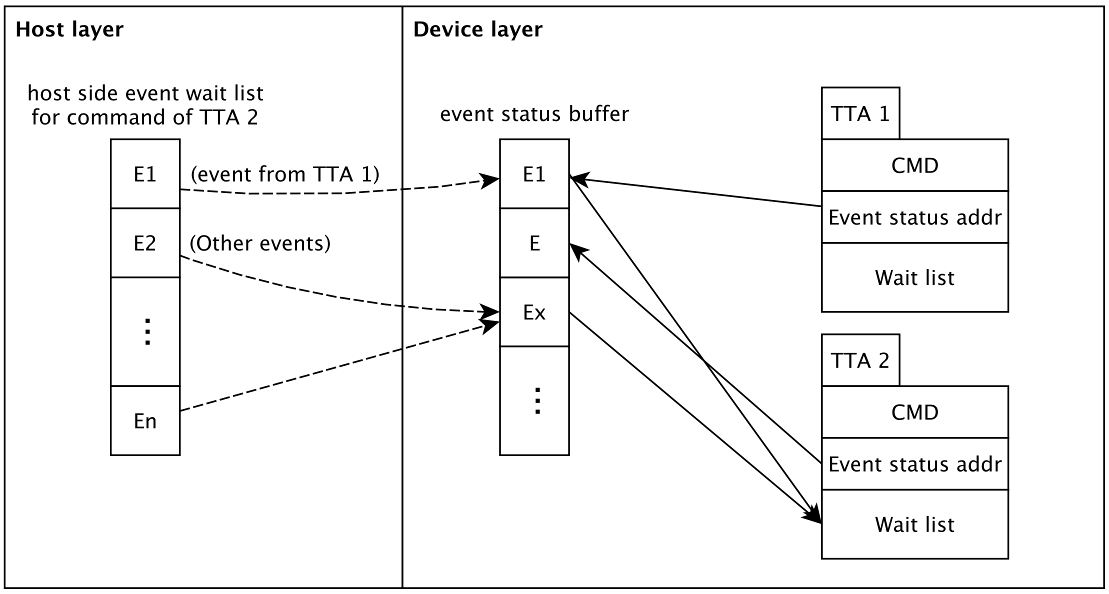

Figure 4.6: Example of event status buffer use.

of the event they are waiting.

Event status buffer is implemented as a static array of event status fields and is managed by the device drivers. Use of the event status buffer is described in Fig. 4.6. When creating command struct to be uploaded to the device, the driver reserves a free slot from the event status buffer to be used by the TTA to notify completion of the command. Next the driver goes through the events wait list. If the wait list contains an event owned by a device in the same global address space (including itself) then the event is removed from the host side wait list and an event status buffer slot is reserved for that event if it does not have one yet. The address of the event status slot is then added to device side wait list of the command being created. If the host side wait list contains any events after picking the same address space events, one more event status slot is reserved and added to device side wait list. This final status slot is for the device driver to indicate that all other events from outside the TTA global address space have been completed.

In the example TTA 2 is waiting one event from TTA 1 and n events from outside the TTA shared global address space. The device driver reserves slots from event status buffer for all the required events. TTA 1 updates the event status in location E1 when the command is completed. The device driver updates the location Ex when all “other” events have completed. TTA 2 polls the locations E1 and Ex and when their status is set to complete, the command can be executed and the events completion is acknowledged to the location E. The static reservation of the command buffer slots and the event status buffer slots in the driver side is required

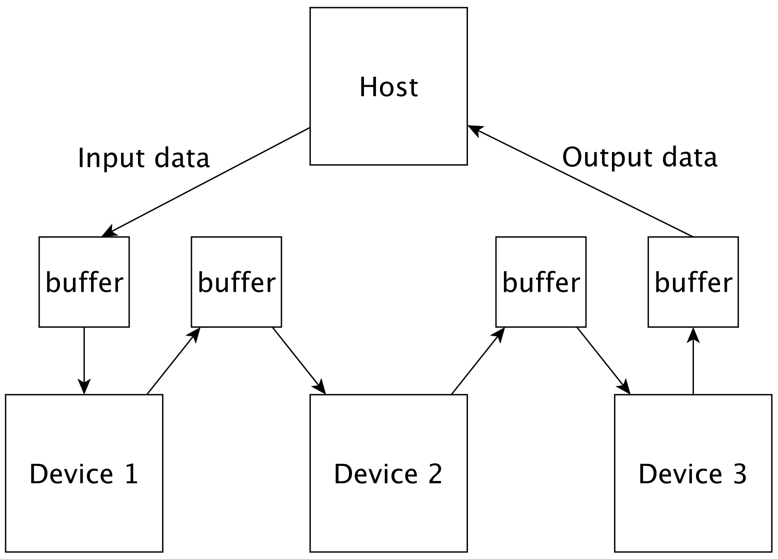

Figure 4.7: Example of device pipeline implemented using buffers.

because at the time of writing the required atomic operations were not available for synchronised memory accesses from the host to the TTA global address space.

In the shared global memory TTA devices can be used to form a device pipeline. OpenCL memory buffers in the shared memory can be utilised by using one devices kernel output buffer as other devices kernel input buffer and synchronise kernel commands with an event. Example of buffer and event based pipeline in Fig. 4.7. Applications able to benefit from the device pipeline are for example modems and video processing tasks that are based on a data stream and distinct phases of execution that needs to be performed before data proceeds to another phase. In OpenCL context each TTA is considered as a device of its own and each TTA can be customised for optimal performance for each step in the pipeline.

4.4.3 Device Interface for basic

Basic device interface is a limited CPU device implementation also suitable for barebone standalone platforms. Kernel execution is performed by serialising work-group execution on a single processor core, thus no task level parallelism is provided by the hardware. The basic driver does not use any threads so all commands and other functionalities are performed as side effects of the driver API calls. This means that all the host side activity is blocked during the driver side operations. The device layer layout of basic interface is in Fig. 4.8.

