ON SIMPLIFYING AND OPTIMIZING PROGRAMS FOR HETEROGENEOUS COMPUTING SYSTEMS

dissertation

by

IVAN GRASSO

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of doctor of philosophy

advisor: Prof. Dr. Thomas Fahringer, Institute of Computer Science
second advisor: Prof. Dr. Sabine Schindler, Institute of Astro and Particle Physics

Innsbruck, March 19, 2017

CERTIFICATE OF AUTHORSHIP

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Ivan Grasso, Innsbruck, March 19, 2017

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth

ACKNOWLEDGEMENTS

First and foremost, I would like to express gratitude to my academic adviser Prof. Thomas Fahringer for his guidance and support. He gave me the freedom to follow my own ideas while steering me in the right direction when necessary. I would also like to thank my second adviser Prof. Sabine Schindler as well as the thesis committee and the external reviewers.

I would further like to thank my friends and colleagues from the DPS Group. In particular, a special thank goes to Simone Pellegrini, Klaus Kofler, and Biagio Cosenza. This thesis would have never been possible without their help.

Furthermore, I would like to thank Alex Ramirez and the Heterogeneous Architectures group for the great experience during my internship at the Barcelona Supercomputing Center.

I would like to thank the funding bodies which enabled this work to be conducted: the FWF Austrian Science Fund, the FWF Doctoral School Computational Interdisciplinary Modelling (DK CIM) and the Mont-Blanc FP7 European Project.

I wish to thank my family for the unconditional encouragement throughout the course of my life. In particular, I owe my accomplishments to my parents who always supported my interest in computer science.

Last but not least, I would like to thank Birgit who has always been there for me.

ABSTRACT

Today, with the growth of highly parallel and heterogeneous architectures, systems composed of a combination of multicore CPUs, GPUs, and accelerators are becoming more common in HPC. Although heterogeneous architectures bring considerable benefits from a performance and energy perspective, they also make application development very challenging introducing the necessity of different parallel programming paradigms. Recently, in order to fully harvest the computational capabilities of such architectures, researchers focused their attention on software development tools to simplify the daunting programming task. In a similar line of investigation, this dissertation tackles the optimization and simplification of programs for heterogeneous computing systems. In the context of low-power architectures, we analyze the performance and energy advantages of embedded GPUs showing the benefits of this architecture for HPC workloads. In order to maximize the performance of heterogeneous compute nodes, we investigate a new compiler/runtime approach to generate programs that concurrently use all the heterogeneous resources and we propose two low-complexity heuristics addressing the problem of scheduling independent tasks. Finally, to simplify the development of heterogeneous distributed applications, we present *libWater*, a library-based extension of the OpenCL programming model that, with a simple interface, abstracts the underlying distributed architecture without losing control over performance.

CONTENTS

1 INTRODUCTION

1.1 MOTIVATION

High Performance Computing (HPC) has been traditionally restricted to important scientific challenges and few industrial domains where investments were large enough to support the massive infrastructure costs. However, nowadays HPC has been recognized as a powerful means to increase the competitiveness of countries through the development of their public and private sectors. Meaningful scientific discoveries have been achieved using HPC and the range of disciplines that depend on it is continuously growing [48]. The current roadmap for HPC aims at building computing systems capable of one exaflops (10^{18} floating point operations per second) by 2021 [35]. Going toward exascale, the current levels of power demand cannot be met for cost and heat dissipation reasons. These constraints have led to the introduction of supercomputers which take into account not only performance but also power consumption. The Green500 list ranks the top 500 supercomputers in the world by energy efficiency using the widely accepted metric FLOPS-per-Watt [107]. Overall, heterogeneous systems demonstrate better energy efficiency than homogeneous systems and the current list is dominated by GPU Clusters based on NVIDIA P100 Tesla GPUs [115]. Although GPUs offer a clear advantage in terms of energy efficiency, recently also other technologies are emerging. Striking examples are the Japanese effort towards building supercomputers based on embedded ARM processors [118] and the new Intel massively-parallel multicore processor Knights Landing [55]. The uncertainty regarding which architecture will be dominant in the coming years poses an additional challenge further complicating the already difficult task of designing HPC applications. Nowadays, to efficiently access different hardware resources and levels of parallelism, developers are required to combine multiple languages and programming paradigms. Therefore, the availability of automatic tools to mitigate the complexity of software

and increase its portability across different hardware platforms has become of paramount importance.

1.2 STATE OF ART

Several paradigms and frameworks have been proposed in the literature for programming parallel heterogeneous architectures. This section will give an overview of the most successful models while the related work for each area of contribution will be discussed in more detail within their respective chapters.

Nowadays the main approaches used to program heterogeneous hardware fall in two categories: explicit programming models and directive-based models.

Explicit Programming Models. Explicit models give the developer complete control over the hardware resources to create very efficient and optimized code. The most renowned explicit programming models are CUDA and OpenCL.

CUDA [86] is a parallel computing model created by NVIDIA. It allows software developers to use CUDA-enabled GPUs for general-purpose processing. The CUDA platform is a software layer, designed to work with programming languages such as C, C++, and Fortran, that gives direct access to the GPU's virtual instruction set and parallel computational elements.

OpenCL [60] is an open standard for programming parallel heterogeneous architectures maintained by the Khronos Group consortium. It provides an interface for writing programs that execute across heterogeneous platforms as central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs), and other hardware accelerators.

Directive-based Models. Programming with explicit models is often a difficult task, especially for non-computing specialists. An alternative is represented by directive-based models that allow developers to mark regions of code for parallel acceleration with a minimum amount of effort. OpenMP and OpenACC are the most representative models of this category.

OpenMP [89] is an API consisting of compiler directives and library routines for high-level parallelism in C, C++, and Fortran programs.

Starting with OpenMP 4.0, it is possible to offload parallel regions to different devices, such as GPUs or accelerators. The OpenMP directives are platform independent, allowing a high degree of portability and performance with little programming effort.

OpenACC [88] provides a collection of compiler directives to specify loops and regions of code in standard C, C++, and Fortran to be offloaded from a host CPU to an attached GPU or accelerator. The OpenACC directives allow programmers to create high-level host/device programs without the need to explicitly initialize the devices or manage data transfers. All of these details are implicit in the programming model and are managed by the OpenACC compilers and runtimes.

Recently, *SYCL* [62], a new abstraction layer that builds on the concepts of portability and efficiency, was introduced. SYCL is a royalty-free, cross-platform layer that enables code for heterogeneous processors to be written in a single-source style using standard C++. Template functions can contain both host and device code to construct complex algorithms that use OpenCL acceleration.

1.3 THESIS GOALS AND ORGANIZATION

This thesis will focus on OpenCL that currently offers a clear advantage in terms of number of accessible devices and level of hardware control. Using the OpenCL programming model, we will explore and discuss novel approaches aiming at both simplifying and optimizing programs for heterogeneous computing systems. This thesis is divided into five additional chapters as follows:

Chapter 2 introduces the models that describe all the abstractions used in this work.

Chapter 3 focuses on embedded systems analyzing the performance and energy advantages of embedded GPUs for HPC. We identify, implement and evaluate software optimization techniques for efficient utilization of the ARM Mali GPU Compute Architecture showing for the first time that embedded GPUs have qualities that make them good candidates for HPC systems.

Chapter 4 investigates the distribution of tasks among the available devices in order to maximize the performance of heterogeneous com-

pute nodes. Based on a powerful compiler/runtime infrastructure, we develop a new approach for converting a single-device OpenCL program to a multi-device OpenCL program able to concurrently take advantage of all the heterogeneous resources.

We also propose and implement two low-complexity heuristics addressing the problem of scheduling independent tasks in heterogeneous compute nodes.

Chapter 5 introduces *libWater*, an extension of the OpenCL programming model that simplifies the development of heterogeneous distributed applications. libWater consists of a simple interface, which is a transparent abstraction of the underlying distributed architecture. It provides a runtime system which tracks dependency information enforced by event synchronization to dynamically build a DAG of commands, on which we automatically apply optimizations.

Chapter 6 concludes the thesis discussing future work and providing the list of peer-reviewed publications which support our findings.

2

MODEL

In this chapter we introduce the models that describe all the abstractions used in the thesis. In Section 2.1, starting from basic concepts we formally describe the structure of the hardware architectures addressed by the analyses and optimizations. The software model, presented in Section 2.2, introduces the terminology for program and parallelism. In Section 2.3, we describe the concepts and the architecture of the Open Computing Language model. Finally, in Section 2.4, we present the message passing model.

2.1 HARDWARE MODEL

In this section, using a bottom-up approach, we introduce and define the concepts that describe the hardware which is utilized in this thesis. The hardware is represented through a formal model composed of entities and relations.

Let E denote an entity set and let us define a basic entity $e \in E$ to be a hardware item which can be distinctly identified. Let us define $r \in R \subseteq E \times E$ as a binary relation on E that enables the transfer of data between two entities. Then a complex entity $c = (E, R)$ consists of a non-empty set E of basic or complex entities, and a set $R \subseteq E \times E$ of relations. Relations can also be expressed in a more compact form $(e_i, e_j), (e_j, e_k) \in R \equiv \langle e_i, e_j, e_k \rangle \in R$. This form defines a sequence of connected entities and is useful to represent how data moves between the different entities. Entities and relations can also feature attributes that describe their elementary properties. An $attribute : E \vee R \rightarrow V$ can be formally defined as a function which maps an entity set or relation set into a value set V, also known as the domain of the attribute.

Definition 2.1 (Latency and Bandwidth)

$\forall\, r = (e_j, e_k) \in R \;\; \exists\, latency : R \to \mathbb{R},\ \exists\, bandwidth : R \to \mathbb{N}$

The attribute *latency* of a relation r is defined as the time, in seconds, required to transfer a minimal amount of data between e_j and e_k. Similarly, the attribute *bandwidth* is defined as the amount of data that can be transferred from e_j to e_k per second.

Definition 2.2 (Computing Unit)

$\forall\, cu = (E, R) \in CU \;\; \exists\, es, eu \in E \mid (es, eu) \in R$

A *computing unit*, cu, is defined as a complex entity that performs arithmetic, logical, and control operations. Such an entity is composed of an *execution state*, es, and one or more *execution units*, eu.

The *execution state* comprises general purpose and interrupt controller registers, while an *execution unit* comprises the resources for the instruction execution. The execution unit also features an attribute *width* that denotes the size of the data that can be processed in a single step of execution. In some cases, to better utilize the resources, a computing unit contains duplicated execution states that share the same execution units. This technique is known as simultaneous multithreading (SMT) and is used to make a single computing unit appear as multiple logical computing units.

Definition 2.3 (Memory Unit)

A *memory unit*, $mu \in MU$, is defined as a basic entity capable of temporarily storing user or system data. Such an entity holds a relation with the entities which can access its content (i.e., computing units or memory units) and features an attribute *size* that expresses the total amount of memory available in bytes.

Particular memory units are *cache* and *local memory*. A *cache*, L_n, is defined as a small and low-latency memory unit whose content is automatically managed in hardware. Caches are usually organized in hierarchies of memory units and feature an attribute level $n : MU \to \mathbb{N}$ that denotes their position in the hierarchy. A cache in a lower level (e.g., L_1) contains only a subset of the data present in a higher level cache (e.g., L_2). Caches that share common data need to be synchronized to offer a coherent view of their content. This difficulty, generally known as *cache coherence problem*, is usually solved in hardware

with the use of a protocol that tracks the state of all the shared data. In addition to the *size* and *level*, a cache entity features the following attributes:

- *associativity* - denotes the number of cache locations in which a memory entry can be mapped.
- *replacement policy* - represents the heuristic utilized to choose the cache entry to evict in order to make room for a new entry.

A *local memory*, LM, is defined as a small and low-latency memory unit that can be directly addressed by the programmer. The local memory does not operate like a cache since it is neither transparent to the software nor does it contain hardware structures able to predict the data to load. For these reasons, it consumes relatively little power compared with hardware-managed caches but also introduces new challenges for the programmer that needs to explicitly maintain the data consistency.

Definition 2.4 (Processor)
$\forall\ CPU = (E, R)\ \exists\ cu, L_1, ..., L_n \in E \mid \langle cu, L_1, ..., L_{n-1}, L_n \rangle \in R$
A *processor*, CPU, represents the core component of a computing system and is defined as a complex entity consisting of one or more computing units connected with one or more levels of cache.

The processor is designed to provide good performance during the execution of the operating system and the broad range of user applications. As depicted in Figure 2.1, based on the number of computing units and shared resources, a processor is named differently:

- *single core* - a processor with a single computing unit and one or more levels of cache.
- *multi-core* - a processor with two or more computing units and one or more levels of cache.
- *multi-core with SMT* - a multicore processor with simultaneous multi-threading.

A processor entity features the following attributes:

- *number of cores* - denotes the number of computing units with related L1 caches.

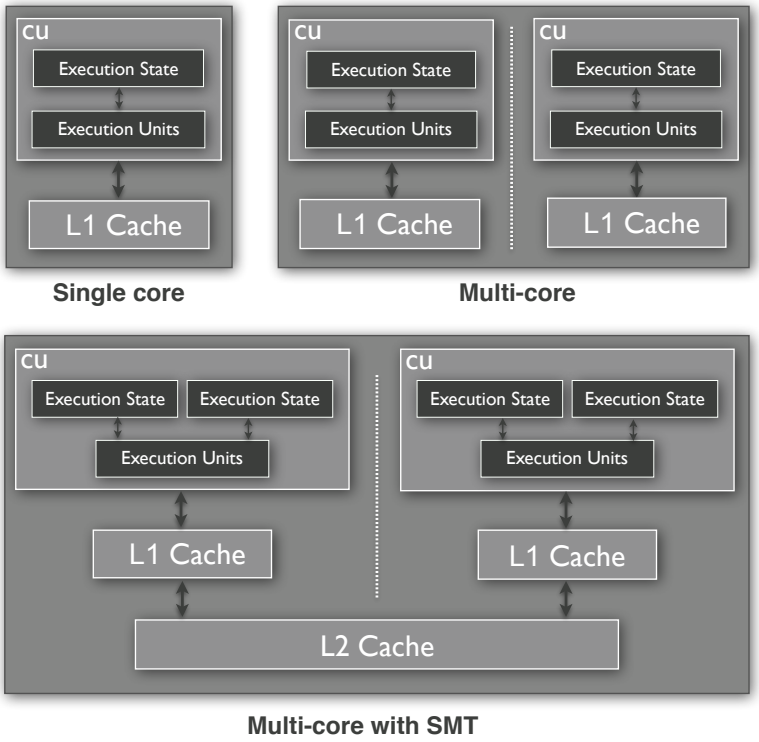


Figure 2.1: Hardware model for different types of processors

- *clock frequency* - the maximum frequency at which the processor can run.

Definition 2.5 (Graphics Processing Unit)
$\forall\ GPU = (E, R)\ \exists\ cu, L_1, L_2, LM, mu \in E\ |$
$\langle cu, L_1, L_2, mu\rangle, \langle cu, LM, mu\rangle \in R$
A *graphics processing unit*, GPU, is defined as a complex entity able to perform highly parallel operations on data. Such an entity consists of a large memory unit and multiple computing units connected with multiple cache levels and local memories.

GPUs have evolved from special-purpose entities purely used for graphics processing to programmable entities that can execute a wide range of applications. As depicted in Figure 2.2, each computing unit of a GPU contains a large number of execution units that, combined together, offer high theoretical peak performance. To absorb the memory request of the computing units, the GPU entity takes advantage of a memory hierarchy and a high bandwidth memory unit. The bandwidth attribute of the relation between the memory unit and

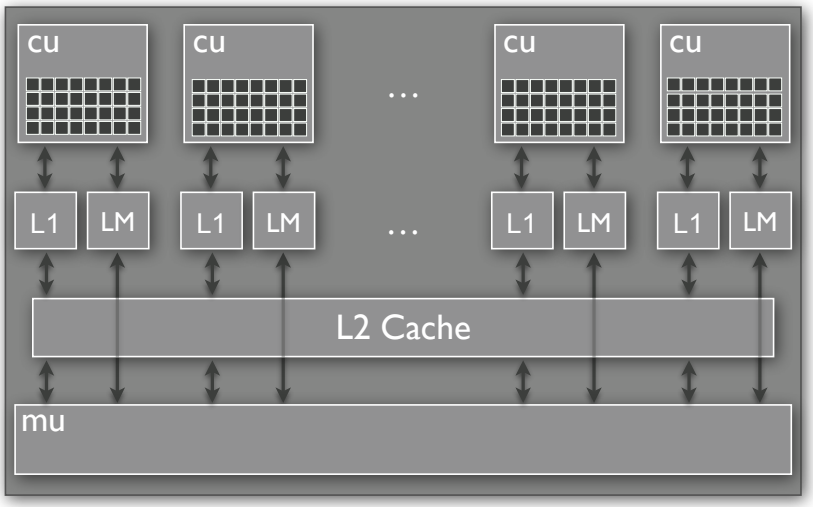


Figure 2.2: Hardware model for the GPU entity

the caches/local memories is in the order of hundreds of GB/s.
A GPU entity features the following attributes:

- *number of computing units* - denotes the number of computing units.
- *clock frequency* - expresses the maximum frequency at which the computing units can run.
- *memory size* - denotes the total amount of GPU memory available in bytes.

Definition 2.6 (Accelerator)

$\forall\, ACL = (E, R)\ \exists\, cu, L_1, L_2, mu \in E \mid \langle cu, L_1, L_2, mu \rangle \in R$

An *accelerator*, ACL, is defined as a complex entity able to accelerate highly parallel computing workloads. Such an entity consists of a large memory unit and many computing units connected with multiple cache levels.

As depicted in Figure 2.3, the accelerator hardware model resembles the CPU hardware model. Modern accelerators (e.g., Intel Xeon Phi) are based on a many-core architecture that consists of a large number of computing units with simultaneous multi-threading running at a low clock frequency. Similar to a GPU entity, an accelerator is composed of a cache hierarchy, a high bandwidth memory unit and

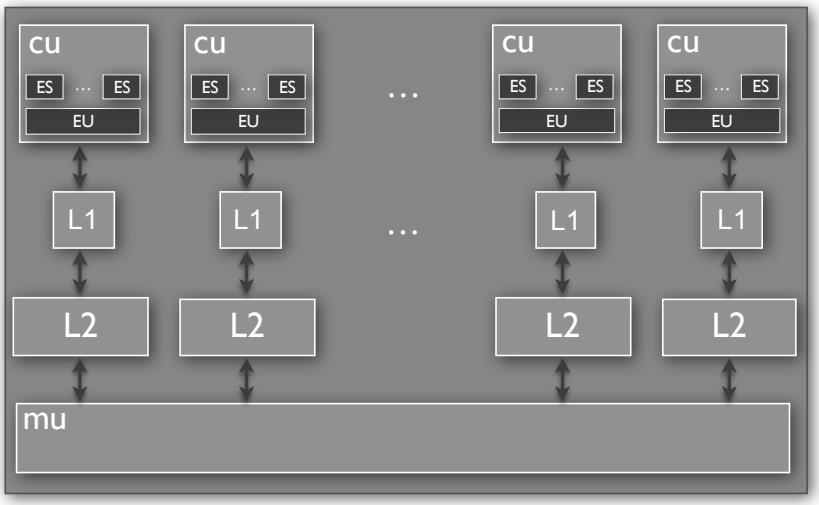


Figure 2.3: Hardware model for the accelerator entity

supports the same entity attributes.

Definition 2.7 (Secondary Storage)
A *secondary storage* or *disc storage*, ss, is defined as a basic entity that is capable of permanently storing user or system data.

Definition 2.8 (Network Interface)
A *network interface*, ni, is defined as a basic entity used to physically interface a compute node with other compute nodes or network-attached secondary storages.

Different network interfaces offer different performance characteristics. Usually, a relation between network interfaces based on high-speed interconnects, like Infiniband EDR, has an attribute bandwidth of around 100 Gbps and an attribute latency of 130 ns. A relation based on more affordable interconnects, such as Ethernet, has an attribute bandwidth of 10 Gbps and an attribute latency of 2 to 4 µs.

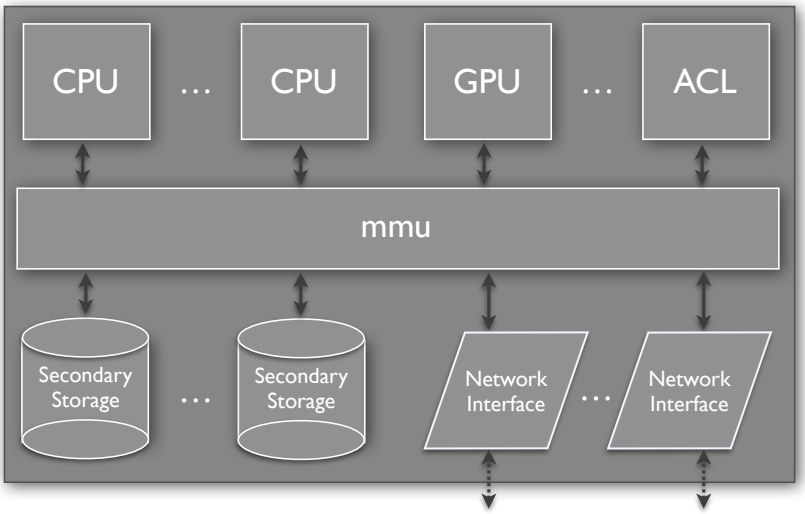


Figure 2.4: Hardware model for the compute node entity

Definition 2.9 (Compute node)

$\forall\, cn = (E, R) \in CN \;\; \exists\, CPU, GPU, ACL, ss, ni, mmu \in E \mid$

$\forall e \in E \mid e \neq mmu \;\exists\, (e, mmu) \in R$

A *compute node* is defined as a complex entity representing a computing system (e.g., desktop, server). As depicted in Figure 2.4, such an entity can consist of multiple CPUs, GPUs, accelerators, secondary storages and network interfaces, all connected with the same *main memory unit*, mmu.

Based on the entities that compose a compute node, it can be defined as:

- *homogeneous* - a node with one or more CPUs but without GPUs or accelerators.
- *heterogeneous* - a node with one or more CPUs and one or more GPUs or accelerators.

GPUs and accelerators are external devices that need to be connected through PCI Express expansion slots. Any data requested from such entities needs to be copied from the compute node memory(mmu) to the entity memory and back before being accessible by the CPU entity. Due to the current bandwidth limitation of the PCI Express (8-16 GB/s), these memory transfers can lead to significant overhead during the transmission of data increasing the overall workload processing time.

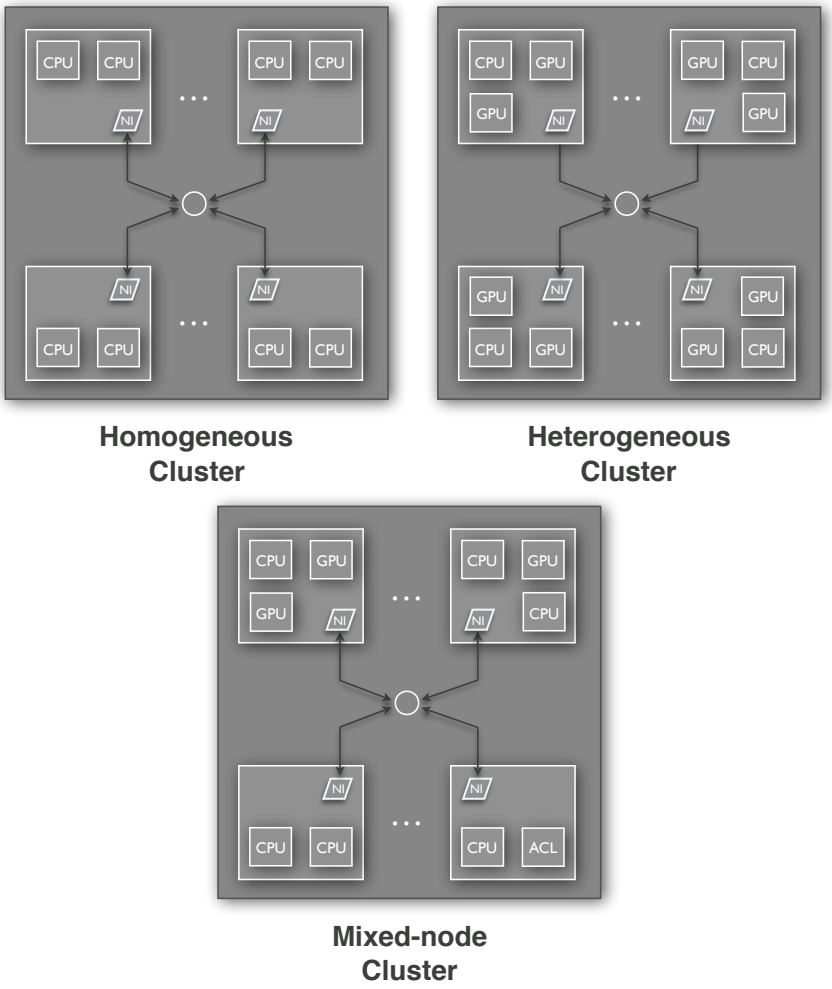


Figure 2.5: Different types of clusters

Definition 2.10 (Cluster)
A *cluster* or *distributed memory system* is defined as a complex entity composed of a set of locally connected compute nodes.

As depicted in Figure 2.5, based on the type of compute nodes that compose a cluster, it can be defined as:

- *homogeneous* - composed of homogeneous compute nodes.
- *heterogeneous* - composed of heterogeneous compute nodes.
- *mixed-node* - composed of compute nodes of different types.

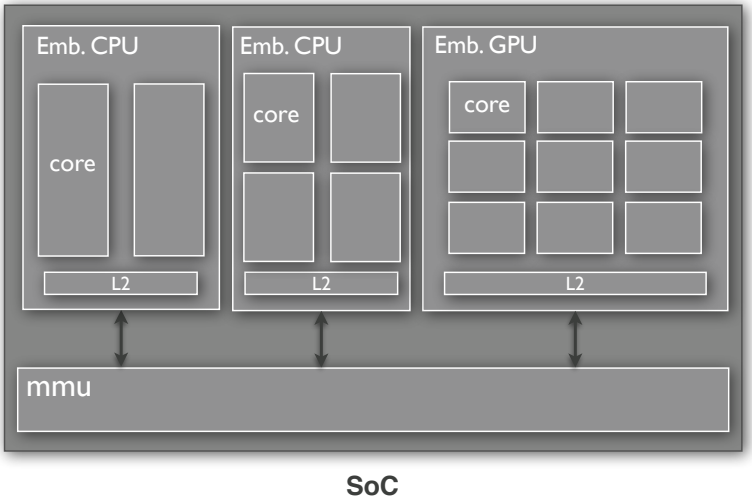


Figure 2.6: Hardware model for the SoC entity

Definition 2.11 (System on Chip)
A *system on chip*, SoC, is defined as a complex entity composed of embedded CPUs, embedded GPUs and memory units, packed together into a single chip.

Differently from CPUs and GPUs present in compute nodes, the embedded versions present in SoCs need to be designed taking into account more strict power and space constraints. As depicted in Figure 2.6, to save space, embedded GPUs do not have a GPU memory unit, but share the main memory unit with the other entities present in the chip. Modern SoCs often also provide a big.LITTLE hardware configuration [40] with multiple CPUs that feature different performance and power characteristics. In such case, to reduce the overall power consumption of the chip, the operating system can choose to activate or deactivate the different CPUs depending on the applications workload.

2.2 SOFTWARE MODEL

As mentioned in the previous section, a compute node is composed of a variety of different hardware entities (i.e., memories, processors). Application programs do not directly access the hardware but use the abstract set of resources that the operating system provides. The operating system accesses and manages the hardware controlling the

allocation of entities among the various programs that request them.

Definition 2.12 (Program)
A *program*, $r = (M, V)$, is defined as a sequence of *statements* M, and a *program state* V.

A *variable* is defined as a storage location, with a specific *data type* and an associated *identifier*, which contains some quantity of information referred to as *value*. The attribute data type determines the set of values that can be stored in a variable and the set of operations that can be executed on it. A *program state* is defined, at any given point in the program's execution, as the content of all the variables $v \in V$.
A *statement*, $m \in M$, represents the basic element of a program and describes the computation by altering the values stored in the variables $v \in V$. Statements are generally executed in sequence from top to bottom, however, the program may contain control flow statements that alter the *flow of execution* (e.g., if, switch, for statements). To facilitate the readability of a program, statements are often grouped together in functions that perform specific tasks.
A *function* is commonly defined as

$$name\ (v_0, ..., v_n) \rightarrow type\ \{m_0, ..., m_m\}$$

where

- *name* - denotes the identifier by which the function can be called.
- $v_0, ..., v_n$ - is an ordered list of variables, called *parameters*, that are initialized with the corresponding arguments passed in the function call.
- *type* - expresses the type of the value returned by the function.
- $m_0, ..., m_m$ - represents the block of statements executed by the function.

