LEVERAGING DATA-FLOW INFORMATION FOR EFFICIENT SCHEDULING OF TASK-PARALLEL PROGRAMS ON HETEROGENEOUS SYSTEMS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2020

Osman Seckin Simsek

School of Computer Science

Contents

Abstract **8**

Declaration **9**

Copyright **10**

Acknowledgements **11**

1 Introduction **12**

1.1 Motivation 13

1.2 Contributions 14

1.2.1 Contributions in the GPU Context 14

1.2.2 Contributions in the FPGA Context 16

1.3 Practical Contributions 17

1.4 Thesis Outline 17

1.5 Publications 18

2 Background **19**

2.1 Parallel Architectures 19

2.1.1 Architecture Models 20

2.1.2 Memory Systems 20

2.1.3 Multi-core CPUs 21

2.1.4 Heterogeneous Many-core Systems 22

2.2 Parallel Programming Models for Many-core Architectures 24

2.2.1 Heterogeneous System Programming 24

2.2.2 GPU Programming Models 25

2.2.3 OpenCL Programming Model 25

2.2.4 FPGA Programming Models 29

2.2.5 Task-Based Programming Models 30
2.3 Related Work 33
2.3.1 XKaapi 33
2.3.2 OmpSs 36
2.3.3 StarPU 39
2.3.4 QUARK 41
2.3.5 Discussion 42
2.3.6 The Effect of Task Granularity 43
2.4 Summary 44

3 OpenStream 46
3.1 Terminology 46
3.2 Syntax and Semantics 48
3.2.1 Declaring Streams 48
3.2.2 Declaring Views 49
3.2.3 Task Creation 51
3.2.4 Tick Construct 53
3.2.5 Taskwait Construct 54
3.3 Execution Model 54
3.3.1 The Workers and The Scheduler 54
3.3.2 Data Structures 56
3.3.3 Memory Management 58
3.3.4 Dependence Management 59
3.4 Compilation of OpenStream Programs 68
3.5 Summary 70

4 Extending OpenStream for Heterogeneous Systems 71
4.1 Extension for GPUs 72
4.1.1 Execution Model of OpenStream-GPU 72
4.1.2 Syntax of OpenStream Programs Employing GPUs 74
4.1.3 Run-time Implementation 77
4.2 Extension for FPGAs 80
4.2.1 Execution Model of OpenStream-FPGA 80
4.2.2 Syntax of OpenStream Programs for FPGA Acceleration . . . 82
4.2.3 Run-time Implementation 84
4.3 Summary 86

5 Dynamic Scheduling on GPUs **87**
5.1 Dynamic Scheduling of Tasks on GPUs 88
5.2 Execution of Tasks on GPUs 90
5.2.1 Accounting for Compute Unit Asymmetry 91
5.3 Experimental Setup 92
5.3.1 Hardware Environment 93
5.3.2 Experimental Baseline 93
5.3.3 Benchmarks 94
5.4 Results 95
5.4.1 Data Locality: Bandwidth vs. Latency 95
5.4.2 Impact on Performance 98
5.4.3 Execution Breakdown 100
5.4.4 Comparison with XKaapi Run-time 102
5.5 Conclusion 104

6 Dynamic Task Scheduling on FPGA-SoCs **105**
6.1 Dynamic Scheduling on FPGAs 106
6.1.1 Scheduling Tasks on FPGA Accelerators 107
6.1.2 Task Execution on FPGA Accelerators 108
6.2 Experimental Setup 111
6.2.1 Hardware Environment 111
6.2.2 Benchmarks 111
6.3 Results 112
6.3.1 Task Distribution Analysis 112
6.3.2 Impact on Performance 116
6.4 Conclusion 117

7 Conclusion and Perspectives **119**
7.1 Summary 119
7.2 Contributions 120
7.3 Future Directions 122

Bibliography **124**

Word Count: 33036

List of Tables

5.1 Execution times of OpenStream and XKaapi run-times for different matrix sizes for Matrix Multiplication and Cholesky 103

6.1 Different configurations of accelerators for different block sizes in Cholesky 114

List of Figures

2.1 Architectural differences between CPU and GPU 23
2.2 OpenCL platform model 26
2.3 OpenCL memory model 28
2.4 HLS design flow 29
2.5 Relationship of the run-time with the other components of a system . 32

3.1 Illustration of stream accesses with burst and horizon 50
3.2 Tasks accessing streams using views and the corresponding dynamic task graph 52
3.3 Persistent workers with their data structures and worker placement in OpenStream 55
3.4 Data structures of OpenStream run-time 56
3.5 Structure of the memory pool 58
3.6 Dependence resolution of two producers and one consumer 62
3.6 Dependence resolution of two producers and one consumer contd. . . 63
3.7 Dependence resolution of one producer and two consumers using broadcast operation 67
3.8 Compilation steps of OpenStream programs 69

4.1 Persistent workers extended for GPU support in OpenStream 73
4.2 Extended Frame data structure and cl_data structure 78
4.3 Extended view data structure 79
4.4 Persistent workers extended for FPGA accelerator support in OpenStream 81
4.5 Extended Frame data structure for FPGA use 84
4.6 FPGA accelerator data structure to manage each accelerator 85

5.1 Total amount of data transferred (normalized to the baseline XKS) . . 96
5.2 Number of tasks executed on the GPU (normalized to the baseline XKS) 97

5.3 Number of transfers between the host and the device (normalized to the baseline XKS) 98
5.4 Execution time (lower is better, normalized to the baseline XKS) . . . 99
5.5 Breakdown of time spent in GPU execution, showing the amount of overlap between computation and communication 101
5.6 Performance in GFLOPs 103

6.1 Percentage of tasks offloaded to accelerators in Matrix Multiplication 113
6.2 Percentage of tasks offloaded to accelerators in Cholesky 114
6.3 Percentage of tasks obtained through work stealing, work pushing or dependence satisfaction in Cholesky 115
6.4 Performance of Matrix Multiplication (normalized to the baseline CPU-only) 116
6.5 Performance of Cholesky (normalized to the baseline CPU-only) . . . 117

Abstract

LEVERAGING DATA-FLOW INFORMATION FOR EFFICIENT SCHEDULING OF TASK-PARALLEL PROGRAMS ON HETEROGENEOUS SYSTEMS

Osman Seckin Simsek

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy, 2020

Writing efficient programs for heterogeneous platforms is challenging: programmers must deal with multiple programming models, partition work for CPUs and accelerators with different compute capabilities, requiring different amounts of parallelism, and manage memory in multiple distinct address spaces. Consequently, programming models which only require expressing parallelism and data dependences can not only unburden the programmer from these technical decisions, but also increase code and performance portability.

Past research has identified data-flow task parallel programming models are a good fit for increasing the programmer productivity as well as unleashing the parallel processing power of massively parallel heterogeneous architectures. Especially, the dependence information readily available in the modern data-flow task parallel programming models can be exploited for better task and data placement decisions to achieve higher performance and portability.

This thesis focuses on the efficient scheduling of data-flow task parallel programs to a wide range of heterogeneous architectures from multi-core CPUs combined with discrete GPUs to multi-core CPUs with FPGA in system-on-chips. The proposed strategies balance the workload across heterogeneous resources, while simultaneously leveraging the task dependence information available in OpenStream–a platform-neutral and heterogeneity-agnostic data-flow programming model– to optimize the scheduling of tasks and data transfers.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.manchester.ac.uk/library/aboutus/regulations`) and in The University's policy on presentation of Theses

Acknowledgements

First of all, I would like to thank my principal supervisor Dr. Antoniu Pop for his constant support and invaluable guidance through my PhD. At the same time, I would like to thank my second supervisor Dr. Mikel Luján for his support and encouragement.

Next, I would like to thank all members of APT group here at the University of Manchester for providing a friendly environment. I also would like to thank my family for their unconditional love.

Finally, I would like to thank my wife Merve for supporting me through thick and thin in this PhD.

Chapter 1

Introduction

In the mid-2000s, the microprocessor industry went through a paradigm shift from aggressive single core processors, to more energy efficient multi-core designs due to power and temperature limits. Until 2000s, microprocessor development went through an era of sequential performance gains through aggressive clock frequency scaling and micro-architectural improvements, pushing the power and temperature boundaries to the point where increasing the clock frequency only lead to limited gains, forcing the industry to find alternative solutions and thus, the era of multi-core processors emerged.

However, the ever-increasing need for more computing power persists and the focus now lies on increasing parallel performance by using many-core accelerators in a heterogeneous setup. The new architectural trend has multi-core processors in its heart, combined with accelerators such as GPUs and FPGAs. Although multi-core chips try to maintain the execution speed of sequential programs by architectural improvements, accelerators favor the execution throughput.

The shift from homogeneous to heterogeneous architectures, in many cases, caused widely used algorithms to be rethought and rewritten to take advantage of heterogeneous architectures containing multi-core and many-core devices. However, due to the large number of available devices and short release cycles of systems with even higher processing units, parallel applications are required to be portable across multiple systems. Since parallel programming is more complex than sequential programming, parallel programming models must provide improved productivity, reduced implementation overhead and efficient execution on a wide variety of architectures.

Task-based programming models respond to these challenges by abstracting from

the underlying architectural details, the operating system and system libraries. In addition to this abstraction, these programming models increase the productivity of the programmer who only needs to focus on the specification of the program by defining fine-grained tasks and task dependences. Generally, all the issues related to efficient interaction with system software, efficient exploitation of all computing resources and performance portability is handled by the run-time system. On heterogeneous architectures, this includes efficient delegation of work to multiple devices in conjunction with efficient memory management in both discrete and shared memory architectures. Providing efficient mechanisms for task and data placement is required for the implementation of task-parallel programming models to achieve high performance in heterogeneous architectures.

1.1 Motivation

The challenges in heterogeneous systems range from efficient delegation of work, to efficient data placement and handling memory transfers between devices in discrete memory platforms. Although there are a multitude of approaches for task and data placement for task-parallel programs, the purpose of this thesis is to explore the challenges and opportunities for the exploitation of data-flow information for making better scheduling decisions in heterogeneous architectures.

A major challenge in heterogeneous systems is the distribution of tasks to all available processing units efficiently since execution performance is highly affected by the differences in the computational capability of each device. On heterogeneous platforms, offloading a task to a device requires loading its compiled binary to the device at run-time and the movement of data in case the devices do not share the same memory. In addition to this, the management of underlying device software such as drivers and libraries create an overhead which can limit the execution performance.

Moreover, due to the differences in computational capabilities between devices in heterogeneous systems, classical load balancing approaches such as work stealing which do not account for the asymmetric compute capabilities are insufficient in exploiting full system resources. In a heterogeneous system where some of the computational units can provide higher throughput compared to the rest, better performing computational units must be occupied in order to increase the system utilization and overall performance.

The modern task-based run-time systems can cope with these challenges by providing transparency for heterogeneous architectures in addition to having fine-grained control over the task and data assignment of task and data to devices. Hence, all decisions regarding the data and task placement becomes the scheduler's responsibility in such run-time systems.

In order to deal with these challenges, the scheduler, which is the core of a run-time system that manages task and data placement must deal with the following issues for efficient execution of applications on heterogeneous systems:

- Resources that can be used independently, such as accelerators and the interconnect between host and device memory, should be used in parallel;
- The scheduler should improve execution performance on accelerators by allowing fully asynchronous operations;
- Assuming the throughput of an accelerator is substantially higher than a CPU, accelerator idle time has a higher impact on performance, so any accelerator present on the system should be prioritized when work is scarce, and unblocking tasks that can be offloaded to accelerators takes precedence over CPU tasks;
- Assuming data and task placement is transparent to the programmer and only the scheduler has fine-grained control over the assignment of task and data to devices, scheduling overhead must not become a bottleneck.

1.2 Contributions

In this thesis, we address the issues above by proposing novel scheduling strategies for heterogeneous systems made of; (1) multi-core CPUs with discrete GPUs and (2) multi-core CPUs with on-chip FPGAs sharing system memory.

1.2.1 Contributions in the GPU Context

In the GPU context, we present a novel scheduling and memory allocation technique for task-parallel data-flow programs executing on heterogeneous platforms composed of CPUs and GPUs that addresses the above challenges through: (1) dynamic load balancing across the host system and its GPUs; (2) a dynamic scheduling mechanism to decrease the number of task dependences crossing devices thus decreasing task

stalls on the GPUs; and (3) fully asynchronous task execution favoring overlapping of data transfers and computations on GPUs. Scheduling and memory allocation decisions are based solely on dynamic information about data accesses and data locality, readily available at execution time within the run-time systems of modern data-flow task-parallel languages. Our approach is fully automatic and thus unburdens the programmer of manual data partitioning and offloading to accelerators.

Our scheduling strategy uses run-time information about task dependences to make dynamic decisions on task and data placement. The approach not only favors data locality by subsequently executing tasks that exchange large amounts of data on the same device, but also keeps track of tasks with smaller dependences in order to prioritize them for offloading to the GPU, should it become idle. The scheduler takes into account which data is still in host memory and which data has already been transferred or will be transferred to device memory in order to identify and exploit opportunities for overlapping data transfers with GPU execution.

Our approach is capable of executing work on both CPUs and GPUs concurrently, with load balancing that dynamically reacts to the available parallelism and load throughout the execution. To this end, the proposed scheme employs work-stealing. Existing scheduling approaches for heterogeneous systems rely only on work-stealing for load balancing. Since work-stealing decreases the data locality, these approaches employ locality-aware techniques to compensate. On the other hand, our approach proactively places the data of future tasks before these become eligible by the scheduler in addition to work-stealing. Furthermore, our approach explicitly manages transfers between host and device memory, as data transfers between devices are the limiting factor for communication intensive workloads.

This work makes the following contributions in the context of GPUs:

- A new dynamic task scheduling and data placement heuristic for GPUs, leveraging task dependence information to enhance data locality.

- An integrated, joint scheduling of tasks and of data transfers between host and device, allowing for overlapping communication and GPU execution.

- A memory management strategy that incorporates task private memory regions in order to facilitate the memory allocation on the corresponding device.

- A dynamic load balancing technique that accounts for the computational capabilities of each device.

1.2.2 Contributions in the FPGA Context

In the FPGA context, we present a novel scheduling technique for task-parallel data-flow programs executing on heterogeneous platforms composed of multi-core CPUs and FPGAs sharing system memory that addresses the aforementioned challenges through: (1) task scheduling through work-pushing to increase the effective use of FPGA accelerators; (2) a dynamic scheduling mechanism to actively schedule dependent tasks on the FPGA accelerators, creating a pipelined execution on the FPGA; and (3) fully asynchronous task execution infrastructure including the management of accelerators for FPGA.

The proposed scheduler takes advantage of the data-flow information readily available in the modern data-flow task-parallel run-times to make scheduling decisions. Our approach can execute tasks on FPGA accelerators transparently and automatically, unburdening the programmer of accelerator management difficulties. The approach prioritizes the accelerators over execution of tasks on CPU cores to take advantage of the higher throughput of the accelerators as well as to create a pipelined execution of tasks on accelerators.

Our approach not only executes work on the FPGA accelerators, but also uses CPU cores to employ of all available computational device on the system. To our knowledge, there has not been any effort in the literature for dynamic task scheduling on FPGA accelerators using a user-level run-time system. The only close approach is OmpSs@Zynq [41] in which the main focus of the study is to generate FPGA accelerators during the compilation phase, combined with a static scheduling heuristic, not a dynamic scheduling approach while we propose a dynamic scheduling technique for heterogeneous systems containing CPU and FPGA on the same chip.

This work makes the following contributions in the context of FPGAs:

- A novel dynamic task scheduling heuristic targeting FPGA MPSoCs, leveraging task dependence information to increase the effective use of FPGA accelerators.

- A dynamic scheduling technique that accounts for the differences in computational capabilities of accelerators by prioritizing the execution on the FPGA.

- A scheduling heuristic taking advantage of the data-flow information available in the run-time to create pipelined execution on the FPGA accelerators.

1.3 Practical Contributions

The implementation and evaluation of the proposed contributions in GPU and FPGA contexts also led to several practical contributions. First, design and development of GPU and FPGA extensions for OpenStream, a state-of-the-art framework for task-parallel applications, have been undertaken. On top of the extended OpenStream, we integrated the proposed scheduling heuristics. The main reasons OpenStream is chosen is the proposed scheduling heuristics take advantage of decentralized memory management which OpenStream run-time provides through the use of task-private buffer usage. However the original OpenStream run-time did not offer support for heterogeneous systems, thus we implemented the support for GPUs and FPGAs as a basis for the proposed contributions of this thesis.

1.4 Thesis Outline

The outline of this thesis is as follows:

Chapter 2 provides background information for parallel architectures, especially heterogeneous architectures made of multi-core CPUs and GPUs as well as multi-core CPUs combined with FPGAs on the same system-on-chip. In addition to this, parallel programming models are detailed starting from heterogeneous programming models to task-based programming models since this study focuses on efficiently bridging these models together. A presentation of related work is given on the scheduling techniques proposed in the literature for efficient scheduling of task-based programs on heterogeneous architectures where we also discuss the advantages and disadvantages of the proposed approaches and how these can be improved.

Chapter 3 presents OpenStream, a data-flow extension for OpenMP that enables task parallel programming which we chose for the implementation of the concepts proposed in this thesis. We present the syntax and semantics of the original OpenStream run-time with simple examples and discuss its execution model before any changes are made for the heterogeneous context for the purpose of this thesis.

Chapter 4 describes the practical contributions made in the context of this thesis. GPU and FPGA extensions to the OpenStream run-time are implemented in order to support heterogeneous execution. The extended run-times are then used to implement the proposed scheduling techniques and the evaluation of the contributions. We show how the syntax and execution model changes with these extensions providing example

codes.

Chapter 5 describes our novel dynamic scheduling heuristic for heterogeneous systems made of multi-core CPUs and GPUs. We give detailed information on how data-flow information can be exploited for efficient scheduling of tasks on heterogeneous systems. This chapter ends with the experimental evaluation of our proposed heuristic.

Chapter 6 describes our novel scheduling approach for employing FPGA accelerators on a system-on-chip devices made of low power multi-core CPUs and FPGAs. We take advantage of the data-flow information in OpenStream for creating software pipelined tasks that are executed on FPGA accelerators, followed by the experimental evaluation.

The conclusions on the work presented in this thesis and directions for future research are given in Chapter 7.

1.5 Publications

Some of the material used in this thesis has been published in the following papers:

[100] Simsek, O.S., Drebes, A. and Pop, A., 2018, May. Leveraging Data-Flow Task Parallelism for Locality-Aware Dynamic Scheduling on Heterogeneous Platforms. In 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 540-549). IEEE.

Chapter 2

Background

In this chapter, we introduce the scientific and technical background for this thesis. In Section 2.1, we first explain the basic concepts on parallel architectures by classifying architectural models and memory systems followed by a description of multi-core processors and many-core accelerators with the emphasis on heterogeneous many-core systems. Next, we introduce the concept of parallel programming models in Section 2.2, specifically the ones targeting heterogeneous many-core architectures followed by a general description of task-based programming models. The context of task-based programming models is to deal with the difficulties of programming heterogeneous many-core architectures by abstracting the underlying architectural details using a run-time system. Section 2.3 presents the solutions proposed in the literature for efficient scheduling of task-based programs onto heterogeneous systems followed by a discussion of how our approach differs from existing approaches. Finally, in Section 2.4 we give a summary of this chapter.

2.1 Parallel Architectures

Modern high performance hardware architectures are comprised of multi-core and many-core systems which integrate multiple processing units on a single chip and combine multiple chips with potentially different processing powers into large and highly parallel systems that provide large amounts of processing power. The emergence of highly parallel processors such as GPUs and FPGAs lead to a heterogeneous trend in high performance computing (HPC) systems that tightly couples processing units with different processing capabilities. Heterogeneous systems exploit the large throughput processors as accelerators which is controlled by multi-core CPUs. Such systems can

take advantage of the sequential processing power of the CPUs and incorporate the accelerators for parallel execution to achieve higher performance.

2.1.1 Architecture Models

Flynn taxonomy [42] provides a taxonomy that is generally used for the classification of the parallel systems which classifies the systems according to the number of instruction streams and number of data streams that can be utilized simultaneously. The four classes are: Single instruction single data (SISD), single instruction multiple data (SIMD), multiple instruction single data (MISD) and multiple instruction multiple data (MIMD). SISD corresponds to a classic Von Neumann architecture while MISD is only mentioned in theory, but an actual design of this architecture has never been implemented.

SIMD architectures, or vector architectures are widely used in parallel computing and take advantage of data parallelism, applying the same instruction on multiple data elements. SIMD architectures are often found in mainstream processors such as Intel X86's MMX [87], SSE and AVX [73] instructions, ARM's NEON and SVE extensions [103], AMD's 3DNow! extension [82], Sparc's VIS extension [109], and PowerPC's AltiVec [38]. GPUs also fall into this category, although with the recent advances, GPUs are able to operate on multiple stream of instructions on multiple different elements making them closer to a MIMD architecture.

MIMD systems support multiple simultaneous instruction streams operating on multiple data streams, which generally consist of multiple processing units each including their own control units. MIMD systems are generally asynchronous unlike SIMD systems. Multi-core processors of today are MIMD architectures which include SIMD units in each processing core.

2.1.2 Memory Systems

Parallel systems can be of either shared memory or distributed memory categories and these classifications are based on how processors access memory.

In shared memory systems, all processor cores are connected to the memory system through an interconnection network and each processor core can access each memory location. There are two types of shared memory systems: (1) uniform memory access (UMA) and (2) non-uniform memory access (NUMA). In UMA systems, the access

cost of any memory address is constant for all processor cores whereas in NUMA systems, each processor or a group of processors have its local memory block where local accesses are faster than accessing a memory address that reside in a remote memory unit.

In distributed memory systems, each processor has its own private memory and the communication between processors are managed through a network subsystem by sending and receiving messages. The most widely used distributed memory systems are clusters which are composed of large number of nodes.

Aside from these two classifications, memory systems are evolving in a direction which can be described as hierarchical where the system is composed of multiple nodes in a distributed layout and each node consists of not only one type of processor, but multiple different processors in a heterogeneous context. Accelerator type processors such as GPUs and FPGAs are included in each node where the accelerators are connected to the node through a bus, mostly PCI-e, where every accelerator has its own separate memory. Studies in the literature show that efficient memory management is required in order to reach the performance potential of heterogeneous systems [48, 71, 76, 61].

2.1.3 Multi-core CPUs

CPUs are the backbone of any processing system, designed as latency oriented processors with large control units to decrease the latency of each instruction and focused on maximizing the execution speed of sequential programs. The simplest approach to increase the amount of work performed by a CPU is to increase the number of cores available on the chip. Each core can execute independently, sharing data through the memory sub-system.

The core counts of commercial and server-grade CPUs keep increasing as the technology sizes decrease, creating more possibilities to exploit parallelism and to increase parallel performance of multi-core CPUs [36]. Aside from the sequential optimizations CPUs employ such as pipelining, out-of-order and superscalar execution, branch prediction and speculative execution, today's processors include SIMD instructions to take advantage of data parallelism as well. However, the main bottleneck in these systems remain to be the movement of data between the functional units and the memory [49, 35, 80, 21]. Although large caches are incorporated to tackle this problem, scalability issues still persist.

The latest trend is to use the CPU cores for the sequential sections of a program

while offloading the parallel sections to an accelerator that is designed as a throughput processor which can take advantage of the available parallelism to its fullest [67, 105, 69, 77]. In such case, the CPUs are at the center, executing the sequential sections of the program and orchestrating the accelerators in an efficient way to increase overall system performance.

2.1.4 Heterogeneous Many-core Systems

As the processors are moving towards heterogeneous architectures where the emphasis is on the execution throughput of parallel programs, the processors with higher core numbers that can execute many threads simultaneously has seen a lot of attention. The hardware takes advantage of the massive number of processing units that execute the same instruction in a lock-step fashion, mapping the high amount of parallelism a program can offer to the processing units while using techniques such as latency hiding [112, 66] for memory accesses to achieve teraflops of throughput. An example of such devices is GPUs. However, GPUs are not able to execute any program by itself, thus a CPU is required to orchestrate the execution of a program while offloading parts of the program to the GPU.

GPGPU Architecture

The modern GPUs are designed to support execution of general purpose programs as well as graphics applications. Figure 2.1 shows the distinction between the designs of CPUs and GPUs. GPUs have multiple large cores that in NVidia terminology are called streaming multiprocessors (SM) which includes high number of execution units called streaming processors (SP), control units to handle branch divergence and instruction and data cache. The GPU chip also includes a scratchpad memory named shared memory. While multi-core CPUs have larger control units and larger caches that take a lot of chip area in order to increase the sequential execution performance, GPUs include larger number of small, simple execution units which include smaller control units and caches that can execute large number of instructions simultaneously. The control units in the GPU are mainly responsible for the mapping of threads that are created by the GPU programming model to the execution units. The small, per execution unit caches need not be very large [57, 54], since the memory bus widths of the GPUs are quite large, thus can bring large amounts of data from the memory to the execution units.

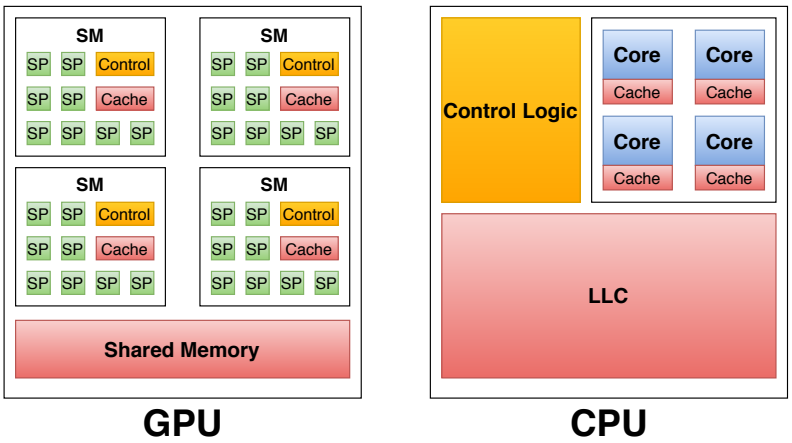


Figure 2.1: Architectural differences between CPU and GPU

The large number of execution units within a GPU are grouped together into blocks and every block must execute the same instruction while different blocks can execute different instructions of the same program [113, 85]. Aside from the large execution units, GPUs have a memory subsystem that is separate from the CPU memory, resulting in a requirement of data transfers whenever the application needs to offload work onto the GPU. The disjoint address spaces makes GPU acceleration non-trivial, but if used efficiently, leads to performance benefits. The details of the GPU memory model is discussed in Section 2.2.3.

FPGA Architecture

Field-programmable gate arrays (FPGAs) are reconfigurable integrated circuits. The configuration of the FPGA is generally specified using a hardware description language (HDL) which is also widely used to describe application-specific integrated circuits (ASIC). FPGAs contain an array of programmable logic blocks as well as configurable interconnects to connect the programmable blocks to create a hardware design [68].FPGAs can be reprogrammed to implement different logic functions, allowing flexible reconfigurable computing as performed in computer software.

In recent years, as the developments on the programmability of the FPGA devices increased [8, 7, 88, 34, 32, 99, 115], the use of FPGAs as accelerators became common. Although there are discrete FPGA boards that can be used in a system through PCI-e, SoC type FPGAs [20] are widely used and are becoming more widespread even in high performance computing systems [62, 90, 97].

Although FPGAs provide potential to greatly accelerate a wide variety of applications, their use was limited due to the amount of effort and expertise it requires to program these devices. Its key feature is the ability to perform computations in hardware to increase performance, while retaining much of the flexibility of a software solution [33]. However, the use of high-level programming languages such as C/C++ and especially OpenCL has made it easier to program an accelerator on FPGA fabric and thus increase the possible use cases. Especially after Intel bought FPGA manufacturer Altera in 2015, the programmability of FPGAs are becoming easier and the use of FPGA accelerators are becoming omnipresent [115, 11, 98].

2.2 Parallel Programming Models for Many-core Architectures

2.2.1 Heterogeneous System Programming

The nature of heterogeneous architectures require different programming models to be employed simultaneously. As mentioned in Section 2.1.4, heterogeneous system architecture generally include CPUs as the main processing unit that orchestrates the execution of applications which include the combination of multiple programming models for the efficient use of system resources. CPU only parallel programming models are omnipresent, but in recent years have been taken over by the heterogeneous models which include multi-core CPUs and accelerators.

A heterogeneous programming model is required to provide the execution capability and the memory management on all the processing units of the heterogeneous system. Firstly, the model is responsible for enabling the execution of parts of the program in every processing unit which is generally achievable through dedicated APIs. Secondly, the programming model is also responsible to provide a data view of the abstracted architecture. Aside from the well-known memory models such as shared or distributed memory models, heterogeneous systems provide a memory model that is neither shared nor distributed, since the memory space of the accelerator is separate from that of CPUs, but is only accessible by the host CPU. Therefore, the memory management becomes the most important part of the model to avoid creating bottlenecks in memory management and communication.

This section describes the programming models for heterogeneous systems, specifically for GPUs and FPGAs. The details of both models are given from the execution

and memory management perspectives.

2.2.2 GPU Programming Models

The programming model of modern GPUs follow a SIMD model with many processing units in parallel executing the same instruction to multiple data elements. Each unit operates on integer or floating-point data with a general-purpose instruction set, and can read or write data from a shared global memory that has its own address space for each GPU.

In the context of General-Purpose Computing on the GPU (GPGPU), programming for GPUs were not trivial since applications still had to be programmed using graphics APIs. General-purpose programming APIs has been conceived to express applications in a familiar programming language. Examples of such APIs are NVidia's CUDA [81] and Khronos Group's OpenCL [64].

CUDA programming model uses a single instruction multiple thread (SIMT) model which is different than the well-known SIMD model. CUDA creates large number of threads that are mapped to the processing units during execution. Each thread that are mapped to the same work-group execute the same instruction on different data in groups. The size of the group changes between different architectures. In CUDA terminology, these groups of threads that execute in lock-step are called warps and the warp size is generally 32 threads whereas the terminology for OpenCL is wavefront.

OpenCL is an industry standard that is developed for a larger vendor range, not just NVidia and also wider device range, not only GPUs, but also FPGAs, DSPs and CPUs which can execute OpenCL programs.

2.2.3 OpenCL Programming Model

OpenCL is a heterogeneous programming framework that is managed by the Khronos Group [51] which is a non-profit technology consortium. OpenCL is a framework for developing applications that execute across a range of device types made by different vendors. It supports a wide range of levels of parallelism and efficiently maps to homogeneous or heterogeneous, single or multiple device systems consisting of CPUs, GPUs, FPGAs and other types of devices. The OpenCL definition offers both a device-side language and a host management layer for the devices in a system.