The submit() transfers created command from host side to device driver bypassing the host side command queue. If submitted command is ready to run it is added to the drivers ready list and if the command is waiting events it is stored to the

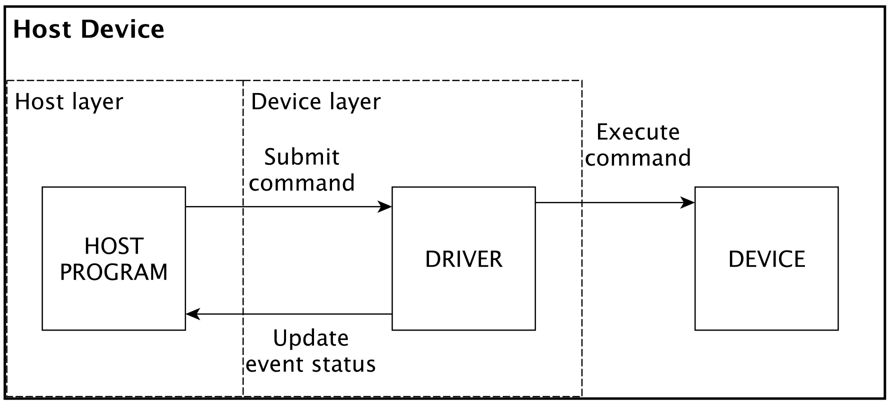

Figure 4.8: Device layer layout of basic device interface.

drivers pending list. Submit also runs a command scheduler which executes all ready to run commands from the ready list. Host program is blocked during command execution. Since commands are executed as soon as possible, the flush() has an empty implementation.

The default broadcast() implementation is used to inform other drivers. The notify() checks if the waiting command becomes ready for execution. When command is ready to run it is transferred from the command list to the ready list and the command scheduler is launched. The implementation of join() calls command scheduler repeatedly until all commands in the given command queue are completed.

4.4.4 Device Interface for Kalray MPPA-256

Kalray implemented OpenCL support for their MPPA-256 many-core by implementing a pocl device interface with the proposed out-of-order execution framework. The MPPA-256 is used as an accelerator type of device that concurrently executes kernels and kernel work-groups on compute clusters and work-items on cores of the clusters. The command scheduling is handled in the device but the device itself does the mapping of work-groups to the clusters.

The driver is constructed for MPPA chip on a PCIe development card. The device layer of MPPA-256 driver is in Fig. 4.9. The device driver consists of two memory transfer threads, kernel handling thread and event status thread. The devices DMA engine allows concurrent read and write operations, thus two threads. Memory transfer commands and kernel commands can be executed concurrently to hide memory transfer latencies caused by the PCIe bus. The kernel handling thread takes care of the kernel command scheduling. Signalling command completion back to the host is managed with interrupts and messages. The event status thread catches the messages and propagates the information to the host and to the other drivers by broadcast.

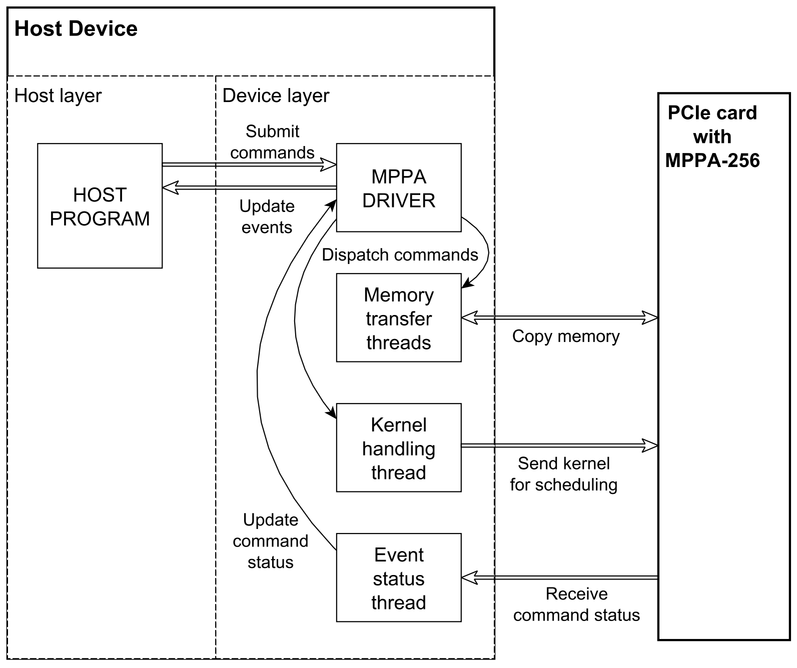

Figure 4.9: Device layer layout of Kalray MPPA-256 device interface.

The submit() hands the commands from host side to the device driver without storing commands to the host side command queue. The driver schedules commands to execution as soon as the command is free to be executed. Similarly to pthread interface, the flush() is not used.

When event status thread notices that a command has been completed, it calls the broadcast() to notify the required drivers about the completed command. The notify() checks if the notified event is ready for execution and if so, pushes the command to the scheduler.

The join() checks if the given command queue is already completed and returns immediately. When the queue has unfinished commands the driver puts host side to sleep. When event status thread notices that the last command from a command queue is finished, it signals the host to wake up. The join() then checks if the given command queue was finished and returns from execution or goes back to sleep.

5. EVALUATION

The performance of the out-of-order framework's task scheduling properties was evaluated using a multi-core processor and the pthread device driver implementation in pocl. The purpose of the benchmarks is to examine how the task scheduling runtime behaves when executing different task graphs in different scenarios and what are the trends when benchmark parameters are changed. All benchmark kernels are synthetic workloads that are not doing anything relevant, but are designed to stress test different aspects of the task level scheduling implementation. Synthetic benchmarks were chosen because suitable real application benchmarks for out-of-order queues and task parallelism in general, were not available at the time of writing.

Aspects that were tested are gathered in Table 5.1. Benchmarks are constructed to measure two aspects, performance of synchronisation and in-order/out-of-order command queues, and dynamic work balancing.