Every program contains a *main function* which designates the starting point of the flow of execution.

Definition 2.13 (Compiler)
A *compiler* is a program capable of translating a *source program* into a semantically equivalent *target program*.

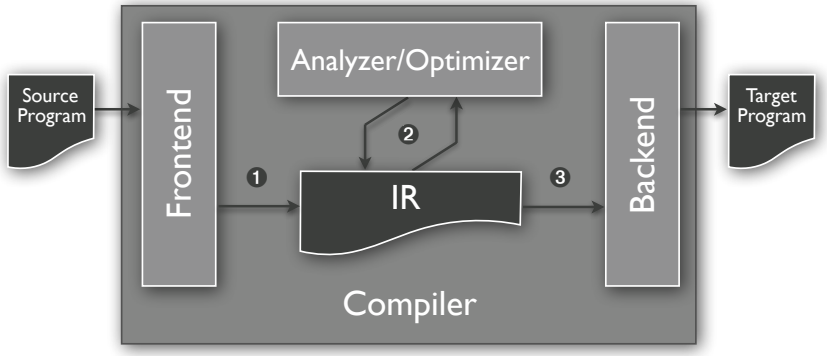


Figure 2.7: Compiler software model

As depicted in Figure 2.7, a compiler is composed of a frontend, an analyzer/optimizer, and a backend. In step ❶, the frontend translates the input code into an internal *intermediate representation* of the source program, called IR. An important task of the frontend is to collect and provide information regarding the syntactic and semantic errors encountered during the translation phase, so that the user can take corrective actions in the source program. In step ❷, the analyzer applies control-flow and data-flow analysis [2] on the IR to collect useful information. The optimizer, based on the results of the analysis, applies some transformations with the purpose of optimizing the representation and improving the quality of the generated code. Finally, in step ❸, the backend constructs the desired target program translating the IR into the target language representation. A compiler is often coupled with a *runtime system*, which implements the dynamic features and capabilities of the source language and controls the execution of the target program applying further optimizations.

Definition 2.14 (Process)
A *process*, $p \in P$, is a software entity that represents an instance of an executing program. Such an entity is composed of an address space, and one or more threads of execution.

The *address space*, A_p, represents the list of memory locations which the process p can read and write and contains program text and data, as well as other resources. These resources may include open files, signal handlers, and more. A *thread* represents a basic entity capable of processing a sequential flow of execution. Multiple threads of the same process share the same address space and consequently the

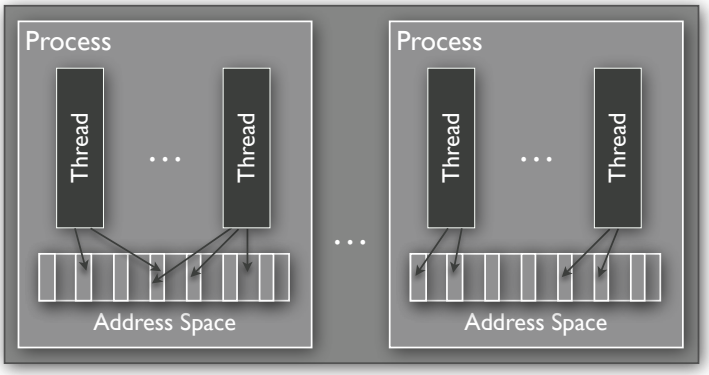


Figure 2.8: Representation of multiple process entities

same resources. Each physical or logical computing unit of a CPU runs only one thread at the time and quickly switches from thread to thread. A CPU thread features an attribute *state* that can transit at any time between three possible values:

- *running* - currently using a computing unit.
- *ready* - temporarily stopped to let another thread run.
- *blocked* - waiting for some external event.

As depicted in Figure 2.9, four transitions are possible among the three values. Transition ❶ occurs when the operating system discovers that a thread cannot continue. Transition ❷ occurs when the operating system decides that the running thread has run long enough, and it is time to let run another thread. Transition ❸ occurs when the operating system selects a thread to run. Transition ❹ occurs when the external event for which a thread was waiting happens.

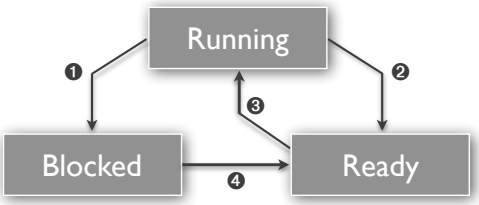


Figure 2.9: Values of the attribute state of a thread

As previously describe in Section 2.1, GPUs contain computing units with a large number of execution units. Differently from CPUs, the GPU computing units are capable of executing multiple threads in parallel. Following a model known as *Single-Instruction, Multiple-*

Thread (SIMT), all the individual threads running in the same computing unit (*thread group*) start at the same program address and execute in lock-step the same instructions. If some of these threads diverge via a data-dependent conditional branch, the computing unit serially executes each branch path taken, disabling the threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. The execution context (program counters, registers, etc.) for each thread group processed by a computing unit is maintained until all threads have completed the execution. A computing unit can therefore switch between different thread groups with no additional cost. Full utilization of the GPU resources is achieved when all computing units have always some instruction to issue for some thread group, or in other words, when memory latency is completely hidden.

Definition 2.15 (Sequential/Parallel Program)
A *sequential program* is defined as a program which uses a single process with a single thread of execution. On the contrary, a *parallel program* is defined as a program which uses multiple threads of execution in one or multiple processes.

Sharing the same address space, multiple threads of the same process can communicate by accessing or updating the common memory resources. This communication mechanism requires the use of synchronization constructs to coordinate the execution of multiple threads and allow exclusive access to share data. Differently, in case of multiple processes, the communication is done by message passing that allows to exchange data by sending and receiving explicit messages between processes. A formal description of this model will be presented in Section 2.4.

Definition 2.16 (Speedup and Efficiency)

$S_n = T_s/T_n \quad E_n = S_n/n$

Let T_s be the execution time of a sequential program, and T_n be the execution time of the parallel version of the same program using n processing entities (e.g., threads, processes, compute nodes). The *speedup*, S_n, is then defined as the relation between the sequential and the parallel execution times, while the *efficiency*, E_n, is defined as the relation between the speedup and the number of processing entities.

The *scalability* of a parallel program describes how its execution time varies with the number of processing entities n. There are two distinct scalability measurement: *strong scaling* and *weak scaling*. In case of strong scaling, the number of processing entities are increased, while maintaining a constant problem size. A program is considered to scale linearly if the speedup remains close to the *ideal speedup*, which is obtained when $S_n = n$. In case of weak scaling, the problem size increases with the number of processing entities. A program is considered to scale linearly if the execution time stays constant while the workload is increased. In general, it is harder to achieve good strong-scaling with a large amount of processing entities since the communication overhead for many algorithms increases in proportion to the number of entities used.

2.3 THE OPEN COMPUTING LANGUAGE MODEL

The Open Computing Language, *OpenCL* [60], is the first open industry standard for programming heterogeneous compute nodes composed of devices with different capabilities such as CPUs, GPUs, and accelerators. In the past few years, OpenCL has emerged as the de facto standard for heterogeneous computing, with the support of many industry vendors such as Adapteva, Altera, AMD, Apple, ARM, Intel, Imagination Technologies, NVIDIA, Qualcomm, Vivante, and Xilinx. In the next sections we will introduce the basic concepts and the architecture of OpenCL, followed by a detailed description of its execution and memory models.

2.3.1 *Platform Model*

OpenCL views heterogeneous computing entities through an abstract, hierarchical *platform model*. As depicted in Figure 2.10, this model consists of a *host* connected to a set of *compute devices*.

Definition 2.17 (Host)
A *host*, is defined as a software abstraction of the hardware entity that starts the execution of the OpenCL program. In a compute node the host is always identified with the CPU.

Definition 2.18 (Compute Device)
A *compute device* is defined as a software abstraction of a device hardware entity such a CPU, GPU or ACL. Each compute device is composed of one or more *compute units*, and each compute unit is composed of one or more *processing elements*.

A compute node with multiple vendor compute devices features multiple vendor platforms. OpenCL offers an API to discover the set of vendor platforms and query the specific available devices. The relationship between the elements of a compute device and the components of the respective hardware entity depends on how the platform compiler optimizes the OpenCL source code, to best utilize the available physical resources. Consequently, every industry vendor offers a customized compiler that generates device-specific executable code and a runtime system capable of optimizing the execution of such code.

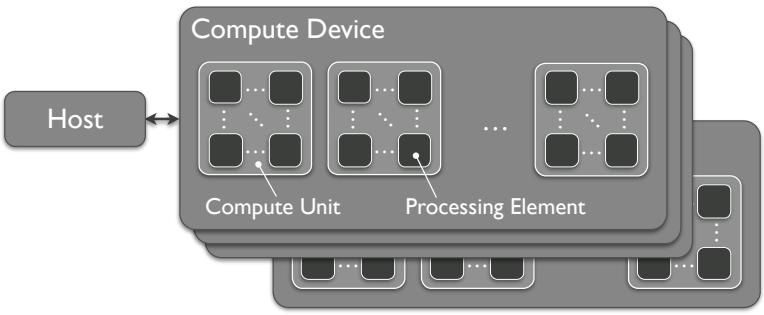


Figure 2.10: OpenCL Platform Model

2.3.2 *Execution Model*

An OpenCL application consists of a *host program*, executed on the host, and one or more *kernels*, executed on the compute devices.

Definition 2.19 (Host Program)
The *host program* is the part of the OpenCL program that coordinates the execution by setting up the OpenCL environment, transferring data to and from the available compute devices and invoking the execution of the kernels.

Definition 2.20 (Kernel)
A *kernel* is a special function of an OpenCL program that can be executed on a compute device.

To enable host-device interaction, during the creation of the OpenCL environment, a *command-queue* is associated with each compute device. The command-queue is then used by the host program to coordinate the program execution using different types of commands:

- *kernel execution commands* - to enqueue a kernel for execution on a device.
- *memory commands* - to transfer data between the host and device memory.
- *synchronization commands* - to constrain the order of execution of commands.

When a kernel is executed, based on a n-dimensional index space (*NDRange*), a certain number of kernel instances are created and run in parallel across multiple processing elements. Each instance, called *work-item*, is identified by a global ID that represents its position in the index space. The index space can also be subdivided into equally sized *work-groups*, each of them consisting of many work-items. Work-items can only communicate and synchronize locally, within a work-group, providing flexible scheduling options to the runtime system during the program execution. To request the execution of a specific kernel from a compute device, the OpenCL API offers a function defined as

EnqueueKernel (queue, kernel, work_dim, global_size, local_size)

where

- *queue* - denotes the command-queue of the target device.
- *kernel* - identifies the kernel to be executed.
- *work_dim* - is the number of dimensions of the index space.
- *global_size* - denotes the number of work-items in each dimension of the index space.
- *local_size* - specifies the number of work-items in each dimension of the workgroups.

The programming language used to express the computation inside the kernel functions is called *OpenCL C* and it is derived from the ISO C99 specification. On the one hand, to ensure hardware portability, some of the C99 features such as recursive functions, function pointers and bit fields were removed in the kernel language specification. On the other hand, OpenCL C provides several beneficial features such as support for vector intrinsics and vector data types, atomic operations and a large set of optimized built-in functions.

The OpenCL specification also defines two different profiles: a profile for compute devices (*Full Profile*) and a profile for embedded compute devices (*Embedded Profile*). Embedded devices present in SoCs have significant area and power constraints that require a relaxation in the requirements defined by the Full Profile specification. The Embedded Profile relaxes the floating-point precision requirements and support for mathematical functions, does not require atomic functions and reduces some of the minimum parameters of different components of the framework.

2.3.3 *Memory Model*

The OpenCL memory model describes the structure and behavior of the memory exposed by an OpenCL platform during the program execution. Memory in OpenCL is divided into two parts: *host memory*, available to the host, and *device memory*, available to kernels executing on the compute devices. As depicted in Figure 2.11, the device memory comprises four memory regions:

- *global memory* - largest memory space available to the device, visible to all work-items.
- *constant memory* - read-only region of global memory.
- *local memory* - memory local to a work-group, to share data between work-items in a work-group.
- *private memory* - memory private to a work-item.

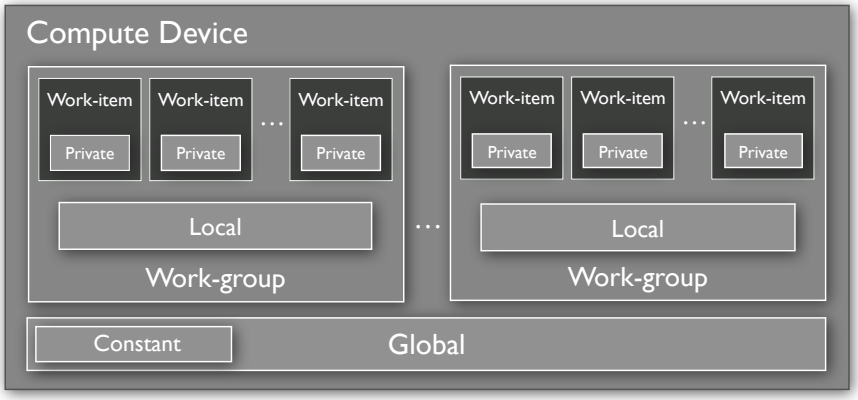


Figure 2.11: Device Memory regions

To provide good scalability, OpenCL offers a relaxed consistency model in which, as the computation progresses, different work-items may see a different view of global memory. Within a work-item, reads and writes to all memory spaces are consistently ordered, but between work-items, synchronization is necessary in order to ensure consistency.

Definition 2.21 (Buffer)
A *buffer* is defined as an OpenCL memory object that stores a one-dimensional collection of data elements. Elements of a buffer can have a scalar data type (e.g., int, float), a vector data type (e.g., float4, double8), or a structure data type composed of scalar or vector data types.

Buffers are used to exchange data between host and compute devices and are passed as arguments to the kernels. The interaction between the host memory and the device memory can occur by explicitly copying buffer data between the two memory regions, or by allowing the host to map/unmap a region of the buffer into its own address space.

2.3.4 *Programming Model*

To effectively allow the mapping of parallel algorithms, OpenCL supports two different programming models: task parallel and data parallel.

Definition 2.22 (Task Parallel Programming Model)
In a *task-parallel programming model*, problems are decomposed in tasks that can be executed in parallel on the available hardware resources.

In OpenCL to execute a program composed of multiple kernels, the host program can asynchronously push many commands in different command-queues without waiting for their completion. Each command can generate an event and be associated with a list of events that specifies dependencies on other commands. The set of commands and their respective dependencies define a task graph that the OpenCL runtime system will schedule during the program execution.

Definition 2.23 (Data Parallel Programming Model)
In a *data-parallel programming model*, problems are decomposed in collections of data elements that can be updated in parallel applying the same stream of instructions to each data element.

This data-centric model matches perfectly the OpenCL execution model in which work-items, defined in the index space, execute in parallel the sequence of instructions specified in the kernel.

2.4 THE MESSAGE PASSING MODEL

The OpenCL programming model is a complete solution for programming heterogeneous compute nodes but it can not be used for inter-node communications in distributed memory systems. Although in some particular distributed memory system the physically separate memories can be addressed as one logically shared address space (distributed shared memory system), in this thesis we will refer only to distributed systems that rely on the message passing model to transfer data between processes. In this model, processes from different compute nodes communicate through the exchange of messages. The Message-Passing Interface, *MPI* [83], is the de facto message-passing

standard for parallel processing. The technical specification of the standard is nowadays implemented in multiple libraries (e.g., Open-MPI [90], MPICH [84], MVAPICH [85]). In the next sections we will introduce the formal model of the communication protocol.

2.4.1 *Point-to-Point Communication Model*

In the message passing model data is moved from the address space of one process to that of another process. This procedure is accomplished through the cooperation of the two processes involved in the communication. The process responsible for sending the data is called *sender*, while the one responsible to receive the data is called *receiver*. Below, we formally describe multiple functions that the model offers to allow collaborative point-to-point communications between processes. In each function we denote with P the set of all processes of a parallel program.

Definition 2.24 (Send/Receive)
The function *send*($addr_s$, *size*, p_d) specifies that a sender $p_s \in P$ transfers *size* bytes of data from the address $addr_s \in A_{p_s}$ to a destination process $p_d \in P$.
The function *receive*($addr_d$, *size*, p_s) specifies that a receiver $p_d \in P$ receives *size* bytes of data from the process $p_s \in P$ and stores them starting from the address $addr_d \in A_{p_d}$.

It is worth noting the asymmetry between send and receive functions: a receiver may accept messages from an arbitrary sender, on the other hand, a sender must specify a unique receiver. To allow the receiver to accept a message from any source process $p \in P$ a wildcard * can be specified as a parameter in the receive function.

Send and receive are blocking primitives. A send operation can be started whether or not a matching receive function in a remote process is called. However, the send will return only when a receive will be invoked and the data transfer will be completed. Better performance is often achievable with the use of nonblocking communication primitives.

Definition 2.25 (Isend/Ireceive)
The functions *isend*($\mathtt{addr}_s$, *size*, p_d, rq) and *ireceive*($\mathtt{addr}_d$, *size*, p_s, rq), differently from the blocking version defined in Def. 2.24, need an additional request parameter rq used to query the status of the of the asynchronous data transfer.

The isend and ireceive functions start respectively the send and the receive operations on the data, but return immediately without waiting for a matching operation. This allows the program to proceed with the computation while the communication is taking place. During the data transfer, the sender must not modify the memory locations $[\mathtt{addr}_s, \mathtt{addr}_s + size) \in A_{p_s}$, while the receiver must not access the memory locations $[\mathtt{addr}_d, \mathtt{addr}_d + size) \in A_{p_d}$. To check for the completion of nonblocking operations the model offers two functions: *test* and *wait*.

Definition 2.26 (Test/Wait)
The function *test*(rq, flag) sets the flag parameter to true if the operation identified by the request parameter rq is completed. The function sets the flag parameter to false otherwise.
The function *wait*(rq) returns when the operation identified by the request parameter rq is completed.

2.4.2 *Collective Communication Model*

In addition to point-to-point communications, the message passing model also defines communications that involve groups of processes. In our model, we assume that all the processes of a parallel program participate in collective communications. Below, we formally describe multiple functions that the model offers to allow such communications.

Definition 2.27 (Broadcast)
The function *broadcast*(addr, *size*, p_r) specifies that the root process $p_r \in P$ transfers *size* bytes of data from the address $\mathtt{addr}_r \in A_{p_r}$ to the processes $P - p_r$. All the processes need to call the broadcast function with the same root process $p_r \in P$, but specifying a different local destination address $\mathtt{addr}_d$ such that $\forall\ p_d \in (P - p_r)\ \exists\ \mathtt{addr}_d \mid \mathtt{addr}_d \in A_{p_d}$.

It is worth noting that while for the root process the broadcast represents a read operation, for all the remaining processes it represents a write operation in their local address spaces.

Definition 2.28 (Scatter)
The function *scatter*($addr$, *size*, p_r) specifies that the root process $p_r \in P$ distributes *size* bytes of data from the address $addr_r \in A_{p_r}$ to the processes $P - p_r$. The outcome of the function is described by the following formula:

$$\forall\, p_d \in (P - p_r)\ \exists\ addr_d \in A_{p_d} \mid [addr_d, addr_d + (size/|P|)) = [addr_r + (size/|P|) * id_{p_d}, addr_r + (size/|P|) * (id_{p_d} + 1))$$

Definition 2.29 (Gather)
The function *gather*($addr$, *size*, p_r) specifies that the root process $p_r \in P$ receives $(size/|P|)$ bytes of data from each process $p_s \in (P - p_r)$ and stores them starting from the address $addr_r \in A_{p_r}$. The outcome of the function is described by the following formula:

$$\forall\, p_s \in (P - p_r)\ \exists\ addr_s \in A_{p_s} \mid [addr_r + (size/|P|) * id_{p_s}, addr_r + (size/|P|) * (id_{p_s} + 1)) = [addr_s, addr_s + (size/|P|))$$

Definition 2.30 (Barrier)
The function *barrier*() blocks each caller process until all other processes have reached the same barrier.

The barrier function is only used for synchronization purposes and does not exchange any data between the processes.

3

ENERGY EFFICIENT HPC ON EMBEDDED SOCS

In order to improve energy efficiency, academia and industry have been studying the suitability of low-power embedded technologies for high performance computing. Although state-of-the art embedded SoCs include GPUs that could deliver significant performance and energy improvements, until now, the HPC capabilities of such components have not been examined.

This chapter explores whether embedded GPUs can provide energy and performance advantages over the embedded CPUs, and emphasizes the importance of OpenCL software optimizations.

In Section 3.2 we provide an overview of related work. Section 3.3 describes the ARM Mali T-604 GPU Architecture followed by the OpenCL optimization techniques. In Section 3.4 we evaluate and present the results of our experiments. Finally, Section 3.5 summarizes and concludes our findings.

3.1 INTRODUCTION

The high performance computing community has recognized GPUs as powerful parallel computing entities, well suited for computationally demanding applications with extensive data-level parallelism [69]. A broad range of computational algorithms from different HPC domains have been successfully ported to GPUs, and heterogeneous clusters have shown distinct performance and energy-efficiency improvements over their homogeneous counterparts [15, 44].

On the other hand, in order to improve energy efficiency, the community has been deploying large-scale HPC clusters using SoCs [100, 36, 51, 52, 82, 81]. Although state-of-the-art SoCs do integrate GPUs that could provide significant improvements in terms of performance and energy efficiency, up to now, the embedded GPUs have not been used for high performance computing. The main reason behind this is that the GPUs integrated in SoCs did not support parallel program-

ming models such as OpenCL or CUDA, and neither did they support 64-bit floating-point precision.

Nowadays the situation is changing, and the latest embedded GPUs satisfy 64-bit floating-point arithmetic precision constraints, and support computational languages such as OpenCL or CUDA [80, 114, 96]. Aware of this upcoming trend, we analyze the use of the ARM Mali GPU Compute Architecture for HPC workloads. The contributions which will be presented in this chapter are as follows:

- First, we investigated the possibility of using embedded GPUs for high performance computing. We successfully ported nine HPC benchmarks to OpenCL and executed them on ARM Mali-T604 GPU [80] – the first embedded GPU with OpenCL Full Profile support. We evaluated performance, power consumption and energy-to-solution of the benchmarks and compared the results with an embedded CPU composed of ARM Cortex-A15 cores.

- Second, we identified the important OpenCL software optimization techniques for efficient utilization of the ARM Mali GPU Compute Architecture. This architecture differs in many aspects from high-end GPU architectures and OpenCL code must be optimized taking into account the particular characteristics of the underlying hardware.

- Finally, we deployed the proposed Mali GPU optimization techniques on nine HPC benchmarks and we evaluated the impact of the optimization techniques on system performance and energy consumption.

3.2 RELATED WORK

Recently, embedded GPU computing has drawn a lot of attention from the research community and the industry. In the embedded world, the primary graphics programming interface is OpenGL ES [61]. OpenGL ES is a subset of the well-known OpenGL 3D graphics application programming interface (API). Unlike the standard used in desktop systems and gaming consoles the Embedded Systems version removes some of the functionalities and at the same time extends

it to better support the computational abilities of embedded GPUs. Although the OpenGL ES API is mainly designed for graphics, some recent studies have successfully investigated the acceleration of algorithms in embedded devices.

Singhal et al. [109, 110] explored the implementation, optimization, and evaluation of image processing and computer vision algorithms. Cheng et al. [25] investigated the computing power and energy consumption of an embedded CPU-GPU platform for computer vision applications. Lopez et al. [74] presented the first embedded GPU implementation of Local Binary Pattern feature extraction. Rister et al. [102] implemented an efficient implementation of the Scale-Invariant Feature Transform (SIFT) feature detection algorithm. Due to the difficulty in mapping algorithms to graphics operations and accessing the unexposed low-level hardware characteristics, all the previously mentioned works are specific to the field of image analysis and processing, leaving out a wide range of possible applications.

With the growth of the Open Computing Language (OpenCL) [60] as an open standard for general-purpose parallel programming of heterogeneous compute node, also the embedded community has become interested in simplified models for embedded GPU computing. Leskelä et al. [70] created a programming environment based on the standard OpenCL Embedded Profile and tested their embedded CPU-GPU backends implementation against an image processing workload. Wang et al. [122] studied the performance of an exemplar-based inpainting algorithm on the Qualcomm Snapdragon S4 chipset which supports the OpenCL Embedded Profile for both CPU and GPU [97]. Although these studies are using OpenCL, they are still restricted to the image processing area and do not investigate whether non-graphics workloads may benefit from embedded GPUs usage.

Other works have investigated how to simulate and evaluate GPUs present in embedded devices. TEAPOT [8] provides full-system cycle-accurate GPU simulation and power model for embedded workloads. The system has been used to analyze techniques that trade image quality for energy saving [8], exploit similarity between consecutive frames to save memory bandwidth [7] and hide memory latency using a decoupled access/execute paradigm [6]. Maghazeh et al. [77] investigated if a heterogeneous embedded platform with GPUs and CPUs can be an attractive solution for different kinds of applications.

They implemented five benchmarks from different application domains and performed experiments on an i.MX6 Sabre Lite development board with OpenCL Embedded Profile support.

3.3 ARM MALI T-604 GPU ARCHITECTURE

While the main focus of embedded GPUs is mobile 2D and 3D high-quality graphics, both industry and academia are becoming interested in the general purpose compute capabilities of such devices. The first effort in this direction from ARM Holdings is the Mali-T600 GPU Midgard series. The Mali GPU Compute Architecture is designed to fully comply with the OpenCL Full Profile which has strict requirements for precision and support for mathematical functions. The first GPU based on this architecture is the Mali-T604 [80]. In addition to satisfying IEEE-754-2008 precision requirements for single and double-precision floating point, Mali-T604 natively supports 64-bit integer data types and implements barriers and atomics in hardware.

Figure 3.1 depicts the architectural details of the Mali-T604 GPU. The GPU supports up to four shader cores with two arithmetic pipes per core. The *Job Scheduler*, implemented in hardware, abstracts the GPU core configuration from the driver and distributes the computational tasks to maximize the GPU resource utilization. The *Memory Management Unit* can easily map memory from the CPU's address space into the GPU's address space to quickly copy or share information. The level 2 cache is shared between the shader cores and maintained coherent by the *Snoop Control Unit*.

3.3.1 *OpenCL Optimizations for Mali GPU*

OpenCL is a well designed language that allows access to the compute power of heterogeneous devices from a single multi-platform source code. Although the portability at code level ensures the execution of programs in all the devices that support the language, the same portability is not present from a performance perspective [120]. Currently, the ARM Mali GPU Architecture, as depicted in Figure 3.1, differs in many aspects from the high-end GPU architectures used

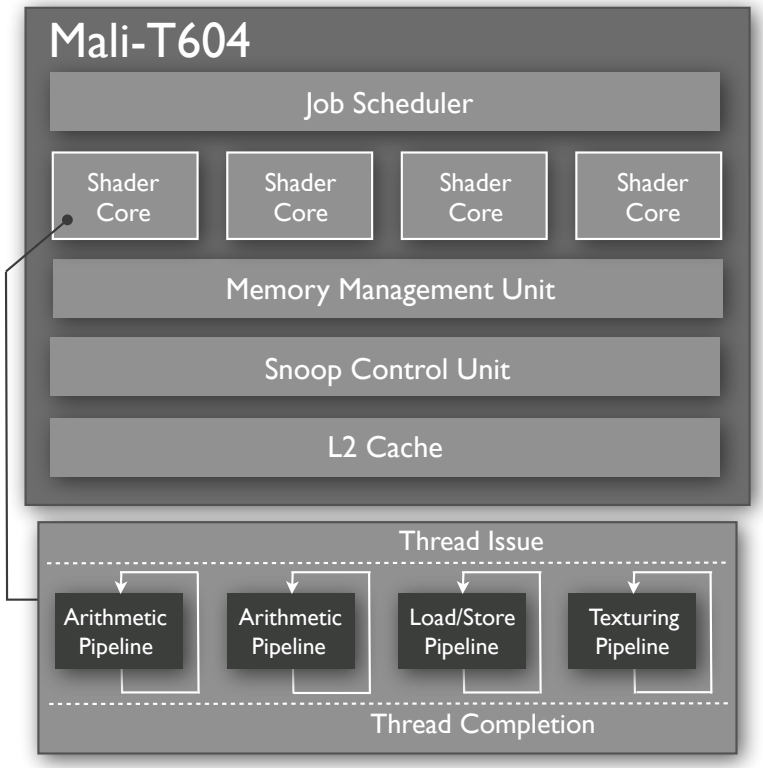


Figure 3.1: ARM Mali-T604 GPU Architecture

in desktops and servers. Under this consideration, the OpenCL code must be optimized taking into account the particular characteristics of the hardware.

