Politecnico di Milano
Department of Electronics, Information, and Bioengineering
Doctoral Programme In Information Technology

Towards Improving Programmability of Heterogeneous Parallel Architectures

Doctoral Dissertation of:
Michele Scandale

Advisor:
Prof. Giovanni Agosta

Tutor:
Prof. Andrea Bonarini

The chair of the Doctoral Program:
Prof. Carlo Fiorini

2015 – XXVIII edition

To my parents

Acknowledgements

The description of my study path along the last four years, at the beginning as research assistant and then as a Ph.D. student, can not be summarized in few lines of acknowledgements. Everything started with my dream at the end of my master of science to acquire strong skills and experience in the wonderful field of compilation techniques. The experience of the Ph.D. is the first step forward to transform my dream into reality.

First of all, I would like to thank my advisor prof. Giovanni Agosta, for giving me the unique possibility to work on topics related to parallel programming and compiler technologies. Thanks to prof. Gianfranco Lamperti, my master thesis advisor, and prof. Stefano Crespi Reghizzi I was able to met prof. Giovanni Agosta and his team.

From the compiler group I would like to thank Ettore Speziale and Michele Tartara for the initial bootstrap in the compiler development and parallel programming worlds, my colleagues Michele Beretta and Alessandro Di Federico, that worked with me in the development and support of the OpenRISC toolchain, and finally master students Giulio Sichel and Marco Bonacina that contributed in the development of OpenCRun.

Indeed, I would like to thank prof. Gerardo Pelosi and Alessandro Barenghi for granting me the possibility to explore the mysterious world of applied cryptography within my minor research topic, and indeed for the useful hints and support provided during my Ph.D. career.

Furthermore, I would like to Marco Cornero for the great experience during my internship in ARM Ltd., granting me the possibility to deeply work on GPU compiler technologies and architectures, and exploring the new features of the next generation of programming models for tightly coupled CPU-GPU platforms.

A special thanks goes again to the people from DEIB office no. 127, Giovanni Agosta, Gerardo Pelosi, Alessandro Barenghi, Ettore Speziale, Michele Tartara, Michele Beretta, Alessandro Di Federico, with whom I shared during the last four years at Politecnico di Milano a great experience with lot of fun and great results.

Finally, I would like to thank my parents and my friends for support to “survive” during this convoluted experience.

Sommario

La computazione parallela è da lungo tempo considerata una tecnica efficace per combinare prestazioni ed efficienza energetica. A partire dal High Performance Computing (HPC) fino ai moderni sistemi embedded, l'adozione di architetture parallele eterogenee diventa una pratica sempre più comune, dato che queste consetono di raggiungere un buon compromesso in termini di efficienza energetica. Il raggiungimento di prestazioni *exascale* per la prossima generazione di sistemi HPC è vincolata da un consumo complessivo tra i 20 MW e 30 MW. Gli attuali sistemi HPC "green" non sono tuttavia in grado di raggiungere questo grado di efficienza nonostante l'utilizzo di moderne architetture parallele eterogenee. Infine, le piattaforme hardware ultra-low-power guadagnano sempre più visibilità dato che possono essere componenti chiave per consentire ai prossimi sistemi HPC di raggiungere il livello di efficienza necessaria per raggiungere l'obiettivo exascale.

La programmabilità di quesi sistemi è un aspetto critico che ha un forte impatto sull'efficienza raggiungibile e ancor di più nel costo per ottenere tale obiettivo. Programmare architetture parallele è un'operazione complessa, dato che generalemente molte caratteristiche hardware sono esposte completamente e direttamente ai programmatori. Per questo motivo esistono infrastrutture di programmazione che cercano di nascondere questa complessità, tuttavia le prestazioni ottenibili sono sub-ottime rispetto ad implementazioni dedicate, o sono infrastrutture limitate a specifici domini applicativi.

In questa tesi si affrontano le sfide legate alla programmabilità di architetture parallele eterogenee, operando su modelli di programmazione e architetture sia esistenti che futuri. In particolare, si presenta OpenCRun, un runtime OpenCL che supporta varie piattaforme con caratteristiche molto differenti tra loro, come multi-core X86 e acceleratori paralleli embedded. Nell'ambito delle architetture ultra-low-power, si presentano i risultati della collaborazione tra sviluppatori hardware e software per la piattaforma PULP, mostrando i benefici di oculate estensioni della ISA e il corrispettivo supporto nel compilatore per massimizzare l'efficienza energetica della piattaforma. Inoltre, per migliorare la portabilità funzionale e prestazionale di codice OpenCL tra GPGPU e acceleratori many-core embedded con memorie esplicitamente gestite

come PULP e STHorm, si presenta una trasformazione, *work-item coalescing*, che supera le limitazioni mostrate dalle piattaforme embedded, e una ottimizzazione dei trasferimenti di memoria per incrementare le prestazioni del codice finale. Al fine poi di innalzare il livello di astrazione in modo più radicale, assumendo piattaforme dotate di memoria virtual condivisa in quanto caratteristica hardware attesa a breve nelle prossime generazioni di piattaforme eterogenee, si presenta un metodo per implementare puntatori a funzione condivisi in piattaforme eterogenee con due o più ISA, un mattone fondamentale per ottenere il supporto al linguaggio C++ tra ISA eterogenee. In aggiunta si presenta un meccanismo per supportare chiamate a funzione il cui codice non è presente per il dispositivo invocante. Tale meccanismo è necessario per ottenere un supporto trasparente del linguaggio C++ e fornire una maggiore flessibilità ai programmatori che lavorano con applicazioni complesse per portarle all'utilizzo di acceleratori paralleli eterogenei.

Abstract

Parallel computing has been considered an effective approach to combine performance and power efficiency for a long time. Starting from High Performance Computing (HPC) to modern embedded systems, the employment of heterogeneous parallel architectures is becoming the common case, since they provide a good tradeoff in terms of power efficiency. The exascale objective for the next generation of HPC systems is constrained to a target power envelope ranging from 20 MW to 30 MW. The existing "green" HPC systems are not yet able to reach the such power efficiency although they already employ modern heterogeneous parallel architectures. Ultra-low-power hardware platforms are gaining an increasing traction, as they may represent the key component to allow future HPC systems to match the required power efficiency.

The programmability of such systems is a critical aspect that has an huge impact on the reachable power efficiency and the effort required to reach such target. Programming parallel architectures is a complex task, since many hardware features are directly exposed to the programmers. Programming frameworks that try to hide such complexity exist, however they either provide only sub-optimal performance with respect to hand tuned implementations, or they are limited to specific application domains.

This dissertation tackles challenges related to the programmability of heterogenous parallel architectures, acting on both existing and future programming models and hardware architectures. In particular, we present OpenCRun, an OpenCL runtime implementation supporting a range of platforms with very different architectures characteristics, such as X86 multicores and embedded parallel accelerators. In the context of ultra-low-power architectures we report the joint effort between hardware and software developers towards the PULP platform, showing the benefits of selected ISA extensions and their compiler support to maximize the power efficiency. Moreover, to improve functional and performance portability of OpenCL code between GPGPUs and embedded many-core accelerators with explicitly managed memory such as PULP and STHorm, we have proposed a code transformation technique, *work-item coalescing*, that bypasses the limitations of the embedded platforms, allowing code developed for GPGPU to be ported seamlessly, as

well as a memory transfer optimization technique to tune the resulting code to improve performance. Finally, to increase the abstraction level in a more radical way, leveraging Shared Virtual Memory that is expected to be available in future architectures, we have presented a method to transparently implement shared function pointers in heterogeneous platforms with two or more ISAs, a building block for enabling full C++ support across heterogeneous ISAs. Indeed we presented a fallback solution to implement function calls from device side to functions not available on the device itself. This mechanism is needed to enable the transparent support of C++, and to provide more flexibility to the programmers dealing with large and complex applications to be ported towards heterogeneous parallel accelerators.

Contents

Introduction 1

1 Parallel Computing 5
1.1 Introduction 5
1.2 Hardware: evolution of computer architectures 6
1.2.1 Flynn's Taxonomy 7
1.2.2 Single-core architectures 7
1.2.3 Multi-core architectures 9
1.2.4 Many-core architectures 12
1.2.5 Future perspective 19
1.3 Parallel programming models 20
1.3.1 Overview 22
1.3.2 Comparison 28
1.4 Conclusion 31

2 Software/Hardware Architecture 33
2.1 Introduction 33
2.2 OpenCRun 34
2.2.1 Host-runtime architecture 35
2.2.2 CPU device 36
2.2.3 STHorm device 37
2.2.4 OpenCL builtin library 38
2.3 PULP Platform 39
2.3.1 Overview of the Platform 41
2.3.2 Instruction-Set Extensions for OpenRISC 43
2.3.3 Implementation in the ULP-Cluster 47
2.3.4 Performance, Area, And Power Results 49
2.4 Conclusion 53

3 Cross-Platform Functionality and Performance for OpenCL 55
3.1 Introduction 55
3.2 Preliminaries on OpenCL and Background 57
3.3 Kernel Transformations 60
3.3.1 Work-item Coalescing 61
3.3.2 Memory Transfers Optimization 64

3.4 Evaluation 66
3.5 Related Work 70
3.6 Conclusion 71

4 C++ Support across Heterogeneous Systems **73**
4.1 Introduction 73
4.2 Transparent Function Pointers 76
4.3 Integrating C++ Support across Heterogeneous Systems . 82
4.3.1 Virtual member functions 83
4.3.2 Generalized system calls 86
4.3.3 Other C++ features 88
4.3.4 Heterogeneous Linker 88
4.4 Evaluation 91
4.5 Related Work 94
4.6 Conclusion 95

Conclusion **95**

List of Figures **99**

List of Tables **102**

Bibliography **105**

Introduction

Parallel computing has been considered an effective approach to combine performance and power efficiency for a long time. Originally, highly parallel architectures were designed primarily for application specific purposes in the field of High Performance Computing (HPC) – typically solving large simultaneous equation sets by means of finite elements methods, tackling problems coming from metereology and fluidodynamics among the others. More recently, the application of parallel computing to computer graphics and the shift from fully-dedicated graphics hardware to programmable graphic accelerators has brought this kind of massively parallel hardware into the domain of general purpose computing, and from there back into HPC.

At the same time, in embedded computing smaller-scale parallelism has been exploited primarily to achieve energy-efficiency – e.g. through Very Long Instruction Word architectures.

Nowadays, there is a widespread trend towards convergence between systems targeting very different domains.

HPC systems are increasingly built using heterogeneous components such as General Purpose Graphics Processor Units (GPGPUs), with the goal of achieving significant gains in terms of the FLOPS/W metric – the target power envelope for future exascale system ranges between 20 MW and 30 MW. Such "Green" HPC systems are espected to make HPC facilities available to a wide range of smaller-scale clients, expanding the industrial application of HPC from a few capital intensive domains (e.g., oil & gas, financial services) to SMEs in fields such as drug design, mechanical engineering, and so on.

High-end embedded systems feature programmable graphics accelerators. These architectures are clearly moving towards laptop- or even desktop-grade capabilities, with applications e.g. in gaming.

Finally, ultra-low-power hardware platforms are being designed around the concept of a fabric of numerous, very small processing elements loosely coupled to a host processor. Such architectures target a range of application domains that are gaining increasing traction, such as mobile image processing for autonomous vehicles.

Heterogeneity seems to be the common denominator of these architectures – combining larger cores for processing control-intensive tasks with

smaller but numerous cores for processing data-parallel tasks allows, through appropriate resources management, to limit energy consumption while guaranteeing Quality of Service. This heterogeneity, however, comes at a cost – programming heterogeneous platforms is all but an easy task, since these platforms and the programming models built for them tend to leave to the programmer the explicit management of data allocation and communication.

Currently, industry standards such as OpenCL [50] are employed to program heterogeneous, massively parallel platforms. OpenCL supports functional and performance portability, albeit by exposing both to the programmer through platform introspection capabilities – the programmer is in charge to read the platform capabilities and encode in his application all the necessary logic to optimize code execution, or even simply make it possible on a range of different platforms. Typically, programmers tends to focus on just a set of platforms that are optimal for their application domain, foregoing performance and even functional portability to other classes of hardware platforms.

While this may appear sufficient for current purposes, the fast evolution of hardware platforms in all of the computing continuum may easily mean that code written today for an HPC application may need to be rewritten entirely, leading to a slow-down of adoption of new architectures and to increased development costs.

Contributions

In this work, we tackle the abovementioned issue through a two pronged approach. On one hand, we aim at providing improved functional and performance portability to OpenCL code through compiler transformations for low-end parallel architectures. We believe that such techniques will become relevant to other application domains, up to HPC, as hardware approaches more commonly found in embedded systems become prevalent in higher-scale systems to cope with the power wall. For maximizing the performance on heterogeneous parallel architectures, OpenCL applications are usually written in a target specific way, affecting the portability the application code itself. A common source of non-portability for embedded heterogeneous parallel architectures is represented by hard constraints on the work-group size. In the general case the work-items in a work-group cooperate together exploiting local memories, thus work-groups reshaping may not be applicable. Chapter 3 shows how to transparently run OpenCL applications presenting non native work-group sizes on embedded heterogeneous accelerators. Indeed,

it shows a compiler transformation that introduce double-buffering in order to exploit DMA units available on such embedded platforms.

On the other hand, we look at future parallel architectures featuring Shared Virtual Memory, and, aiming at providing a radically simpler programming model, we simplify host/device interaction by allowing a fully transparent programming model, abstracting the coexistence of multiple Instruction Set Architectures (ISA) and Application Binary Interfaces (ABI). Chapter 4 presents a very efficient implementation of shared function pointers on ISA-heterogeneous architectures, as well as a transparent programming model employing full C++ support. The function pointer support in the heterogeneous context is a key component for exploiting the shared virtual memory mechanism. From the programmer perspective, the proposed approach ensures the complete sharing of both code and data between host and device. This mechanism allows the full C++ support on heterogeneous systems, providing a simpler but much more flexible programming model that allows the porting of large C++ application towards the use of heterogeneous parallel platform in a incremental fashion with a reduced programming effort, since no strong restriction are applied to the code will be promoted to computational parallel kernel.

The baseline for our work is provided by the Open Source OpenCL runtime we have developed, OpenCRun, which we also describe in this thesis.

Organization of the dissertation

The rest of this dissertation is organized as follow. Chapter 1 introduces the world of parallel computing, from both the hardware and software perspective. Chapter 2 presents the software architecture developed as baseline for this work, and the contribution on the PULP platform definition in the context of ultra low-power parallel accelerators. Chapter 3 deals with the problem of portability of OpenCL application between GPGPUs and embedded parallel accelerators. Chapter 4 discusses about the challenges and solutions for full C++ support on heterogeneous parallel architectures. Concluding remarks close this dissertation.

1 Parallel Computing

To understand the requirements imposed on future programming models for parallel computing and the gap with respect to the state of the art, it is necessary to analyze the historical evolution of the field, as well as the current and future trends. Therefore, we present the evolution of hardware architectures starting from uni-processors to many-core architectures discussing the evolution path of hardware architectures, and the converging point of modern heterogeneous parallel architectures. We then discuss parallel programming models for multi-cores and many-cores architectures, pros and cons in terms of programmability of the underlying architecture versus the ability to exploit the available hardware features.

1.1 Introduction

High Performance Computing (HPC) has been traditionally the domain of grand scientific challenges and a few industrial domains such as oil & gas or finance, where investments are large enough to support massive infrastructures. However, nowadays HPC has been recognized as a powerful tool to increase the competitiveness of nations and their industrial sector, including small scale but high-tech businesses – *to compete, you must compute* has become an ubiquitous slogan [26].

The current road-maps [2, 85] for HPC systems aim at reaching exascale levels (10^{18}FLOPS) within 2023 – a $\times$1000 improvement over petascale, which was reached in 2009, and a $\times$100 improvement over current systems. Reaching exascale poses the additional challenge of significantly limiting the energy envelope while providing massive increases in computational capabilities – the target power envelope for future exascale system ranges between 20 MW and 30 MW. Thus, a distinct class of HPC systems, dubbed Green HPC systems, are being designed aiming at maximizing a FLOPS/Watt metric, rather than the typical FLOPS one. Such systems are increasingly moving towards heterogeneous architectures employing GPGPUs as accelerators - in the November 2014 Green500 list, the top 23 systems have parallel accelerators. The efficiency of such heterogeneous systems is more than double that of homogeneous ones (i.e., 5271 MFLOPS/W vs. 2304 MFLOPS/W

considering the 1st and 24th entries of the current Green 500 ranking www.green500.org). This level of efficiency is still one order of magnitude lower than that needed for supporting exascale systems at the target power envelope of 20 MW. To this end, European efforts have been focused towards building supercomputers out of the less power-hungry ARM cores and GPGPUs [40, 74]. On the semiconductor industry side, the wide margin provided by modern chip manufacturing techniques, combined with the inability to exploit this silicon headroom to produce faster, more complex cores due to the breakdown of Dennard scaling, has given rise to a pervasive diffusion of a number of parallel computing architectures, up to the point where embedded systems are also characterized by multi-core processors. The large design effort has led to a variety of approaches in terms of core interconnection and data management. Thus, the ability to port applications designed for current platforms, based on GPGPUs like the NVIDIA Kepler or Tesla families, to heterogeneous systems such as those currently designed for embedded systems is critical to provide software support for future HPC.

Designing and implementing HPC applications is a difficult art, which requires mastering many specialized languages and tools for performance tuning. This is incompatible with the current drive of opening HPC infrastructures to a much wider range of users – the current model of having the HPC center staff directly support the development of the application will become unsustainable in the long run. Thus, the availability of effective standard programming languages and APIs is critical to provide migration paths towards novel heterogeneous HPC platforms as well as to guarantee the ability of developers to work effectively on these platforms.

In the following, Section 1.2 presents the evolution of hardware architectures starting from uni-processors to many-core ones discussing the evolution path of hardware architectures, and the converging point of modern heterogeneous parallel architectures, while Section 1.3 discusses parallel programming models for multi-cores and many-cores architectures, pros and cons in terms of programmability of the underlying architecture versus the ability to exploit the available hardware features.

1.2 Hardware: evolution of computer architectures

Starting from the beginning of the Computer science, hardware has driven the evolution of languages, programming models and software architectures. The main goal of the evolution of hardware architectures has been the improvement of the running time of applications. De-

pending on the application domain and the target requirements, several strategies have been proposed.

1.2.1 Flynn's Taxonomy

Flynn's taxonomy [35] is a useful classification of the main directions of the computer architecture evolution. In Flynn's taxonomy the hardware is modelled as a set of processing units, executing program instructions fetched from an instruction pool employing an instruction stream. Data is stored in a data pool, and accessed employing a data stream. The classification considers the number of processing units and how they are connected to the instruction and data pools.

Single Instruction Single Data (SISD) There is only a single processing unit, and a single data stream. This was the architecture dominating the general purpose market up to year 2005. Nowadays it represents a building block for large multi-cores, and it is used in embedded low-power devices, such as low end micro-controllers.

Single Instruction Multiple Data (SIMD) There are multiple processing units executing the same instruction, and each unit fetches data from different streams. Vector processors represent the first architecture of this class, while GPGPUs represent the current design choice in this class.

Multiple Instruction Multiple Data (MIMD) There are multiple, independent processing units. Each one executes instructions fetched from different instruction streams, and fetches data from different data streams. Modern multi-core processors fall into this category: each core is independent, so it is possible to execute completely unrelated instruction flows manipulating independent data streams on each core.

Multiple Instruction Single Data (MISD) There are multiple processing units, executing different instructions on the same data. This class was defined by Flynn for the sake of symmetry, however basically no architecture, save for systolic arrays is classified as MISD.

1.2.2 Single-core architectures

Until to 2005 the majority of general-purpose architectures were SISD. The main target during this period were single-threaded applications,

thus the focus in the computer architectures development was the improvement of the execution time of a sequential application. To this end, the first attempt made was the implementation in hardware of the most recurrent complex operations present in target user applications. The result of such a design trend are architectures known as *Complex Instruction Set Computing* (CISC) architectures. Common features among them are the support of complex operations and complex addressing modes. However the hardware complexity required to support such operations impacts adversely on the critical path, thus limiting the maximum working frequency.

As a consequence of this mindset, the design criteria of computer architectures were aimed at improving the throughput of completed instructions, measured in terms of *Instructions Per Cycle* (IPC). *Reduced Instruction Set Computing* (RISC) architectures exploit better the pipelining technique. The processing of a single instruction is split into stages. At each clock cycle, the architecture executes all the stages of the pipeline in parallel, thus the clock cycle is determined by the latency of the slowest stage. The latency of an instruction is equivalent to the latency of the pipeline, however the throughput is increased since an instruction is completed at each cycle.

Pipelining exploits the parallelism between instructions in a single execution flow – *Instruction Level Parallelism* (ILP) – executing them partially in parallel while still exposing a sequential programming model. However, the benefit of pipelining holds if and only if the pipeline is kept full, i.e. at each clock cycle an instruction must be issued. If this is not possible a stall is inserted in the pipeline, leading to a performance loss. This may happen due to hazards induced by the instruction sequence. There are three kind of hazards: control, data, and structural. A *control hazard* is generated whenever the address of the next instruction is not ready as an input to the fetch stage. A *data hazard* is generated when data needed by an instruction are not available yet. A *structural hazard* is triggered when all hardware resources required to execute an instruction are not free.

To reduce the effect of hazards, several techniques were proposed: cache hierarchy, pipeline forwarding, super-scalar pipelines, VLIWs, speculative execution, out-of-order execution [84], branch prediction [91]. In order to increase the clock frequency pipelines have been split in multiple stages (e.g. with the last Intel Pentium 4 being designed with a 31-stages pipeline). However, the intrinsic limitation of the ILP [88] in a single instruction stream acts against advanced and complex techniques when considering the performance gains versus the power and silicon costs. This increased power consumption also prevents further increases

of the clock frequency, as that would also worsen the the dynamic power consumption issue. Indeed advanced techniques to minimize the amount of stalls inserted in the pipeline require complex logic, often contributing more than a clock increase to the overall power consumption. This problem is generally identified with the term *power wall* [18].

Orthogonally, cache hierarchy is used to amortize the cost of accessing data in memory, however feeding the processor with instructions to be executed, and data to be computed at each clock cycle is still an issue. The inability of the memory hierarchy to fulfill the data requests is identified with the term *memory wall* [89].

1.2.3 Multi-core architectures

Hitting the power wall forced architecture designers to move towards other strategies beyond ILP to improve the overall performance and reduce the power consumption.

To enclose the power consumption, designers removed power-hungry components required by aggressive ILP techniques. Pipelines became shorter, the issue width narrowed, and static scheduling with in-order execution was preferred on low power designs. These design criteria do not necessarily imply that we cannot execute more complex applications. At the same time, such simpler designs fit better the emerging figures of merit, such as the power consumption, and power/energy efficiency. However, some applications still require raw performance. For example HPC applications need faster processors to perform more accurate simulations. Computer-graphics is another field where raw performance is a stringent requirement.

To provide raw performance, techniques beyond pure ILP have been employed. Moore's law is still in effect, increasing each year the number of transistors that can be packed in a given amount of silicon area, thus allowing the implementation of new techniques to improve performance.

Starting from 1995, computer architectures contain explicitly parallel features. Vector instruction set extensions [30, 73] are a good example of this trend. This concept is borrowed from vector processors, the idea is to have instructions operating on fixed length vector types.

The trend of exposing hardware parallel features reached the critical point around 2005. Starting from 2002, *Simultaneous Multi Threading* (SMT) designs (e.g. Intel Hyper-Threading [61]) have been employed to allow the execution of more than one independent execution flow. Pipeline stalls due to hazards in one execution flow are used to execute another execution flow.

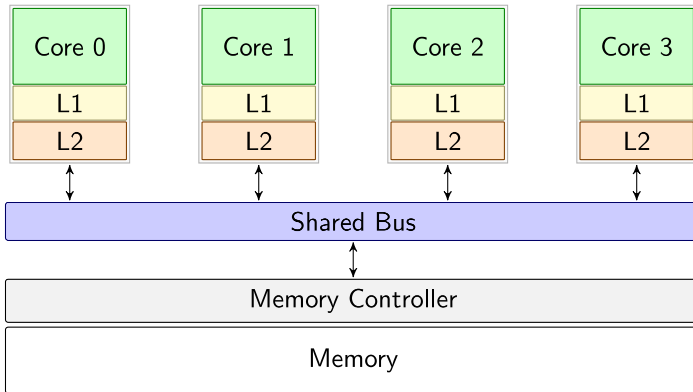


Figure 1.1: Quad-core SMP UMA architecture.

The evolution of this approach is *Symmetric Multi Processing* (SMP) design, were multiple independent processing elements are explicitly exposed. This technique, initially exploited at multiple package level – i.e. installing more than one single-core processor on the same motherboard –, has been widely applied at the single package level – i.e. putting more than one core on the same die – starting from 2005. This new design strategy leads to architectures that can be classified as MIMD in Flynn's taxonomy.

The problem of feeding processors with data is orthogonal to the power problem. However the same techniques can be useful to deal with the memory wall. On SISD architectures there is an unique path for accessing the main memory – all accesses go through the *memory controller*. The primary measure of its efficiency is the *bandwidth*, that is the amount of bytes it can transfer from/to the memory per time unit. To increase this amount, the memory controller has been integrated in the same die on which the CPU resides. It works closely with the cache hierarchy, and they are responsible for ensuring *memory consistency*. Considering that the memory access latency and bandwidth do not evolve like the performance of the core, the memory controller becomes rapidly a bottleneck of the architecture. Moreover, the increasing number of cores per die imposes a further load on the memory controller due to the fact that it has to provide data for all of them.

Architectures where the memory access latency is constant among all processors are called *Uniform Memory Access* (UMA). When an access to the main memory is generated, the cores send a request to the memory controller. The communication between cores and memory controller is a shared bus. Figure 1.1 shows the structure of a quad-core UMA architecture.

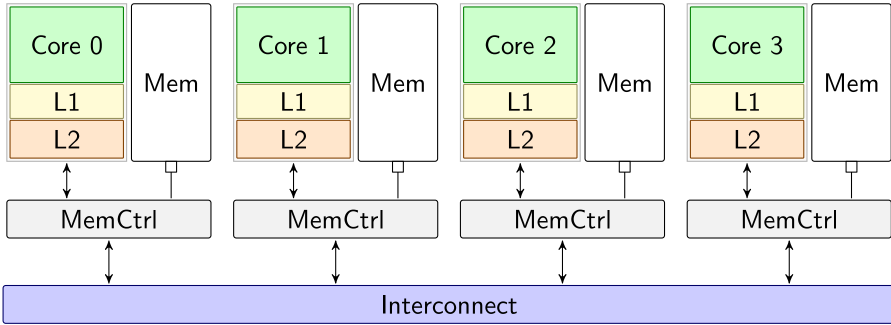


Figure 1.2: Quad-core SMP NUMA architecture.

This design, however, does not scale well increasing the number of cores. To remove the bottleneck, the number of paths to the main memory must be increased. Each core can have its own memory controller, directly connected to a different memory module. In this scenario, a memory access can be *local* – i.e. a memory request satisfied by the private memory controller – or *remote* – i.e. a memory request thus satisfied by the memory controller of another core. This choice makes the access latency dependent on which core performs the memory request and on which memory module contains the data. On the other hand, the overall bandwidth is increased. This kind of architectures are called *Non Uniform Memory Access* (NUMA). A NUMA node is composed by the code, the cache hierarchy, the memory controller, and the local memory module. The communication channel between nodes is generally an *interconnect network*. Figure 1.2 shows the structure of a quad-core NUMA architecture.