To exclude as much as possible unwanted runtime actions from the measurements all commands were enqueued to command queues and released for execution all at once. Only total execution time of all commands was measured in wall time. Every benchmark was executed five times and the best result was selected from each platform.

Benchmarks were performed on desktop computer with Intel Core i5-3470 3.2GHz processor quad-core processor with one HW-thread per core, and 16Gb memory.

Table 5.1: Benchmarks for the task scheduling runtime.

Benchmark	**Description**
In-order	In-order command queue handling speed
Out-of-order	Out-of-order command queue handling speed
Kernel-imbalance	Task level load balancing with out-of-order command queue and parallel kernels with different workloads
Work-group-imbalance	Work-group balancing with in-order command queue
Kernel&work-group-imbalance	Combines the two previous tests
In-order-parallelisation	Command parallelisation feature of the in-order queue

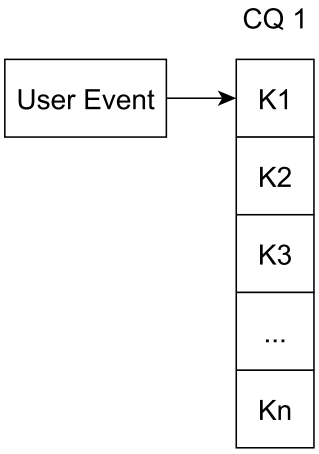

Figure 5.1: The kernel chain in in-order command queues.

Operating system was Red Hat Enterprice Linux release 6.5.

The first benchmarks *In-order* and *Out-of-order* measures the performance of the in-order and out-of-order command queue handling. The efficiency of the queues is relevant with small tasks. If pocl is used as a middleware for other task parallel programming model, the tasks might be quite small. For example in OpenMP and Cilk accelerated applications loop iterations can be declared as independently schedulable tasks. The loop iterations might be as short as few arithmetic operations and a couple of memory operations. In that context, the runtimes efficiency defines if it is beneficial to parallelise the loop at all.

The In-order benchmark configuration is illustrated in Fig. 5.1. The benchmark task graph and kernels are the same as in the Out-of-order benchmark, only command queues are switched to a single in-order command queue. The synchronisation is left for the in-order command queue itself. This benchmark also uses user event on first kernel commands to ensure that command execution does not start until everything is enqueued.

Results from the In-order benchmark are in Fig. 5.2. The benchmark was executed with different amounts of kernels to determine if the runtime behaves consistently with increasing task graph sizes. Results are displayed as μs per kernel. The execution time of a single kernel remains constant in all cases regardless of the task graph size. With one worker thread the execution time per kernel is about 0.86 μs which is a fairly good result. When the number of worker threads is increased the threads begin to interfere each other leading to significantly increased execution time. The execution time gets linearly worse with the number of the workers, leading to about 3.3 μs per kernel with four workers. Lock contention and/or polling of volatile pointers the work queues is causing a lot of overhead when the executed

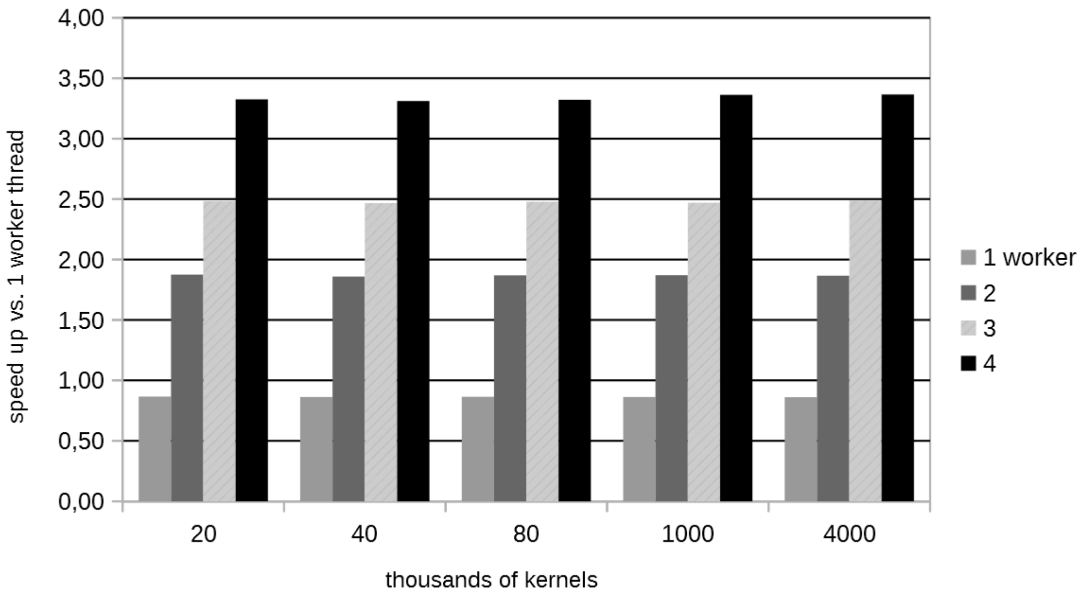

Figure 5.2: The results from the In-order benchmark.

kernel does not actually take any time.

The out-of-order command queue benchmark structure is illustrated in Fig. 5.3. The benchmark enqueues a serial task graph to two out-of-order command queues. In the task graph, a kernel in one command queue produces an input for the next kernel in the other command queue. Kernels are synchronised with events. The kernel contains only one integer addition to minimise kernels execution time from the measurement. The benchmark measures only the total execution time of the task graph. Premature command execution is prevented by synchronising the first kernel command to a user event. When all commands are enqueued the status of the user event is set to complete on the host side and command queues are flushed. Time measurement is started just before setting the user event status and is stopped after both command queues are finished.