Below we analyze and explain the OpenCL software optimization techniques that enable the ARM compiler to efficiently map the program to the specific ARM Mali GPU Compute Architecture.

Memory allocation and mapping. OpenCL is typically executed in desktop and server systems where the application processor and the GPU have separate memories and copy operations have to be used to exchange data between the host and the device. Differently, the Mali GPU Architecture provides a unified memory system where traditional copy operations are not required. Although from a hardware point of view, time and power consuming data transfers can be avoided, the Mali GPU cannot access memory buffers created with the standard *malloc* function because they are not mapped into the GPU memory space. To allow the GPU to access the data, OpenCL buffer objects have to be created using the *CreateBuffer* function with the *CL_MEM_USE_HOST_PTR* flag and data has to be copied by

the host processor with the *EnqueueWriteBuffer* and *EnqueueReadBuffer* OpenCL functions. Although this method enables data exchange between host and device, it does not solve the additional copy issue. To eliminate all the computationally expensive copies it is necessary to allocate the memory through the OpenCL API and to use memory mapping functions. Buffers have to be allocated using the *CreateBuffer* function with the *CL_MEM_ALLOC_HOST_PTR* flag and the *EnqueueMapBuffer* and *EnqueueUnmapMemObject* have to be used to enable both the application processor and the Mali GPU to access the data.

Load distribution. Another important aspect of an OpenCL program is the load distribution between the processing elements of a device. As already described in Section 2.3.2, the API offers a function called *EnqueueKernel* to execute a kernel on a device. To allow control over the load distribution some arguments of the function describe the dimensionality of the data, the number of work-items to be processed for each dimension (*global_size*) and the number of work-items in a work-group for each dimension (*local_size*).

As suggested in the official Mali OpenCL Developer Guide [79] the optimal *global_size* can be calculated as the device *maximum_work-group_size* multiplied by the number of shader cores multiplied by a constant. This constant for the Mali-T604 is four or eight. More generally, the *global_size* must be in the order of several thousand to maximize the GPU resources utilization and to achieve high performance. With regard to the local work-group size, if the application is not required to share data among work-items, the Developer Guide [79] suggests to set the *local_size* parameter to NULL and let the OpenCL driver determine the most efficient *work-group_size* for the kernel. Although this practice simplifies the selection of the *work-group_size* value, during our experimental evaluation we noticed some performance degradation and we strongly suggest to manually tune the *local_size* parameter.

Memory Spaces. Defined as a generic language for heterogeneous computing, OpenCL exposes a memory model composed of several memory spaces that every vendor maps differently to the corresponding available hardware. Usually dedicated GPUs present local memories with much higher bandwidth and lower latency than the GPU memory unit. The OpenCL implementations offered by the two ven-

dors map the local memory space to the local memories making the exploitation of locality at code level one of the most important factors in achieving high performance. Differently, Mali GPUs have a unified memory system where local memory is physically mapped to the global memory. For this reason, traditional code locality optimizations are not required, simplifying the coding of optimized OpenCL kernels.

Thread Divergence. Other optimizations usually used in GPU computing are those regarding warp or wavefront execution. Warps or wavefronts represent the smallest executable units of parallelism on NVIDIA or AMD devices (respectively 32 or 64 work-items). This means that if two work-items inside a warp or a wavefront take divergent paths of the control flow, all work-items of the same unit go through both paths. This issue is called thread divergence and can significantly affect the instruction throughput of the computational kernel. The thread divergence problem is not present in the ARM Mali GPU Architecture because the smallest unit of parallelism is a single thread (work-item) that, being independent, cannot diverge.

Vectorization. One of the most important hardware characteristic to take into account during the development of OpenCL code for the ARM Mali GPU Architecture is that the shader cores contain 128-bit wide vector registers. To allow the effective programming of such units, OpenCL offers vector load and store instructions and vector types of different sizes. Vectorization of an OpenCL scalar code can be done by converting the scalar data types (e.g. float) to vector data type (e.g. float4) increasing the number of elements processed by each work-item and consequently reducing the *global_size* in the host code. This optimization not only promotes the use of hardware resources, but also allows a reduction of the run-time scheduling overheads due to the decrease in the number of work-groups.

Vector Sizes. OpenCL, as already mentioned, provides vector types of different sizes that the compiler must map to the corresponding hardware vector registers. In case of the ARM Mali-T604, we noticed that, after vectorizing the kernel, the best achievable performance is not bound to a particular vector size but can vary from case to case. Using types wider than the underlying hardware can improve the instruction-level scheduling, but also increase register pressure. For this reason, we suggest, whenever the code allows it, to experiment

with different vector sizes (e.g. size of 4, 8, 16). It is also important to keep in mind that vector load and store operations access multiple data elements in memory with a single instruction leading to a more efficient use of the available bandwidth. For this reason, such operations should be also used in kernels that do not take advantage of vector registers but process each element of the vector array individually.

Loop Unrolling. Another opportunity for vectorization of the kernel code is within loops. A loop can be unrolled and multiple instructions can be replaced with a single vector instruction which operates on multiple data elements. Although this optimization usually leads to performance improvements on relatively long loops, in case the number of iterations is not a perfect multiple of the vector size, the overhead due to the correct handling of the last iterations of the loop has to be considered. The best performance for different kernel codes can be achieved using vectorization and unrolling, taking into consideration that code replication can also lead to performance degradation.

Data Organization. In vector architectures, the organization of data is of primary importance. Typically in application code data is packed in an Array of Structures (AOS). If we consider a set of three dimensional points, their representation would be an array in which each element is a structure with x, y, z coordinates. Although this representation is the most natural, it typically requires multiple load/shuffle/insert or gather instructions to correctly populate the vector registers. A more efficient data-packing approach is Structure Of Arrays (SOA). In this representation, the data types are the same across the vector. With this approach, the list of points would be organized as three arrays containing respectively the x, y and z values that would facilitate the application of vector instructions increasing the code performance.

Directives and Type Qualifiers. Better performance can also be achieved by passing additional information to the OpenCL compiler through the use of directives and type qualifiers. With function inlining the user can increase the size of basic blocks (sequences of consecutive instructions without branches) and eliminate the overhead associated with the function call. The use of the *const* keyword in pointer declarations makes possible to declare pointers to constant

data and allows the compiler to make more assumptions and therefore produce significant optimizations. The restrict qualifier, applicable to the arguments of a kernel, enables the compiler to assume that the pointer is the only way to access the object to which it points, limiting the effects of pointer aliasing.

3.4 EVALUATION

In order to evaluate the Mali-T604 GPU and the OpenCL optimizations techniques presented in Section 3.3.1, we used a set of nine HPC benchmarks. These benchmarks cover a wide range of algorithms employed in HPC applications and stress various architectural features. They have been already used in previous studies [99, 101] to evaluate the suitability of embedded platforms for HPC systems. Below, we briefly describe each of the benchmarks.

Sparse Vector-Matrix Multiplication (spvm) benchmark multiplies a vector and a sparse matrix to produce a new vector. It is useful as metric to measure performance in cases of load imbalance.

Vector Operation (vecop) benchmark performs an addition of two vectors in an element-by-element basis. Given the memory-bound nature of the kernel, this benchmark stresses the memory bandwidth of the platform under study.

Histogram (hist) benchmark computes the histogram of the values present in a vector using a configurable bucket size. It uses local privatization that requires a reduction stage which can become a bottleneck on highly parallel architectures.

3D Stencil (3dstc) benchmark produces a new 3D volume from an input 3D volume. Each point of the output is a linear combination of the point with the same coordinates in the input and the neighboring points on each dimension. This benchmark is useful to evaluate the performance in presence of memory accesses with regular strides.

Reduction (red) benchmark applies the addition operator to produce a single (scalar) output value from an input vector. Reduction is a common operation in many computational kernels and allows to measure the capability of the compute accelerator to adapt from massively parallel computation stages to almost sequential execution.

Atomic Monte-Carlo Dynamics (amcd) benchmark performs a number of independent simulations using the Markov Chain Monte Carlo method. Initial atom coordinates are provided and a number of randomly chosen displacements are applied to randomly selected atoms which are accepted or rejected using the Metropolis method.

N-Body (nbody) benchmark takes as input, a list of bodies described with a set of parameters (position, mass, initial velocity) and updates their information after a given simulated time period based on gravitational interference between each body.

2D Convolution (2dcon) benchmark produces a new matrix from an input matrix of the same size. Each point of the output is a linear combination of the point with the same coordinates in the input and the neighboring points. Differently from the 3D stencil computation, neighboring points can include points with the same coordinates as the input point plus/minus an offset in one or two dimensions. This benchmark is useful to evaluate the performance in presence of spatial locality and strided memory accesses.

Dense matrix-matrix Multiplication (dmmm) benchmark performs the multiplication of two dense input matrices. Matrix multiplication is a common computation in many numerical simulations and measures the ability of the compute accelerator to exploit data reuse and compute performance.

Each benchmark was implemented in four different versions: *Serial*, *OpenMP*, *OpenCL*, and *OpenCL Opt*. The *Serial* benchmarks were designed to execute on a single core. The *OpenMP* benchmarks, through the use of threads, were designed to execute in parallel on several CPU cores. The *Serial* and the *OpenMP* versions have been already developed and used by previous studies [99, 101] that tested the suitability of SoCs for high performance computing. These versions do not make use of vector instructions. This is due to the fact that the ARM Cortex-A15 CPU does not incorporate a double-precision SIMD unit and full IEEE-754-2008 floating-point vector support. The *OpenCL* versions of the benchmarks were developed in the Open Computing Language to allow a parallel execution on the GPU.

Recently, data structures and data layout transformations have been proposed to efficiently implement some of the algorithms used in our benchmarks [17, 73, 45]. However, in order to maintain a similar code base for all CPU and GPU implementations, we do not take advan-

tage of them. One of the main goals of the study is to quantify the performance and energy improvements of the proposed OpenCL optimization techniques. In this regard, we enhanced the *OpenCL* benchmarks following ***only*** the guidelines presented in Section 3.3.1. The new versions of the benchmarks are referred to as *OpenCL Opt*.

To avoid rewriting the host code with different data transfer techniques, both, *OpenCL* and *OpenCL Opt* benchmarks make use of the host memory mapping, exploiting the unified memory system offered by the architecture. Due to the problem of data transfer already being well known in the heterogeneous computing field of research [41], we want to place more emphasis on other important optimization aspects during the comparison between the *OpenCL* and *OpenCL Opt* versions of the benchmarks.

The experiments were executed on Samsung Exynos 5 Dual Arndale Board [105]. The board comprises the Samsung Exynos 5250 embedded system-on-chip (SoC) and is equipped with 2 GB of DDR3L-1600 memory. The Samsung Exynos 5250 integrates a dual-core ARM Cortex-A15, running at 1.7 GHz with 32 KB of private L1 instruction and data cache, and 1 MB of shared L2 cache. Alongside the CPU, the SoC features a four-core ARM Mali-T604 GPU whose architecture has been already described in Section 3.3.

The operating system running on the platform is Ubuntu 11.10 with a Linux kernel version 3.0.31. The operating system image includes the driver needed for executing OpenCL programs on the Mali-T604 GPU. The benchmarks were compiled with GCC version 4.6.1 with $-$O3 optimization flag. Even if the CPU floating-point hardware includes the NEON extension, floating-point vector operations are not automatically generated by GCC's auto-vectorization pass and we did not make use of the *-funsafe-math-optimizations* flag to enable them.

In all the experiments, the problem size for all the four versions of the same benchmark is maintained constant, so that each version has the same amount of work to perform. The measurements were collected only in the parallel regions of the benchmarks, excluding the initialization and finalization phases. We adjusted the number of iterations of the considered regions so that each benchmark runs for long enough to get an accurate energy consumption figure. We repeated each experiment 20 times and we computed the mean value

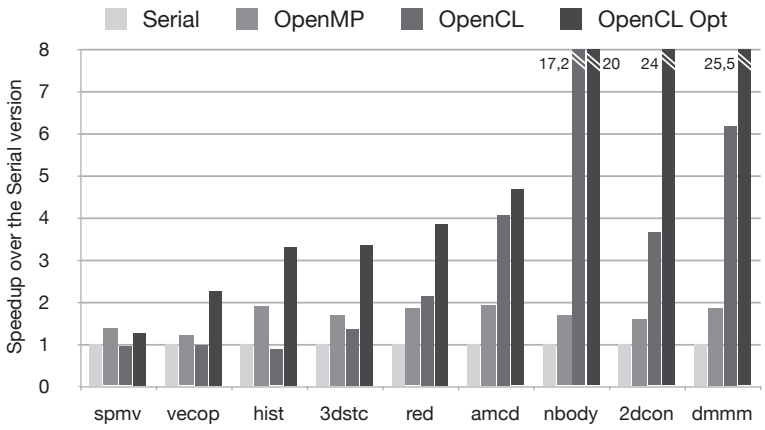
and the standard deviation of the measured performance and power consumption. In all the presented experiments, the standard deviation is negligible, thus we do not report it. The power consumption of the board was measured with the Yokogawa WT230 power meter that offers a sampling frequency of 10 Hz with 0.1% accuracy.

3.4.1 *Performance Analysis*

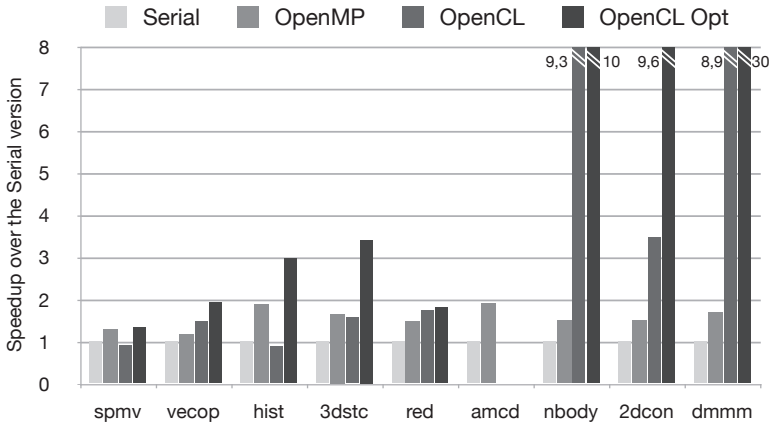
In Figure 3.2, we present the performance comparison between different versions of the benchmarks. The benchmarks under study are listed along the X axis of the figure, while the Y axis shows the speedup relative to the *Serial* version of the code. We present the result for two sets of experiments, in single-precision and double-precision.

Single-precision: Single-precision results, depicted in Figure 3.2a, show that the speedup of the parallel *OpenMP* benchmarks varies between 1.2× and 1.9× with an average of 1.7×. Regarding the *OpenCL* version, three out of the nine benchmarks (*spmv*, *vecop*, *hist*) experience performance degradation with respect to the *Serial* code. *OpenCL* version of *3dstc* benchmark experiences a 1.4× speedup over the *Serial* version, however, the performance is lower than the *OpenMP* variant. For *red*, *amcd*, and *2dcon* benchmarks, the improvement over the *Serial* version is 2.1×, 4.1×, and 3.6×, respectively, outperforming the *OpenMP* version. Finally, *dmmm* and *nbody* benchmarks experience speedup of 6.2× and 17.2×. From these results, we can conclude that porting code to OpenCL and running on the GPU, on its own, does not guarantee significant performance improvement, and that a multithreaded *OpenMP* version may lead to better performance.

Significantly different results were obtained on applying optimization techniques to the OpenCL code (*OpenCL Opt*). Sparse Vector-Matrix Multiplication (*spmv*) is the only application that does not perform well, but, we still detect some performance improvement over the *Serial* implementation (1.25× speedup). Four of the nine benchmarks (*vecop*, *hist*, *3dstc*, *red*) show a speedup between 2× and 4×. Benchmark *amcd* experiences speedup of 4.7×, while benchmarks



(a) Single-precision



(b) Double-precision

Figure 3.2: Performance results on the Exynos SoC

nbody, *2dcon*, and *dmmm* show improvement of 20×, 24× and 25.5×, respectively.

The significant difference in performance improvement for different *OpenCL* and *OpenCL Opt* benchmarks can be explained by analyzing the benchmarks' characteristics. Each of the benchmarks under study requires a certain amount of data from main memory. Therefore, in absence of sufficient computation, the memory bandwidth can limit the performance. Sparse Vector-Matrix Multiplication (*spmv*) and Vector Operation (*vecop*) with large working sets and little computation are good examples of this scenario. As already mentioned in Section 3.4, our OpenCL versions do not take advantage of special

data structures and for this reason, *spmv* can only partially exploit the available bandwidth.

Histogram (*hist*) makes use of atomic operations supported at hardware level to compute the result and shows good performance compared to the *Serial* implementation. 3d Stencil (*3dstc*) does not take advantage of vector instruction and limits the optimizations to work-group size tuning and data reuse. Reduction (*red*) makes use of a two-stage reduction, that performs a constant number of parallel reductions based on the number of used work-groups. The main difference in performance between *OpenCL* and *OpenCL Opt* for this benchmark is due to the vectorization and the use of a tuned work-group size. The *OpenCL* version of *amcd* without optimizations can already reach a speedup of 4.1×. We did not find many hot spots for optimizations and the *OpenCL Opt* is only slightly faster.

The last three applications can reach significant speedups (up to 25.5×) over the *Serial* implementations. The *OpenCL* version of N-Body (*nbody*) does not apply any change to the main data structure representation that would lead to an easier applicability of vector optimizations. For this reason, the *OpenCL Opt* version does not show significant improvements over the non-optimized version. Differently, 2D Convolution (*2dcon*) and Dense matrix-matrix Multiplication (*dmmm*) provide extensive parallelism at both vector and thread level. In these cases, most of the optimizations can be successfully applied (loop unrolling, vectorization, group-size and vector-size tuning) leading to a considerable increase in performance.

Double-precision: Double-precision results are presented in Figure 3.2b. The Atomic Monte-Carlo Dynamics (*amcd*) *OpenCL* versions are not presented due to a compiler issue that does not allow the correct termination of the compilation phase for the *OpenCL* kernel in double precision. Since the compiler source code is not publicly available, we could not solve the problem. The problem is reported, and it will be corrected in a future version of the compiler. For the remaining benchmarks, we detect similar trends as for single-precision.

For the *OpenCL* version, two of the eight benchmarks (*spmv*, *hist*) show lower performance than the *Serial* version. *OpenCL* version of *3dstc* benchmark experiences a 1.6× speedup over the *Serial* version, however, it still experiences lower performance than the *OpenMP* vari-

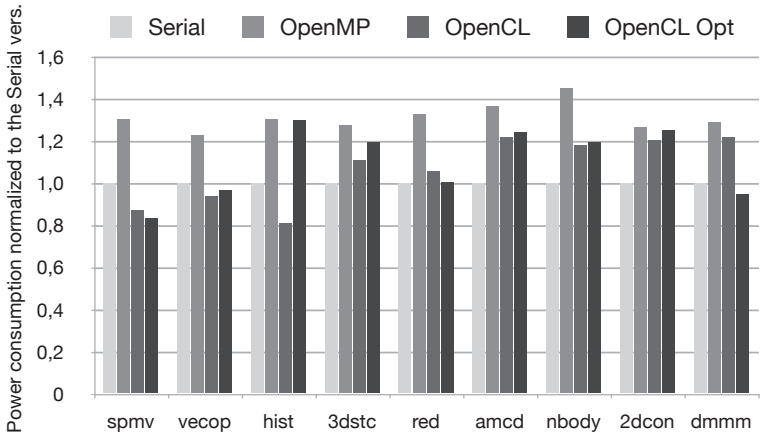
ant. Benchmarks *vecop* and *red* show 1.5× and 1.7× speedup over the *Serial* version of the benchmarks, and negligible performance improvement over the *OpenMP* variant. Finally, three benchmarks experience a significant speedup: *2dcon* (3.5×), *dmmm* (8.9×) and *nbody* (9.3×). Regarding the *OpenCL Opt* version, three of the eight benchmarks (*spmv*, *vecop*, *red*) show a performance improvement below 2×. Benchmarks *hist* and *3dstc*, experience a speedup of 3× and 3.4×. Finally, *2dcon*, *nbody*, *dmmm* benchmarks show speedup is 9.6×, 10×, and 30×, respectively.

The speedup of *OpenCL Opt* over the *OpenCL* benchmarks in single-precision (Figure 3.2a) and double-precision (Figure 3.2b) is comparable. However, for the *red*, *nbody* and *2dcon* benchmarks, we have detected a significantly different trend. The reduction of the performance gap between the *OpenCL* and *OpenCL Opt* version of benchmarks *nbody* and *2dcon* in double-precision is due the failure of optimized version of the kernels (with *CL_OUT_OF_RESOURCES* Error). The benchmarks were executed without the error in single-precision.

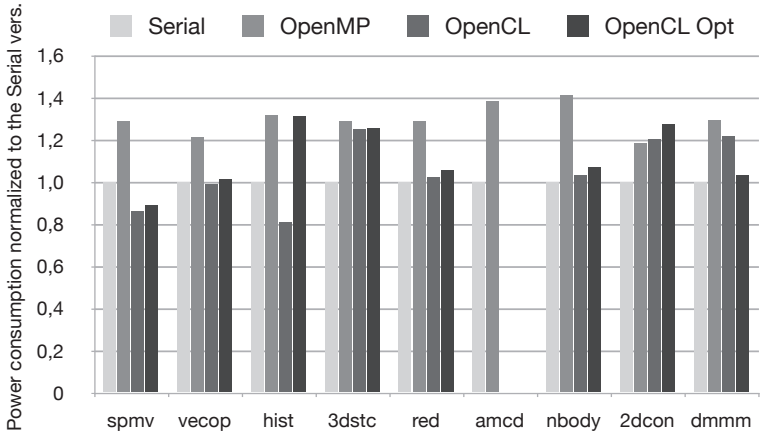
3.4.2 *Power Analysis*

In Figure 3.3, we present the comparison between power consumption of *Serial*, *OpenMP*, *OpenCL* and *OpenCL Opt* versions of the benchmarks. The benchmarks under study are listed along the X axis of the figure, while the Y axis shows the power consumption relative to the *Serial* version of the code. We present the result for two sets of experiments, in single-precision and double-precision.

Single-precision: Single-precision results are presented in Figure 3.3a. The increase in power consumption of the *OpenMP* benchmarks over the *Serial* version varies between 23% (*vecop*) and 45% (*nbody*) with an average of 31%. Results vary insignificantly between *OpenCL* and *Serial* versions of the benchmarks. For *spmv*, *vecop*, and *hist*, the power consumption of the *OpenCL* version is 13%, 7%, and 19% decreased with respect to the *Serial* version. The remaining benchmarks have a slightly higher power consumption of up to 22% (*amcd* and *dmmm* benchmarks). On average, *OpenCL* benchmarks show only 7% higher power consumption than the *Serial* version. Seven out of



(a) Single-precision



(b) Double-precision

Figure 3.3: Power consumption results on the Exynos SoC

nine *OpenCL Opt* benchmarks have power consumption that is very similar to the consumption of the corresponding *OpenCL* benchmarks. Significant difference is detected only for *hist* benchmark that experienced significant power increase with respect to *OpenCL* version, and *dmmm* benchmark that showed significant power reduction.

Double-precision: Double-precision results are presented in Figure 3.3b and follow similar trends as the single-precision results.

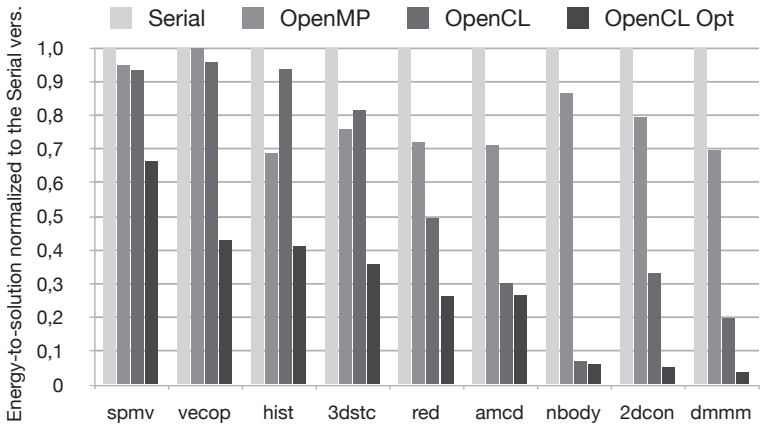
3.4.3 *Energy-to-Solution Analysis*

Figure 3.4 shows the energy-to-solution of the benchmarks in single-precision and double-precision. The results presented are normalized with respect to the energy-to-solution of the *Serial* implementation of the benchmarks.

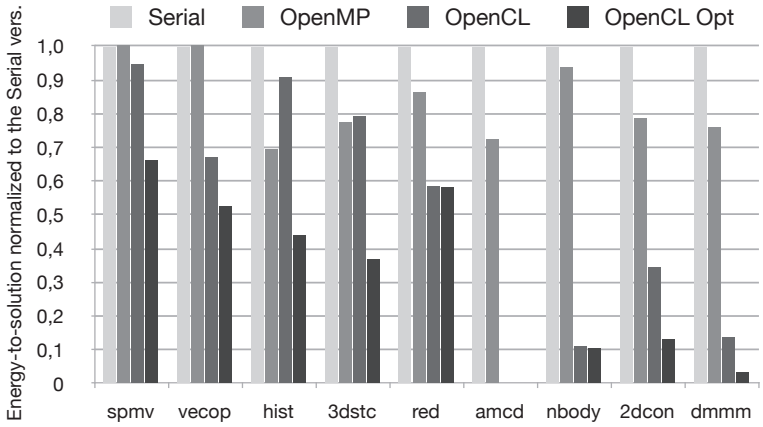
Single-precision: Single-precision results are shown in Figure 3.4a. Energy-to-solution reduction of the *OpenMP* benchmarks over the *Serial* version is 20% on average. The results for the *OpenCL* versions of the benchmarks can be clustered in two groups. Benchmarks *spmv*, *vecop*, *hist* and *3dstc* experience moderate energy-to-solution improvement (less than 20%) over the *Serial* code. Actually, the *OpenCL* versions of *hist* and *3dstc* consume more energy than corresponding *OpenMP* implementations. For the remaining five *OpenCL* benchmarks (*red*, *amcd*, *nbody*, *2dcon*, *dmmm*), energy-to-solution improvement (reduction) is significant, and it varies between 51% (*red*) and 93% (*nbody*). Finally, for all the benchmarks under study, the *OpenCL Opt* versions experience the lowest energy-to-solution. Improvement with respect to the *Serial* version of the benchmarks varies between 34% (*spmv*) and 96% (*dmmm*) with an average of 72%.

For all the benchmarks under study, the *OpenCL Opt* versions have better energy-to-solution than the corresponding non-optimized implementations. For seven out of nine benchmarks (*spmv*, *vecop*, *hist*, *3dstc*, *red*, *2dcon*, and *dmmm*), the energy consumption improvement due to proposed OpenCL optimization is significant. On average, the *OpenCL Opt* benchmarks require only 28% of the energy consumed by the *Serial* implementation, while non-optimized *OpenCL* implementations require 56%.

Double-precision: Double-precision energy-to-solution results are presented in Figure 3.4b. The results follow similar trends as the single-precision results. The only significant difference is detected for *red OpenCL Opt* benchmark. In this case, energy-to-solution of double-precision increased significantly with respect to the single-precision version. This is due to the relative performance loss of *OpenCL Opt red* benchmark in double-precision with respect to single-precision (see



(a) Single-precision



(b) Double-precision

Figure 3.4: Energy-to-solution results on the Exynos SoC

Figure 3.2). On average, the *OpenCL Opt* benchmarks require only 36% of the energy consumed by the *Serial* implementation, while non-optimized *OpenCL* implementations require 56%.

3.5 SUMMARY

We analyzed the suitability of the ARM Mali GPU Compute Architecture for HPC workloads. We successfully ported nine HPC benchmarks to OpenCL and executed them on the ARM Mali-T604 GPU – the first embedded GPU with OpenCL Full Profile Support. It is important to emphasize that, despite much higher theoretical peak performance, the ARM Mali-T604 GPU usually cannot be effectively

exploited with the creation of a simple OpenCL parallel version from the sequential code. Based on the presented results, we see that achieving high performance requires proper utilization of the underlying architecture that can only be obtained through the use of code optimizations and parameter tuning. The OpenCL optimization techniques that we presented resulted in a significant performance increase for most of the benchmarks under study. On the other hand, we showed that power consumption varies insignificantly between optimized and non-optimized versions of the OpenCL benchmarks. Significant performance improvement and similar power consumption translate into significantly lower energy-to-solution of the optimized OpenCL benchmarks. Our results also show that the Mali-T604 GPU provides distinct improvements in terms of performance and energy-to-solution over ARM Cortex-A15. On average, single-precision and double-precision GPU benchmarks achieve a speedup of 8.7× over the benchmarks running on a single Cortex-A15 core, while consuming only 32% of the energy. Our study confirms that, in high performance computing, embedded GPUs can offer performance and energy advantages over embedded CPUs, similar to their high-end counterparts present in clusters.