UMA designs are used in small multi-core architectures, while NUMA designs are used whenever the number of cores grows beyond the practical scalability limit for the UMA approach. Hybrid designs are generally adopted, e.g. Intel QuickPath Interconnect [46] based multi-core show in Figure 1.3. In these multi-cores we have groups of cores. Each group has one integrated memory controller. Groups of cores are connected via a *point-to-point* interconnect.

The advances of semiconductor technologies, according to Moore's law, allow in principle to pack more cores in the same die area. In practice this is not true for the breakdown of Dennard scaling law – i.e., constant power density reducing the size and voltage of transistors. Shrinking the transistors leads to an increment in the power density, and consequently the temperature of the silicon itself. It becomes an issue to keep the entire silicon die to a proper working temperature, thus not all the transistors can be powered on at the same time. The

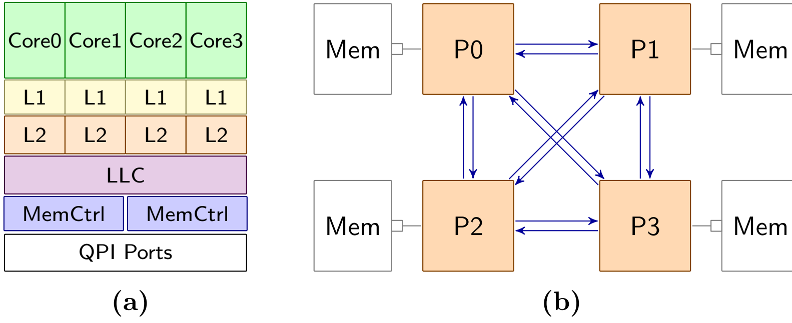


Figure 1.3: Intel QuickPath Interconnect architecture. In (a) is shown the Nehalem architecture. In (b) the interconnection through point-to-point links of four Nehalem processors.

transistors that cannot be powered at a given time instant are defined as *dark silicon* [33].

As a consequence of the dark silicon problem, it's not feasible to increase unconditionally the number of cores, as a thermal runaway phenomenon would ensue. Because of this, multi-core designs are moving towards heterogeneous architectures, in order to better employ the available silicon, and create better power efficient multi-cores. The Cell Broadband Engine [42] is one the first heterogeneous multi-core architectures. It is composed by a *Power Processor Element* (PPE) and eight *Synergistic Processing Elements* (SPEs). These are linked toghether by a high speed bus. The SPEs are optimized for single precision floating point computation. The PPE is based on the PowerPC architecture, a two way multithreaded core acting as a controller for the SPEs. Each SPE is composed of a *Synergistic Processing Unit*, and a *Memory Flow Controller* (DMA, MMU, and bus interface). The SPU instruction set is characterized by 128 bit vector floating point instructions. Indeed each SPE contains 256 KiB of SRAM for instruction and data.

The big.LITTLE [8] architecture designed by ARM is another example of heterogeneous multi-core. Unlike the CellBE architecture all the cores implement the same base instruction set. The multi-core is composed by 4 high performance Cortex-A15 cores, and 4 power efficient Cortex-A7 cores.

1.2.4 Many-core architectures

As a consequence of the power wall, processors do not scale anymore towards higher frequencies. The major trend goes to the integration of more cores per chip. Architecture designs considering a large num-

ber of cores are generally called *many-cores* architectures. We refer to *heterogeneous many-cores* architectures whenever different kind of cores are involved. These architectures fit well applications where the same computation is performed independently on a huge set of input data. Examples of such application are N-Body simulations, PDE solvers, and high-end computer graphics such as animation rendering.

For such workloads the most important figure of merit is the *throughput* rather than the *latency*. To this end, considering the trade-off between number of cores, their complexity, and the total power budget, increasing the number of cores has turned out to be the best choice. To manage the power wall and the dark silicon issues, the trend is to balance the number of cores versus their complexity, and to reduce the clock frequency of the cores.

General Purpose GPUs computing

The first kind of many-core architecture is represented by *Graphic Processing Unit* (GPU) architectures. Starting off in 1999 with NVIDIA's GeForce 256 [67], graphics rendering started to be offloaded on graphics board. Graphics hardware evolved towards stand-alone processors becoming able to completely replace and outperform general purpose CPUs for complex graphics operations.

GPU architectures focus on performing large numbers of mostly independent operations on vertices, and triangles, composing a scene to be rendered. The affinity with general purpose massive parallel applications leaded to *General Purpose GPU*s, where typically computationally intensive parallel tasks are offloaded to exploit the parallel hardware features. In June 2007 NVIDIA released the first version of CUDA – *Computed Unified Device Architecture* – as parallel computing platform working on Tesla micro-architecture.

Modern GPGPU architectures are composed by an array of computational units, i.e. *Stream Multiprocessor*s for NVIDIA or *Compute Unit*s for AMD. These architectures can be classified as SIMD machines in the Flynn's taxonomy.

NVIDIA Fermi Stream Multiprocessor is composed by 32 cores – *Stream Processor*s – containing an integer ALU and a floating point unit each, 16 load/store units, 4 special function units, a shared register file of 32k registers, and 64 KiB of shared memory [66]. The execution unit of a Stream Multiprocessor is called *warp*, that is a group of 32 threads. In a NVIDIA Fermi SM there are 2 *warp-scheduler*s and 2 *dispatch unit*s. The dispatch unit sequentially feeds all the 16 cores with

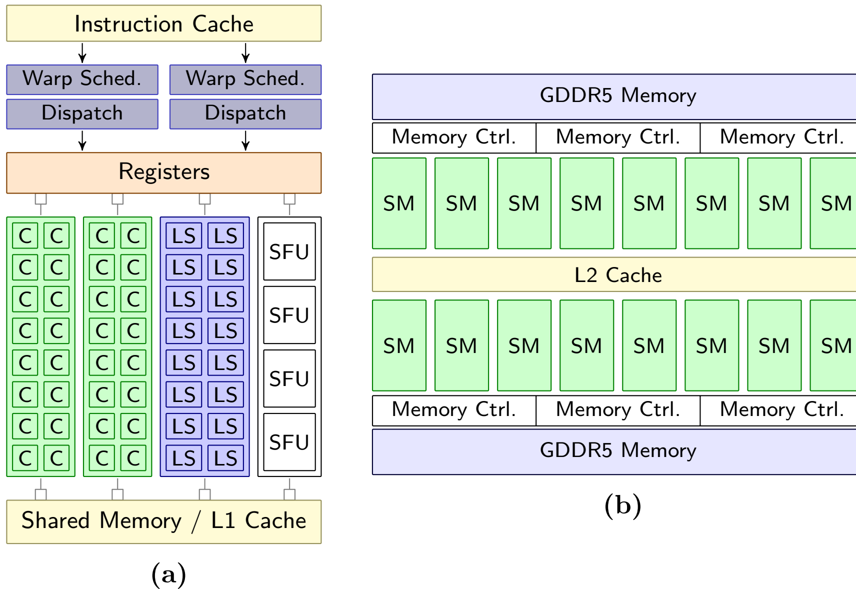


Figure 1.4: NVIDIA Fermi architecture. In (a) the architecture of a Fermi SM, while in (b) the global architecture of a Fermi GPGPU.

an instruction for the selected warp as depicted in Figure 1.4. The result is that the 16 cores work in a step locked fashion.

AMD GCN Compute Unit [5] is composed by a 4 independent SIMD units 16 element wide, a scalar unit, a branch and messages unit, and 64 KiB of shared local memory as depicted in Figure 1.5. Each SIMD unit has 64 KiB of registers, while the scalar unit has 8 KiB registers partitioned in 512 entries for each SIMD unit. The execution unit is called *wavefront*, that is a group of 16 threads – i.e. one for each SIMD lane. Each SIMD unit can hold up to 10 wavefronts. The scalar unit is used to support control flow instructions, including jumps, calls, and returns. Indeed, predication for vector instructions is managed by the scalar unit.

Standalone many-core architectures

Apart from GPGPUs, that were designed to accelerate specifically graphic tasks, and subsequently have been adapted to support also general purpose computation, proper many-core architectures designed for computing have been proposed starting from 2007.

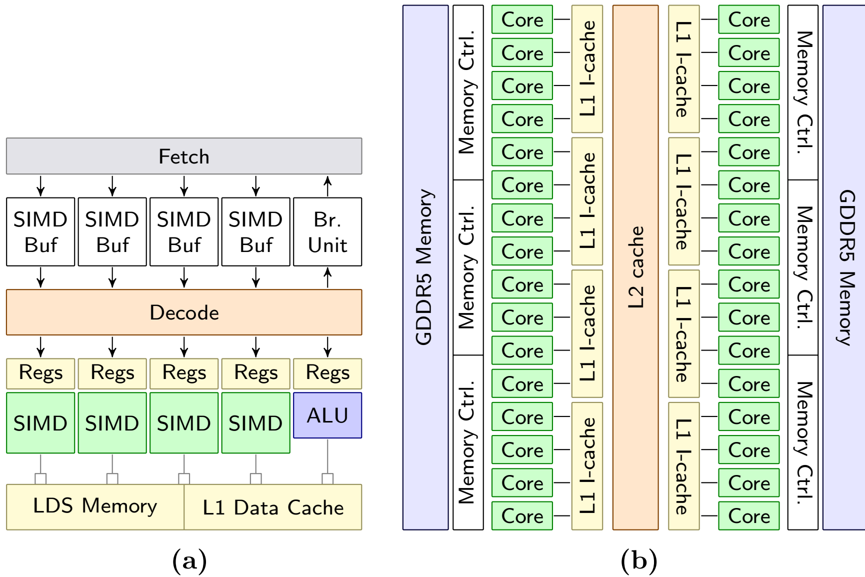


Figure 1.5: AMD GCN architecture. In (a) the architecture of a single GCN core, while in (b) the global architecture of a AMD GCN graphic processor.

Tilera TILE64 released in 2007, it consists of a mesh network of 64 *tiles* [14]. Each tile is composed by a general purpose core, L1 and L2 caches, and a non blocking router used for communication with other tiles. Figure 1.6 shows the tiled architecture of TILE64. Each core is a 3-way VLIW architecture, with two integer ALUs and a load/store unit. The running frequency of the system is ranging from 600 MHz to 900 MHz.

Intel 80-core Teraflops Research Processor presented in 2007, is a tiled architecture [86] similarly to TILE64. Each tile is composed by two single precision FPMAC units, 3 KiB of instruction memory, and a 5 port router for 2D mesh and 3D stacking. The project investigated inter-core communication methods, per-chip power management, and achieved 1.01 TFLOPS at 3.16 GHz consuming 62 W of power. This chip has never been a product, but from this research in 2009 Intel presented the *Single-Chip Cloud computer* [43]. This processor contains 48 *P54C Pentium* cores connected with a 2D mesh. Each tile contains 2 cores, and a router.

Intel Xeon Phi is the brand name of Intel *Many Integrated Core* [31] architectures. In 2010 Intel presented the first prototype – codename *Knights Ferry*. The first commercial version of the MIC architecture –

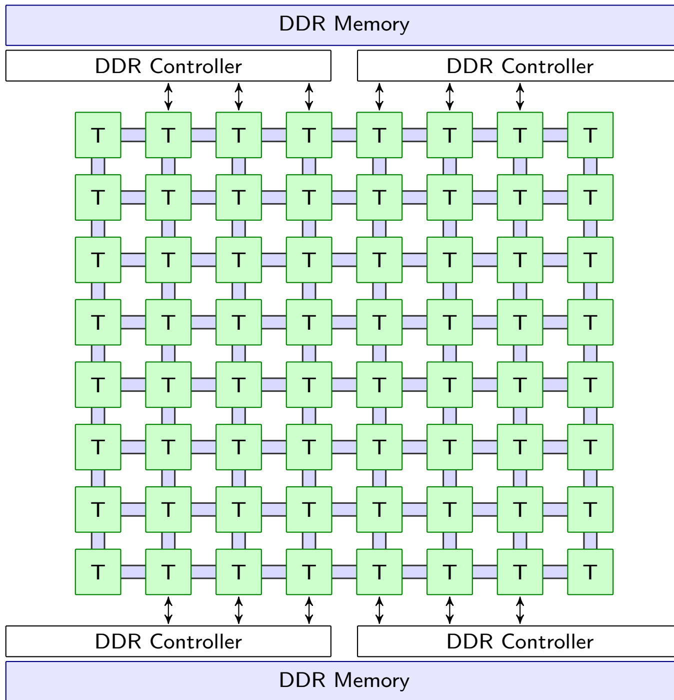
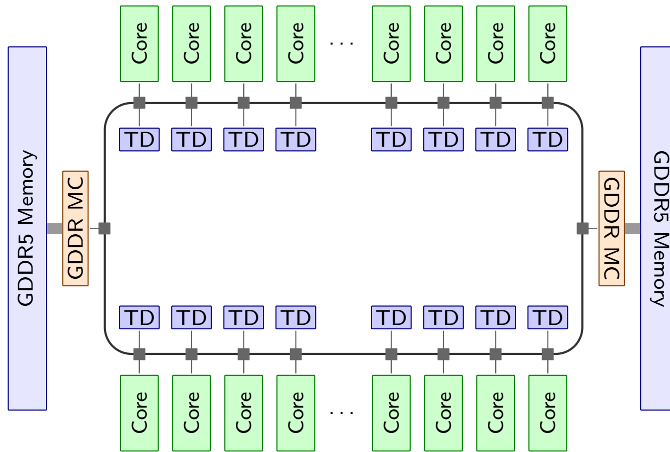


Figure 1.6: TILE64 architecture

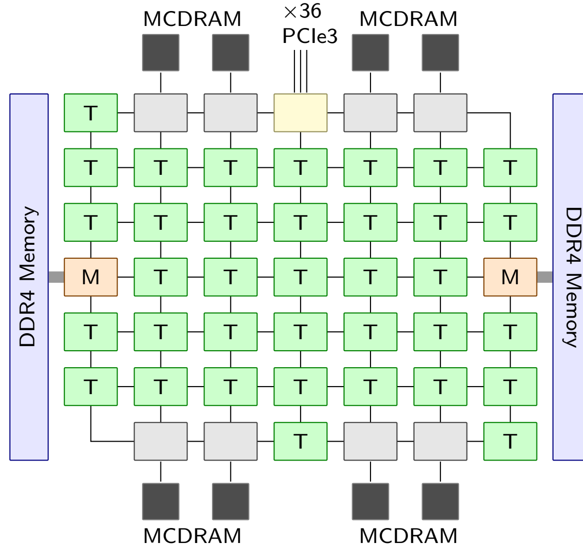
codename *Knights Corner* (KNC) – was released in 2011, and started being employed in HPC systems. The KNC is composed by 60 small in-order execution x86 cores. Each core has 64 KiB L1 cache, 512 KiB L2 cache, and a 512 bit wide vector unit. As shown in Figure 1.7a, the cores are connected through a 512 bit bi-directional ring bus. L2 caches are kept fully coherent by a global distributed tag directory. The peak performance of KNC is ∼1 TFLOPS double precision consuming 300 W. In 2013 Intel presented the second generation of Xeon Phi – codename *Knights Landing* (KNL). The KNL is composed by up to 72 cores, structured in 36 tiles, connected through a 2D mesh interconnect rather than the previous ring bus, as shown in Figure 1.7b. The cores are based on the Silvermont architecture: 4 threaded, out-of-order execution, two 512 bit vector units, supporting almost the entirety of the common Xeon ISA. The peak performance of KNL is ∼3 TFLOPS double precision consuming 300 W.

Power-efficient many-core architectures

GPGPU architectures have been employed in HPC systems since their beginning. Their massive peak performance come at a cost of a con-



(a)



(b)

Figure 1.7: Xeon Phi architectures. In (a) the *Knights Corner* architecture, while in (b) the *Knights Landing* architecture.

siderable power consumption (150 W to 200 W for a single GPGPU). As reported in Section 1.1, the current HPC road-maps aim to reach exascale levels (10^{18}FLOPS) within 2020, which poses the challenge of limiting significantly the overall power budget while providing a massive increases in computational capabilities. In particular the target power budget for future exascale systems ranges between 20 MW and 30 MW.

The best system of The Green500 list of November 2014 reach a power efficiency of 5271 MFLOPS/W which is still one order of magnitude lower than the efficiency needed for supporting exascale systems within 20 MW.

Because of this designers are looking for alternative, more power efficient solution to many-core architectures. Nowadays are becoming more interesting designs from embedded architectures where the power efficiency is a primary goal from long time, while the performance requirements is constantly increasing.

Three interesting architectures are *Kalray MPPA-256* [28], *STMicroelectronics STHorm* [16], and *PULP* [23].

Kalray MPPA-256 released at the end of 2012, is a clustered architecture of 16 clusters of cores interconnected through an explicitly addressed *Network-on-Chip* (NoC). The NoC is a 2D-wrapped-around torus structure providing a bandwidth up to 3.2 GiB/s between adjacent clusters. Indeed a quality of service mechanism is used to guarantee predictable latencies for all data transfers. Each cluster is composed by 16 cores as processing elements, one system core, 2 MiB of shared memory, a DMA unit, L1 instruction and data cache, and the NoC interface. Each core is a 5-issue VLIW architecture with one branch unit, two ALUs, one load-store unit with a reduced ALU, double precision FPU with FMA, and private MMU. The architecture provides a power efficiency of 23 GFLOPS/W at 400 MHz consuming typically 10 W.

STMicroelectronics STHorm – also known as P2012 – is an heterogeneous many-core architecture where the host processor is a dual core ARM A9 paired with a fabric used as accelerator. The fabric is a clustered architecture composed by 4 clusters connected through a global asynchronous NoC. Each cluster is composed by 16 cores as processing elements, one core as cluster controller, 256 KiB of L1 shared memory, and dual channel DMA unit. Each core is an STxP70, an extendible core designed by STMicroelectronics. On P2012 each core is extended with a floating point unit. The core is a 2-issue VLIW with DSP oriented ISA. Barrier operations intra-cluster are implemented in hardware. Indeed, there is one extra core – i.e. fabric controller – used as coordinator for all the clusters on the fabric. At fabric level there is also 1 MiB of L2 shared memory across all the clusters. Indeed within cluster dedicated IPs can replace the general purpose core to accelerate specific functions. The cores within a cluster are connected to the shared memory through a logarithmic interconnect. The fabric can access to a portion of the

main memory of the host processor. DMA units are useful to implement asynchronous data transfer between memories. The fabric with 4 clusters provides a peak performance of 80 GFLOPS (single precision) consuming 500 mW per cluster at a frequency of 600 MHz.

PULP Platform – the Parallel Ultra-Low Power Platform – is a on going project developed by ETH Zurich and Università di Bologna under the lead of Prof. Luca Benini, the chief architect of P2012. The design of such platform is inspired by the results of P2012. However the target of this platform are ultra-low power embedded systems. Each core is an OpenRISC architecture based on the opensource OR1200 microarchitecture. The idea was to replace the proprietary STxP70 cores with smaller but more power efficient ones. To this end, the microarchitecture has been optimized, and the ISA has been extended to add DSP-like features to improve the performances (more details in Section 2.3). The fabric is composed by a single cluster of 4 cores. Within the cluster there are also 48 KiB of TCDM, 256 KiB of L2 memory, and a dedicated DMA unit. All the cores within a cluster are attached to a 4 KiB shared instruction cache. PULP version 3 has been implemented using ST28 FDSOI technology with RVT transistors. The maximum frequency is 66 MHz with a power supply voltage of 0.6 V. In such condition the consumed power is roughly 1.59 mW. Another implementation using UMC65 technology reaches the maximum frequency of 362 MHz and is capable of processing 1.4 GOPS at a power budget of 93 mW.

1.2.5 Future perspective

It is clear the hardware evolution trends is converging towards many-core heterogeneous architectures. The limits in scaling with huge and complex cores, and the increasing emphasis on the power consumption moved the interests of architectures designers in architectures with a considerable amount of cores with different features and complexity. Many-core architectures started with GPU designs, where a great number of simple cores is employed to execute a single instruction flow on multiple independent data. However the interest in irregular parallel problems has shown limitations of such architectures. This is leading towards hybrid architectures, i.e. a mix of multi-cores and many-cores, where the complexity of each core is increased to better fit irregular control-flows and non predictable data access patterns. Indeed the current trend is moving towards integrating such cores in the same die area to reduce the cost of communication between heterogeneous cores. An example of such trend is the introduction of *shared virtual memory* between CPU

and GPUs, e.g., the HSA Foundation [44] requires all the agents to share the same virtual address space. Once the decision of integrating general purpose CPUs and GPU cores on the same die, it is feasible, and much desirable, to interface them with the same main memory. Indeed to simplify the programmability and to remove the overheads due to useless memory transfers, it is interesting to make CPUs and GPUs share the same virtual address space similarly to what already happens within modern multi-cores. This is currently a challenge considering the goal to have transparent coherency between cache hierarchies of both CPUs and GPUs. The OpenCL 2.0 programming model exposes this feature to the programmer in two forms: *coarse-grain SVM* and *fine-grain SVM*. The former is the minimum requirement where at least explicitly allocated buffers can be shared between CPUs and GPUs, and are kept coherent at given synchronization points. The latter is optional as it requires that the whole address space to be shared, and must be coherent also on atomic operations. Modern GPU architectures implement only the coarse-grained model of SVM, as it can be implemented without the need for support for lazy page loading, i.e. no page-fault exceptions, and without the support for cache coherency protocols, because the coherency must be ensured only at given synchronization points, thus this can be enforced by the software runtime. The fine-grain SVM model requires much more hardware support still not available. However the fact that such features have been introduced in programming models suggests that in the near future the hardware will move towards providing them.

1.3 Parallel programming models

Exposing the control of parallel hardware features to the programmers allows architecture designers to increase the overall architecture performance. However this makes the task of programming these architectures much more complex.

To take advantage of parallel processing elements there are two possible approaches: *implicit* and *explicit* techniques.

Implicit techniques aim to hide as much as possible the details of the target architecture from the programmer perspective. The actual exploitation of parallelism is achieved transparently at run-time or compile-time. For example ILP exposes to the programmer a sequential programming model, while the actual execution may be performed in parallel. In an out-of-order architecture ILP acts at runtime dynamically identifying independent instructions, and executing them in parallel using

different functional units. At the other end of the architecture design spectrum, in a VLIW architecture the compiler is in charge of scheduling instructions to exploits the different functional units available.

Explicit techniques, instead, expose the hardware parallelism to the programmers. This allows developers to have a greater control on the mapping of the application on the hardware. On the other hand, it becomes the programmer's responsibility to identify and map the parts of the application which can be efficiently parallelized. To this end explicit parallel programming models are needed.

A programming model is the description of an abstract architecture used as reference architecture to describe how computation is performed. The concepts of a programming model are strongly connected with the class of hardware architectures it targets. Parallel programming models are built around the concept of multiple execution flows. The ways multiple execution flows may interact with each other are determined by the programming model rules, and generally depend on the hardware features available.

How a given computation on a data set is mapped on multiple execution flows determines the nature of the parallelism: the two base approaches are named *data* and *task* parallelism.

Data parallelism partitions the input data set mapping the processing of each chunk to an execution flow. All the execution flows are fed by the same instruction stream but compute on different data streams. This computational model fits SIMD class of Flynn's taxonomy.

It fits also extensions of this class, like *Single Instruction Multiple Thread* (SIMT) – the same instruction is executed by different cores in a step-locked fashion – and *Single Program Multiple Data* – the same program is executed by different cores on different data but the threads are independent. It is often said that an application suitable for a data-parallel programming model exhibits a *regular behaviour*: not control intensive, with a large number of operations often on large arrays accessed with fixed strides.

Task parallelism partitions the computation in independent tasks mapping each one to a different execution flow. The *task* is the unit of parallelism. A parallel computation is originated by the creation of new tasks. The act of creating a new task is called *spawning*. Spawning a task represents the intent of execute the task in parallel with the current execution flow. Synchronization between tasks is usually achieved by *joining*: a task can wait for the termination of another task before

moving on with its execution. The relationships between tasks can be represented by means of a *task-graph*. Each node in the graph is a spawned task. Arcs between nodes can be either spawning actions or joining actions. This computational model fits MIMD class of Flynn's taxonomy and its extensions such as the *Multiple Program Multiple Data* (MPMD). This model is said to be suitable for applications with an *irregular behaviour*: they are generally control-intensive, accessing data in a non predictable way.

In the following, a brief overview of the most adopted parallel programming models suitable for programming of multi-core and many-core architectures, and a comparative analysis of the interesting features of such programming models is provided. This analysis aims to highlight modern trends of parallel programming models illustrating the different approaches to express the parallelism exploiting the available hardware features.

1.3.1 Overview

Several programming models and frameworks have been proposed in the literature for programming parallel and heterogeneous architectures. The interest in this chapter is towards the most adopted programming models for multi-cores and many-cores architectures. Programming models for distributed systems, such as MPI and PGAS models, are not discussed as they lie beyond the scope this overview. The selected programming models are *OpenCL* [50], *CUDA* [65], *OpenMP* [6], *OpenACC* [70], *Codeplay Offload* [24], *C++AMP* [64], *SYCL* [53], *Cilk* [36], *Intel TBB* [25], *StarSs* [15, 19], *SkePU* [32, 10], and some *DSLs*.

OpenCL is a framework for programming parallel heterogeneous platforms. The architecture of the OpenCL platform, shown in Figure 1.8a, is composed by an *host* connected to one or more *devices*. Each device is composed by one or more *compute units*. Each compute unit is composed by one or more *processing elements*.

An OpenCL application is generally split into two parts: the *host program*, and the *kernels* for the devices. The kernels represent the computation executed on the devices. The host program defines the context for the kernels and manages their execution.

The OpenCL execution model describes how kernels are executed. When a kernel is submitted to a device for execution by the host, an index space is defined. An instance of the kernel executes for each point in the index space. This kernel instance is called *work-item* and it is

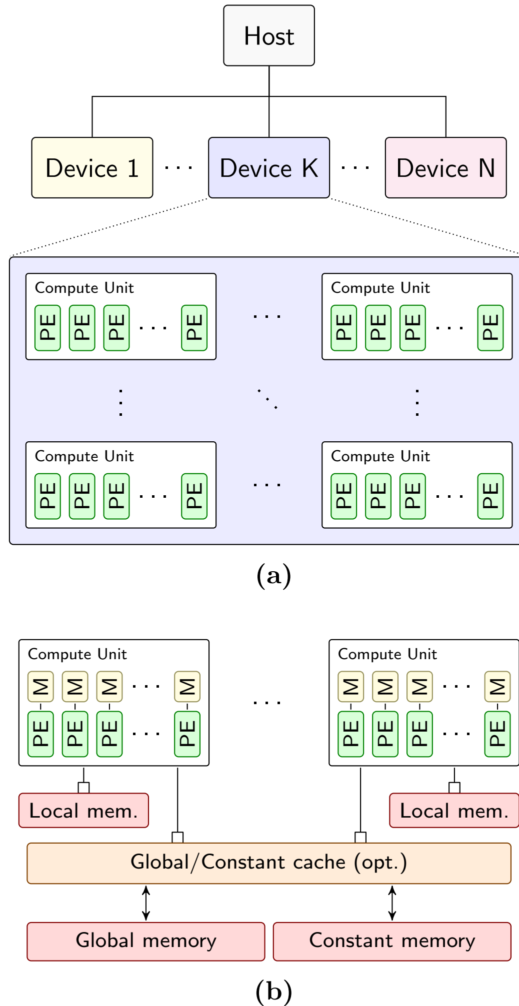


Figure 1.8: The OpenCL platform. In (a) the relationship between host and devices, and the structure of a device, while in (b) the memory hierarchy within a generic device.

identified by its position in the index space. Work-items are organized in *work-groups*. The work-groups are the result of a tiling process onto the index space. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. A *barrier* operation is indeed defined between the work-items in a given work-group.