The device-side language is designed to efficiently map to a wide range of memory

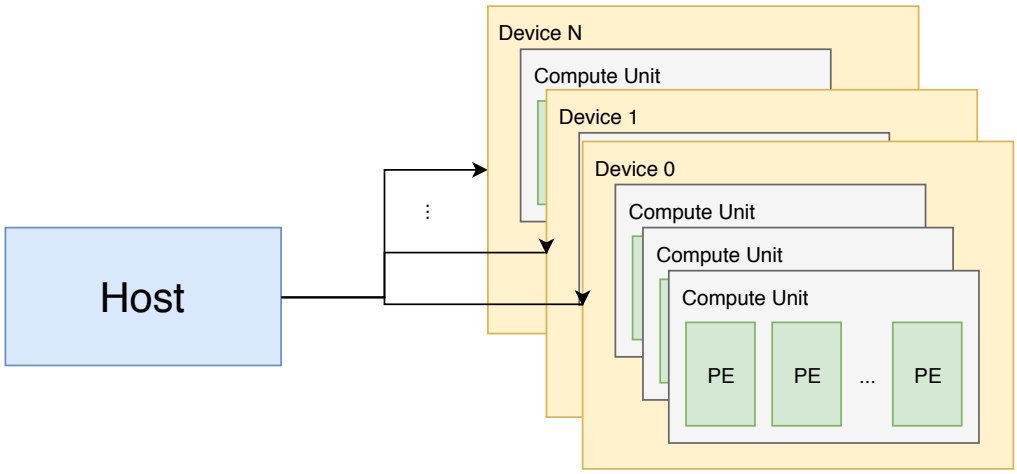


Figure 2.2: OpenCL platform model

systems while the host API aims to support efficient management of complex parallel programs with low overhead. Together, these provide the programmer a path to efficiently move from algorithm design to implementation. This section presents the platform, execution and memory models for the OpenCL programming environment.

Platform Model

An OpenCL platform consists of a host connected to one or more OpenCL devices. The platform model defines the roles of the host and devices and provides a common interface for the OpenCL capable devices. Each device consists of one or more *compute units* that are composed of one or more *processing elements*. Compute units of the devices are functionally independent from each other. Figure 2.2 shows the OpenCL platform model consisting of one host and multiple devices.

The platform model also offers an abstract device architecture which the programmer targets using OpenCL C device language and OpenCL API. The vendors on the other hand, map this abstract architecture to the physical hardware to create an OpenCL compatible device. The OpenCL platform model allows building a topology of a system with a host processor coordinating the execution, and one or more devices that are targeted to execute the OpenCL kernels.

Execution Model

In OpenCL, the host request a kernel to be executed on a device using, a *context* which is configured for a specific device and a *command queue* to pass the execution commands to the device be it a kernel execution or a data transfer. The context must be unique for a device since the kernel binary needs to be compiled for a specific device architecture. Generally, GPU architectures support a variety of binaries for multiple generations of devices from the same vendor. However, a different context must be created for different devices from different vendors.

In OpenCL execution model, devices perform tasks based on commands that the host issue to the device such as kernel execution, data transfer or synchronization. In order to issue any command to a device, at least one *command queue* must be created and used by the host. OpenCL command queues are essentially FIFO structures which do not require any synchronization if the queue is created with in-order property. Command queues also support out-of-order mode, but only some vendors support this option in their driver implementations.

Although not supported by the driver, an out-of-order execution can be achieved by employing multiple command queues for a device, but only for different types of commands such as data transfers and kernel execution. Multiple commands that require the same resource whether it is computational units or PCI-e bus, cannot be used concurrently unless out-of-order execution is supported. However, the use of multiple command queues enable asynchronous execution of commands, creating opportunities to overlap kernel execution and data transfers. The synchronization between command queues can be ensured using OpenCL events. When a command is submitted to the command queue, an OpenCL event can be attached to the command which can be given to other command submissions as an input dependence. Therefore, using multiple command queues and events enables synchronization at the device level which decreases overhead. Moreover, the execution model that uses multiple command queues and events are compatible with data-flow model.

Memory Model

OpenCL classifies memory as either host memory or device memory. Host memory is directly available to the host, and is defined outside OpenCL. Data moves between the host and devices using functions within the OpenCL API or through a shared virtual memory interface. Alternatively, device memory is memory which is available to

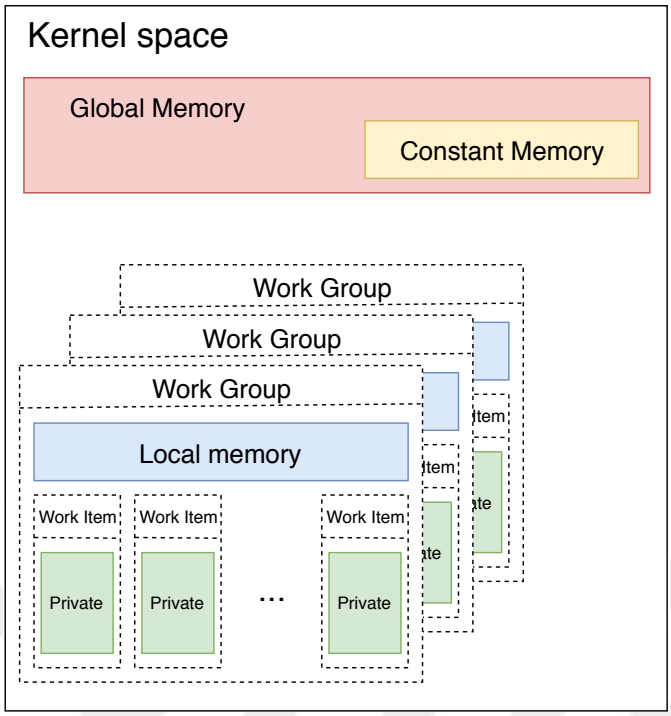


Figure 2.3: OpenCL memory model

executing kernels.

OpenCL divides device memory into four memory regions as shown in Figure 2.3. These memory regions are relevant within OpenCL kernels. Within a kernel, keywords are associated with each region, and are used to specify where a variable should be created. Memory regions are logically disjoint, and data movement between different memory regions is controlled by the programmer. Each memory region has its own performance characteristics. Following these characteristics, accessing data for computation from the right memory region can greatly affect performance.

The private memory corresponds to the registers of processing elements and access to this memory region is the fastest. Private memory can only be accessed from a single work item. Multiple work items within the same work group can access the same local memory which corresponds to the scratch-pad cache, available on each processing element. Global memory is the RAM that is available to all the work groups of a kernel while constant memory is a part of global memory which can be written once, read multiple times. Accessing the constant memory is slightly faster compared to the global memory.

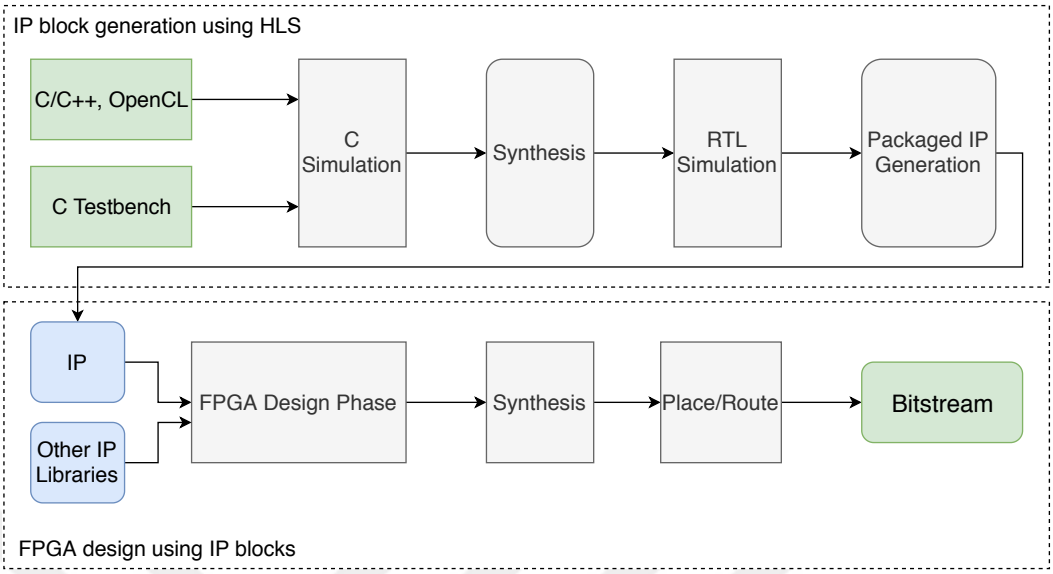


Figure 2.4: HLS design flow

2.2.4 FPGA Programming Models

Devices such as CPUs and GPUs are static architectures which they can only execute specific instructions on hardware using software, whereas FPGAs consists of billions of programmable gates that enable programming the hardware. Describing a hardware design can be done using a hardware description language (HDL) such as Verilog or VHDL in which a programmer can describe how the hardware must behave. However, using HDLs for creating FPGA accelerators are difficult and requires expert knowledge.

Another option is to use High Level Synthesis (HLS) which is a design process that interprets an algorithmic description of a desired behavior, written in a high level programming language such as C, that is used to create hardware that implements that behavior. In recent years, the era of accelerators has started and HLS became mature enough to allow widespread deployment of FPGAs for general purpose acceleration. Especially with the support of OpenCL language, the use of FPGAs became widespread even though using HLS creates a trade-off between the programmability and efficiency since HLS generated designs are generally not as efficient as HDL design cases.

The HLS design flow is illustrated in Figure 2.4 and the design flow includes the following steps:

1. The source code written in a high-level programming language such as C/C++

or OpenCL is provided to the HLS design suite as well as a testbench implementation. The C simulation phase is used to test the implemented accelerator is functionally correct.

2. After the initial functionality tests, both source code and the testbench are used as input to synthesize Register Transfer Level (RTL) design.

3. RTL designs are simulated in the next phase to ensure the functional consistency between the high-level language and the RTL-level generated design.

4. Packaged IP generation step includes wrapping the design in HDL interfaces for general use and the IP block is generated.

5. In the FPGA design phase, all the required IP blocks are added to the design to create the final bitstream.

6. The design is then synthesized. Place/route step manages the placement of each component on the FPGA blocks as well as the routing between the components.

7. The last step is the generation of the bitstream which then is used to program the FPGA device.

Although the developments regarding high level synthesis opened a path for FPGA acceleration, FPGA run-time management has not progressed at the same pace. Designing an efficient accelerator for FPGAs is itself a great challenge. Moreover, controlling FPGA accelerators are generally not trivial since the control program must use accelerator-specific interfaces and functions.

2.2.5 Task-Based Programming Models

Task-parallel programming has become a popular approach in recent years to address the productivity, performance portability and scalability issues in high performance computing systems. Many different approaches have been proposed, ranging from general-purpose [91] and specialized libraries [27, 116] to language extensions [44, 19, 92, 95, 94, 30, 29, 24, 9, 25, 46, 111]. The key concept behind the task-parallel programming models is to create small, fine-grained units of work called tasks, that can be executed in parallel to expose large amounts of parallelism and to specify the interaction between tasks to determine which tasks can run in parallel. The declaration of tasks, the interaction of tasks and the methods of synchronization varies between the

approaches of task-parallel programming. The tasks and the synchronization between tasks do not have to be managed statically, and can be managed dynamically during execution by a run-time system.

Productivity in task-based programming models is addressed by eliminating the technical details in the specification of the program and focusing on the declaration of the tasks and their interactions. This specification includes *what* each task does, rather than *where* or *when* a task is executed, leaving these decisions to the run-time system. Abstraction from such details removes the requirement of providing an application code for a specific architecture or operating system, allowing the programmer to focus on the algorithmic part of the implementation.

Performance portability is addressed similar to the productivity, by abstracting from the platform-specific details, a program implementation can be reused on different platforms, given the run-time system and the programming model provides support. In this case, the run-time system is responsible for the adaptation and execution of the application on a wide variety of target platforms in order to ensure the correct execution of the programs as well as its efficient execution. The run-time systems can achieve the correctness and efficiency by providing a well-defined and properly parameterized platform-independent interface, in addition to supporting multiple platforms, exploiting the features of every platform.

In large many-core systems, parallelism is of paramount importance to provide scalability which can be addressed mainly by encouraging the specification of fine-grained tasks with fine-grained inter-task synchronization. Fine-grained task specification inherently increases the parallelism, enabling the exploitation of large numbers of processing units simultaneously. Not only the fine-grained tasks are required for scalability, but also the efficient use of hardware resources, operating system functions and other system libraries. These responsibilities are also managed by the run-time system by mapping the parallelism to the platform and using all system resources efficiently.

The Run-time System

The run-time system is the central component of a task-parallel programming model and it is responsible for the correct and efficient execution of task-parallel applications. Figure 2.5 shows the relationship of the run-time system with other components of an execution environment. The run-time system consists of task manager, scheduling, synchronization and memory allocation components and acts as a mediator between a

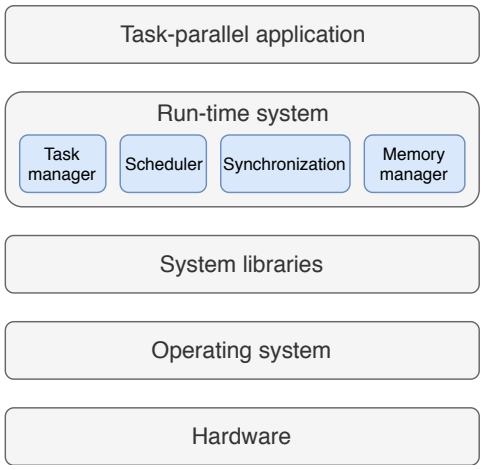


Figure 2.5: Relationship of the run-time with the other components of a system

task-parallel application and system libraries, operating system and even the hardware of the platform. In many cases, the run-time is provided as a library which the application uses the library provided function calls. The application is then linked against the run-time library dynamically and each run-time function satisfies the calls by using appropriate system library functions. System libraries provide an interface to the operating system which enables access to the underlying hardware.

The functionality provided by the run-time system can be grouped into multiple components that depend of the specific programming model and its implementation. For task-parallel programming models, the run-time manages the creation and destruction of tasks, implements task synchronization, contains a scheduler to distribute the ready tasks to the hardware workers. In case the run-time is also responsible for the memory management, a memory allocator is another component of the run-time. The efficiency of a run-time depends on the implementation of these components:

- The algorithms and data structures of the run-time should not become the bottleneck for performance. The overhead of task management and dependence tracking is required to be sufficiently low in order to handle large amount of tasks. Decentralized algorithms need to be preferred for achieving a scalable run-time system implementation.

- The run-time interaction with the lower layers needs to be efficient. For example, slow system calls should be avoided or if mandatory, should not be invoked frequently.

- The execution of tasks need to be arranged in a way, such that the hardware

resources are used efficiently and effectively to increase performance benefits. This requirement is difficult to achieve for all cases, since the efficient use of hardware resources is platform-specific and requires knowledge about the target architecture.

In a nutshell, the run-time system is the layer between the application and the system libraries which implements a specific programming model. Additionally, an efficient run-time system needs to use any system resource efficiently, be it hardware or software resource, in order to provide performance benefits to the programmer.

2.3 Related Work

Efficient scheduling on heterogeneous platforms has seen a lot of attention in recent years. Although the efforts mostly focus on workload division to exploit CPUs and GPUs of the system simultaneously [74, 58, 70, 52, 3, 59, 114, 15], scheduling for task based programming models have been proposed to tackle heterogeneous scheduling problems. Since the novel techniques in this thesis focus on scheduling data-flow task parallel programs on heterogeneous systems, we will not go into detail on workload partition and distribution, but rather give detailed literature review on task based schedulers, specifically the ones that are proposed for run-times based on data-flow models.

2.3.1 XKaapi

XKaapi [47] is a data-flow task parallel run-time system which specializes in multi-CPU and multi-GPU heterogeneous systems. In Xkaapi's programming model, the parallelism is explicit and requires the programmer to describe the parallelism in the code while the synchronization is implicit, meaning the dependences and memory transfers are handled automatically by the run-time. XKaapi tasks are function calls that do not return a value except through the list of function arguments. Each task has a signature that includes its parameters and each parameter's access mode. The available access modes are read, write, reduction or exclusive which is provided by the user to indicate if tasks share data. Tasks share data if they have access to the same memory region which corresponds to data dependences between the tasks. The data dependences are then used for the creation of data-flow graph of the program.

XKaapi uses multi-versioning for the tasks, allowing multiple implementations for the same task, preferably for different devices. Depending on the scheduling mechanism, the version of the task that must execute is determined by the scheduler.

The execution model of XKaapi creates a system thread for each worker which is generally a processor core. Each thread has a private work queue which is represented as a stack. The tasks are created recursively and pushed to the work queue. When a task finishes its execution, the thread pops another task from the work queue using FIFO ordering and executes it.

XKaapi memory manager incorporates the use of different address spaces by keeping track of the host memory and the memories of each GPU on the system through a data structure called Kaapi Memory Data (kmd). Each instance of kmd associates one memory address for each address space which may create replication of data for different address spaces. kmd also keeps meta-data on each address space: a pointer to the data, a bitmap to track which address space has a valid copy and a bitmap to track which address space has a pointer allocated previously. The kmd manages GPU memory through a software cache based on the least recently used (LRU) policy and the consistency is guaranteed by a lazy strategy using a write-back policy. Data transfers to or from GPU occur only when a task accesses data and when the data is in an invalid state in the target address space.

XKaapi run-time employs scheduling by work stealing inspired by Cilk [19]. Work stealing is a scheduling strategy for multi-threaded programs. In a work stealing scheduler, when a worker thread becomes idle, it looks at the queues of other workers to find eligible work. When found, the idle thread *steals* the work item by copying the task and leaving the original task marked as stolen. Work stealing distributes the work over idle processors effectively and no scheduling overhead occurs as long as all processors have work to do. During execution, if a worker finds a stolen task, it switches to the work stealing scheduler mode which computes the data-flow dependences. Otherwise, tasks are executed in FIFO order since the correct execution order is ensured by the data-flow graph.

The classic work stealing is cache unfriendly and does not consider data locality which is especially important in a heterogeneous context since the overhead of data movement between different address spaces is far more expensive than cache misses in multi-core CPUs. In order to tackle this inefficiency they propose two heuristics: the H1 heuristic which is a data-aware strategy, and the H2 heuristic which is a locality aware strategy. Note that both heuristics employ overlapping data transfers between

devices with computation on the GPU to increase the overall execution performance.

The goal of the H1 heuristic is to reduce the memory transfers between host and devices using the meta-data information from the run-time. In this strategy, each ready task to be pushed to a worker's queue, the algorithm goes through the dependences of the task to find out the dependence with the largest data in bytes and checks if the data is valid. The worker that owns the largest dependence in valid state is then chosen as the target to execute the task. The ready task will then be pushed to the target worker's mailbox which is another queue used for work pushing between workers.

Additionally, in the H2 heuristic, the goal is to reduce the invalidations of data replicas which is a similar strategy to locality-guided work stealing presented by Acar et al. [1] and Guo et al. [53]. The H2 heuristic searches a dependence that specifically has a write or exclusive access mode. It pushes a ready task to the mailbox of a worker that has a valid copy of the data by querying the memory manager. In case more than one worker is available, the task is pushed to a randomly selected worker. This selection of the target worker to reduce cache invalidations is performed when the consumer task is activated during work pushing. This heuristic increases preformance especially in multi-GPU platforms since data is replicated for transferring to multiple devices from the centralized memory manager.

Aside from the aforementioned dynamic scheduling heuristics based on locality-aware work stealing since the classic work stealing is cache-unfriendly and does not consider data locality. However, the proposed heuristics do not consider the processing power of different available computational resources. In order to fully exploit the power of different devices, XKaapi run-time offers a profiling based scheduling heuristic called Distributed Affinity Dual Approximation (DADA) [18]. This heuristic uses a cost model for raw performance of CPU and GPU as well as including data transfer costs between devices. The cost model aims to increase overall execution performance even if it means decreasing locality. This comprise of locality comes from the GPU throughput being higher than CPU, thus task execution on GPU may increase overall performance even though the data is on the CPU and requires a transfer.

DADA heuristic consists of two successive phases: a first local phase targets the reduction of the communications using affinity score, calculated for each task. The score is computed using the run-time information using the amount of data updated in software cache by each task. For instance, a task that writes or modifies a data stored on a resource R has a higher affinity score and thus is more likely to be scheduled on the resource R. Therefore, maximizing the affinity score results in increased data locality.

The second phase of the heuristic uses basic dual-approximation [63] for optimizing the make-span of the task graph. The additional α parameter ($0 \leq \alpha \leq 1$) is calculated for each task where a value of 0 denotes the affinity score is not taken into account while 1 denotes affinity score has a higher impact on the overall scheduling.

2.3.2 OmpSs

Omp Superscalar (OmpSs) [26] [89] is an extension designed to incorporate data-flow model into OpenMP using new directives. OmpSs is a continuation of StarSs [92] programming model which exploits task-level parallelism using OpenMP-like pragmas and directives. StarSs programming model evolved into SMPSs [12] for multi-core CPUs and GPUSs [10] for heterogeneous platforms and the combination of SMPSs and GPUSs resulted in the creation of OmpSs run-time.

The programming model of OmpSs is based on OpenMP pragmas and OpenMP task construct with additional directives to handle task dependences. The additional directives are *in*, *out* for input and output dependences respectively and *inout* the dependences that are going to be reused by the run-time. Although false dependencies may occur caused by data reuse, OmpSs run-time is capable of dynamically renaming data objects to eliminate false dependencies. leaving out only true dependencies. This technique is identical to register renaming used in current superscalar processors.

The OmpSs run-time consists of a source-to-source compiler called Mercurium [13] and a run-time library called Nanos [26]. The Mercurium compiler is required to transform the high-level directives into parallelized version of the application while the Nanos run-time is responsible for managing task creation, synchronization, data movement and scheduling. The programming model extension for heterogeneous platforms also require a *target* directive which is used to pass information to the compiler to generate binary for the tasks for the target platform. The target platforms include; multi-core processors (smp directive), GPUs (cuda directive) and FPGAs (fpga directive). The run-time uses the generated binary in order to schedule tasks to the specified devices while managing the data movements between devices as well as data object renaming.

The execution model of OmpSs uses a thread-pool model where three types of threads exist; master thread, helper threads and worker threads. The master thread is responsible for the execution of the user program, intercepting calls to annotated tasks, generating tasks and inserting them in a task dependence graph. Helper threads consumer the created tasks as GPUs on the system become idle mapping the execution

on the most suitable device. Every GPU on the system is assigned one worker thread which waits for available tasks, performs data transfers between devices and also responsible for invoking the low level GPU function calls to manage the execution on the GPU. Since the GPUs are passive processing units, the management of GPUs are handled by the worker thread, on CPUs. Once all the required functions are called, execution on the GPU finished and optionally the results are transferred back to main memory, the worker thread notifies the helper thread, which then can continue assigning new tasks to the worker thread.

OmpSs memory model assumes multiple address spaces exist and the data of a task may reside in a memory location that is not directly accessible from the computational resource. Tasks can safely access the private data and shared data through the use of the directive extensions. The host device memory spaces portray a two-level memory hierarchy. Before executing a task, the worker thread transfers the data to the corresponding GPU and transfers the updated data back to the main memory when the task execution finishes. In order to reduce the redundant data transfers, a software cache of read-only blocks is stored in the memory of each GPU which uses an LRU replacement policy. In addition to the software cache, two memory coherence policies are in order to reduce the amount of data transferred; *write-through* and *write-back*. In write-through policy, when the execution finishes, the worker thread invalidates the read-only copies of the blocks on the remaining GPUs by notifying the corresponding worker threads. In write-back policy on the other hand, data blocks written by a GPU only need to be updated when another GPU needs to use the new data.

The OmpSs run-time library, Nanos++, offers two different dynamic scheduling strategies: dependencies and locality-aware. The former strategy tries to schedule a consumer task on the same device when the producer of that task finishes. The idea behind this strategy is, the producers and the consumers are bound by data dependences and share data and this strategy tries to take advantage of the shared data by reducing the number of data transfers. The latter strategy is based on the work from Martinell et al. [78] in which the scheduler calculates an affinity score for each location when a task is submitted for execution. The affinity score is calculated considering where the data resides as well as the size of the data. The task is then placed to the device which has the highest affinity score. In case the affinity score is the same for multiple devices, the task is placed in a global queue. The threads request work from their local queue first, then into the global queue. If both queues do not have any available task, the worker tries to steal work from other worker's local queues using work stealing

to avoid load imbalance between devices. However, the proposed strategies have the limitation of working strictly on GPUs while the CPUs are only used for managing the run-time routines.

The initial version of OmpSs run-time includes using *target* annotation to determine the device for the execution. However, the target annotation was limited to one type of device which results in the task that uses target annotation is strictly executed on the defined target device. Later versions of OmpSs [93] extend this usage, allowing multiple versions of the same task to be described where the scheduler decides the target device during the execution of the application. This extension includes an extension to the programming model as well as the scheduler. The programming model includes *implements* annotation which enables multiple task implementations while the *versioning scheduler* uses an online scheduler to decide which implementation is going to be used.

The versioning scheduler uses profiling information for each task, recording the average execution time of a task. Each task is run on each processing unit in a round-robin scheme during the initial learning phase of the execution. The scheduler then calculates the *fastest executor* of a task as well as keeping track of the workers to measure when a worker is going to be available by estimating the *OmpSs worker estimated busy time* metric. The scheduler then makes the scheduling decisions based on the average execution time, each worker's busy time and determines the *earliest executor of a task* which is the OmpSs worker that can finish the execution of a task version at the earliest time. The scheduler keeps updating the execution information until the execution of the application finishes.

Compared to dependencies and locality-aware scheduling strategies, the versioning scheduler has little performance benefits due to the overhead of online profiling in addition to the limited performance gains obtained from the CPU cores.

OmpSs-Zynq [41] is an effort to employ FPGAs using OmpSs run-time system by extending its compiler to create and employ FPGA accelerators. The OmpSs code is passed through the source-to-source compiler Mercurium [13] which includes a specialized FPGA compilation phase to process annotated FPGA tasks [22]. For each task, the compiler generates two binaries; one for ARM processors and one is a Vivado High Level Synthesis (HLS) annotated code for the bitstream generation. The annotated code is then supplied to the Xilinx EDK tool to create a complete integrated system consisting of the hardware accelerators as well as the interconnection between the processing system and the accelerators. The run-time employs Xilinx DMA library

to manage the interconnect from the processing system.

The programming model and execution model is identical to the GPU version of the OmpSs run-time, the only addition being the hardware accelerators are used as devices to offload tasks.

In addition to the compiler infrastructure, Nanos++ task scheduling mechanism has been modified for the FPGA device to allow the submission of several tasks to the accelerators, only when the tasks are independent from each other, to exploit double buffering and pipeline features of the accelerators. In this case, the scheduler is required to keep sending new tasks to the FPGA and cannot stall, waiting for accelerator tasks to finish. Moreover, the number of helper threads dedicated to FPGA management can be limited in order to better exploit the CPU resources by avoiding context switches.

Additionally, OmpSs-Zynq is extended as OmpSs@FPGA [23] to provide an ecosystem where the programmer is able to use FPGA accelerators generated directly from the provided functions. This effort is further developed into an FPGA implementation of task manager for OmpSs run-time called picos [106].

2.3.3 StarPU

StarPU [9] is a run-time system that provides an infrastructure for the implementation of efficient scheduling algorithms on heterogeneous platforms. StarPU supports plug-in based scheduling implementations in addition to a run-time API and C language annotations. It is designed to be used as a back-end for parallel language compilation environments and high performance libraries. The two main principles of StarPU are: tasks can have multiple implementations and the most suitable implementation will be chosen during execution, the data which is required by a task may reside on a different processing unit and the transfer of the data is handled transparently by the run-time. The former principle is generally used to make efficient scheduling decisions while the latter principle is important to reduce the execution overhead of heterogeneous platforms.

The StarPU programming model relies on the use of a *codelet* which is essentially a task description. The codelet includes meta-data information about the task such as pointers to the task implementations for different devices, input and output dependences, arguments of the tasks and data access modes for the dependences. StarPU tasks can be executed by as many processing units as possible as long as an implementation for a target device is provided by the programmer. All the input and output

dependences are also need to be explicitly included in the codelet description, so the run-time system can automatically handle data transfers before the execution happens on an accelerator. Therefore, programmers are neither concerned by where the tasks are executed, nor how valid data replicas are available to these tasks. They simply need to register data and provide multiple implementations for tasks for the various processing units.

In the execution model of StarPU, once all the input dependences of a task are satisfied, a task becomes ready and is submitted to the scheduler. StarPU uses a centralized scheduler where all the ready tasks are submitted and each task is consumed by a processing unit of the system depending on the scheduling algorithms decision. Since the run-time is designed to provide an infrastructure for different scheduling methods, the execution model is a generic and simple and different execution models and load balancing mechanisms can be incorporated by implementing a new scheduler.

The StarPU memory model is based on a decentralized asynchronous data management policy. When a task is assigned to a processing unit, the corresponding data is replicated to the processing unit. Data replicas are updated using a lazy replacement policy when it is strictly required to avoid redundant data transfers. StarPU employs a coherence protocol similar to MESI cache coherency protocol to keep all data up-to-date on different processing units. They use the access mode parameter of the codelet to determine which processing unit has the most up-to-date version. The memory manager also uses asynchronous data transfers to keep the overhead of data movement minimal.

Although StarPU provides an infrastructure, enabling the implementation of different scheduling strategies, a set of predefined scheduling policies are also included in the run-time implementation such as a greedy policy with (greedy) and without priority support (no-ws), a greedy policy based on work stealing (ws), a policy based on random weights of processor speeds (w-rand) and a policy based on HEFT (heft-tm) [108]. The priorities can be defined in greedy policy using the StarPU programming model by passing hints to the scheduler. In all greedy policies, whenever a task becomes ready, it is pushed to an available processing unit. In case of the ws policy, the scheduler uses work stealing for load balancing when a processing unit becomes idle. In w-rand policy, each processing unit has a predetermined *acceleration factor* which corresponds the performance capability of each processing unit. This ratio can be provided by the programmer, or can be measured by prior profiling of applications. The acceleration factor is then used as a probability metric by the scheduler which means

a processing unit with the highest performance has a higher chance of receiving a task for execution. Although these policies are easy to implement, the performance benefits are limited. Therefore, they suggest an efficient scheduling policy not only has to consider the heterogeneity and performance differences of the platform, but also requires load balancing [9]. In accordance with this finding, they propose heft-tm scheduling policy in which each time a task executes the run-time measures the amount of time spent on the execution on each processing unit to create a performance model. The performance model is then used to make scheduling decisions to achieve the shortest amount of execution time. The model uses measurements for tasks executions, but for data transfers, an off-line sampling of bandwidth between devices is employed. However, the scheduler is able to hide the data transfers by overlapping them with kernel execution on the GPUs using the off-line calculation.