4

SCHEDULING TECHNIQUES FOR HETEROGENEOUS NODES

The transition from homogeneous to heterogeneous compute nodes is challenging with respect to the efficient utilization of the hardware resources and the reuse of the software stack. As heterogeneous computing opens many new opportunities for developing fast parallel algorithms, it also introduces additional levels of complexity. One of the main challenges, in order to maximize the system performance, is the partitioning and scheduling of tasks among the available compute devices.

As previously described in Section 2.3.4, the OpenCL language is based on a task-parallel model, in which each kernel task is data parallel. Data-parallel tasks can often be split into smaller sub-tasks and distributed across multiple devices. However, finding an efficient partitioning is not trivial. The best performing partitioning is likely to change with different applications, different (input) problem sizes, and different hardware configurations. Furthermore, as pointed out by other studies [42], dynamic scheduling approaches may not lead to an optimal solution, due to the large difference in performance and transfer bandwidth of the single devices.

In this chapter we investigate different techniques to tackle the partitioning and scheduling of tasks with a specific focus on heterogeneous compute nodes.

Section 4.1 describes an automatic, problem size sensitive method for partitioning OpenCL tasks, while Section 4.2 describes two low-complexity dynamic heuristics for the scheduling of OpenCL independent tasks. Finally, Section 4.3 summarizes and concludes our findings.

4.1 AUTOMATIC TASK PARTITIONING ON HETEROGENEOUS NODES

In this section, we present a novel approach which automatizes task partitioning of OpenCL programs on heterogeneous compute nodes.

Our work is based on machine learning which combines compile time analysis with runtime feature evaluation to predict an effective task partitioning.

The contributions are as follows:

- We developed a new compiler approach for converting a single-device OpenCL program to a multi-device OpenCL program.
- We extended the Insieme runtime system to support the parallel execution of OpenCL code across multiple heterogeneous devices.
- We optimized the task partitioning of OpenCL programs on heterogeneous compute nodes with an off-line machine learning generated problem size sensitive model.
- We empirically demonstrated the benefits of our approach compared to traditional static task partitioning techniques using 23 different applications on two different heterogeneous multi-device nodes.

This work was developed with the collaboration of Klaus Kofler. The analysis, the compiler back-end for the generation of the code, the runtime system, and the Support Vector Machine model are my personal contributions.

4.1.1 *Related Work*

In recent years, heterogeneous systems have received great attention from the research community. Several projects [113, 5, 12, 111, 64, 65] mainly focused on OpenMP, CUDA, and OpenCL extensions, have investigated how to facilitate the programming of heterogeneous clusters. Our work, while following the same idea, is focused on automatic management of multiple devices in a single compute node. A similar study was done by Chen et al. [24]. The authors introduce an automatic parallelization process to use multiple GPUs. This work targets mainly the analysis of access patterns for data decomposition, showing that many applications can be parallelized automatically. Our approach, based on a similar analysis, not only derives the

data partition schemes but also provides a solution for optimal task partitioning on heterogeneous devices.

In a different perspective, much work has been done to address the scheduling of tasks in heterogeneous compute nodes. Several frameworks [113, 10, 71] have been created to support the developer in the use of all available computing resources of a heterogeneous compute node. Although these studies propose several possible solutions to the problem, they are mostly based on performance estimations provided by the user. On the contrary, our approach is automatic and does not require any additional user-supplied information. Furthermore, these approaches focus on optimizing the scheduling of multiple available tasks, assuming that several parallel tasks are available. Our system is designed to optimize the execution of a single task and can, therefore, optimize also programs with a single task.

Other works have investigated the problem of automatic task partitioning. Luk et al. [75] introduced an adaptive mapping approach based on a regression model. Their system considers every first run of a program as a training run that can be then used to determine the computation-to-processor mapping for the same program with a new input problem. This approach expects that a program is trained once and then used many times afterward. In contrast to our work, they only show results of one target architecture equipped with only one CPU and one GPU.

A similar approach was adopted by Kai et al. [58]. They proposed a holistic energy management framework for heterogeneous nodes which dynamically splits and distributes the workload over GPU and CPU based on the observed performance. Their algorithm dynamically adjusts the task partitioning based on the runtime difference between devices. Our approach, on the other hand, does not require any profiling or training runs of the program to optimize it. We can derive an optimized task partitioning during the first run of a new program by using a previously, off-line trained model.

Hong et al. [47] proposed MapCG, a framework that supports source code level portability between CPU and GPU. By incorporating a MapReduce programming model, a program can be compiled and executed on either CPUs or GPUs without modification. However, they observed that CPU/GPU combinations did not yield significant performance improvement for the 8 test cases they examined. In contrast

to this work, on our target architectures, we observed the important role of the hybrid task partitioning to achieve the best performance for our test cases.

Grewe et al. [42] developed a purely static task partitioning approach based on predictive modeling and program features. Starting from a multi-device OpenCL code, the authors predict the partitioning of a task with a machine learning model based on static features analysis for fixed problem sizes. Our work uses a similar machine learning approach but combines static program features detected at compile time with dynamic features collected at runtime that allow the adaptation of the task partitioning to different problem sizes. We test our approach for different target architectures emphasizing the importance of the problem size and the hardware configuration for the tuning of the task partitioning. Furthermore, our system is not limited to a CPU-GPU configuration but can handle an arbitrary number of heterogeneous devices in a single compute node.

4.1.2 *The Insieme Compiler-Runtime Framework*

Heterogeneous compute nodes are difficult to program, and moreover the performance capability of individual devices can vary significantly across different applications and problem sizes which often makes static, problem size insensitive distribution techniques unsuitable. Our work based on the Insieme Compiler and Runtime framework relieves the developer from this difficult task. It consists of a source-to-source compiler that translates single-device OpenCL programs into multi-device OpenCL programs that the runtime system executes among the available heterogeneous devices. Using such a framework we proposed a machine learning approach that, based on analysis of the program structure and input data, try to predict the optimal partitioning for an OpenCL program *a priori*. Our approach is composed of two main phases: *training* and *deployment*. The goal of the training phase, depicted in Figure 4.1, is to build a task partitioning prediction model. In step ❶, a set of OpenCL programs is provided to the system and translated into the Insieme parallel intermediate representation [57] by the code analyzer. This IR offers a formal and compact representation of programs that facilitates code analysis

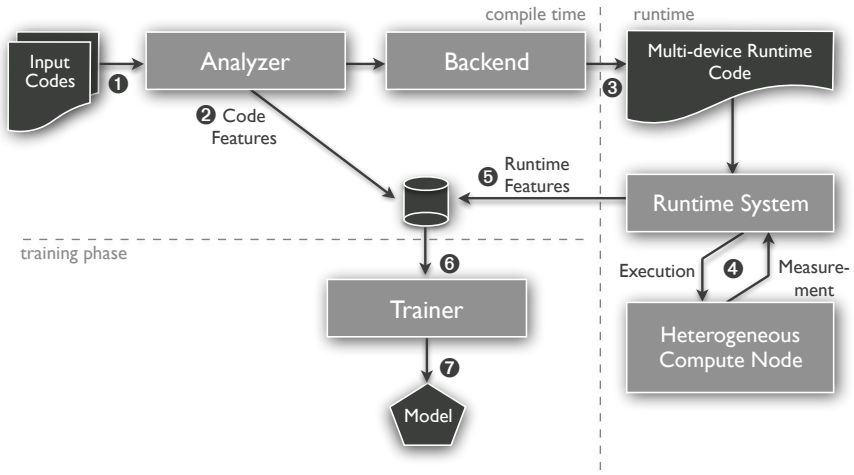


Figure 4.1: Training phase

and transformation. In step ❷, from this representation, the features of the program (*static program features*) are extracted and stored in a database. The static program features count the occurrence of certain activities (e.g., arithmetic operations, memory accesses) or describe the ratio between two characteristics (e.g., the ratio between computation and memory accesses or the ratio between the number of branches and all instructions). In step ❸, the intermediate representation of the program is then passed to the backend which generates multi-device OpenCL code. Once generated, in step ❹, the new program will be executed by the Insieme Runtime system with various problem sizes and different available task partitionings. In step ❺, the obtained performance measurements, together with the problem size-dependent features of the program (i.e. *runtime features*), are collected and added to the database. Apart from the problem size itself, the runtime features describe how much data has to be transferred between the host and the devices during the program execution. Finally, after all the previous steps have been accomplished for all programs, the trainer uses the features and the performance measurements stored in the database (step ❻) to generate a task partitioning prediction model (step ❼).

To generate a multi-device OpenCL program, we extended the Insieme Compiler to analyze the generated IR of the input program and infer the bounds of OpenCL buffer regions. This analysis identifies whether a buffer should be replicated or distributed evenly among several devices. After deriving and collecting all the buffer's access

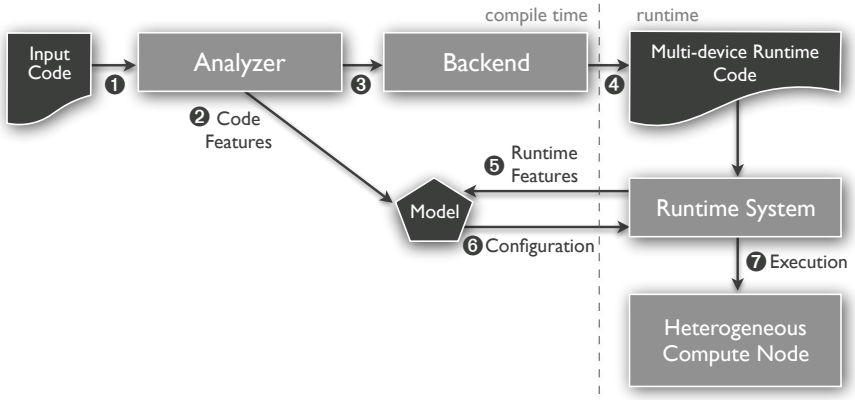


Figure 4.2: Deployment phase

patterns, the analysis checks if the access expression is (a) a constant, (b) the result of a convex function depending on the thread id, or (c) something different. If only accesses of type (b) occur, the buffer is split among all devices (i.e. it is splittable). If accesses of type (a) or (c) happen, part of it (a) or the entire buffer (c) has to be copied to every device (i.e. the buffer is non-splittable). In case of (a) and (c), the amount of data to be transferred increases linearly with the number of devices used. Obviously, copying the same data to each device is only feasible if there are only read accesses to these buffers. When write accesses of type (a) or (c) occur, our framework is not capable of distributing the kernel. A kernel can be distributed over several devices if and only if all its buffers with write accesses are splittable. However, due to the limited synchronization capabilities of OpenCL, in the majority of all kernels, the write accesses use access pattern (b). The access pattern analysis is based entirely on the kernel code. However, also the host code has to be adapted according to the results of the access pattern analysis in order to guarantee the correct distribution of data. For this reason, our source-to-source compiler connects the host and kernel code during the translation of the input OpenCL program into IR, enabling the analysis of the entire program. After the analysis, the IR is translated by the backend to a multi-device OpenCL program. The generated code is semantically equivalent to the input code, but its kernels can be distributed among a generic number of devices by the runtime system. This implies that some buffers are replicated while others are distributed over the selected devices, depending on their access pattern inside the kernel.

During the deployment phase, depicted in Figure 4.2, the partitioning prediction model is applied to a new input OpenCL program. In step ❶ and ❷, the OpenCL program is provided to the analyzer and the static features are extracted. In step ❸ and ❹, the intermediate representation is passed to the backend which generates a multi-device OpenCL program. When the program is executed, the runtime features are provided to the previously trained model (step ❺), which combines them with the static program features to predict an effective task partitioning for the current program with the selected problem size (step ❻). Finally, in step ❼, the runtime system executes the program on the given hardware using the predicted task partitioning.

To access the multiple devices available in the heterogeneous compute node, we extended the Insieme runtime system with OpenCL support. During compilation, the source code generated by the backend is embedded with appropriate runtime calls responsible for devices initialization, data transfers, and kernels execution. It is worth noting that OpenCL cannot synchronize the execution across different vendor platforms. We enhanced the Insieme runtime system to avoid such limitation and orchestrate the concurrent execution in the available devices. This investigation laid the foundation for other studies that culminated with the creation of the *libWater* library. Chapter 5 will analyze and describe the library in more detail.

4.1.3 *Experimental Environment*

To evaluate the performance of our approach we used the set of 23 programs listed in Table 4.1. These programs have been drawn from OpenCL vendors example codes, benchmark suites [27, 38, 23], applications from our department and VRC at the Universität Stuttgart [93]. As shown in Figure 4.1, all training codes are compiled with the Insieme source-to-source Compiler and their static program features are collected in a database. After the compilation, the programs are executed with various problem sizes (9 to 18 problem sizes, depending on the program) and task partitionings, adding to the database information about runtime features and execution times. In order to generate the training patterns needed for the model generation, we perform an exhaustive search on the set of explored task partitionings,

Training codes description	Performance on *mc1*				Performance on *mc2*			
Application	CPU	GPU	SVM	ANN	CPU	GPU	SVM	ANN
Data Transfer to/from Device	90	37	92	**98**	84	72	88	**94**
Vector Addition	77	40	93	**94**	71	69	**87**	85
Matrix Multiplication	64	49	78	**87**	45	79	**98**	90
Black-Scholes Option Pricing	82	41	91	**93**	65	76	93	**95**
Vertex positions in Sine Wave Pattern	15	**70**	34	47	7	70	83	**95**
2D 3x3 Convolution	70	50	94	**98**	38	82	95	**96**
Molecular Dynamics Simulation	81	57	94	**99**	68	87	83	**94**
Sparse Matrix Vector Multiplication	96	59	97	**100**	82	93	**98**	96
Linear Regression	51	59	51	**60**	22	74	70	**83**
K-Means clustering	86	48	97	**98**	76	80	85	**88**
K-Nearest-Neighbor Classification	22	**68**	45	48	5	68	69	**87**
Symmetric Rank-2k Operations	**95**	24	87	78	**94**	49	51	54
Sobel Filter	75	58	91	**97**	51	**90**	85	85
Median Filter	82	54	96	**98**	56	93	90	**96**
Ray-triangle Intersection	90	62	94	**97**	74	**98**	89	94
Finite-time Lyapunow Exponent Field Calculation	77	56	**95**	94	59	82	**85**	84
Flow Map Calculation	91	35	60	**92**	75	81	81	**88**
Chunked Reduction	72	41	84	**89**	61	73	**88**	87
Perlin Noise Generator	**94**	17	81	73	83	49	84	**85**
Chunked Calculation of the Geometric Mean	68	45	81	**92**	54	81	**94**	93
Mersenne Twister Random Number Generator	79	41	**91**	89	67	72	90	**91**
Bytewise Integer Compression	77	39	90	**94**	70	69	89	**95**
Simulation of a Swinging Pendulum	20	**75**	20	20	19	70	58	**76**

Table 4.1: Description of test cases used for model training and performance of various task partitioning strategies.

	Machine	
Name	*mc1*	*mc2*
CPU Vendor	AMD	Intel
CPUs	2x Opteron 6168	2x Xeon X5650
Compute Units	24	24
Clock Frequency	1.9 GHz	2.67 GHz
Peak Performance	364 GFLOPS	256 GFLOPS
Memory Size	32 GB	24 GB
Memory Bandwidth	83 GB/s	62 GB/s
Compiler	GCC 4.6.3 w/ "-O3"	
Operating System	CentOs 5.8	
OpenCL Version	AMD APP SDK 2.7	
GPU Vendor	Ati	NVIDIA
GPUs	Radeon HD 5870	GeForce GTX 480
Compute Units	20	15
Clock Frequency	850 MHz	1401 MHz
Peak Performance	2.7 TFLOPS	1.3 TFLOPS
Memory Size	2 GB	1.5 GB
Memory Bandwidth	153 GB/s	177 GB/s
Host Connection	PCIe 2.0 x16	PCIe 2.0 x16
OpenCL version	AMD APP SDK 2.7	CUDA 4.1.1

Table 4.2: Experimental target architectures.

finding the one with the best execution time. Each training pattern consists of the static features of a program, its runtime features for a certain problem size as well as the best task partitioning for the given program with the current problem size. Such task partitioning will then be used as target value during the training of our model. Based on the training patterns we build a model with one input for each feature and one output, which represents the task partitioning predicted by the model. Our framework currently offers Support Vector Machines [26] (SVM) and Artificial Neural Networks [26] (ANN) for the construction of the model. To ensure a fair comparison between different task partitionings, we measured the execution time of the kernels including the memory transfer overhead [41]. For each task partitioning, we executed a series of five experiments, recording the average execution time. The result has been validated with the *Student's t test* [98], ensuring reliable results with a confidence level of 95%.

The experiments were performed on two different heterogeneous target architectures composed of three OpenCL devices: two GPUs and two multi-core CPUs in a dual-socket infrastructure. While both GPUs represent a separate device, the two CPUs are reported as a single OpenCL device. The first platform, *mc1*, consists of two AMD Opteron CPUs and two Ati Radeon GPUs, while the second, *mc2*, holds two Intel Xeon CPUs and two NVIDIA GeForce GPUs. Table 4.2 gives a more detailed listing of the two systems' characteristics.

For the target architectures used in this study, consisting of one CPU device and two GPU devices, we characterize each task partitioning with a tuple of three numbers representing the percentage of the workload executed on a specific device. The first number represents the portion to be executed on the CPU while the second and third number represent the percentage for the first and second GPU, respectively. Task partitioning $(100, 0, 0)$, for example, means that the entire workload is assigned to the CPU, while $(0, 50, 50)$ means that the work is distributed evenly among the two GPUs while nothing is assigned to the CPU. The entire set of task partitionings P is constructed as follows:

$$X = \{0, 10, 20, ..., 100\}$$
$$P = \bigcup_{x \in X} \left\{(x, 100 - x, 0), (x, \tfrac{100-x}{2}, \tfrac{100-x}{2})\right\}$$

Where X is the set of different percentage values of the workload considered to be executed by the CPU. The remaining workload is then executed by the first GPU or it is distributed evenly among the two GPUs. The resulting set P consists of 21 different task partitionings. From this set P our runtime system tries to select an effective task partitioning using the prediction model as described in Section 4.1.2.

To evaluate the performance of our approach we compare the execution times of a program with two different task partitionings. The first one is proposed by the Insieme Runtime system and the second one is found by an exhaustive search over all task partitionings in the set P. In order to evaluate the quality of our models we do a leave-

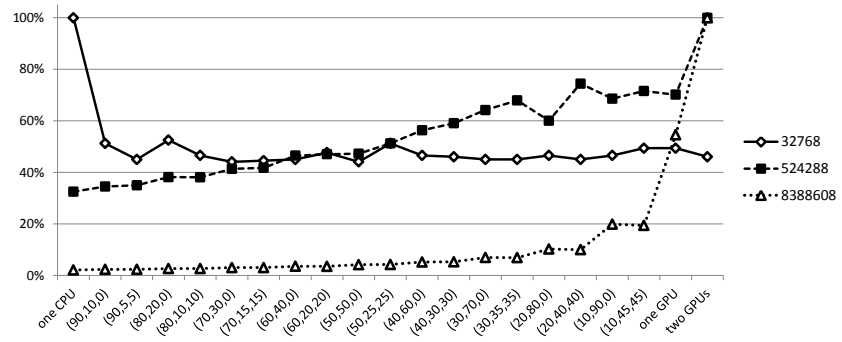
one-out cross validation [34] on all our training programs of the set C listed in Table 4.1. For each program $c \in C$, we train the model with all programs except c. Obviously, this means not leaving out only one training pattern, but all training patterns related to program c (all different problem sizes).

4.1.4 *Evaluation*

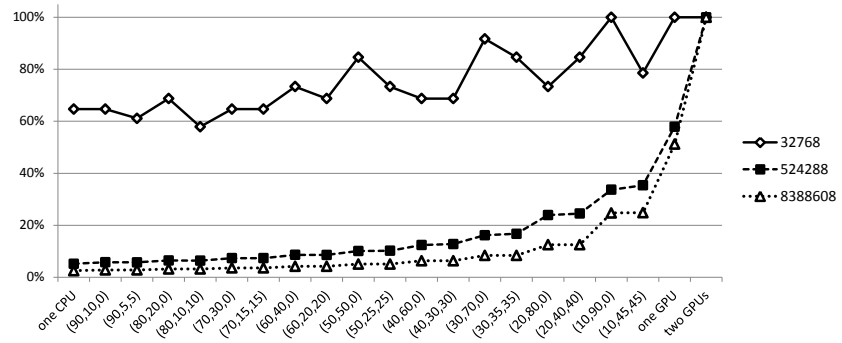
As performance metric for our evaluation, we use the achieved percentage of the maximum performance. We calculate it using the execution time of the best task partitioning (identified with an exhaustive search over all task partitionings used) and the actual execution time of the selected task partitioning.

Depending on the target architecture, the problem size and the program, it can be important to select a certain task partitioning, whereas in other cases, several different task partitionings may deliver similar good performance. For instance, as can be seen in Figures 4.3a and 4.3b, when executing *matrix multiplication* with large problem sizes it is very important to distribute the workload over both GPUs. Furthermore, for hybrid solutions, it is not important if one or two GPUs are used since the CPU is always the limiting factor. For smaller problem sizes, in particular for *mc2*, several task partitionings yield good performance. In contrast to that, on *mc1* small matrices should be multiplied on the CPU alone. The penalty for selecting a non-optimal task partitioning on intermediate problem sizes on *mc1* is less severe than on *mc2*.

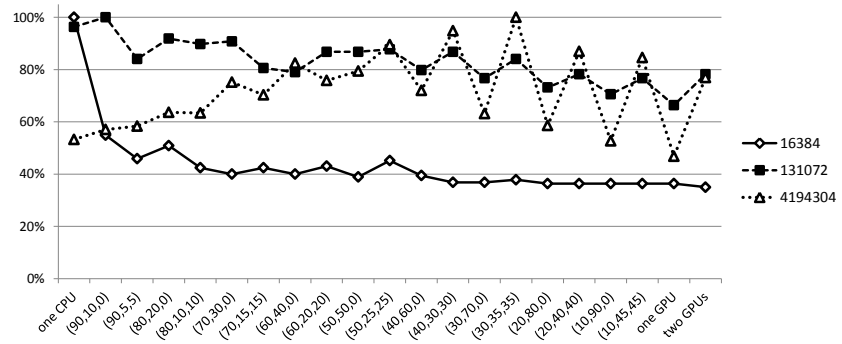
The situation is different when running our *integer compression* implementation. Figure 4.3c shows that on *mc1* with a problem size of 16384 work items, the CPU substantially outperforms all other task partitionings, while on *mc2* the difference is much smaller and all task partitionings deliver 40% or more of the maximum performance, as revealed in Figure 4.3d. For the larger problem sizes, on both target architectures, a hybrid task partitioning delivers the best performance. However, the best performing task partitioning is different for each problem size and target architecture. In this test case, using a heterogeneous distribution can reduce the execution time by up to 23% over any homogeneous task partitioning (including the dual GPU



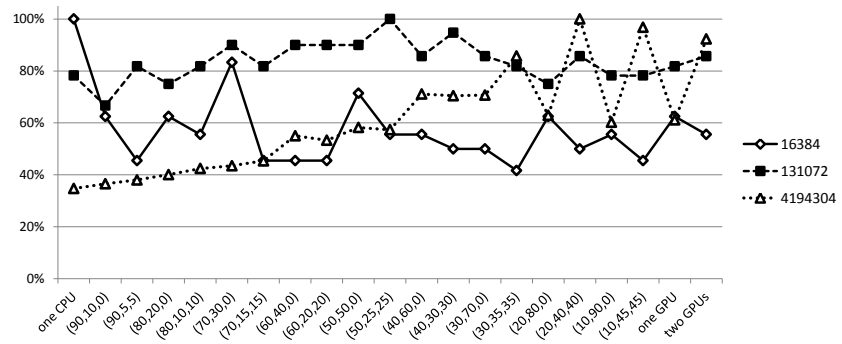
(a) Matrix Multiplication on *mc1*



(b) Matrix Multiplication on *mc2*



(c) Bytewise Integer Compression on *mc1*



(d) Bytewise Integer Compression on *mc2*

Figure 4.3: Performance of different programs on two target architectures

Task Partitioning Approach	Execution Time				Performance		
	Training (sec)		Deployment (ms)				
	mc1	*mc2*	*mc1*	*mc2*	*mc1*	*mc2*	**Avg.**
CPU only	-	-	-	-	73	58	**65.5**
GPU only	-	-	-	-	48	77	**62.5**
Random	-	-	0.12	0.09	44	55	**49.5**
SVM	22	19	0.28	0.18	82	85	**83.5**
ANN	317	201	0.07	0.06	86	89	**87.5**

Table 4.3: Properties and performance of different machine learning algorithms

task partitioning). From the 355 training patterns considered for this study, more than 25% deliver best performance when using a hybrid task partitioning.

For the Insieme Runtime system, we tested a variety of models, generated either with a Support Vector Machine [26] (SVM) or an Artificial Neural Network [26] (ANN). For both techniques, we used the implementation provided by the Shark library [53]. In this section, we compare the performance of our model-guided runtime system with the performance of the two default strategies which use either one CPU or one GPU. These are the only available options when using the unchanged input programs, without the generation of multi-device code by the Insieme Compiler. Furthermore, without using Insieme framework, the challenging task of choosing the most appropriate device is left to the user. We also show the advantage of our approach over the expected performance of a random scheduler, calculated by taking the average execution time over all task partitionings in our set P (described in Section 4.1.3).

Table 4.3 shows the average performance for a cross validation over all test cases in Table 4.1 using different scheduling approaches. On *mc1* the CPU-only strategy outperforms the GPU-only strategy while on *mc2* we observe the opposite behavior. This underlines the complexity of choosing the most appropriate device in a heterogeneous environment. On average, over the two target architectures, both default strategies fail to reach 70% of the maximum performance. In most cases, there are only a few well performing task partitionings while the others show rather poor performance. Therefore, the ran-

dom scheduler is not a good solution and even lags behind the two default strategies.

Our SVM approach uses the MulticlassSVM implementation of [53]. As kernel function we used Radial Basis Function [26] (RBF). This kernel function is the most widely used for classification with SVMs. The parameter γ of the RBF was set to 2.5, the regularization parameter c was set to 15 for both positive and negative examples. We observed, that the performance does not vary by more than 4 - 5% when changing these values, which demonstrates the robustness of SVMs with regard to these parameters.

The ANNs used for our study are three-layer feed-forward perceptron networks with a sigmoid activation function and five neurons in the hidden layer [26]. All three layers are fully connected with their neighboring layers. For our ANN we use the FFNet implementation of [53]. All weights inside an ANN are initialized randomly within the same range, equal to $+/-0.125$.

As training algorithm we used the conjugate gradient method provided by Shark, which automatically adapts the training rate. To determine the number of training iterations for the neural network, we use the *early stopping* method which terminates the training automatically after a certain level of convergence is reached. The training data is split into a training set, used to train the model, and a validation set which is not used for training. The level of convergence is measured by observing how the error on the validation set evolves over consecutive training iterations [26]. Depending on what test case is removed from the training set to perform the cross validation, the training is stopped after 36 to 749 iterations. The training times shown in Table 4.3 refer to the training for all test cases without cross validation.

As shown in Table 4.3, the ANN shows a better performance than the SVM and it is also faster to predict the task partitioning of a program. For both of our approaches, the time to predict the task partitioning is negligible (in the range of 0.06 to 0.28 ms). The downside of ANN is the corresponding relatively long training time as well as the associated sensitivity regarding the tuning parameters like network structure or weight initialization range. SVMs do not have these many tuning parameters and the quality of the result does not depend that much on the parameters' value.

In Table 4.1 we compare the performance of the task partitionings predicted by the Insieme Runtime based on an SVM and ANN, with the performance delivered by the CPU/GPU only strategy for each code and each target architecture individually. For almost all test cases, the CPU-only strategy delivers a higher performance on *mc1* than on *mc2*, while the GPU-only strategy usually performs better on *mc2*. This is related to the weaker performance of the GPU (Ati Radeon HD 5870) in *mc1*. Its VLIW architecture with a very wide instruction width and high branch miss penalty would require specific fine-tuning of each code to perform well [120]. However, none of our test cases was tuned for a specific device.