There are four distinct memory regions in OpenCL, illustrated in Figure 1.8b. The *global* memory can be accessed by all the work-items in all the work-groups. The *constant* memory is specialization of global memory for constant data. Both constant and global memory can be

accessed by the compute units through a cache hierarchy. Each compute unit has a dedicated *local* memory, that is a shared memory space between work-items in a given work-group. The *private* memory is used for private variable of each work-item.

The OpenCL execution model is suitable for both data-parallel and task-parallel approaches. However the execution model fits better data-parallelism .

The host-program and the OpenCL kernels are expressed in different languages. The OpenCL runtime has a standardized set of C APIs. The runtime offers full control on the management of memory buffers allocation, and the execution of kernels across the devices. OpenCL kernels are described through the OpenCL-C programming language. This language is a restriction of C99 language (e.g. no recursion, no variable length array) plus some extensions to support vector types and builtin functions.

OpenCL claims functional portability of an application, as the programming language for the kernels is target independent. However it is still possible to write code for specific targets relying on implicit knowledge of the underlying architecture. This freedom is useful to extract more performance from the underlying parallel hardware, however it limits the portability of the code itself.

OpenCL 2.0 introduced some new features, such as the support to *shared virtual memory* between host and devices, the generic address space, and the support for *dynamic parallelism*, i.e. the possibility of “spawning” new kernels on a device with no needs for host interaction.

CUDA is the programming framework of NVIDIA for its GPUs. Almost all the key concepts are the same as in OpenCL. In the CUDA parlance work-items are called *threads*, and work-groups are called *blocks*. The memory model is equivalent to the OpenCL 2.0 one.

The main differences resides in the interface to the programmer. In particular, CUDA allows to specify the host program and kernels in a single translation unit. The CUDA programming language is an extension to C/C++ with attributes for tagging functions that must be compiled only for the device, for tagging variables with the appropriate address space modifier. Indeed a new syntax to invoke kernels from the host part of the program is provided.

OpenMP is a programming framework for multi-cores. It consists of a set of compiler directives, library routines, and environment variables. It is available for C, C++, and Fortran. OpenMP follows the *fork-join*

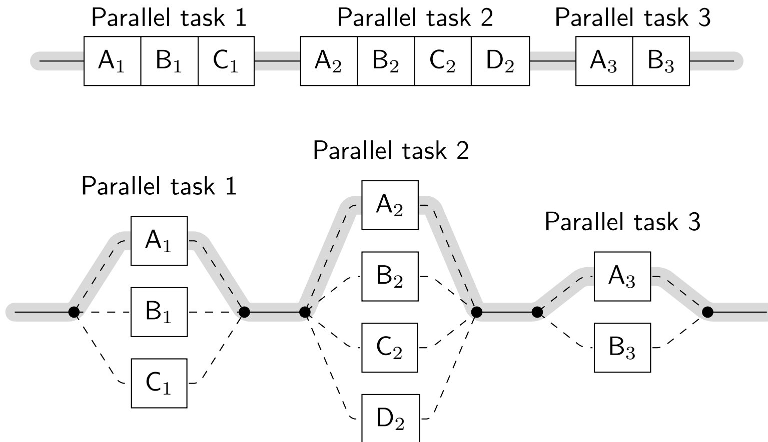


Figure 1.9: Fork-join model. The execution path of the master thread is highlighted in grey. On the top of the picture the sequential execution flow, where only the master thread is active. On the bottom of the picture the parallel execution flow, where the master thread forks to generate worker threads to execute the tasks in parallel, and joins them to synchronize.

model, show in Figure 1.9. The program starts with a single thread called *master-thread*. At the beginning of a parallel region, the master threads creates – *fork*s – a team of parallel worker threads. An instance of the parallel region body is executed by all threads. At the end of the parallel region all the threads synchronize, and *join* the master thread.

Using OpenMP both task-parallelism and data-parallelism strategies can be implemented. The compiler directives are used to describe parallel sections and whether some variables are shared or private. These directives are platform independent, allowing an high degree of portability. Furthermore, it enables incremental parallelism as ideally only additional compiler directive should be added to enable parallel execution of a piece of code. However synchronization bugs and race conditions may occur. Starting with OpenMP 4.0 it is possible to offload parallel regions to accelerators such as GPGPUs.

OpenACC is a programming framework for parallel computing that targets CPU-GPGPUs heterogeneous architectures. Similarly to OpenMP it is based on compiler directives to describe parallel regions, and manage data movement from/to the GPU memory. Like OpenMP, it supports C, C++, and Fortran programming languages. It is common for an OpenACC compiler to emit OpenCL/CUDA code (or equivalent intermediate representations such as SPIR) for the kernels, and rely on

vendor specific OpenCL/CUDA drivers to offload the computation on the GPUs.

Codeplay Offload is programming model for both homogeneous and heterogeneous multi-core processors such as the Cell Broadband Engine. It allows to offload portions of large C++ applications to be run on accelerator cores. The code that must be offloaded is wrapped in a *offload* block, indicating that the code should be compiled for an accelerator, and executed asynchronously as a separate thread. The main point is to ensure the portability across heterogeneous and homogeneous platforms, and relieve the programmer from the burden of writing data movement and accelerator setup code.

C++ AMP is programming framework that extends C++ to support data-parallel computation on parallel accelerators like GPUs. The initial implementation by Microsoft was targeting DirectX 11 drivers. It has been extended to target both OpenCL and HSA compliant platforms. C++ AMP is based on the C++11 language and introduces function modifiers to specify the fact a function should be compiled for the device or not. The codebase of kernels is expressed using lambda functions as arguments to the `parallel_for_each` template function. The data access by kernel code can only be done throughout the `array` and `array_view` wrappers. These wrapper are used as proxy for data that must be transferred on the accelerator. Although the base language is C++, the code constructs which can be employed within a kernel are constrained to match the requirements for offloading the computation using existing OpenCL drivers.

SYCL is a programming framework built on top OpenCL. It allows to specify a parallel program exploiting OpenCL devices in a single C++ source. Kernels are expressed similarly to C++AMP through a lambda function as a parameter to the `parallel_for` template function. This function takes as input the work-item identifier, and access global and local data through accessor objects captured by value. The accessors acts like a proxy for data that must be transferred on the accelerator. The SYCL compiler identifies kernels, generates OpenCL code (or equivalent intermediate representations like SPIR). The SYCL runtime is responsible for wrapping the OpenCL driver in order to offload kernels on the accelerator. The SYCL compiler is responsible for auto-deducing address spaces of variables in the most common cases, however the explicit pointer classes can be used to indicate the belonging of a given memory

region to a specific address space. A SYCL program is valid C++ program that can be compiled by any C++ compliant compiler, and can be executed on the host as fallback for platforms with no OpenCL support.

Cilk/Cilk++ are general purpose programming language designed for multi-threaded parallel computing. They are based on C and C++ languages, and they extend these with constructs to express parallel loops and spawn/join tasks. Indeed they simplify the syntax for array operations to help the compiler to effectively vectorize the operations. The task scheduler in Cilk uses the *work-stealing* policy to balance the workload across the cores.

Intel TBB is a C++ template library intended to support task parallelism on multi-core processors. Tasks are scheduled using the work stealing policy like in Cilk. It is composed by generic algorithms (such as `parallel_for`, `parallel_reduce`), concurrent containers, scalable memory allocators, mutual exclusion primitives, and a task scheduler.

StarSs is a programming framework developed at Barcelona Supercomputing Center. It is directive based like OpenMP and OpenACC, it targets multi-core and heterogeneous accelerators. The directives are used to express data dependencies between tasks. These are organized in a task-graph according to their data dependencies. Task with no predecessors represent ready tasks. The runtime moves them from the task-graph to the ready queues of worker threads. Data dependencies refer to memory locations, expressed as base address and size. An input dependency represents a location that is read by the task. An output dependency a location that is written by the task. An input/output dependency represents a location read and written by the task.

SkePU is C++ template library of "skeletons". A skeleton is a predefined, generic component such as *map*, *reduce*, *scan*, *farm*, *pipeline*, etc. that implements a common specific pattern of computation and its data dependencies. Skeletons provide an high degree of abstraction and portability, as their implementations encapsulate all low-level and platform specific details, such as parallelization, synchronization, and communication. It supports multi-cores and multi-GPUs systems both with OpenCL and CUDA. It implements an auto-tuning framework for a context-aware dynamic selection of the expected fastest implementation variant of each skeleton. Indeed it ca be integrated with the *StarPU* runtime system. StarPU is a task programming library for heterogeneous

architectures, it manages automatically all the data transfers between devices.

Domain specific languages are employed to simplify the specification of algorithms and programs relative of a given application domain. They generally are high-level languages that completely hide the detail of the target architectures. Such approaches to parallel programming can be very effective for those domains where applications have intrinsic parallelism. Some examples are the fields of *image processing*, *computer graphics*, *streaming applications*, *linear algebra*, and *graph analysis*. The use of high-level languages gives the programmers the ability to express the target application in a natural way for the given domain. The compilers for DSLs can exploit the properties derived by the application domain to generate more efficient code for multi-cores and accelerators.

StreamIt [83] is an example of DSL for streaming applications. A StreamIt application is the composition of *filters*. These can be connected in cascade to generate a *pipeline*, or through special nodes, *splitter* and *joiner*, for splitting/joining the data stream on several parallel filters.

GLSL – OpenGL Shader Language – is a domain specific language for computer graphics [72]. This language allows the specification of graphics shaders to be executed by GPUs.

Writing a DSL with the relative compiler is not a trivial task. There exist frameworks that simplify the implementation of a DSL compiler, like *Delite* [82] and *Pencil* [12]. These frameworks provide a platform-neutral intermediate representation for DSLs.

1.3.2 Comparison

The aforementioned programming frameworks cover most of the approaches to write program for parallel architectures. Interesting dimensions of analysis are the following:

- *Abstraction Level*, the degree of detail exposed to the programmer to expressthe parallelism. This aspect is important in terms of programmability because more details explicitly under control of the programmer, on one side they imply the possibility of fine-tuning the mapping onto specific target architectures, while on the other side limit the portability the application as the parameters value are inherently target specific. An high abstraction level ensures a high degree of portability as all the details of the architecture are hidden from the programmer.

- *Parallelism flavor*, the ability to express data or task parallelism. While data-parallelism has been the first kind of investigated parallelism, task-parallelism is becoming much more interesting as also irregular applications are moving towards heterogeneous architectures.

- *Heterogeneity*, the ability to target heterogeneous parallel architecture, thus suitable for modern many-cores.

- *Extended Language*, whether the parallelism is expressed by means of extensions to the base language. Solutions employing language extensions are less portable as an extended compiler is needed, while a pure library based approach is immune to this issue. On the other side the use of an extended compiler gives more opportunities to accurately optimize the generated code.

- *Elision property*, this feature implies the fact that if the features introduced by a language extension are ignored, the resulting program is a correct sequential version of the parallel program. This property is useful for testing purposes and to ensure compatibility of the source code with platforms where such extensions are not available.

- *Data management kind*, whether data moves are under the explicit control of the programmer or are implicitly managed by the compiler/runtime system. This an important part related to the abstraction level.

Table 1.1 shows the classification of the analyzed programming frameworks. Almost all the frameworks support both data and task parallelism, even if few of them, e.g. OpenCL, are biased towards data parallelism. Most of them support heterogeneous architectures confirming the trend towards heterogeneous parallel architectures.

OpenCL, CUDA, and with a lower degree SYCL and C++AMP are classified as low/mid-low level of abstraction programming framework because the programmer is either required or allowed to specify how to map a data-parallel computation onto a given device. In particular for OpenCL and CUDA, the programmer is also responsible for data transfers between host and accelerators increasing the boiler-plate code required and the complexity of programming for such parallel accelerators.

Frameworks based on compiler directives such as OpenMP, OpenACC, StarSs offer a high degree of portability, as their directives should be

Table 1.1: Comparison of parallel programming frameworks

	Abstraction Level	Data/Task	Heterogeneity	Extended Language	Elision property	Data moves
OpenCL	low	both[1]	yes	yes	no	explicit
CUDA	low	both[2]	yes	yes	no	explicit
OpenMP	mid	both	no[3]	pragmas	yes	implicit
OpenACC	mid	data	yes	pragmas	yes	implicit + hints
Codeplay Offload	mid	task	yes	yes	no	implicit
C++AMP	mid-low	data	yes	yes	no	implicit
SYCL	mid-low	both	yes	no	n.a.	implicit
Cilk/Cilk++	mid-high	both	no	yes	yes	implicit
Intel TBB	mid-high	both	no	no	n.a.	implicit
StarSs	mid-high	task	yes	pragmas	yes	implicit + data deps
SkePU	mid-high	both	yes	no	n.a.	implicit
DSLs	high	both	yes	custom	n.a.	generally implicit

[1] OpenCL support both data and task parallelism, however data parallel is the preferred model.
[2] CUDA initially supported only data parallel model. Task parallelism has been introduced with CUDA 5.0
[3] OpenMP historically targets homogeneous multi-cores. Offload to accelerators has been introduced in OpenMP 4.0.

target-independent. This is a lightweight approach to integrate parallelism annotations within pragmas or comments. However directives for offloading the computation to parallel accelerators may present parameters that depend on the specific knowledge of target architecture, limiting the portability of such approach.

A more intrusive way to express parallelism is to extend existing programming languages such as C, C++, or Fortran to support parallel constructs. This allows for a better integration of such constructs while keeping the quantity of new notions to learn for the programmer at minimum. The compiler for such languages is in charge of lowering such constructs in the best possible manner, on a runtime layer that exploits parallel hardware features. When dealing with language extensions a desirable property is *elision*. Cilk/Cilk++ ensure this property, while C++AMP does not. Directive based extensions such as OpenMP, OpenACC, and StarSs ensure the elision of the directives that are added on the sequential version of the program.

On the other hand, SYCL acts like a C++ library, thus not extending the language. However it relies on compiler modification for the generation of parallel code targeting OpenCL devices. This implicitly ensures the elision property as the language, from a syntactical and semantic point of view, is just pure C++ with a fallback implementation to be used whenever the code is compiled with a standard C++ compiler.

Data management is another critical point with respect to the programmability of heterogeneous architectures. High-level approaches hide the data management to the programmer. This solution reduces the programming effort, however the runtime system must properly handle data buffers. When a compiler is used to generate parallel code, it may optimize the data movements reducing the communication overhead.

The main weakness of DSLs is the inherited limitation to the given domain, thus the specific solutions may not be applicable to other domains. On the other hand, a DSL ensures the most compact and natural representation of a program, allowing the compiler to apply domain specific optimizations, thus leading to a more efficient generated code for parallel accelerators.

1.4 Conclusion

The analysis of the evolution of hardware architectures tells us that architecture designs are converging towards heterogeneous many-cores, that seem to be the path to exploit as much as possible the silicon area available with the current semiconductor technologies. Such architec-

tures are characterized by a large number of different cores, generally with a few complex cores coupled with several simpler ones used to accelerate parallelizable computations.

The programmability of such architectures is still a complex task since a great portion of the hardware features are directly exposed to the programmers. Programming frameworks trying to hide such complexity exist, but either they are either sub-optimal with respect to hand tuned implementations, or constrained to specific domains.

OpenCL is the de-facto standard for programming heterogeneous parallel architectures. With OpenCL programmers have full control on the mapping of the parallel computation onto the hardware, and the interaction between host and accelerators. Even if OpenCL applications should be portable, it is still possible to write non-portable target specific code with respect to both functionality and performance. Under these conditions, the problem of porting applications between different platforms cannot be automated, and requires a significant re-engineering effort. It is interesting to understand whether it possible to mitigate this problem, allowing programmers to be able to execute OpenCL applications written with a different platform in mind.

Moreover, new architectures introduce interesting features designed to simplify the programmability of such platforms. The most interesting example is shared virtual memory. This is a strict requirement for both OpenCL 2.0 and HSA compliant platforms. This feature will become commonly available in the near future. However current programming models are not exploiting completely such feature. The presence of a shared and coherent virtual address space between host cores and accelerators will allow further improvements to the programmability of heterogeneous architectures. Thus it turns out to be very interesting to explore the possibilities behind this opportunity for the next generation of heterogeneous hardware architectures.

2 Software/Hardware Architecture

In this chapter, we present OpenCRun, our implementation of the OpenCL runtime support targeting platforms with a wide range of characteristics, from high-end multi-cores to low-power accelerators. We provide an overview of the most relevant engineering decisions. Since low-power many-core accelerators are the main target of our investigation, we also provide insights on the co-development of ISA extensions and the related compiler support for the PULP platform.

2.1 Introduction

The development of a compiler and runtime for a complex, industry grade language such as OpenCL is a complex task. Yet, to effectively perform research on OpenCL, there is a need to access the internals of the runtime and compiler. Therefore, proprietary, closed source tools are not a choice. For this reason, in 2011 an effort was started within the EU FP7 2PARMA project [78] to develop a compiler front-end [58], back-end [62, 38], and a runtime for OpenCL, called *OpenCRun*. In parallel, other research groups started working on Open Source OpenCL runtimes. Notable among these efforts are POCL [47], FreeOCL [3], and SnuCL [54]. POCL focuses mostly on performance, and leverages LLVM and Clang as its compiler, whereas FreeOCL aims at being independent from the compiler, allowing the user to select either LLVM/Clang or GCC, and possibly other compilers. SnuCL instead target heterogeneous clusters, exposing to the programmer a set of devices derived from the computational units distributed among the whole cluster. It is worth noting that our research goal is to study functional and performance portability. Thus our runtime targets architectures belonging to widely different classes – server-class x86-64 machines and low-power many-cores such as STHorm and PULP. This focus is unique among the currently available implementations of OpenCL, and makes OpenCRun an interesting choice for those interested in exploring OpenCL on embedded many-core accelerators.

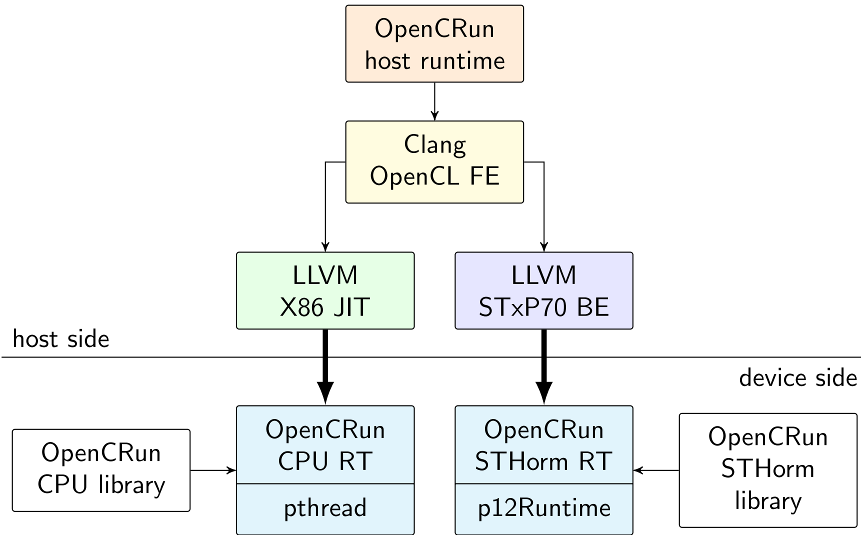


Figure 2.1: OpenCRun toolchain overview for X86 and STHorm.

2.2 OpenCRun

OpenCRun is written in C++ and aims at providing a multitarget OpenCL 1.2 infrastructure, based on the LLVM and CLANG open source frameworks. Currently the runtime targets *x86 multi-cores*, and the *STHorm* platform. In the following a description of main components of the runtime is provided. The infrastructure is composed of four logical components:

- **host runtime**, which implements the OpenCL APIs;
- **device description**, as seen by the host;
- **device runtime** providing for each device the OpenCL runtime;
- **device runtime library**, providing the implementation of the builtin functions.

Figure 2.1 depicts the complete toolchain overview: from the host runtime the OpenCL source code is compiled using the OpenCL frontend. After the translation in LLVM IR is performed and the LLVM optimizer is run on it, depending on the selected device, native code is generated. For the x86 device, native code is emitted directly in memory through the JIT component of LLVM. For STHorm, a shared object must be generated to be deployed later on the device memory. The execution environments are specific for each device. For the x86 device, each core

has a pinned POSIX thread where the computation of work-groups is issued. For STHorm, an active runtime layer is used to coordinate the execution of OpenCL commands. The generation of OpenCL builtins implementations is fully automated using a TableGen-based tool[1]. The generation is based on an abstract description of each builtin variant, a basic implementation for scalar variants, and the strategy that must be used to build the vectorized variants. Target specific overrides are allowed for optimized implementation.

In 2.2.2 and 2.2.3 the details of the implementation of the x86-CPU device and the STHorm device respectively are described. In 2.2.4 the details of the automatic generation of all the builtin functions are provided.

2.2.1 Host-runtime architecture

The host runtime implements the OpenCL API exposed to the programmer. All the concepts of the OpenCL programming model are represented by classes:

- `Platform`, a singleton class representing the OpenCRun platform,
- `Device`, representing an OpenCL device,
- `Context`, used to track objects such as memory objects, programs, and queues for a fixed set of devices,
- `Program`, representing an OpenCL program, usually created from a string of OpenCL-C code,
- `Kernel`, a single kernel function,
- `CommandQueue`, a queue of commands for a specific device in a given context
- `Event`, associated to commands, to provide synchronization among them, useful to impose a partial order between commands for an out-of-order queue,
- `MemoryObject`, any memory buffer used for the computation.

The host-runtime is also in charge of handling the compiler for OpenCL programs. The compiler is based on CLANG and LLVM frameworks. OpenCRun employs CLANG to translate OpenCL-C code to LLVM-IR for each device. During this process custom *metadata* are computed

[1] TableGen is a component of the LLVM framework

and stored in the LLVM module. Once the LLVM module has been generated, a custom optimization pipeline is set up by the runtime. The optimized module is stored then in the corresponding instance of `Program`.

Command queues handle the different OpenCL commands: they are in charge of submitting commands to the corresponding device and manage the coherency among cached versions of buffers across the devices in the same context.

The class `Device` is the abstract class for all the OpenCL devices we target, i.e. `CPUDevice` and `STHormDevice`. Each device must specify how to allocate memory on the device, how to handle the various commands such as buffer reads/writes and kernel executions.

2.2.2 CPU device

The `CPUDevice` class represents the OpenCL device mapped on the host machine. It is used to run OpenCL applications on multi-core systems. The device uses POSIX threads to run several kernels in parallel. When the runtime is initialized, an instance of `CPUDevice` is created. The device is composed of a set of threads, one for each core exposed by the operating system. Each of the threads is pinned to a core modifying the affinity of the given thread. These threads are the pool of workers where commands are dispatched onto. Each thread has its own device command queue, and processes commands sequentially.

The device uses the LLVM JIT component to generate native code from the LLVM-IR module on the fly. When an enqueue kernel command is received, device-specific commands for each work-group are generated and dispatched on the thread pool. The work-items of a given work-group are executed sequentially by the worker thread. Barrier operations are implemented by means of a fast context-switch mechanism between work-items. At the beginning of the command processing, a single chunk of memory is allocated for all the work-items. The layout for each work-item, as shown in Figure 2.2, is composed of a pointer to the next work-item slice, a region dedicated for saving the processor context, a pointer to the arguments, a pointer to the function, and space for the work-item stack. The private memories of each work-item are linked together in a circular single-linked list.

A thread-local pointer variable is set to the current work-item private memory. On a barrier operation, the status of the registers is saved in the dedicated region of the current work-item private memory. The thread-local variable is then updated to point to the next work-item private memory, and finally the context of the next-work item restored.

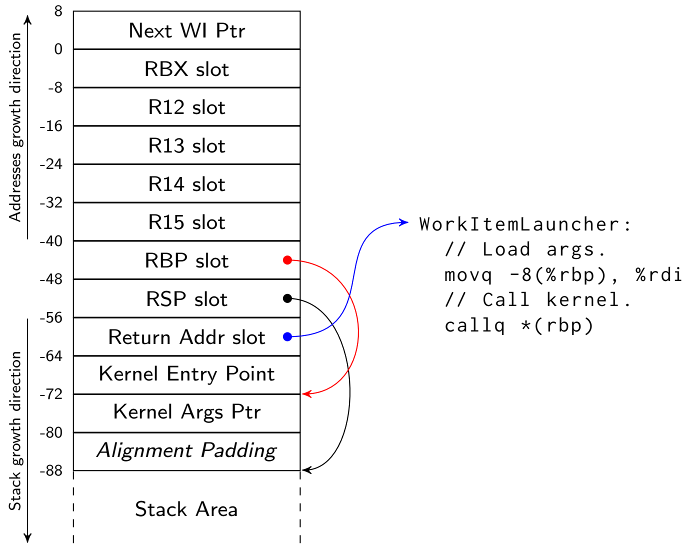


Figure 2.2: Initialization of work-item private memory for the x86-64 architecture. The `rsp` slot is initialized with the address of the *end* of the area reserved to the stack as it grows downward. The *return address* slot is initialized with the address of a custom trampoline used to start the execution of the work-item. This trampoline assumes that `rbp` points to the *kernel entry point* slot, thus the `rbp` slot is initialized with the address of the kernel entry point slot. Apart for the initialization value before the execution, the return address slot is used to hold the proper return address value across work-item switches.

An experimental support to whole-function vectorization [48] is currently under development. This approach is useful to vectorize kernels across work-items allowing for a more aggressive use of SIMD capabilities of x86 processors.

2.2.3 STHorm device

The `STHormDevice` represents the OpenCL device mapped on STHorm fabric. This class uses the Linux driver interface for the interaction with the fabric. The native runtime resides on the fabric, implemented as thin layer, offering the basic services such as memory allocation on the different memory spaces, and function execution on the processing elements. The OpenCL device-runtime is built on top of this resident runtime in form of DSO (dynamic shared object) that can be loaded by the resident runtime. The interaction between host and fabric is based on message boxes between the host and the fabric controller. The initialization of the OpenCL device starts with the deployment of the OpenCL

device-runtime on the fabric and its loading. OpenCL memory allocations are forwarded as memory requests for the resident runtime. The LLVM module representing the OpenCL program is compiled through the STxP70 LLVM backend [2] and linked with the runtime libraries to generate a DSO. The DSO is then loaded on the fabric. Kernel execution commands are lowered to the corresponding device-runtime command referencing the appropriate entry point in the loaded DSO.

On the device side, the fabric controller receives all the commands. It dispatches the execution of work-groups to the cluster controllers. On the completion of all the work-groups the fabric controller notifies to the host the kernel completion.