2.3.4 QUARK

There are also run-times that focus on specific subjects. PLASMA [27] is a dense linear algebra library, designed to deliver high performance that targets multi-core processors with multiple sockets. PLASMA relies on the scheduling of parallel tasks, creating a task graph in a data-flow fashion and uses dynamic scheduling. Additionally, MAGMA [107] is designed to work on multi-core CPUs and GPUs in a heterogeneous context. These components are combined to create QUARK [116] run-time which is designed to enable dynamic task execution with data dependences and targets heterogeneous environments.

The programming model of QUARK uses a centralized queue which is used for inserting tasks into the run-time using a serial task-insertion API. The arguments for the tasks are then used to make a DAG using the reads and writes on the data that are queued. The DAG is created using dependence information using data access annotations such as input, output and inout.

The execution model of QUARK consists of a master thread and multiple worker threads. The master thread is responsible for determining the dependences between tasks and inserting tasks to the ready queue. Worker threads then take tasks from the ready queue, execute tasks and handle the descendant tasks.

Scheduling in QUARK is achieved using a data-aware scheduling technique. When all dependences of a task are satisfied, the task is scheduled to the worker private ready queue. The default scheduler assigns the task to the worker thread that has most recently written its output data, attempting to reuse the data in the cache for the same

thread. Additionally, the user can provide hints to the scheduler using task flags or argument flags in order to increase the efficiency of the scheduler, replacing the default scheduling policy. The scheduler uses work stealing in order to balance the workload between workers.

2.3.5 Discussion

We have presented the past research on the scheduling methods proposed for the run-time systems that are based on data-flow task parallel model targeting heterogeneous platforms. Although the programming models and the execution models of these run-times are similar, there are multiple approaches for efficient scheduling of data-flow tasks. The programming models of these run-times require explicit information on the data dependences between tasks to create a DAG of tasks. The consensus is that the task descriptions take input, output and inout dependences and use this information for making scheduling decisions.

The execution models can be divided into two: while StarPU, OmpSs and QUARK use centralized schedulers with a master thread and multiple worker threads, XKaapi uses decentralized workers to avoid possible bottlenecks. Although using a centralized scheduler has the benefit of keeping the load balanced, as the number of workers increase as well as the computational units, scalability becomes a limiting factor. Therefore, using a decentralized scheduler with improved load balancing mechanisms can avoid the scalability problem both in homogeneous and heterogeneous architectures.

In the perspective of memory models, all aforementioned run-times employ the use of a software cache to keep track of where the data of each task reside. The disadvantage of the software cache is the possible false-sharing of data between devices and the expensive cost of resolving invalid data regions. On the other hand, instead of using a software cache for memory management, employing task-private buffers had been shown to create more opportunities for more efficient task and data placement [95, 39]. Especially in heterogeneous systems where there are multiple address spaces, using task-private buffers is advantageous for making better scheduling decisions as well as reducing memory management overhead.

In heterogeneous systems, the movement of data between host and device address spaces is an expensive operation. Although GPUs can deliver immense amount of throughput compared to CPUs, the movement of data impact the performance, diminishing the advantage of GPU usage if operations on the GPU require large amount of data movements. The most common method to overcome this problem is to overlap

data transfers between devices with kernel executions on the GPU to hide the cost of transfers. All the aforementioned run-times employ such methods to reduce the overhead of data movement and overlapping data transfers with task execution is mandatory for any efficient run-time system.

Work stealing is a technique, widely employed by task parallel programming models in order to reduce the idle time of the processing units as well as load balancing. Although work stealing is an effective method for load balancing, it is known as cache-unfriendly due to random stealing of tasks. Therefore, the studies in the literature use work stealing combined with locality-aware schemes to overcome the reduced locality of scheduling. However, work stealing can only passively react to the distribution of tasks and memory transfers whereas a heuristic that can pro-actively make decisions on task distribution increases the efficiency of the scheduler [100].

In related work, we also discussed different dynamic scheduling strategies in data-flow task parallel run-times. The proposed strategies mainly focus on using the dependence or locality information stored within the run-time to make dynamic scheduling decisions. In addition to dynamic strategies, on-line scheduling methods based on cost models are proposed as well. The drawback on the cost model strategies is that the scheduling depends on regular computations and does not adapt to execution variations.

2.3.6 The Effect of Task Granularity

As mentioned in Section 2.1.4 the computational capability of devices on heterogeneous systems are different due to architectural differences. In order to take full advantage of the systems resources, the devices with higher throughput such as GPUs, which can execute immense number of threads in parallel, require larger tasks compared to CPU cores. Although there are efforts to partition the tasks and data to the corresponding devices in a way to increase the effective utilization [43, 86, 114, 59], these efforts are generally not applied to task-parallel run-time systems. The reason for this is, the run-times that are presented in Section 2.3 all require programmer effort to define the task sizes which do not change throughout the execution of the program. Although in theory, it is possible to implement the applications using the aforementioned run-times and partitioning the tasks in a way that coarse-grained tasks are executed on the GPU and fine-grained tasks are executed on the CPU, such implementations are not practical.

To overcome the granularity problem, the run-times can either support fine-grained

tasks and compensate by assigning more tasks to the higher throughput devices, or employ recursive decomposition of tasks such as in Cilk [44, 19]. Cilk is a multi-threaded run-time system that extends the C language with simple keywords to create and synchronize tasks. The programmer is responsible to expose parallelism and exploit locality, and the run-time system is in charge of scheduling tasks on the target platform. Cilk is a recursive fork/join language where each spawned task can create more tasks. Cilk language is not compatible with data-flow model with the exception by Vandierendonck et al. in [111] where a unified scheduler is proposed combining recursive and task data-flow parallelism. However, the assumption of a task spawning more tasks is incompatible with today's heterogeneous architectures where a GPU or FPGA is not able to create child tasks which can be executed on any device. This shortcoming makes Cilk-like task creation incompatible with heterogeneous platforms.

Consequently, in task-parallel run-times targeting heterogeneous systems, choosing the correct granularity for tasks is problematic. The state-of-the-art task-parallel run-times solve this problem by assigning more tasks to the higher throughput devices and trying to minimize the overhead of large data transfers by overlapping transfers with execution. Additionally, GPU throughput highly exceeding CPU throughput, this kind of mechanism provides high execution performance.

2.4 Summary

The ongoing shift in HPC from homogeneous to heterogeneous architectures, integrating multi-core CPUs and accelerators such as GPUs and FPGAs require multiple programming models to be used in conjunction. Programming models for heterogeneous systems give developers control over memory allocation and execution, but burden them with technical decisions that require expert knowledge of the targeted system in order to use resources efficiently. Ideally, programmers are only responsible for expressing parallelism, which is then mapped efficiently to computing and memory resources automatically considering the architectural differences between the host and accelerator devices.

Task-parallel programming models allow programmers to specify fine-grained parallelism as a set of dynamically created dependent tasks whose execution is managed by a runtime system. This enables exploitation of a wide variety of parallelism types, such as loop, task and pipeline parallelism. Parallelism is expressed in an abstract and portable way, as both tasks and dependences abstract from the actual computing and

memory architecture of heterogeneous systems.

Combining task-parallel programming models with the heterogeneous programming models reveal difficulties as integrating both is not trivial, but important for efficient execution of programs on heterogeneous systems. Related work in Section 2.3 presents efforts in the literature to combine data-flow task-parallel run-times targeting heterogeneous platforms which focus on efficient scheduling of data-flow tasks onto multiple devices with different capabilities.

As discussed in Section 2.3.5, heterogeneous run-time systems in the literature use centralized schedulers aside from XKaapi as well as keeping track of the memory regions in the application by using a software cache which is managed implicitly by the run-time. XKaapi [47] demonstrates the advantages of a decentralized execution model as well as proposing dynamic scheduling strategies. However, the management of a software cache is cumbersome since the run-time needs to manage the coherency of the memory regions on multiple address spaces. The use of task-private memory regions such as the one proposed in OpenStream run-time[94] creates further opportunities for efficient dependence management and can be leveraged for efficient scheduling methods.

The next chapter provides an introduction to OpenStream presenting the basic concepts of OpenStream, the syntax and semantics of streams, its execution model and the compilation of OpenStream programs.

Chapter 3

OpenStream

This chapter provides an overview of OpenStream [94], which is a task parallel language and a data-flow extension to OpenMP 3.0 [83]. This chapter covers the details of original version of OpenStream which is chosen as the data-flow task-parallel run-time system to implement the contributions presented in this thesis between Chapters 4-6. The original version of OpenStream only targets homogeneous multi-core and multi-socket systems which this thesis extends the original OpenStream to target heterogeneous platforms.

In this chapter, we first explain the terminology of OpenStream in Section 3.1 such as streams, data-flow tasks, task dependences and the synchronization of tasks based on streams before moving on to syntax and semantics, in which we discuss how the aforementioned structures and their semantics come together with the syntax of OpenStream in Section 3.2. The execution model of OpenStream is presented in Section 3.3, giving detailed information about scheduling mechanisms, memory management and dependence management followed by the details on compilation of an OpenStream program in Section 3.4.

3.1 Terminology

OpenStream [94] is a data-flow extension to OpenMP which supports fine-grained task parallelism, data parallelism and pipeline parallelism concepts in the C programming language. The implementation of OpenStream is based on OpenMP 3.0 [83] in which the OpenMP task construct has been introduced.

Control Program The control program is the part the programmer describes the

tasks and the dependences between tasks. The control program can either be sequential or parallel. In the sequential case, the root task of the OpenStream program, which corresponds to the main function, is responsible for creating future tasks. The sequential control program guarantees the deterministic behavior. Additionally, under certain conditions the control program can be parallelized without losing the determinism of an OpenStream application. In parallel control programs, the task creation can be delegated to other threads. However, in a heterogeneous context where not all the devices present on the system can create tasks, implementing a parallel control program in itself is a challenge and falls outside of the scope of this thesis.

Streams Streams are unbounded FIFO queues with theoretically infinite capacity which holds elements of the same type. Each element of a stream has a unique index and can be written once, but can be read many times. Elements that have not been written are undefined and cannot be accessed for reading. Each stream has separate read and write positions that are updated for each read and write access. A write operation to a stream directly updates the write position index whereas a read operation may or may not update the read position depending on the access type. The read accesses that do not advance the read position can be used to broadcast data where the data is written to a stream once and read multiple times, advancing the read position once all readers finish accessing the data.

Views Streams are not accessible directly, but can only be accessed through views. A view is a sliding window that allows the task to access a set of consecutive elements from a single stream or from several streams. A view has three attributes: the *access type* (read or write), the size of the sliding window called *horizon* and the *burst* which corresponds to the number of elements the read or write position of the stream is advanced. The access type can be either a read or a write. The horizon attribute must be a positive integer number since it corresponds to the size of the data that is actually accessed in a stream. On the other hand, the burst size can be zero, allowing a set of stream elements to be read multiple times. However, if the access type is write, the burst size must match the horizon whereas if the access type is read, the burst size can either be equal to horizon or zero. Another constraint is that the burst size cannot be bigger than the corresponding horizon size which would result in an access outside the allowed memory region of a task. Further details about the constraints on horizon and burst sizes are explained in Section 3.3.4.

Data-flow Tasks Tasks in OpenStream are dynamic instances with short lifespan, defined by a work-function and a set of views. Work-function corresponds to the body

of the task which is executed when the task is scheduled for execution. A task can only be scheduled for execution when all of its dependences are satisfied. The input dependences of a task are satisfied when all of their producer tasks finish their execution and the output dependences are satisfied when the task allocates all of its output buffers.

3.2 Syntax and Semantics

OpenStream is built as an extension to OpenMP, OpenStream language also uses pragmas for the declaration of OpenStream specific constructs. The OpenStream compiler translates these pragmas into data structures and code to be used by the OpenStream run-time. All pragmas, similar to OpenMP, start with *#pragma omp*, followed by more specific constructs and optionally, a set of clauses that can be passed as certain parameters to the corresponding constructs. The constructs that are supported by OpenStream are:

- *task construct*: Can be used to create tasks. Accepts additional clause specification for accessing streams.
- *taskwait construct*: Can be used to define a barrier which blocks the execution until all the tasks finish execution and reach the barrier.
- *tick construct*: Can be used to advance the read position of a stream. In order to use this construct, the task needs to have a view with zero burst value, so that the specific view's read position can be advanced by the declared value using *tick*.

The aforementioned constructs can be declared in the control-flow of the OpenStream application. Streams can be created not by using pragmas, but by using the special attribute *stream* that needs to be added to the definition of any variable.

3.2.1 Declaring Streams

The syntax of declaring a stream is straightforward since the streams are managed by the run-time system. The programmer only needs to specify the type of the stream elements and an identifier, adding the special attribute *stream* at the end of the definition. This allows the compiler to distinguish streams from regular variables as shown below:

```
1  element_type stream_identifier __attribute__((stream));
```

Streams in OpenStream are first class objects, therefore references to streams are also supported using *stream_ref* attribute as follows which allows creating arrays of streams:

```
1  element_type stream_ref_identifier __attribute__((stream_ref));
```

Similar to other data types, it is possible to create arrays of streams or stream references using a size expression as follows:

```
1  element_type stream_identifier[size_expr] __attribute__((stream));
2  element_type stream_ref_identifier[size_expr] __attribute__((stream_ref));
```

The following example shows different types of stream declarations, as scalar variables for float element type, arrays of streams for integer element type with size 10, a stream reference with integer type, as well as an assignment of a stream reference:

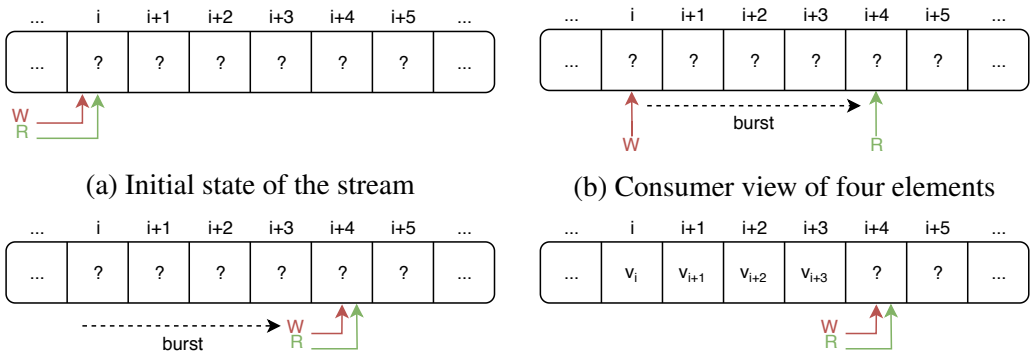
```
1  // A scalar stream of floating point elements
2  float float_stream __attribute__((stream));
3
4  // Array of 10 streams of integer elements
5  int int_array_stream[10] __attribute__((stream));
6
7  // A stream reference of integer elements
8  int int_stream_ref __attribute__((stream_ref));
9
10 // Assignment of a stream reference
11 int_stream_ref = int_array_stream[0];
```

3.2.2 Declaring Views

The syntax of declaring views consists of two parts. The first part is similar to declaring a scalar or an array variable that declares its type and horizon of the view. The second part is a reference of the declaration as a clause inside a task construct. The clause specifies the stream that is to be accessed as well as the access type, either a read or a write. The view declaration is syntactically the same as a declaration of an array in the C language where the size of the array corresponds to the horizon of the view.

```
1  element_type view_identifier[size_expr];
```

The size expression can be specified as static or dynamic as in the C language. The example below declares two views with static and dynamic sizes. The first declaration



(a) Initial state of the stream

(b) Consumer view of four elements

(c) Producer view of four elements

(d) State after the execution of the producer

Figure 3.1: Illustration of stream accesses with burst and horizon

is a view on a stream of floating point elements with statically sized horizon of 5 elements. The second declaration is a dynamically sized view on a stream of type integer:

```
1  float static_view[5];
2  int dynamic_view[5*x+10];
```

Additionally, similar to multi-dimensional arrays in the C language, views can be declared using multiple dimensions, enabling access to multiple streams at once:

```
1  double view[num_streams][horizon];
```

In case the expression which corresponds to the number of streams is not constant, the view is called a *variadic view*.

Since the streams are not directly accessible, Figure 3.1 illustrates how stream accesses are handled using views of two tasks; one read view for a consumer task, and one write view for a producer task. Figure 3.1a shows an initial state of a stream prior to any access. The read and write positions are specified with R and W respectively and point to the same starting index *i*. When the consumer task's view accesses the stream with horizon and burst of four elements, it enables the access to four elements in read mode, advancing the read position to *i+4* as shown in Figure 3.1b. The subsequent read access to the same stream is set to start from index *i+4*. At this point, the consumer task has access to the elements, but the task cannot yet execute since the required data is still undefined. Figure 3.1c shows the write access to the stream from the producer task's view with horizon and burst of four elements, identical to the read access. The write position is advanced to point to index *i+4*, enabling access to

the elements *i, i+1, i+2, i+3*. When the writing view's elements to be accessed are determined, the producer task becomes ready to execute. After the producer task is scheduled to execute, execution happens and the output of the task is written to the stream elements *i, i+1, i+2, i+3*. Figure 3.1d shows that the elements now have values of v_i, v_{i+1}, v_{i+2}, v_{i+3}. When the producer task terminates, the consumer task becomes ready, since its only input dependence is satisfied.

The read and write accesses to streams only start from the read and write positions and are determined with horizon and burst values of a view. Therefore, arbitrary access to stream elements is not possible in OpenStream. Although this example shows two dependent tasks with one dependence, it is possible to create different dependence patterns with variable horizon and burst sizes.

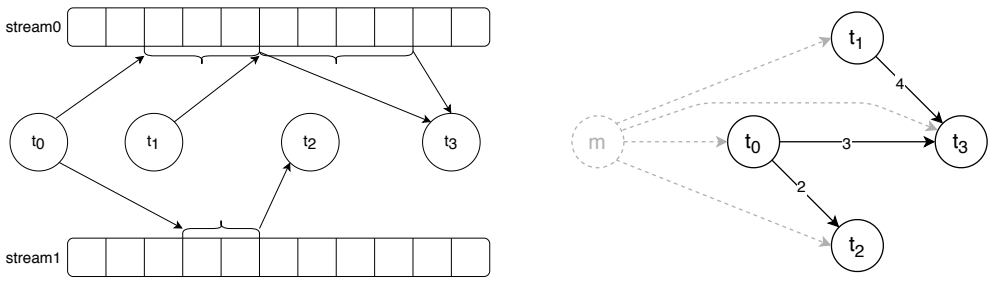
3.2.3 Task Creation

The task creation in OpenStream uses a modified version of the task construct in OpenMP. The modifications allow OpenStream to use additional clauses in order to match views and stream elements, enabling dynamic task creation. The clauses that are used to specify views are *input*, *output* and *peek*. Input and output clauses provide read and write access to stream elements. The peek clause is a special version of the input clause where the burst value of the view is zero. In this case, the view can access the stream elements to read, but does not advance the read position of the stream. Therefore, multiple views can access the same stream elements using peek clause.

If the task construct is used without any clause, the task does not have access to any stream. Therefore, the created task does not have any producer or consumer tasks, making it an independent task which executes when the control program reaches the task implementation during execution. Moreover, independent tasks do not belong to the data-flow semantics. The syntax of the task construct in OpenStream is as follows:

```
1  #pragma omp task input(stream_expr >> view_expr, ...)   |
2                   output(stream_expr << view_expr, ...)  |
3                   peek(stream_expr >> view_expr)         |
4                   sharing_clauses
5  {
6      task_body
7  }
```

The << and >> operators are used to provide access for the views to stream elements. The direction of these operators are in conjunction with the access type. Additionally, sharing clauses in the task construct allow the programmer to define how



(a) Tasks and views

(b) Resulting dynamic task graph

Figure 3.2: Tasks accessing streams using views and the corresponding dynamic task graph

scalar variables declared outside the task are accessed inside the task body which is identical to the sharing clauses in the OpenMP standard [83].

The stream and view expressions in the clauses define single or multiple stream usage as well as the burst size of the views. A stream expression can either be: (1) the name of the stream or stream reference where the view expression provides access to a set of consecutive elements, (2) an array expression that consists of the name of the stream and the index expression in brackets, (3) the name of an array of streams, providing multi-dimensional access to the elements of a variable number of streams.

The view expression, on the other hand is either: (1) the name of the view which provides access to only one element, (2) a view in an array form where the size of the array corresponds to the burst size, (3) a multi-dimensional or variadic view which references multiple streams with an explicit burst for all streams.

The task body consists of one or multiple statements and during compilation the body is transformed into a *work function*. The run-time uses the outlined work function to execute the task body when the task is scheduled to a processing unit. The task body has access to the stream elements through its views as well as other shared variables if the variables are included in OpenMP sharing clauses.

Task Creation Example

The following example code shows four tasks with varying number of input and output dependences and Figure 3.2 illustrates the tasks accessing the streams using views and the resulting task graph.

```
1 #pragma omp task output(stream0 << v0[3]) \
2                  output(stream1 << v1[2])
```

```
3 {
4   task_body_0
5 }
6
7 #pragma omp task output(stream0 << v2[4])
8 {
9   task_body_1
10 }
11
12 #pragma omp task input(stream1 >> v3[2])
13 {
14   task_body_2
15 }
16
17 #pragma omp task input(stream0 >> v4[3]) \
18                  input(stream0 >> v5[4])
19 {
20   task_body_3
21 }
```

The first task t_0 has two output clauses with horizon sizes three and two, accessing two separate streams stream0 and stream1 respectively. The task t_1 has only one output clause in the code which uses stream0 with the view horizon of four. These two tasks are the producer tasks that produces the data which is going to be consumed by the tasks t_2 and t_3. The tasks t_2 and t_3 have input clauses in their task description which makes them consumers, where t_2 has only one input clause for stream1 with horizon of two and t_3 has two input clauses for stream0 with horizons of three and four. Figure 3.2a illustrates all the tasks that are described in the code, their accesses to streams using views with the defined horizons and how the producer-consumer relationship is established. Figure 3.2b shows the resulting task graph where each node represents a task and each edge represents a dependence, including the size of the dependence in number of stream elements. The node named *m* in the figure represents the main thread that is responsible for creating the tasks and the dashed lines are used to describe that the main thread is creating each task in the main program of the OpenStream application.

3.2.4 Tick Construct

The tick construct is used to advance the read position of a stream when the stream elements are required to be read by multiple tasks in a broadcast. In a broadcast, the producer task which is to broadcast the data uses output clauses in a regular manner. However, the consumers cannot use regular input clauses since using the input clause advances the read position of the stream, resulting in only one read access. For multiple

read accesses to the same stream elements, the peek clause must be used so that the read position is not advanced. Once all the tasks read the required data, the read position needs to advance to allow subsequent read operations to the stream. Therefore, in order to advance the read position in case of a broadcast, the tick construct is used. The syntax of the tick construct is as follows:

```
1  #pragma omp tick (stream_expr >> size_expr)
```

The stream expression must either be a single stream reference, or an array expression addressing a single stream. The size expression defines how much the read position is advanced which is required to match the producer's burst size.

3.2.5 Taskwait Construct

The taskwait construct provides local barrier synchronization for OpenStream programs. Every task that encounter this barrier is suspended until all the tasks in the context reaches the barrier. The syntax of the barrier is as follows:

```
1  #pragma omp taskwait
```

Employing such barriers during task execution is a disadvantage for data-flow task parallel programs which causes over-synchronization. Since data-flow model supports point-to-point synchronization, taskwait construct is generally used at the end of the control program, in order to ensure all tasks terminate and the resources freed.

3.3 Execution Model

3.3.1 The Workers and The Scheduler

One of the main components of the run-time is the scheduler. OpenStream is intended to run on massively parallel systems, hence the scheduling structures are distributed to avoid creating any bottlenecks. The scheduler uses lock-free implementations for the most important data structures to avoid synchronization overheads. In OpenStream, each execution unit has a persistent worker thread based on POSIX threads, running a scheduling loop which executes ready tasks on the dedicated core. All worker threads are created at the beginning of the application and terminated when the execution of the application finishes. By default, one persistent worker thread is dedicated for each

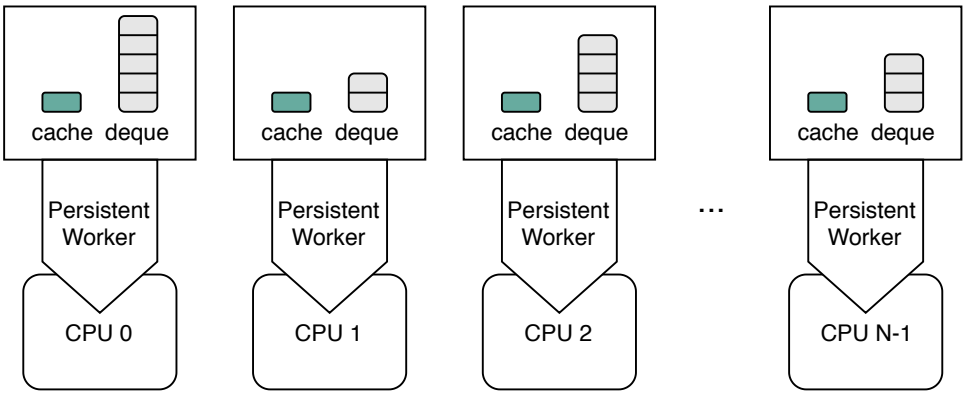


Figure 3.3: Persistent workers with their data structures and worker placement in OpenStream

CPU core as shown in Figure 3.3. Although the workers can be placed in any order on the cores of the CPU, the mapping of workers to cores are set at the beginning and remains unchanged until the execution finishes.

Figure 3.3 also shows two main data structures that each persistent worker include; *work stealing queue* and *work cache*. The work stealing queue is a double ended queue that contains any number of ready tasks. The work cache on the other hand, can only contain a single ready task. When a worker activates a task, it tries to add the task to the work cache. In case the cache is empty, this operation is successful. If not, the task in the cache is moved to the work stealing queue, followed by the activated task being added to the work cache. Therefore, the work cache always contains the most recently activated task.

When a worker finishes the execution of a task, it first checks the work cache, removes the task if there is one, and executes it. In case the work cache is empty, the worker then pops a task from the bottom of the *work stealing queue*. If both the work cache and the work queue are empty, the worker randomly chooses a victim worker and tries to *steal* a task from the top of victim's work queue. The work cache is private to each worker and is inaccessible to other workers, thus work stealing is only allowed on work queues.

The advantages of the work cache is two-fold. First, since the work caches are worker private data structures, adding or removing tasks does not produce any synchronization overhead. Secondly, the task in the work cache, which is the last activated task by the worker, cannot be stolen from other workers which not only increases locality, but also avoids redundant work stealing.

The implementation of the work queue is based on the dynamic, circular, lock-free

prod_queue	ptr
cons_queue	ptr
elem_size	size_t
refcount	int

(a) Stream

horizon	size_t
burst	size_t
next	ptr
owner	ptr
reached_pos	size_t
data	ptr
cons_view	ptr

(b) View

synchronization_counter	size_t
input_view_chain	view_ptr
output_view_chain	view_ptr

(c) Frame

Figure 3.4: Data structures of OpenStream run-time

deque proposed by Chase and Lev [31]. Tasks are added to the work queue from the bottom end and can only be stolen by other workers from the top end. A task can only be removed from the bottom by the owner of the work queue to execute that task. In each worker, tasks are executed in LIFO order which results in the most recently activated task to be executed, favoring local execution of the tasks and increasing cache locality. On the other hand, work stealing takes place in FIFO order, which indicates the data of the stolen task is less likely to be found in the cache.

3.3.2 Data Structures

The OpenStream run-time has three main data structures that correspond to streams, views and tasks as shown in Figure 3.4. The stream data structure consists of the following as shown in Figure 3.4a:

- producer_queue: a list of unmatched output views on the stream
- consumer_queue: a list of unmatched or partially matched views of the stream
- elem_size: the size of each element in bytes
- refcount: a reference counter for garbage collection

When a stream is created, its producer and consumer queues are empty, the element size is set to the size of elements in the stream declaration, and the reference count value is set to one. As stream elements are allocated, the producer and consumer queues become accessible using views, and the refcount is incremented by one for each stream reference is created.

The view data structure is illustrated in Figure 3.4b and includes the following fields:

- horizon: the horizon size of the view in bytes
- burst: the burst size of the view in bytes
- next: a pointer to the next view to create a chained linked list
- owner: a pointer to the owner task's frame data structure, always the consumer task
- reached_position: a field used for indexing the data buffer to check if the view is matched, unmatched or partially matched
- data: a pointer to the elements of the sliding window of the stream
- consumer_view: if the view is an output view, this field points to the matched input view of the consumer task. If the view is an input view, this pointer points to itself. This field becomes useful when the task graph is dynamically traversed

When a view is created, the data location is not yet known, thus is set to NULL. The reached position is set to zero which indicates the view is not matched to any producer or consumer. Horizon and the burst fields are initialized as the horizon and burst of the view. In case the view is created using a *peek* clause, the burst field is set to zero. The consumer view field is also set to NULL at the initialization phase and is set when dependence resolution happens. All the views are created without any indicator of a read or write access, since this information is kept by the compiler and passed to the run-time when views are matched with stream elements dynamically.

The last data structure is called *data-flow frame* or *frame* for short which corresponds to a data-flow task as illustrated in Figure 3.4c:

- synchronization_counter: a synchronization counter *sc* for short which indicates if a task is ready for execution
- input_view_chain: a pointer to the first view in the linked list that holds all input views of this frame
- output_view_chain: a pointer to the first view in the linked list that holds all output views of this frame

The synchronization counter of a task is initialized as the sum of the horizons of its input views and the number of its output views. Each output view contributes as 1 to the

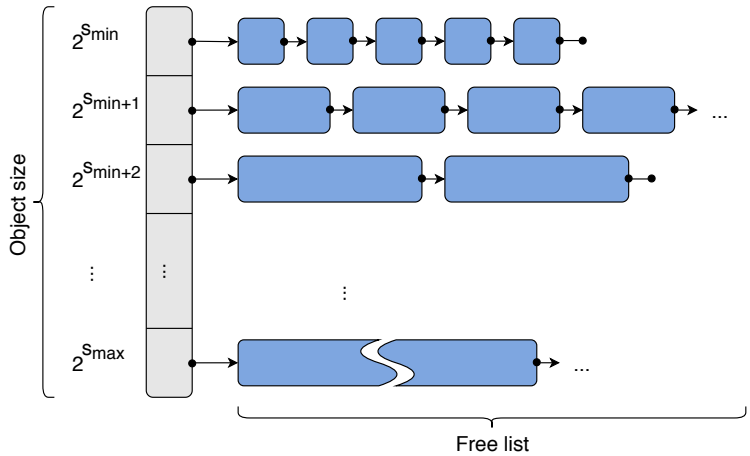


Figure 3.5: Structure of the memory pool

synchronization counter which indicates the consumer of the output view has not yet been created. When the output view is matched with a consumer task, synchronization counter is decreased by 1.