The results from the Out-of-order benchmark are in Fig. 5.4. The results are similar to In-order benchmark. Only execution times per kernel are lower in every case, 0.7 μs with one worker and 3.2 μs seconds with four workers. The out-of-order queues seem to suffer from lock and/or memory access contention even more than the in-order queues. The In-order queue has the parallelisation feature which requires a little more work than the out-of-order queue which only uses the event wait lists in command synchronisation. With real life applications a reasonable workload decreases the lock contention and reduces the significance of other overheads.

The three following benchmarks are testing the dynamic work balancing capabilities of the thread pool. Good work balance is important to obtain good utilisation of

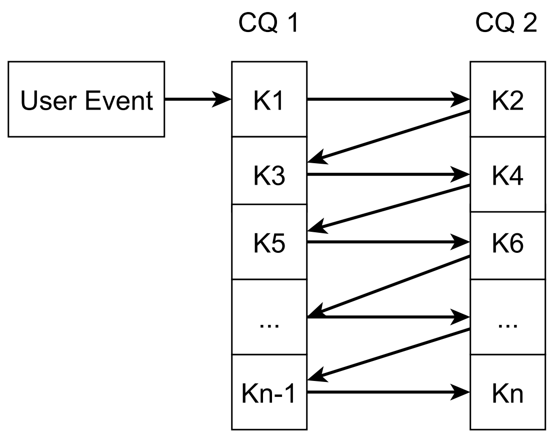

Figure 5.3: Kernel task graph interleaved in two command queues.

the computational resources available. The benchmarks measure the execution time of various task graphs with different workloads and with different number of worker threads. The results are shown as speedup of multiple worker threads, compared to a single worker thread. One worker is a good reference point because it represents the worst possible work balance on a multi-core processor. The benchmarks were executed with kernels that have varying workloads between the kernels and/or the work-groups in a kernel. The used kernel is more compute intensive than memory intensive to prevent cache misses from dominating the results.

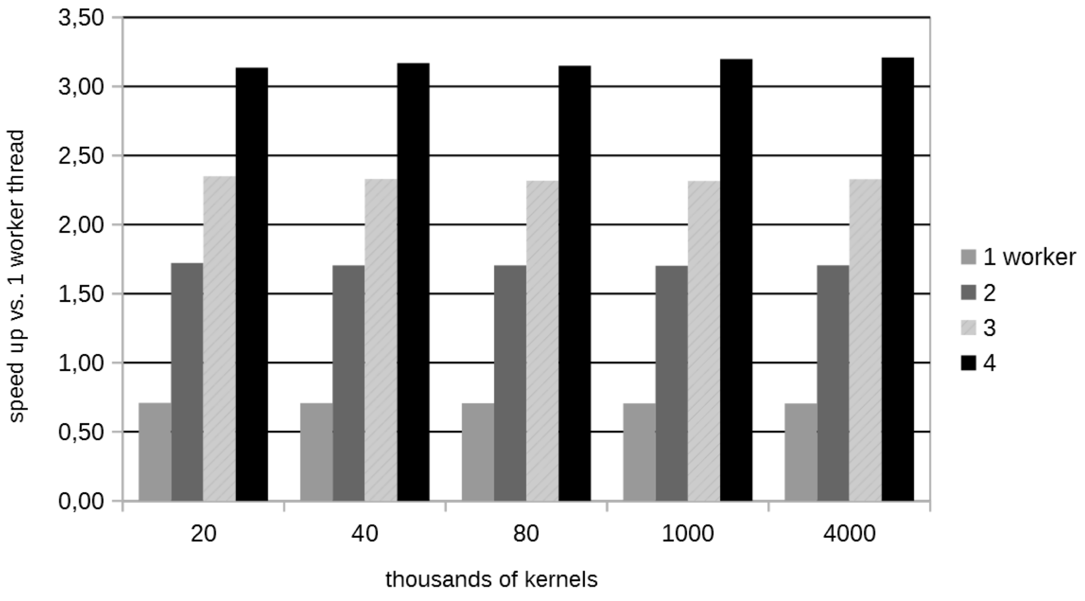

Figure 5.4: The results from the Out-of-order benchmark.

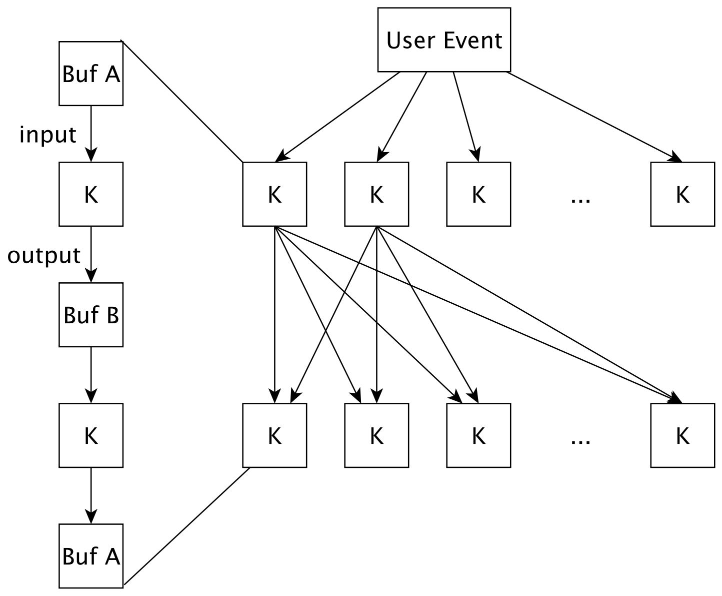

Figure 5.5: Benchmark for work balancing with imbalanced kernels. All parallel kernels are synchronised with kernels from the previous set of parallel kernels. every parallel kernel has different work load.

The *Kernel-imbalance* benchmark illustrated in Fig. 5.5. A batch of kernels with different workloads, in a single work-group, are launched in parallel and they produce input data for the kernel batch of the next iteration. The workload of the kernels varied by tuning the amount of data that kernels operate on; 8192 floats at most to 8192 / (amount of parallel kernels) floats at least. Every kernel in a batch is synchronised with every kernel in the previous batch. All kernels are enqueued to one out-of-order command queue. Amount of parallel kernels varied from one to 64. In each configuration, 10 000 kernel batches were launched.