2.2.4 OpenCL builtin library

A considerable effort to support OpenCL is the implementation of all the builtin functions, i.e. more than 6600 functions. Most of them are overloaded versions for vector types, thus it is possible to implement them starting from the scalar version only. To simplify the task of specifying each builtin function and automatically generate all the overloaded versions of builtins, we created a simple tool called `oclgen`. This tool is based on TableGen, i.e. a description language used within LLVM, and its purpose is to produce the list of declarations and an OpenCL-C implementation for all the builtins.

```
class MathBuiltin_rrSn<string name>
  : OCLGenericBuiltinSimple<
  name,
  [ocl_gentype_real, ocl_gentype_real, ocl_s_int],
  [isSameAs<Id<0>, Id<1>>, isSameDimAs<Id<1>, Id<2>>]
>;
```

Figure 2.3: OpenCL builtin prototype description within `oclgen`

We described the OpenCL type system in terms of TableGen entities, and from this we described the prototype families of the builtins based on their signature and the constraints between arguments and return types. Figure 2.3 shows an example of the prototype family used to specify a builtin with a generalized floating point return type, a generalized floating point type as first argument, and a generalized signed 32 bit integer as second argument. The generalized types include vectors and

[2] Further details about the STxP70 LLVM backend can not be disclosed since it was developed under a non disclosure agreement with STMicrolectronics within the framework of 2PARMA EU-FP7 project.

scalar types of the given base type. The prototype is constrained by the fact that the type of the return value and the one of the first argument must be the same, and that the first and the second argument must have the same dimension, thus either be both scalars or both vectors with the same number of elements.

To describe the default implementation of builtins, there are three strategies. The *direct split* strategy is the simplest one: vector types are directly scalarized, thus the scalar version of the builtin is invoked starting on each element of the vector value. The *recursive split* is a more efficient strategy for exploiting native vector types: the vector types are split in sub-vector halving their sizes, and applying the same builtin on each sub-vector value. The *template* strategy is used to generate a builtin function starting from a template of the body. Symbolic names are defined as template arguments and lowered exploiting the C preprocessor. The recursive and direct split are tailored to handle the overload of builtins for vector types, while the template strategy is useful whenever the implementation of a given builtin can be expressed as an instantiation of a template, e.g. when a pointer argument can have different address spaces.

To allow for more flexibility we provided a way to add useful custom code to define common data structures and/or including external files. This approach provides a common default implementation for most of the builtins for all devices. For each device the implementation of any builtin can be overridden in order to provide a more optimized one, e.g., a faster implementation using vector extensions such as SSE and AVX on x86 targets.

Such builtins are pre-compiled to LLVM-IR for each device, and provided to the runtime in form of a bitcode library. The runtime loads this library and links it to the LLVM module containing the kernels, before the actual conversion to machine code, to allow a more aggressive optimization of kernels through inlining of small builtins.

2.3 PULP Platform

The increasing performance requirements in computing intensive and mobile devices raise the need for powerful, but also ultra-low power (ULP) processors and computing architectures. Since power scales quadratically with the supply voltage, it is more energy efficient to operate a digital circuit near the threshold voltage of transistors [75]. The reduced speed can be compensated with multiple processors working in parallel at lower voltages [11]. To maximize energy efficiency, single processors

can be turned off depending on the current workload, while resources like memories and caches can be shared among the cores. To form an energy-efficient multi-core cluster capable of processing computation intensive applications, the processing element (PE) needs to fulfill certain requirements like having a small area footprint, being energy efficient and having a shallow pipeline allowing fast inter-core communication and data exchange to allow fine-grained data and task-level parallelism. Moreover, great attention must be paid to integration of the cores to avoid the creation of a bottleneck in the system due to the instruction and data interface. It is not enough to just use the most energy efficient 32 bit processor on the market, the ARM Cortex-M0+, to construct an energy efficient multi-core cluster, but point out the requirement of careful optimization for an operation in a tightly-coupled cluster. First, the ARM RTL code is not publicly available which makes an efficient cluster integration unfeasible. And second, while its energy efficiency of only 9.39 µW/MHz [7] would suite very well for ULP-operation, it is not meant to be used for computation intensive or parallel applications. Other architectures, like the ARM Cortex-M4 are more powerful in this domain but also show $3 - 4\times$ higher requirements in area and power consumption [7]. This increase in area and power consumption comes from moving to a deeper pipeline and from enriching the instruction set with DSP like features and more advanced ALU operations. In order to keep the area and power figures under control a careful evaluation of additional features and instructions must be done, in particular in ultra-low power contexts.

PULP platform represents an effort to design a many-core platform responding to the demands of heavily-constrained embedded applications. The choice for the base core is the OpenRISC architecture which is a) open source and suitable to be optimized to work in a multi-core environment, and b) shows a low area footprint with only 35.5 kGE which allows the core to achieve an energy efficiency of 25.8 µW/MHz in 65 nm technology, which is competitive with an ARM Cortex-M4 with an energy efficiency of 32.8 µW/MHz at similar area costs in a 90 nm technology [7].

While the computational efficiency of the OpenRISC architecture has been improved in terms of instructions per cycle (IPC) [57], it is missing important DSP-like features which allow a Cortex M4 to execute a program in less cycles due to its enriched instruction set. Enhancing an instruction set with very specific instructions is the key to increase performance in application-specific computing [9].

The PULP platform has been designed by group leaded by Prof. Luca Benini at Università di Bologna and ETH Zurich. The evolution of such

platform is on-going. Up to now three version of PULP have been realized. A collaboration between the Formal Languages & Compilers group of Politecnico di Milano and the group of Prof. Benini was established at the beginning of 2014 to work together for the development of PULP platform on both software and hardware aspects. The main contributions of our group are (1) the development and maintenance of a production quality compilation toolchain based on LLVM and CLANG, and (2) active partecipation in the definition of the ISA extensions and the implementation of the corresponding compiler support.The results of this collaboration has been published in [39].

2.3.1 Overview of the Platform

PULP platform is based on a low-power, flexible computing fabric that is able to provide significant performance when needed and remain in a very low-consumption state otherwise. To achieve these goals, PULP features *clusters* of simple PEs that can be used to exploit both coarse- and fine-grain data- or task-level parallelism. At the same time, voltage and frequency scaling can be controlled at a fine granularity to achieve high energy efficiency when the performance constraints are more relaxed or when the power budget is tighter. Figure 2.4 shows the PULP cluster architecture. In its current configuration it consists of four PEs, a shared 4-way associative I$ with four cache banks of 1 KB each and a L0 buffer of 128 bits per core holding the most recent cache-line to reduce access contentions at the cache banks [56]. The refill port of the shared I$ connects to the system bus together with an L2 memory, and several peripherals (not shown). The PEs have a multi-banked tightly coupled data memory (TCDM) acting as a shared scratchpad memory, instead of private data caches to avoid memory coherency overhead. TCDM is further divided into 16x4 KB SRAM banks, and 16x0.5 KB standard cell based memory (SCM) banks allowing the cluster to operate at ultra low voltages to achieve maximum energy efficiency [63]. Intra-cluster communication is based on a high bandwidth, low-latency interconnect, implementing a word-level interleaving scheme to reduce the access contention to TCDM banks. A lightweight, low-programming-latency, multi-channel DMA enables fast and flexible communication with other clusters, the L2 memory and external peripherals. In the following we focus on a single core and its integration in the cluster.

The OpenCores community [71] developed the OpenRISC architecture, an open-source processor using a GCC-based toolchain. A RISC architecture is well-suited to be integrated in a tightly-coupled multi-core cluster because of its low area footprint and the low pipeline depth

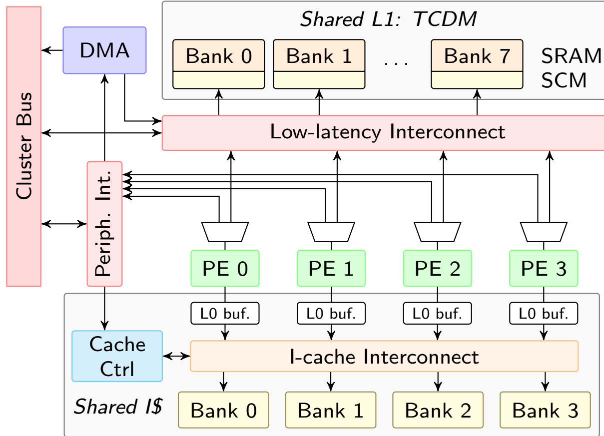


Figure 2.4: PULP cluster featuring four OpenRISC processor cores, a shared instruction cache, eight TCDM banks utilized as L1 memory and a DMA for fast, concurrent data movements.

allowing to interact with other processors in a single cycle. Or10n is a complete redesign of the micro-architecture in order to balance pipeline stages, and increase IPC [57]. The redesigned core is divided into four pipeline stages, *instruction fetch (IF)*, *instruction decode (ID)*, *execute (EX)*, and *write back (WB)* and achieves near-optimal IPC values of 1. All operations can be completed in a single cycle except for multiplications which are pipelined once, and can lead to stalls if the result is used in the subsequent cycle. While the core was being attached to an instruction and data memory [57], it has now been integrated in a PULP-cluster by connecting it to an I$ and a low-latency interconnect. Implemented in the cluster, the Or10n core utilizes 35.5 kGE. In this work we focus on increasing the efficiency of the generated machine code by introducing zero-overhead hardware loops, extended addressing modes, a more efficient multiplier architecture, and vector ALU operations. As we will show, getting rid of control code in small loops can lead to speedups of up to 2×. Extended addressing modes allow to get rid of instructions to maintain counters and addresses, by storing the updated memory address back to the latch-based register file. Ultimately, a vector unit that allows to concurrently process four 8-bit or two 16-bit values has been implemented in order to increase the throughput of the core. However, such instruction extensions are only useful if the compiler is able to produce such instructions. Therefore, we have modified the backend of the LLVM compiler to automatically generate code for the proposed ISA extensions and used it to evaluate the costs and benefits of the introduced ISA-extensions.

2.3.2 Instruction-Set Extensions for OpenRISC

Zero-overhad Hardware Loops

Zero-overhead hardware loops are a common feature in many processors, especially DSPs. Basically, a hardware loop is an implementation of a countable loop that avoids the need to explicitly test the loop counter and perform the branch. This is achieved by providing the hardware with information about the trip count and the beginning and end address of a loop, which are then used by specialized hardware in the computation of the next program counter (PC). The impact of hardware loops can be amplified by the presence of a loop buffer, i.e. a specialized cache holding the loop instructions, which removes any fetch delay [37]. In Or10N, we evaluated up to four nested hardware loops through the instructions shown in Table 2.1. Each hardware loop has associated 3 special purpose register: `HWLP_START` and `HWLP_END` for the start and end address of the loop, and `HWLP_COUNT` for the loop counter.

Table 2.1: Zero-overhead hardware loop operations. Each hardware loop is identified with an ID: `L0-L3`. The notation `HWLP_START[J]` is used to refer to the `HWLP_START` register of the Jth hardware loop.

Instruction format and Opcode	Semantics
`lp.start J, S (eg. lp.start L0, 10)` `000010 000 JJ SSSSSSSSSSSSSSSSSSSSS`	`HWLP_START[J]=sext(S*4)+PC`
`lp.end J, S (eg. lp.end L0, -8)` `000010 001 JJ EEEEEEEEEEEEEEEEEEEEE`	`HWLP_END[J] =sext(E*4)+PC`
`lp.counti J, C (eg. lp.counti L0, 8)` `000010 010 JJ CCCCCCCCCCCCCCCCCCCCC`	`HWLP_COUNT[J]=zext(C)`
`lp.count J, rA (eg. lp.count L0, r5)` `000010 011 JJ AAAAA----------------`	`HWLP_COUNT[J]=[rA]`

`lp.setupi J, E, C (eg. lp.setupi L0, 4, 8)`
`000010 100 JJ CCCCCCCCCCCCCEEEEEEEE`
	`HWLP_START[J]=PC+4`
	`HWLP_END[J] =zext(E*4)+PC`
	`HWLP_COUNT[J]=zext(C)`

`lp.setup J, E, rA (eg. lp.setup L0, 8, r5)`
`000010 101 JJ AAAAAEEEEEEEEEEEEEEEE`
	`HWLP_START[J]=PC+4`
	`HWLP_END[J] =zext(E*4)+PC`
	`HWLP_COUNT[J]=[rA]`

Hardware loop setup can be performed either explicitly, by initializing each SPRs using the *lp.start*, *lp.end* and *lp.count* (or *lp.counti*) instructions, or initializing them in a single instruction using *lp.setup* (or *lp.setupi*).

Hardware loops are implemented in the micro-architectural level with only two additional blocks – a controller, and a set of registers to store the hardware loop variables. The controller is a purely combinational block which checks if the current PC matches one of the end addresses, and if the counter is greater than 1. In case both checks are true, the hwlp-controller informs the main controller to set the next PC to the corresponding start address of the loop. If two or more end points are equal, the controller gives priority to the lowest hwlp-ID (i.e., `L0` has the highest priority). Since the performance gain is maximized when the loop body is small, it is not beneficial to support a lot of register sets. Therefore, we introduced two register sets in order to allow two nested loops. The support of additional hardware loops would bring marginal performance improvements at a non-negligible cost in terms of area (≈1.5 kGE per register set).

Extended Addressing Modes

Along with *Zero-overhead hardware loops*, we evaluated the performance and area impact of extended addressing modes. In the basic Or10n implementation only one type of load and store (*ld/st*) was available, in which the effective address was computed by adding a base address stored in a register and an offset encoded as an immediate value. The Or10n core was extended by implementing *ld/st* with both base and offset in register and *ld/st* with pre- and post-increment with both immediate and register offset.

Supporting all the variants of those instructions requires the addition of 65 new opcodes. To avoid saturation of available opcodes in the OpenRISC ISA, the new instructions are encoded using only 4 main opcodes and a sub-opcode field, which imposes a limitation on the immediate size from 16 to 11 bits. Out of those 65 new instructions, 3 instructions (1 for word, 1 for half word, 1 for byte transfer) are dedicated to each type of store, 6 instructions, coding different data sizes and sign extension, for each type of load with overall 5 new types of *ld/st*. These instructions are encoded reusing as much as possible the encoding scheme of the OpenRISC, keeping the source and destination registers at the same positions as shown in Table 2.2.

In Or10n the effective address is calculated in the ALU and then used to access the memory during regular *ld/st* and pre-increment operations. In case of a post-increment the address generation is bypassed and the memory is addressed with the base address. Loads with pre- or post-increment need to write two registers at the same time (the data read from memory and the incremented address pointer) and this required

Table 2.2: Load/store addressing modes. Register-register addressing mode is expressed with the notation `rX(rY)`. Auto-incrementing addressing modes are identified with a `!` before (preincrement) or after (postincrement) the base address. MMMMM in the sub-opcode is used to indicate bits that are dedicated to differentiate each type of load/store.

Old Instruction format	Opcode
`l.s[bhw] I(rA), rB` `l.l[bhw][zs] rD, I(rA)`	`MMMMMM IIIII AAAAA BBBBB IIIIIIIIIII` `MMMMMM DDDDD AAAAA IIIII IIIIIIIIIII`
New Instruction format	**Opcode**
`l.s[bhw] rD(rA), rB` `l.l[bhw][zs] rD, rB(rA)`	`010101 DDDDD AAAAA BBBBB -----0 1MMMM` `010111 DDDDD AAAAA BBBBB -----0 1MMMM`
`l.s[bhw] I(rA!), rB` `l.s[bhw] rD(rA!), rB`	`010100 IIIII AAAAA BBBBB IIIIII 0MMMM` `010100 DDDDD AAAAA BBBBB -----1 1MMMM`
`l.l[bhw][zs] rD, I(rA!)` `l.l[bhw][zs] rD, rB(rA!)`	`010110 DDDDD AAAAA IIIII IIIIII 0MMMM` `010110 DDDDD AAAAA BBBBB -----1 1MMMM`
`l.s[bhw] I(!rA), rB` `l.s[bhw] rD(!rA), rB`	`010101 IIIII AAAAA BBBBB IIIIII 0MMMM` `010101 DDDDD AAAAA BBBBB -----1 1MMMM`
`l.l[bhw][zs] rD, I(!rA)` `l.l[bhw][zs] rD, rB(!rA)`	`010111 DDDDD AAAAA IIIII IIIIII 0MMMM` `010111 DDDDD AAAAA BBBBB -----1 1MMMM`

the addition of an extra write port to the register file. Due to the non criticality of this path, storing the incremented address in the write back stage can be avoided and the register file can be written directly in the current cycle. To support stores with the offset or increment in a register an extra read port was required, in fact, 3 different values have to be fetched from the register file at the same time (base address, offset and data to write). The additional write and read port costs less than 1 kGE due to its non critical timing and its latch based implementation.

ALU Vector Support

Aiming for a more efficient processing of 8, and 16 bit data leads to the introduction of a vectorized ALU where the datapath is segmented into two, or four parts and allows to compute up to four bytes in parallel leading to a speedup of up to a factor of four. Such operations are also known as subword parallelism [55], packed-SIMD or micro-SIMD [77] instructions. However, the 32 bit operations remain the norm, meaning that extending a processor with such vector capabilities is only promising if the area and power overhead can be kept at a minimum. In order to

extend our 32 bit OpenRISC ISA with subword parallelism, we have added two new opcodes for vector operations, one for ALU operations and one for vectorial comparisons. Each operation is available in three different vector modes for halfword (h) and byte (b) operations:

- `lv.inst.{h,b}      rD, rA, rB`,
- `lv.inst.{h,b}.sc  rD, rA, rB`,
- `lv.inst.{h,b}.sci rD, rA, I`,

where the first is a register to register operation, and the other two are vector operations with a register `rA` and a scalar replication of the register `rB`, or an immediate. The operations based on an adder and shifter have been realized by splitting the data path in four segments. The full 32 bit result is computed by chaining the four adder results with the carry. Multiplications and MACs on the other hand, are complicated because of the additional muxing to support four concurrent multiplications.

The OpenRISC ISA supports full 64 bit result multiplications and MAC operations, which cannot be implemented within a single cycle without impacting the maximum frequency. In a previous implementation [57], the multiplication was realized with a two cycle multiplier and MAC operations were based on a special purpose accumulator which is accessible through special instructions. Given the fact that the full 64 bit result is often not required, and that the compiler is not always able to group instructions in a way such that no stalls occur, we have decided to simplify the multiplication by only generating the 32-bit results. This allows to support vectorial multiplications with subword selection [55] in a single cycle. Notice, that the full 64-bit product can still be generated by the addition of four partial products which can be generated in sequence. While in the original implementation three cycles were required to generate a 64-bit result, with subword selection it is possible to create it with 10 instructions. Reducing the multiplications to 32 bit makes the 64 bit accumulation register useless. Instead of the large accumulator, a normal register can be used as accumulation register which leads to two major advantages: a) it is possible to concurrently maintain multiple accumulation registers and b) the additional delay of moving data back and forth from the accumulation special register to a general purpose register vanishes since the accumulator is placed in the GPR in the first place.

Implemented in the micro-architecture of Or10n, the vectorial ALU and fused MULT/MAC unit increase the core area by 6 kGE which accounts for 13 % of the core but less than 1 % of the complete cluster. The

additional read port of the register file is shared with the advanced addressing modes and costs less than 1 kGE. In 2.3.4 we show the benefit of the new multiplier with and without vectorial support.

Other Extensions

Along with the presented instruction extensions, several small ALU extensions such as *min, max, avg, abs* have been implemented. The area penalty of these extensions is less than 1 %. These instructions are seldom used, however they can speed up significantly operations, such as normalization.

2.3.3 Implementation in the ULP-Cluster

To keep the hardware complexity at a minimum, the introduced ISA-extensions have been implemented to a large extent with existing resources. The combined costs of all extensions, including all vectorial operations, are 9 kGE which is 25 % of the core area. As pointed out, a good core architecture is not enough for an efficient integration in a multi-core cluster which is why we optimized the data and instruction interface without increasing the critical path. In the following we highlight three cluster-specific integration details which allow the processor to work more efficiently in a multi-core cluster.

Reducing Cache Accesses with a L0 Buffer

The amount of energy consumed by an instruction cache is not to be underestimated. In fact, a core with a very high IPC is accessing the instruction cache every cycle. Keeping this in mind and the fact that hardware loops are most effective if the loop body is small and contains no branches, adding a small L0 buffer between the instruction cache and the fetch interface (shown in Figure 2.4) is a promising approach to lower power consumption by reducing the cache accesses. We have chosen a 128 bit wide L0 buffer, which is capable of holding the most recent cache-line. Hence, this architecture benefits if the compiler is capable of placing hardware loops aligned with 128 bits. In particular, when paired with a shared instruction cache, this approach is very effective since it significantly reduces access contention at the cache banks [56] and allows a loop to be fetched only once if the size does not exceed four instructions. Even though the buffer can only hold 4 instructions, this is not a restriction on the loop size. Larger loops would still benefit from the L0 buffer because only every fourth request has to be forwarded to the cache controller.

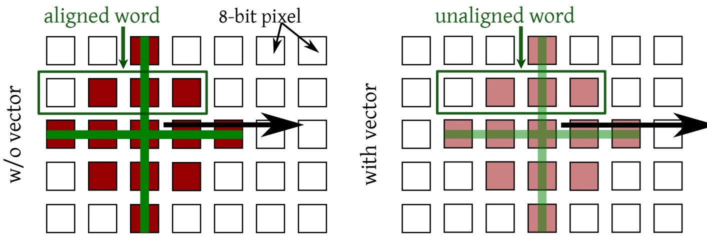


Figure 2.5: Example showing the benefit of unaligned access when computing a stencil over an image with 8 bit pixels.

Unaligned Memory Access

Data structures are not always word aligned, and even if the compiler is aware of a vector unit, it is not always possible to align data. For example, in a simple 2D-filter on 8 bit data types, as depicted in Figure 2.5, it is not possible to read aligned data in every cycle, thus leading to unnecessary masking and shifting or resorting to byte-wise loads. To support unaligned memory accesses in the tightly-coupled cluster, we have two options: 1) extend the data interface to 64 bit — 2) implement the unaligned memory access in 2 cycles.

Since the critical path in the cluster is already on the return path of the data memory, and that reading 64 bit of data means doubling the size of the interconnect and also the number of banks, the former option is not the most promising. In fact, by doubling the size of the interconnect, the depth increases due to its logarithmic tree. Moreover, since the delay of the interconnect depends on the depth, it would impact the length of the cluster's critical path, leading to a lower energy efficiency. For this reason we did not further evaluate this solution. The second approach enables unaligned memory access with two subsequent memory requests, which are then merged into a single word by the load store unit. This hardware implementation does not add to the critical path and brings no additional hardware complexity to the core. Since the TCDM serves requests within a single cycle, it is possible to read unaligned data in only two cycles. On the other hand, to perform the same load operations, an architecture without support for unaligned data requests would require at least five instructions. Two cycles to read the two data items, two cycles to shift data and one cycle to combine them in a single register.

Our implementation of unaligned memory requests does allow to use the vector unit more often[3] and is much more efficient in code size, and

[3]Without unaligned load support the vectorizer can not vectorize most of the loop due to the lack of proper guarantee on the alignment. Moreover the cost of software

latency than a software only support. Compared to a single cycle load operation, our implementation does not require significant additional hardware resources and is therefore very well suited for the multi-core cluster.

Branch Prediction to Balance the Paths to I$

Branching was initially performed by using the forwarded flag from the ALU, thus allowing to branch without stalling the pipeline or having to predict and probably revert the pipeline. This was particularly important when processing loops. With hardware loop support this argument falls apart and stall free branching is no longer required for efficient loop handling. Moreover, since the L0 buffer delays the instruction request path, branch prediction becomes a requirement to prevent slowing down the system. Specially at lower voltages, where the cluster becomes extremely energy efficient, this path becomes critical. Synthesizing the cluster without branch prediction and an L0 buffer of 128 bit in a 28 nm process leads to a frequency degradation of 13 % at 0.6 V. Branch prediction would split this path, but also slightly increase the runtime due to mispredictions. Given the fact that we want to operate at ultra-low power, and that the misprediction penalty is only one cycle in our flat pipeline, we have chosen the most simple branch predictor, which is to always take backward branches, never take forward branches.

The branch prediction removes the critical path to the I$, while increasing the number of cycles in our benchmarks by only 1.32 % on average, with a maximum of 3.4 %. In combination with hardware loops, this number decreases to only 0.7 % additional cycles. Hence, we conclude that even a very simple branch predictor, if paired with hardware loops, does only increase the number of cycles by 0.7 % while allowing the core to be clocked 13 % higher at low voltages.

2.3.4 Performance, Area, And Power Results

In the following we are discussing the performance, area, and power impact of the introduced ISA-extensions. The execution time in num-

unaligned memory operations overcomes the benefit of vectorization itself as our vector unit lacks efficient vector shuffle instructions. For simple loops where only one array is processed it would be possible to peel some iterations of the loop in order to align the pointer and vectorize the remaining iterations. Alternatively the vectorizer should generate different versions of the loop dominated by runtime checks to ensure the required alignment of pointers. In the context of this work it has been preferred to extend the core implementing unaligned memory operations since their cost is almost negligible.

ber of cycles has been measured in RTL-simulations, while the area increase was determined with Synopsys Design Compiler in topographical mode, and the power estimations were performed on a back-annotated placed&routed netlist at nominal voltage of 1.2 V and a frequency of 50 MHz. The complete design, including a final tapeout of the PULP chip, was done in UMC 65 nm technology. Unless otherwise specified, all power and area numbers are related to this technology.

Execution Time

Each proposed instruction set extension has been analyzed individually by enabling it in the compiler through dedicated flags. Specifically, we compared the basic ISA with 4 different configurations, which, respectively, enable 2 sets of hardware loops (H), pre- and post-increment addressing modes (I), the new single cycle multiplication and three operand MAC unit (M) and vector operations with unaligned access (V). Finally, we also performed a comparison with the ARM Cortex-M4 (Cortex-M4).

Figure 2.6 incrementally shows the benefit of each ISA-extension on our benchmark applications which range from basic matrix multiplications, based on 8, 16 and 32 bits, through convolutions, filters and cryptographic algorithms. In particular matrix multiplications are presented in two forms: the classical implementation with row-by-column products, and an optimized version with row-by-row products preceded by the transposition of the second matrix, being a good candidate for auto-vectorization. All benchmarks are executed on a single core. Hardware loops and automatic addressing updates bring speedups up to 1.75×. The MAC unit has a good impact on computational intensive benchmarks like convolution and matrix multiplications and allows to process those benchmarks up to 2.25× faster. The use of the vector unit can bring an additional boost when it is applicable. E.g., on the matrix multiplication on 8 bit data, our vector optimization achieves a 5× speedup with respect to the original ISA. Besides the impact of the ISA extensions, this figure results from the ability to vectorize row-by-row products. Similarly, on the 16 bit optimized matrix multiplication we achieve an overall 3.5× speedup. On the classical row-by-column implementation on 8 bit data, data access on the column cannot be efficiently vectorized as there is no hardware support for either strided or gather/scatter load/store operations. On 16 bit data matrix multiplication the compiler does not apply vectorization as it is detected as not effective by the heuristics.