3.3.3 Memory Management

As discussed in 3.3.2, OpenStream has multiple data structures within the run-time that are allocated and freed dynamically throughout the execution. In general, these data structures are allocated when a task is created and freed when a task terminates. Since the tasks in OpenStream are fine-grained and short-lived, memory allocation and deallocation calls are frequent. In addition to this, due to the parallelism within the run-time, a centralized memory manager which handles high amounts of memory operations with frequent invocations can easily become a bottleneck. To overcome this issue, OpenStream uses a decentralized memory management approach based on per-worker memory pools.

The basic concept of a memory pool is to allocate a large portion of memory for the application, so that instead of a memory allocation request to the operating system, the memory manager can return a chunk of previously allocated memory instead. The memory pool assumes each object used by the run-time is between $2^{s_{min}}$ and $2^{s_{max}}$, where s denotes the minimum and maximum sizes respectively. For each, size 2^i where $s_{min} \leq i \leq s_{max}$, a linked list of free blocks of size 2^i bytes is maintained, as shown in Figure 3.5. In case an allocation request with size bigger than $2^{s_{max}}$ takes place, the request cannot be handled by the memory pool and thus is redirected to the standard C memory allocator function *malloc*. If the size of the memory request is within the

restricted limits, the allocator checks whether there is a free block in the free list. For example, let's assume the memory request has size S_{req}. The corresponding block size, i.e. 2^j, from the memory pool is the next greatest power of two, at least of size $2^{S_{min}}$ and $2^j \geq S_{req}$. If such block exists, the allocator removes the block from the free list and returns it as the response to the memory request. If there is no free block available in the free list, the allocator then performs a *refill* operation. Refill operation allocates a contiguous chunk of memory of size M_{refill} and divides it to *d* equal sized chunks where each size is 2^j bytes, since the refill operation is performed on this specific size. The allocator then adds the *d-1* chunks to the free list and returns the last chunk as a response to the memory request. Deallocating a block works similarly. The allocator finds the corresponding free list and adds the freed chunk as the head of the free list. If the free request is larger than the size $2^{S_{max}}$, the request is redirected to the standard C memory allocator to call the function *(*free).

There are two main advantages of using memory pooling. First, per-worker memory pools guarantee that the free lists are private to each worker, meaning there is no need for additional synchronization which eliminates synchronization overhead. Secondly, all allocation and deallocation requests can be handled in constant time. Although, refill operations cause additional overhead, the frequency of refill operations decrease as the maximum number of the used blocks increase during execution.

3.3.4 Dependence Management

In OpenStream, the producers and the consumers are matched dynamically in the runtime using streams. This section describes how the dependences are matched by giving examples for ordinary input and output views, followed by how the broadcasts are handled using peek views.

Management of Ordinary Input and Output Views

The Listing 3.1 shows a simple OpenStream program where two producers and one consumer are operating using a single stream. The example code calculates the square of each index value within the producers, and the consumer uses the calculated values to print the results.

```
1  int main{}
2  {
3      // Declaration of the stream
```

```
4       float a_stream __attribute__ ((stream));
5
6       // Declaration of horizon sizes
7       int horizon_out = 3;
8       int horizon_in = 6;
9
10      // Declaration of views
11      float out_view[horizon_out];
12      float in_view[horizon_in];
13
14      // Producer p0
15      #pragma omp task output(a_stream << out_view[horizon_out])
16      {
17          for (int i = 0; i < horizon_out; i++)
18          {
19              out_view[i] = i*i;
20          }
21      }
22
23      // Producer p1
24      #pragma omp task output(a_stream << out_view[horizon_out])
25      {
26          for (int i = horizon_out; i < 2*horizon_out; i++)
27          {
28              out_view[i] = i*i;
29          }
30      }
31
32      // Consumer c
33      #pragma omp task input(a_stream >> in_view[horizon_in])
34      {
35          for (int i = 0; i < horizon_in; i++)
36          {
37              printf("Result[%d] = %.2f\n", i, in_view[i]);
38          }
39      }
40
41      #pragma omp taskwait
42
43      return 0;
44  }
```

Listing 3.1: Two producers and one consumer operating on a single stream

In the example code, there are two producers, each produce three floating point type elements and one consumer consumes the total of six floating point elements that are matched using one stream *a_stream*. The horizons are the same size for the producers which is three, but the consumer's horizon is different, thus there are two horizon variables declared for the producers and the consumer. Although in the declaration of both producers, the out_view array is used in the output clauses, the producers, do not access the same elements of the stream. The compiler only uses the declaration of a

view to determine the element type and the horizon size of a view. Therefore, it is syntactically allowed to use the same view which can be used to access different data locations.

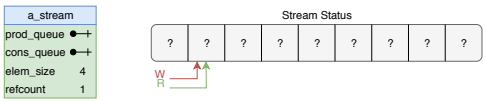
When a task is created, its frame is initialized, including all the data structures for the task's views. After the initialization, a run-time function *resolve_dependences* is called for each view in order to match the output views with consumers and input views with producers. The process of dependence matching for Listing 3.1 is illustrated in Figure 3.6.

Figure 3.6a shows the state of the run-time after the stream *a_stream* is created. The data structure for the stream is initialized, but still empty since there are no task present at this point. The stream status in the figure only represents the read and write positions of the stream, as well as the content. Furthermore, such data structure is not present in the run-time, but only shown for illustration purpose.

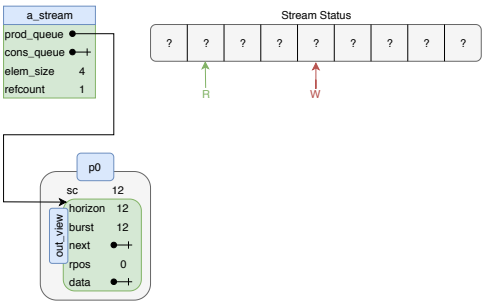
Figure 3.6b shows the state of the run-time after the first producer task *p0* is created. The task and its frame is created and this task is added to the *prod_queue* of the stream since it has not been matched with a consumer view yet. Although the matching of the output view is incomplete, the write position of the stream is advanced by the burst of the view, allowing subsequent producers to access the same stream. In the frame data structure, the horizon and burst values are set to 12 calculated by the size of each element type which is a single precision floating point, multiplied by the number of elements which is 3. The synchronization_counter of the frame is also set to 12 since the task only has one view and the horizon of that view is 12. The *next* pointer normally holds a pointer to the next unmatched view, but at this point, the lack of any other unmatched views result in this field to be set to *NULL*. The field *data* is also *NULL* due to incomplete matching.

The next step in the code is the creation of the second producer task *p1*. The producer_queue of the *a_stream* already has a pointer to the first producer task, so the second producer task is chained to the first one using the *p0's next* pointer. The task *p1* has identical characteristics for its burst, horizon and synchronization_counter fields. Task *p1* has only one output clause with the same burst size, resulting in the advance of three elements of the write position of the stream as shown in Figure 3.6c. The *rpos* value is 0 for both producer tasks at the beginning since both tasks have unmatched dependences.

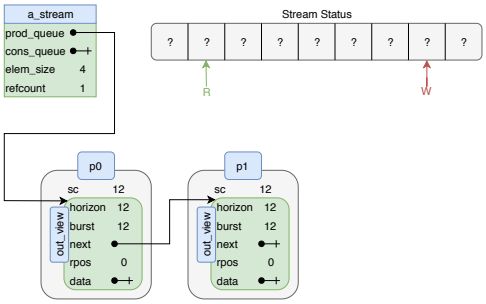
The declaration of the consumer task follows the declaration of two producer tasks. The creation of the consumer is illustrated in Figure 3.6d. The horizon and burst values



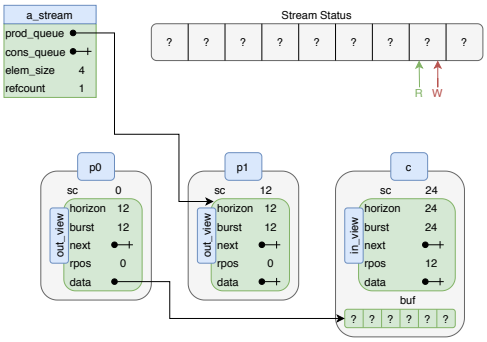
(a) Initial state of the stream



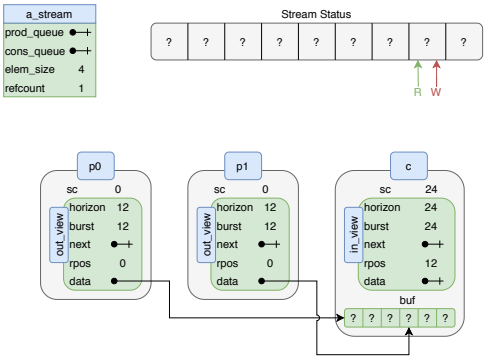
(b) Creation of the first producer



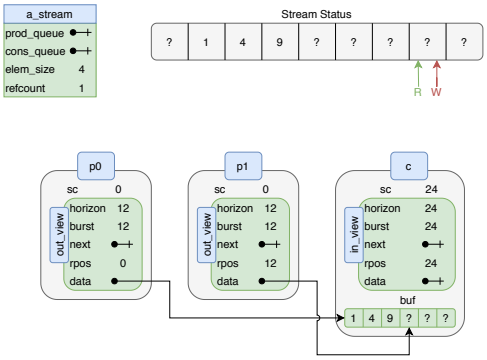
(c) Creation of the second producer



(d) Creation of the consumer and matching with the first producer

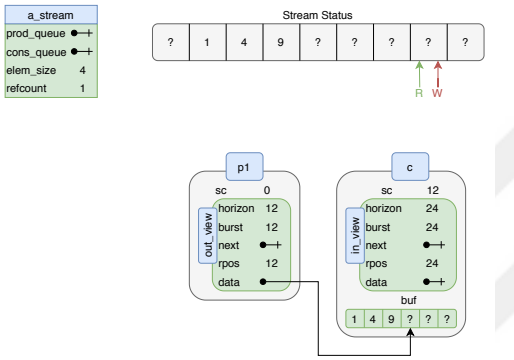


(e) Matching of the second producer and the consumer

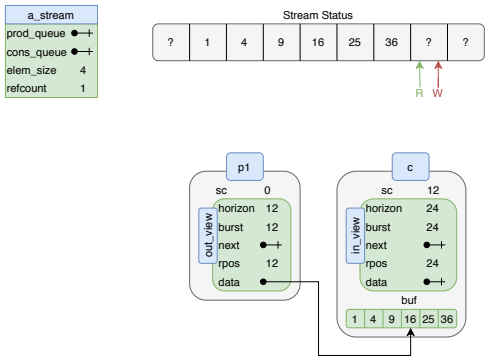


(f) Execution of the first producer

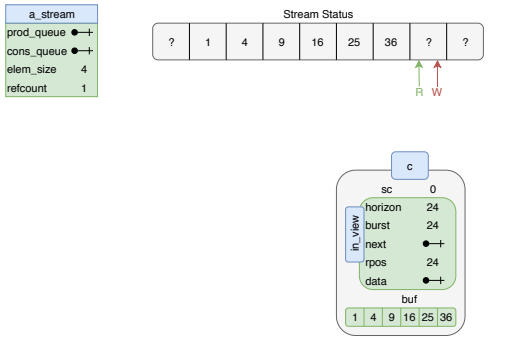
Figure 3.6: Dependence resolution of two producers and one consumer



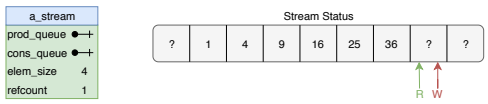
(g) Synchronization counter of the consumer is updated



(h) Execution of the second producer



(i) Tasks and views



(j) Execution is finished and the data structures freed

Figure 3.6: Dependence resolution of two producers and one consumer contd.

are 24 in the consumer and the synchronization_counter is also 24 since the task declaration has only one input clause. The consumer frame has a field called *buf* which is the actual memory location that holds the data. Once the consumer is initialized, the read position of the stream is advanced by 24 to allow subsequent stream accesses. When the *resolve_dependences* function is called within the run-time for the input view of the consumer, the run-time first checks the unmatched producers in order to find a match between the input and output views. If a match is not found, then the input view is added to the *cons_queue* of the stream. In this case, the output view of *p0* matches the input view of the *c* partially, in several steps. First, the data pointer of the output view is set to the current write position of the input view, using *rpos* value of as an index to access the data pointer of the input view. The *rpos* value is then updated according to the horizon of the output view of *p0*, i.e. by 12 as shown in the figure. Secondly, the output view of task *p0* is removed from the *prod_queue* of the stream. Lastly, the synchronization_counter of the producer is subtracted by the burst value, resulting in the value 0, which indicates the dependence matching for task *p0* is complete and the task is ready for execution. However, the reached position for the task *c* has not yet reached the horizon value, therefore *resolve_dependences* is called once more to match the remaining views. In addition to this, the *data* pointer for the task *p0* now points to the *buf* of the consumer, pointing to the actual memory address the producer is going to write once its execution finishes.

Task *p1* follows the same steps as *p0* to match its output view with the partially unmatched input view of *c*. Before the second matching, the *rpos* of the output view of *p1* is set to the *rpos* value of the input view at this point, which is 12. This is done in order to provide the required offset when multiple producers are matched with a single consumer. The result of this process is shown in Figure 3.6e. After the second dependence matching is complete, both producer tasks are ready to execute. However, the synchronization_counter of the consumer task is still unchanged and its value is 24, due to the fact that its input data becomes available only after the execution of the producers.

After the dependences are matched and the producers are ready to execute, each producer task gets scheduled to a persistent worker. Assume the task *p0* is executed first and finishes its execution. Figure 3.6f shows the state of the data structures. After the execution, the consumer has its synchronization_counter reduced by the burst of the output view of *p0*, resulting in an updated value of 12 in Figure 3.6g. After the

execution of the second producer *p1* as shown in the Figure 3.6h, the synchronization_counter of *c* becomes 0 and the consumer is now ready to execute. Consumer is executed as shown in Figure 3.6i followed by the freeing of the data structures as shown in Figure 3.6j.

Management of Broadcasts

The Listing 3.2 shows a simple OpenStream program where one producer and two consumer are using broadcasts on a single stream.

```
1  int main{}
2  {
3      // Declaration of the stream
4      float a_stream __attribute__ ((stream));
5
6      // Declaration of horizon
7      int horizon = 6;
8
9      // Declaration of views
10     float out_view[horizon];
11     float in_view[horizon];
12
13     // Producer p0
14     #pragma omp task output(a_stream << out_view[horizon])
15     {
16         for (int i = 0; i < horizon; i++)
17         {
18             out_view[i] = i+1;
19         }
20     }
21
22     // Consumer c0
23     #pragma omp task peek(a_stream >> in_view[horizon])
24     {
25         float sum = 0.0;
26         for (int i = 0; i < horizon; i++)
27         {
28             sum += in_view[i];
29         }
30         printf("Sum = %f\n", sum);
31     }
32
33     // Consumer c1
34     #pragma omp task peek(a_stream >> in_view[horizon])
35     {
36         float sums = 0.0;
37         for (int i = 0; i < horizon; i++)
38         {
39             sums += in_view[i]*in_view[i];
```

```
40            }
41            printf("Sum of squares = %f\n", sums);
42        }
43
44        #pragma omp tick (a_stream >> horizon)
45
46        #pragma omp taskwait
47
48        return 0;
49    }
```

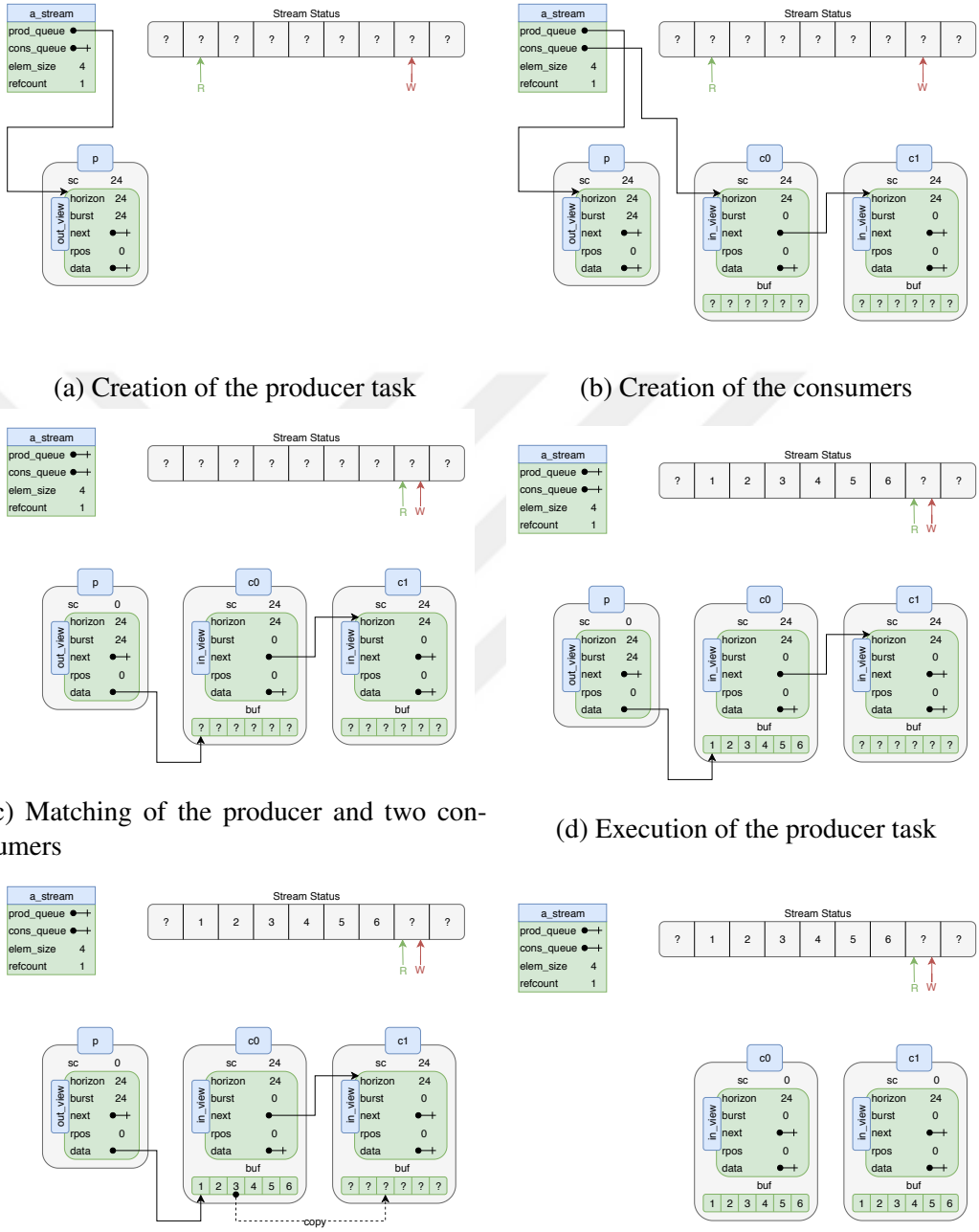
Listing 3.2: One producer and two consumers operating on a single stream using broadcasts

The code for the broadcast is essentially similar to ordinary input and outputs, the main difference being the usage of peek clause instead of input and the use of tick construct in order to advance the stream when the consumers read the broadcast data. The illustration of the broadcasts is shown in Figure 3.7.

The creation of the producer is identical to the one described in Section 3.3.4. Figure 3.7a shows the state when the producer is created. The two consumers are created next as shown in Figure 3.7b where the unmatched input dependences are chained to the cons_queue of the stream. When the *resolve_dependences* function is called by the run-time with a peeking view (input view with a burst of 0), the run-time does not match the dependences directly, but the matching is deferred until the execution reaches the tick construct. At this point, neither the producer, nor the consumers are ready to execute. Until the run-time reaches the tick construct, the created consumers are chained to the *cons_queue* of the stream. When the tick construct is reached, the read position of the stream is advanced by the declared amount, and the dependence resolution happens as shown in Figure 3.7c. The producer is then removed from the list of unmatched views and the synchronization_counter reaches zero. The producer task then executes as shown in Figure 3.7d and all elements of the first consumer view is written, but the task remains blocked until all consumers receive the broadcast data. When all consumers receive the broadcast data by copying the corresponding data from the first matched consumer as shown in Figure 3.7e, the consumers become ready for execution and the producer task's data structures can be freed as illustrated in Figure 3.7f. When the execution finishes, the remaining data structures are freed.

Constraints

As shown in the previous sections, OpenStream does not store any data directly in the streams, but in the input buffers of the associated views located in the data-flow



(a) Creation of the producer task

(b) Creation of the consumers

(c) Matching of the producer and two consumers

(d) Execution of the producer task

(e) Broadcast the data to the remaining consumers

(f) Consumers are ready for execution

Figure 3.7: Dependence resolution of one producer and two consumers using broadcast operation

frames of each task. Each view has only one field, *data*, pointing to the elements accessible through the view. The advantage of this layout is that the consecutive elements of a stream are stored at consecutive addresses which can be accessed by simply dereferencing the corresponding pointer. However, in order to guarantee the correct dependence information, some constraints need to be satisfied for a valid OpenStream program.

Constraint 3.1: *Burst of a reading view must either be equal to the horizon value or must be zero.*

This constraint prevents an arbitrary number of elements of an output view to become copied to multiple input views. For ordinary input views, the burst is equal to the value of horizon and for broadcasts the burst value is zero.

Constraint 3.2: *The elements of an output view cannot be scattered across multiple input views.*

This constraint prevents different horizons of output and input views of producers and consumers. For example, an output view with horizon 4 cannot be matched with two input views whose horizons are 2, it can only be matched with one input view of horizon 4. If there is a need for multiple consumers using partial data, a broadcast is required and tasks can mask out the unnecessary elements. Additionally, this constraint cannot be handled by the compiler due to the dynamic matching of producers and consumers, thus is handled by the run-time.

Constraint 3.3: *There must not be leftover stream elements.*

All the stream elements that are written into a stream must be read. Since the written elements are actually stored in input buffers of the reading views, unmatched output views cannot store any data. Therefore, each element in a stream must be read at least once.

Constraint 3.4: *For broadcasts, the number of consumers must be finite.*

The broadcast mechanism requires the *tick* construct in order to advance the reading position of the stream. In practice, this mechanism limits the broadcasts to a finite number of consumers.

3.4 Compilation of OpenStream Programs

There are two main steps in executing an OpenStream application: the compilation of the OpenStream program, and the run-time library that is linked to the application in order to dynamically execute the program.

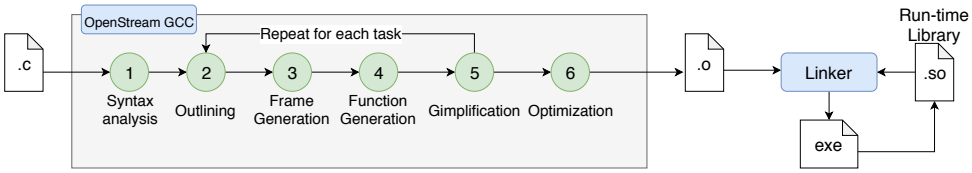


Figure 3.8: Compilation steps of OpenStream programs

During the compilation of an OpenStream program, the constructs and clauses described in Section 3.2 are translated into code that links with the OpenStream run-time library. The OpenStream compiler used in this thesis is implemented on top of GNU C Compiler version 5.4 [101], where the compiler retains its ability to compile valid C programs with the addition of compiling OpenStream specific constructs and clauses.

Figure 3.8 shows the required steps for compiling an OpenStream application. The steps can be ordered as follows:

1. **Syntax analysis:** The parser analyzes the input files and transforms the C statements into a tree representation called GENERIC [79]. OpenStream-specific clauses are translated into custom nodes of the tree and processed in later stages.

2. **Outlining:** The compiler creates the corresponding work-function for each task.

3. **Frame generation:** The compiler determines how much memory is required for the data-flow frame and its views.

4. **Function generation:** After the required memory space for a task frame is determined, this step generates the code for initialization of frame fields and the appropriate run-time functions. For instance, the required memory for the data-flow frame is allocated by calling the function of the memory pool as described in Section 3.3.3. Additionally, *resolve_dependence* function is added for each view.

5. **Gimplification:** In this step, the generated code is translated into the GIMPLE intermediate representation, widely used in GCC.

6. **Optimization:** Optimization steps are applied to the result of the Gimplification. This step finishes with the generation of instructions for the target architecture.

After the compilation steps finish, the resulting object files are supplied to the linker. The OpenStream run-time library is a separate shared library. Therefore, in order to resolve the symbols of run-time calls, the object files are linked with the run-time library after the compilation to create the final program executable.

3.5 Summary

In this chapter, we introduced OpenStream, a data-flow extension to OpenMP. We briefly discussed the terminology used in OpenStream, followed by the presentation of the syntax of OpenStream programs. Moreover, we presented the execution model of OpenStream, explaining persistent workers and the main data structures of the run-time as well as details on memory and dependence management during execution. Finally, we gave an overview of the compilation steps.

OpenStream is a state-of-the-art extension enabling data-flow task-parallel programs that is mainly used in the development of high performance applications [95]. The general trend for task-parallel languages is to use point-to-point data dependences between tasks in order to overcome the overhead created by barrier synchronization. However, the concepts presented in this chapter regarding stream accesses using views and dynamically matching producers and consumers are unique to OpenStream. These OpenStream specific concepts enable opportunities for efficient scheduling and data placement in NUMA systems [39] and can be applied to heterogeneous platforms as well [100].

The introduction of GPUs and FPGAs in high performance computing created programming difficulties for HPC applications due to the difference in programming models between different devices. If done efficiently, task-parallel run-times can handle the device specific aspects of the application, unburdening the programmer from manually coordinating the accelerators with potentially higher performance gains. To this end, we propose efficient scheduling mechanisms that exploit the traits OpenStream provides in order to increase performance in heterogeneous platforms.

The next chapter presents changes to the original run-time and the execution model in order to take advantage of the performance of heterogeneous systems. We have extended the OpenStream language to support GPUs and FPGAs, so that, the run-time has the necessary infrastructure to efficiently schedule tasks on heterogeneous systems.

Chapter 4

Extending OpenStream for Heterogeneous Systems

In heterogeneous systems that include accelerators be it GPUs or FPGAs or any other kind that may become mainstream in the future, it is essential for a run-time system to abstract the hardware details as much as possible while providing a programming model that requires few architectural details. It is the run-time's responsibility to make low level decisions on scheduling and memory management to unburden the programmer from such details.

In this chapter, we describe how the OpenStream run-time is extended in order to support accelerators mainly used in HPC systems composed of multi-core CPUs, GPUs and FPGAs. The extensions implement a programming model based on asynchronous execution of tasks on the accelerators and abstracts memory management details such as copying data from/to a device connected through PCI-express bus. Although accelerators can be included in the system using PCI-e bus as in GPUs, for FPGAs, we focused on next generation devices that are intended to be used in HPC systems which include multi-core CPUs and FPGAs on the same die in an SoC fashion.

This chapter starts with the extension of OpenStream targeting CPU-GPU platforms, describing how the programming model is extended to support task execution on GPUs. Section 4.1.2 discusses how the syntax is extended, followed by the run-time details Section 4.1.3. The extension for FPGAs are presented in Section 4.2 which includes the syntax additions as well as run-time implementation details.

4.1 Extension for GPUs

4.1.1 Execution Model of OpenStream-GPU

OpenStream's execution model is described in Section 3.3.4 that target homogeneous multi-core systems. In the aforementioned execution model, each *persistent worker* is mapped to one CPU core. However, in heterogeneous systems, there are multiple devices present with different capabilities, in this case, GPUs. GPUs are throughput-oriented devices that has thousands of execution units that can run thousands of thread simultaneously. Although GPUs offer high throughput, the main difficulty in GPU programming is the device memory being separate from the system memory. Therefore, extending the homogeneous OpenStream execution model for heterogeneous platforms bears three fundamental challenges for memory management arising from the existence of multiple memory resources with distinct address spaces:

- Since tasks can be executed either on CPUs using the host's main memory or on GPUs with dedicated memory and a distinct address space, data buffers must be allocated according to the execution location of the accessing producers and consumers.
- If a producer and its consumer do not execute on computing units sharing the same address space, data must be transferred between memory resources before execution of the consumer.
- Since GPUs cannot access host memory directly, run-time data structures, such as work queues, remain inaccessible and scheduling for GPUs must be performed by a core of the host.

Using task-private buffers, memory allocation and data transfers can be handled transparently by the run-time, unburdening the programmer from memory management. If data is handled explicitly, management must be performed either by the application itself or through additional steps by the run-time or compiler to transparently rewrite memory addresses. The last two challenges of the list above also involve invocation of specialized APIs for GPUs. Dedicating one CPU core for each device simplifies the coordination and scheduling of tasks and data transfers on GPUs. With an increasing number of cores per accelerator in recent systems and the instruction throughput of GPUs largely exceeding the throughput of CPUs, this sacrifice has only very limited impact on performance. Therefore, our approach dedicates one core to

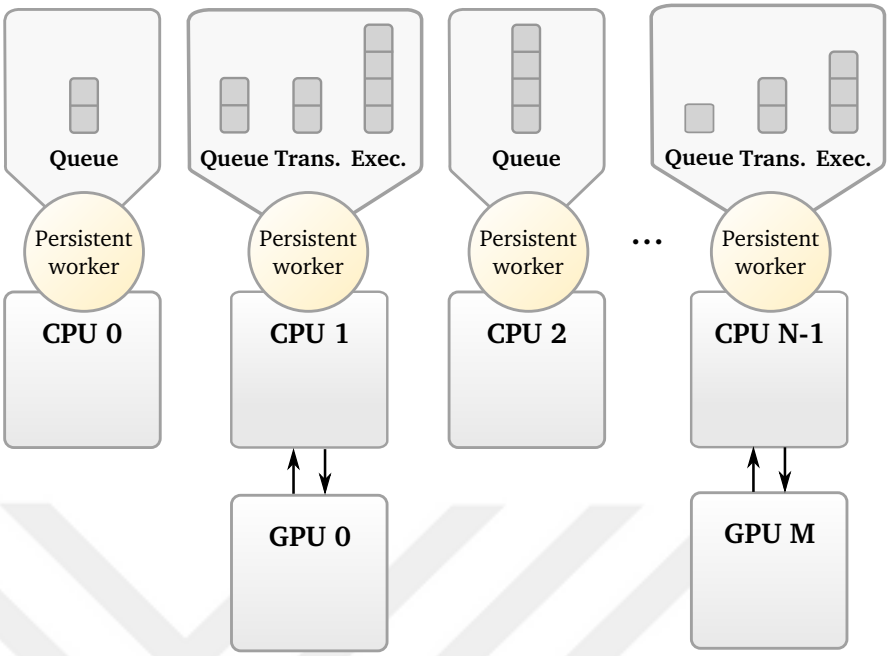


Figure 4.1: Persistent workers extended for GPU support in OpenStream

each GPU, named as *GPU dedicated core*, and uses the persistent worker thread of that core as a proxy for offloading.