The results of kernel imbalance benchmarks are in Fig. 5.6. With only one parallel kernel the four worker threads setup suffers a 9% performance loss. Also performance gains begins to diminish somewhere between 16 and 32 parallel kernels. This trend requires more investigation to find out why it is causing this behaviour.

The Work-group-imbalance benchmark is described in Fig. 5.7. Kernels were enqueued to an in-order command queue and kernels produce data for the the next kernel in order to force serial execution. Serial execution brings up the difference in the kernel execution time. Kernels were launched with multiple work-groups, each containing one work-item with different workload. Workload of the work-groups was varied by iterating through the input buffer from one to number of work-groups in a kernel times. In each configuration, 1 000 kernels were launched.

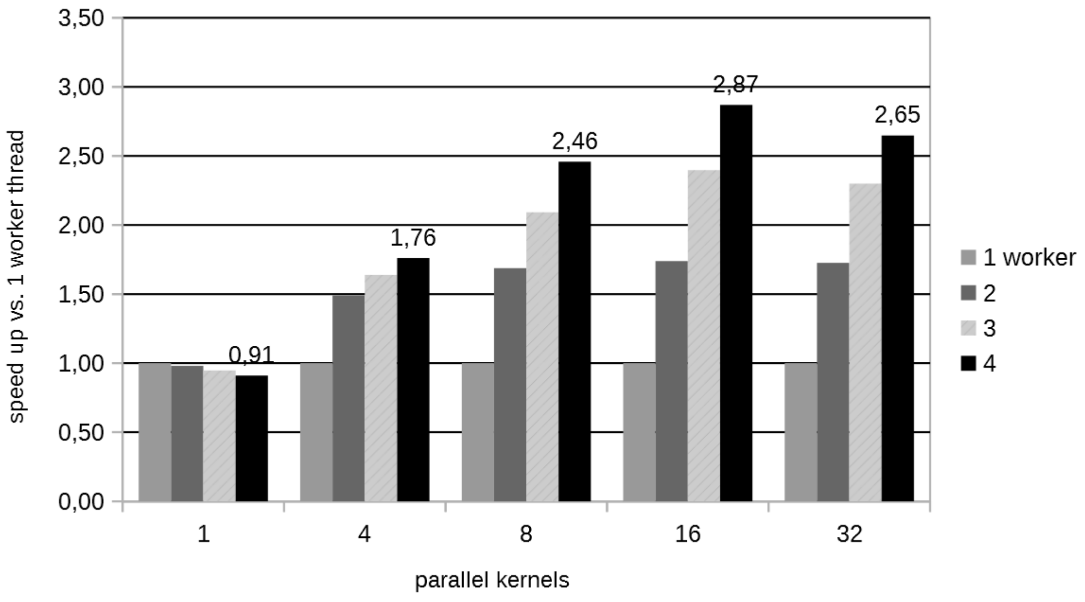

Figure 5.6: Kernel imbalance benchmark results.

The results of the Work-group-imbalance benchmark are in Fig. 5.8. Benchmark was measured with one to 64 work-groups per kernel. With one work-group per kernel again a 9% performance loss occurs when using four worker threads. The multi worker setups consistently increase their speedup when there is more work-groups to execute. This might be due to the fact that the less the work-groups the more idle time there is for workers that have no longer groups to execute. Especially

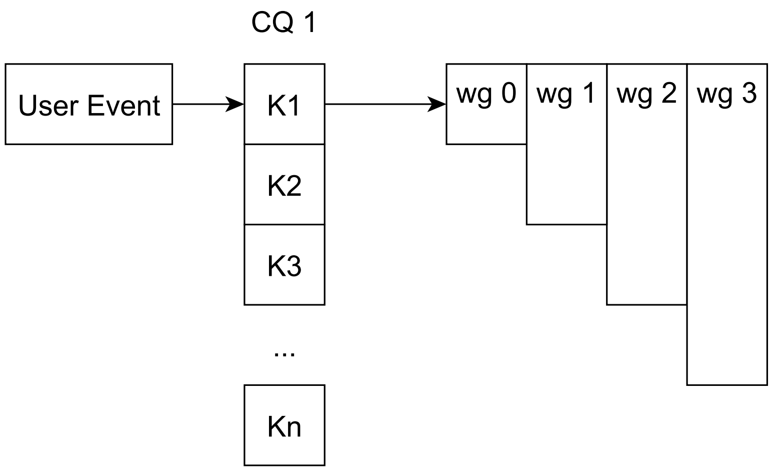

Figure 5.7: Benchmark for work balancing with imbalanced work-groups. Workload of each work group is dependent on the global id of the work item in the group.