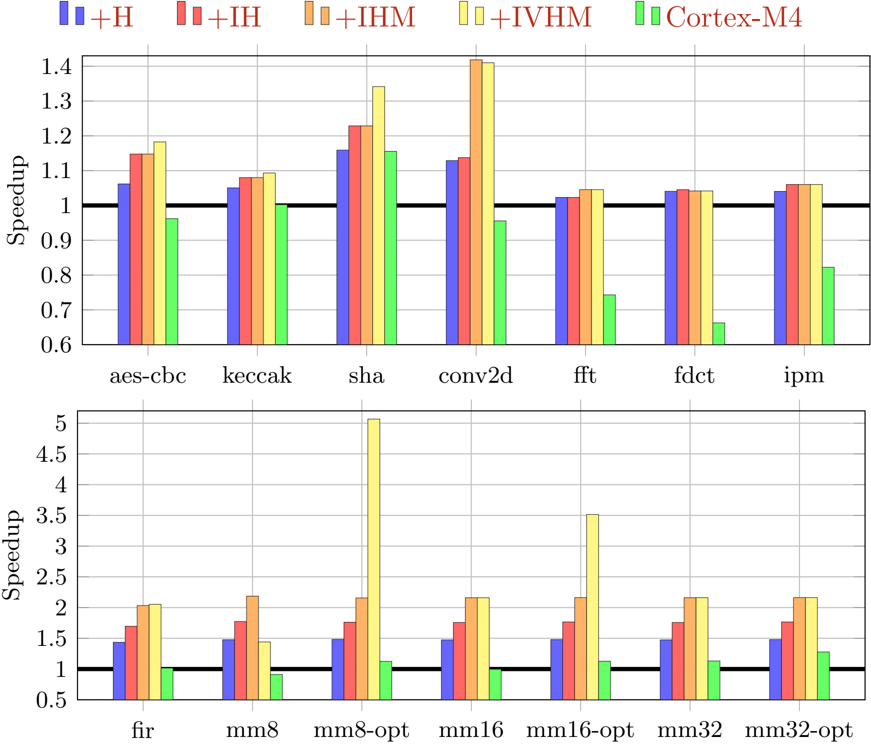


Figure 2.6: Comparison of the speedup with the introduced ISA-extensions versus the initial ISA. Number of cycles are measured in RTL simulations for the OR10N core and in case of the Cortex-M4 on the real hardware. All the results are normalized w.r.t. the number of cycles on the plain OpenRISC ISA.

Area and Power Efficiency

Figure 2.7 shows the increase in area and power with each additional feature. We can conclude that even though the area per core increases by 25 % to 44.5 kGE, the overhead at cluster level remains small with only 2.3 % of overhead due to the presence of interconnects, peripherals, and large amount of memory in the cluster.

The total power consumption of one core running at 50 MHz with all features enabled is 1.688 mW, which translates to an average energy efficiency of 33.8 µW/MHz. The core shows a higher power consumption during the execution of vector heavy code where an energy efficiency of 47.8 µW/MHz is achieved.

At the maximum frequency of 362 MHz, which is not affected by the ISA-extensions, the four-core cluster is capable of processing 1.4 GOPS at a power budget of 93 mW of which 55 % is consumed by the cores.

With respect to a single core implementation, the four-core cluster can process four times more operations, at a power increase of only 67 %. Thus, moving from a single core to a four-core cluster increases the power-efficiency by a factor of 2.4.

Core and Cluster Energy Efficiency

In Figure 2.8 the total amount of energy required to run each benchmark is shown for the original ISA, the extended ISA with and without vector unit, and a Cortex-M4. To calculate the numbers for the Cortex-M4, an energy efficiency of 16.7 µW/MHz is assumed, which is scaled down to 65 nm from a 90 nm technology [7].

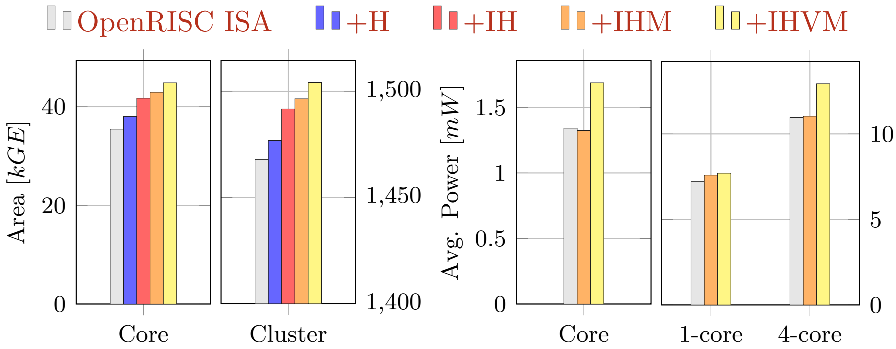


Figure 2.7: Area and power comparison with different ISA-extensions on core and cluster level. Power is shown on a cluster with 1 and 4 cores.

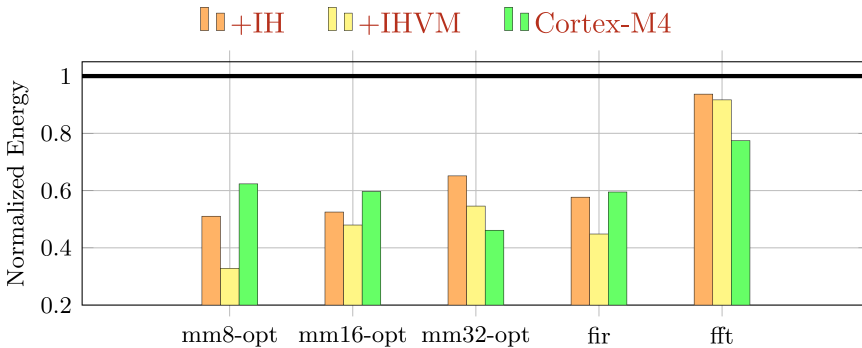


Figure 2.8: Energy efficiency of the core with the optimized ISA integrated in the multi-core cluster in UMC 65 nm. All the results are normalized w.r.t. the absorbed energy on the plain OpenRISC ISA.

In normal execution, where no vector code is used, the energy savings are 41.1 % on average. Due to the additional speedup of the vector unit, the savings increase to 59.6 % in the vector intensive matrix multiplications. When comparing the efficiency to an ARM Cortex-M4, we observe that the extended ISA is up to 48 % more energy efficient when processing vector code due to the vector instructions. If no vector code can be used, the performance of the proposed ISA shows no clear trend towards Cortex-M4 or OpenRISC.

Taking into account all cluster resources, a cluster consisting of the improved cores consumes 18 % more power with respect to a cluster with the initial micro-architecture. Taking the complete cluster power into account, the energy savings of the cluster based on the extended ISA with all features enabled ranges from 39 % to 66 %. On average the cluster is 47.8 % more energy efficient than the initial architecture.

2.4 Conclusion

In this chapter we have introduced OpenCRun, our OpenCL runtime implementation. The goal of OpenCRun is to support the execution of OpenCL kernels on a wide range of platforms, from high-end multi-core processors to low-power many-core accelerators such as PULP. Since the PULP platform is based on an Open Source Hardware approach, and employs the OpenRISC core as its processing element, we have participated in the definition of ISA extensions for the PULP implementation of OpenRISC. The close cooperation between compiler and hardware designers has led to major performance and energy efficiency improvements. The runtime and compiler described in this chapter serve has a baseline for our exploration of functional and performance portability of OpenCL kernels.

3 Cross-Platform Functionality and Performance for OpenCL

OpenCL applications may present tight constraints on work-group sizes either due to the algorithm design or to the chosen implementation strategy. This may prevent functional or performance portability across different platforms, with the current solution being a re-design of the implementation, optimized for the new platform. However, the required effort represents a hindrance for the exploitation of future heterogeneous platforms, where benchmark suites and applications should be ported in the near future. We aim at tackling the functional portability issue through applying work-item coalescing techniques to optimize the mapping of OpenCL work-items onto the processing elements of the underlying architecture. However, this may not be sufficient to achieve sound performance portability: to this end we show how additional target specific transformations can improve the performance with respect to the work-items coalescing baseline. We employ two case studies to show how the work-item coalescing transformations impact functional portability, together with providing an opportunity of automatically inserting the use of asynchronous copies on embedded many-core platforms endowed with such a feature.

3.1 Introduction

To obtain viable programming frameworks for different parallel architectures, the Khronos consortium proposed the Open Computing Language (OpenCL) programming model as an open standard [49]. OpenCL describes the computation of a data-parallel program in terms of sets of work-items, called work-groups, which are mapped on the underlying architecture usually by the OpenCL runtime, while providing an architecture-agnostic structure to the programmer. Thus, OpenCL helps in providing functional portability of parallel programs across different multicore platforms.

However, OpenCL applications may present tight constraints on the size of work-groups due to either the algorithm design or the chosen implementation strategy. If the target architecture does not provide

enough resources, the ability of OpenCL to provide functional code portability is disrupted, as an attempt to execute more work-items than those allowed by the underlying architecture will fail. Thus, the application developers are in charge of inserting code to verify that the kernels they are writing are actually executable on the available platforms, and include fallback code to deal with a possible lack of resources. This approach entails major development cost overheads, as multiple versions of the code need to be written, and significant boilerplate code is necessary for the introspection phase. Even when the aforementioned constraints are satisfied, the performance figures across different heterogeneous platforms may exhibit notable differences. Thus, the functional and performance portability of code across different platforms and different domains (from embedded to HPC) has grown into a major research problem [17, 4, 13].

A typical scenario where the aforementioned issues take place is when the following architectures are involved: embedded many-core platforms with explicitly managed memory (EMM) and platforms for general-purpose computing on graphics processing units (GPGPU). Embedded EMM architectures, such as P2012 [16], exploit independent processing elements grouped in clusters, whereas GPGPU architectures are built on streaming multiprocessors (SMs) with step-locked processing elements. On P2012, due to the small amount of tightly coupled dedicated memory bound to the cores, no context-switch mechanism is provided, thus limiting the maximum size of an OpenCL work-group to the number of cores in a cluster. By contrast, on GPGPUs a work-group is mapped on a single SM and a context-switch mechanism is triggered if the work-group size is bigger than the number of processing elements in the SM. Thus, an OpenCL application with a large work-group size tailored for a given GPGPU cannot be run on platforms like P2012.

Contributions

We propose an execution model for transparently running OpenCL applications (characterized by tight constraints on the work-group size) on EMM architectures lacking context-switch support. This goal is to be achieved without significant performance penalties, that is, maximizing the utilization of the processing elements. The gist of the proposed approach is to provide a transparent decoupling between the programmers view of an OpenCL work-group and the actual mapping of its work-items on the underlying processing elements. This can be realized via a combined action of both the OpenCL compiler and the OpenCL runtime platform support. The proposed strategies for mapping the work-groups

onto the underlying hardware ranges from coalescing all the work-items into a single work-group and mapping it on a single processing element, to grouping the work-items into a number of sets fitting the amount of processing elements. The memory requirements of the aforementioned approach are the ones of the un-coalesced kernel, multiplied (at most) by a factor linear in the amount of coalescing done. In particular, the highest possible multiplicative factor is given by the amount of processing elements when the first strategy is applied, while the second strategy does not significantly impact on the memory requirements. In addition to the coalescing approach, we also propose a technique for inserting automatically asynchronous copies in the code, so to provide a better exploitation of parallel computing platforms supporting them, with the resulting performance enhancement, while guaranteeing functional portability.

Organization of the Chapter

Section 3.2 describes the functional portability provided by OpenCL, and the architectural differences present in the computing platforms. Section 3.3 describes our mapping technique as well as the asynchronous copy insertion strategy. Section 3.4 reports experimental evidence gathered on two case studies. Section 3.5 provides an overview of related works, while Section 3.6 draws our conclusions.

3.2 Preliminaries on OpenCL and Background

The OpenCL language and programming model are designed to provide functional portability of parallel programs across different platforms. This is achieved providing the programmer with an abstract architectural model, which assumes that the underlying hardware will be executing parallel tasks (*work-items*), clustered in *work-groups*. Work-items within a work-group may communicate directly through a shared *local memory*. The parallel program is represented as an OpenCL *kernel*, which is the function executed by every work-item. The kernel is executed by a collection of work-groups that can be run in any order by the platform. The OpenCL barrier synchronization construct provide a memory fence for all the work-items of a work-group, while inter-work-group synchronization is not available as a primitive. To support a wide range of hardware platforms, OpenCL relies on exposing a large amount of information to be explicitly managed by the programmer. Thus, the OpenCL API includes introspection primitives able to return which is the maximum number of work-items per work-group that can be used

to perform the proper program management so that the implementation runs on any platform. This requirement towards the programmer may act as an hindering factor in the seamless functional portability of parallel programs – the main goal of OpenCL.

To provide a practical validation of this claim, we examined how frequent it is to encounter OpenCL code samples which do not include explicit work-group management. Table 3.1 shows an analysis of the work-group size constraint over four well known OpenCL benchmark suites. The column labeled as "Fixed" reports the number of programs (in the corresponding benchmark suite) where the work-group size is statically chosen. The column labeled as "CT/RT Param." shows the number of programs with work-group size parameters configurable either at compile-time or at runtime, while the column labeled as "Adaptive" reports the number of programs where the work-group size is actually computed from the device information retrieved by introspection primitives at runtime, as the best programming practices suggest. Only 10 programs out of the 63 analyzed allow for seamless functional portability with OpenCL, while the others mandate either a compile-time or run-time tuning, or a code modification altogether. This points to the fact that even experienced programmers working on a benchmark suite are likely to develop code without taking into account performance (or even functional) portability beyond the range of platforms immediately available. It is clear that the cost of coding an application in a portable way is much higher, not only because the developer needs to make an additional effort, but also because this effort is not immediately useful and cannot be easily tested outside of the current target platform. Indeed, it is well known that the performance of OpenCL programs strongly depend on the characteristics of the target platform, the applied compiler transformations, and the way the application is implemented [76]. In this work, we employ a source-destination platform pair to illustrate how it

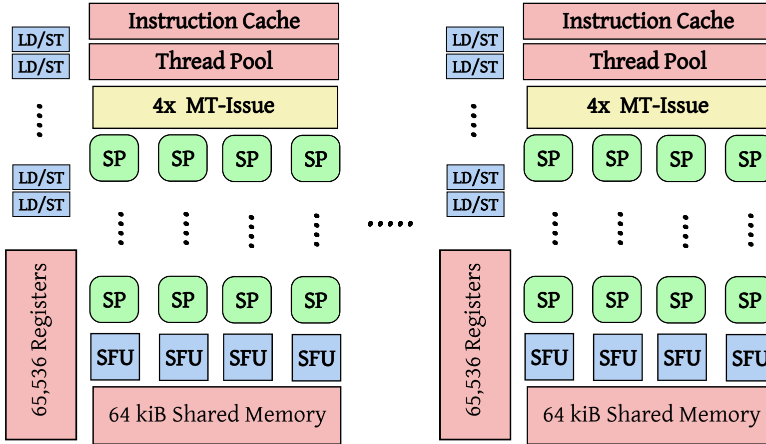
Table 3.1: Analysis of OpenCL benchmark suite programs according to the management of work-group size parameter

Benchmark Suite	Numb. of Programs	Work-group Size Management		
		Fixed	CT/RT Param.	Adaptive
Rodinia [21]	20	6	11	3
Parboil [81]	13	11	0	2
SHOC [27]	11	7	1	3
NVIDIA [69]	19	15	2	2

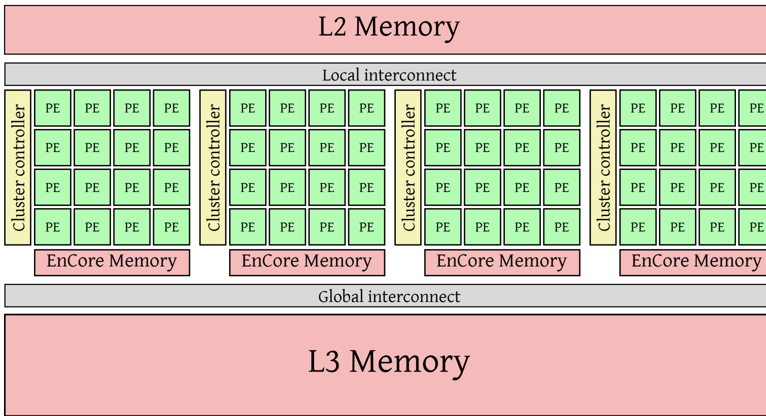
is possible to provide to the programmer seamless functional portability for OpenCL applications even when they do not manage explicitly work-group sizes. Moreover, we also propose the automated introduction of asynchronous memory copies to exploit DMA-based global memory access mechanisms, which benefit from double-buffering strategies.

In particular, in line with the stated goal of simplifying performance portability from current platforms to future HPC architectures based on low-power many-core accelerators, we chose as the source platform for the porting a typical General Purpose computing Graphics Processing Unit (GPGPU), while the target platform is the STM-P2012, a many-core accelerator with explicitly managed memory targeting the domain of embedded applications [16, 79]. A sketch of a representative of the GPU architecture, the *NVIDIA Kepler*, and of the STM-P2012 architecture is reported in Figure 3.1. The choice of a GPU as the source architecture for the porting is justified by the OpenCL programming model being strongly influenced by it, thus implying that most of the existing OpenCL code has been implemented as best fitting on such an architecture. In particular, the GPU architecture is characterized by a set of *streaming multiprocessors* (SMs), each composed of several processing elements called *streaming processors* (SPs). Processing elements in a SM are step-locked, so that they run the same instruction at the same time – thus leading to a low efficiency in the case of control flow divergence within the work-group. The tight resemblance of the architecture to the OpenCL programming model allows the work-groups to be mapped directly onto an SM, provided their size does not exceed the number of SP. An hardware context-switch mechanism allows work-groups with size larger than the number of processing elements in a SM to run correctly, by partitioning the work-items in batches called *warps* and serializing the execution of multiple warps.

Differently from the GPU, STM-P2012 is a clustered architecture connected through an asynchronous global network-on-chip. In P2012, the processing elements within a cluster are independent, and therefore able to support control flow divergence more efficiently. Memory availability is much more limited than in a GPGPU, with 256 KiB EnCore memory (a tightly coupled data memory) for each cluster and 1 MiB of shared memory for the overall fabric. Since no context-switch mechanism is provided by the hardware and runtime, a cluster can run only a single work-group at a time, with a number of work-items less or equal to the number of processing elements, thus providing a hard limitation on the work-group size.



(a) GPGPU Architecture: NVIDIA Kepler



(b) STM-P2012 Architecture

Figure 3.1: Structure of two multicore architectures with OpenCL support: the *NVIDIA Kepler* GPU and STM-P2012

3.3 Kernel Transformations

In this section we present our solution to the functional portability problem induced by a constrained work-group size and then we present an optimization useful for platforms like P2012 where DMA units are available. In the following, we will be adopting the compiler-transformation point-of-view; we thus recap the OpenCL concepts from this perspective. An OpenCL *kernel function* represents the body of a N-dimensional parallel loop-nest, where each iteration is named *work-item*. The iteration space of this loop-nest is the *global-space*. In OpenCL the work-items can be grouped: such a collection of work-items is named *work-group*.

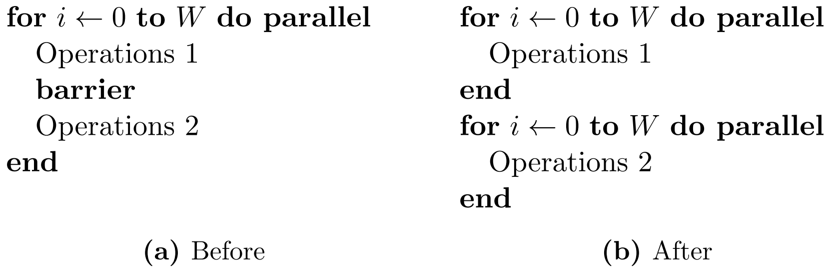
This allow us to interpret the kernel function as the body of a $2N$-dimensional parallel loop-nest: the outer N-dimensional loop iterates on the work-groups, while the inner N-dimensional loop iterates on the work-items in a work-group. We recall that memory barriers are effective only between work-items in the same work-group and that both parallel loop-nest structures are *collapsible*, i.e., it is possible to map each N-dimensional iteration space into an equivalent mono-dimensional one. The transformation targets a single work-group, so from here onwards we ignore the outer parallel loop, as our focus is the *inner* one.

3.3.1 Work-item Coalescing

The key idea is to re-structure the iterations in a work-group map them on the available processing elements. The mapping strategies are implemented through a compiler transformation named *work-items coalescing* with cooperation of the OpenCL device runtime. The transformation must ensure that each processing element will execute the iterations assigned to it preserving the semantic of the original OpenCL program. This requires handling both synchronization barriers and divergent variables (i.e., variables depending on the loop induction variable). In particular, a trivial serialization of the iterations would fail to preserve the semantics, since the execution must reach the synchronization barrier for all iterations before progressing after it for any of them. The mapping of a set of iterations (i.e., work-items) to the same processing element must ensure that all of them execute the kernel code preceding a synchronization barrier prior to execute the code beyond it. To this end, the proposed work-item coalescing transformation is defined as a sequence of two steps: a *kernel fission* pass, which splits the kernel in multiple fragments using the barriers as borders, and a *work-items remapping* pass, which is in charge of re-structuring the kernel fragments to compose new, semantically equivalent, work-items.

Kernel Fission

The first step of the transformation analyzes the kernel code to split its *control-flow graph* on the barrier operations, identifying the kernel *parallel region*s. A parallel region typically has a single entry point and multiple exit points. Each exit point transfers the execution flow either to another region or to the kernel end. The step splits the original parallel loop into multiple ones, replacing the explicit synchronization barriers in the original loop body with the implicit ones resulting from the end of each parallel region.



(a) Before

(b) After

Figure 3.2: Example of a kernel fission transformation

Figures 3.2a and 3.2b show how fission is performed. The induction variables of the new parallel loops are simple clones of the original one. In addition, the live-out variables of each iteration must be preserved to keep the original semantic. To this end, a divergence analysis is applied to move outside the newly created loops the variables which do not depend on the induction variable. After fission, the iterations within each parallel region can be executed in any possible order as their code does no longer include synchronization barriers.

This steps recall a classical loop transformation known as *loop fission*. A typical scenario for loop fission is a *sequential* loop where two arrays are processed *independently*. The goal of the transformation is to split such processing in two different loops to improve both spatial and temporal locality within the loop body to take advantage of the cache hierarchy avoiding the trashing effects due to sequential accesses to different data structures that may be allocated far apart in memory. Data dependence analysis is a requirement for such transformation to prove its legality. Kernel fission is conceptually a specialization of loop fission applied to parallel loops. In this particular case data dependence analysis is not required, since iterations are assumed to be indepedent, and the parallel loop is split on synchronization barriers.

Work-items Remapping

The *work-items remapping* step coalesces the iterations of each parallel loop resulting from *kernel fission* pass on the processing elements. The consecutive iterations of the parallel loops are serialized to reduce the number of parallel iterations. The reduction in number of parallel iterations allows to pair them with the available processing elements on the target platform. This mapping will also aim at assigning the same amount of computation to each one of the processing elements, in order to balance the workload and reduce the synchronization time. The *work-items remapping* transformation takes two runtime parame-

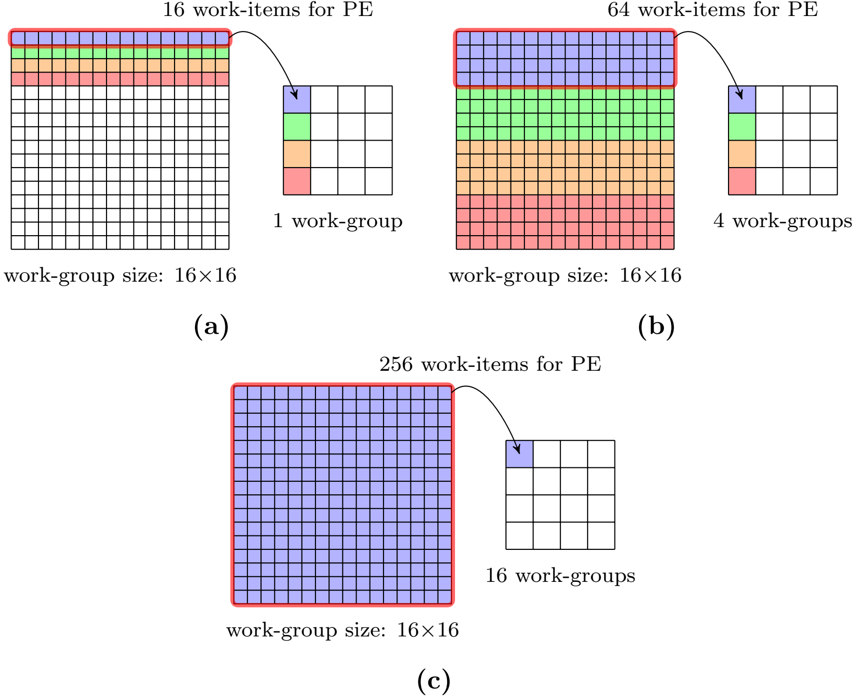


Figure 3.3: Work-items mapping strategies on a P2012 cluster. Two different strategies based on the partial mappings of the work-items are shown in (a) and (b), while in (c) a full work-items coalescing strategy is shown

ters: CF which is the *coalescing factor*, and NW_r, the number of iterations executed by the r-th processing element. The input of the transformation is a parallel loop with induction variable $i \in [0, W-1]$ with W the size of the source work-group. The result of the transformation is a loop-nest, with the external loop being a parallel one with induction variable $i' \in [0, K-1]$, where K is the maximum number of processing elements available to execute the kernel. The inner loop of the nest is a sequential one with induction variable $j' \in [0, NW_{i'}-1]$. Note that all the uses of the induction variable i contained in the loop body can be performed reconstructing the correct value of i as $i = CF \cdot i' + j'$. After the transformation the external parallel loop has a number of iterations that is at most the number of processing elements that the runtime decided to allocate for the current work-group, thus all the iterations of the resulting parallel loop can be executed in parallel, effectively emulating a number of work-items greater than the number of processing elements.

The parameters CF and NW_r are computed by the runtime when an *NDRange command* is put in execution from the device queue. An

NDRange command identifies the kernel that must be executed, its arguments, the global size, and the local size $L=(L_0, L_1, \ldots, L_{N-1})$ expressed as an array of values containing the the sizes of work-groups. Given the work-group size $W=\prod_{i=0}^{N-1} L_i$ and K the maximum number of processing elements to execute the iterations, with $PF=\lfloor W/CF \rfloor$ being the number of processing elements that will run exactly CF iterations each, CF and NW_r are computed as stated in Equation 3.1 and Equation 3.2.

Note that it is possible that some processing elements processing elements do not get any iterations assigned assigned to them, as NW_r may be zero if the work-group size W is not divisible by K.

$$CF = \left\lceil \frac{W}{K} \right\rceil \tag{3.1}$$

$$NW_r = \begin{cases} CF & r \in [0, PF-1] \\ W \bmod CF & r = PF \\ 0 & r \in [PF+1, K-1] \end{cases} \tag{3.2}$$

Figures 3.3a, 3.3b and 3.3c show three viable mapping strategies for a 256 work-item wide work-group, to be mapped to a 16 processing element underlying architecture. In each figure, we represent on the left the original work-group as defined in the application code, with a fixed size of 16×16 elements. On the right we depict the target architecture element composed of 16 processing elements, as in the case of a P2012 cluster. Through the coloring, the figures show how each mapping is performed. In Figure 3.3a, each group of 16 work-items is mapped to a single processing element ($CF = 16$), thus the work-group is executed entirely by the cluster . In Figure 3.3b the work-group is split over four processing elements thus the cluster can execute up to 4 work-groups in parallel, while in Figure 3.3c the entire work-group is mapped on a single processing element ($CF = 256$) thus the cluster can execute up to 16 work-groups in parallel, since all the work-items of the original work-group are coalesced to be executed by a single architectural processing element.