Our models are capable of representing the target architecture's characteristics in order to find performance efficient task partitionings. Our approaches also determine which device is to be favored on a specific target architecture. This is underlined by the fact that they show their worst performance in atypical test cases, i.e. test cases which perform better on the GPU than on the CPU on *mc1* (e.g. Simulation of a Swinging Pendulum) or vice versa on *mc2* (e.g. Symmetric Rank-2k Operations on *mc2*). On average considering both target architectures, our machine learning guided approaches reached up to 87.5% of the optimal performance across 23 programs outperforming the default strategies of using only the CPU or only the GPU, which achieve 65.5% and 62.5%, respectively.

4.2 TENSOR COMPUTATION ON HETEROGENEOUS NODES

In the previous section we focused on automatic methods for partitioning OpenCL tasks on heterogeneous compute nodes. Differently, in this section we will focus on the scheduling of independent OpenCL tasks and in particular on exploiting the computational power of emerging heterogeneous compute nodes in order to improve the tensor computation of massive datasets of millions of points in the VISH visualization shell [18]. VISH is a productive framework that provides functionalities for both efficient data processing and visualization of big data.

The contributions are as follows:

- First, we implemented a new tensor computation code in OpenCL using a uniform grid space partitioning approach, and evaluated its performance against the current KD-Tree implementation available in VISH.
- Second, we investigated the performance of the OpenCL implementation on 8 different devices, comprising four GPUs, three CPUs and one accelerator, from desktop and server domains.
- Finally, we proposed two low-complexity dynamic heuristics for the scheduling of independent dataset fragments and compared them with three static scheduling heuristics in two multi-device heterogeneous compute nodes.

4.2.1 *Related Work*

The study of the interaction of millions of points, present in modern datasets, requires scalable systems capable of supporting the large computational demands. In order to actually improve the scalability of such systems many spatial partitioning methods were proposed and investigated [13, 124, 125]. Some of these approaches are suitable for simulations which frequently have high density in one or several spatial locations and some perform best with uniformly distributed points. In recent years, among such methods, uniform grid data structures have received great attention from the research community. Erra et al.[32], leveraging the GPU processing power, implemented an efficient framework which permits to simulate the collective motion of high-density individual groups. Aaby et al. [1] presented the parallelization of agent-based model simulations (ABMS) with millions of agents on multiple GPUs and multi-core processors. Vigueras et al. [121] proposed different parallelization strategies for the collision check procedure that takes place in agent-based simulations. Green [39] described how to implement a simple particle system in CUDA using a uniform grid data structure. Husselmann et al. [49] presented a GPU solution for grid-boxing in multi-spatial-agent simulations.

While the uniform grid approach of these works is similar to ours, they are restricted by the language of choice to some specific hardware. Differently, using OpenCL, our work is not limited to a single platform and can be executed in multiple heterogeneous devices. This

advantage allows us to compare different platforms and fully exploit the computational performance of heterogeneous compute nodes as shown in other recent works [66, 37].

4.2.2 *Tensor Computation*

For a set of N points $\{P_i : i = 1, ..., N\}$ the point distribution tensor S at the point P_i is defined as:

$$S(P_i) = \frac{1}{N} \sum_{k=1}^{N} \omega(|t_{ik}|)(t_{ik} \otimes t_{ik}^{\tau}), \tag{1}$$

whereby $\omega(x) = \theta(r - x)$ is a threshold function dependent on a radius r [104], $t_{ik} = P_i - P_k$, $^{\tau}$ is the transpose and $\otimes$ denotes the tensor product. Figure 4.4 depicts a graphical result of the computation applied to the river Rhein dataset.

The naive approach for the tensor computation is, therefore, to test every point with all the others, leading to a quadratic algorithmic complexity. In real models composed of millions of points this approach is not applicable due to the inherent performance problem. To mitigate this problem spatial partitioning methods have been investigated [13, 124, 125]. Currently, the VISH visualization shell offers a tensor field computation algorithm that makes use of a KD-Tree data structure to find the neighbors of a certain point. After the tree building phase, in which the points of the dataset are inserted into the KD-Tree, the computation of the tensor distribution is executed in parallel for each point with a series of range queries dependent on a given search radius (threshold function). The KD-Tree code was implemented via C++ STL containers and parallelized using OpenMP [89] with dynamic scheduling and packets of 10000 loop iterations.

Differently, a uniform grid space partitioning approach involves a spatial partitioning of the model system into equally-sized boxes (cells) containing different numbers of points. It is important to ensure that the grid box size is not smaller than the radius size, as this would force the algorithm to check many surrounding grid boxes. On the other hand, if the grid box is larger than the radius, each box would contain numerous points and the process of locating neighbors would once again be checking many points outside the radius area.

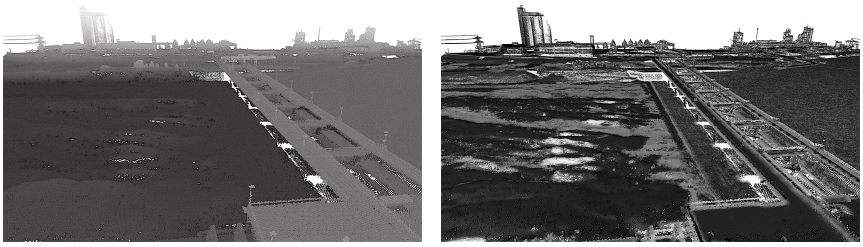


Figure 4.4: Input point distribution (left) and output tensor (right) of the river Rhein dataset

In our case, the uniform grid approach is effective because the radius is an input parameter of the program and therefore we are able to tune the grid box size accordingly.

We implemented the tensor computation application in OpenCL using a uniform grid space partitioning approach. We used a grid with a cell size of two times the radius, which implies that each point can only interact with points in the neighboring cells (27 in a 3D space). The complete program, described in Algorithm 4.1, is composed of three phases: initialization, computation, and finalization. During the initialization phase, the OpenCL devices are initialized, the OpenCL kernels are compiled and the metadata of the dataset is loaded. The metadata contains information about the number of fragments present in the dataset, the number of points for each fragment, plus other additional information useful for the graphical visualization. The dataset consists of independent fragments of spatially ordered points to facilitate the data manipulation and visualization. Each fragment contains a small percentage of replicated data necessary for the computation of the tensor algorithm at points close to the border of the fragment. Once the initialization phase is completed the system is ready to schedule the fragments on the available devices and the computation phase will start. For each fragment, the point's coordinates will be loaded in main memory and transferred to the device memory where the computation will take place. On the device, the uniform grid will be created and used during the tensor computation in the search for the neighboring points. Once the computation is done, the computed tensor data is transferred back to the host's main memory and finally saved to the disk. The finalization phase releases all the devices and the used memory.

Algorithm 4.1 The OpenCL tensor computation algorithm

```
 1: devices_initialization()                                  ▷ Initialization Phase
 2: metadata ← load_dataset_metadata()
 3: for all fragments in dataset do                           ▷ Computation Phase
 4:     pts_ar ← load_points_data(fragment)
 5:     write_points_to_device(pts_ar)
 6:     create_uniform_grid(pts_ar, radius){
 7:         hash_ar ← compute_hash_values(pts_ar)
 8:         index_ar ← sort_points_indices(hash_ar)
 9:         begin_end_ar ← compute_interval(hash_ar)
10:     }
11:     compute_tensor(pts_ar, index_ar, begin_end_ar)
12:     tsr_ar ← read_tensor_from_device()
13:     write_tensor_to_disk(tsr_ar)
14: end for
15: devices_finalization()                                   ▷ Finalization Phase
```

The steps necessary for the creation of the uniform grid are described in Algorithm 4.1 (lines 6-10). The algorithm consists of multiple OpenCL kernels. The first kernel (line 7) calculates a hash value for each point based on its cell ID and stores them in an array in device main memory (*hash_ar*). The array is then sorted based on the cell IDs while updating at the same time the order of the point IDs. Sorting is performed using a bitonic algorithm [14]. The result of this computation is an array of point IDs sorted by cell (*index_ar*). The last kernel (line 9) is then executed to find the begin and the end position of any given cell. The kernel generates an OpenCL work-item for each point and compares the cell ID of the current point with the cell ID of the previous one in the *hash_ar* array. If the two indices are different, the current work-item ID is used as start index of the current cell and the *begin_end_ar* array is updated using a scattered write operation. During the execution of the *compute_tensor* kernel (line 11), using the *begin_end_ar* and *index_ar* arrays, we calculate the neighbor cells for each point in the fragment and for each point present in the cells we compute the difference to the current point in each dimension (x, y, z). If the length of the difference vector is less than the radius, the tensor array and the points counter are updated. Finally, in the last step, each element of the tensor array is divided by the points counter.

4.2.3 *Scheduling Independent Fragments*

As mentioned in the previous section, the tensor computation is applied on single fragments that compose the complete dataset. The fragments are completely independent of each other and can be computed in parallel using the available devices present in the compute node. During program execution, a scheduler is responsible for the allocation of the fragments among the heterogeneous devices. The scheduling problem has been extensively investigated and numerous methods have been reported in the literature [20, 30, 76, 72]. In our program we implement two low-complexity scheduling heuristics: *SimpleH* and *SimpleHS*. *SimpleH* analyzes the dataset metadata and sorts the list of fragments based on the number of points contained in each of them. The algorithm then proceeds by dynamically assigning the fragment with the smallest number of points to the slowest device and the fragment with the biggest number of points to the fastest device. Following this pattern, the scheduler continues to dynamically assign fragments until all of them are processed. *SimpleHS* follows a similar pattern. A fragment is assigned to the slowest device if the predicted execution time of the fragment on that device is lower than the predicted execution time of all the remaining fragments on the fastest device. The execution time for each fragment is predicted with a quadratic regression model using the number of points of the fragment. During the program execution, information regarding the number of points per fragment and execution times are stored. This information will then be used to build a more accurate model whenever the slowest device is ready to compute a new fragment. Although for simplicity the heuristic algorithms are described taking into consideration only two devices, they can be applied to heterogeneous compute nodes composed of a single slow device (CPU) and multiple equally fast devices (e.g. GPUs). In Section 4.2.5 we evaluate and compare *SimpleH* and *SimpleHS* with three heuristics which are widely used to address the problem of scheduling independent tasks in heterogeneous compute nodes: *Min-Min* [50, 20], *Max-Min* [50, 20], and *Sufferage* [78]. Because these are static heuristics, it is assumed that an accurate estimation of the expected execution time for each fragment on each device is known prior to execution and con-

tained within an ETC (expected time to compute) matrix. The *Min-Min* heuristic proceeds by assigning a previously unassigned fragment to a device in every iteration. The assignment is decided based on a two-step procedure. In the first step, the algorithm computes the minimum completion time (MCT) of each unassigned fragment over the devices in order to find the best device which can complete the processing of that fragment at earliest time. This decision is made taking into account the current loads of the devices and the execution time of the fragment on each device. In the second step, the algorithm selects the fragment with the minimum MCT among all unassigned fragments and assigns the fragment to its best device found in the first step. The *Max-Min* heuristic differs from the *Min-Min* in the fragment selection policy adopted in the second step of the fragment-to-device assignment procedure. Unlike *Min-Min*, which selects the fragment with the minimum MCT, *Max-Min* selects the fragment with the maximum MCT and then assigns it to the best device found in the first step. *Sufferage* is also similar to *Min-Min* but adopts a different fragment selection policy. In the first step of the process, the algorithm computes the second MCT value in addition to the MCT value for each fragment. In the second step, the sufferage value, which is defined as the difference between the MCT and the second MCT values of a fragment, is taken into account. *Sufferage* selects the fragment with the largest sufferage and assigns it to the best device found in the first step.

4.2.4 *Experimental Environment*

In order to evaluate the performance of the KD-Tree and OpenCL implementations presented in Section 4.2.2, we use a dataset of 58 million points, generated using a combination of LIDAR and echo sounding data captured at the river Rhein in Rheinfelden [29]. The dataset is stored in the HDF5 [116] format, based on the scientific data format F5 [103, 19], to be easily manipulated with the VISH infrastructure. The dataset is composed of 65 fragments that contain between one thousand and 3.5 million points each.

To represent the broad spectrum of OpenCL-capable hardware we selected eight devices, comprising four GPUs, three CPUs, and one

Device	S9000	K20m	Phi7120	2x E5-2690v2	2x Opt.6168
OpenCL Vendor	AMD	NVIDIA	Intel	Intel	AMD
OpenCL Version	SDK v2.9	CUDA 6.5	SDK 2014	SDK 2014	SDK v2.9
Operating System	CentOS6.5	CentOS6.5	CentOS6.5	CentOS6.5	CentOS6.5
Host Connection	PCIe 3.0	PCIe 3.0	PCIe 2.0	-	-
Device Type	GPU	GPU	ACL	CPU	CPU
Class	server	server	server	server	server
Compute Units	28	13	240	40	24
Max Workgroup	256	1024	8192	8192	1024
Clock (MHz)	900	705	1333	3000	1900
Cache	R/W	R/W	R/W	R/W	R/W
Cache Line	64	128	64	64	64
Cache Size (KB)	16	208	256	256	64
Global Mem (MB)	3072	4799	11634	129006	64421
Constant (KB)	64	64	128	128	64
Local Type	Scratch	Scratch	Global	Global	Global
Local (KB)	32	48	32	32	32

(a) Server devices

Device	Radeon5870	GTX480	i7-2600K
OpenCL Vendor	AMD	NVIDIA	Intel
OpenCL Version	SDK v2.9	CUDA 6.5	SDK 2014
Operating System	CentOS5.9	CentOS5.9	Mint16
Host Connection	PCIe 2.0	PCIe 2.0	-
Device Type	GPU	GPU	CPU
Class	consumer	consumer	consumer
Compute Units	20	15	8
Max Workgroup	256	1024	8192
Clock (MHz)	850	1401	3400
Cache	None	R/W	R/W
Cache Line	-	128	64
Cache Size (KB)	-	240	256
Global Mem (MB)	1024	1536	7965
Constant (KB)	64	64	128
Local Type	Scratch	Scratch	Global
Local (KB)	32	48	32

(b) Consumer devices

Table 4.4: Benchmarked OpenCL devices

accelerator. Their device characteristics as reported by OpenCL are summarized in Table 4.4.

To exploit the computational capabilities of heterogeneous compute nodes, we evaluated different scheduling heuristics. The experiments were performed on two different heterogeneous target architectures composed of three OpenCL devices: two GPUs and one CPU. The first platform, *mc3*, consists of an Intel i7-2600K CPU and two NVIDIA GTX 480, while the second, *mc4*, holds two Intel Xeon E5-2690 v2 CPUs (reported as a single OpenCL device) and two AMD Fire Pro S9000 GPUs. For the static scheduling heuristics we utilized, as estimation time for each fragment (ETC matrix), the actual execution time of the fragment on the different devices. Differently, for the computation of the coefficients in the *SimpleHS* heuristic, we used the multi-parameter fitting present in the GNU Scientific Library.

All the benchmarked programs were compiled with GCC version 4.8.1 with the -O3 optimization flag. In each different device, the OpenCL kernels were compiled by the respective vendor compilers at runtime during the program initialization. All the experiments were conducted on the previously described dataset. The measurements were collected for the computational phase of the program, excluding the initialization and finalization phases. We repeated each experiment 10 times and we computed the mean value and the standard deviation of the measured performance. In all the presented experiments, the standard deviation is negligible, thus we do not report it.

4.2.5 *Evaluation*

To compare the performance of the KD-Tree version and our OpenCL implementation, we executed the tensor computation on the input dataset on the same multi-core CPU (Intel i7-2600K). Both implementations are parallel: the KD-Tree version uses OpenMP to parallelize the loop over all points, while the OpenCL approach is inherently parallel. The building phase of the tree in the KD-Tree implementation is sequential, however, it represents a very small part of the overall execution time. The OpenCL version of the program experiences a significant speedup (24×) over the currently implemented VISH KD-

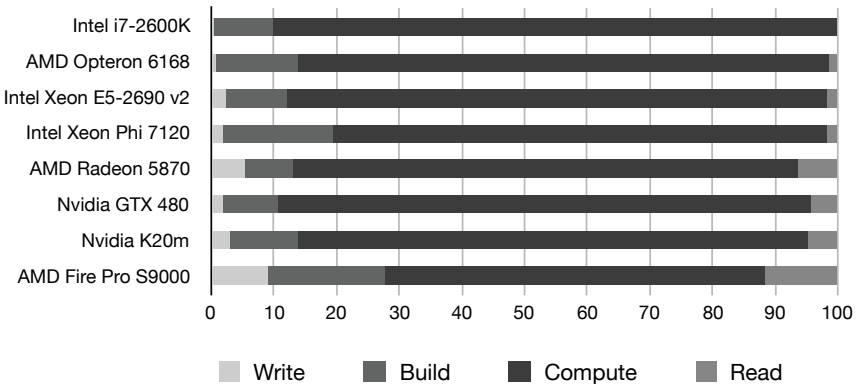


Figure 4.5: Normalized execution time spent in the different parts of the OpenCL tensor computation algorithm

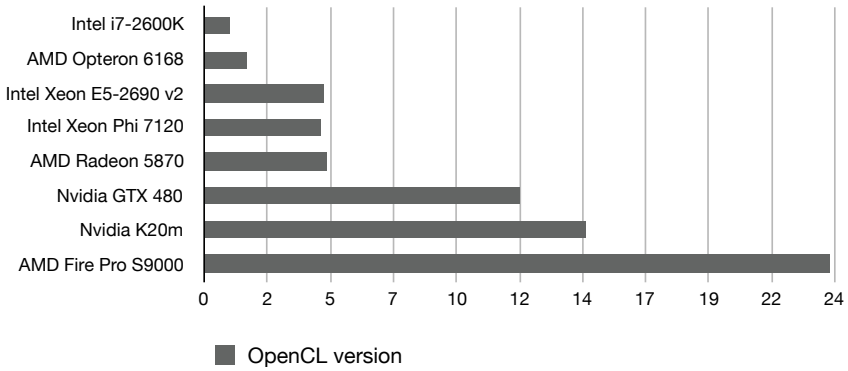


Figure 4.6: Speedup of the different devices over the Intel i7-2600K

Tree version, reducing the execution time from 1 hour to 150 seconds. The performance improvement comes from different reasons. First, grid data structures are more suited for range queries (all the particles around a point in a given radius) while KD-Tree structures are more suited for k-nearest neighbors queries (first N-points close to a given point). Second, vectorization is rather hard in KD-Tree codes where many data-dependent branches are present. In contrast, the uniform grid OpenCL code can be more easily autovectorized by compilers. Third, we applied a few code optimizations that improve the performance of the OpenCL code. However, the optimizations only partially affect the speedup over the KD-Tree version, which remains significant even in their absence (12.9×).

Since OpenCL supports heterogeneous devices, we analyzed the performance of our OpenCL code on a set of heterogeneous devices described in Table 4.4. Figure 4.5 depicts the percentage of execution

time spent in the different phases of the OpenCL tensor computation described in Algorithm 4.1. *Write* represents the transfer of the fragment points to the device (line 5), *Build* represents the time spent building the uniform grid structure (line 6-10), *Compute* indicates the time spent in the computation (line 11), while *Read* identifies the transfer of the tensors to the host device (line 12). In all the tested hardware the movement of data does not represent an important part of the execution time. *Write* and *Read* functions are always under 5% of the total time. The only exception is the AMD Fire Pro S9000 where the data transfers represent 9.0% and 11.6% of the execution time, respectively. This is mainly due to the small amount of time spent in the tensor computation thanks to the strong computational capabilities of the device.

In Figure 4.6 we present the performance comparison of the heterogeneous devices. The speedup of the CPUs respects the characteristics of the hardware. The AMD Opteron, with a higher number of compute units but a lower clock rate, experiences a 1.6× speedup over the Intel i7 while the Xeon, with 40 compute units and a similar clock rate, reaches a 4.5× speedup. All the GPUs show significant improvements in performance compared to the Intel i7. The desktop GPUs AMD Radeon 5870 and NVIDIA GTX 480 reach a speedup of 4.7× and 12.0×, respectively. The server GPUs NVIDIA K20m and AMD Fire Pro S9000, designed for the HPC market, show a speedup of 14.5× and 23.8×, respectively. It is worth underlining that although the NVIDIA K20 offers higher theoretical peak performance, in our test the AMD Fire Pro S9000 is around 1.5 times faster. The only accelerator present in our test is the Intel Xeon Phi. Although its peak performance is comparable with the tested server GPUs, it reaches only a speedup of 4.4× compared to the Intel i7. The difference in performance between the GPUs and the Xeon Phi is difficult to investigate as it derives from the differences in the architecture and from the different maturity of the OpenCL toolchains.

In conclusion, the results show that the problem is well-suited for GPUs, reducing the processing time of the complete input dataset to 6.3 seconds in case of the AMD Fire Pro S9000.

As previously described in Section 4.2.4, we conducted a set of experiments with scheduling heuristics in two heterogeneous compute nodes. The objective of our scheduler is to find a fragment-to-device

	Device	Scheduling Heuristics				
Consumer Platform		Sufferage	Min-Min	Max-Min	SimpleH	**SimpleHS**
	mc3-CPU1	5976.34 [15]	0.00 [0]	5980.55 [7]	6124.42 [21]	4807.08 [19]
	mc3-GPU1	5971.16 [25]	5993.26 [32]	5988.02 [29]	5962.76 [23]	6014.41 [24]
	mc3-GPU2	5974.84 [25]	6502.75 [33]	5988.23 [29]	5984.76 [21]	6049.29 [22]
	Ex. time (ms)	5976.34	6502.75	5988.23	6124.42	6049.29
	Norm. to Suff.	100.00%	91.90%	99.80%	97,58%	**98.79%**
Server Platform		Sufferage	Min-Min	Max-Min	SimpleH	**SimpleHS**
	mc4-CPU1	2708.42 [21]	1341.58 [7]	2711.67 [12]	2758.30 [28]	2758.30 [28]
	mc4-GPU1	2706.41 [22]	2986.75 [29]	2710.42 [26]	2797.53 [20]	2797.53 [20]
	mc4-GPU2	2709.77 [22]	2766.57 [29]	2712.34 [27]	2838.69 [17]	2838.69 [17]
	Ex. time (ms)	2709.77	2986.75	2712.34	2838.69	2838.69
	Norm. to Suff.	100.00%	90.73%	99.91%	95.45%	**95.45%**

Table 4.5: Performance of the different scheduling heuristics in two heterogeneous compute nodes

assignment that minimizes the total execution time (makespan). Table 4.5 shows, for each device in the two compute nodes, the time spent to execute the number of assigned fragments (in square brackets) for the particular scheduling policy. The table also presents for each heuristic the makespan and the normalized result to the *Sufferage* heuristic. In both nodes *Sufferage* reaches an almost perfect load balancing between the three available devices, fully utilizing the available hardware. In both nodes the static scheduling heuristics obtain similar results, with *Max-Min* that reaches almost the same performance of *Sufferage*, while *Min-Min* shows 91.90% and 90.73% of the performance, respectively. These results are justified by the structure of the dataset. Usually, datasets collected with LIDAR technology contain few fragments with a big number of points and many small fragments with fewer points. Due to the fragment selection policy, *Sufferage* and *Max-Min* perform the assignment of the large fragments in early iterations resulting in a better load balancing between the devices. Differently, *Min-Min* favors the assignment of fragments with lower cost in early iterations, not reaching the same performance in terms of makespan. It is noteworthy that the three static scheduling heuristics assume an accurate estimation of the expected execution time for the fragments (ETC matrix) that will not be available at scheduling time. The resulting performance of the heuristics is therefore only useful as a comparison parameter for our low-complexity heuristics *SimpleH* and *SimpleHS*, which are only based on information available at schedul-

ing time. *SimpleH*, based on the assumption that the GPUs are always faster than the CPU, dynamically assigns the fragments with more points to the GPUs and the one with fewer points to the CPU. This simple mechanism facilitates the devices load balancing by avoiding assigning large fragments to slow devices. Although *SimpleH* is capable of reaching good performance, it also shows its weakness with our input dataset. The heuristic does not take into account the number of remaining fragments to assign and, when few are left, continues to distribute them to the CPU. This behavior can lead to load imbalance if the GPUs have to wait for the CPU that received one of the last fragments. This issue is solved with the *SimpleHS* heuristic, previously described in Section 4.2.3. *SimpleHS* tries to predict the approximate execution time of a new fragment based on the execution time of the previous ones. Although at the beginning the prediction error is high, it rapidly decreases during the scheduling of fragments. It is noteworthy that the overhead introduced by the prediction model is negligible and does not impact the performance of the scheduler. In our tests, *SimpleHS* is able to correctly predict when to stop the assignment of fragments to the CPU, obtaining a better load balancing between the devices. As depicted in Table 4.5, *SimpleHS*, scheduling fewer fragments to the CPU, always achieves better or equal performance compared to *SimpleH*, reaching 98.79% and 95.45% of the *Sufferage* performance in the two compute nodes.

These results validate the success of the proposed heuristics which, using only information available at scheduling time, show performance comparable to more sophisticated methods which require an accurate estimation of the expected execution times.

4.3 SUMMARY

In this chapter we introduced different techniques to partition and schedule OpenCL tasks on heterogeneous compute nodes.

In Section 4.1 we proposed a novel approach which can automatically distribute OpenCL programs on heterogeneous compute nodes. It consists of a source-to-source compiler, which translates a single-device OpenCL program into a multi-device OpenCL program and a runtime system which distributes the workload over all heteroge-

neous resources using a machine learning based, off-line generated prediction model. On average our approach outperformed default strategies showing that state-of-the-art compilers can automatize complex tasks with substantial impact on performance and productivity.

Differently, in Section 4.2 we proposed an OpenCL implementation of the second order tensor field computation of massive point datasets. We investigated the performance of our implementation on a set of heterogeneous devices, showing a remarkable reduction of the execution time. Furthermore, we investigated different scheduling policies on two heterogeneous compute nodes. The obtained results validate the success of the proposed heuristics, which show performance comparable to more complex static ones, only using information available at scheduling time.

5

SIMPLIFYING THE PROGRAMMING OF CLUSTERS

Ease of programming and best performance exploitation are often conflicting goals while designing programming models and abstractions for high performance computing (HPC). For instance, when programming a cluster, better performance can be obtained directly using low level and error prone communication layers like MPI [83]. Alternatively, high level models like domain specific languages and frameworks can be employed to simplify the programmability and portability of the code. This simplification, however, may also reduce performance due to the level of abstraction that is too far away from the underlying hardware.

In this chapter, we introduce *libWater*, a library-based extension of the OpenCL programming model that simplifies the development of applications for heterogeneous clusters improving both productivity and implementation efficiency. *libWater* does not alter the kernel logic of OpenCL kernels, but replaces the host-side API with a new, simpler and transparent interface which abstracts the underlying distributed architecture.

The remainder of this chapter is structured as follows. Section 5.2 provides an overview of related work. Section 5.3 and section 5.4 describe respectively the *libWater* programming model and the distributed runtime system with the underlying optimizations. The experimental evaluation is presented in Section 5.5. Finally, Section 5.6 summarizes and concludes our findings.

5.1 INTRODUCTION

Although OpenCL is a big leap forward in order to assure portability between different hardware, potentially replacing other standards, it also presents some limitations. A first problem is that it does not allow interactions between different platforms; for example, it is not possible to use event synchronization between devices from different vendors. Secondly, the semantics of OpenCL host ap-

plications is too verbose, as it includes different levels of abstraction (e.g., platform, device, queue). Moreover, while writing an application targeting heterogeneous or mixed-node clusters, we still require an intricate mix of OpenCL with a communication layer like MPI. Despite OpenCL can be easily extended in order to support remote, distributed devices[65, 3, 59, 33], the host-device paradigm forces the use of a centralized communication pattern, which is a strong limitation for scaling on large-scale distributed systems.

The contributions which will be presented in this chapter are as follows:

- The *libWater* programming model, which extends the OpenCL standard by replacing the host code with a simplified and concise interface. It defines a novel device query language (DQL) for OpenCL device management and discovery, and introduces new features such as inter- and intra-platform synchronization.

- A lightweight distributed runtime environment, which dispatches the work between remote devices, based on asynchronous execution of both communications and OpenCL commands. *libWater* runtime also collects and arranges dependencies between commands in the form of a powerful representation called command DAG.

- Two effective uses of the command DAG in order to improve scalability: (a) a Dynamic Collective Replacement (DCR) optimization, which identify collective communication patterns and replaces them with MPI collective operations; (b) a Device-Host-Device Copy Removal (DHDCR), where device-device communications supersedes device-host-device ones. Both optimizations overcome the limitation of the OpenCL host-device semantic, improving scalability on large-scale clusters.

- A study of the scalability of *libWater* on two real production clusters using up to 64 devices. Results show high efficiency and demonstrate the suitability of the presented command DAG optimizations for seven computational application codes. Finally we demonstrate the suitability of *libWater* for a mixed-node cluster for two codes.