Figure 4.1 illustrates the extended execution model for GPUs. In the extended model, the *work cache* is no longer present. The activated tasks are directly pushed to the bottom of the work queue instead of being pushed to the work cache. The fundamental reason for removing the work cache is, the tasks within the work cache cannot be stolen and are executed by the CPU workers. When there is not enough parallelism in the program, the use of work cache limits the number of tasks that can be assigned to the GPU, especially for systems with high number of CPU cores. Considering the GPU has higher throughput compared to a CPU core, the use of work cache possibly limits the performance whereas removing the work cache does not introduce additional overhead for the CPU workers.

CPU 1 and CPU N-1 in the figure are used as *GPU dedicated cores* and include additional queues for handling the task and memory management operations on the GPUs, named *transfer queue* and *execution queue* respectively. The *execution queue* is responsible for offloading tasks onto GPUs while *transfer queue* is responsible for handling the data transfers between the host and the GPUs. The detailed usage of these queues are given in Section 5.2 since these queues are mainly related with the scheduling technique presented in this thesis.

The fundamental functional challenges above must be addressed in order to be capable of executing tasks on both CPUs and GPUs at all. In addition to these exist a number of challenges for efficient execution that need to be addressed, for leveraging the computational capabilities of every resource in the system to increase performance.

- Since the raw instruction throughput differs significantly between CPU cores and GPUs, the amount of work required to fully utilize computing capabilities also differs.
- In order to avoid round-trip delays between the host and GPUs, multiple tasks should be offloaded at once.
- Resources that can be used independently, such as GPU cores and the interconnect between host and device memory, should be used in parallel.
- Assuming the raw throughput of a GPU is higher than for a CPU, the GPU idle time has a higher impact on performance. Hence, unblocking tasks as fast as possible gains importance.

Control over the amount of work can be achieved by either breaking larger tasks into smaller units of work or by aggregating very small tasks to the desired granularity. The former approach might require sophisticated mechanisms to extract parallelism from a sequence of instructions of a task, which forms a field of research on its own [56, 16, 102, 104, 84]. The latter approach makes use of parallelism already made available by the program and might be implemented as part of a strategy addressing round-trip delays and data locality of GPU tasks. Finally, overlapping of transfers with computation requires transfers to be scheduled for periods of GPU activity.

4.1.2 Syntax of OpenStream Programs Employing GPUs

As described in Section 3.2, OpenStream uses OpenMP task constructs with run-time specific additional clauses. In order to offload tasks to the GPUs, the same task construct is used with additional clauses to pass additional information related with the GPU execution of the task. The clauses for the use of a GPU task is as follows:

```
1  #pragma omp task input(stream_expr >> view_expr, ...)  |
2                   output(stream_expr << view_expr, ...)
3                   cl_source (kernel_filename_str)
4                   cl_kernel (kernel_name)
```

```
5                    cl_args (arg1, arg2, ...)
6                    cl_dimensions (ND_dimensions)
7                    cl_global_work_offset (size_dim0, size_dim1, ...)
8                    cl_global_work_size (size_dim0, size_dim1, ...)
9                    cl_local_work_size (size_dim0, size_dim1, ...)
10 {
11     task_body_CPU
12 }
```

The original task construct requires a task body that executes when the task is scheduled to a persistent worker. The generation of the task body is done during the compilation of OpenStream programs as discussed in Section 3.4. However, the GPU binary for each kernel is not present during the compilation process, but is only available during run-time. Therefore, each GPU kernel needs to be compiled during run-time after the environment and the context for the GPU is created. The *cl_source* clause is used to determine where the GPU kernel source code resides, and requires a string literal as the filename. The *cl_kernel* is the kernel name that corresponds to the GPU implementation of the task. *cl_args* clause is used to pass the necessary pointers to the GPU kernel. The order of the arguments is the same as the order they are declared in the kernel definition. Each argument must be a pointer which will be transferred to the GPU memory space before the GPU task can execute.

The last three clauses are related with the kernel execution mechanisms of OpenCL. OpenCL can execute the GPU binary in multiple data dimensions, hence called N-dimensional execution. The programmer is required to define the number of dimensions the kernel uses using *cl_dimensions* clauses. The OpenCL API also requires the work size for each dimension which can be declared using *cl_global_work_size* clause as well as offsets to be used in case the programmer aims to access only some specific part of the data using *cl_global_work_offset* clause. Although GPUs implicitly use predefined local work sizes, the programmer is able to change the local sizes using the *cl_local_work_size* clause for tuning the kernel performance if required.

The declaration of a task body is a requirement in OpenStream. In case the programmer declares the additional clauses for the GPU, the task keeps both task bodies to decide whether to use the CPU task body or the GPU task body during execution. On the other hand, the tasks that has only CPU task bodies can only execute on the CPU workers.

Example of a GPU Task in OpenStream

Listing 4.1 shows a valid OpenStream program which has one task with a GPU implementation. The example code is simple and multiplies all the elements of a vector with a constant value. The GPU tasks require two things: the kernel implementation of the task in OpenCL, and the additional GPU clauses to be declared in the task construct of the corresponding task.

```
1  int main()
2  {
3      int a = 5, N = 10;
4      int x __attribute__((stream)), y __attribute__((stream));
5      int vx[N], vy[N];
6      int *a_ptr = &a;
7
8      #pragma omp task output(y << vy[N])
9      {
10         for (int i = 0; i < N; i++)
11             vy[i] = i;
12     }
13
14     #pragma omp task output(x << vx[N])          \
15                      input(y >> vy[N])           \
16                      cl_source("kernels.cl")     \
17                      cl_kernel(vadd)             \
18                      cl_args(vy, vx, a_ptr)      \
19                      cl_dimensions(1)            \
20                      cl_global_work_offset(0)  \
21                      cl_global_work_size(N)      \
22                      cl_local_work_size(16)
23         {
24             for (int i = 0; i < N; i++)
25                 vx[i] = a * vy[i];
26         }
27
28     #pragma omp task input(x >> vx[N])
29         {
30             for (int i = 0; i < N; i++)
31                 printf("vx[%d] = %d\n", i, vx[i]);
32         }
33
34     #pragma omp taskwait
35
36         return 0;
37 }
```

Listing 4.1: A valid OpenStream program with a GPU task is declared.

The task declared in line 14 of the example code has one input and one output with the additional GPU clauses. The clause *cl_source* is set to the name of the file that has

the OpenCL kernel implementation for this task. During execution, the run-time reads the *kernels.cl* file and compiles the OpenCL kernel into GPU specific binary. The run-time queries the available devices on the system at the beginning of the execution and uses on-line compilation to generate the kernel binary. The content of the *kernels.cl* file is shown in Listing 4.2. The clause *cl_kernel* is declared with the kernel name that corresponds to the task which must be declared inside the specified file. The arguments are passed in the order of the kernel function and specified in the *cl_args* clause. Although OpenCL API allows literal arguments to be passed, OpenStream restricts the arguments to be pointers, thus a pointer is declared to the constant value of *a* and this pointer is used within the clause. The dimensions and the work sizes are also declared as required to execute any kernel on the GPU. Global work size corresponds to the number of global work items that will execute the kernel. Local work size is the number of work items that make up a work-group that will execute the kernel. The breakdown of executing kernels to work-groups and work-items are described in detail in Section 2.2.2. The programmer can also provide work offset vector if any offset needs to be used for each work size in every dimension.

```
1  __kernel void vadd (__global const int *A,
2                      __global int *B,
3                      __global const int *k)
4  {
5    int id = get_global_id (0);
6
7    B[id] = *k * A[id];
8  }
```

Listing 4.2: The content of kernels.cl file which includes the implementation of one OpenCL kernel, vadd.

Listing 4.2 shows the kernel code implemented in OpenCL C language. The function is identical to the CPU implementation, but each GPU work-item calculates one element of the output vector instead of a CPU worker calculating all the elements in an iterative fashion.

4.1.3 Run-time Implementation

Extended Data Structures

OpenStream run-time for supporting GPU execution includes additional data structures to hold the meta-data information on GPU tasks as shown in Figure 4.2. Each frame

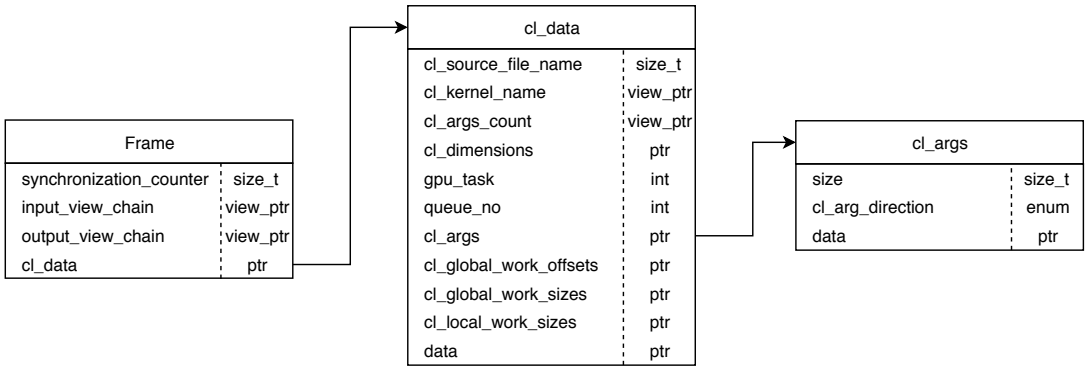


Figure 4.2: Extended Frame data structure and cl_data structure

includes a pointer to a data structure named *cl_data* which has the following fields:

- cl_source_file_name: the name of the file where the kernel implementation resides.
- cl_kernel_name: the name of the kernel that is to be executed on the GPU.
- gpu_task: the initial value of this field is -1, meaning the task is a CPU task. This field is updated when the task is decided to be executed on the GPU and the value of the field is updated to the GPU id of the system which is assigned by the OpenCL driver on the system. Using non-negative values are required in order to support multiple devices.
- cl_args: a pointer to each argument of the OpenCL kernel in data structure form.
- cl_global_work_offsets: required parameter by the kernel execution in case the user wants to offset the data pointer to be processed in the kernel.
- cl_global_work_sizes: a vector that holds the global work size values passed by the clause with the same name.
- cl_local_work_sizes: a vector that holds the local work size values passed by the clause with the same name.

The *cl_args* data structure consists of the following fields:

- size: size of the argument in bytes
- cl_arg_direction: whether the argument corresponds to an input view, an output view or a *firstprivate* scalar variable

horizon	size_t
burst	size_t
next	ptr
owner	ptr
reached_pos	size_t
data	ptr
cons_view	ptr
opencl_buffer	ptr
opencl_event	ptr

Figure 4.3: Extended view data structure

- data: the address of the data for a particular argument. Essentially used for passing *firstprivate* arguments

The additional data structures are used to manage the execution in run-time. When a data-flow frame is created and has the GPU clauses present, *cl_data* is allocated for each frame and its fields are declared using the information included in the GPU clauses. The tasks are set to CPU task by default at creation time and the scheduler decides where the task is going to be executed during run-time.

The additions on the data-flow frame data structure is required to determine the target device of a task as well as used to offload a task to the GPU. However, as discussed in Chapter 3, dependences are satisfied using view data structures. Therefore, in order to manage the dependences between different devices, the view data structure is extended as shown in Figure 4.3. Each view has two additional fields as follows:

- opencl_buffer: a pointer to the buffer object created for the data region that resides on GPUs

- opencl_event: a pointer to an OpenCL event to orchestrate the asynchronous data transfers and kernel execution on the GPU

Once a task is decided to be executed on the GPU, the run-time first checks where the input data of the task resides. If the input view data resides on the CPU address space, the *opencl_buffer* field of the view is used to create a data buffer on the GPU address space and the data transfer between address spaces is issued. The data transfer is asynchronously issued and it creates an OpenCL event which is saved in the *opencl_event* field. The saved event is then passed to the kernel execution API call for synchronization between data transfers and kernel execution.

OpenCL Environment Management

The OpenCL environment is created and managed dynamically inside the OpenStream run-time. At the beginning of the execution of an OpenStream application, the run-time queries the available OpenCL compliant platforms present in the system. Different vendors have different platform identification, thus in order to use multiple devices from different vendors, an OpenCL context must be created for each platform and each device. After the creation of contexts, three command queues are created that correspond to each device; one queue for task execution, and two queues for each direction of data transfers. Having multiple command queues enable asynchronous enqueuing of kernel execution and transfer operations.

After the creation of the context for each device, the next step is to create an executable binary for each kernel for each device. Although a kernel implementation can be executed in all supporting devices, the OpenCL code must be compiled for each device using the vendor provided OpenCL C compiler. Therefore, the run-time compiles each kernel for each device and saves the compiled binary for future use to avoid the compilation overhead.

When the program execution finishes, the OpenCL environment is destroyed by freeing all program information, command queues and the OpenCL contexts. The management of OpenCL environment is integrated with OpenStream run-time and does not require any additional effort from the programmer.

4.2 Extension for FPGAs

4.2.1 Execution Model of OpenStream-FPGA

In order to employ FPGA accelerators in OpenStream, the execution model is updated since the FPGA accelerators are not able to execute any task independently. Therefore we have followed a similar path to that of GPU extension by assuming a host-device model whereas FPGA is the device and multi-core CPU is the host.

Figure 4.4 illustrates the extended execution model for FPGAs. Similar to the GPU execution model, one CPU core is dedicated for orchestrating the FPGA accelerators. However, the difference between the GPU and FPGA extensions is that the FPGA dedicated core uses only one work dequeue and additional accelerator buffers. While GPU execution model uses three queues to manage data transfers and execution on a discrete device, the FPGA execution model targets system-on-chip FPGA devices

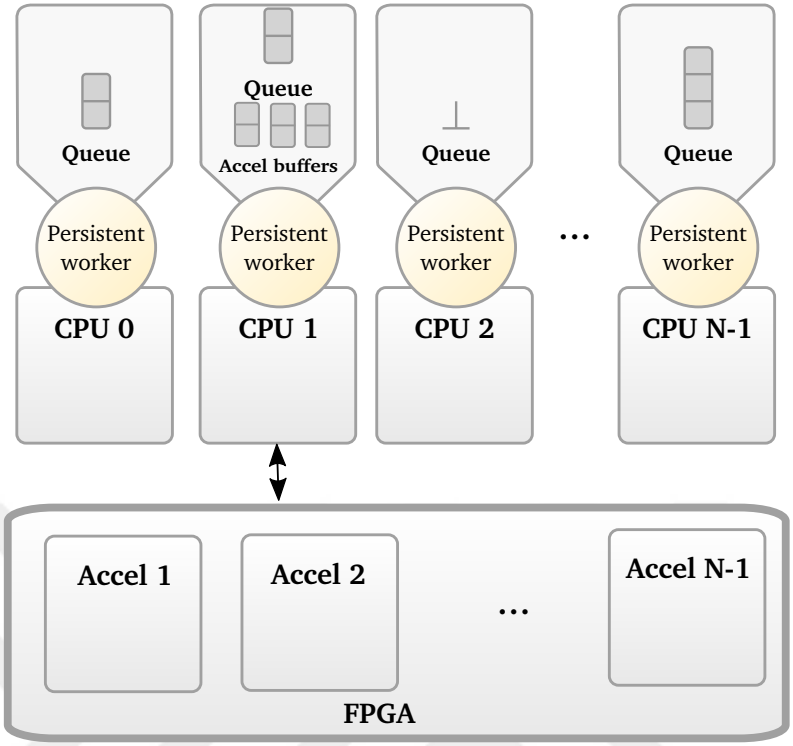


Figure 4.4: Persistent workers extended for FPGA accelerator support in OpenStream

which does not require data to be transferred over the PCI-e bus. Therefore, one local work queue is sufficient to manage task execution and data placement on the FPGA.

In the FPGA case, our extended model assumes there is only one FPGA present on the system with multiple accelerators with possibly different types where only one CPU core is dedicated to manage all the accelerators. Aside from the local queue, the FPGA dedicated core has a structure called accelerator buffers which keeps the ready tasks that can be accelerated using the FPGA. The tasks in the accelerator buffers cannot be stolen by other workers, similar to the work cache in the original OpenStream run-time. The size of the accelerator buffers is determined at the beginning of the execution of the program and set to the number of each accelerator type. For example, if there are three different types of accelerators available on the FPGA three buffer instances are created. In addition to this, the run-time counts how many accelerators of each type is available on the FPGA to set the size of each accelerator buffer to the corresponding number. Using the accelerator buffers, the run-time ensures the tasks assigned for FPGA execution are not stolen by other workers and the FPGA accelerators are kept occupied by task execution. The accelerator buffers are simple double

ended queues, only accessible by the FPGA dedicated core which handles the enqueue-dequeue operations to the buffers.

4.2.2 Syntax of OpenStream Programs for FPGA Acceleration

The OpenMP task construct is extended with additional FPGA specific clauses in order to employ FPGA accelerators during execution. The clauses for the use of an FPGA task is as follows:

```
1  #pragma omp task input(stream_expr >> view_expr, ...)  |
2                   output(stream_expr << view_expr, ...)
3                   accel_name (accelerator_name)
4                   args (arg1, arg2, ...)
5                   work_offset (size_dim0, size_dim1, ...)
6                   work_group_size (size_dim0, size_dim1, ...)
7 {
8     task_body_CPU
9 }
```

The run-time assumes the FPGA is already programmed to include the accelerators by the programmer. Therefore, the syntax extension for the FPGAs does not include any accelerator specific clauses, but only includes a file name for a configuration file whose content is the physical addresses of the accelerators.

The run-time reads the configuration file at the beginning of the execution and creates the corresponding data structures according to the accelerator and address specifications. Additionally, the FPGA clauses are used to determine the accelerator for each task to be executed on the FPGA accelerators. The *accel_name* clause is used to match the task with the corresponding accelerator on the FPGA, requiring the accelerator name in string literal form that matches the one provided in the configuration file. The *args* clause requires pointers to the arguments that is going to be passed to the accelerator. The order of the arguments is the same as the order they are declared in the accelerator design phase. When an accelerator is programmed using OpenCL in HLS for FPGAs, the HLS compiler creates OpenCL specific control fields such as *work_group_size* and *work_offset*. In order to match these control variables, the OpenStream syntax requires additional clauses with the same names. Note that, different from the GPU clauses, there is no requirement for a dimension clause since the HLS compiler creates three dimensional work group and work offset sizes regardless of the implementation and the OpenStream compiler by default sets the unused dimensions to the default value of 1, indicated by the OpenCL specification [64].

Example of an FPGA Task in OpenStream

Listing 4.3 shows a valid OpenStream program which has a task with additional FPGA clauses. The example code is similar to the GPU example and calculates the multiplication of two vectors.

```
1  int main()
2  {
3      int N = 10;
4      int x __attribute__((stream)), y __attribute__((stream));
5      int vx[N], vy[N], vz[N];
6
7      #pragma omp task output(y << vy[N])
8      {
9          for (int i = 0; i < N; i++)
10             vy[i] = i+1;
11     }
12
13     #pragma omp task output(z << vz[N])
14     {
15         for (int i = 0; i < N; i++)
16             vz[i] = i+1;
17     }
18
19     #pragma omp task output(x << vx[N])          \
20                      input(y >> vy[N])           \
21                      input(z >> vz[N])           \
22                      accel_name(VectorAdd)       \
23                      args(vy, vz, vx)            \
24                      work_offset (0)             \
25                      work_group_size (N)
26
27         {
28             for (int i = 0; i < N; i++)
29                 vx[i] = vy[i] * vz[i];
30         }
31
32     #pragma omp task input(x >> vx[N])
33         {
34             for (int i = 0; i < N; i++)
35                 printf("vx[%d] = %d\n", i, vx[i]);
36         }
37
38     #pragma omp taskwait
39
40         return 0;
41 }
```

Listing 4.3: A valid OpenStream program with one FPGA task.

The task declared in line 14 of the example code has two inputs and one output

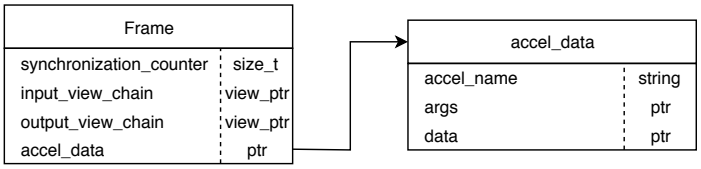


Figure 4.5: Extended Frame data structure for FPGA use

with the additional FPGA clauses. The clause *accel_name* is set to the name of the function, VectorAdd, that is declared in the configuration file. The content of the *accelerators.cfg* file is shown in Listing 4.4. The clause *args* are set in same order the accelerator requires. The clause *work_offset* is set to zero since the calculation needs to start at the beginning of the array and the clause *work_group_size* is set to the size of the task which translates the FPGA accelerator executes one work group per accelerator.

```
1   VectorAdd 0xA0100000
```

Listing 4.4: accelerators.cfg file includes the accelerator names and their base addresses defined during FPGA design

4.2.3 Run-time Implementation

Data Structures for FPGA Tasks

Similar to the GPU extension, in the FPGA case, we have extended the OpenStream run-time with additional data structures to manage the task execution on FPGA accelerators. The extensions are applied to the public version of OpenStream, not to the GPU extended version. Figure 4.5 shows the extended data-flow frame data structure and the frame has the following fields:

- accel_name: accelerator name that matches one of the accelerators defined in the configuration file.
- args: a pointer to each argument of the accelerator
- work_offsets: required parameter by the accelerator in case the user wants to offset the data pointer to be processed in the kernel.
- work_group_sizes: a vector that holds the work group size values passed by the clause with the same name.

fpga_accelerator	
accel_name	string
base_address	uint
state	uint

Figure 4.6: FPGA accelerator data structure to manage each accelerator

In addition to the extended data-flow frame, the run-time also requires additional data structures to manage each accelerator. Figure 4.6 shows fpga_accelerator data structure that is declared at the beginning of the execution for each accelerator defined in the configuration file and used for offloading tasks to the corresponding accelerators, keeping track of the accelerator state. The *fpga_accelerator* data structure consists of the following fields:

- accel_name: accelerator name that matches one of the accelerators defined in the configuration file.
- base_address: the base physical address of the accelerator on the FPGA fabric.
- state: the state of the accelerator that corresponds to the control bits of the accelerator.

Since the argument addresses are defined during the design phase of the accelerator, the argument addresses are statically managed by the run-time. Although it is possible to manage dynamic argument addresses, we chose static management since it does not affect the scheduling strategy in any way except the requirement of additional constraints during the design phase of the accelerator. Moreover, this study focuses on the scheduling aspect rather than creating a fully automated infrastructure, thus we have only implemented the basic requirements to show the effect of our novel dynamic scheduling technique.

FPGA Accelerator Management

As the program execution starts, the configuration file is read from *accelerator.cfg* file and the accelerator objects are created for each defined accelerator. All the accelerator states are set to *IDLE* at this point by the FPGA upon programming and the control bits are changed every time a task starts executing on the corresponding accelerator as well as it finishes the execution. The state is then used to offload more tasks or wait until the

execution finishes, while looking for available accelerators by pulling the state control bits of the accelerators.

Aside from the accelerator setup, the run-time also requires additional memory buffers to manage the FPGA context. At the beginning of the execution, a chunk of memory is allocated for the FPGA use and mapped to the virtual address space of the program. Since the streams are created for the CPU address space and due to the virtual-to-physical address translation differences between CPU and FPGA, additional memory buffers are necessary to pass data to the FPGA physical address space. The size of the mapped memory region is dependent on the number of accelerators defined in the configuration file. For our experiments, we have used 1024 pages of memory for each accelerator, but this value may change depending on the size requirements of the accelerator arguments.

When the program execution finishes, the accelerator data structures are destroyed as well as all the mapped memory region used for the accelerator management are freed.

4.3 Summary

In this chapter, we introduced GPU and FPGA extensions to OpenStream. These extensions were required to execute OpenStream programs on heterogeneous platforms. We discussed how the syntax changes with additional compiler annotations and the additional programmer effort in order to execute the tasks on GPU and FPGA accelerators. We also explained how the additional run-time data structures are implemented for heterogeneous execution.

All these run-time extensions are used for efficient scheduling of data-flow tasks on heterogeneous platforms. In the next Chapters we explain the novel scheduling techniques we propose for efficient scheduling of data-flow tasks, taking advantage of the data-flow information provided by the OpenStream run-time. Chapter 5 introduces our novel scheduling technique for GPUs and Chapter 6 introduces our dynamic scheduling approach on heterogeneous systems that incorporate FPGAs.

Chapter 5

Dynamic Scheduling on GPUs

One of the main advantages of data-flow task-parallelism is that the relation between tasks and data is explicit. As the working set of each task is known, as well as the flow of data between each producer and consumer task, the run-time system can make precise decisions about task and data placement. For example, it might decide to offload a consumer task to the GPU if the producer has already been executing on the GPU and the output data is already present in GPU memory. This choice might further depend on the size of the data being potentially reused on the device. For example, if a producer has several consumers that cannot all be offloaded to the GPU, it might decide to offload the consumer with the highest amount of data reuse on the GPU. More generally, information on data dependences enables reconstruction of the task graph and allows the run-time to plan ahead and make decisions for entire groups of tasks.

The combination of data-flow information with implicit buffer management facilitates better memory management decisions. Since tasks do not access fixed memory addresses, the run-time has full control over memory allocation and can decide whether a buffer should be allocated in host or device memory. If needed, the run-time can transparently change the location of buffers. This is not the case for run-times that use explicit memory handling, such as OmpSs [26], which requires an intermediate memory copy operation in order to transfer data between devices. Furthermore, whenever a producer and its consumer execute on different devices, and a lengthy memory transfer is necessary, the run-time can transparently transfer data and schedule the transfer such that it overlaps with task execution. As the run-time is able to plan the task schedule ahead, it is also able to schedule the data transfers required to minimize the amount of time wasted waiting for data to arrive.

In this chapter, a novel scheduling strategy for dynamically scheduling data-flow

tasks on heterogeneous platforms is presented. This description is divided into two parts: a presentation of the scheduling algorithm selecting tasks for offloading to the GPU and a description of the actions carried out when a task is about to execute. After establishing our scheduling strategy for CPU-GPU heterogeneous platforms, the experimental setup is presented in Section 5.3 followed by the results in Section 5.4.

5.1 Dynamic Scheduling of Tasks on GPUs

The aim of the proposed scheduling technique is twofold: (1) to improve data locality, increasing on-device data reuse, which reduces the data transfers between host and device as well as the GPU idle time while tasks wait for data; and (2) to balance the load between devices, taking into account the different computational capabilities of resources in heterogeneous systems. In contrast to existing work, where locality-aware techniques focus on the amount of data transferred and use random work-stealing for load balancing, our technique optimizes the scheduling at a finer resolution, additionally taking into account the smaller dependences and the platform asymmetry for task and data placement rather than randomly choosing a new task to execute when the GPU becomes idle.

The scheduling algorithm starts by selecting a ready task that is GPU compatible, which is called as an *entry task*. GPU compatibility in this case means the task has additional GPU clauses in its description as well as a kernel implemented in OpenCL. Once an entry task is found, this task is marked for offloading to the GPU and the scheduler starts traversing the task graph by following the entry task's output dependences. The task graph is traversed in breadth-first fashion, looking for the consumer task with the largest amount of dependence. Once the task with the largest dependence is found, the consumer task is marked as a GPU task and the traversal continues following the largest dependence consumers ensures that the tasks offloaded to the GPU will reuse the most data produced on the GPU, thus improving locality. This recursive marking scheme ends when there are no more descendants of the of the entry task eligible for GPU execution within the portion of the task graph that is dynamically instantiated. Once the marking of the tasks finishes, a data transfer is initiated for each input data dependence of the entry task that resides on the CPU. The pseudo-code for the task marking is shown in Algorithm 1.

When a CPU task finishes and has an output dependence to a GPU task, the output data needs to be transferred to the GPU. Upon completion of the CPU task, the data

Algorithm 1 mark_tasks(entry_task)

```
1:  entry_task.gpu_task ← true
2:  List.addLast(entry_task)
3:
4:  while !List.empty() do
5:      T ← List.getFirst()
6:      D ← out_deps(T)
7:      LD ← get_largest_dependent_task(D)
8:      if has_gpu_implementation(LD) then
9:          LD.gpu_task ← true
10:         List.addLast(LD)
11:     end if
12: end while
13:
14: transfer_queue.enqueue(entry_task)
```

transfer call is issued to the GPU dedicated core. Conversely, when a GPU task finishes execution and satisfies the last input dependence of a CPU task, the CPU task is pushed to the local queue of the GPU dedicated core. This decision results in smaller dependences of a GPU task being kept in the local queue of the GPU dedicated core. When the GPU becomes idle, these tasks can be offloaded to the GPU. As their data is already on the GPU, this increases data locality compared to randomly stealing work from other CPU cores.

To obtain new tasks for GPU execution, the GPU dedicated core first checks its local queue to identify a new entry task that has at least some input data on the GPU, allowing for exploitation of smaller dependences. When the local queue is empty, a new task is obtained through work-stealing from other CPU cores as a last resort.

Keeping tasks with GPU dependences locally improves efficiency in two ways. First, the scheduler avoids data transfers between devices, even if the amount of data is small. This helps not only to issue less transfers over the PCI-e bus, but also decreases the amount of input data that needs to be transferred for next task. Secondly, acquiring tasks from the local queue is faster than random work-stealing.