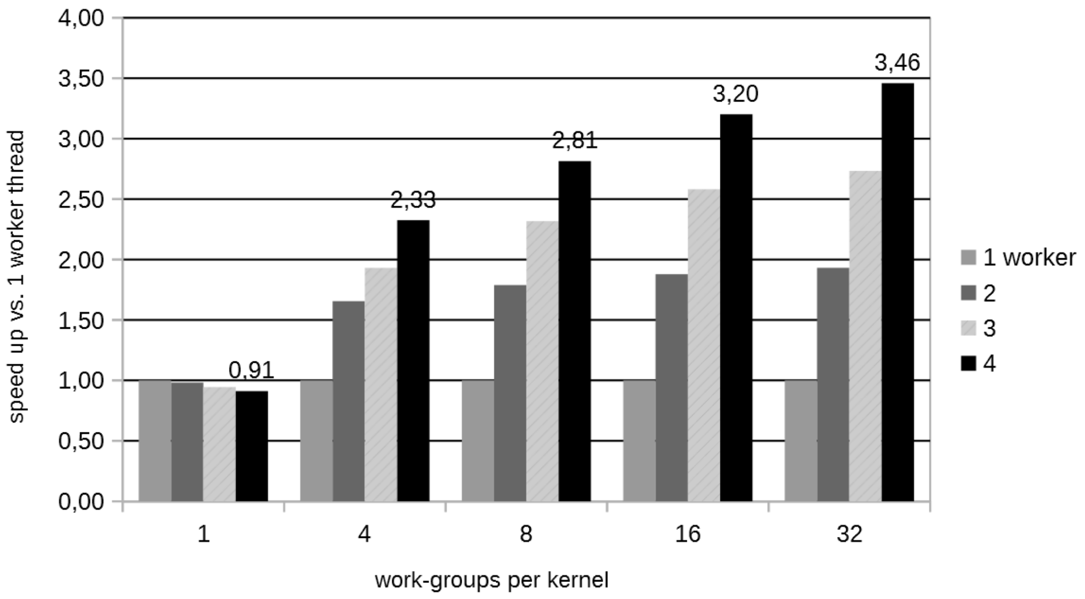

Figure 5.8: Results from imbalanced work-groups benchmark.

in this benchmark where all work-groups have a different work-load. For example the four groups case where the runtime of slowest work-group dominates the execution time since there is only one group per worker thread available and the completion of the kernel depends on the execution time of the slowest work-group.

Kernel and work-group imbalance benchmarks were combined to the benchmark Kernel&work-groups-imbalance benchmark to test both at the same time. Kernels were enqueued and launched just like in the kernel-imbalance benchmark. Inside the kernels, workgroups are imbalanced as in the Work-group-imbalance benchmark. In each configuration, 1 000 kernel batches were launched.

The results are in Fig. 5.9. The results have similarities to both work-group imbalance and kernel imbalance benchmarks. In 1/1 case, the usual 9% setback occurs with four worker threads. In this benchmark, the speedups are better than in the kernel or the work-group imbalance benchmarks. Four worker threads reaches rather good 3.75x speedup.

The in-order command queue parallelisation feature was also tested. The *In-order-parallelisation* benchmark configuration is in Fig. 5.10. First, a write command is enqueued targeting buffer X, which is defined as read-only from kernel point of view. After write command comes 10000 kernel commands reading the buffer X and each kernel writes to its own buffer, which is defined as write-only from kernel point of view. There are data dependencies only between the write command and each kernel command, thus kernels can be executed in parallel. The kernels used in this benchmark contain the maximum workload in one work-group so it is easier to

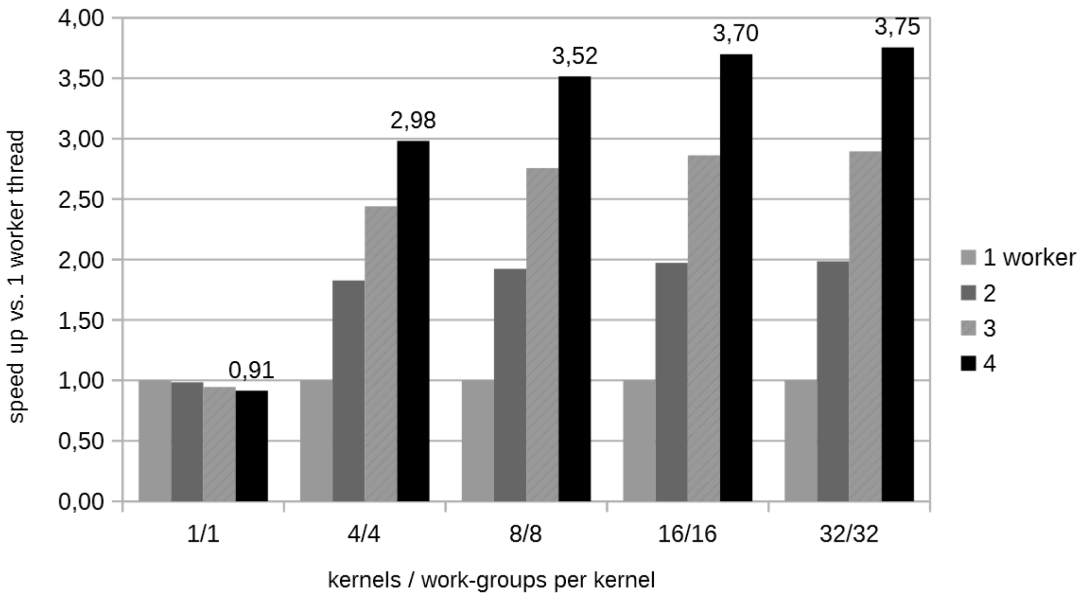

Figure 5.9: Results from kernel and work-group imbalance benchmark.

determine if execution is parallel or not.

The results of in-order command queue parallelisation are in Fig. 5.11. It is evident that unrelated commands are parallelised because the speedup is roughly the same as the number of worker threads used. Although, increasing the number of workers seems to diminish the gains, the speedup is nearly ideal 1,98x with two workers and 3.73x with four workers. Just for reference, the same benchmark was executed with only changing the input buffer from read-only to read-write. The

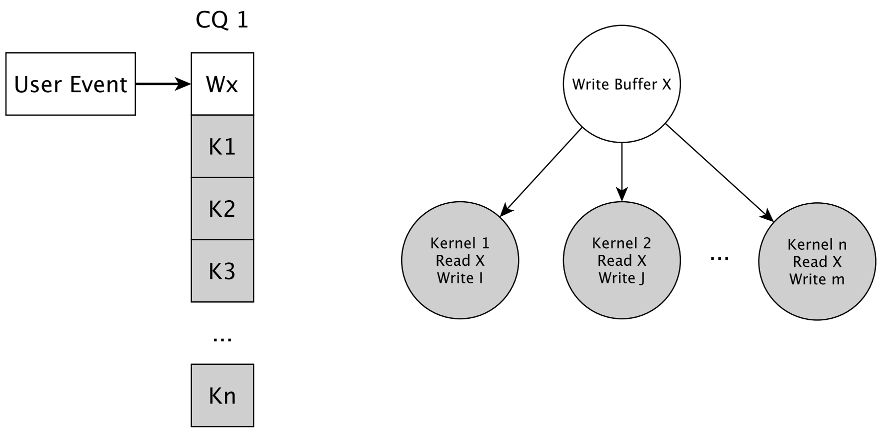

Figure 5.10: In-order-parallelisation benchmark. First, the is a write operation from the host to the buffer X and then multiple kernels reading that buffer.