In recent years, heterogeneous systems have received a great amount of attention from the research community. Although several projects have been recently proposed to facilitate the programming of heterogeneous clusters [63, 65, 11, 28, 5, 3, 59, 33, 91, 126, 92], none of them combines support for high performance inter-node data transfer, support for a wide number of different devices and a simplified programming model. Our work takes into account all these aspects through the development of the *libWater* library.

Kim et al. [63, 65] proposed the *SnuCL* framework that extends the original OpenCL semantics to heterogeneous cluster environments. Their work is closely related to ours. SnuCL relies on the OpenCL language with few extensions to directly support collective patterns of MPI. Indeed, in SnuCL, it is the programmer responsibility to take care of the efficient data transfers between nodes. In that sense, end users of the SnuCL platform need to have an understanding of MPI collective calls semantics in order to be able to write scalable programs. This deeply differs from our system where such optimizations are transparently applied by the *libWater* runtime system.

Also other works have investigated the problem of extending the OpenCL semantics to access a cluster of nodes. The Many GPUs Package (*MGP*) [11] is a library and runtime system that using the MOSIX VCL layer enables unmodified OpenCL applications to be executed on clusters. *Hybrid OpenCL* [5] is based on the FOXC OpenCL runtime and extends it with a network layer that allows the access to devices in a distributed system. The *clOpenCL* [3] platform comprises a wrapper library and a set of user-level daemons. Every call to an OpenCL primitive is intercepted by the wrapper which redirects its execution to a specific daemon at a cluster node or to the local runtime. *dOpenCL* [59] extends the OpenCL standard, such that arbitrary compute devices installed on any node of a distributed system can be used together within a single application. *Distributed OpenCL* [33] is a framework that allows the distribution of computing processes to many resources connected via network using JSON RPC as communication layer. *OpenCL Remote* [91] is a framework which extends both OpenCL's platform model and memory model with a network

client-server paradigm. *Virtual OpenCL* [126], based on the OpenCL programming model, exposes physical GPUs as decoupled virtual resources that can be transparently managed independently of the application execution.

While the objectives of these approaches are similar to ours, none of them provides an abstraction layer to reduce the complexity associated with the OpenCL development and, furthermore, they show a very limited scalability in clusters of 4 to 8 compute nodes. In particular, none of them employs dynamic communication optimizations as we do.

Besides OpenCL-based approaches, also CUDA solutions have been proposed to simplify distributed systems programming. *CUDASA* [112] is an extension of the CUDA programming language which extends parallelism to multi-GPU systems and GPU-cluster environments. *rCUDA* [31] is a distributed implementation of the CUDA API that enables shared remote GPGPU in HPC clusters. *cudaMPI* [68] is a message passing library for distributed-memory GPU clusters that extends the MPI interface to work with data stored on the GPU using the CUDA programming interface. All of these approaches are limited to devices that support CUDA, i.e. NVIDIA GPU accelerators, and therefore they cannot be used to address heterogeneous systems which combine CPUs and accelerators from different vendors.

Other projects have investigated how to simplify the OpenCL programming interface. Sun et. al [113], proposed a task queueing extension for OpenCL that provides a high-level API based on the concepts of work pools and work units. *Intel CLU* [54], *OCL-MLA* [87] and *SimpleOpencl* [108] are lightweight API designed to help programmers to rapidly prototype heterogeneous programs. DIANA [94] provides a common interface to hide the complexity of managing different application programming interfaces APIs and libraries for different many-core devices. OmpSs [22] relies on user directives to avoid the boilerplate OpenCL host code configuration and generate a DAG for task scheduling purpose. FastFlow [4] is a structured parallel programming framework targeting clusters of multi-core workstations. StarPU [9] provides a runtime and a programming language extensions to support task-based programming model in a cluster. Besides the simplified interface, *libWater* differently from other approaches

provides fine-grained control over device selection (i.e. DQL) and an improved device synchronization based on events.

5.3 THE LIBWATER PROGRAMMING INTERFACE

libWater is a C/C++ library-based extension of the OpenCL programming model that simplifies the development of distributed heterogeneous applications. It inherits the main principles from OpenCL trying to overcome its limitations. While maintaining the notion of host and device code, *libWater* exposes a very simple programming interface based on four key concepts: *device*, *buffer*, *kernel* and *event*. A *device* represents a compute device but, differently from the original paradigm, this single object is an abstraction of the OpenCL platform, device, and queue concepts. Such simplification reduces the number of source code lines necessary for the initialization of the devices, and thus avoids the boilerplate configuration code that is usually present in every OpenCL program. Furthermore, the library is not restricted to a single node but, taking internally advantage of the message passing model, it provides access to devices on remote nodes as if they were locally available.

Since *libWater* can grant access to a large number of distinct devices, the selection of a particular one can be cumbersome. In order to simplify this important aspect, *libWater* introduces a novel domain specific language for querying devices. A *device query language* (DQL) query statement follows an SQL-like structure, that is composed of 4 basic clauses with the following syntax:

```
SELECT          [ALL | TOP k | POS i]
    FROM NODE   [n [, ...]]
    WHERE       [restrictions attribute values]
    ORDER BY    [attribute [, ...]]
```

The `SELECT` clause (the only one which is mandatory) respectively allows the selection of all the devices, the first top `k`, or a particular device from the device list generated under the restrictions on the following clauses. With `FROM NODE` a single node or a list of nodes can be specified narrowing the range of selectable devices to those particular nodes. The clauses `WHERE` and `ORDER BY` allow the control of

Device Management (**wtr_**)	
`void` **init_devices**`('DQL', ...)`	`device` **get_device**`('DQL', ...)`
`int` **get_num_devices**`()`	`void` **release_devices**`()`
`void` **print_device_infos**`(device)`	
Buffer Management (**wtr_**)	
`buffer` **create_buffer**`(device, mem_flag, size, evt)`	
`void` **write_buffer**`(buffer, size, source_ptr, wait_evt, evt)`	
`void` **read_buffer**`(buffer, size, dest_ptr, wait_evt, evt)`	
`void` **release_buffer**`(buffer, wait_evt, evt)`	
Kernel Management (**wtr_**)	
`kernel` **create_kernel**`(device, name, kernel_name, build_options, flag, evt)`	
`void` **run_kernel**`(kernel, work_dim, global_size, local_size, wait_evt, evt, num_args, ...)`	
`void` **release_kernel**`(kernel, wait_evt, evt)`	
Event Management (**wtr_**)	
`event` **create_event**`()`	`void` **release_event**`(evt)`
`event` **merge_events**`(num, ...)`	`void` **wait_for_events**`(num, ...)`
`void` **init_event_array**`(num, evt)`	
`void` **release_event_array**`(num, evt)`	

Table 5.1: The complete *libWater* API

the device restrictions on attribute values and the order in which the devices will be returned. The possible attribute values are currently those exposed by the OpenCL *GetDeviceInfo* function. A DQL use case is shown and discussed in Section 5.4. DQL queries can be used for both device initialization and device selection. The latter must be a subset of the former and since *libWater*'s device concept represents a single device only, the function `wtr_get_device` only accepts queries that make use of the `POS` clause.

Table 5.1 presents the *complete* API of the *libWater* library. The prefix `wtr_` and the C language pointer syntax has been removed from the table for readability reasons. Initialization and selection of devices is done, respectively, by using the `wtr_init_devices` and the

`wtr_get_device` routines. Once a *device* is created, it is possible to allocate data and execute computation on it. In *libWater*, this is done through the use of the *buffer* and the *kernel* concepts. These two objects are similar to their respective OpenCL versions, with the main difference that, during their creation, they are bound to a specific device. For this reason, no device must be specified for buffer and kernel related functions. The principal kernel functions are `wtr_create_kernel` and `wtr_run_kernel`. The former receives as parameter a `flag` that specifies whether the `name` input argument contains the kernel code or it is the name of a file containing the OpenCL kernel. The latter is used for executing a kernel in the previously bound device. The parameters *work_dim*, *global_size* and *local_size* are the same specified in the OpenCL *EnqueueKernel*. The `num_args` parameter states the number of input arguments accepted by the kernel. This parameter is followed by a list of a variable number of pairs. Each pair consists of a size (in bytes) and a pointer to the corresponding kernel argument. The first value of the pair distinguishes between buffers – when is equal to 0 – or a valid address in the host memory. The fourth concept in *libWater* is the *event* object. Most of *kernel* and *buffer* functions have one or two parameters called `wait_evt` and `evt`. The latter is an output argument which is used by the invoked command to generate an event object. If not specified, *libWater* assumes blocking semantics for the routine. The former specifies the event object on which the execution of the command depends. If not present, the command has no dependencies and thus it can be immediately executed. Since there can be a dependency between several commands, the `wtr_merge_events` function can be used to merge multiple event objects into one.

The last major difference between *libWater* and the OpenCL model is the fact that initialization and release of buffers and kernels can be invoked using a non-blocking semantics. The main reason for this is to increase the amount of operations that the runtime system can overlap. In the next section we explain how dependency information enforced by events are then exploited by *libWater*'s runtime system.

While the main focus of the programming interface of *libWater* is on simplicity and productivity, the underlying runtime system aims at low resource utilization and high scalability. Calls to *libWater* routines are forwarded to a distributed runtime system which is responsible for dispatching the OpenCL commands to the addressed devices and for transparently and efficiently moving data across the cluster nodes. The *libWater* distributed runtime is written in C++ and internally uses several paradigms, such as pthreads, OpenMP and MPI for parallelization.

Figure 5.1 shows the organization of the *libWater* distributed runtime system. The *host code*, which directly interacts with *libWater*'s routines, runs on the so called *root node*, which by default is the cluster node with rank 0. This thread will be referred to as the *host thread*. In the background, a second thread, i.e. the *scheduler thread*, is allocated to execute an instance of the `WTRScheduler`. On the remaining cluster nodes, a single *scheduler thread* is spawned independently of the number of available devices (only one MPI process is allocated per node). This thread executes an instance of the `WTRScheduler` which represents the backbone of *libWater*'s distributed runtime system.

Each `WTRScheduler` continuously dequeues `wtr_commands` from the local command queue. `wtr_commands` in the system are generated in two ways, either by (*i*) *libWater*'s routines (step ❶), or (*ii*) by delegation from the root scheduler (step ❸). Calls to the *libWater*'s interface are converted into command descriptors (i.e. command design pattern) and immediately enqueued into the root node local command queue (step ❶) of Figure 5.1. Since all `wtr_commands` are generated by the root node itself, we refer to its queue as the runtime *global* command queue.

`wtr_commands` are either wrappers for OpenCL commands or data transfer jobs (i.e. `send_job` or `recv_job`) which are generated by the library routines whenever the device addressed by a read or write buffer operation is located in a remote (i.e. $\texttt{rank} \neq 0$) compute node. The descriptor of a `wtr_command` is *self-contained* since it carries all the information necessary for its execution. To be portable across cluster nodes, OpenCL objects such as kernels, buffers, and events

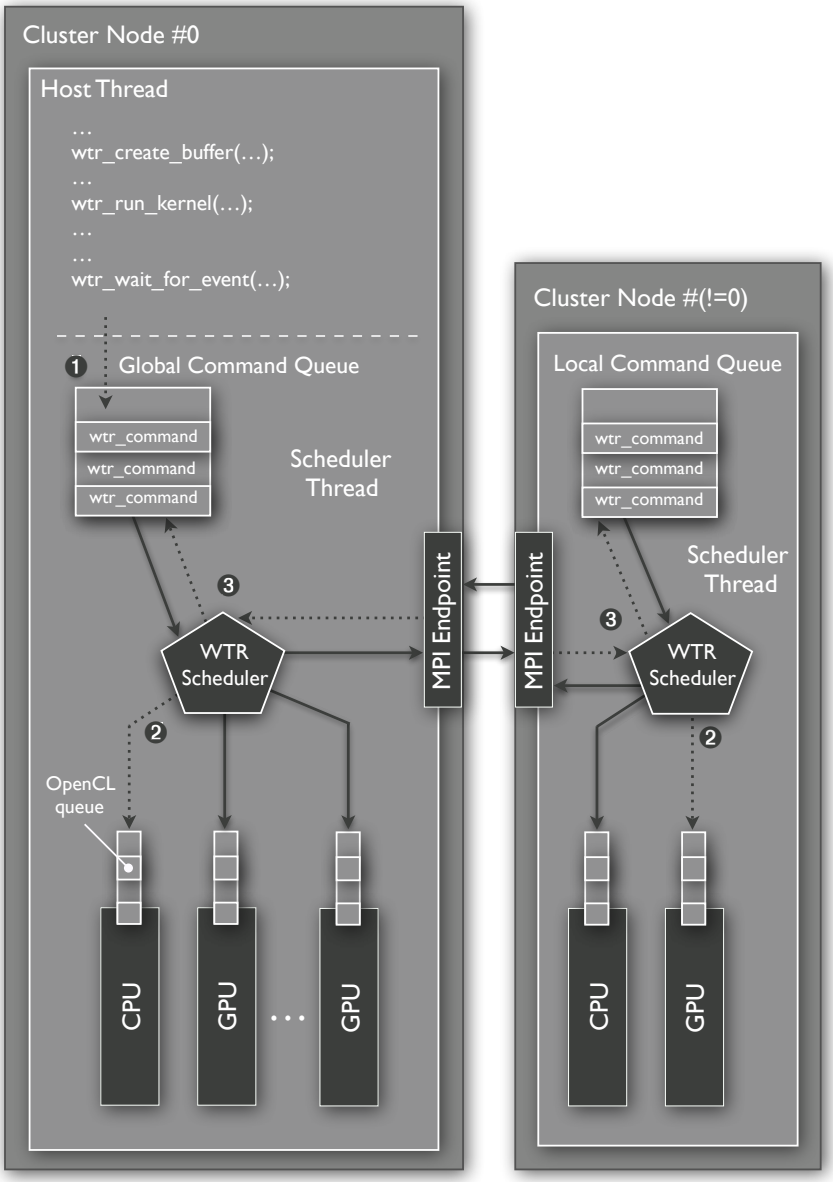


Figure 5.1: *libWater*'s distributed runtime system architecture

are identified, within the `wtr_command` object, by a unique ID. The root scheduler continuously fetches the `wtr_commands` from the global command queue, decodes its content and – depending on the targeted device – dispatches the command to the correct node. When the `wtr_command` addresses one of the local OpenCL devices, the corresponding OpenCL command is created and enqueued into the device command queue (step ❷). When a remote OpenCL device is addressed, an MPI message is generated – serializing the content of the `wtr_command` descriptor – and dispatched to the cluster node host-

ing the requested device. The `WTRScheduler` of the target node then de-serializes the `wtr_command` and, instead of immediately executing it, enqueues the `wtr_command` instance into the local command queue (step ❸). The same `WTRScheduler` is then responsible to dispatch the corresponding OpenCL command into one of its local device queues (step ❷).

The heartbeat of the `WTRScheduler` is an advanced event system which allows the management of an entire compute node – hosting multiple OpenCL devices – using only a single application thread. Indeed, because one instance of the `WTRScheduler` runs on every cluster node, trying to keep the resource usage as low as possible is of paramount importance in order to avoid wasting CPU cycles which can be used to run an OpenCL kernel. Different from related work, which exclusively reserves an entire cluster node and a physical CPU core in each compute node only for scheduling purposes, our system does not exclusively reserve any user resources for scheduling. Furthermore, using a single thread, for both executing local `wtr_commands` and for performing scheduling decisions, reduces the amount of synchronization since accesses to events and the command queues do not need to be synchronized.

Relying on a single thread can, however, easily become a performance bottleneck. An interesting example is the interaction with MPI routines. By default many MPI implementations implement blocking behaviour with a *spin-lock* mechanism in order to minimize latency. This means for example that a blocking receive, waiting for a message from the communication channel, continuously checks for incoming data usually saturating the cycles of a CPU core. In an environment like ours, where CPU cores may be used to run OpenCL kernels, this behaviour must be avoided. Our solution is to avoid in every event handler routine any call to blocking MPI or OpenCL routines and always use the non-blocking semantics. The main idea is the creation of periodic events, handled by the event system using a priority queue based on timestamps, to check for the completion of pending operations. For OpenCL routines, we exploit the OpenCL event system and the associated callback mechanism. In this way, the `WTRScheduler` is able to dispatch several commands on the OpenCL devices, or MPI data transfers, which although being issued sequentially (by the single flow of the execution) are concurrently executed by the available

```
1  wtr_init_devices("SELECT ALL WHERE (type = gpu AND vendor = nvidia)");
2  wtr_event* evts[2];
3  for (int i=0; i<2; ++i) {
4    size_t offset=size/2*i;
5    wtr_device* dev = wtr_get_device("SELECT POS 1 FROM NODE %d
6                                      WHERE global_memory > 1024MB",i);
7    assert(dev != NULL && "Device does not exist!");
8    wtr_event* e[8];
9    wtr_init_event_array(7,e);
10   wtr_kernel* kern = wtr_create_kernel(dev,"kernel.cl","fun", "", WTR_SOURCE, e+0);
11   wtr_buffer* buff = wtr_create_buffer(dev, WTR_MEM_READ_WRITE, size/2, e+1);
12   wtr_write_buffer(buff, size/2, ptr+offset, e+1, e+2);
13   e[7] = wtr_merge_events(2, e+0, e+2);
14   wtr_run_kernel(kern,1,(size_t[1]){size/2},NULL,e+7,e+3,2,
15                0, buff,
16                sizeof(size_t), &offset);
17   wtr_read_buffer(buff, size/2, ptr+offset, e+3, e+4);
18   wtr_release_buffer(buff, e+4, e+5);
19   wtr_release_kernel(kern, e+3, e+6);
20   evts[i] = wtr_merge_events(2, e+5, e+6);
21   wtr_release_event_array(8, e);
22 }
23 /* Blocks until buffers and kernels are released */
24 wtr_wait_for_events(2, evts+0, evts+1);
25 wtr_release_event_array(2, evts);
```

Listing 5.1: A complete multi-device program example using *libWater*'s routines

resources (i.e. OpenCL devices and the network controller). The same event-based technique utilized to manage multiple OpenCL devices in a single node is also exploited on the large scale across cluster nodes.

As already explained in the previous section, *libWater* puts a strong emphasis on events. Following the semantics of OpenCL, dependency information enforced by programmers are used to select `wtr_commands`, which can be safely enqueued into one of the cluster nodes. *libWater* provides an event object, i.e. `wtr_event`. Internally, `wtr_events` are mapped either to an OpenCL *event* object, or to a `wtr_command` identifier which is automatically generated for each `wtr_command` enqueued into the system. These dependencies allow the runtime system to organize enqueued `wtr_commands` into a DAG.

A complete multi-device *libWater*-based host program is shown in Listing 5.1. This code initializes all the available NVIDIA GPU devices. It then selects two devices belonging respectively to node rank 0 and 1, with a global memory larger than 1024MB. For each device the code in Listing 5.1 does the following: create a kernel (i.e. `kern`,

in line 10) and a read/write buffer (i.e. `buff`, line 11). Then the contents from the host memory is written into the device buffer by the `wtr_write_buffer` command (line 12) and the `wtr_run_kernel` command is issued providing `buff` as an input argument (lines 14-16). The computed result is then retrieved by the `wtr_read_buffer` command (line 17) which moves data from the device memory back to the host memory. From the runtime system point of view, the execution of the previous code generates a set of dependent commands structured as the DAG depicted in Figure 5.2. The DAG $G(V, E)$ is composed of vertices, i.e. `wtr_commands` $\in V$, interconnected through directed edges $(a, b) \in E \,|\, a, b \in V$, or *events*, which guarantee that the correct order of execution, and therefore the semantics of the input program, is maintained. The *set* of dependencies associated with a command $c \in V$ is defined as $c.deps = \{v \in V \,|\, (v, c) \in E\}$. It is worth mentioning that not all *libWater* library routines generate a corresponding `wtr_command`. For example, creation, merging and release of events are only meaningful in the root node, therefore there is no need for serializing them. In Figure 5.2, each `wtr_command` carries a descriptor in the form $x|y$ where x represents the node rank, $c.node\_id$, on which the targeted device, $c.dev\_id$, is hosted and y is the unique command identifier assigned by the runtime system. As already mentioned, for buffer operations on remote devices (i.e. device on node 1) explicit data transfers are automatically inserted by the *libWater* library (e.g. `wtr_commands` 10 and 14).

Events determine when a `wtr_command` can be scheduled for execution. The scheduler uses a *just-in-time* strategy to select the next `wtr_command` from the local command queue. The logic works as follows: enqueued `wtr_commands` are analyzed in a FIFO fashion and, for each ready command, the scheduler checks whether dependencies – explicitly specified by event objects – are satisfied. If a command has no dependencies, it can be executed. Since the host program generates all the commands solely on the root node, scheduling is done at this node. However, a centralized scheduler on a single node is not an effective strategy since it limits command throughput and thus the overall scalability of the system.

In order to solve this problem, we rely on the fact that the OpenCL runtime system already has the capability of scheduling commands and handling dependencies by using events. It is worth noting that

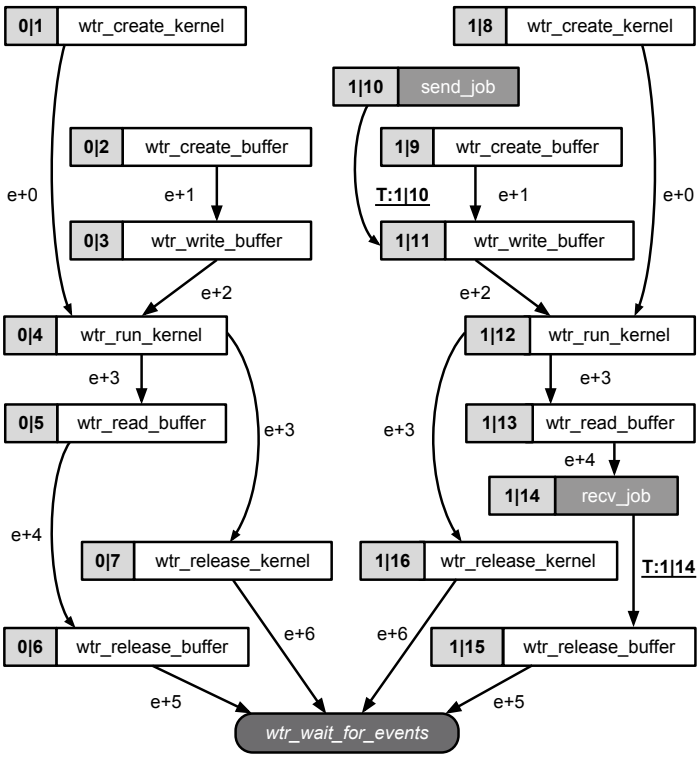


Figure 5.2: DAG of `wtr_commands` generated during the execution of the code snippet in Listing 5.1

in OpenCL this mechanism is limited since events cannot be used to perform command synchronization across different platforms. *libWater* unifies event handling through `WTRScheduler` instances which manage inter-platform synchronization and offload intra-platform synchronization to the OpenCL driver.

We implemented a *three-level hierarchical scheduling* approach as described in Algorithm 5.1. At the top level, the root node of the *libWater* runtime system *pro-actively* schedules `wtr_commands` from the global queue to the targeted cluster nodes. cmd, fetched from the command queue, is sent to the target node (i.e. $cmd.node\_id$) only if each of its dependent commands (i.e. the set $cmd.deps$) are to be executed on the same remote node (lines 6–9). The second level scheduling is local to each node (lines 11–14). The scheduler checks whether cmd only depends on `wtr_commands` addressing the same OpenCL device. In such case, the command is enqueued into the corresponding device queue (i.e. $dev.dev\_id$) and dependencies are mapped to local OpenCL events. Alternatively, if a `wtr_command` C_1 depends on a second `wtr_command` C_2, scheduled in another platform

Algorithm 5.1 The `WTR_Scheduler`'s algorithm

```
1: cmd_queue                                   ▷ Local FIFO wtr_command queue
2: my_rank                                                  ▷ MPI process rank
3: while true do
4:   cmd ← cmd_queue.pop();
5:   if cmd.node_id ≠ my_rank then
6:     if ∀ d ∈ cmd.deps | d.node_id = cmd.node_id then
7:       send(cmd, cmd.node_id, SCHED)              ▷ Delegates cmd to node
8:       continue
9:     end if
10:  else
11:    if ∀ d ∈ cmd.deps | d.dev_id = cmd.dev_id then
12:      issue(cmd.cl_cmd, cmd.deps)               ▷ Delegates to corresp. dev.
13:      continue
14:    end if
15:  end if
16:  cmd_queue.push(cmd)              ▷ Failed to schedule event due to deps.
17: end while
```

Algorithm 5.2 Update `wtr_command` dependencies

```
1: function CALLBACK_CMD_COMPLETION(c)
2:   for cmd in cmd_queue do
3:     cmd.deps.remove(c)                  ▷ Removes c from the dependencies
4:   end for
5:   if my_rank ≠ 0 then
6:     send(c, 0, DONE)                ▷ Notifies the root node of c completion
7:   end if
8: end function
```

(of the same node), the local `WTRScheduler` ensures that C_1 is not enqueued into the OpenCL device queue before C_2 is completed. The third-level scheduling is implemented by the OpenCL runtime system itself which is responsible of managing single device queues. If cmd cannot be scheduled, due to unsatisfied dependencies, then it is pushed back in the command queue.

Dependencies are automatically updated when a `wtr_command` c completes. Locally, a *command completion event* is generated. The associated *callback* function is depicted in Algorithm 5.2. The function removes, for every command in the local queue, any dependence on c. Additionally, nodes notify the root scheduler with a message (lines 5–7) triggering a similar completion event internally at node 0. In such a way, commands in the global queue waiting for the completion of c can be scheduled – depending on the targeted device – either to a local device or to a remote node.

This multi-level scheduling allows the runtime system to hide the costs of the scheduling, as well as data transfers, with the actual work being done by the devices in the background. The main idea is to use non-blocking semantics when OpenCL commands are scheduled in the corresponding devices. In this way, the `WTRScheduler` can continuously dispatch commands to other devices or move data from and to the root node. In the example in Figure 5.2, commands `0|1` and `0|2` can be executed in parallel. Events at addresses $e+0$ and $e+1$ are handled by the root `WTRScheduler` since the OpenCL standard does not allow non-blocking semantics for these operations. The remaining commands (i.e. `0|3`, `0|4` and `0|5`) are inserted asynchronously into the OpenCL device queue of node 0, upon completion of commands `0|1` and `0|2`. Events `e+2` and `e+3` are therefore handled directly by the OpenCL runtime system. Following the same logic, `wtr_commands` addressing the second OpenCL device (i.e. $1|*$) are sent to the node with rank 1. The blocking function `wtr_wait_for_events` stops the execution of the host until the release operations on both nodes have completed.

5.4.1 *Runtime System Optimizations*

The underlying architecture of the *libWater* runtime system and the emphasis on events, promoted by its interface, enables several runtime optimizations which are transparent to the user. This capability is a direct consequence of adhering to the OpenCL queuing semantics. Indeed, while commands are being enqueued into the system, a command DAG (as shown in Figure 5.2) is internally created. Since OpenCL issues commands to the appropriate device only when an explicit flush is invoked by the programmer, the runtime system can analyze large portions of the application DAG and optimize it for improving scalability.

An optimization which has been implemented in the *libWater* runtime system is the dynamic detection and replacement of collective communication patterns (`DCR`). Whenever the addressed device is not hosted in the root node, a call to `wtr_write_buffer` and `wtr_read_buffer` respectively generates an MPI send and receive operation. When an OpenCL application is distributed among all avail-

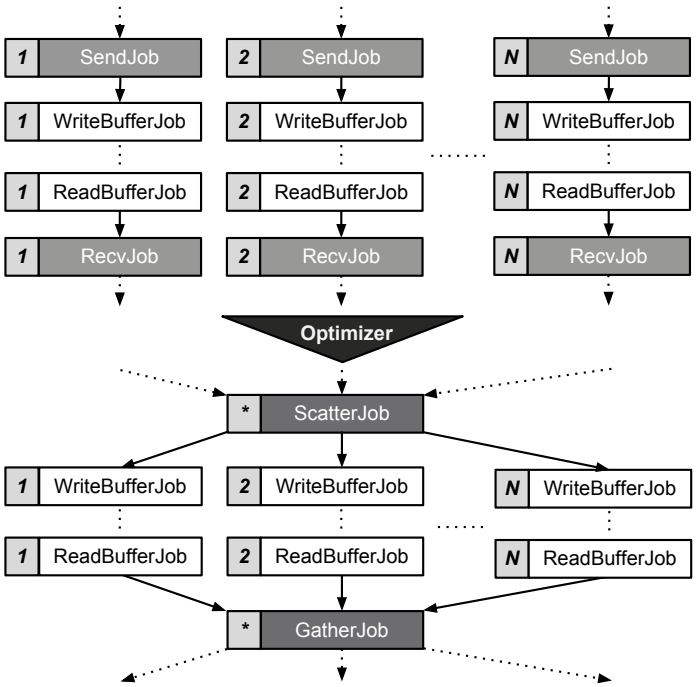


Figure 5.3: DCR *libWater* runtime DAG optimization

able devices, input buffers are usually either split or replicated between compute nodes. This parallelization strategy is common and it results in a DAG containing several send/receive transfer operations for every device of the cluster. An example is depicted in Figure 5.3 which represents a realistic DAG resulting from the splitting of an input and output buffer among a set of OpenCL devices.