3.3.2 Memory Transfers Optimization

The *work-items coalescing* is a solution to the functional portability problem, but may not be sufficient to provide performance portability. This fact emerges especially while moving an application between significantly different target architectures, where some relevant features of

```
kernel myKernel(global uint *in, global uint *out,
                uint seq_len) {
  local uint buf_in[BUF_SIZE];
  local uint buf_out[BUF_SIZE];
  uint l = get_local_id(0);
  uint b = get_block_id(0);
  global uint *base_in = in + b * seq_len;
  global uint *base_out = out + b * seq_len;

  for (uint i = 0; i < seq_len; i += BUF_SIZE) {
    // Collaborative global to local transfer
    buf_in[l] = base_in[i + l];
    barrier(CLK_LOCAL_MEM_FENCE);

    // Computation
    compute(buf_out, buf_in);
    barrier(CLK_LOCAL_MEM_FENCE);

    // Collaborative local to global transfer
    base_out[i + l] = buf_out[l];
    barrier(CLK_LOCAL_MEM_FENCE);
  }
}
```

Figure 3.4: Structure of kernel for profitable memory transfer optimization.

the source architecture are not present in the target one, and no code is present to explicitly exploit them. In our case study, P2012 has DMA units that are employed by the native OpenCL runtime to implement asynchronous copies between local memory and global memory. On P2012 the use of *double-buffering* with DMA transfers is welcome as it allows to overlap in time memory transfers and computations. Thus it is useful to introduce this pattern automatically when is potentialy profitable from a performance perspective.

In GPGPU specific OpenCL kernels, it is common to find parallel regions split by barrier operations, where a region pre-fetches data in a local memory buffer, while a subsequent region consumes the data producing the results, and, optionally, a further region writing them back to a global memory buffer. This strategy is used due to the smaller overhead of accessing local memory variables than global memory ones. Usually, the *pre-fetch* actions are performed by all the work-items in a work-group: each work-item copies a portion of the data that will be processed later on.

We propose a target specific transformation for P2012 to identify this pattern, and, whenever profitable, to transform it introducing double

buffering and asynchronous copies. This transformation is generally profitable when this pattern resides inside a loop, as shown in Figure 3.4: in such case the pattern gets repeatedly executed, thus allowing to pipeline the execution introducing the double-buffering and exploiting DMA transfer to overlap memory transfers with the computation. Given the parallel regions containing memory transfers, the transformation starts by finding the `read` accesses from the global memory and the related `store` accesses in the local memory.

The main observation is that the source and destination addresses depend on the induction variable of the parallel loop, thus knowing the loop iteration space it is possible to generate an equivalent combination of asynchronous copies.

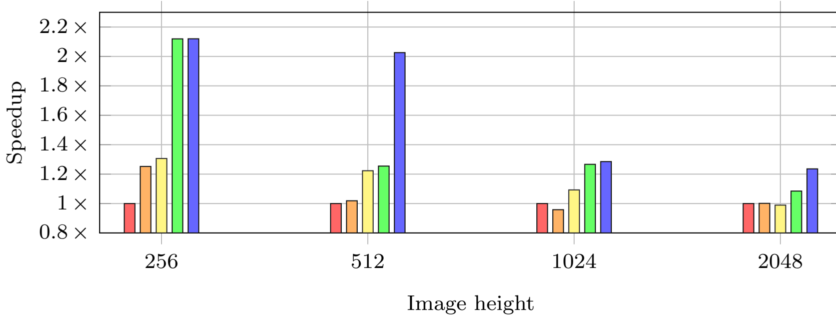
Figure 3.5 shows the resulting kernel after the transformation. Note that the introduction of asynchronous copies may increase the performance at the cost of requiring a larger amount of local memory: it is thus crucial to ascertain that the transformed kernel will not exceed the architectural memory constraints. Furthermore, additional constraints may be imposed by the number of available DMA request slots, which in turn bound the maximum number of request which can be served in parallel.

3.4 Evaluation

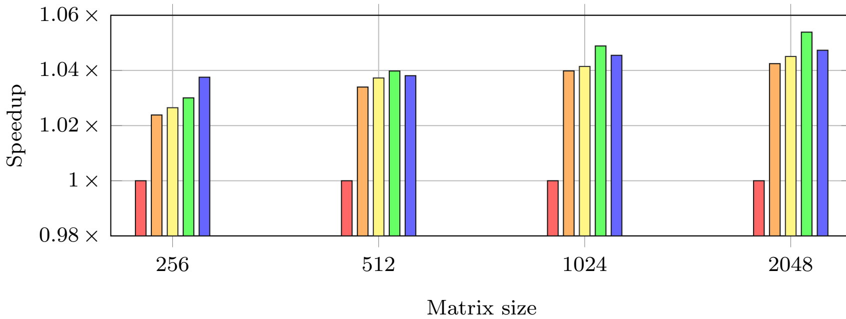
In this section we validate our approach through two case studies. The first is a simple *image processing* kernel that computes the difference between two input images. Albeit being a straightforward computation, its execution patternis the same of many image processing primitives, such as contrast and brightness alteration, or color balancing. Each work-group processes a slice of the image row-wise, each work-item processing the pixels in a row strided by the work-group size. The second case study is an implementation of the *matrix-multiply*, drawn from the NVIDIA SDK [68] sample programs. The operation is performed blockwise, and each work-group computes a block of 16×16 elements of the output matrix; each work-item in the work-group computes one element of the output block. To this end, the first input matrix is sliced by rows while the second is sliced by columns. The computation of the output block is done iteratively accumulating the partial results obtained from the corresponding input blocks. Both kernels have a fixed work-group size of 256 work-items, thus they cannot be executed on P2012 in their native form requiring the application of the *work-item coalescing* technique. Both kernels have a structure similar to the one

```
kernel myKernel(global uint *in, global uint *out,
                uint seq_len) {
  local uint buf_in[2][BUF_SIZE];
  local uint buf_out[2][BUF_SIZE];
  uint l = get_local_id(0);
  uint b = get_block_id(0);
  global uint *base_in = in + b * seq_len;
  global uint *base_out = in + b * seq_len;
  event_t ev_o = 0;
  event_t ev_i = async_work_group_copy(buf_in[0], base,
                                       BUF_SIZE, 0);

  uint i, x = 0;
  for (i = 0; i < seq_len; i += BUF_SIZE) {
    // Wait for previous async operations
    wait_group_events(&ev_i, 1);
    wait_group_events(&ev_o, 1);
    uint nx = (x + 1) % 2;

    // Issue the write of the previous iteration output
    if (i >= BUF_SIZE)
      ev_o = async_work_group_copy(base_out + i - BUF_SIZE,
                                   buf_out[nx], BUF_SIZE, 0);

    // Issue the read of the next iteration input
    if (i + BUF_SIZE < seq_len)
      ev_i = async_work_group_copy(buf_in[nx],
                                   base + i + BUF_SIZE,
                                   BUF_SIZE, 0);

    // Compute the current output
    compute(buf_out[x], buf_in[x]);

    // Swap the buffers
    x = nx;
  }
  wait_group_events(&ev_o, 1);

  uint nx = (x + 1) % 2;
  ev_o = async_work_group_copy(base_out + i - BUF_SIZE,
                               buf_out[nx], BUF_SIZE, 0);
  wait_group_events(&ev_o, 1);
}
```

Figure 3.5: Kernel after *Memory transfers optimization*: the local buffer is duplicated to implement double-buffering and asynchronous copies are used to overlap memory transfer with computation.



(a) Image difference

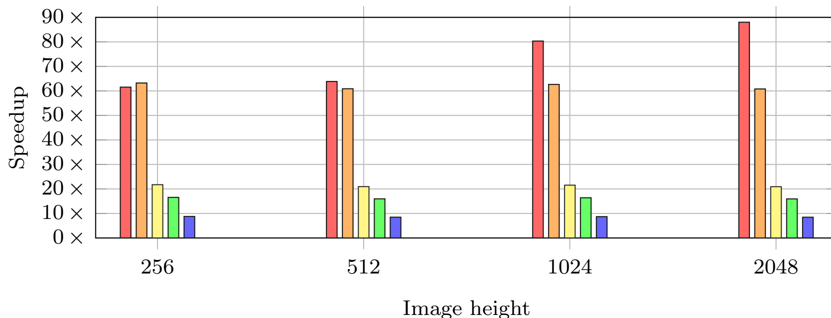


(b) Matrix multiply

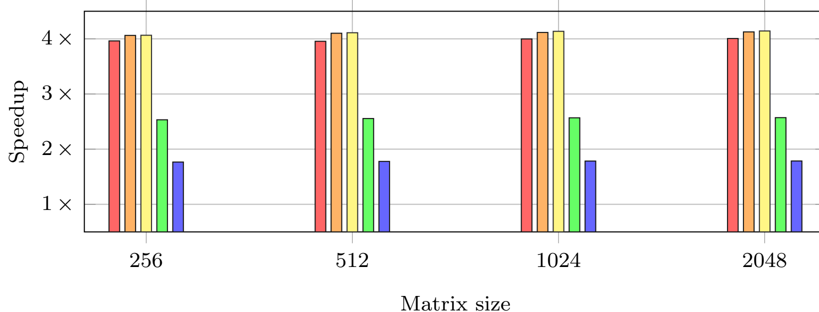
Figure 3.6: Speedup achieved through different mapping strategies, varying the size of the inputs. The figure shows the different mapping strategies with: 1 (our baseline – red), 2 (orange); 4 (yellow); 8 (green); and 16 (blue) work-groups per cluster.

reported in Listing 3.4, making them good candidates for the *memory transfer optimization*, with the matrix multiplication exhibiting a higher computation-to-memory-transfer ratio.

To perform the experiments, we ran the kernels using our LLVM-based OpenCL compiler and the P2012 OpenCL runtime, both with and without the proposed transformations, to ascertaining their impact on performance. Figure 3.6a shows the speedups achieved employing different mapping strategies on the image processing kernel, while Figure 3.6b reports the ones achieved on the matrix multiplication kernel. The speedups are normalized with respect to the basic strategy which maps 1 logical work-group to 1 cluster, employing a coalescing factor of 16. The proposed remapping strategy provides effective functional portability for the original code, which had fixed work-group sizes, since it is always possible, for all the sizes considered in our exploration to



(a) Image difference



(b) Matrix multiply

Figure 3.7: Speedup after the application of *memory pre-fetch optimization* varying the size of the inputs. Different colors represent strategies mapping 1 (our baseline – red), 2 (orange); 4 (yellow); 8 (green); and 16 (blue) work-groups per cluster.

have at least a working configuration after the proposed transformation. It is worth noting that coalescing multiple work items also yields a performance improvement when increasing the coalescing factor: this can be ascribed to the increase in the time spent in effective computation by the processing elements due to the larger size of the computation performed by a single transformed work-item. This in turn raises the ratio of useful computation-to-data transfer of the whole kernel.

To evaluate the effects of the automated insertion of asynchronous copies in the code, we performed a second round of experiments, measuring the effective speedup provided by such technique. Figure 3.7a reports the speedup after the *memory transfer optimization* for the image processing kernel while Figure 3.7b reports the speedup for the matrix multiplication kernel. As it can be seen, the overall speedups provided by this code transformation on P2012 are quite significant. On the image processing kernel the ability to overlap computations and memory

transfers decreases the entire kernel execution time up to a factor 60 to 88. On the matrix multiplication being able to overlap computation and memory transfers decreases the entire kernel execution time by a factor of 4.

We note that, depending on the coalescing factor, the number of parallel asynchronous copies may exceed the limits imposed on the currently available runtime support. In such a case the obtained speedup is lower as the runtime needs to serialize some of the asynchronous copies effectively reducing the overall performance. This effect is evident on mapping strategies with 8 and 16 work-groups per cluster in the matrix multiplication case, while 4, 8, and 16 work-groups per cluster are enough to saturate the runtime resources in the image processing kernel. However, we note that there is no limitation to the number of parallel asynchronous copies which can be inserted by our technique, thus the speedup limitation is to be ascribed to the runtime-architecture pair the code is running on.

The difference in the speedups achieved in the two case studies is due to the structure of the two computations. In particular, the image processing kernel obtains significant benefits from asynchronous copies as both its inputs and outputs are transferred at each iteration of the kernel. Moreover, the size of the outputs produced by each transformed kernel computation varies depending on the coalescing factor, allowing to better exploit the effects of the asynchronous copies. In addition to this factor, the relatively lightweight nature of the image difference kernel makes the memory transfers a dominating portion of the computation time: exploiting the possibility of asynchronous block-packed transfers has thus a significant impact. By contrast, the matrix multiplication computation produces constant-sized blocks of output data, regardless of the size of the input matrices, and is characterized by a higher computation-to-data-transfer ratio, in turn making the process of employing a double buffering strategy less advantageous, when compared with our other case study.

3.5 Related Work

Several recent works have highlighted the impact of tuning or even rewriting OpenCL applications to enhance the performance on specific architectures. Zhang *et al.* report that a number of manual tuning techniques can improve the performance of OpenCL applications on general purpose CPUs from 15% to more than 60% [92], while Cao *et al.* report performance penalties and the need of architecture-specific tuning when

implementing typical HPC workloads (e.g., linear algebra libraries) in OpenCL across a range of devices currently employed in HPC systems (GPGPUs and Intel Xeon Phi) [20]. Huang *et al.* and Fang *et al.* demonstrated how automated transformations can be used to achieve performance portability of GPGPU oriented OpenCL code to multicore CPUs [45, 34]. However, the proposed transformations in both works are simplifications of the original code (obtained removing explicit synchronization and local memory use), enabled by the more complex nature of the CPU cores w.r.t. the GPGPU architectures. Similarly, [80] targets the use of OpenCL on CPU devices. The work-item coalescing technique employed in our work has been used, for different purposes, in [59, 60]. A less directly related class of works aims at presenting multiple, heterogeneous devices as a single OpenCL device, performing automated load-balancing, taking into account the heterogeneity of the overall platform. Techniques such as those presented in this work or in [45] could be used to provide a greater range of options for the automated scheduler. Yang *et al.* analyzed several open source CUDA and OpenCL applications, detecting typical "performance bugs" affecting them, including sub-optimal usage of memory due to unnecessarily large data types or poor contention management, work-group formation, and lack of data reuse [90]. They also found performance portability issues between NVIDIA and AMD GPGPUs, mostly related to the different micro-architecture of the underlying processing elements. Consequentially, as this work proposes a transformation to provide functional portability of the parallel application on different platforms, it is sensible that it might introduce some of such performance bugs. The transformations required to provide the functional portability can also be used as an enabling technique to introduce further transformations for performance portability during the process, thus mitigating the loss of computational performance or even enhancing it.

3.6 Conclusion

In this work, we have studied the problem of portability of OpenCL code between two classes of devices, namely GPGPUs and embedded many-core accelerators with explicitly managed memory, both from the point of view of functionality and of the one of performance. We have proposed a code transformation technique, work-item coalescing, which can bypass limitations of the embedded platforms, allowing code developed for GPGPU to be ported seamlessly, as well as a memory transfer optimization technique to tune the resulting code to improve performance.

Our results on two case studies show the effectiveness of the proposed technique, which allowed the code, developed for NVIDIA devices and not designed with portability to smaller-scale ones in mind, to run on the STM-P2012 target. Future developments include improvements to the memory transfer optimization increasing the number of kernels that can be transformed. A possible approach is to apply pipelineing with double buffering exploiting the implicit loop iterating across work-groups overlapping data transfers and computations of two different work-groups.

4 C++ Support across Heterogeneous Systems

In this chapter, we present new techniques to enable full sharing of C++ code and data across heterogeneous systems composed of a host and multiple accelerator devices. We introduce a new approach for efficiently handling function pointers, which require special attention in the context of heterogeneous systems because multiple instruction-sets are involved. For data sharing, on the other hand, we exploit system-wide Shared Virtual Memory.

The proposed techniques enable not only plain function pointers shared across heterogeneous devices, but also the implementation of virtual member functions. Thus, we show how to enable full and transparent sharing of C++ objects, without breaking the host Application Binary Interface. We also illustrate the automatic generation of remote procedure calls for functions that are not present in the devices, either because the source code is not available or because the invoked functionality may be present only on the host, such as for operating system services.

The combination of system-wide Shared Virtual Memory with the proposed techniques enables the use of the accelerator devices from the very beginning of the application parallelisation and optimization process, eliminating the need of adapting existing C++ code to the APIs exposed by currently available heterogeneous programming models, such as OpenCL or CUDA.

4.1 Introduction

Programming heterogeneous systems composed of a host subsystem and a number of accelerator devices is notoriously challenging for two main reasons: 1) accelerators are typically parallel architectures that require the critical kernels of the applications to be parallelized in order to provide benefits; and 2) the interaction between the host and the devices is exposed to the programmer at different levels of abstraction depending on the adopted programming model, greatly increasing the effort required to port complex applications to heterogeneous systems.

We focus on the latter aspect, i.e. on simplifying the host/device interaction to the greatest possible extent. We leverage on the current hardware architecture trend of providing increasingly tightly integrated systems, and in particular on the opportunities offered by fine-grained system Shared Virtual Memory (SVM) as defined by the OpenCL 2.0 standard [51], or the Heterogeneous System Architecture Foundation (HSA) SVM model [44]. SVM greatly simplifies heterogeneous system programming by enabling the sharing of data pointers, and therefore of any data structure, across host and devices. However, one last step is still needed to enable the easiest possible heterogeneous programming model, that is, offering to developers a fully transparent model. To reach this goal, it is necessary to abstract the coexistence of multiple instruction-set architectures (ISAs), each endowed with its own Application Binary Interface (ABI), running concurrently on the heterogeneous system while sharing the same source code.

We assume C++ as the single source language for both host and devices. This assumption is in line with the current trends, both in language popularity among programmers and in adoption in standards (e.g. OpenCL C++ static kernel language [52], SYCL [53], AMP [64]). Current efforts in abstracting the host/device interface are still limited by the underlying assumption of the lack of SVM, for which data transfers are still explicit (although abstracted as in SYCL and AMP), and by the lack of support for the dynamic aspects of C++ in the devices, virtual member functions in particular. In our approach, we avoid the need for explicitly expressing data transfers by relying on system-wide SVM, while we focus instead on the required additional support from the toolchains to also enable full sharing of the source code among the host and devices, including function pointers and virtual member functions, so that *full* C++ programming across the entire heterogeneous system is made possible.

More precisely, the problem with function pointers in presence of multiple ISAs is that multiple versions of each function are needed — one for each ISA. Since function pointers are shared as any other data among host and devices, dereferencing, i.e., calling, a function pointer requires specific support to make sure that the right version of the function is invoked, depending on the device on which the call is performed. C++ is particularly sensitive to this issue due to virtual member functions, which are implemented through function pointers, demanding therefore an efficient implementation of the dereferencing mechanism. It is worth noting that, while in typical HPC applications features such as virtual member functions are rarely used, there are many potentially highly parallel applications which would benefit from such features. An exam-

ple is network packet filtering — a massively parallel application, where the nature of each packet determines the specific operations that need to be applied [87]. Such a problem can be easily modeled in object-oriented programming through a packet class, which is subclassed into several different types of packets, where each derived class redefines the operations to fit the structure and semantics of its own data.

In this chapter we illustrate a very efficient implementation that enables full sharing of function pointers in general as well as C++ objects, inclusive of their virtual member functions, across the host and any device on the system, while maintaining full compatibility of legacy ABIs on the host side. Note that the proposed technique and implementation apply to the C language as well, although their impact is comparatively lower, since C is not inherently object oriented, and therefore less sensitive to the function pointer sharing issue. Our focus is to provide a transparent implementation of heterogeneous function pointers, thus not relying on any a-priori knowledge about the actual mapping between functions and computational devices. Our approach does not prevent the exploitation of such additional knowledge to generate more efficient code, however this is beyond the scope of this work.

Even if full C++ is enabled, some functionalities are still unavailable on the devices. This is the case for system calls that require the host operating system intervention, but also external functions (e.g., library functions) for which source code is not available. For these cases, we provide a fallback solution that transparently invokes remote functions from the devices to the host. Invoking remote host services from the devices is obviously very expensive, as it implies the assembly and dispatch of messages between the device and the host. Moreover, the device execution is suspended while waiting for the responses. Thus, for performance optimization, remote calls should be either eliminated or kept to the minimum. However, it is still critical to provide this functionality for enabling an iterative optimization process.

In such a process, the developer focuses first on the more intellectually challenging task of parallelization of the application code for efficient execution on the accelerators. The compiler and language runtime can provide feedback on the remote calls still performed on the host, so that the developer can incrementally remove them. For example, our approach enables the use of STL (Standard Template Library) containers across the host and the device, which may require dynamic memory allocation when new elements are added. If the code adding such elements is executed on a device, a remote invocation on the host will eventually occur to perform the required memory allocation, incurring in very significant performance loss. However, it is easy to notify developers about such

events, so that they can take actions to remove such inefficiencies, e.g. by pre-allocating memory and/or by using customizedd memory allocators that can work transparently on the devices as well. Thus we remove the initial effort of porting data and code across host and devices, while enabling the iterative optimization process. We believe such a process to be much more efficient than the current approach, which forces the developers to apply a large number of transformations to move from a functional specification of their algorithms to even an initial version of the same code able to run on an heterogeneous platform.

Contributions

In this work, we provide two main contributions:

1. A lightweight method for implementing function pointers on ISA-heterogeneous architectures, so that on each architecture the function pointer refers to the appropriate code compiled for that architecture. This is achieved transparently, so that the programmer does not need to worry about explicitly identifying the different versions of the same code.

2. A parallel programming model employing full C++ support, obtained through the abovementioned transparent function pointer implementation and Shared Virtual Memory, to allow parallel code to be deployed on heterogeneous platforms.

Organization of the Chapter

The remainder of this chapter is organized as follows. In Section 4.2, we introduce the technique developed to efficiently implement transparent function pointers, while in Section 4.3 we show how to use them to implement a full parallel programming model. In Section 4.4, we report on the efficiency of the proposed implementation techniques, by means of an experimental campaign, while in Section 4.5 we compare our approach with recent related works. Finally, in Section 4.6 we draw some conclusions and highlight future research directions.

4.2 Transparent Function Pointers

In this section, we present the first major contribution of this work, a mechanism to efficiently implement transparent C++ function pointers on heterogeneous architectures.

In C/C++, a function pointer is a data type that represents a reference to a given function. The most common and straightforward implementation consists in a data pointer holding the memory address of the body of the function it points to. This representation is not well suited for heterogeneous platforms, where multiple implementations of a given source code function (one for each different ISA, e.g.) could be present. In this case, a single memory address cannot represent adequately the function, and a *shared function pointer* implementation is needed.

```
extern void (*fnptr)();

static void func() {
  // ...
}

void init() {
  fnptr = func;
}

void use_fnptr() {
  (*fnptr)();
}
```

Figure 4.1: Example of shared function pointer, in C language. The function pointer is a global variable, thus shared among all devices. All the functions in the translation unit are compiled for all the devices. The function pointer can be initialized either from the host or from a device, and it can be used by both host and devices.

Figure 4.1 shows an example of a shared function pointer. At source code level, the global variable `fnptr` represents a function pointer. The `init` function initializes the function pointer value, associating it with the body of function `func`. The `use_fnptr` function uses the function pointer to make an indirect call. Since in a fully transparent heterogeneous platform all source code functions can potentially be executed on all devices, the compiler does not statically know which version of `func`, i.e. which memory address, must be stored in `fnptr` in the initialization. Let us consider how the compiler can store the necessary information.

A first viable solution is the use a *fat pointer*: instead of representing a function pointer as a single data pointer, we represent it as structure containing a data pointer for each device in the system. With this strategy, there is no ambiguity as indirect function calls can be compiled to use the correct field of the fat pointer depending on the target device. Figure 4.2a and Figure 4.2b show in pseudo-C respectively the code compiled for the host and for a device. Function `init` stores all the

```
extern struct {
  void *host;
  void *device;
} fnptr;

void init() {
  fnptr = {
    __host__(func),
    __device__(func)
  };
}

void use_fnptr() {
  @icall(fnptr.host)();
}
```

(a) Host code

```
extern struct {
  void *host;
  void *device;
} fnptr;

void init() {
  fnptr = {
    __host__(func),
    __device__(func)
  };
}

void use_fnptr() {
  @icall(fnptr.device)();
}
```

(b) Device code

Figure 4.2: Fat pointer representation of function pointers for heterogeneous platforms. The operators __host__ and __device__ are used to represents respectively the host and device version of the symbol operand. Indeed we use the operator `@icall` to represent an call operation.

memory addresses of the different version of the function `func` in `fnptr`, while in function `use_fnptr` only one field at time is used depending on the device the function is compiled to.

Although this solution is compliant with the C and C++ standards, it does break the host ABI as the new memory layout of function pointers is not interoperable with the old one, introducing an incompatibility with pre-compiled libraries. Furthermore, the fat pointer solution breaks another common assumption — that the conversion between function pointers and data pointers and vice-versa is well defined and lossless. This property is commonly implemented in modern compilers and is used in many occasions in general purpose software, e.g., it is exploited by JIT compilers and within debuggers. Since compatibility with pre-compiled libraries is a highly desirable property, we investigate ABI-preserving solutions.

To preserve the host ABI compatibility, we cannot change the memory layout of a function pointer. Thus the representation of function pointers must be a *data pointer* containing the memory address of the *host version* of the function it must point to. Therefore, an indirect call performed on a device cannot use the function pointer value as the address of the called function. The function pointer value, however, can be used as a unique identifier of the function. Thus, it is possible to use

```
void indirect_call(void (*fnptr)()) {
  void *key = (void*)fnptr;
  void *value = map_lookup(key);
  if (value) {
    @icall(value)();
  } else {
    // fallback path
  }
}
```

Figure 4.3: Pseudo-code of an indirect call to a device.

an associative map for each device to bind the function pointer value to the addresses of the corresponding code for that device. Figure 4.3 shows the pseudo-code of an indirect call to a device adopting this approach. The function pointer value is used as key for the lookup in the associative map (`map_lookup`). The result of the lookup is not null if the indirect call is possible — i.e., if there is a compiled version of the function code for the target device. In the general case, of course, there is no guarantee that each function will be compiled for all the devices — e.g., system calls will be compiled only for the host. Thus, we need a fallback path to perform the call even if there is no version of the function available for the selected device.

An efficient implementation of the associative map is critical for the performance of the indirect calls. Straightforward options include vectors of key-value pairs and hashtables. Unfortunately, neither is particularly effective. A vector of key-value pairs, sorted by key, can be built at link time. Using a binary search to implement the lookup function, the lookup cost would vary depending on the number of possible function pointer values ($\Theta(\log(n))$, where n is the number of function pointer values), which is an undesirable property. A hashtable would guarantee an amortized constant time complexity for the lookup. However, in the general case, the hashtable should be built at runtime to support relocatable code. This negatively affects the startup time of each program.