Finally, our scheduler avoids introducing communication latency on the critical path by pro-actively scheduling tasks and data transfers to the GPU instead of waiting for the GPU to become idle before seeking new work.

5.2 Execution of Tasks on GPUs

The novel scheduling technique proposed in this thesis uses three FIFO queues for the management of data transfers and kernel execution on the GPUs. The first two queues are used for data transfers from host to GPU, named *hostToDevQueue*, and GPU to host *devToHostQueue*. The third queue, *executionQueue*, is used for executing tasks on the GPU. Using different queues allows the run-time to issue data transfers and tasks execution asynchronously, which enables overlapping of data transfers in both directions with kernel execution. Although any CPU core can issue transfer requests to the *hostToDevQueue*, the GPU dedicated core is responsible for starting the transfers between devices using OpenCL API to avoid transfer initiation overhead on CPU *compute* cores.

Asynchronous calls prevent the GPU dedicated core from being blocked when data transfers and kernel executions are enqueued. By using OpenCL events for synchronization between asynchronous data transfers and kernel executions, the responsibility of keeping the consistency between transfers and execution can be delegated to the GPU. This delegation not only moves the need for synchronization from CPU to the GPU to prevent task stalls, but also allows future GPU tasks to be scheduled before their dependences are satisfied.

When all the dependences of a task are satisfied and the data transfers are asynchronously enqueued, the task is pushed to the *executionQueue*. Note that a kernel execution can already be enqueued before the data transfer is completed—the only requirement is that the transfer has been initiated. Finally, as the *executionQueue* is in-order, it is sufficient to ensure that the total order of tasks enqueued on it is a compatible restriction of the partial order defined by the task dependence graph. This guarantees that all of the task dependences satisfied within the GPU are implicitly enforced by the enqueuing order.

Once the execution of a GPU task is initiated, the scheduler initiates data transfers from the device to the host for each task among the consumers that is to be executed on a CPU. Although such transfers do not block the execution on the GPU, promptly handling device to host transfers enables more CPU tasks that depend on any GPU task to be executed. A callback mechanism is employed for handling the device to host data transfers, which informs the CPU task that the transfer is finished and the dependence satisfied. The event callback mechanism is supported since the OpenCL 1.1 [64] specification allowing notification from the GPU when the state of an OpenCL event changes to a specified state. In this case, once the state of a data transfer call from

the device to the host reaches *CL_COMPLETE* state, the callback function is executed in order to update the corresponding synchronization counters to keep the dependence management up-to-date.

Algorithm 2 summarizes the algorithm executed by the GPU dedicated core, which initiates the data transfers and kernel executions on the GPU. When there is no work in neither transfer queues nor the execution queue, the GPU dedicated core obtains work by first checking its own local queue. The tasks in the local queue of the GPU dedicated core are the ones that are not yet defined as a GPU task. Failing to acquire a task from the local queue results in random work-stealing from another CPU core.

Algorithm 2 execution loop of GPU dedicated core

```
1:  if transfer_queue.front() then
2:      T ← transfer_queue.dequeue()
3:      transfer_data_host_device(T)
4:      execution_queue.enqueue(T)
5:  end if
6:  if execution_queue.front() then
7:      T ← execution_queue.dequeue()
8:      execute_on_gpu(T)
9:      D ← out_deps(T)
10:     for all d ∈ D do
11:         if d.gpu_task ≠ true then
12:             transfer_data_device_to_host(d)
13:         end if
14:     end for
15: end if
16:
17: T ← obtain_work()
18: mark_tasks(T)
```

5.2.1 Accounting for Compute Unit Asymmetry

By definition, heterogeneous systems consist of processing units with different computational capabilities. In order to exploit the full performance of a system, a run-time is required to account for the asymmetry of the system to increase scheduling efficiency. The compute units with higher processing capabilities, in this case GPUs, are able to execute tasks faster, hence they require higher number of tasks to saturate the processor. Although this is not always the case, we assume the programmer provides the GPU

implementations of tasks where the GPU task outperforms the same task executed on a CPU core.

In order to account for the compute unit asymmetry, we introduce an artificial measure called *compute ratio* that represents the fraction of raw compute power of each compute unit within the entire system which is used to determine how work is distributed when there is not enough work to saturate the machine. This artificial measure ensures that less capable compute units (i.e., CPU cores) do not introduce delays by acquiring more work than their *compute ratio*. For example, let's assume there are two ready tasks in the local queue of the *GPU dedicated core* and all the CPU cores as well as GPU are idle, waiting for a task to execute. In this case, one of the tasks can be offloaded to the GPU immediately and start executing as soon as its data is transferred to the GPU memory. Normally, at this point, the second ready task may be stolen by a CPU worker in order to execute the task since the CPU cores are still idle. However, assuming the computational capability of the GPU exceeding one CPU core, executing the second task also on the GPU might result in a lower overall execution time. Therefore, it is not ideal to steal the task from the local queue of the GPU dedicated core.

Using a compute ratio is similar to schedulers that use profiling information such as HEFT [108] and StarPU [9]. In these approaches the profiling information is used to determine where a task is executed whereas we use the compute ratio only to decrease the idle time of the processing units with higher computational capability.

Furthermore, increasing the amount of work that a more capable compute unit can execute has a positive impact on performance due to Amdahl's Law [55]: the tasks on the critical path need to be executed in priority by the fastest compute units available. The compute ratio is only evaluated once, at library installation on a given system, but in the future could be biased, depending on whether tasks are compute or I/O bound.

5.3 Experimental Setup

The novel scheduling technique presented in this thesis is implemented on top of the extended version of OpenStream [95] run-time. The OpenStream compiler and run-time preserve the task dependence information, as specified by the programmer, and implement implicit buffer management as described in Chapter 3. In order to support GPUs, the run-time and compiler are extended to use the OpenCL [65] programming interface with the execution model described in Section 4.1. The scheduling technique

exploits the asynchronous features implemented in the extension for overlapping computation and data transfers between disjoint memory address spaces.

5.3.1 Hardware Environment

Two systems are used for the experiments named Xeon-K20m and Volta. The first experimental platform, Xeon-K20m, has 12 cores, two sockets with Intel Xeon E5-2620, each with 6 CPU cores running at 2.00 GHz, 32 GiB RAM and runs CentOS 6.8 with kernel 2.6.32-573.3.1.el6.x86_64. Hyper-Threading was disabled in all experiments. The GPU of the system is an NVidia K20m with 2496 GPU cores operating at 706 MHz and 5 GiB of memory. The GPU driver version 361.42 with OpenCL 1.2 support is used. The GPU of the system supports PCI-e version 2.0 with a theoretical bandwidth of 8 GiB/s.

The second system Volta has an AMD A10-7890K CPU with 4 cores running at 4.10 GHz, 16 GiB RAM and runs Ubuntu 16.04.1 with kernel 4.15.0-42-generic. The GPU of the system is an NVidia Titan V with 5120 GPU cores operating at 1455 MHz and 12 GiB of memory. The driver version for this GPU is version 396.37 with OpenCL 1.2 support.

5.3.2 Experimental Baseline

To demonstrate the effectiveness of the novel scheduling technique presented in this thesis, the scheduling strategy implemented by the XKaapi run-time is used as the baseline for comparison. XKaapi schedules tasks dynamically on CPUs and GPUs, using random work-stealing for load balancing with locality-aware optimizations. XKaapi uses CUDA instead of OpenCL for GPU acceleration and the scheduler does not attempt to decrease the total number of data transfers between devices, nor to deal with compute power asymmetry. To exclude bias arising from the difference in GPU programming models and to focus only on the effectiveness of the different scheduling techniques, the baseline and the proposed strategies are implemented in OpenStream run-time.

The heuristic chosen as the baseline is XKaapi's H1 scheduling strategy that uses a locality-aware work-stealing heuristic which iterates over each input dependence of a task and chooses the target device where the largest amount of the input data resides. XKaapi also proposes a second heuristic, H2, which aims to reduce the data replicas created due to the explicit data management in addition to the software cache to keep

track of the data buffers employed in the run-time. Since OpenStream does not have a software cache and uses implicit task-private data buffers which already eliminates data replicas, it is impossible to implement the H2 heuristic as a baseline.

5.3.3 Benchmarks

For the evaluation of the scheduling technique, three benchmarks are chosen reacting sensitively to task and data placement, data transfers and load balancing: matrix multiplication, Cholesky Factorization and Jacobi-1D.

The matrix multiplication benchmark is an OpenStream implementation of tiled matrix multiplication, calculating $C = \alpha A \times B + \beta C$, where A, B and C are square matrices. Multiplication of tiles is carried out by an optimized version the *dgemm* routine from BLAS [17] library.

Cholesky is a linear algebra kernel that calculates the lower triangular matrix L of a dense, symmetric, positive definite matrix A, such that $A = L \times L^T$. The $N \times N$-matrix A is divided into $S^B \times S^B$ sub-matrices. In order to analyze how the approach reacts for different task granularities, this block size is varied throughout the experiments. To calculate the Cholesky Factorization of A, it is necessary to apply different operations to the sub-matrices and to propagate updated values accordingly. Each of the operations is carried out by a highly optimized BLAS [17] and LAPACK [6] functions, namely *dgemm* for the matrix multiplication, *dsyrk* for the symmetric rank k update, *dpotrf* for the block-level Cholesky Factorization and *dtrsm* for solving the remaining part of the equation.

Jacobi-1D is an OpenStream implementation of a Jacobi-style stencil operating on a one-dimensional matrix of double precision floating point elements. This benchmark is particularly interesting for the evaluation of the proposed approach, as it provides a communication-intensive workload, reacting sensitively to data placement. At each iteration of Jacobi-1D, each matrix element is updated by averaging the values from the previous iteration for the elements in its Von Neumann Neighborhood. The benchmark implements spatial tiling by dividing the matrix into blocks. For each block and each iteration, an OpenStream task with three input and three output dependences is generated. The *main input dependence* is on the task's assigned block of input data from the previous iteration and the *main output dependence* is on the generated block of output elements. The remaining *auxiliary dependences* are on single elements at the border of the blocks.

Since the extended OpenStream run-time only requires the kernel implementation,

the GPU kernels are generated using AMD's clBLAS library [5] and both CPU and GPU implementations have the same functionality.

5.4 Results

In the following experimental evaluation, the proposed scheduling technique is compared with the baseline scheduling technique of XKaapi's H1 heuristic implemented in OpenStream. Throughout the results section, *OS* denotes the scheduling technique proposed in this thesis and *XKS* denotes the baseline technique.

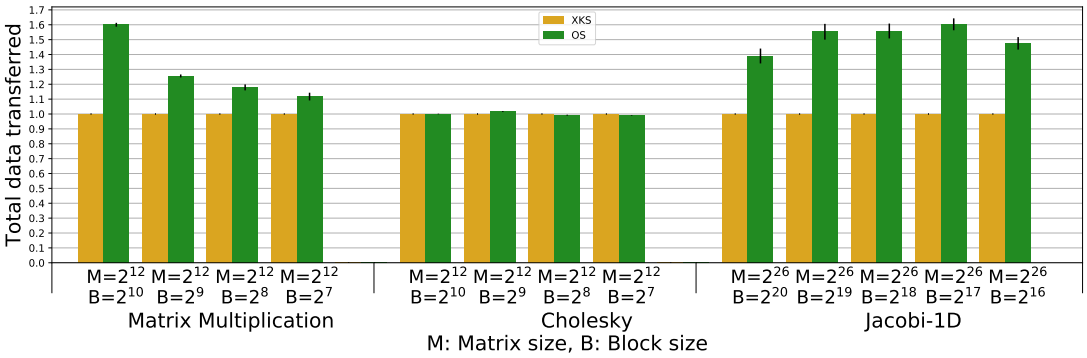
Characterization of both approaches use five metrics: the total number of tasks executed on the GPU, the total amount of data transferred between GPU and host memory in both directions, the number of data transfers, execution time and a breakdown of the time spent in different states showcasing the overlaps of execution and data transfers. All results were obtained from 5 consecutive runs of each configuration.

In all experiments, the block sizes are varied to show the effects of different task granularities. Labels on horizontal axes are of the form $M = 2^n$ $B = 2^m$, where M indicates the number of elements in each dimension of the matrix and B stands for the number of elements per block (e.g., Cholesky's matrix size is $2^n \times 2^n$ and the block size is $2^m \times 2^m$). Since Jacobi-1D operates on a one dimensional array, M and B directly stand for the total number elements and the number of elements in a block, respectively. The number of iterations for Jacobi-1D was set to 60 in all experiments.

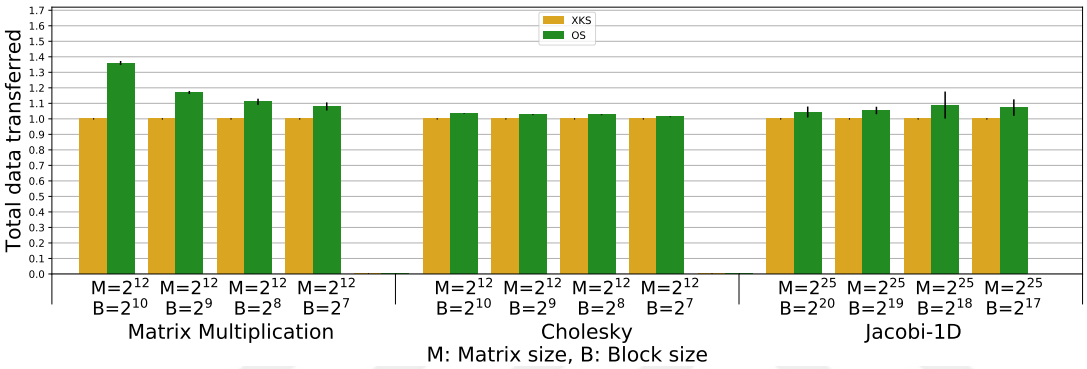
5.4.1 Data Locality: Bandwidth vs. Latency

The amount of data that needs to be transferred between the host and a device is one of the key factors for efficient acceleration. If data transfers cannot be overlapped with execution, this can constitute a substantial overhead on the critical path. Figure 5.1 presents the total amount of data transferred between host and device during the execution of each program configuration, normalized to the average value for XKS. These results show that OS is transferring more data than XKS, especially for Jacobi-1D benchmark, for which the amount of data increases by up to 60%.

An increased amount of data transfers can have a negative impact on performance if the increase is due to decreased memory locality on the device and this results in more time spent waiting for data. However, the amount of data also increases if a higher number of tasks are offloaded to the GPU, and this does not need to incur a



(a) Xeon-K20m System



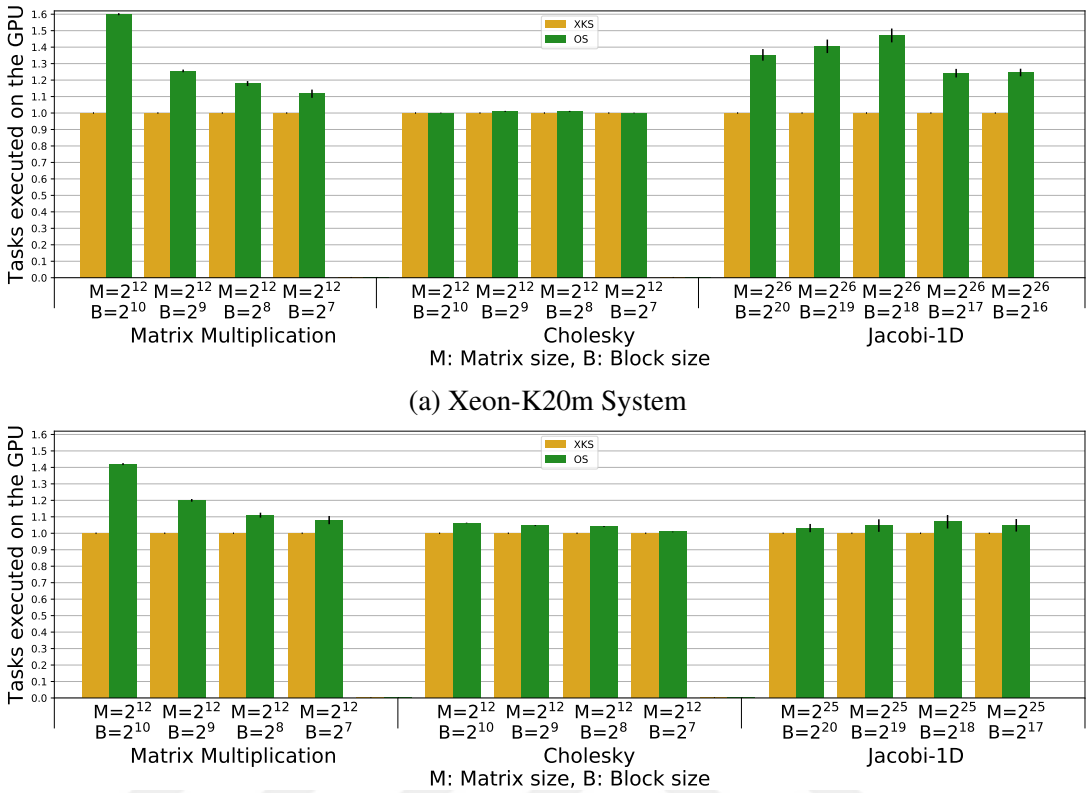
(b) Volta System

Figure 5.1: Total amount of data transferred (normalized to the baseline XKS)

performance penalty if the transfers are done concurrently to execution. As shown in Figure 5.2, OS is indeed offloading significantly more work to the GPU compared to XKS As each task performs a similar amount of work, the number of tasks executed on the GPU provides a good approximation of the amount of work effectively offloaded, and there is a clear relation between the amount of work offloaded and the amount of data transferred.

The difference between the Xeon-K20m system and the Volta system regarding the data transfer sizes and the number of tasks executed on the device is due to the baseline XKS technique performing more tasks on the GPU, thus the gains the OS technique offers is not as substantial. Since the Volta system has only 3 CPU cores for task execution, more tasks compared to the Xeon-K20m system is offloaded to the GPU. Therefore the benefit of the OS scheduling strategy is not as great.

Beyond the ratio of data transferred to offloaded work, the OS scheduling strategy has one key advantage: its objective is not only to increase the number of tasks



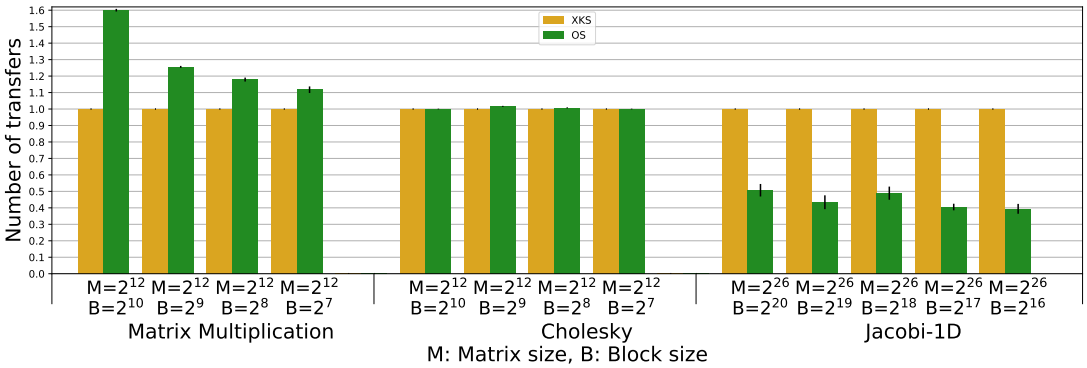
(a) Xeon-K20m System

(b) Volta System

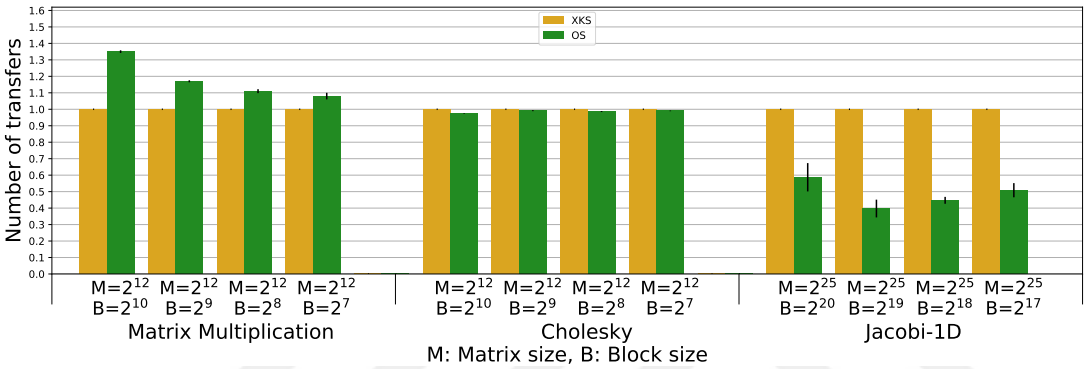
Figure 5.2: Number of tasks executed on the GPU (normalized to the baseline XKS)

executed on the GPU while maintaining the load balanced, but also to minimize synchronization between host and device. Figure 5.3 shows the number of transfers issued, both from host to device and vice-versa. For matrix multiplication and Cholesky, the number of data transfers correlates with the number of tasks executed on the GPU. Since matrix multiplication is embarrassingly parallel, there are no inter-task dependences, resulting in the same ratio of data transfers issued as the ratio between number of tasks offloaded.

For Cholesky, the results for the proposed strategy are similar to the XKS baseline. This is mainly due to the benchmark's inter-task dependences, which are of the same size. The baseline approach uses a locality-aware heuristic and yields similar schedules with only minor differences due to a different load balancing heuristic. One could argue that for a benchmark with static dependences, a static scheduling scheme offloading all tasks to the GPU should perform similarly well. However, such a strategy is inherently limited: it only works for very regular benchmarks and does not account for dynamic behavior at execution time, while the novel approach proposed in this



(a) Xeon-K20m System



(b) Volta System

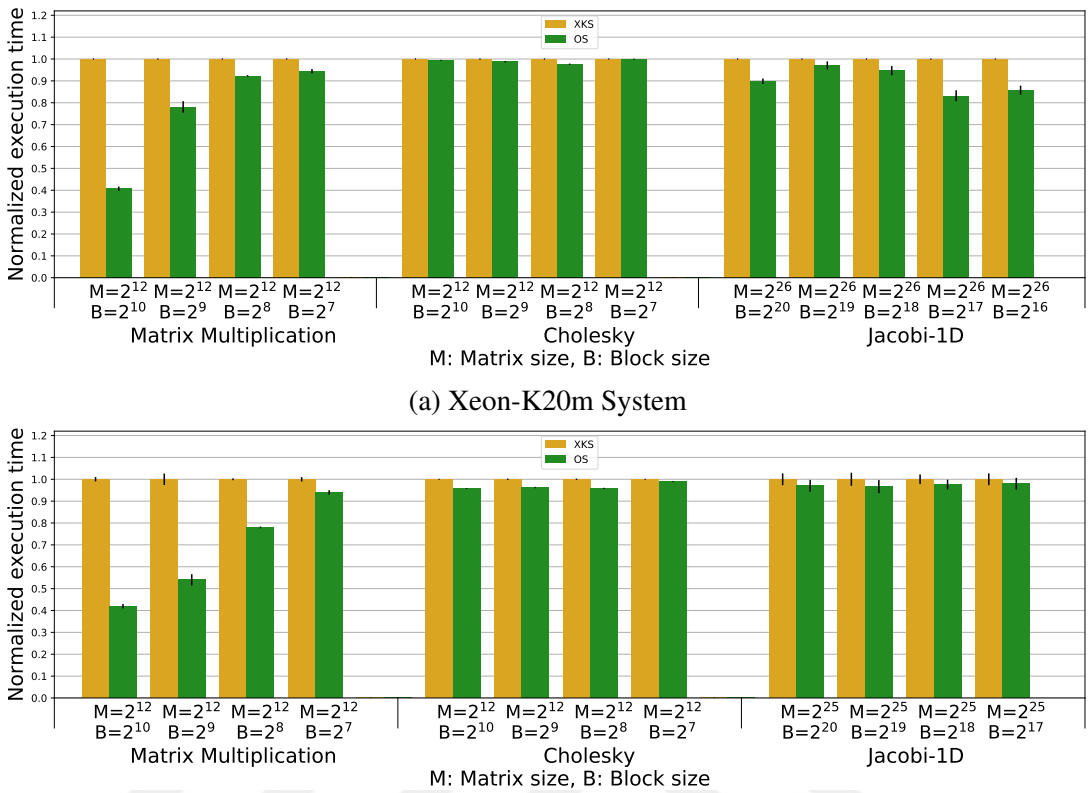
Figure 5.3: Number of transfers between the host and the device (normalized to the baseline XKS)

thesis covers both static and dynamic benchmarks.

Figure 5.3 also shows that, for Jacobi-1D, the number of data transfers issued by OS is up to 60% less than for XKS, despite the fact that OS is executing more tasks on the GPU than XKS. A large portion of the transfers that are avoided are transfers for the many auxiliary dependences present in this benchmark. While the contribution of these small transfers to the total amount of data exchanged between the host and device is negligible, their latency adds up to a substantial delay with a significant impact on performance. OS manages to ensure that all dependences are satisfied entirely within the GPU for a subset of the tasks. This helps not only avoiding such delays, but also reduces the time spent looking for work when the GPU is idle.

5.4.2 Impact on Performance

The reduced GPU idle time, either waiting for work or waiting for data, has a positive overall impact on performance. This is illustrated by Figure 5.4, showing the execution



(a) Xeon-K20m System

(b) Volta System

Figure 5.4: Execution time (lower is better, normalized to the baseline XKS)

time for both scheduling strategies, normalized to the average execution time for XKS. OS is able to achieve higher performance with speedups of up to 1.2× for Jacobi-1D (geometric mean of 1.11×) in Xeon-K20m and 1.03× in the Volta system (geometric mean of 1.027×); up to 2.5× for matrix multiplication (geometric mean of 1.37×) in Xeon-K20m and up to 2.38× in Volta system (geometric mean of 1.56×); up to 1.03× for Cholesky (geometric mean of 1.02×) in Xeon-K20m and up to 1.044× in Volta system (geometric mean of 1.03×).

Given that both implementations offload almost the same number of tasks to the GPU for Cholesky, their performance is—as expected—similar. However, for matrix multiplication, OS strategy improves performance significantly as it is able to offload more tasks to the GPU, especially for larger block size configurations. For larger block sizes, task execution on a CPU core takes a substantial amount of time, which can delay termination if started too late—or at all. The OS scheduler is thus able to balance load according to the respective computational capabilities of each resource.

For Jacobi-1D, the proposed scheduling technique increases the number of tasks

offloaded to the GPU while reducing the number of data transfers, achieving a significant performance increase on an I/O bound kernel. The most beneficial part of the proposed scheduling technique in the Jacobi-1D case is the elimination of the small auxiliary dependences which in return eliminates the task stalls due to the smaller dependences, increasing the number of tasks executed on the GPU. The benefit is less clear in the Volta system due to the smaller number of CPU cores available for task execution which already increases the number of tasks offloaded to the GPU in the baseline XKS heuristic, thus the OS scheduler can increase the performance slightly.

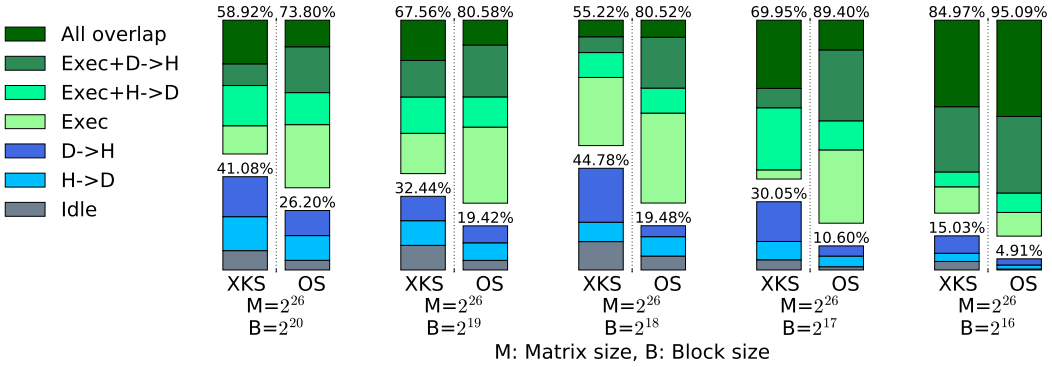
5.4.3 Execution Breakdown

To provide a better understanding of the impact of the technique presented in this thesis, Figure 5.5a shows a breakdown of GPU execution for each benchmark, indicating the relative amount of time the GPU spent in each of seven possible states. The states are: *Idle* (the GPU is neither executing a task nor transferring any data), *Exec* (execution without overlapping data transfers), *D→H* (data transfer from the device to the host without execution), *H→D* (data transfer from the host to the device without execution), *Exec+D→H* (execution while transferring data from the device to the host), *Exec+H→D* (execution while transferring data from the host to the device), and *All overlap* (execution while transferring data in both directions).

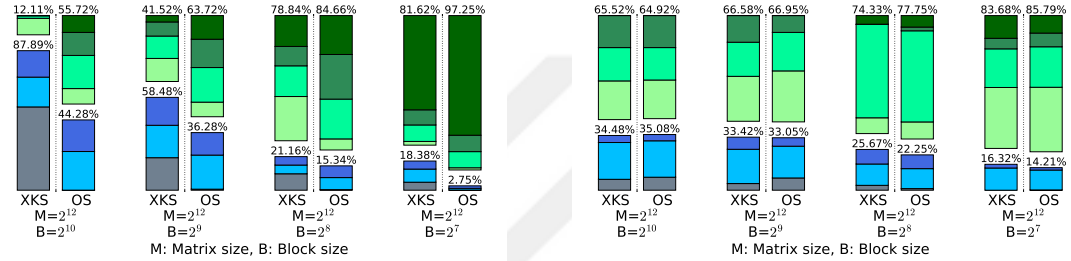
The upper part of each bar in Figure 5.5a is composed of all states in which the GPU is executing a task (*Exec*, *Exec+D→H*, *Exec+H→D*, *All overlap*), while the lower part of each bar shows the amount of time spent in states with inefficient use of the GPU, in which no task is executed (*Idle*, *D→H*, *H→D*).

For each problem and block size, the time spent on kernel execution on the GPU is significantly higher for OS compared to XKS, as indicated by the larger upper part of the bars and the corresponding ratios. This is a consequence of the proposed scheduling strategy, which selects follow-up GPU tasks from the local queues to reduce synchronization between devices, in addition to the load balancing mechanism that keeps the load balanced and GPU occupied. However, in Cholesky on the Xeon-K20m system there is no significant difference since both techniques lead to similar schedules while on the Volta system, the difference is larger as the corresponding performance benefits as shown in Section 5.4.2.