Point-to-point data transfers performed by the *libWater* runtime system imply an increased latency when compared with the native MPI send or receive routines. The reason for that is the polling mechanism implemented by the *libWater* runtime system – mainly employed to save node resources – which replaces the spin-lock mechanism commonly used by MPI libraries. Additionally, the number of required data transfers is directly proportional to the cluster nodes (and thus devices). This results in a large number of commands being dispatched by the runtime system and consecutively negatively impacts the overall scalability. MPI offers a large set of communication patterns called *collective operations* [83]. These routines are highly efficient since nearly all modern supercomputers and high-performance networks provide specialized hardware support for collective operations [67]. Additionally, the implementation of such collective op-

erations employs dynamic runtime tuning techniques which choose, among a set of semantically equivalent algorithms, which best fit the underlying network topology and architecture [21, 95, 106].

Related work analyzed the problem of automatic detection of collective patterns from a set of point-to-point communications. This technique is common in MPI performance tools which are capable of detecting such patterns via post-mortem analysis of program traces. The general problem of collective communication pattern detection is NP-hard, however, under particular restrictions, the problem can be solved in polynomial time. A more recent work [46] proposed a fast solution, with a complexity of $\mathcal{O}(n \log n)$, which makes the approach more suitable for runtime systems.

The goal of our `DCR` optimization algorithm is to analyze the command DAG isolating point-to-point data transfers and detect whether a subset of those resembles one of the collective patterns supported by MPI. This is possible since – if the application is carefully written using events for command synchronization – the command DAG will be available to the runtime system scheduler before the first blocking command is invoked (e.g. `wtr_wait_for_event(s)`). Since data transfers in our environment have all the same root (the node 0), the analysis for patterns is simplified.

The optimization algorithm is composed of two phases. First, the command DAG is traversed and all the transfer commands are collected into n separate lists, one per device. Second, on the extracted n lists, pattern analysis is performed. The collective pattern check is done by considering elements having the same position within the transfer job lists. Furthermore, the check is simplified by the fact that every send and receive `wtr_command` carries information of the buffer location (*buf*) and the amount of bytes being transferred (*size*). The pattern analysis starts by taking the first transfer `wtr_command` from the n lists and by checking against a supported pattern, i.e. *broadcast*, *scatter* or *gather*. For instance, in a broadcast n send operations are expected where $\forall\, i \mid 0 \leqslant i < n-1,\ buf_i = buf_{i+1} \vee size_i = size_{i+1}$. If the check fails, the transfer jobs are tested against a scatter or gather pattern $\forall\, i \mid 0 \leqslant i < n-1,\ buf_i + size_i = buf_{i+1}$.

Once a pattern is recognized, single point-to-point transfers are removed from the command DAG and replaced by the corresponding collective communication operation. A visual example of this opti-

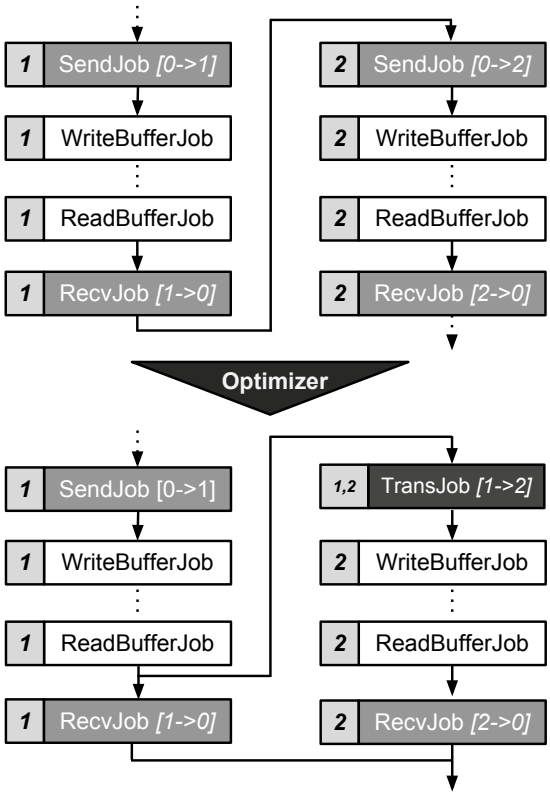


Figure 5.4: DHDCR *libWater* runtime DAG optimization

mization is depicted in Figure 5.3, where multiple send operations are collapsed into a single scatter operation and correspondingly, receives are rewritten as a gather operation. By doing so, dependencies between successive commands are updated in order to keep the semantics of the input program unchanged.

Since collective operations must involve all the processes in a communicator, the current implementation of the `DCR` optimization works when all the initialized devices participate in the computation. Therefore, the analysis is limited to regular applications which *must* involve all OpenCL devices in data transfers. This is important to keep the pattern recognition algorithm simple and fast since this optimization is applied during runtime.

Another optimization which has been implemented as part of the *libWater* runtime systems is the detection and optimization of *device-host-device* copy patterns. As the *libWater* API closely matches the OpenCL host-device model, it does not include any device-device communication. This limitation is based on the OpenCL model which does not include functions operating on different platforms. However,

on distributed computing environments, this limitation imposes the use of centralized host-device instead of more efficient device-device distributed communication.

An example of this problem arises when a buffer which has been distributed over n devices to be used as the output in a first kernel, is later used as input of one or multiple devices of a second kernel. For instance, let's consider the matrix chain multiplication $ABCD$. As matrix multiplication is associative, we can compute first AB, then CD, and finally the product $(AB)(CD)$. While the first two multiplications work normally, the latter requires device-host-device communications that drastically affects scalability.

To address this issue, we implemented a new optimization which attempts to replace similar device-host-device communications with direct device-device data transfers. This optimization, called device-host-device copy removal (`DHDCR`) is implemented as follows. Whenever an application contains call to `wtr_write_buffer` and `wtr_read_buffer` involving devices not belonging to the root node, *libWater* generates MPI send and receive operations. If a sequence of write, read, write occurs on the same buffer (or on part of the same buffer) then this sequence is a candidate for optimization. Once the pattern is recognized, the two consecutive device-to-host and host-to-device transfers are removed from the command DAG and replaced by a single device-to-device transfer. A visual example of this optimization is depicted in Figure 5.4. The `TransJob`, generated by the `DHDCR` optimization, is a `wtr_command` which the root scheduler dispatches on both nodes involved in the data transfer (node 1 and 2 in the example), the other nodes are not involved. However, in order to maintain the host semantics of the program unchanged, the updated value of the buffer (generated by node 1) must also be copied back on the host node. Therefore a `RecvJob` command is generated to collect the buffer. The main difference with the original code is that this operation can be completely overlapped with the execution of the second kernel on the node rank 2.

Note that simple applications such as the ones listed in Table 5.2, only show a simple pattern (write, run kernel, read) and do not show any possibility to apply `DHDCR`. However, more complex applications are usually consisting of several kernels, with nontrivial inter-node

Application	OpenCL LOC	*libWater* LOC	Input size	In/Out buffers (*splittable*)	Short Description
`PerlinNoise`	412	**301**	20K x 20K	0(0) / 1(1)	Gradient Noise Generator
`Nbody`	450	**324**	600K bodies	2(0) / 2(2)	N-body Simulation
`kNN`	234	**101**	*ref*: 8M, *query*: 80K	2(1) / 2(2)	k-Nearest Neighbor
`Floyd`	222	**113**	Vertices 8K, Adjacency matrix 64K	1(0) / 1(1)	Floyd-Warshall
`MatrixMul`	219	**104**	7K x 7K (A = B = C)	2(1) / 1(1)	Matrix Multiplication
`LinReg`	298	**149**	1000K	4(2) / 1(1)	Linear Regression

Table 5.2: Application codes used for *libWater* evaluation

data transfers, and are more suitable for this optimization (e.g. matrix chain multiplication).

5.5 EVALUATION

We used *libWater* to encode 6 computational kernels, some of them taken from various OpenCL benchmarking suites (i.e. AMD and IBM), and studied their scalability. In four of them, the kernels were optimized for local memory, i.e. `PerlinNoise` (from IBM), `Nbody` (from AMD), `Floyd` and `kNN` manually written by us. For the remaining two codes, `MatrixMul` and `LinReg` we used a naive implementation unoptimized for what concern local memory. Table 5.2 shows, for each kernel, the number of input and output buffers used by the kernel. We define a buffer as *splittable* when its content can be distributed among the devices. The nature of a buffer is strictly related to the algorithm being implemented within the OpenCL kernel, and thus the application. Non-splittable buffers are always replicated on every device. All six applications utilized for our study do not contain unsplittable output buffers. In the presence of such buffers, the merge of the result coming from different devices would generate memory consistency issues that *libWater* is currently not able to handle. Table 5.2 also shows the reduction, in terms of lines of code, achieved when the application is written using our library. It is worth mentioning

Site	Vienna Scientific Cluster	BSC
Cluster	VSC2	MinoTauro GPU Cluster
Max # of Nodes	1314	128
CPUs	2 x AMD Opteron 6132 HE	2 x Intel Xeon E5649
Cores per Node	2 x 8	2 x 6
Clock Frequency	2.2 GHz	2.5 GHz
Memory per Node	32 GB DDR3	24 GB DDR3
GPUs	–	2 x NVIDIA M2090
Interconnection	Infiniband 4x QDR	Infiniband 4x QDR
Open MPI Version	1.6.1	1.6.1
OpenCL Version	AMD APP 2.6	CUDA 4.1

Site	University of Innsbruck		
Cluster	Ortler		
Nodes	mc5	mc6	mc7
CPUs	2 x Intel Xeon E5-2690v2		
Cores per node	2 x 10		
Clock Frequency	3.0 GHz		
Memory per Node	128 GB DDR3		
GPUs or ACLs	2 x AMD FirePro S9000	2 x NVIDIA Tesla K20m	2 x Intel Xeon Phi 7120P
Interconnection	Infiniband 4x QDR		
Open MPI Version	1.6.5		
OpenCL Version	AMD APP 2.9	CUDA 5.5	XE 2013 R3

Table 5.3: The experimental target architectures

that while the original OpenCL applications were single device codes, the *libWater* based implementation is instead multi-device code. On average, we were able to reduce the lines of the host code by approximately a factor of 2 due to the higher level abstractions provided by *libWater*.

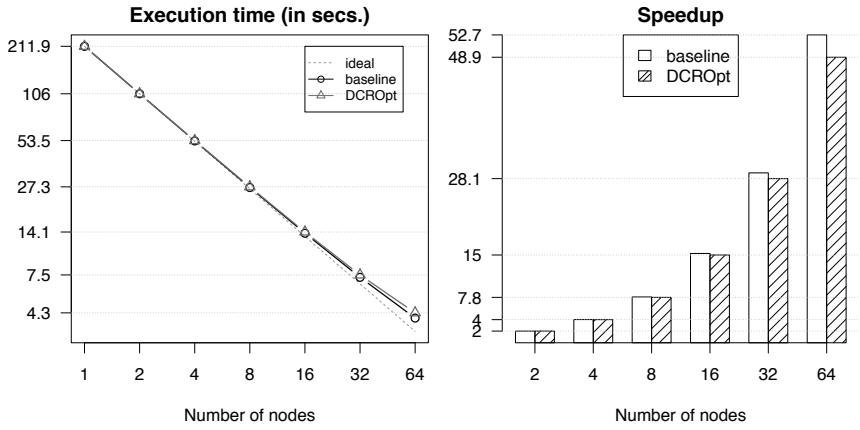
For the scalability analysis, we used two large-scale production clusters, the Vienna Scientific Cluster VSC2 [119] and the MinoTauro Barcelona Supercomputing Center GPU Cluster [117]. A second study was conducted to test the suitability of *libWater* to exploit the computational capabilities of a mixed-node cluster configuration. For this purpose, we used the Ortler Cluster at the University of Innsbruck, composed of three heterogeneous compute nodes (i.e. mc5, mc6 and mc7). The hardware details of the clusters are depicted in Table 5.3.

5.5.1 *VSC2 CPU Cluster*

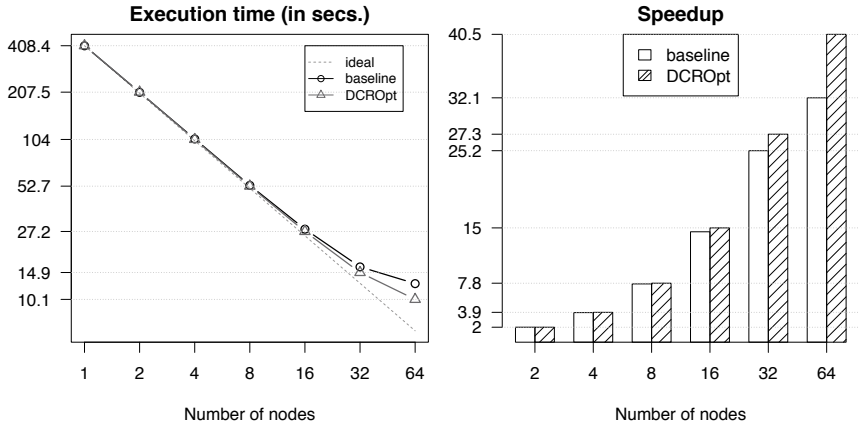
The applications shown in Table 5.2 were executed on the VSC2 CPU cluster. We were able to access up to 64 nodes with a total of 1024 CPU cores. Since the 2 AMD CPUs which are hosted per node are considered by the OpenCL driver as a single device, the speedup was computed based on the number of compute nodes (and thus OpenCL devices) instead of single CPU cores. The workload partitioning is implemented, for each test case, by assigning to each OpenCL device an equal amount of work.

The scalability tests were performed in the following way: the original OpenCL versions of the applications were executed on a single node and their execution times used as a reference measurement. *libWater* was then used for node numbers ranging from 2 to 64. The main differences between the original version of the application codes and the one written using *libWater* are mainly in the host code. The kernel code was slightly modified only to forward the *offset* value used by the workload partitioning (as shown in Listing 5.1). We computed the ideal scaling for each application using the reference execution time and dividing it by the number of nodes. We conducted experiments with *libWater* by using two different settings: the first, named *baseline*, uses the runtime system without dynamic optimizations enabled; the second, `DCR`, uses the collective pattern replacement mechanism as described in Section 5.4.1. The results of our experiments are depicted in Figure 5.5 and Figure 5.6.

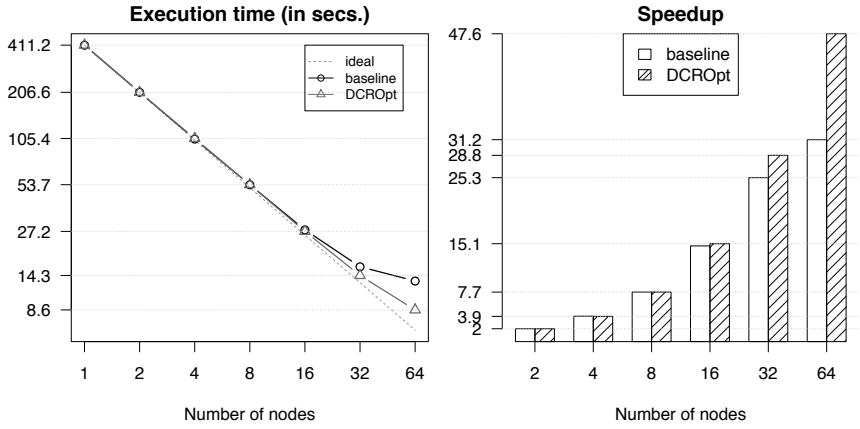
For each of the six applications, we show the execution time (in seconds) for up to 64 nodes and the corresponding speedup with respect to a single node. Overall, we observe that our approach scales almost linearly, especially for those codes using few input/output buffers. `PerlinNoise`, Figure 5.5a, is an example of those since it has no dependencies on input buffers and the data produced by the kernel is distributed between the devices. For such code, the baseline configuration of our runtime system achieves a speedup of 53 for 64 nodes, and thus an efficiency of 83%. When the number and size of the input/output buffers increases, the efficiency of our system decreases. The worst case is represented by the `LinReg` application, Figure 5.6c, which stops scaling after 32 nodes. This kernel has 4 input buffers, 2



(a) `PerlinNoise`



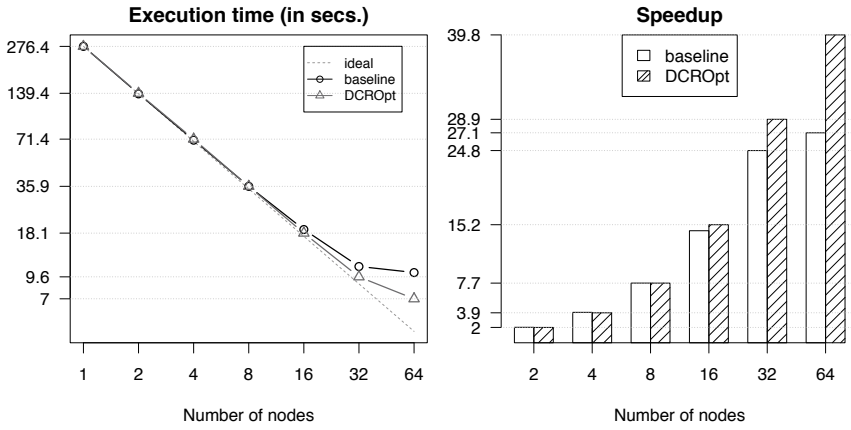
(b) `Nbody`



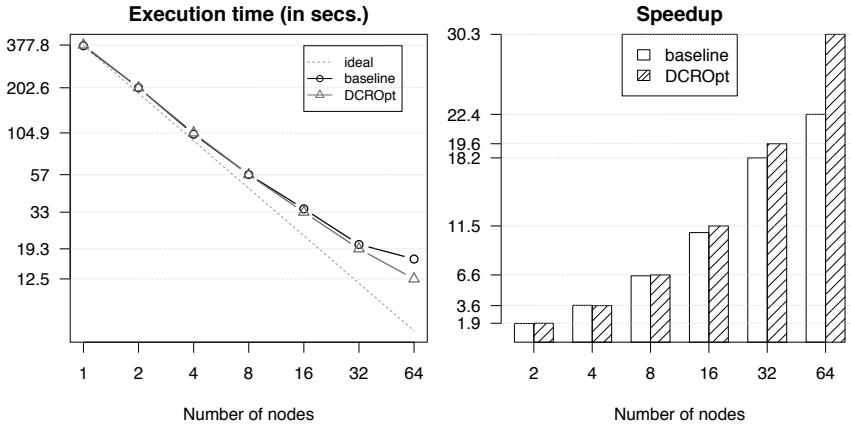
(c) `Floyd`

Figure 5.5: Strong scaling of *libWater* on the VSC2 (1 of 2)

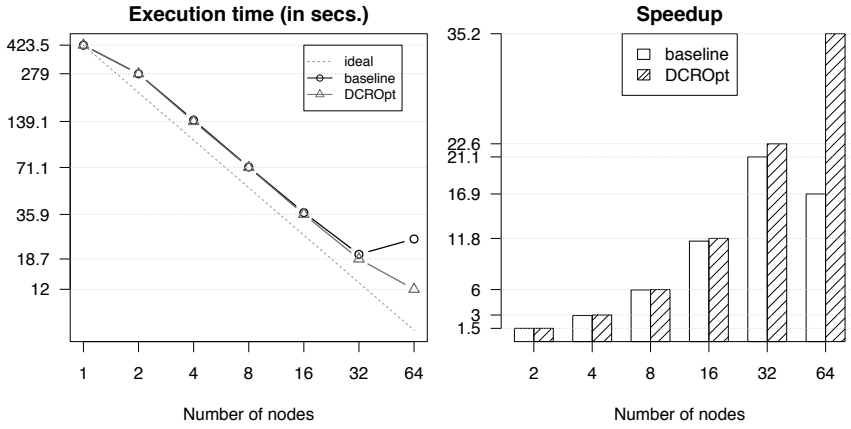
of them are not splittable (because of dependencies within the kernel code) and therefore must be replicated on every node. The remaining 2 input and output buffers are instead splittable. For such code,



(a) `kNN`



(b) `MatrixMul`



(c) `LinReg`

Figure 5.6: Strong scaling of *libWater* on the VSC2 (2 of 2)

we have an immediate decrease (75% on two nodes) of the efficiency. This is because the kernel execution is delayed due to the fact that several `wtr_commands` are executed (and transferred to the target nodes)

to create and initialize the input/output buffers. However, this delay is a constant and system efficiency remains almost unvaried up to 16 nodes. On 32 and 64 nodes the efficiency of the baseline runtime system starts decreasing significantly.

This problem is largely addressed by the dynamic collective pattern replacement, i.e. `DCR`, optimization which was introduced in Section 5.4.1. This optimization reduces the load on the scheduler since it replaces several single transfer jobs with one collective operation. In `LinReg` this optimization improves the scalability of the system by a factor of 2 achieving an efficiency of 55%. A small effect of this optimization can be observed for smaller node configurations because collective operations are optimized for a large number of nodes. An interesting result is the effect of the `DCR` optimization on the `PerlinNoise` test case. In such a case, the `DCR` optimization fails to improve performance over the baseline. The reason is that collective operations are blocking while point-to-point communications in the runtime system are non-blocking thereby allowing overlapping of multiple transfers. The synchronization costs introduced by the gather operation is therefore not properly compensated by the amount of exchanged data. We believe that this problem can be eliminated by using *non-blocking collective* routines which have been introduced in the latest MPI standard [83]. Additionally, since this optimization is done dynamically, and therefore the amount of data being transferred is known by the scheduler, heuristics can be integrated to decide when such optimization should be applied.

On average, *libWater* achieves an efficiency of 80% on 32 nodes and 64% when 64 nodes are used. Without the `DCR` optimization, the system has an efficiency of 47% on 64 nodes. This means that the `DCR` optimization improves the system efficiency by 17% on 64 nodes and we expect this value to increase proportionally with the number of nodes.

To show the effectiveness of the device-host-device copy removal optimization (`DHDCR`) we conducted another experiment on the VSC2 Cluster. Using *libWater* library, we manually coded a multi-device version of the matrix chain multiplication $ABCD$. We ran the experiment using two different settings: the first (`baseline`), uses the runtime system without the optimization while the second (`DHDCROpt`), uses the device-host-device copy removal mechanism as described in Sec-

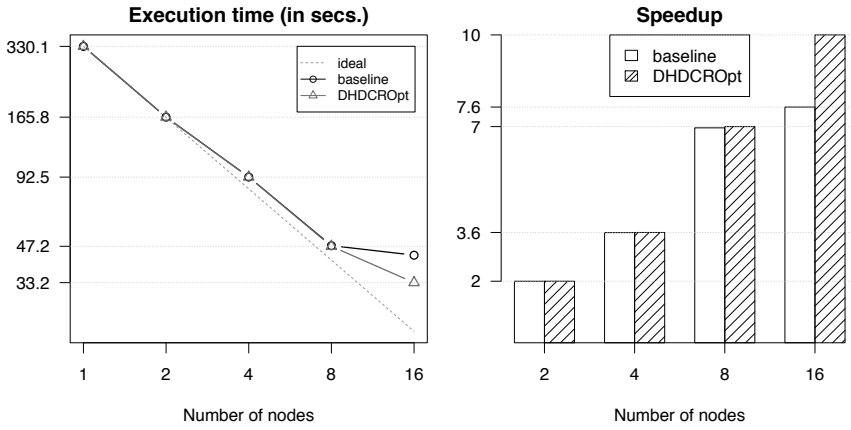


Figure 5.7: Strong scaling of matrix chain multiplication on the VSC2 Cluster

tion 5.4.1. Notice that in both cases the `DCR` optimization is also performed. When both runtime optimizations are enabled, the optimizer first tries to rewrite indirect data transfers to direct ones (using the `TransJob` command). Then, in a second pass, `DCR` is applied. In order to optimize the execution even further, the `DCR` analysis has been updated to also take into account `TransJob` commands during the collective pattern analysis phase.

The results of our experiments are depicted in Figure 5.7. For this application, we show the execution time (in seconds) for up to 16 nodes and the corresponding speedup with respect to a single node. The baseline approach scales almost linearly up to 8 nodes with an efficiency of 87%. For 16 nodes the runtime system efficiency decreases significantly reaching 48%. The main reason is the high communication overhead caused by the unnecessary copies of intermediate buffers to the root node. Before proceeding with the $(AB)(CD)$ operation, the results of AB and CD have to be gathered by the root scheduler and then distributed again on the remaining nodes. While the buffer containing AB can be directly reused, the result of CD can be copied to remaining nodes using a more efficient collective pattern.

The benefit of this optimization starts to show with a large number of nodes because of the increased pressure on the root scheduler. For smaller node counts, the data movement of AB is completely overlapped with computation, so that by the time AB is distributed to the nodes also CD is available and the final computation can start without any delay. For larger nodes, the execution of the last kernel

is delayed since there is not enough computation (kernel execution becomes shorted since more devices are used) to overlap the communication overhead. This causes a sensible decrease in the efficiency. By avoiding this communication, the `DHDCR` optimization improves the speedup from 7.6 to 10 achieving an efficiency improvement of 15%.

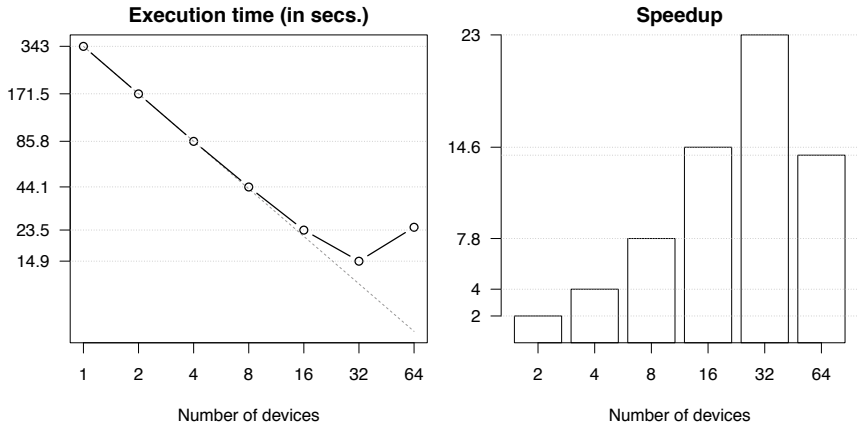
5.5.2 *MinoTauro GPU Cluster*

Another scalability study was conducted executing the N-body simulation described in Table 5.2, line 2, in a GPU cluster. We were able to access up to 32 nodes of the MinoTauro cluster with a total of 64 GPU devices. In all the experiments, the workload was equally partitioned between the available devices. The optimization of the N-body simulation on the GPU processor is an active research problem [123, 16, 43, 56]. The problem is well known to be suitable for the GPU architecture and in case of a high number of particles for cluster of GPUs.

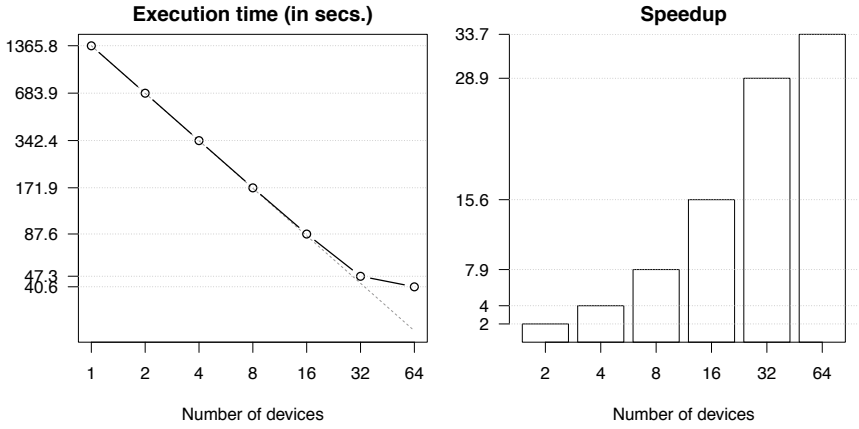
We ran the `NBody` test case using 3 different input sizes that show the benefit of using a high number of GPUs in case of large number of bodies. The results of our experiments are depicted in Figure 5.8. The 3 tests were conducted respectively with an input size of 2, 5 and 10 Million bodies. With the smallest input size, the application scales almost linearly up to 16 GPUs and stops scaling after 32 GPUs. Increasing the input size by a factor of 2 increases the execution time by a factor of 4, due to the quadratic complexity of the implemented algorithm. With an input size of 5 and 10 million bodies the application becomes more suitable for a GPU cluster and with the biggest tested input size achieves a speedup of around 49 on 64 GPUs with an efficiency of 77%. It is worth mentioning that in such environment is important from a user perspective to find a trade-off between the number devices and the desired efficiency.