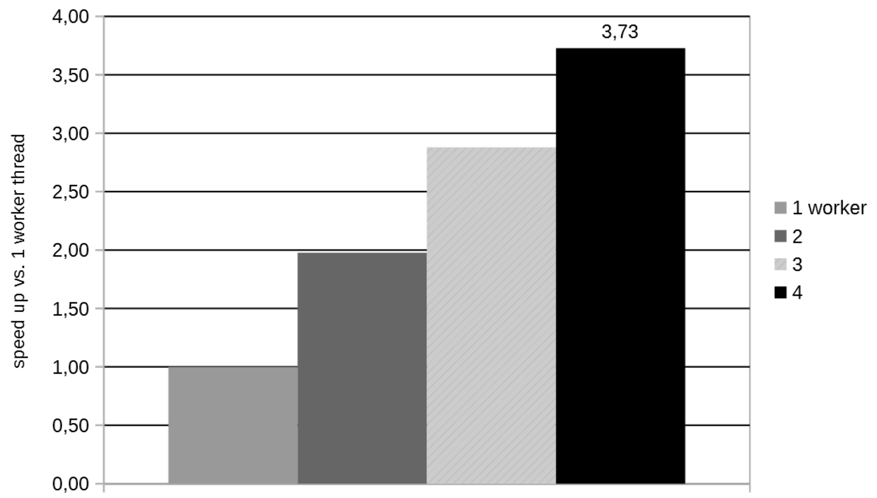

Figure 5.11: Results from in-order command queue parallelisation benchmark.

results are in Fig. 5.12. Kernels are executed serially and the usual 9% is yet again present.

At the time of writing, the only way to determine whether the kernel reads or writes to argument buffers, was to check the flags which was given when the buffer was created. Thus, kernel parallelisation cannot be done when read-write buffers are used.

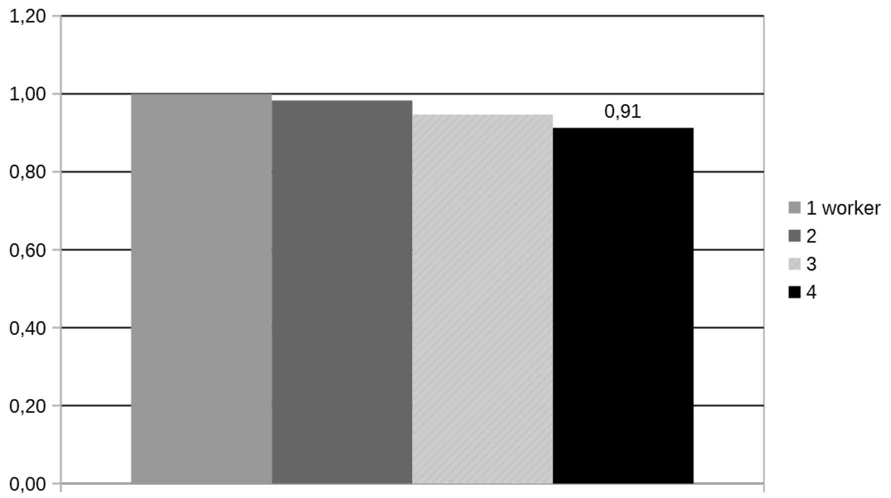

Figure 5.12: Results from in-order command queue parallelisation benchmark with read-write input buffer.

6. CONCLUSIONS

In this thesis a portable OpenCL out-of-order execution framework was introduced along with modified example device interfaces. The framework allows task scheduling features to be implemented on a large range of devices and platforms. The implementation was built on Portable Computing Language open source project. The work consists of an expansion to the driver API and modifications to OpenCL runtime implementation and existing driver interfaces.

This thesis covered a survey on common parallel computing platforms available. A brief introduction to programming of heterogeneous platforms in form of OpenCL was provided with emphasis on details relevant to this thesis. Also task graphs and task scheduling problem was covered since it is a crucial aspect in programming of parallel compute platforms.

The outcome was a simple framework with a simple interface in device driver API, which allows flexible synchronisation of commands on vast variety of devices and device interconnect topologies. The out-of-order framework was proven feasible by using the framework to successfully implement out-of-order features to the existing device drivers. Kalray's successful port on their MPPA-256 many-core platform shows the applicability of the framework to a manycore accelerator platform. The framework is suitable for devices for simple accelerators (basic interface), that are explicitly controlled by the device driver. The framework is also suitable for multi-core processors (pthread interface) and to independent devices, that operate in their own or shared memory and are able to perform tasks in cooperation with the device driver (ttasim interface).

The focus in the runtime's efficiency somewhat affected the frameworks usability. Trying to minimise the amount of actions in every turn lead to a bit too constrained event handling. The event handling might be a little confusing and error prone. This problem can be fixed by giving a device driver programmer more control over the memory management of the event objects, but it would introduce a minor extra overhead to the event handling. Before open source publishing of the framework this, option should be evaluated.

With synthetic benchmarks the task scheduling performance with multi-core driver was evaluated. The Kernel-imbalance benchmark revealed inconvenient performance loss when the amount of parallel kernels is large. The problem needs to be addressed because if pocl is used as an implementation layer for other programming models, these other models might be heavily task parallel. If the implementation chokes on greater amount of parallel kernels, the scalability is lost.

Parallelisation of the in-order queues introduced about 22% overhead compared to the out-of-order queue with a trivial kernel and one worker thread. The algorithm for keeping track of the data dependencies to buffers allocates and frees the memory of the synchronisation data structures and it takes some time. Some more static algorithm could be faster if applicable.