To avoid these undesirable side effects, we avoid building an explicit data structure to represent the associative map, preferring instead a sparse encoding of the key-value pairs. In particular, we rely on an *heterogeneous linker*, discussed later in 4.3.4, for the generation of *decorated trampolines*. A trampoline is a short code stub that is generally used to reach far locations through an absolute jump. We define the trampoline

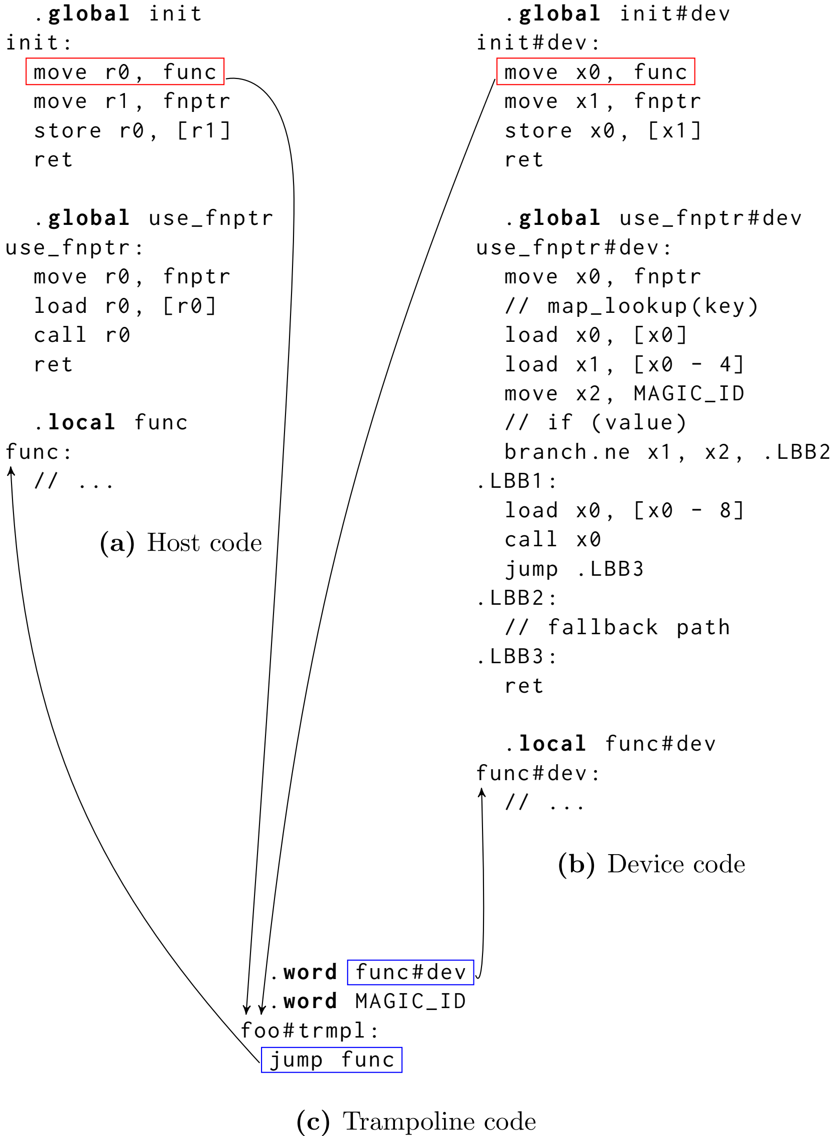
decoration as the set of all the data encoded just before the absolute jump.

Figure 4.4 shows in pseudo-assembly the code generated for the host and a device for the sample source code in Figure 4.1, and the trampoline code. Note that functions are compiled for both host and device. When compiling for a device, the name of each function is mangled with a unique suffix, e.g. the *#dev* suffix. For both the host and the device the function pointer is initialized using the symbol corresponding to the host version of the function. The linker will then resolve this symbol to the trampoline. The translation of the indirect call on the host is therefore straightforward — `fnptr` points to the address of the trampoline, which contains a jump instruction to the actual location of the function code. The indirect call for the device needs to perform the lookup first. For this purpose, we introduce a *magic identifier*, that is a fixed bit pattern corresponding to an invalid/reserved encoding for the host ISA, which identifies the presence of a trampoline. For the device, the code generated for the indirect call is a specialization of the pseudo-code in Figure 4.3. The map lookup implementation tries to match the *magic identifier*. If the match is successful then the address of the function can be loaded from the predefined offset. Otherwise, a fallback path is taken. A possible fallback implementation is discussed in Section 4.3.2.

In case of multiple devices, the trampoline decoration is composed of the magic identifier, and different data pointers at a predefined distinct offset – one for each device – containing the addresses of the corresponding device versions of the function if available, the null pointer otherwise. Therefore the call sequence must also be extended with a check to avoid the use of null values.

This solution guarantees a lookup in $O(1)$ time, indeed this approach can be employed to support shared libraries with *position independent code*: instead of encoding the absolute address the device version of the function, we put the distance between the addresses of the target function and the trampoline.

The implementation can be further optimized to ensure that the actual value of a function pointer is the address of the host version of the function, thus removing completely the overhead for host. This result can be achieved by placing the trampoline decorations exactly above the corresponding function, eliminating the need for the unconditional jump below the magic identifier. There are different ways to implement this optimization in practice. Our proposal is to let the assembler allocate the space required by a trampoline just before each function, and emit a special relocation to inform the linker that the trampoline space has been pre-allocated. Figure 4.5 shows the pseudo-assembly of the optimized



(a) Host code

(b) Device code

(c) Trampoline code

Figure 4.4: Example of decorated trampoline use. (a) shows the host code, (b) shows the device code, and (c) shows the trampoline code. Marked in *red* the relocations that the linker shall resolve using the address of the trampoline, while in *blue* the references within the trampoline of function `func` for both host and device.

```
  .global func
  .hctrmpl func
func:
  // ...
```

(a)

```
  .global func
  .word func#dev
  .word MAGIC_ID
func:
  // ...
```

(b)

Figure 4.5: Decorated trampoline optimization. In (a) the pseudo-assembly containing the directive `.hctrmpl` to allocate the space for the trampoline, while in (b) the equivalent pseudo-assembly after the materialization of the trampoline performed by the linker.

trampoline that contains the directive to allocate the trampoline itself, and the equivalent expanded version after the initialization of trampoline decorations. However the base solution can be useful for cases where a legacy library is specialized for a device keeping the original interface and linked against the original host library.

It is worth noting that the proposed technique can be also implemented entirely within the linker, allowing the use of unmodified legacy compilers for the host.

4.3 Integrating C++ Support across Heterogeneous Systems

In this section we extend the efficient implementation of transparent function pointers illustrated in Section 4.2 to enable full C++ support across heterogeneous instruction sets. As we are motivated by parallel accelerator devices, we also briefly introduce a `parallel_for` construct inspired by SYCL, as an example of a simple way to express parallelism without requiring language extensions. However the mechanisms exposed for supporting C++ are not limited to this specific parallel construct, on the contrary they are generic and applicable to any programming model. Once more, we assume Shared Virtual Memory to be provided by the architecture, together with cache coherency across all computational units.

Within the aforementioned setting, we represent computational kernels as parallel loops defined in terms of an *N-dimensional iteration space* and a function representing the *loop body*. In the following, we will employ a SYCL-like [53] syntax to encode our computational kernels, as it can be implemented using only templates, thus without any extension to the language. The basic form to express a kernel, shown in

```
template<typename T, int K>
void parallel_moving_average(const std::vector<T> &in,
                             std::vector<T> &out) {
  assert(in.size() > 2 * K);

  out.resize(in.size() - 2 * K);
  parallel_for(range<1>(in.size() - 2 * K),
               [&](id<1> it) {
    T sum(0);
    for (int i = -K; i <= K; ++i)
      sum = in[it.get(0) + i + K];
    out[it.get(0)] = sum / (2 * K + 1);
  });
}
```

Figure 4.6: An example of parallel programming using the proposed model.

Figure 4.6, is the `parallel_for` construct. This construct takes as parameters a C++ lambda function and the range of the iteration space. The lambda function is the body of each iteration thus takes as argument a given element of the iteration space. We also borrow from SYCL the way to explicitly represent a tiled iteration space defining the global size and the local size, i.e. the size of each tile. Indeed a `barrier` operation between iterations of a same tile is provided.

The assumption of full SVM, i.e., the whole address space is shared, and the shared function pointer implementation described in Section 4.2 allow us to support transparent calls between the host and the devices, and to support the full range of C++ constructs in our programming model, preserving ABI compatibility on the host side.

For this last purpose, it is necessary to guarantee that the data layout used within the host is unchanged, including class layout and virtual tables.

Finally, we need to support the fallback path for calls to functions for which an implementation is absent on a given device — in particular system calls.

In the rest of this section, we will provide an in-depth look at how *virtual member functions*, *pointer to member functions* and *generalized system calls* can be implemented in an heterogeneous environment, using our transparent function pointer implementation.

4.3.1 Virtual member functions

A virtual member function is a mechanism provided by the C++ language to support dynamic dispatch. The dispatch policy depends only

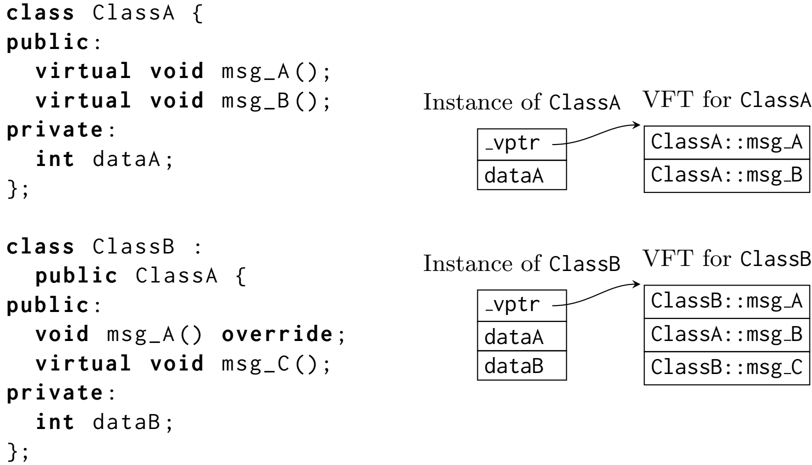


Figure 4.7: Single inheritance virtual tables.

on the runtime type of the target object. The most common implementation of this mechanism is based on *virtual function table*s (VFTs), i.e. a table of function pointers. The table indices are statically associated with the member functions, allowing access to the actual addresses. Dynamic dispatch in C++ is complicated by the need to support different cases depending on the type of inheritance: *single inheritance*, *multiple inheritance*, and *virtual inheritance*. We apply the trampoline technique used for transparent function pointers to manage virtual member functions as well. The application of this technique depends on the type of inheritance.

Single inheritance. In the case of single inheritance, virtual member functions of the derived class are assigned consecutive indices in the VFT, after those used for virtual member functions of the base class. Figure 4.7 shows a simple example of single inheritance and lists the virtual tables for two classes, `ClassA` and `ClassB`.

In an heterogeneous context, the implementation of shared function pointers is transparent w.r.t. single inheritance VFT, since the host function address also serves as the address of the trampoline to reach the device implementations. If the trampoline is not located immediately above the host function implementation, then its address replaces the host function address in the VFT.

Multiple inheritance. In the case of multiple inheritance the issue is the resolution of the conflicts of the index ranges of virtual member functions inherited by the different base classes. Such conflicts are resolved

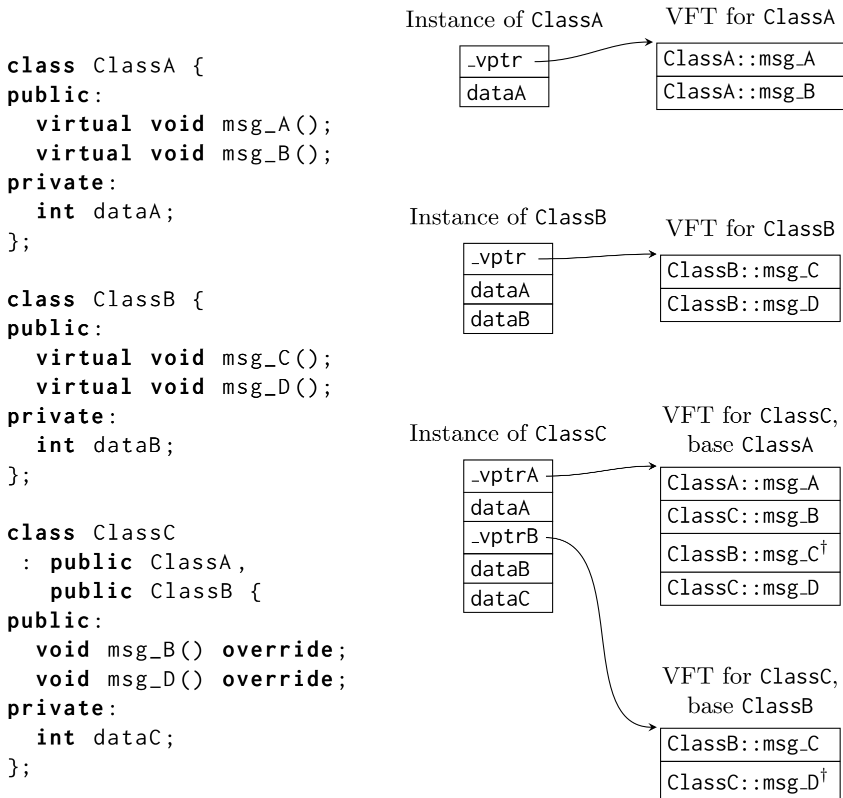


Figure 4.8: Multiple inheritance virtual tables. Entries marked with † indicate that the `this` pointer must adjusted before calling the method.

by creating one VFT for each base class. When dealing with multiple base classes, the object pointer must be properly adjusted before calling methods. Figure 4.8 shows an example of multiple inheritance and lists the virtual tables for the base classes `ClassA` and `ClassB`, and for the derived class `ClassC`. For `ClassC` there are two different virtual tables. The first is the proper virtual table of `ClassC`, used also when accessing the object using as static type `ClassA`, while the second is the virtual table of `ClassC`, used when the object is accessed using as static type `ClassB`. The object pointer adjustment is required whenever we need to access the object using one of the base type. Generally such adjustment is handled using a *thunk*. The pointer adjustment code is folded within the function registered in the virtual table. Thus, a thunk can be described as a small assembly stub that simply adjusts the object pointer and jumps to the expected function. This solution is efficient as the

pointer adjustment is performed only when needed and allows to keep the same layout of virtual tables as the single inheritance case.

In an heterogeneous context, each thunk must be emitted for all the devices. Thus, each entry of the virtual table allows to reach trampolines either for virtual member functions or for the corresponding thunks.

Virtual inheritance. In the case of virtual inheritance the sub-object offset of the virtual base is not fixed in the class hierarchy. The offset is encoded in each virtual table to to be used whenever a pointer adjustment is necessary. However this does not affect the references to virtual functions. Thus, in an heterogeneous context there is no need for any further extension to the function pointer solution to support virtual inheritance.

4.3.2 Generalized system calls

It is generally infeasible to have all functions compiled for both host and devices. This assumption would prevent the use of third party libraries, and/or pre-compiled ones. An extreme example are all the services offered by the operating system of the platform, i.e. system calls. We define as *generalized system calls* the set of all the functions, and libraries that either cannot be, or are not desirable to be supported natively for a given device. The use of such functionalities it is generally common, e.g. it is common to use the *libc* (the standard C library) to perform heap allocation through `malloc`. In principle this could accidentally happen in even in code that we would like to offload to devices. Because of this, a general mechanism to handle such cases should be provided to guarantee the functional behavior. The idea is to provide a mechanism that allows a device to forward the call of foreign function to the host, i.e. a *remote procedure call*.

Figure 4.9 shows a simple example of a stub and skeleton pair for function `fun`. Function `fun#dev_stub` is the stub generated by the device compiler, while `fun#skel` is the skeleton generated by the host compiler.

The compiler generates the code for both *stubs* and *skeletons*, as the former is the interface for the device while the latter is the interface for the host. A function stub must have the same signature of the function itself, such that it is possible for the linker to decide whether to solve relocations on call instructions with either the function address or the stub address. The stub code must prepare a request message for the host containing the parameters of the call, and send it to the host along with the skeleton address that must be invoked. A skeleton of a function is in charge of unpacking the parameters from the message, perform

```
int func(int a);

typedef struct {
  int res;
  int a;
} msg_func;

void func#skel(void *m) {
  msg_func *msg =
    (msg_func*)m;

  msg->res =
    func(msg->a);

  __rpc_done(msg);
}
```

(a) Host code

```
int func(int a);

typedef struct {
  int res;
  int a;
} msg_func;

int func#stub(int a) {
  msg_func msg =
    { 0, a };

  __rpc(func#skel,
        &msg);

  return msg.res;
}
```

(b) Device code

Figure 4.9: Stub and skeleton example.

the actual call to the host version of function, store the result into the response message and send it.

In the case of direct calls the compiler generates conservatively stub and skeleton for each external call, and the linker must resolve the call relocation. In the case of indirect calls, the generated code implements the general schema in Figure 4.3 where the fallback path is the invocation of the stub. On indirect calls both stub and skeleton are derived using the function pointer type, and one extra field is added in the message to store the function pointer value.

Furthermore, to support variadic functions, stubs and skeletons must be dependent on the actual call site parameter list. This is necessary as the called function does not statically know the number of the parameters, thus to properly serialize the parameters within the message we need to assume the caller point of view.

A tricky case is represented by those function that contain **`va_list`** as argument or return type. The `va_list` is the descriptor to have access to the variable arguments of a variadic function. The definition of this type is platform specific, and is part of the ABI. There is no easy way to support a remote call to a function dealing with such parameter without any change to the calling convention on the host or device. Thus, a viable solution is to enforce a particular calling convention for the device that, in case of variadic function, is compatible with the one

of the host. However, the use of such functions is infrequent, allowing an implementation to avoid supporting remote calls for this case.

4.3.3 Other C++ features

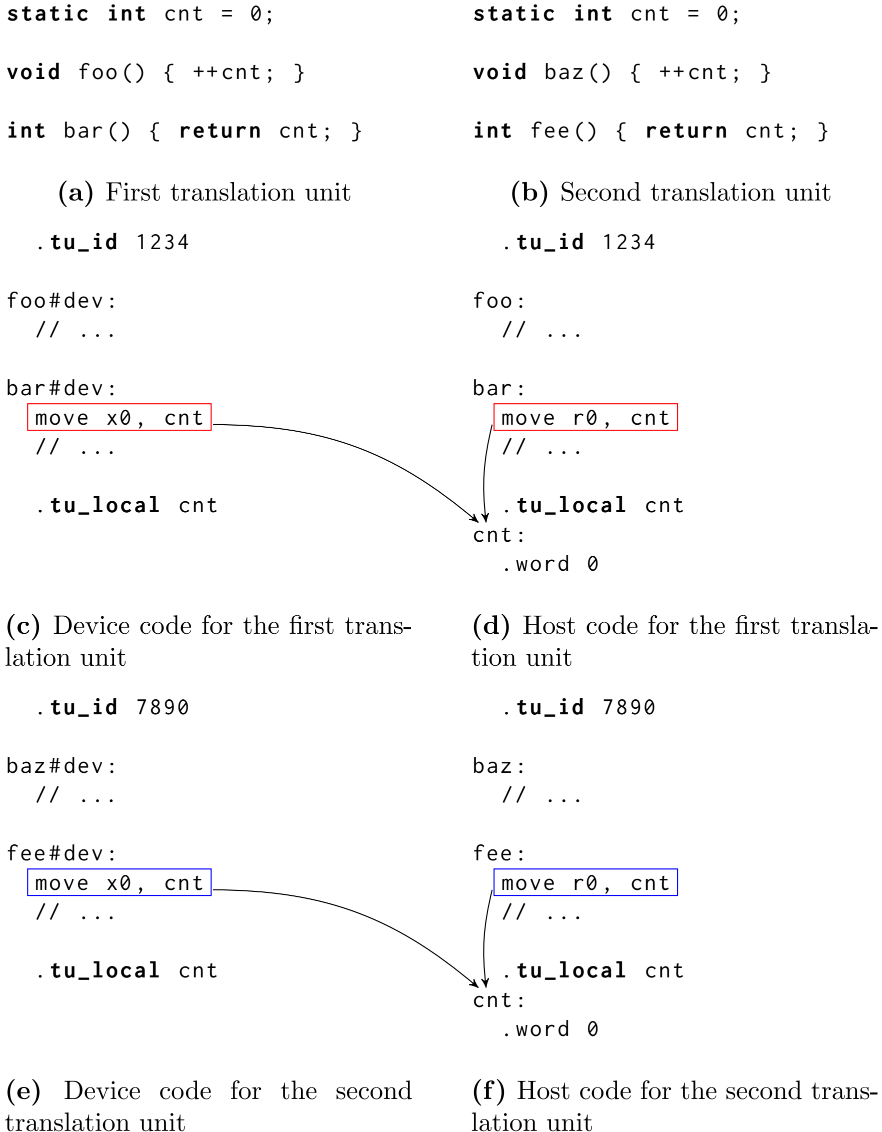
There are other, minor, C++ language features that can be easily supported within an heterogeneous context, like *pointer to member functions* and *runtime type information*. Pointer to member functions are the generalization of function pointers to member functions. Depending on the kind of function we want to point to, the memory layout of pointers to member function may be different. In the case of static member function or non-virtual member function, the pointer to member function is just like a raw function pointer. In the case of a virtual member function, the pointer to member function can be a struct containing the index of the virtual table entry to call and the adjustment value that must be added on the object pointer. The function pointer solution described in Section 4.2 can be applied to pointer to member functions as well.

Runtime type information are used to support `dynamic_cast`, `typeid` operators, and exceptions. Under the assumption of shared virtual memory, there is a unique instance of the runtime type information by construction, as it is encoded within the virtual table of class.

4.3.4 Heterogeneous Linker

Our proposed solution for shared function pointer implementation requires the linker to cooperate to ensure the correct behavior at runtime. Since the main task of the linker is to link together different modules to generate the final executable, it has a complete view of the compiled program, thus it is able to resolve symbols based on this knowledge. The natural extension of this concept within an heterogeneous context is represented by an *heterogeneous linker*, able to link modules generated for both the host and the devices.

The symbol resolution mechanism must handle the symbols defined in either host or devices modules and referenced potentially in both. Global data must be defined at most once, so we can safely assume these are defined only in host modules. The first task of the linker is to link shared data to both host and devices code. In particular, we need to properly map the concept of `static` global variables of the C language, i.e. a variable visible only within the translation unit. In general, static variables are mapped to local symbols within the object file. However in an heterogeneous context a translation unit is processed by multiple



(a) First translation unit

(b) Second translation unit

(c) Device code for the first translation unit

(d) Host code for the first translation unit

(e) Device code for the second translation unit

(f) Host code for the second translation unit

Figure 4.10: Example of static variable handling: Figure (a) and (b) show two translation units both containing a static variable `cnt`. Figure (d) and (c) show the equivalent pseudo assembly for host and device of the first translation unit, while in (f) and (e) the equivalent pseudo assembly for host and device of the first translation unit. Highlighted in red the relocation of `cnt` for the first translation unit, while in blue the relocation of `cnt` for the second translation unit

compilers that generate different object files, thus a static variable may be referenced in different object files derived from the same translation unit. Therefore we need a new kind of symbol that represents such variables, allowing the linker to unify the references to local symbols of different object files derived from the same translation unit.

Figure 4.10 shows a simple case of static symbols resolution. Figure 4.10a and 4.10b show two C translation units containing the static variable `cnt`, thus not visible across the translation unit boundary. Figure 4.10d, 4.10c, 4.10f, and 4.10e show the pseudo-assembly of the two translation units for both host and device. Each translation unit is uniquely identified, allowing the linker to cluster all the object files derived from the same translation units and solve all the references of static variables within each cluster.

The linker is also responsible for the generation of decorated trampolines. The trampoline address must be used to fixup relocations associated to the assignments of function pointers. Whenever the space for a trampoline has been pre-allocated, the linker encodes only the decoration of the trampoline and uses the host function address as value for the symbol resolution. The same process must be performed also on virtual table entries, to ensure the correct behavior of calls to virtual member functions. Furthermore the linker must properly handle relocations on direct calls for the device code. Having visibility on the whole program, the linker knows whether a function symbol is defined or not, thus it can fixup the relocation on a direct call using the function address, if it exists, or the stub address.

Moreover, to support shared libraries, structures related to exported symbols, e.g. the *global offset table*, must be enriched with device symbols. For each symbol that should be exported and for which a trampoline has been generated, the corresponding entry must point to the trampoline.

Finally, load time relocation shall be emitted for both host and devices code. Relocations of device code sections may be recorded separately from host relocations. This can be useful not to modify the system loader. Under such assumption, a post-loading phase may be required in order to solve separately device relocations before the invocation of the `main` function.

4.4 Evaluation

In this section, we present an experimental analysis of the overhead imposed in supporting heterogeneous function pointers using the proposed approach.

The focus of the evaluation is to show the impact of the heterogeneous function pointers overheads for both host and device. The impact on the host performance is expected to be negligible, even in the non-optimized form, since the overhead is represented by a single unconditional jump to the target function. In the optimized version, on the other hand, there is no overhead for indirect calls to the host function.

The impact on the device side is largely dependent on the device architecture, i.e. the memory hierarchy and access latencies and the cost of branch divergence that may occur on indirect calls. In terms of additional instructions, in the worst case, the overhead of an indirect call on the device can be summarized as two loads, two comparisons and one branch. On modern architectures with complex memory hierarchy and out-of-order pipeline, the actual impact also varies, depending on the application and the frequency of such calls. Additional penalties may be caused by cache misses while accessing the trampoline decorations as well as branch misprediction. In the case of GPGPUs, the limiting factors are, in general, the amount of latency for memory accesses and branch divergence. While the latencies due to memory accesses are generally masked in hardware by interleaving different workloads, the overhead due to branch divergence depends on the ability of the hardware to handle non uniform control flows. In typical GPGPUs, this is a significant issue, because only one active control flow is supported at any time. However, other many-core architectures, such as P2012 [16], support multiple flows.

The performance evaluation has been carried out in an environment which allows heterogeneous ISAs, and that supports shared virtual memory at system level, and cache coherency. The experiments have been run on a ARM big.LITTLE octa-core with four Cortex-A7 and four Cortex-A15. The environment is composed by ARM cores where the host uses the classic ARM instruction set, while the device uses the Thumb instruction set with NEON extension.

We first evaluated the overhead over traditional implementation of function pointers varying the total workload. The experiment is composed of an array of 256 function pointers, pointing to different functions with the same implementation. The code of each function is reported in Figure 4.11.

```
uint64_t test(int i, uint64_t v) {
  while (i > 0) {
    v = ((v >> 13) ^ CONST1) | ((v << 31) ^ CONST2);
    --i;
  }
  return v;
}
```

Figure 4.11: Test function with parametric workload.

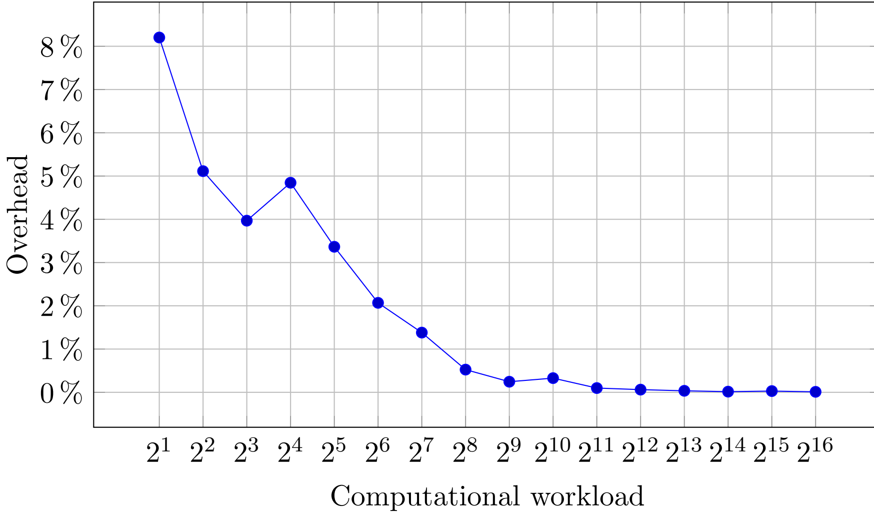


Figure 4.12: Overhead over useful computational workload.