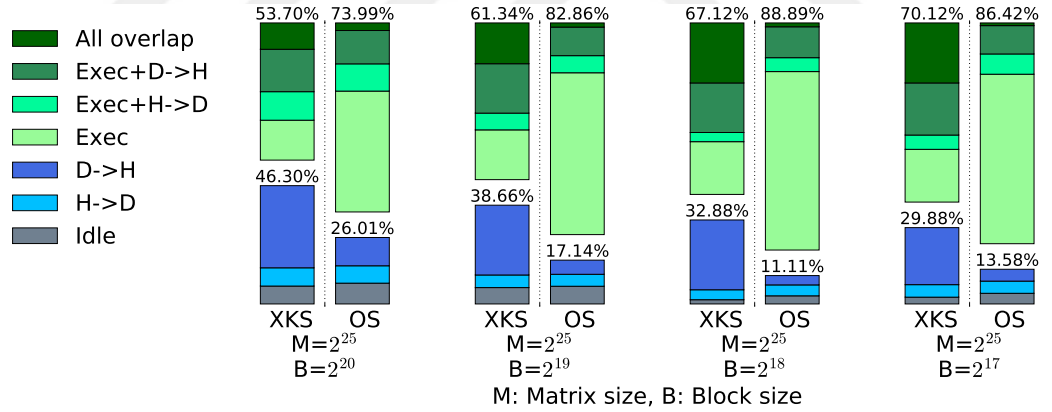
In Jacobi-1D, the time spent in idle states, where no task execution occurs, is minimal. For the larger block sizes, the idle time is mostly spent on data transfers. However,



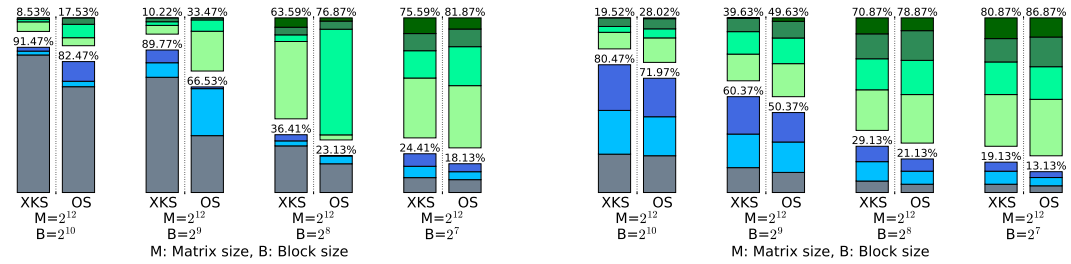
(a) Breakdown for Jacobi-1D of time spent on Xeon-K20m system **Exec**uting or communicating host-to-device **H**→**D** or device-to-host **D**→**H**.



(b) Breakdown of time spent for GPU execution on Xeon-K20m system for Matrix multiplication (left) and Cholesky (right).



(c) Breakdown for Jacobi-1D on Volta system.



(d) Breakdown of time spent for GPU execution on Volta system for Matrix multiplication (left) and Cholesky (right).

Figure 5.5: Breakdown of time spent in GPU execution, showing the amount of overlap between computation and communication

for smaller block sizes, the OS scheduler is able to overlap more transfers with execution, as the time it takes to transfer the data to each direction decreases. Although this is also the case for the XKS strategy, OS is able to decrease the idle time by decreasing task stalls and task acquisition overhead.

On the other hand, in matrix multiplication, there are no inter-task dependences since it is an embarrassingly parallel workload. For this case, the load balancing mechanism succeeds in dynamically balancing the load between CPU and GPU, decreasing the time spent in the *Idle* state to under 1% on the Xeon-K20m system while on the Volta system the idle time is caused by the tasks executed on the CPU cores which take a lot longer compared to the GPU. However, the idle time for the OS strategy is still significantly smaller than the XKS strategy since OS strategy executes more tasks on the GPU. The time for data transfers is significant, but cannot be avoided unless a smaller block size is used, leading to a finer-grained scheduling. In all cases, OS is achieving a better overlap of computation and communication than XKS.

5.4.4 Comparison with XKaapi Run-time

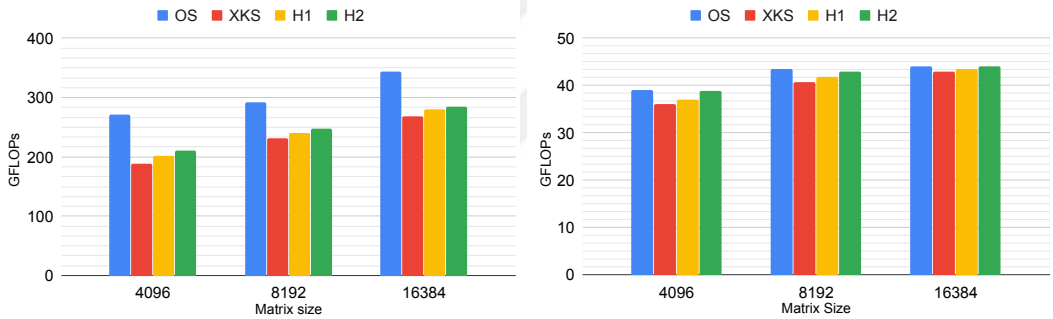
In order to provide a better understanding of how the proposed scheduling heuristic performs, we also present a comparative analysis to the XKaapi run-time using H1 and H2 heuristics for matrix multiplication and cholesky factorization benchmarks. Table 5.1 shows the execution times of all heuristics where OS denotes the proposed scheduling heuristic in this thesis while XKS denotes the implementation of H1 heuristic using OpenStream run-time. On the other hand H1 and H2 heuristics are the original implementations of XKaapi run-time. We have used three matrix sizes for both benchmarks as 4096x4096, 8192x8192 and 16384x16384 to show how heuristics perform using larger matrices while block size is statically selected as 1024. The experiments are conducted in Xeon-K20m computer and 11 CPU cores as well as 1 GPU are used as execution units while 1 CPU core is reserved for the handling of GPU operations for both OpenStream and XKaapi.

The execution times show that the XKS implementation performs slightly worse than XKaapi's H1 implementation. Although two heuristics are semantically the same, the difference between execution times are caused by different run-times as well as the difference between OpenStream using OpenCL as the GPU language while XKaapi uses CUDA. Between the XKaapi heuristics, H2 heuristic performs similarly with H1 heuristic using 1 GPU which is consistent with the evaluation of the XKaapi paper [47]. On the other hand, the heuristic proposed in this thesis denoted as OS performs better

Table 5.1: Execution times of OpenStream and XKaapi run-times for different matrix sizes for Matrix Multiplication and Cholesky

	Matrix Multiplication			Cholesky		
	4096	8192	16384	4096	8192	16384
OS	0.41s	3.76s	25.52s	0.59s	4.21s	33.37s
XKS	0.73s	4.75s	32.67s	0.63s	4.51s	34.01s
H1	0.68s	4.58s	31.41s	0.61s	4.38s	33.76s
H2	0.65s	4.42s	30.88s	0.59s	4.27s	33.42s

for matrix multiplication benchmark while Cholesky benchmark results are similar. This is due to the fact that OS heuristic can offload more tasks to the GPU for matrix multiplication as discussed in Section 5.4.1.



(a) Matrix Multiplication (b) Cholesky

Figure 5.6: Performance in GFLOPs

In addition to the execution times, Figure 5.6 shows the execution performance in GFLOPs for all heuristics of OpenStream and XKaapi. OS heuristic performs better in all cases. In addition to this, the difference between the H1 heuristic and H2 heuristic of XKaapi run-time is minimal. The main reason for both heuristics performing similarly is, in single GPU platforms the data management for XKaapi does not take advantage of the reduction of data invalidations, thus leading to similar results. Overall, our experiments show that the our proposed heuristic is not only better than the state-of-the-art heuristics in run-times where decentralized memory management is employed, but also can compete with optimized heuristics such as H2 heuristic that are employed for run-times with centralized memory management such as XKaapi.

5.5 Conclusion

In this chapter, we presented a new scheduling technique for load-balancing data-flow tasks on heterogeneous systems, accounting for asymmetric compute capabilities, while simultaneously increasing on-device data reuse and decreasing synchronization between host and accelerators. We showed that our technique improves on-device data reuse, minimizing inter-task communication across devices, and improves the overlapping of task execution with inter-device communication, effectively hiding the cost of communication between host and device memory.

We used OpenStream run-time for the implementation of our strategy and exploited OpenStream-specific task-private buffers for the traversal of the task graph to provide efficient task and data placement for multi-core CPUs and discrete GPUs. Additionally, the proposed technique employs an artificial measure called compute ratio in order to account for the asymmetric compute capabilities of different devices to ensure the tasks on the critical path are executed by the fastest compute units.

The experimental evaluation shows that our approach effectively reduces the number of transfers required, reduces synchronization between CPU and GPU, increases the overlap of computation and communication, reduces GPU idle time, and increases the number of tasks offloaded to the GPU. Our technique transparently places data and tasks on the host and accelerators without additional annotations by the programmer—all task and data placement decisions are based on inter-task dependence information readily available in modern task-parallel, data-flow run-time systems. We compared our approach to the H1 dynamic scheduling heuristic of the XKaapi run-time on two systems, showing a substantial performance improvement of up to 2.5× for matrix multiplication, 1.2× for Jacobi-1D and 1.03× for Cholesky in Xeon-K20m system and 2.38× for matrix multiplication, 1.03× for Jacobi-1D and 1.04× for Cholesky in Volta system compared to the XKS scheduling heuristic baseline.

Although GPUs are widely used in heterogeneous systems for high performance computing, FPGAs are also becoming mainstream. Therefore, in the next chapter we present a novel scheduling strategy for data-flow task parallel programs targeting low-power CPU-FPGA system-on-chips. In particular, we show how the OpenStream-specific features can be used for efficient task and data placement in such systems, in addition to providing an infrastructure for dynamically scheduling data-flow tasks onto multi-core CPUs and FPGA accelerators.

Chapter 6

Dynamic Task Scheduling on FPGA-SoCs

In the previous chapter, we introduced a novel scheduling heuristic for efficiently scheduling task-parallel programs on heterogeneous systems that consist of multi-core CPUs and discrete GPUs by using the additional data-flow information provided by the OpenStream run-time. The use of task-private buffers in OpenStream enable the dynamic traversal of the task graph in order to make better task and data placement decisions. In addition to this, the proposed scheduling heuristic also uses a threshold value that is calculated according to the differences in the computational capabilities of each device in the system, ensuring the effective use of higher-throughput devices.

In this chapter, we introduce a novel scheduling strategy for heterogeneous systems that incorporate multi-core CPUs and FPGAs on the same chip. There are two main reasons for a different scheduling strategy is required for CPU-FPGA systems; (1) the target architecture is an SoC rather than a discrete device system and (2) the architectural and programming model differences between GPUs and FPGAs. Firstly, in an SoC system both the multi-core CPU and the FPGA share the system memory, where in discrete systems it is a necessity to move data between devices through PCI-e bus. Secondly, FPGAs can contain multiple accelerators that can execute different tasks simultaneously while in the GPU case, multiple kernels can only be executed concurrently, not simultaneously [113, 50, 85].

6.1 Dynamic Scheduling on FPGAs

The aim of the proposed scheduling technique for FPGAs is twofold: (1) to dynamically select tasks to be offloaded to the FPGA accelerators while keeping the load balanced between different accelerators and CPU execution cores; and (2) to provide an asynchronous execution infrastructure for FPGA accelerators. To our knowledge, there has not been any effort in the literature for dynamic task scheduling on FPGA accelerators using a user-level run-time system. The only close approach is OmpSs@Zynq [41] in which the main focus of the study is to generate FPGA accelerators during the compilation phase, combined with a static scheduling heuristic, not a dynamic scheduling approach while we propose a dynamic scheduling technique for heterogeneous systems containing CPU-FPGA on the same chip.

Existing high-level synthesis (HLS) tools are successful in providing efficient accelerators. However, the performance benefits can be increased in case the advanced opportunities provided by FPGAs such as pipelined execution are exploited [40]. Although the FPGA programming model proposed in this thesis is not able to fully exploit the pipelined execution, in case of multiple accelerators present on the FPGA, a software pipelining approach [96, 28, 4] is still a useful technique for increasing the efficiency of the schedule by incorporating the dependence information when making scheduling decisions.

Reconfigurable architectures such as FPGAs excel in performance efficiency when fine-grained pipelining is employed in the accelerator design. However, due to our programming model which only considers using the accelerators, rather than generating the accelerators, achieving a fine-grained pipelining is not possible. In a fine-grained pipelined accelerators, every output data region can be fed into the dependent accelerator for further operations with every clock cycle, but requires extensive design efforts as well as expert knowledge of accelerator design.

In our model, we assume the accelerators are stand-alone blocks and the data dependences between the accelerators are written or read through the system memory. Moreover, creating a coarse-grained pipelined execution of dependent tasks require dynamic management of tasks and dependences. OpenStream provides dynamic traversal of the task graph, allowing dependent tasks to be scheduled on the FPGA accelerators dynamically. Therefore, in this study, we exploit the data-flow information on task dependences in OpenStream as well as the ability to traverse the task graph in order to create pipelined execution on FPGA accelerators.

6.1.1 Scheduling Tasks on FPGA Accelerators

The aim of the proposed scheduling technique is is twofold: (1) to keep the FPGA accelerators occupied with task execution to increase performance; and (2) schedule tasks on the FPGA accelerators in a pipelined manner when possible by following a dependence-aware heuristic. While the first aim is to increase the effective use of the FPGA accelerators while with the second aim not only trying to create pipelined execution on FPGA accelerators, but also increasing data reuse in the FPGA address space. The proposed scheduler uses dependence-aware heuristic instead of a locality-aware approach because, although locality-aware approaches increase cache reuse on CPU cores and can increase performance on homogeneous systems, the performance benefits can be better realized executing tasks on a more powerful accelerator rather than the tasks being executed on a less powerful device with increased data reuse. In addition to that, we aim to take advantage of the pipelining ability of FPGA accelerators in a coarse-grained manner by task pipelining. Although FPGA accelerators can be designed to take advantage of more fine-grained pipelining, optimized accelerator design is outside the scope of this thesis.

To keep the accelerators occupied, the proposed scheduler uses a structure called *request_mode* that has as many variables as there are different types of accelerators to determine when a type of accelerator on the FPGA is idle. When an accelerator becomes idle and there are no ready tasks available in its corresponding accelerator buffer, our scheduler attempts to push a task to the local work queue of the FPGA dedicated core which then is offloaded to the accelerator by prioritizing the accelerators instead of executing the task on the CPU. The aim of this part of the scheduling strategy is to increase the effective use of the accelerators, enabling higher number of task execution on the accelerators and increasing performance gains in return.

In addition to increasing the effective use of FPGA accelerators, the scheduler also tries to execute dependent tasks on the accelerators, creating a pipelined execution of tasks. When a task executes on an accelerator, the consumers of the task are traversed. The traversal is possible since OpenStream run-time uses task-private buffers. For each consumer, two conditions are checked: (1) if it has FPGA clauses to offload the task to an accelerator; and (2) if the executing producer is the last remaining input dependence. When these conditions are met, the consumer task is pushed to the bottom of the corresponding accelerator buffer upon completion of the producer task. Finally, when an accelerator of the same type as the consumer task becomes available, the consumer is offloaded to the accelerator for execution.

Creating pipelined execution using dependent tasks increases the efficiency by reusing the data pointers that reside in the FPGA address space, decreasing the overhead of memory movement.

6.1.2 Task Execution on FPGA Accelerators

Similar to the GPU extension of OpenStream, in the extended FPGA version, only the tasks that have the special FPGA clauses that are detailed in Section 4.2.2 can be executed on the accelerators. However, different from the GPU case, the management of the FPGA accelerators is the responsibility of the FPGA dedicated core instead of a combined effort of GPU dedicated core and the OpenCL driver which is able to handle the execution on the GPU. For example, the GPU dedicated core is able to traverse the task graph to find tasks that have all of the input data dependences on the GPU and make a decision to offload the task by enqueuing them on the OpenCL command queue where the tasks are in FIFO order. This decision allows tasks to be scheduled preemptively and results in reduction of task offload overhead. However, the absence of such driver for the FPGA devices puts more responsibility on the FPGA dedicated core for task scheduling.

Essentially, the FPGA dedicated core has three responsibilities for task execution: (1) management of the accelerator buffers; (2) distribution of tasks from the local work queue to the accelerators and in case the local work queue is empty, retrieval of tasks using random work stealing; and (3) the management of the accelerators, checking the availability of the accelerators of multiple types and offloading tasks to the available accelerators.

The accelerator buffers are the data structures used to orchestrate the efficient mapping of data-flow tasks to the FPGA accelerators. The first responsibility of the FPGA dedicated core is to manage the accelerator buffers by assigning tasks from the local work queue and to offload tasks to the available accelerators that reside in the buffers. The FPGA dedicated core constantly checks all the accelerator buffers to see if the buffers have any room for a task. In this case, the task at the bottom of the local work queue is checked to determine its accelerator type to assign the task to the corresponding accelerator buffer. The tasks are distributed from the local work queue to the accelerator buffers only if the corresponding buffer is not full.

On the other hand, if the local work queue is empty, the FPGA dedicated core obtains a task using random work stealing. The restriction in work stealing in our scheduler is that only the tasks with FPGA implementations can be stolen to avoid

obtaining a task that the FPGA accelerators cannot execute. Once a task is stolen and pushed to the local work queue, the FPGA dedicated core attempts to assign the task to the corresponding accelerator buffer. The tasks are pushed to the local work queue in two ways; by work stealing from a random victim, or if a task finishes its execution on an FPGA accelerator and satisfies the last remaining data dependence to its consumer, the consumer task is pushed to the consumer task's accelerator buffer and if the buffer is full, the task at the top of the buffer is pushed to the local work queue. If the task is obtained through work stealing, it is pushed to the top of the queue while if the task is pushed due to dependence satisfaction, it is pushed to the bottom of the queue making it the first task to be offloaded to the accelerator in order to increase the locality.

The third duty of the FPGA dedicated core is the management of the accelerators by polling the control bits of each accelerator to determine the state of the accelerators on the FPGA. Whenever an accelerator becomes *IDLE*, the FPGA dedicated core tries to obtain a task from the corresponding accelerator buffer to offload the task to the available accelerator. If a task is found in the buffer, the task is offloaded to the available accelerator. The execution of tasks on the FPGA buffers is done asynchronously by the FPGA dedicated core. Once a task is offloaded to an accelerator, the FPGA dedicated core does not wait for its execution to finish, but uses polling to determine which accelerators are idle to offload remaining ready tasks.

Algorithm 3 summarizes the algorithm executed by the FPGA dedicated core. The first loop iterates all the accelerator buffers and checks if there are ready tasks waiting to be executed in the buffers. If there is a ready task in the accelerator buffer, the accelerator availability is checked, meaning the accelerator state is *IDLE*. In this case a function call is made to select the available accelerator followed by the retrieval of the task from the buffer. The *obtain_task_from_buffer* function requires the type of the task to obtain a task from its accelerator buffer. Once the task is obtained from the buffer, this function checks the local work queue to see if the task at the bottom of the queue has the same type as the buffer. In case it is a match, the task is dequeued from the local work queue and put to the buffer to keep the accelerator buffers full. The scheduler then offloads the task to the available accelerator using the index of the accelerator returned by the *select_available_accelerator* function.

On the other hand, if there are no ready tasks in the accelerator buffer, the availability of the accelerators is checked, similar to the previous case. However, in this case, the accelerator is waiting idly for a task. In order to decrease the idle time of the

Algorithm 3 Execution loop of FPGA dedicated core

```
1:  for all t ∈ Types do
2:      if buffer_has_task(t) then
3:          if is_accelerator_available(t) then
4:              index ← select_available_accelerator(t)
5:              task ← obtain_task_from_buffer(t)
6:              execute_task_on_accel(task, index)
7:          end if
8:      else
9:          if is_accelerator_available(t) then
10:             request_task(t)
11:         end if
12:     end if
13: end for
14:
15: if type ← work_queue_has_task() then
16:     if is_buffer_available(type) then
17:         T ← obtain_work_local()
18:         push_task_to_buffer(T)
19:     end if
20: else
21:     obtain_work_steal()
22: end if
```

accelerator as well as increase the effective use of the accelerator, *request_task* function is called. This function updates the *request_mode* variable of the corresponding accelerator type to notify the CPU workers that an accelerator of type t is idle. The *request_mode* variable is updated using atomic *compare_and_swap* operation to avoid possible deadlocks. During task execution on CPU workers, the type of each ready task is identified and the *request_mode* of that type of accelerator is checked if the task should be scheduled to the CPU worker or it can be offloaded to the accelerator. If the accelerator is in *request_mode*, the CPU worker uses work-pushing to push the task to the FPGA dedicated core followed by an update to the *request_mode* variable, setting it to 0. The work-pushing pushes the ready task to the bottom of the local queue of the FPGA dedicated core, which then can be scheduled to the idle accelerator.

In case the condition is not met, this means either there are no available tasks in the buffer, or all the accelerators on the FPGA are busy, executing tasks. Rather than waiting idly, the FPGA dedicated core handles the task distribution to the accelerator buffers to avoid local work queue becoming empty. For this, it first checks if the local

work queue has any tasks by calling *work_queue_has_task* function. This function returns the type of the task if the local work queue has any task. The type information is then used to check if the accelerator buffer has any room to put the task that resides at the bottom of the local work queue. This step is followed by the retrieval of the task from the local work queue and put at the top of the corresponding accelerator buffer if the condition is met. In case the work queue is empty, a task is obtained through work stealing from a random victim using *obtain_work_steal* function.

6.2 Experimental Setup

The novel scheduling technique for heterogeneous systems that incorporate FPGA accelerators is implemented on top of the extended version of OpenStream [95] run-time. In order to support FPGA accelerators, the run-time and compiler are extended to use the execution model described in Section 4.2.

6.2.1 Hardware Environment

For the experimental evaluation, Xilinx Zynq UltraScale+ MPSoC platform has been used, featuring 64-bit quad-core ARM Cortex-A53 CPU cores running at 1.20 GHz and XCZU9EG-FFVC900-2I-ES1 FPGA on the same chip. The system incorporates 4 GiB RAM and runs Petalinux with kernel version 4.14.0-xilinx-v2018.2.

6.2.2 Benchmarks

For the evaluation of the FPGA scheduling technique, two benchmarks are chosen and evaluated using differing number of tasks, task granularities and number of accelerators: Cholesky factorization and matrix multiplication. Cholesky factorization and matrix multiplication workloads are similar to the ones used in the GPU experiments explained in Section5.3.3. The CPU versions of the tasks are identical while the GPU clauses are replaced with corresponding FPGA clauses for using FPGA accelerators. For the experiment on FPGAs, the benchmark implementations use single-precision floating point elements.

Moreover, for accelerating matrix multiplication, namely *gemm* function, we have used the implementation from the Spector benchmark suite [45]. The benchmarks in the Spector suite contains OpenCL implementations that can be used with Xilinx HLS

tools in order to generate the accelerator designs. Although the implementations originally target Altera based FPGAs, we were able to reuse the implementations in Xilinx tools without any modification. For the *trsm* function in Cholesky, the implementation from the FBLAS library [37] is used which is also designed for Altera FPGAs. The implementation of *syrk* and *gemm* functions in FBLAS library use some Intel-Altera specific functions which do not exist in our Xilinx based target system, thus we were not able to use these implementations in our experiments.

Furthermore, the scope of this study does not include well tuned accelerator implementations, hence we have used publicly available implementations of the accelerators that are implemented in OpenCL. The disadvantage of such choice is that the accelerator performance is limited, but can be improved with better tuned implementations.

6.3 Results

In the following experimental evaluation, the proposed scheduling technique is compared with the baseline where only all the CPU cores on the system are used for execution of tasks. Characterization of our approach is done by varying the number of accelerators on the FPGA and we measure the execution times as well as the total number of tasks executed on the accelerators in the evaluation. All results were obtained from 5 consecutive runs of each configuration.

In all experiments, the matrix size of 1024×1024 is used and the block sizes are varied to show the effects of different task granularities. Changing the task granularity affects the size of each accelerator, thus in smaller task granularities it is possible to use larger number of accelerators on the FPGA.

6.3.1 Task Distribution Analysis

Our approach aims to keep FPGA accelerators effectively utilized by increasing the amount of work that can be offloaded to the accelerators. Figure 6.1 shows the percentage of tasks that are offloaded to the accelerators in different number of accelerator configurations for matrix multiplication benchmark. As shown in the figure, even with using 1 accelerator, more than half of the tasks can be offloaded to the accelerator taking advantage of the higher throughput of the accelerators. Using different block size for execution has a small effect on the percentage of the tasks offloaded to the accelerators, since with the decreasing block size, the amount of tasks in the application

increases.

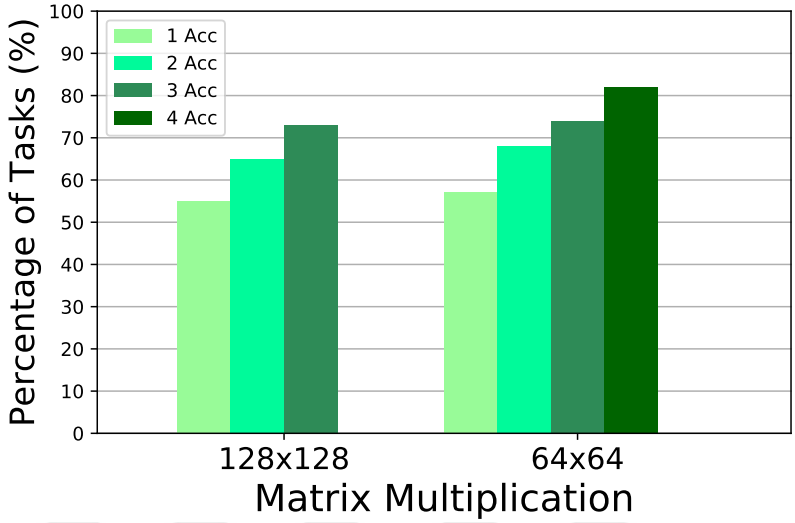


Figure 6.1: Percentage of tasks offloaded to accelerators in Matrix Multiplication

Using the proposed scheduling strategy the percentage of the tasks offloaded to accelerators are; 55% and 57% using 1 accelerator, 65% and 68% using 2 accelerators, 73% and 74% using 3 accelerators with block sizes of 128×128 and 64×64 respectively. Using 4 accelerators with block size 64×64 results in offloading 82% of the tasks to the accelerators.

In Cholesky benchmark, there are multiple types of accelerators whereas in matrix multiplication there is only one. Therefore throughout the rest of the results section, we show different configurations of accelerator types. Table 6.1 shows different accelerator configurations. We used different configurations in order to evaluate the effect of accelerator buffers. For 128×128 block size, configuration 1 includes one of each accelerator type while for configuration 2, we have used one additional *gemm* accelerator due to the higher number of *gemm* tasks available in Cholesky. Using larger block sizes create larger accelerators and the resources on the FPGA only allowed four accelerators for 128×128 block size to be programmed. However, in 64×64 block size case, the accelerators are smaller and we were able to fit more accelerators, allowing more configurations. Note that, these configurations are not tuned for performance nor the accelerator implementations. Moreover, the adjustment of different configurations of accelerators using partial reconfiguration is an interesting use case [110] and our run-time infrastructure can be used as a complementary tool to pursue such studies.

In the first configuration for each block size, the same number of accelerators for different type of tasks has been used. In configuration 1, for 128×128 block size,

Table 6.1: Different configurations of accelerators for different block sizes in Cholesky

	Configuration 1	Configuration 2	Configuration 3
128x128	1 gemm 1 syrk 1 trsm	2 gemm 1 syrk 1 trsm	N/A
64x64	1 gemm 1 syrk 1 trsm	2 gemm 2 syrk 2 trsm	3 gemm 2 syrk 2 trsm

we were able to fit only 1 accelerators of each type, while for 64×64 block size, 2 accelerators of each type is used. However, in configuration 2, we increase the number of *gemm* accelerators by one due to the higher number of *gemm* tasks present in the Cholesky benchmark.

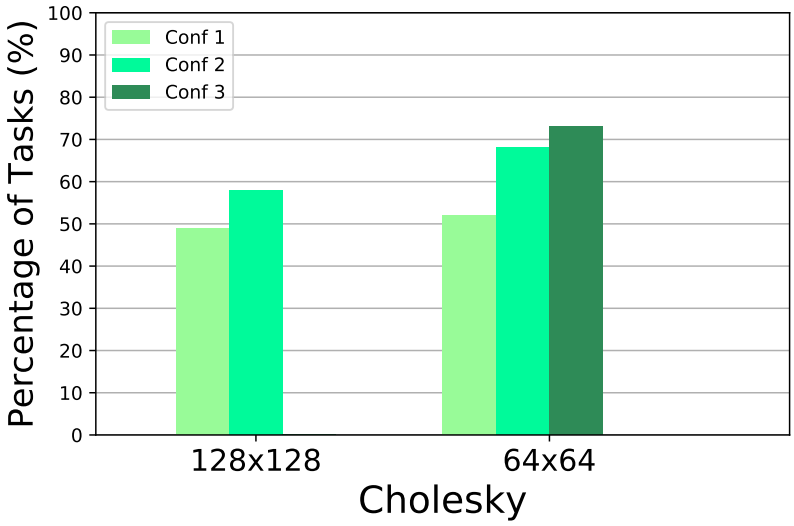


Figure 6.2: Percentage of tasks offloaded to accelerators in Cholesky

In addition to the amount of tasks offloaded to the accelerators we also measure how the tasks obtained for FPGA execution. In our approach, there are three ways that the FPGA dedicated core can offload tasks to the accelerators. Firstly, a task can be obtained through work pushing from another CPU worker. Work pushing is only available when an accelerator is idle and another CPU worker has a ready task on its local queue as the same type in which the accelerators are prioritized due to the assumption of having better performance. Secondly, the FPGA dedicated core can assign a consumer work where it satisfies the last input dependence by executing the last producer

task on an accelerator. In this case, the consumer task is assigned to the corresponding accelerator buffer for execution, creating a pipelined execution of tasks on FPGA accelerators. The third way is through work stealing. When there are no available tasks in any of the accelerator buffers an the local queue of the FPGA dedicated core, work stealing is used to find work to offload to the accelerators. Figure 6.3 shows the percentage of methods used for obtaining work for execution on the FPGA for both benchmarks using different configurations and number of accelerators in addition to varying the block size of the tasks.