5.5.3 *Ortler Mixed-node Cluster*

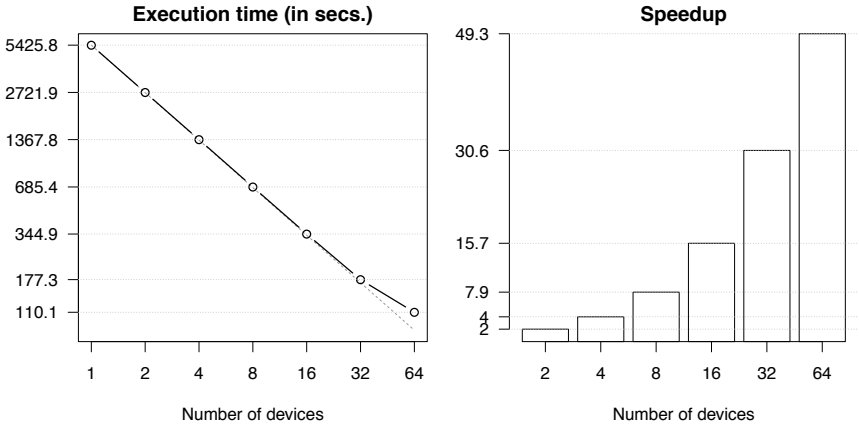
Since OpenCL allows access to heterogeneous devices we conducted a second experiment which demonstrates *libWater* on a mixed-node



(a) `NBody 2400K`



(b) `Nbody 4800K`



(c) `NBody 9600K`

Figure 5.8: Strong scaling of NBody on the MinoTauro BSC GPU Cluster

cluster as described in Table 5.3. In order to run applications on such environment, the input code was rewritten so that the workload distribution was controllable via command line arguments. It is worth

	Device	Workload Partition Configurations							
		C1	C2	C3	C4	C5	C6	C7	C8
Nbody	mc5-GPU1	100%	50%	-	-	-	-	23%	22.5%
	mc5-GPU2	-	50%	-	-	-	-	23%	22.5%
	mc6-GPU1	-	-	100%	50%	-	-	27%	26.5%
	mc6-GPU2	-	-	-	50%	-	-	27%	26.5%
	mc7-ACL1	-	-	-	-	100%	50%	-	1%
	mc7-ACL2	-	-	-	-	-	50%	-	1%
	Ex. time (sec.)	42.2	21.2	35.9	18.2	659.6	335.8	9.9	**9.7**
LinReg	mc5-GPU1	100%	50%	-	-	-	-	15%	11%
	mc5-GPU2	-	50%	-	-	-	-	15%	11%
	mc6-GPU1	-	-	100%	50%	-	-	-	14%
	mc6-GPU2	-	-	-	50%	-	-	-	14%
	mc7-ACL1	-	-	-	-	100%	50%	35%	25%
	mc7-ACL2	-	-	-	-	-	50%	35%	25%
	Ex. time (sec.)	15.5	7.8	11.8	6.0	6.9	3.9	3.2	**2.8**

Table 5.4: Performance of Nbody and LinReg on the Ortler cluster

mentioning that workload partitioning for heterogeneous compute nodes is still an active research problem [37, 66, 58, 42]. However, this aspect is completely orthogonal to our library and for the sake of this experiment, we derive workload partitionings in an empirical way. We ran the NBody and the LinReg test cases using different combinations of devices. For each device configuration, several different workload splittings were tested and the fastest one was chosen. The partitionings and their corresponding execution times are shown in Table 5.4. For example, in NBody, configuration C1 assigns all the workload to the first GPU of node mc5. The execution time for this configuration is 42.2 seconds. By equally splitting the workload between the two GPUs on the same node, i.e. C2, we double the performance. Between the devices, the NVIDIA Tesla k20m is the fastest device requiring 35.9 seconds to complete the work. However, *libWater* can be used to improve the execution time even further. The overall execution time can be reduced by 70% by using the workload partition as described by configuration C8 which assigns 22.5% to each GPU in mc5, 26.5% to each GPU in mc6 and the remaining 1% to each accelerator in mc7. For LinReg results are different since the execution times

for the different devices are more balanced. The best performance can be achieved in this case splitting the workload between the nodes by assigning 11% to each GPU in `mc5`, 14% to each GPU in `mc6` and 25% to each accelerator in `mc7`.

5.6 SUMMARY

In this chapter, we introduced *libWater*, a library for simplifying the programming of heterogeneous distributed systems.

The proposed interface demonstrates that raising the abstraction level of the OpenCL programming model is possible without losing control over performance. We showed with an example how a multi-device distributed host program can be written using approximately 25 lines of code. By defining a simple, but powerful, device query language (DQL), *libWater* simplifies the management and discovery of a large number of OpenCL devices. The simple API makes the library a perfect target for automatic code generation tools, thus it can be easily integrated in compilers.

libWater's interface is tightly bound to a lightweight distributed runtime system which is designed from scratch for high scalability and low resource usage. Because of the non-blocking semantics promoted by the library interface, commands can be organized by the runtime system into a DAG to be used for dynamic analysis and optimizations. We studied the performance of the library on three clusters, demonstrating the high efficiency that the system can achieve.

6

CONCLUSION AND FUTURE WORK

Programming parallel architectures is a difficult task since it often requires a combination of hardware and software expertise. High-level models and frameworks can be employed to simplify the programmability and portability of parallel code, however, often they provide limited performance due to the level of abstraction that is too far away from the underlying hardware. Differently, industry standards, such as OpenCL, provide a certain degree of portability between heterogeneous architectures without sacrificing performance, that can be achieved through hardware-specific optimization techniques. In Chapter 3 we showed the importance of such techniques identifying and evaluating OpenCL software optimizations for the ARM Mali GPU Compute Architecture. Our results showed that the Mali-T604 GPU provides distinct improvements in terms of performance and energy-to-solution over ARM Cortex-A15, achieving in average a speedup of 8.7× while consuming only 32% of the energy. Our study confirmed that, with highly optimized code, embedded GPUs can offer performance and energy advantages over embedded CPUs, similar to their high-end counterparts present in clusters.

Although expert programmers can manually optimize OpenCL code for a particular architecture, other challenges arise in case of heterogeneous nodes composed of multiple devices. In such systems, it is really demanding to write OpenCL code to take advantage of all the heterogeneous resources. In order to tackle this problem, in Chapter 4, we introduced a compiler approach that automatically generates, from a single-device OpenCL program, a multi-device OpenCL program. Through the use of a novel runtime system, the generated program is not only capable of running on multiple devices but also to predict an effective distribution strategy using a machine learning based prediction model. On average, our novel approach reached up to 87.5% of the optimal performance across 23 programs outperforming the default strategies of using only the CPU or only the GPU, which achieved 65.5% and 62.5%, respectively. These results

confirmed that the combination of state-of-the-art compilers and runtimes can automatize complex tasks with substantial impact on performance and productivity. In the same chapter, we also proposed two low-complexity heuristics addressing the problem of scheduling independent tasks in heterogeneous compute nodes. *SimpleHS* is able to balance the workload between different devices predicting the approximate execution time of a new task with a quadratic regression model. Our results validated the success of the proposed heuristics which, using only information available at scheduling time, show performance comparable to more sophisticated methods which require an accurate estimation of the task execution times.

Finally, in Chapter 5 we introduced an extension of OpenCL to simplify the development of heterogeneous distributed applications. *libWater* combines the MPI and OpenCL programming models to realize a simple, but yet powerful framework which hides the distributed nature of the underlying system to the developer. It consists of a lightweight interface and a powerful distributed runtime system which automatically recognizes inefficient communication patterns and transparently optimizes them at runtime. We assess *libWater*'s performance for multiple large clusters demonstrating improved performance and scaling with different test applications and configurations.

6.1 CONTRIBUTIONS

The following peer-reviewed papers and journals contributed to the respective chapters of the thesis.

Chapter 3

IPDPS14 : *Ivan Grasso, Petar Radojković, Nikola Rajović, Isaac Gelado, Alex Ramirez*, "Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU", *28th IEEE International Parallel and Distributed Processing Symposium*, May 19-23, Phoenix, Arizona, USA

Chapter 4

PPOPP13 : *Ivan Grasso, Klaus Kofler, Biagio Cosenza, Thomas Fahringer*, "Automatic Problem Size Sensitive Task Partitioning on Heterogeneous Parallel Systems", *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (poster paper), February 23-27, Shenzhen, China

ICS13 : *Klaus Kofler, Ivan Grasso, Biagio Cosenza, Thomas Fahringer*, "An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning", *27th ACM SIGARCH International Conference on Supercomputing*, June 10-14, Eugene, Oregon, USA

ICCS15 : *Ivan Grasso, Marcel Ritter, Biagio Cosenza, Werner Benger, Günter Hofstetter, Thomas Fahringer*, "Point Distribution Tensor Computation on Heterogeneous Systems", *International Conference on Computational Science 2015*, June 1-3, Reykjavík, Iceland

Chapter 5

ICS13 : *Ivan Grasso, Simone Pellegrini, Biagio Cosenza, Thomas Fahringer*, "libWater: Heterogeneous Distributed Computing Made Easy", *27th ACM SIGARCH International Conference on Supercomputing*, June 10-14, Eugene, Oregon, USA

JPDC14 : *Ivan Grasso, Simone Pellegrini, Biagio Cosenza, Thomas Fahringer*, "A uniform approach for programming distributed heterogeneous computing systems", *Journal of Parallel and Distributed Computing - Volume 74, Issue 12*

6.2 FUTURE WORK

HPC systems have become increasingly complex and difficult to program with the emergence of highly parallel heterogeneous architectures. The Khronos Group is continuously working on the evolution of the *OpenCL* standard trying to solve some of the programming difficulties faced by developers. Recently, the OpenCL C++ kernel language and the SPIR-V binary intermediate representation were introduced. The first allows the programming of OpenCL kernels with

a static subset of the C++14 standard, while the latter allows to easily map novel high-level languages to heterogeneous hardware.

Although these additions will simplify the programming of heterogeneous devices, they will not solve the resource utilization challenge present in heterogeneous compute nodes and clusters. To overcome this problem, the research community is investigating runtime systems able to exploit semantic information provided by state-of-art compilers. Such systems will utilize machine-learning and auto-tuning techniques to automatically explore the space of possible implementations and code optimizations improving the performance of parallel applications.

In conclusion, this thesis investigated multiple different aspects of heterogeneous computing such as performance tuning techniques, novel programming interfaces, and advanced compiler/runtime systems. We believe and hope that the combination of the previously described approaches in conjunction with the advancements in the field of artificial intelligence will lead us into a new era in which programming of heterogeneous clusters will be extremely simplified, allowing domain scientists to finally focus on their main target: science.

LIST OF FIGURES

LIST OF TABLES

LIST OF DEFINITIONS

LIST OF ACRONYMS

ACL	Accelerator
AOS	Array Of Structures
API	Application Programming Interface
CPU	Central Processing Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GPU	Graphic Processing Unit
HPC	High Performance Computing
IR	Intermediate Representation
MPI	Message-Passing Interface
OpenCL	Open Computing Language
SMT	Simultaneous Multi-Threading
SOA	Structure of Arrays
SoC	System on Chip

LIST OF SYMBOLS

E	set of entities
R	set of binary relations
$\langle ... \rangle$	sequence of entities
V	set of values
M	set of statements
P	set of processes
A_p	address space
T_n	execution time
S_n	speedup
E_n	efficiency

BIBLIOGRAPHY

[1] Brandon G. Aaby, Kalyan S. Perumalla, and Sudip K. Seal. Efficient simulation of agent-based models on multi-gpu and multi-core clusters. In *SIMUTools*, pages 29:1–29:10, 2010.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] Alves Albano, Rufino Jose, Pina Antonio, and Santos Luis Paulo. clOpenCL - Supporting Distributed Heterogeneous Computing in HPC Clusters. In *10th International Workshop HeteroPar*, 2012.

[4] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting distributed systems in fastflow. In *Euro-Par Workshops*, pages 47–56, 2012.

[5] Ryo Aoki, Shuichi Oikawa, Takashi Nakamura, and Satoshi Miki. Hybrid OpenCL: Enhancing OpenCL for Distributed Processing. In *ISPA*, pages 149–154, 2011.

[6] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Boosting mobile GPU performance with a decoupled access/execute fragment processor. In *ISCA*, pages 84–93, 2012.

[7] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Parallel Frame Rendering: Trading Responsiveness for Energy on a Mobile GPU. In *PACT*, 2013.

[8] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. TEAPOT: a toolset for evaluating performance, power and image quality on mobile graphics systems. In *ICS*, pages 37–46, 2013.

[9] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. Starpu-mpi: Task pro-

gramming over clusters of machines enhanced with accelerators. In *EuroMPI*, pages 298–299, 2012.

[10] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par*, pages 863–874, 2009.

[11] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices. In *Workshop PPAC*, pages 224–231, 2010.

[12] A. Barak and A. Shilo. The Virtual OpenCL (VCL) Cluster Platform. In *Proc. Intel European Research & Innovation Conference*, page 196, 2011.

[13] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324, 1986.

[14] K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*, pages 307–314, 1968.

[15] Jeroen Bédorf, Evghenii Gaburov, Michiko S. Fujii, Keigo Nitadori, Tomoaki Ishiyama, and Simon Portegies Zwart. 24.77 pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 gpus. In *SC*, pages 54–65, 2014.

[16] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational n-body code that runs entirely on the gpu processor. *J. Comput. Physics*, 231(7):2825–2839, 2012.

[17] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[18] W. Benger, G. Ritter, and R. Heinzl. The concepts of vish. In 4^{th} *High-End Visualization Workshop*, pages 26–39, 2007.

[19] Werner Benger. *Visualization of General Relativistic Tensor Fields via a Fiber Bundle Data Model*. PhD thesis, FU Berlin, 2004.

[20] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. A. Hensgen, and R. F. Freund. A comparison of eleven static heuristics

for mapping a class of independent tasks onto heterogeneous distributed computing systems. *JPDC*, 61(6):810–837, 2001.

[21] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, and Derrick Weathersby. Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems. In *SPAA*, pages 298–309, 1994.

[22] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Productive Programming of GPU Clusters with OmpSs. In *IPDPS*, pages 557–568, 2012.

[23] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.

[24] Dehao Chen, Wenguang Chen, and Weimin Zheng. Cuda-zero: a framework for porting shared memory gpu applications to multi-gpus. *SCIENCE CHINA Information Sciences*, 55(3):663–676, 2012.

[25] Kwang-Ting (Tim) Cheng and Yi-Chu Wang. Using Mobile GPU for General-Purpose Computing - A Case Study of Face Recognition on Smartphones. In *VLSI-DAT*, 2011.

[26] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[27] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU*, pages 63–74, 2010.

[28] Tahir Diopy, Steven Gurfinkely, Jason Anderson, and Natalie Enright Jerger. DistCL: A Framework for the Distributed Execution of OpenCL Kernels. In *MASCOTS*, 2013.

[29] W. Dobler, R. Baran, F. Steinbacher, M. Ritter, M. Niederwieser, W. Benger, and M. Aufleger. Die Zukunft der Gewässervermessung: Die Verknüpfung moderner und klassischer Ansätze: Air-

borne Hydromapping und Fächerecholotvermessung entlang der Rheins bei Rheinfelden. *WasserWirtschaft*, 9:18–25, 2013.

[30] R. Duan, R. Prodan, and T. Fahringer. Performance and cost optimization for multiple large-scale grid workflow applications. In *SC*, 2007.

[31] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *HPCS*, pages 224–231, 2010.

[32] U. Erra, B. Frola, V. Scarano, and I. Couzin. An efficient gpu implementation for large scale individual-based simulation of collective behavior. In *HIBI*, pages 51–58, 2009.

[33] BariÅ Eskikaya and D Turgay Altilar. Distributed OpenCL Distributing OpenCL Platform on Network Scale. In *IJCA*, volume ACCTHPCA 2, pages 26–30, 2012.

[34] Ethem, Alpaydin. *Introduction to Machine Learning*. The MIT Press, Cambridge, MA, USA, 2004.

[35] US Department of Energy Exascale Computing Project (ECP). www.exascaleproject.org.

[36] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2), 2005.

[37] Ivan Grasso, Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic problem size sensitive task partitioning on heterogeneous parallel systems. In *PPoPP*, 2013.

[38] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, , and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *InPar*, 2012.

[39] S. Green. Particle simulation using cuda. *NVIDIA Whitepaper*, 2010.

[40] Peter Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7 (Improving Energy Efficiency in High-Performance Mobile Platforms). White paper, ARM, 2011.

[41] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS*, pages 134–144, 2011.

[42] Dominik Grewe and Michael F.P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *CC*, 2011.

[43] Tsuyoshi Hamada and Keigo Nitadori. 190 tflops astrophysical n-body simulation on a cluster of gpus. In *SC*, 2010.

[44] Alexander Heinecke, Alexander Breuer, Sebastian Rettenberger, Michael Bader, Alice-Agnes Gabriel, Christian Pelties, Arndt Bode, William Barth, Xiang-Ke Liao, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, and Pradeep Dubey. Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers. In *SC*, pages 3–14, 2014.

[45] Thomas Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In *CC*, pages 225–245, 2011.

[46] Torsten Hoefler and Timo Schneider. Runtime Detection and Optimization of Collective Communication Patterns. In *PACT*, pages 263–272, 2012.

[47] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: writing parallel program portable between cpu and gpu. In *PACT*, pages 217–226, 2010.

[48] HPC Meets AI. www.hpcwire.com/2016/11/10/hpc-meets-ai-creates-new-grand-challenges, 2016.

[49] A. V. Husselmann and K. A. Hawick. Spatial data structures, sorting and gpu parallelism for situated-agent simulation and visualization. In *MSV*, pages 14–20, 2012.

[50] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, 1977.

[51] IBM Blue Gene Team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2), 2008.

[52] IBM Blue Gene Team. Blue Gene/Q: by co-design. *Journal of Computer Science - Research and Development*, 28(2-3), 2013.

[53] Institut für Neuroinformatik, Ruhr-University Bochum. Shark Machine Learning Library.
github.com/Shark-ML/Shark.

[54] Intel Corporation. Computing Language Utility.
software.intel.com.

[55] Intel Xeon Phi Product Family x200.
ark.intel.com/it/products/family/92650/Intel-Xeon-Phi-Product-Family-x200.

[56] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *SC*, 2010.

[57] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. Inspire: The insieme parallel intermediate representation. In *PACT*, 2013.

[58] Ma Kai, Li Xue, Chen Wei, Zhang Chi, and Wang Xiaorui. GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures. In *ICPP*, 2012.

[59] Philipp Kegel, Michel Steuwer, and Sergei Gorlatch. dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems. In *IPDPS Workshops*, pages 174–186, 2012.

[60] Khronos OpenCL Working Group. The OpenCL Specification.
www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf.

[61] Khronos OpenGL Working Group. The OpenGL ES specification.
www.khronos.org/opengles.

[62] Khronos SYCL Working Group. The SYCL Specification. www.khronos.org/sycl.

[63] Junghyun Kim, Gangwon Jo, Jaehoon Jung, Jungwon Kim, and Jaejin Lee. A distributed opencl framework using redundant computation and data replication. In *PLDI*, pages 553–569, 2016.

[64] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *ICS*, pages 341–352, 2012.

[65] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *ICS*, pages 341–352, 2012.

[66] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning. In *ICS*, 2013.

[67] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Philip Heidelberger, Dong Chen, Mark E. Giampapa, Michael Blocksome, Ahmad Faraj, Jeff Parker, Joseph Ratterman, Brian Smith, and Charles J. Archer. The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer. In *ICS*, pages 94–103, 2008.

[68] Orion S. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *CLUSTER*, pages 1–8, 2009.

[69] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, pages 451–460, 2010.

[70] Jyrki Leskelä, Jarmo Nikula, and Mika Salmela. OpenCL embedded profile prototype in mobile device. In *SiPS*, pages 279–284, 2009.

[71] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Y. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS*, pages 287–296, 2008.

[72] C. Liu and S. Baskiyar. A general distributed scalable grid scheduler for independent tasks. *J. Parallel Distrib. Comput.*, 69(3):307–314, 2009.

[73] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *ICS*, pages 273–282, 2013.

[74] Miguel Bordallo Lopez, Henri Nykanen, Jari Hannuksela, Olli Silven, and Markku Vehvilainen. Accelerating image recognition on mobile devices using GPGPU . In *Proceedings of SPIE*, volume 7872, 2011.

[75] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, pages 45–55, 2009.

[76] P. Luo, K. Lü, and Z. Shi. A revisit of fast greedy heuristics for mapping a class of independent tasks onto heterogeneous computing systems. *J. Parallel Distrib. Comput.*, 67(6):695–714, 2007.

[77] Arian Maghazeh, Unmesh D. Bordoloi, Petru Eles, and Zebo Peng. General Purpose Computing on Low-Power Embedded GPUs: Has It Come of Age? In *SAMOS*, 2013.

[78] M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *J. Parallel Distrib. Comput.*, 59(2):107–131, 1999.

[79] Mali-T600 Series GPU OpenCL Developer Guide. malideveloper.arm.com.

[80] Mali-T604 GPU Architecture. www.arm.com/products/graphics-and-multimedia/mali-gpu.

[81] Mont-Blanc 2 (FP7-ICT-2013-10 European project): European scalable and power efficient HPC platform based on low-power embedded technology, 2013.

[82] Mont-Blanc (FP7-ICT-2011-7 European project): European scalable and power efficient HPC platform based on low-power embedded technology, 2011.

[83] MPI Forum. MPI: A Message-Passing Interface Standard. www.mpi-forum.org, 2012.

[84] MPICH. www.mpich.org.

[85] MVAPICH. mvapich.cse.ohio-state.edu.

[86] NVIDIA Corporation. CUDA Programming Model. developer.nvidia.com/cuda-toolkit.

[87] OCL-MLA. tuxfan.github.com/ocl-mla.

[88] OpenACC Application Program Interface. openacc.org.

[89] OpenMP Architecture Review Board. OpenMP application program interface. www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[90] OpenMPI. www.open-mpi.org.

[91] Ridvan Özaydin and D. Turgay Altilar. OpenCL Remote: Extending OpenCL Platform Model to Network Scale. In *HPCC-ICESS*, pages 830–835, 2012.

[92] Alexandros Panagiotidis, Daniel Kauker, Steffen Frey, and Thomas Ertl. DIANA: a device abstraction framework for parallel computations. In *PARENG*, 2011.

[93] Alexandros Panagiotidis, Daniel Kauker, Filip Sadlo, and Thomas Ertl. Distributed computation and large-scale visualization in heterogeneous compute environments. In *Proceedings of the 11th International Symposium on Parallel and Distributed Computing*, 2012.

[94] Alexandros Panagiotidis, Daniel Kauker, Filip Sadlo, and Thomas Ertl. Distributed computation and large-scale visualization in heterogeneous compute environments. In *ISPDC*, pages 87–94, 2012.

[95] Jelena Pjesivac-Grbovic, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[96] PowerVR Graphics. www.imgtec.com/powervr/powervr-graphics.asp.

[97] Qualcomm Snapdragon Processor. www.qualcomm.com/chipsets/snapdragon.

[98] R-manual:Student's t-Test. stat.ethz.ch/R-manual/R-devel/library/stats/html/t.test.html.

[99] Nikola Rajovic, Paul Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? . In *SC*, 2013.

[100] Nikola Rajovic, Alejandro Rico, Filippo Mantovani, Daniel Ruiz, Josep Oriol Vilarrubi, Constantino Gomez, Luna Backes, Diego Nieto, Harald Servat, Xavier Martorell, Jesus Labarta, Eduard Ayguade, Chris Adeniyi-Jones, Said Derradji, Herve Gloaguen, Piero Lanucara, Nico Sanna, Jean-Francois Mehaut, Kevin Pouget, Brice Videau, Eric Boyer, Momme Allalen, Axel Auweter, David Brayford, Daniele Tafani, Volker Weinberg, Dirk Brömmel, René Halver, Jan H. Meinke, Ramon Beivide, Mariano Benito, Enrique Vallejo, Mateo Valero, and Alex Ramirez. The mont-blanc prototype: An alternative approach for hpc systems. In *SC*, 2016.

[101] Nikola Rajovic, Alejandro Rico, James Vipond, Isaac Gelado, Nikola Puzovic, and Alex Ramirez. Experiences with mobile processors for energy efficient HPC . In *DATE*, pages 464–468, 2013.

[102] Blaine Rister, Guohui Wang, Michael Wu, and Joseph R Cavallaro. A fast and efficient sift detector using the mobile GPU. In *ICASSP*, 2013.

[103] M. Ritter. Introduction to HDF5 and F5. Technical Report CCT-TR-2009-13, Center for Computation and Technology, Louisiana State University, 2009.

[104] M. Ritter and W. Benger. Reconstructing Power Cables From LIDAR Data Using Eigenvector Streamlines of the Point Distribution Tensor Field. In *Journal of WSCG*, pages 223–230, 2012.

[105] Samsung Exynos 5 Dual Arndale Board. www.arndaleboard.org.

[106] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[107] Sushant Sharma, Chung-Hsing Hsu, and Wu-chun Feng. Making a case for a green500 list. In *IPDPS Workshops*, 2006.

[108] Simple-opencl. code.google.com/p/simple-opencl.

[109] Nitin Singhal, In Kyu Park, and Sungdae Cho. Implementation and Optimization of Image Processing Algorithms on Handheld GPU. In *ICIP*, 2010.

[110] Nitin Singhal, Jin Woo Yoo, Ho Yeol Choi, and In Kyu Park. Implementation and Optimization of Image Processing Algorithms on Embedded GPU. In *IEICE TRANSACTIONS on Information and Systems*, volume 95, 2012.

[111] Magnus Strengert, Christoph Muller, Carsten Dachsbacher, and Thomas Ertl. Cudasa: Compute unified device and systems architecture. In *EGPGV*, pages 49–56, 2008.

[112] Magnus Strengert, Christoph Müller, Carsten Dachsbacher, and Thomas Ertl. CUDASA: Compute Unified Device and Systems Architecture. In *EGPGV*, pages 49–56, 2008.

[113] Enqiang Sun, Dana Schaa, Richard Bagley, Norman Rubin, and David Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Workshop GPGPU*, pages 84–93, 2012.

[114] Tegra K1.
www.nvidia.com/object/tegra-k1-processor.html.

[115] TESLA P100.
www.nvidia.com/object/tesla-p100.html.

[116] The HDF Group. HDF5 - Homepage.
www.hdfgroup.org/HDF5.

[117] The MinoTauro GPU Cluster.
www.bsc.es/marenostrum-support-services/other-hpc-facilities/nvidia-gpu-cluster, 2013.

[118] The Post-K Computer.
www.fujitsu.com/global/Images/fujitsu-next-endeavor-the-post-k-computer.pdf.

[119] The Vienna Scientific Cluster 2.
www.vsc.ac.at, 2013.

[120] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. Automatic opencl device characterization: guiding optimized kernel design. In *Euro-Par*, pages 438–452, 2011.

[121] G. Vigueras, J. M. Orduña, M. Lozano, J. M. Cecilia, and J. M. García. Accelerating collision detection for large-scale crowd simulation on multi-core and many-core architectures. *Int. J. High Perform. Comput. Appl.*, 2014.

[122] Guohui Wang, Yingen Xiong, Jay Yun, and Joseph R. Cavallaro. Accelerating Computer Vision Algorithms Using OpenCL on the Mobile GPU - A Case Study. In *ICASSP*, 2013.

[123] Wei Wang, Hanli Wang, Dong Guo, Haoyang Wei, and Guosun Zeng. Parallel time-space processing model based fast n-body simulation on gpus. In *PMAM*, 2013.

[124] M. S. Warren and J. K. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *ACM/IEEE Conference on Supercomputing*, pages 570–576, 1992.

[125] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *ACM/IEEE Conference on Supercomputing*, pages 12–21, 1993.

[126] Shucai Xiao and Wu chun Feng. Generalizing the Utility of GPUs in Large-Scale Heterogeneous Computing Systems. In *IPDPS Workshops*, pages 2554–2557, 2012.