Multi-threading was shown to lower the performance with the serial task graphs that did not have any task level parallelism. With all four worker threads the performance was categorically 9% worse than single worker thread setup. This property needs to be optimised away because it ruins the scalability. A dynamic back-off technique could help. In this technique the idle threads run in a busy loop doing nothing with increasing time intervals. This lessens the lock and the memory bus contention, thus allowing the worker with an actual job to perform more efficiently.

Heterogeneous System Architecture (HSA) standard is an attempt to create an uniform way for devices in the heterogeneous platform to communicate with and to command one another in a shared virtual address space. This standard is interesting in the scope of this thesis for at least two reasons. First, in the software stack HSA situates in between OpenCL layer and the physical device layer. HSA continues standardisation towards the device details where OpenCL just assumes implementation defined behaviour. Second, OpenCL focuses on describing how the host manages the heterogeneous platform by enqueueing commands and synchronising events. The slave-master philosophy is still present in standard's 2.0 version. What comes to heterogeneous platforms, HSA sees the devices as peers that may communicate and dispatch work to each other. HSA defines device command queues and signaling/synchronisation methods at the level of a single bit. Any HSA conforming device may communicate with and enqueue commands to any other conforming device regardless of device type or vendor.

At the time of out-of-order execution framework implemention HSA standard was not released in time and the framework could not be implemented on that basis. When the standard is officially released the HSA *Architecture Queueing Language (AQL)* feature is likely to be adopted on pocl's command passing and command queues. This allows HSA compliant devices to implement P2P feature with any other compliant device. However, the HSA standard requires a shared virtual address space with certain atomic operations which limits the range of platforms and devices where the standard is applicable.

During the implementation some optimisation ideas came up. Maybe the most obvious place for improvement is the multi-core interface pthread. In addition to fixing discovered problems, there are other aspects. The data locality could be improved to avoid expensive cache misses. Executing tasks available at random is not optimal way to use a multi-core processor, even if it may yield a good utilisation. Tasks that operate on the same data, should be executed on the same core. If the data buffer fits entirely to core specific cache it is immediately ready to be used by the following task. By analysing task graphs this can be implemented. Kernel command objects in pocl contain the information about the buffers and it can be used in combination with the event dependency data to arrange more intelligent, cache friendly task distribution.

Another aspect for improving data locality is work-groups of a kernel. Work-groups that operate on data items located next to each other in the memory have inherently good mutual data locality. It might increase efficiency to execute these work-groups on the same compute unit in serial or even in parallel, than to scatter them on different compute units. It could be possible to analyse the kernel code and the intermediate representation of LLVM to find good execution order and compute unit for work-groups to obtain better data locality, thus more efficient cache usage.

BIBLIOGRAPHY

[1] E. Aragon, J. Jiménez, A. Maghazeh, J. Rasmusson, and U. Bordoloi. Pattern matching in OpenCL: GPU vs CPU energy consumption on two mobile chipsets. In *International Workshop on OpenCL*, May. 2014.

[2] D. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Journal of the ACM (JACM)*, volume 46, pages 720–748, Sept. 1999.

[3] D. Blumofe and C. Leiserson. The performance of spin lock alternatives for shared-money multiprocessors. In *Parallel and Distributed Systems, IEEE Transactions*, volume 1, pages 6–16, Jan. 2002.

[4] M. Daoud and N. Kharma. A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. In *Journal of Parallel and Distributed Computing*, volume 68, pages 399–409, April. 2008.

[5] A. Downey. *The Little Book of Semaphores Version 2.1.5.* 2008. Web page: http://www.greenteapress.com/semaphores/.

[6] X. Geng, G. Xu, and Y. Zhang. Dynamic load balancing scheduling model based on multi-core processor. In *Frontier of Computer Science and Technology (FCST)*, pages 398–403, Aug. 2010.

[7] X. Geng, G. Xu, and Y. Zhang. Fast and effective task scheduling in heterogeneous systems. In *Heterogeneous Computing Workshop*, pages 229–238, Aug. 2010.

[8] K. George and V. Venugopal. Design and performance measurement of a high-performance computing cluster. In *Instrumentation and Measurement Technology Conference (I2MTC)*, pages 2531–2536, May. 2012.

[9] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Elsevier, 2013.

[10] P. Jääskeläinen, C. de la Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A performance-portable OpenCL implementation. In *International Journal of Parallel Programming*, Aug. 2014.

[11] Kalray Inc. MPPA MANYCORE, 2012. Web page: http://www.kalray.eu/products/mppa-manycore/mppa-256/.

[12] Khronos Group. *OpenCL Specification v2.0*, March. 2014.

[13] Qualcomm Technologies, Inc. . Qualcomm snapdragon 800 processors, 2013. Web Page: https://www.qualcomm.com/media/documents/files/snapdragon-800-processor-product-brief.pdf.

[14] L. Sha, T. Abdelzaher, A. Cervin, T. Baker, A. Burns, G. Buttazo, M. Caccamo, J. Lehoczky, and A. Mok. Real time scheduling theory: A historical perspective. In *Journal Real-Time Systems*, volume 28, pages 101–155, Dec. 2004.

[15] G. Wang, B. Rister, and R. Cavallaro. Workload analysis and efficient opencl-based implementation of SIFT algorithm on a smartphone. In *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 759–762, Dec. 2013.

[16] G. Wang, Y. Xion, J. Yun, and R. Cavallaro. Accelerating computer vision algorithms using opencl framework on the mobile GPU - a case study. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference*, pages 2629–2633, May. 2013.

[17] Y. Wang and K. Cheng. Energy-optimized mapping of application to smartphone platform - a case study of mobile face recognition. In *IEEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 84–89, Dec. 2011.

[18] Y. Wang and K. Cheng. Energy and performance characterization of mobile heterogeneous computing. In *IEEE Workshop on Signal Processing Systems (SiPS)*, pages 312–317, Oct. 2012.