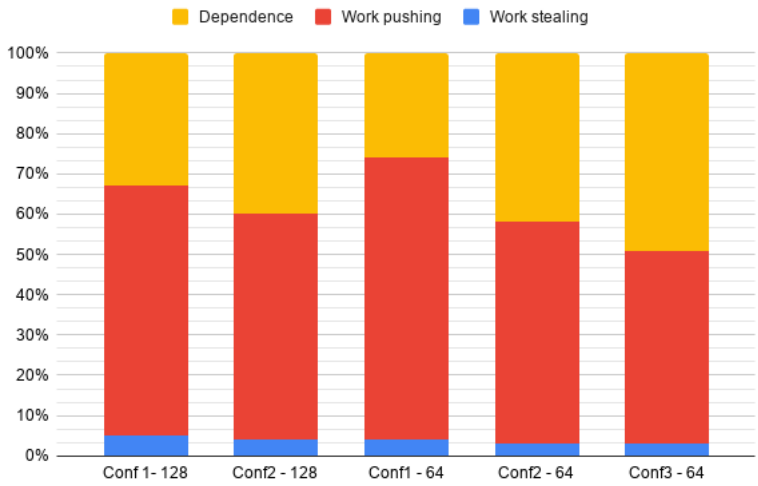


Figure 6.3: Percentage of tasks obtained through work stealing, work pushing or dependence satisfaction in Cholesky

In all configurations, the percentage of tasks obtained through work stealing is smaller than 3% which indicates our scheduler is actively keeping the FPGA accelerators occupied by pushing tasks to the FPGA dedicated core. Moreover, as the number of accelerators increase, the proposed scheduler can schedule more tasks in a pipelined manner, following the task dependences of the tasks executed on the FPGA and offloading the consumers to the FPGA accelerators as well.

For block size of 128×128, the percentage of tasks offloaded to the FPGA using work pushing is 62.44% and 57.73% on average, for configuration 1 and configuration 2 respectively. On the other hand, the percentage of tasks scheduled through dependence satisfaction reaches 34.06% and 39.77%. For block size of 64×64, the percentage of tasks offloaded to the FPGA using work pushing is 71.33%, 56.15% and 48.31% on average, for configuration 1, 2 and 3 respectively while the percentage of tasks for dependence is 27.3%, 42.6% and 50.44%.

Using smaller block sizes enable larger number of accelerators to be programmed on the FPGA which can take advantage of the pipelined behavior of the proposed scheduling approach. Furthermore, the proposed scheduling approach can offload larger number of tasks to the FPGA accelerators, increases the occupancy of the accelerators by actively scheduling tasks and in return provides performance gains.

6.3.2 Impact on Performance

Increased number of tasks executing on the FPGA accelerators has a positive overall impact on performance. This is illustrated by Figure 6.4, showing the speedup of matrix multiplication benchmark using variable number of accelerators, normalized to the average execution time of the CPU-only version of OpenStream. The proposed scheduling strategy is able to achieve higher performance with speedups of 1.38× and 1.3× using 1 accelerator, 1.72× and 1.79× using 2 accelerators, 2.25× and 2.15× using 3 accelerators with block sizes of 128×128 and 64×64 respectively. Using block size of 128×128 creates larger accelerators, we were only able to fit maximum of three accelerators due to the BRAM constraints of the target FPGA. On the other hand, using block size 64×64, the number of accelerators that can be used on the FPGA fabric increase due to the smaller space requirements. Using 4 accelerators with block size of 64×64 can achieve 2.76× over the baseline CPU-only execution.

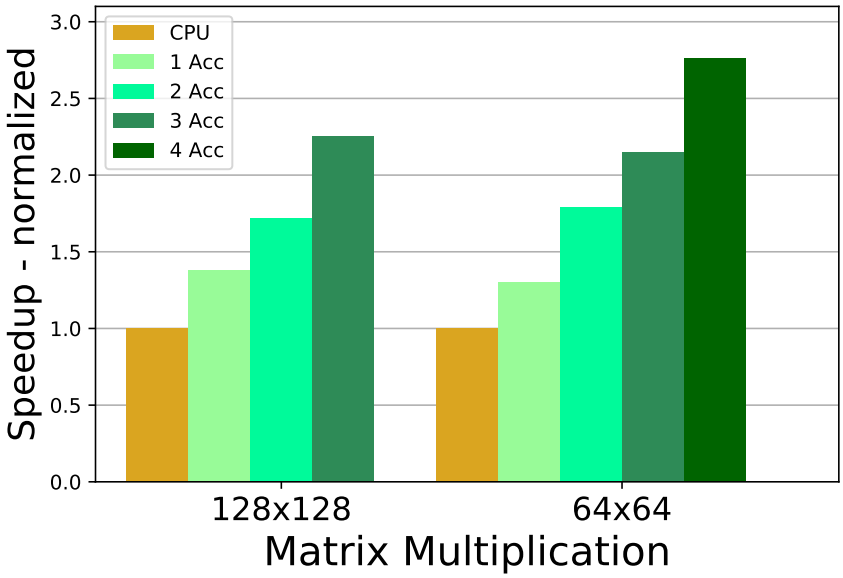


Figure 6.4: Performance of Matrix Multiplication (normalized to the baseline CPU-only)

Increased number of tasks also improve the performance in Cholesky benchmark. This is illustrated in Figure 6.5, showing the speedup of Cholesky benchmark using variable number of accelerators, normalized to the average execution time of the CPU-only version of OpenStream. The proposed scheduling strategy is able to achieve speedups of 1.86× and 1.74× using configuration 1, 2.38× and 2.25× using configuration 2 with block sizes of 128 × 128 and 64 × 64 respectively. The third configuration is only available in block size of 64 × 64 which can achieve 2.66× over the baseline.

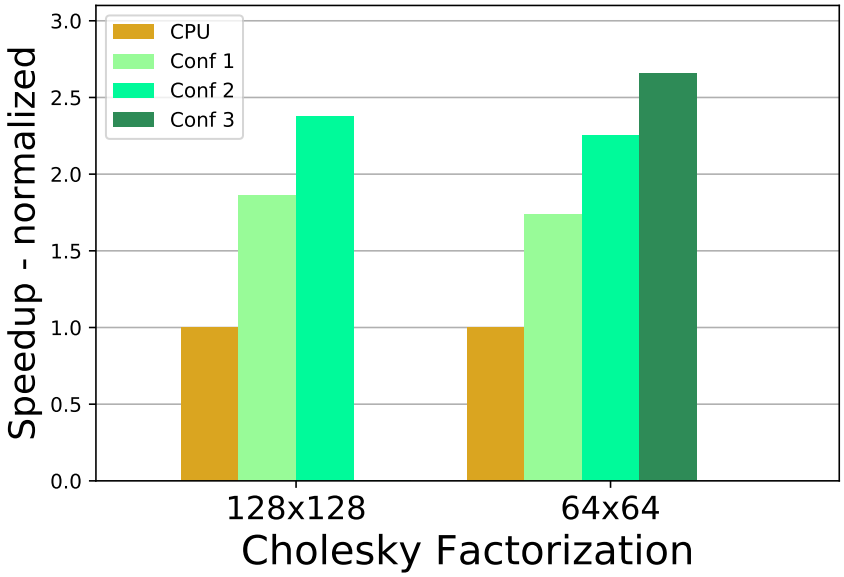


Figure 6.5: Performance of Cholesky (normalized to the baseline CPU-only)

Using different block sizes also has an effect to performance gains. In Cholesky, while the block size of 128 can achieve up to 1.86× speedup, using block size of 64 achieves a smaller increase in performance of 1.74× speedup. This difference is caused by the increased number of tasks with smaller block sizes where scheduling overhead increases and thus decreases the performance gains. This comparison is plausable in configuration 1, because both 128 and 64 block sizes have the same type and same number of accelerators on the FPGA fabric.

6.4 Conclusion

In this chapter, we presented our novel scheduling strategy for incorporating FPGA accelerators into a data-flow task parallel run-time targeting system-on-chip devices that contain multi-core CPUs and FPGAs. Our approach aims the effective use of

FPGA accelerators by actively scheduling tasks to the accelerators using work pushing to increase the occupancy of the accelerators. In addition to this, our strategy aims to offload dependent tasks to the FPGA in case there are multiple types of accelerators present on the FPGA, executing tasks in a coarse-grained pipelined fashion by taking advantage of the data-flow information provided by the OpenStream run-time.

Our technique transparently handles the data and tasks placement as well as the accelerator management without additional annotation by the programmer.The task and data placement decisions are based on inter-task dependence information readily available in modern task-parallel, data-flow run-time systems.

The experimental evaluation shows that our approach can offload up to 82% of the tasks in an application to the FPGA accelerators and shows up to 50.44% of the tasks can be scheduled to the FPGA in a pipelined fashion. We compared our approach to the CPU only execution of the same benchmarks, showing a substantial performance improvement of up to 2.76× for matrix multiplication and 2.66× for Cholesky benchmarks.

Chapter 7

Conclusion and Perspectives

This chapter summarizes the work presented in this thesis and discusses the conclusion on the findings followed by a discussion of directions for future research.

7.1 Summary

The ongoing shift in high performance computing from homogeneous to heterogeneous architectures, integrating multi-core CPUs and accelerators, exacerbates the requirements on data locality and load balancing at execution time for the efficient exploitation of all computing resources. Programming models for heterogeneous systems (e.g., OpenCL [65]) give developers control over memory allocation and execution, but burden them with technical decisions that require expert knowledge of the targeted system in order to use resources efficiently. Moreover, while such models generally provide portability across different accelerators, such low-level decisions and hard-coded optimizations make performance portability an issue. Ideally, programmers should only be responsible for expressing parallelism and data dependences, which are then mapped to hardware resources automatically.

As shown in Chapter 2, there exists multitude of approaches for efficient scheduling of task-parallel programs on heterogeneous systems that contain multi-core CPUs and GPUs. The scheduling approaches in the literature are mainly divided into three; locality-aware, data-aware and dependence-aware where each approach uses the information provided by a run-time system to make scheduling decisions using a centralized scheduler except the XKaapi [47] run-time system. However, none of the run-time systems use task-private buffers to handle data dependences between tasks except OpenStream [95] run-time which includes more information compared to other run-times as

well as provides more control over task and data for efficient placement.

Chapter 3 discusses the details of OpenStream run-time, a data-flow extension to OpenMP based on the concepts of short-lived, fine-grained tasks and streams. Through the use of streams, the communication and synchronization between tasks can be achieved in this model. The stream elements are only accessible to a task through views that reside in the task body. The synchronization is managed solely by the run-time by matching output views with input views through the same stream. The programming and execution model of OpenStream is discussed, as well as its syntax which forms the basis of the work in this thesis.

Although OpenStream provides better control for task and data placement through task-private buffers compared to similar run-times, in order to target heterogeneous architectures, we have extended OpenStream run-time with GPU and FPGA support. These extensions are detailed in Chapter 4 explaining which run-time structures are changed for employment of accelerators. A combined compiler and run-time support is detailed in addition to example programs that can execute tasks on heterogeneous platforms.

The main contributions of this thesis are presented in Chapter 5 and Chapter 6 where we explain how the additional data-flow information provided by OpenStream is exploited for efficient scheduling of task-parallel programs on heterogeneous systems. In Chapter 5 we present our novel scheduling strategy for efficiently scheduling tasks on the GPUs by reducing the synchronization between the host and the device by leveraging the data-flow information for task and data placement. In Chapter 6 we give detailed information on our novel scheduling strategy, targeting MPSoCs consisting of multi-core CPUs and FPGAs on the same chip. We propose a scheduling strategy prioritizing the FPGA accelerators over CPU workers to increase the effective use of the accelerators. In addition to this, our scheduler exploits the dependence information between tasks to execute dependent tasks on the available accelerators, creating a pipelined execution on the FPGA.

7.2 Contributions

Throughout this thesis, we have shown that data-flow task-parallel programming models can be used to increase the efficiency of the heterogeneous systems. In task-parallel models, the parallelism is explicit and favors fine-grained tasks which is essential in

modern heterogeneous architectures. Moreover, data-flow dependences provide an explicit memory view of the underlying architecture and abstract the details of memory management from the programmer. Using task-private buffers for the management of data in this model creates more opportunities for better task and data placement that can be exploited to increase efficiency and to decrease the idle time of the accelerators on heterogeneous systems.

We proposed two novel scheduling strategies for heterogeneous systems targeting multi-core CPUs and discrete GPUs as well as multi-core CPUs and FPGAs on the same chip. We proposed a strategy for GPUs for load-balancing data-flow tasks on heterogeneous systems, accounting for asymmetric compute capabilities, while simultaneously increasing on-device data reuse and decreasing synchronization between host and accelerators. The experimental evaluation shows that our approach effectively reduces the number of transfers required, reduces synchronization between CPU and GPU, increases the overlap of computation and communication, reduces GPU idle time, and increases the number of tasks offloaded to the GPU. All of these results are achieved while transparently placing data and tasks on the host and accelerators without annotation by the programmer.

In addition to the novel scheduling strategy for GPUs, we also propose a novel scheduling strategy targeting FPGA SoCs. Assuming the FPGA can be programmed to contain multiple accelerators with possible different types, we presented a novel scheduling strategy, taking advantage of the flexibility of the FPGA. While GPUs can only execute different kernels concurrently, FPGA accelerators can execute multiple tasks simultaneously decreasing the overhead of task management and providing better flexibility for heterogeneous execution. To this end, we take advantage of the data-flow information provided by the OpenStream run-time to create pipelined execution on the FPGA accelerators in addition to prioritizing FPGA accelerators rather than executing tasks on CPU workers.

Aside from the above contributions, the study presented in this thesis led to an integration and implementation of the contributions to the OpenStream run-time. This practical contributions also opens a way for future research for better analysis of data-flow task-parallel programming models as well as heterogeneous architectures.

7.3 Future Directions

The work on this thesis lead to multiple opportunities for future research. In this section, we detail some of the possible research opportunities for better exploitation of heterogeneous systems.

Dynamic adjustment of task granularity How much data is processed by each task has a strong influence on the amount of available parallelism and the scheduling overhead of task assignment and data transfers between devices. The optimal granularity not only vary between different applications, but also different systems. Although currently it is the programmers responsibility to decide the task granularity, the dynamic adjustment of different granularities can increase the performance benefits, especially in the heterogeneous system context. Achieving an optimal task granularity is itself a challenge in the context of graph optimizations. However, the data-flow information in addition to task graph traversal can be used for techniques such as task fusion, creating larger tasks for executing on compute units with higher capabilities. This not only reduces the run-time overhead of managing tasks, but also increases the efficiency of execution, leading to overall performance increase. Although using static methods for task fusion, or requiring programmer to provide hints to the run-time for fusing tasks is an option, leveraging data-flow information to achieve transparent and dynamic fusion is an interesting research opportunity.

Using machine learning for efficient task graph partitioning Although dynamic scheduling techniques try to compensate the inefficiencies of the program execution by using techniques such as load balancing, it is difficult to achieve highly efficient schedules. Since machine learning techniques are becoming omnipresent for every aspect of computer science, applying machine learning techniques that automatically learn a highly efficient workload-specific scheduling policies have been proposed [75, 2]. Applying machine learning for the optimization of task graphs in data-flow task parallel programs can be achieved with the use of data-flow information leading to optimal schedules in task parallel programs.

Using dynamic partial reconfiguration for FPGA accelerators Dynamic partial reconfiguration is the ability to reconfigure select areas of an FPGA anytime after its initial configuration. The benefits of the dynamic partial reconfiguration are: (1) it allows the adaptation of the accelerators to different parts of a program by replacing the unused accelerators with necessary ones, (2) provides wider range of accelerator support for programs that require multiple different accelerators to be used in different stages of execution, (3) increases resource utilization by allowing larger areas for

accelerators when required [14, 72, 60]. Employing dynamic partial reconfiguration within a run-time where the scheduler has full control over the task and data placement can increase the performance benefits of the accelerators, increases resource utilization on FPGA while accounting for the overhead of dynamic partial reconfiguration. Since the scheduler can dynamically traverse the future tasks, the partial reconfiguration overhead can be avoided using the data-flow information.

Integration with FPGA tool-sets to generate accelerators for creating fine-grained pipelines In this thesis, we only focused on the execution of tasks on FPGA accelerators, assuming the FPGA is already programmed and the required information is provided by the programmer. However, using HLS tools to also create the FPGA accelerators can increase the pipelining benefits. As discussed in Section 2.3.2, OmpSs run-time can create the FPGA accelerators using HLS tools provided by Xilinx. Integration of HLS tools to our run-time is also possible, but the more interesting approach is to use the execution information for the optimization of the accelerators in order to create a fine-grained pipeline behavior. Although achieving a highly optimized accelerator set is challenging, adjustment of task granularities can also be employed as a supplementary technique for the generation of accelerators.

Bibliography

[1] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12. ACM, 2000.

[2] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Placeto: Efficient progressive device placement optimization. In *NIPS Machine Learning for Systems Workshop*, 2018.

[3] Omer Erdil Albayrak, Ismail Akturk, and Ozcan Ozturk. Effective kernel mapping for opencl applications in heterogeneous platforms. In *2012 41st International Conference on Parallel Processing Workshops*, pages 81–88. IEEE, 2012.

[4] Vicki H Allan, Reese B Jones, Randall M Lee, and Stephen J Allan. Software pipelining. *ACM Computing Surveys (CSUR)*, 27(3):367–432, 1995.

[5] AMD. clblas: Opencl-based blas library. `https://github.com/clMathLibraries/clBLAS`, 2017.

[6] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide*. SIAM, 1999.

[7] David Andrews, Douglas Niehaus, and Peter Ashenden. Programming models for hybrid cpu/fpga chips. *Computer*, 37(1):118–120, 2004.

[8] David Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbie, Jorge Ortiz, Ed Komp, and Peter Ashenden. Programming models for hybrid fpga-cpu computational components: a missing link. *IEEE micro*, 24(4):42–53, 2004.

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[10] Eduard Ayguadé, Rosa M Badia, Francisco D Igual, Jesús Labarta, Rafael Mayo, and Enrique S Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *European Conference on Parallel Processing*, pages 851–862. Springer, 2009.

[11] David F Bacon, Rodric M Rabbah, Sunil Shukla, et al. Fpga programming for the masses. *Commun. ACM*, 56(4):56–63, 2013.

[12] Rosa M. Badia, José R. Herrero, Jesús Labarta, Josep M. Pérez, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, dec 2009.

[13] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, volume 8, page 56, 2004.

[14] Christian Beckhoff, Dirk Koch, and Jim Torresen. Go ahead: A partial reconfiguration framework. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 37–44. IEEE, 2012.

[15] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization*, 9(4):1–20, jan 2013.

[16] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the conference on high performance graphics 2009*, pages 159–166. ACM, 2009.

[17] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[18] Raphaël Bleuse, Thierry Gautier, João VF Lima, Grégory Mounié, and Denis Trystram. Scheduling data flow program in xkaapi: a new affinity based algorithm for heterogeneous architectures. In *European Conference on Parallel Processing*, pages 560–571. Springer, 2014.

[19] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[20] Vamsi Boppana, Sagheer Ahmad, Ilya Ganusov, Vinod Kathail, Vidya Rajagopalan, and Ralph Wittig. Ultrascale+ mpsoc and fpga families. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–37. IEEE, 2015.

[21] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *ACM SIGPLAN Notices*, volume 53, pages 316–331. ACM, 2018.

[22] Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jimenez-Gonzalez, Carlos Alvarez, and Xavier Martorell. Exploiting parallelism on gpus and fpgas with ompss. In *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*, pages 1–5, 2017.

[23] Jaume Bosch, Xubin Tan, Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, and Jesus Labarta. Application acceleration on fpgas with ompss fpga. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 70–77. IEEE, 2018.

[24] François Broquedis, Thierry Gautier, and Vincent Danjean. Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In *International Workshop on OpenMP*, pages 102–115. Springer, 2012.

[25] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[26] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Productive programming of gpu clusters with ompss. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 557–568. IEEE, 2012.

[27] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[28] Timothy J Callahan and John Wawrzynek. Adapting software pipelining for reconfigurable computing. In *CASES*, pages 57–64. Citeseer, 2000.

[29] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011.

[30] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[31] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.

[32] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *2008 Symposium on Application Specific Processors*, pages 101–107. IEEE, 2008.

[33] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[34] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[35] Bill Dally. Power, programmability, and granularity: The challenges of exascale computing. In *2011 IEEE International Test Conference*, pages 12–12. IEEE, 2011.

[36] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. Cpu db: recording microprocessor history. *Communications of the ACM*, 55(4):55–63, 2012.

[37] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. Fblas: Streaming linear algebra on fpga. *arXiv preprint arXiv:1907.07929*, 2019.

[38] Keith Diefendorff, Pradeep K Dubey, Ron Hochsprung, and HASH Scale. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[39] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. Scalable task parallelism for NUMA: A uniform abstraction for coordinated scheduling and memory management. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 125–137, -, 2016. IEEE.

[40] Feng Liu, S. Ghosh, N. P. Johnson, and D. I. August. Cgpa: Coarse-grained pipelined accelerators. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[41] Antonio Filgueras, Eduard Gil, Daniel Jimenez-Gonzalez, Carlos Alvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees Vissers. Ompss@ zynq all-programmable soc ecosystem. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 137–146. ACM, 2014.

[42] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[43] Gabriel Freytag, Matheus S Serpa, João Vicente Ferreira Lima, Paolo Rech, and Philippe OA Navaux. Non-uniform partitioning for collaborative execution on heterogeneous architectures. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 128–135. IEEE, 2019.

[44] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[45] Quentin Gautier, Alric Althoff, Pingfan Meng, and Ryan Kastner. Spector: An opencl fpga benchmark suite. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 141–148. IEEE, 2016.

[46] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23. ACM, 2007.

[47] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1299–1308. IEEE, 2013.

[48] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 347–358. ACM, 2010.

[49] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. The processing-in-memory paradigm: Mechanisms to enable adoption. In *Beyond-CMOS Technologies for Next Generation Computer Design*, pages 133–194. Springer, 2019.

[50] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, 2012.

[51] Khronos Group. Opencl specification. `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`, 2015.

[52] Jayanth Gummaraju, Ben Sander, Laurent Morichetti, Benedict R Gaster, Michael Houston, and Bixia Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 205–215. IEEE, 2010.

[53] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

[54] Joel Hestness, Stephen W Keckler, and David A Wood. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 150–160. IEEE, 2014.

[55] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[56] Jared Hoberock, Victor Lu, Yuntao Jia, and John C Hart. Stream compaction for deferred shading. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 173–180. ACM, 2009.

[57] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Characterizing and improving the use of demand-fetched caches in gpus. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 15–24. ACM, 2012.

[58] Mark Joselli, Marcelo Zamith, Esteban Clua, Anselmo Montenegro, Aura Conci, Regina Leal-Toledo, Luis Valente, Bruno Feijó, Marcos d'Ornellas, and Cesar Pozzer. Automatic dynamic task distribution between cpu and gpu for real-time systems. In *2008 11th IEEE International Conference on Computational Science and Engineering*, pages 48–55. IEEE, 2008.

[59] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Chunling Hu, Brian T Lewis, and Keshav Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 151–162. IEEE, 2014.

[60] Cindy Kao. Benefits of partial reconfiguration. *Xcell journal*, 55:65–67, 2005.

[61] K. I. Karantasis and E. D. Polychronopoulos. Programming gpu clusters with shared memory abstraction in software. In *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 223–230, Feb 2011.

[62] Vinod Kathail, James Hwang, Welson Sun, Yogesh Chobe, Tom Shui, and Jorge Carrillo. Sdsoc: A higher-level programming environment for zynq soc and ultrascale+ mpsoc. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 4–4. ACM, 2016.

[63] Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. Scheduling independent tasks on multi-cores with gpu accelerators. In *European Conference on Parallel Processing*, pages 228–237. Springer, 2013.

[64] Khronos Group. Opencl specification 1.1. `https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`, 2011.

[65] Khronos Group. Opencl specification 1.2. `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, 2012.

[66] Keunsoo Kim, Sangpil Lee, Myung Kuk Yoon, Gunjae Koo, Won Woo Ro, and Murali Annavaram. Warped-preexecution: A gpu pre-execution approach for improving latency hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 163–175. IEEE, 2016.

[67] Rakesh Kumar, Dean M Tullsen, Norman P Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[68] Ian Kuon, Russell Tessier, Jonathan Rose, et al. Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253, 2008.

[69] Changmin Lee, Won Woo Ro, and Jean-Luc Gaudiot. Boosting cuda applications with cpu–gpu hybrid computing. *International Journal of Parallel Programming*, 42(2):384–404, 2014.

[70] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 245–256. IEEE Press, 2013.

[71] W. Li, G. Jin, X. Cui, and S. See. An evaluation of unified memory technology on nvidia gpus. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1092–1098, May 2015.

[72] Wang Lie and Wu Feng-Yan. Dynamic partial reconfiguration in fpgas. In *2009 Third International Symposium on Intelligent Information Technology Application*, volume 2, pages 445–448. IEEE, 2009.

[73] Chris Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, pages 1–21, 2011.

[74] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55. IEEE, 2009.

[75] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. ACM, 2019.

[76] Christos Margiolas and Michael F. P. O'Boyle. Portable and transparent host-device communication optimization for gpgpu environments. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 55:55–55:65. ACM, 2014.

[77] Ami Marowka. Extending amdahl's law for heterogeneous computing. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 309–316. IEEE, 2012.

[78] L Martinell. *"Memory usage improvements for the SMPSs runtime*. PhD thesis, Master's thesis, Computer Architecture Department, Universitat Politècnica . . . , 2010.

[79] Jason Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179. Citeseer, 2003.

[80] Onur Mutlu, Saugata Ghose, and Rachata Ausavarungnirun. Recent advances in overcoming bottlenecks in memory systems and managing memory resources in gpu systems. *arXiv preprint arXiv:1805.06407*, 2018.

[81] CUDA Nvidia. Programming guide 1.1. *Introduction to CUDA*, 2007.

[82] Stuart Oberman, Greg Favor, and Fred Weber. Amd 3dnow! technology: Architecture and implementations. *IEEE Micro*, 19(2):37–48, 1999.

[83] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, 2008.

[84] Marc S Orr, Bradford M Beckmann, Steven K Reinhardt, and David A Wood. Fine-grain task aggregation and coordination on gpus. *ACM SIGARCH Computer Architecture News*, 42(3):181–192, 2014.

[85] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. Improving gpgpu concurrency with elastic kernels. In *ACM SIGPLAN Notices*, volume 48, pages 407–418. ACM, 2013.

[86] Prasanna Pandit and R Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 273–283, 2014.

[87] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel mmx for multimedia pcs. *Communications of the ACM*, 40(1):24–38, 1997.

[88] David Pellerin and Scott Thibault. *Practical FPGA programming in C*. Prentice Hall Press, 2005.

[89] Borja Pérez, Esteban Stafford, Jose Luis Bosque, Ramon Beivide, Sergi Mateo, Xavier Teruel, Xavier Martorell, and Eduard Ayguadé. Extending ompss for opencl kernel co-execution in heterogeneous systems. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 1–8. IEEE, 2017.

[90] Khoa Dang Pham, Anuj Vaishnav, Malte Vesper, and Dirk Koch. Zucl: A zynq ultrascale+ framework for opencl hls applications. In *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*, pages 1–9. VDE, 2018.

[91] Chuck Pheatt. Intel threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[92] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with starss. *The International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[93] Judit Planas, Rosa M Badia, Eduard Ayguade, and Jesus Labarta. Self-adaptive ompss tasks in heterogeneous environments. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 138–149. IEEE, 2013.

[94] Antoniu Pop and Albert Cohen. A stream-computing extension to openmp. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 5–14. ACM, 2011.

[95] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):53, 2013.

[96] B Ramakrishna Rau. Iterative module scheduling: An algorithm for software pipelining loops. In *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 63–74. IEEE, 1994.

[97] Alvise Rigo, Christian Pinto, Kevin Pouget, Daniel Raho, Denis Dutoit, Pierre-Yves Martinez, Chris Doran, Luca Benini, Iakovos Mavroidis, Manolis Marazakis, et al. Paving the way towards a highly energy-efficient and highly integrated compute node for the exascale revolution: the exanode approach. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 486–493. IEEE, 2017.

[98] Oren Segal, Martin Margala, Sai Rahul Chalamalasetti, and Mitch Wright. High level programming framework for fpgas in the data center. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2014.

[99] Sean O Settle et al. High-performance dynamic programming on fpgas with opencl. In *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*, pages 1–6, 2013.

[100] O. S. Simsek, A. Drebes, and A. Pop. Leveraging data-flow task parallelism for locality-aware dynamic scheduling on heterogeneous platforms. In *2018*

IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 540–549, May 2018.

[101] Richard M Stallman and the GCC Developer Community. Gnu compiler collection internals. 2014.

[102] Michael Steffen and Joseph Zambreno. Improving simt efficiency of global rendering algorithms with architectural support for dynamic micro-kernels. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 237–248. IEEE, 2010.

[103] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. The arm scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[104] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. Gramps: A programming model for graphics pipelines. *ACM Transactions on Graphics (TOG)*, 28(1):4, 2009.

[105] M Aater Suleman, Onur Mutlu, Moinuddin K Qureshi, and Yale N Patt. Accelerating critical section execution with asymmetric multi-core architectures. *ACM SIGARCH Computer Architecture News*, 37(1):253–264, 2009.

[106] Xubin Tan, Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, and Mateo Valero. A hardware runtime for task-based programming models. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):1932–1946, 2019.

[107] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[108] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[109] Marc Tremblay, J Michael O'Connor, Venkatesh Narayanan, and Liang He. Vis speeds new media processing. *IEEE micro*, 16(4):10–20, 1996.

[110] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside. Resource elastic virtualization for fpgas using opencl. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 111–1117, Aug 2018.

[111] Hans Vandierendonck, George Tzenakis, and Dimitrios S Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 1–11. IEEE, 2011.

[112] Vasily Volkov. *Understanding latency hiding on gpus*. PhD thesis, UC Berkeley, 2016.

[113] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *2011 International Conference on High Performance Computing & Simulation*, pages 24–32. IEEE, 2011.

[114] Yuan Wen, Zheng Wang, and Michael FP O'boyle. Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[115] Skyler Windh, Xiaoyin Ma, Robert J Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid A Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3):390–408, 2015.

[116] Asim Yarkhan, Jakub Kurzak, and Jack Dongarra. Quark users' guide. *Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tennessee*, 268, 2011.