Each function takes an argument that controls the amount of computation performed by the function, i.e. the number of iteration of a loop. We measured the time of performing 1000 random selection from the array of function pointers and calling the selected function. We ran several tests with different value for the computation amount ranging in $[2, 65536]$. Each run has been repeated at least 30 times in order to reduce the measurement error.

Figure 4.12 shows the results of the experiment. As expected the overhead decreases while increasing the amount of useful computation. In particular, in the worst case, i.e., with the work factor of the load function equal to 2, the overhead is only 8.2% and it rapidly decreases to zero. The overhead in terms of code size of the text section of the executable is $\approx 1.7\%$: this overhead is computed as the sum of all the trampolines generated and the additional code at each call site. Note

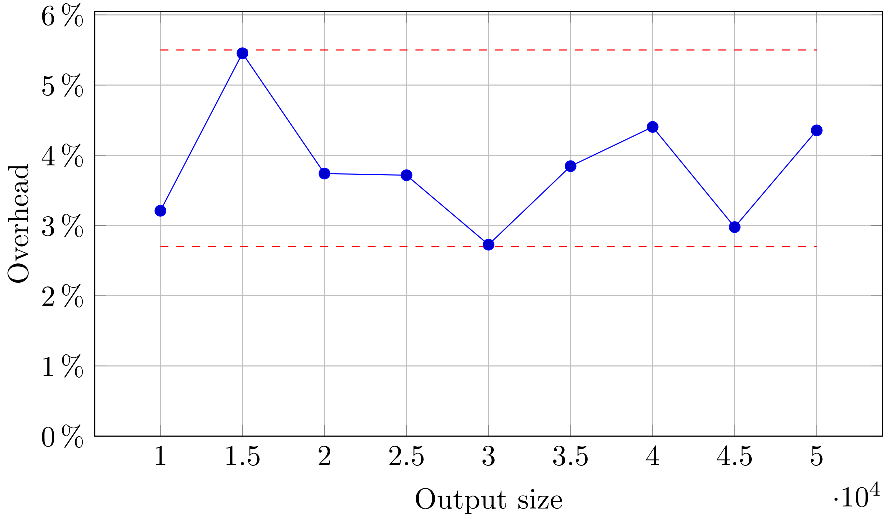


Figure 4.13: Overhead on Xerces-C++ DOM tree creation. The experimental results show that overhead is in the range of 2.7% − 5.5%. Note that 45.4% of the calls performed in each run are virtual.

that in this test we have exactly 256 function pointer values and just a single call site that requires extra code.

To obtain an assessment on real code, rather than on a synthetic benchmark, we also evaluated the overhead of heterogeneous function pointers using an existing C++ library. We select as a benchmark Xerces-C++[1], a C++ library for parsing XML documents and manipulating DOM trees. This library employs a non trivial class hierarchy to represent the elements in a DOM tree with a large number of virtual member functions. Thus, it represents a good choice for our evaluation — libraries less reliant on virtual member functions would obviously demonstrate lower overheads.

The selected test measures the performance of creating a DOM tree completely in memory. We ran the test several times varying the size of the generated DOM ranging in $[1 \cdot 10^4, 5 \cdot 10^4]$. Each run has been repeated 30 times in order to reduce the measurement error.

Figure 4.13 shows the overhead measured on a DOM tree creation program. Despite the fact that 45.4% of the performed calls are virtual in this test, the overhead on the whole execution time is contained within the $[2.7\%, 5.5\%]$ range. The overhead in terms of code size of the text section of the executable is $\approx 11\%$: this overhead is computed as the sum of all the trampoline generated and the additional code at each call site.

4.5 Related Work

The development of programming models for heterogeneous architectures has received the greatest impulse from the GPGPU computing world. NVIDIA CUDA [65], in particular, was the first attempt to provide a programming model for heterogeneous devices to reach significant commercial success. CUDA, being a proprietary programming model and toolchain, only supports NVIDIA devices, but the main ideas behind it were soon adopted by the Khronos Group consortium to define the open standard OpenCL [51]. Both models rely on explicit separation of host and device code – a feature that was critical to their success, giving control to the programmer and simplifying the implementation of compilers and deployment libraries, but that in the long run may prove their downfall. Indeed, both CUDA and OpenCL provide cumbersome syntaxes and require significant boilerplate code to setup the offloading of kernels to the compute devices. More recently, attempts have been done at reducing such drawbacks, by exposing lighter syntax based on pragmas or special keywords marking the parallelizable regions. OpenACC [70], the Codeplay Offload technology [22] and Microsoft's C++AMP [64] fall in this category. These systems share similar characteristics. In short, all need to identify explicitly the code regions that can be offloaded to an accelerator device, in order to perform static checks and code transformations necessary to generate the accelerator code and the offloading wrapper code. Recent attempts to support C++ execution on heterogeneous architectures, such as SyCL [53] and PACXX [41], still need to maintain the programmer aware of the offloading process – in contrast, our approach aims at complete transparency. The programmer only needs to deal with the parallelization of code, which is expressed without need of additional syntax and without imposing special restrictions on the offloaded code.

In [29], the problem of heterogeneity is considered in the context of chip multiprocessors. The authors employ padding to preserve the offsets of function addresses with respect to the base address of the code section across different ISA. Thus, the function pointers have the same value for all ISAs. In our proposal, padding is not needed, so ISAs that lead to very different code sizes suffer less penalties in memory occupation. While our indirect function calls become slightly more complex for the devices, the increased flexibility in linking makes the (very limited) overhead worth paying.

4.6 Conclusion

We have presented a method to transparently implement shared function pointers in heterogeneous platforms where two or more ISAs coexist, and Shared Virtual Memory is available. Our proposal, based on the trampoline technique for indirect calls, allows to preserve compatibility with the host ABI, while introducing only minimal overheads in terms of performance ($2.7\% - 5.5\%$) and code size ($\approx 11\%$). We assume to have an heterogenous linking process to wire up both host and devices object files, and materialize such trampolines. This technique is the building block to support C++ with virtual member functions. We provided a fallback solution to implement device call to function not available on the device itself (e.g. operating system services) based on RPC mechanism where the compiler is responsible to generate stubs and skeletons, and the heterogeneous linker decides at link-time whether stubs shall be used or not depending on the presence of the target function for a given device.

Future developments include the implementation of exceptions in heterogeneous contexts (open problems include the semantics of exceptions and the hardware support needed), and a refinement of the programming model (e.g., allowing task spawning from the devices, with semantics similar to dynamic parallelism in NVIDIA Kepler).

Conclusion

Programming parallel architectures is a complex task, since many hardware features are directly exposed to the programmers. Programming frameworks that try to hide such complexity exist, however they either provide only sub-optimal performance with respect to hand tuned implementations, or they are limited to specific application domains. Industry standard programming frameworks, such as OpenCL, allow to provide functional and performance portability, although they assume that the programmer is willing to shoulder the burden of explicitly supporting it. Indeed, future architectures may help to alleviate the issue through providing advanced features such as Shared Virtual Memory.

We have introduced OpenCRun, an OpenCL runtime implementation aiming at supporting the execution of OpenCL kernels on a range of platforms with very different architectural characteristics. Among these, the PULP platform has been designed to optimize power efficiency. To this end, we cooperated with the PULP hardware design team to co-explore ISA extensions and their compiler support to optimize the OpenRISC core, used as the processing element of PULP. We have reaped significant benefits in terms of energy efficiency with only reduced area costs. At cluster level the energy savings ranges between 39% and 66%, with an area increment is only 2.3% and power consumption increased by 18%. On average the cluster is 47.8% more energy efficient than the initial architecture.

To improve functional and performance portability of OpenCL code between GPGPUs and embedded many-core accelerators with explicitly managed memory such as PULP and STHorm, we have proposed a code transformation, work-item coalescing, to bypass the limitations of embedded platforms, as well as memory transfer optimization to improve the performance. Work-item coalescing allows to deal with non-native work group sizes, transforming the kernel code to remap work-items onto the processing elements, and allowing code developed for GPGPU to be ported seamlessly, while the memory transfer optimization employs double buffering technique to improve performance overlapping memory transfers and computation exploiting the DMA capabilities of embedded platforms. Our results on two case studies show the effectiveness of the proposed technique, which allowed the code, developed for

NVIDIA devices and not designed with portability to smaller-scale ones in mind, to effectively run on a STHorm target with a speedup up to 88× for non computational intensive kernels and a speedup up to 4× for computational intensive kernels.

To increase the abstraction level in a more radical way, leveraging Shared Virtual Memory that is expected to be available in future architectures, we have presented a method to transparently implement shared function pointers in heterogeneous platforms with two or more ISAs. Our proposal, based on the trampoline technique for indirect calls, allows to preserve compatibility with the host ABI, while introducing only minimal overheads in terms of performance ($2.7\% - 5.5\%$) and code size ($\approx 11\%$). This technique is the building block to support C++ with virtual member functions. We provided a fallback solution to implement function calls from device to function not available on the device itself (e.g. operating system services) based on RPC mechanism where the compiler is responsible to generate stubs and skeletons, and the heterogeneous linker decides at link-time whether stubs shall be used or not depending on the presence of the target function for a given device.

Future developments regarding the topic discussed in Chapter 3 include improvements to the memory transfer optimization to increase the number of kernels that can be transformed. A possible approach is to apply pipelining with double buffering exploiting the implicit loop iterating across work-groups overlapping data transfers and computations of two different work-groups. On the other hand regarding the topic discussed in Chapter 4, future developments include the implementation of exceptions in heterogeneous contexts (open problems include the semantics of exceptions and the hardware support needed), and a refinement of the programming model (e.g., allowing task spawning from the devices, with semantics similar to dynamic parallelism in NVIDIA Kepler).

List of Figures

1.1 Quad-core SMP UMA architecture. 10
1.2 Quad-core SMP NUMA architecture. 11
1.3 Intel QuickPath Interconnect architecture. In (a) is shown the Nehalem architecture. In (b) the interconnection through point-to-point links of four Nehalem processors. 12
1.4 NVIDIA Fermi architecture. In (a) the architecture of a Fermi SM, while in (b) the global architecture of a Fermi GPGPU. 14
1.5 AMD GCN architecture. In (a) the architecture of a single GCN core, while in (b) the global architecture of a AMD GCN graphic processor. 15
1.6 TILE64 architecture 16
1.7 Xeon Phi architectures. In (a) the *Knights Corner* architecture, while in (b) the *Knights Landing* architecture. . . 17
1.8 The OpenCL platform. In (a) the relationship between host and devices, and the structure of a device, while in (b) the memory hierarchy within a generic device. 23
1.9 Fork-join model. The execution path of the master thread is highlighted in grey. On the top of the picture the sequential execution flow, where only the master thread is active. On the bottom of the picture the parallel execution flow, where the master thread forks to generate worker threads to execute the tasks in parallel, and joins them to synchronize. 25

2.1 OpenCRun toolchain overview for X86 and STHorm. . . . 34

2.2 Initialization of work-item private memory for the x86-64 architecture. The `rsp` slot is initialized with the address of the *end* of the area reserved to the stack as it grows downward. The *return address* slot is initialized with the address of a custom trampoline used to start the execution of the work-item. This trampoline assumes that `rbp` points to the *kernel entry point* slot, thus the `rbp` slot is initialized with the address of the kernel entry point slot. Apart for the initialization value before the execution, the return address slot is used to hold the proper return address value across work-item switches. 37
2.3 OpenCL builtin prototype description within `oclgen` . . . 38
2.4 PULP cluster featuring four OpenRISC processor cores, a shared instruction cache, eight TCDM banks utilized as L1 memory and a DMA for fast, concurrent data movements. 42
2.5 Example showing the benefit of unaligned access when computing a stencil over an image with 8 bit pixels. . . . 48
2.6 Comparison of the speedup with the introduced ISA-extensions versus the initial ISA. Number of cycles are measured in RTL simulations for the Or10n core and in case of the Cortex-M4 on the real hardware. All the results are normalized w.r.t. the number of cycles on the plain OpenRISC ISA. 51
2.7 Area and power comparison with different ISA-extensions on core and cluster level. Power is shown on a cluster with 1 and 4 cores. 52
2.8 Energy efficiency of the core with the optimized ISA integrated in the multi-core cluster in UMC 65 nm. All the results are normalized w.r.t. the absorbed energy on the plain OpenRISC ISA. 52

3.1 Structure of two multicore architectures with OpenCL support: the *NVIDIA Kepler* GPU and STM-P2012 . . . 60
3.2 Example of a kernel fission transformation 62
3.3 Work-items mapping strategies on a P2012 cluster. Two different strategies based on the partial mappings of the work-items are shown in (a) and (b), while in (c) a full work-items coalescing strategy is shown 63
3.4 Structure of kernel for profitable memory transfer optimization. 65

3.5 Kernel after *Memory transfers optimization*: the local buffer is duplicated to implement double-buffering and asynchronous copies are used to overlap memory transfer with computation. 67
3.6 Speedup achieved through different mapping strategies, varying the size of the inputs. The figure shows the different mapping strategies with: 1 (our baseline – red), 2 (orange); 4 (yellow); 8 (green); and 16 (blue) work-groups per cluster. 68
3.7 Speedup after the application of *memory pre-fetch optimization* varying the size of the inputs. Different colors represent strategies mapping 1 (our baseline – red), 2 (orange); 4 (yellow); 8 (green); and 16 (blue) work-groups per cluster. 69

4.1 Example of shared function pointer, in C language. The function pointer is a global variable, thus shared among all devices. All the functions in the translation unit are compiled for all the devices. The function pointer can be initialized either from the host or from a device, and it can be used by both host and devices. 77
4.2 Fat pointer representation of function pointers for heterogeneous platforms. The operators `__host__` and `__device__` are used to represents respectively the host and device version of the symbol operand. Indeed we use the operator `@icall` to represent an call operation. 78
4.3 Pseudo-code of an indirect call to a device. 79
4.4 Example of decorated trampoline use. (a) shows the host code, (b) shows the device code, and (c) shows the trampoline code. Marked in *red* the relocations that the linker shall resolve using the address of the trampoline, while in *blue* the references within the trampoline of function `func` for both host and device. 81
4.5 Decorated trampoline optimization. In (a) the pseudo-assembly containing the directive `.hctrmpl` to allocate the space for the trampoline, while in (b) the equivalent pseudo-assembly after the materialization of the trampoline performed by the linker. 82
4.6 An example of parallel programming using the proposed model. 83
4.7 Single inheritance virtual tables. 84

4.8 Multiple inheritance virtual tables. Entries marked with † indicate that the `this` pointer must adjusted before calling the method. 85
4.9 Stub and skeleton example. 87
4.10 Example of static variable handling: Figure (a) and (b) show two translation units both containing a static variable `cnt`. Figure (d) and (c) show the equivalent pseudo assembly for host and device of the first translation unit, while in (f) and (e) the equivalent pseudo assembly for host and device of the first translation unit. Highlighted in red the relocation of `cnt` for the first translation unit, while in blue the relocation of `cnt` for the second translation unit 89
4.11 Test function with parametric workload. 92
4.12 Overhead over useful computational workload. 92
4.13 Overhead on Xerces-C++ DOM tree creation. The experimental results show that overhead is in the range of $2.7\% - 5.5\%$. Note that 45.4% of the calls performed in each run are virtual. 93

List of Tables

1.1 Comparison of parallel programming frameworks 30

2.1 Zero-overhead hardware loop operations. Each hardware loop is identified with an ID: `L0-L3`. The notation `HWLP_START[J]` is used to refer to the `HWLP_START` register of the Jth hardware loop. 43
2.2 Load/store addressing modes. Register-register addressing mode is expressed with the notation `rX(rY)`. Autoincrementing addressing modes are identified with a `!` before (preincrement) or after (postincrement) the base address. MMMMM in the sub-opcode is used to indicate bits that are dedicated to differentiate each type of load/store. 45

3.1 Analysis of OpenCL benchmark suite programs according to the management of work-group size parameter 58

Bibliography

[1] Xerces-C++. https://xerces.apache.org/xerces-c.

[2] DOE Pushes Back Plans for Exascale Supercomputing. http://www.hpcwire.com/2012/11/26/doe_pushes_back_plans_for_exascale_supercomputing/, 2012.

[3] FreeOCL: Multi-platform implementation of OpenCL 1.2 targeting CPUs. http://code.google.com/p/freeocl/, Accessed July 2015.

[4] Giovanni Agosta, Alessandro Barenghi, Alessandro Di Federico, and Gerardo Pelosi. OpenCL Performance Portability for General-purpose Computation on Graphics Processor Units: an Exploration on Cryptographic Primitives. *Concurrency and Computation: Practice and Experience*, 2014.

[5] AMD. AMD Graphics Core Next (GCN) whitepaper. http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, 2012.

[6] ARB. *OpenMP Application Program Interface, version 3.0*, 2008.

[7] ARM. ARM Cortex M0+/M4. http://www.arm.com/products/processors/cortex-m/. [Online; accessed 7-May-2015].

[8] ARM. big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms., 2011.

[9] Marnix Arnold and Henk Corporaal. Designing domain-specific processors. In *Int Symp. Hardw./Softw. Codesign*, pages 61–66, 2001.

[10] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[11] A.Y. Dogan et al. Low-power processor architecture exploration for online biomedical signal analysis. *Circuits, Devices Systems, IET*, 6(5):279–286, Sept 2012.

[12] Riyadh Baghdadi, Albert Cohen, Serge Guelton, Sven Verdoolaege, Jun Inoue, Tobias Grosser, Georgia Kouveli, Alexey Kravets, Anton Lokhmotov, Cedric Nugteren, Fraser Waters, and Alastair F. Donaldson. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. *CoRR*, pages –1–1, 2013.

[13] Krzysztof Banaś and Filip Krużel. OpenCL Performance Portability for Xeon Phi Coprocessor and NVIDIA GPUs: A Case Study of Finite Element Numerical Integration. In *Euro-Par 2014 Workshops*, volume 8806 of *LNCS*, pages 158–169. Springer, 2014.

[14] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, Feb 2008.

[15] Pieter Bellens, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. Memory – CellSs: a Programming Model for the Cell BE Architecture. In *SC*, page 86, 2006.

[16] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an Ecosystem for a Scalable, Modular and High-efficiency Embedded Computing Accelerator. In *DATE 2012*, pages 983–987, 2012.

[17] S. Benkner, S. Pllana, Traff J. L., P. Tsigas, and U. Dolinsky *et al.* PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. *Micro*, 31(5):28–41, 2011.

[18] Shekhar Borkar. Thousand Core Chips – A Technology Perspective. In *DAC*, pages 746–749, 2007.

[19] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Productive Cluster Programming with OmpSs. In *Euro-Par (1)*, pages 555–566, 2011.

[20] Chongxiao Cao, Mark Gates, Azzam Haidar, Piotr Luszczek, Stanimire Tomov, Ichitaro Yamazaki, and Jack Dongarra. Performance and Portability with OpenCL for Throughput-oriented HPC Workloads Across Accelerators, Coprocessors, and Multicore Processors. In *Proc. of 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '14, pages 61–68. IEEE Press, 2014.

[21] 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 *Proc. of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009*, pages 44–54, 2009.

[22] Codeplay. Offload Compiler SDK. https://www.codeplay.com/company/documents.html?folder=3.

[23] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini. Energy-efficient vision on the pulp platform for ultra-low power parallel computing. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6, Oct 2014.

[24] Pete Cooper, Uwe Dolinsky, Alastair F. Donaldson, Andrew Richards, Colin Riley, and George Russell. Offload – Automating Code Migration to Heterogeneous Multicore Systems. In Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, volume

5952 of *Lecture Notes in Computer Science*, pages 337–352. Springer Berlin Heidelberg, 2010.

[25] Intel Corp. Threading Building Blocks. http://www.threadingbuildingblocks.org/, 2009.

[26] Joe Curley. HPC and Big Data. *Innovation*, 12(3), July 2014.

[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 *Proc. of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010*, pages 63–74, 2010.

[28] B.D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P.G. de Massas, F. Jacquet, S. Jones, N.M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013.

[29] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution Migration in a heterogeneous-ISA Chip Multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 261–272, New York, NY, USA, 2012. ACM.

[30] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, 2000.

[31] A. Duran and M. Klemm. The Intel Many Integrated Core Architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 365–366, July 2012.

[32] Johan Enmyren and Christoph W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[33] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[34] J.B. Fang, H. Sips, P. Jaaskelainen, and A.L. Varbanescu. Grover: Looking for Performance Improvement by Disabling Local Memory Usage in OpenCL Kernels. In *43rd Int'l Conf. on Parallel Processing (ICPP 2014)*, pages 162–171. IEEE, Sept 2014.

[35] Michael J. Flynn. Very High Speed Computing Systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[36] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998.

[37] G.-R. Uh et al. Techniques for effectively exploiting a zero overhead loop buffer. In *Compiler Construction*, pages 157–172. Springer, 2000.

[38] Davide Gadioli, Simone Libutti, Giuseppe Massari, Edoardo Paone, Michele Scandale, Patrick Bellasi, Gianluca Palermo, Vittorio Zaccaria, Giovanni Agosta, William Fornaciari, and Cristina Silvano. OpenCL Application Auto-tuning and Run-Time Resource Management for Multicore Platforms. In *IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2014, Milan, Italy, August 26-28, 2014*, pages 127–133, 2014.

[39] M. Gautschi, A. Traber, A. Pullini, L. Benini, M. Scandale, A. Di Federico, M. Beretta, and G. Agosta. Tailoring instruction-set extensions for an ultra-low power tightly-coupled cluster of OpenRISC cores. In *Very Large Scale Integration (VLSI-SoC), 2015 IFIP/IEEE International Conference on*, pages 25–30, Oct 2015.

[40] I. Grasso, P. Radojkovic, N. Rajovic, I. Gelado, and A. Ramirez. Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU. In *28th IEEE Int'l Parallel and Distributed Processing Symposium*, pages 123–132, May 2014.

[41] Michael Haidl and Sergei Gorlatch. PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 1–11, Piscataway, NJ, USA, 2014. IEEE Press.

[42] H.P. Hofstee. Power efficient processor architecture and the cell processor. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 258–262, Feb 2005.

[43] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, Feb 2010.

[44] HSA™ Foundation. HSA Platform System Architecture Specification, Provisional 1.0. http://www.hsafoundation.com/?ddownload=4944, April 2014.

[45] Dafei Huang, Mei Wen, Changqing Xun, Dong Chen, Xing Cai, Yuran Qiao, Nan Wu, and Chunyuan Zhang. Automated Transformation of GPU-Specific OpenCL Kernels Targeting Performance Portability on Multi-Core/Many-Core CPUs. In *Euro-Par 2014*, volume 8632 of *LNCS*, pages 210–221. Springer, 2014.

[46] Intel. An Introduction to the Intel QuickPath Interconnect. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009.

[47] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A Performance-Portable OpenCL Implementation. *Int'l J. Parallel Programming*, pages 1–34, 2014.

[48] Ralf Karrenberg and Sebastian Hack. Whole-function Vectorization. In *CGO*, pages 141–150, 2011.

[49] Khronos Group. The Open Standard for Parallel Programming of Heterogeneous Systems. https://www.khronos.org/opencl/, (last accessed Aug. 2014).

[50] Khronos OpenCL Working Group. The OpenCL Specification, Version 1.2. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, October 2014. Aaftab Munshi eds.

[51] Khronos OpenCL Working Group. The OpenCL Specification, Version 2.0. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, October 2014. Lee Howes and Aaftab Munshi eds.

[52] Khronos OpenCL Working Group. The OpenCL C++ Specification, Version 1.0. https://www.khronos.org/registry/cl/specs/opencl-2.1-openclc++.pdf, February 2015. Aaftab Munshi ed.

[53] Khronos OpenCL Working Group – SYCL subgroup. SYCL™ Specification, Version 1.2. https://www.khronos.org/registry/sycl/specs/sycl-1.2.pdf, September 2014. Lee Howes and Maria Rovatsou eds.

[54] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[55] R.B. Lee. Subword parallelism with max-2. *Micro, IEEE*, 16(4):51–59, Aug 1996.

[56] Igor Loi, Davide Rossi, Germain Haugou, Michael Gautschi, and Luca Benini. Exploring Multi-banked Shared-L1 Program Cache on Ultra-Low Power, Tightly-Coupled Processor Clusters. *CF '15*.

[57] M. Gautschi, et al. SIR10US: A tightly coupled elliptic-curve cryptography co-processor for the OpenRISC. In *IEEE Int. Conf. Appl.-specific Syst. Archit. Processors (ASAP)*, pages 25–29, June 2014.

[58] Alberto Magni. Design and Implementation of a LLVM-based OpenCL compiler. Master's thesis, Politecnico di Milano, March 2011.

[59] Alberto Magni, Christophe Dubach, and Michael O'Boyle. A Large-scale Cross-architecture Evaluation of Thread-coarsening. In *Int'l Conf. High Performance Computing, Networking, Storage and Analysis, SC13*, page 11, 2013.

[60] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Exploiting GPU Hardware Saturation for Fast Compiler Optimization. In *7th Workshop on General Purpose Processing Using GPUs, GPGPU-7*, page 99, 2014.

[61] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):1–12, 2002.

[62] Giuseppe Massari, Edoardo Paone, Michele Scandale, Patrick Bellasi, Gianluca Palermo, Vittorio Zaccaria, Giovanni Agosta, William Fornaciari, and Cristina Silvano. Data Parallel Application Adaptivity and System-Wide Resource Management in Many-Core Architectures. In *Reconfigurable Computing: Architectures, Tools, and Applications - 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14-16, 2014. Proceedings*, pages 345–352, 2014.

[63] P. Meinerzhagen, C. Roth, and A. Burg. Towards generic low-power area-efficient standard cell based memory architectures. In *MWSCAS*, pages 129–132, Aug 2010.

[64] Microsoft Corporation. C++ AMP: C++ Accelerated Massive Parallelism, Version 1.2. http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf, December 2013.

[65] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[66] NVIDIA. NVIDIA Fermi whitepaper. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf.

[67] NVIDIA. GeForce 256 – The World's First GPU. http://www.nvidia.com/page/geforce256.html, 2012.

[68] NVIDIA Corp. OpenCL Programming Guide for the CUDA Architecture, Version 2.3. http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf, 2009, (retrieved Aug. 2014).

[69] NVIDIA Corp. CUDA Zone: OpenCL. https://developer.nvidia.com/opencl, (accessed Dec. 2014).

[70] OpenACC.org. The OpenACC™ Application Programming Interface, Version 2.0. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf#overlay-context=About_OpenACC, August 2013.

[71] OpenCores Community. OpenRISC Community Portal. http://opencores.org/or1k/Main_Page, 2014. [accessed 18-September-2014].

[72] OpenGL ARB. OpenGL Shading Language. https://www.opengl.org/documentation/glsl/.

[73] Alex Peleg and Weiser Uri. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[74] Nikola Rajovic, Paul M. Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? In *Proc. of the Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 40:1–40:12, New York, NY, USA, 2013. ACM.

[75] R.G. Dreslinski et al. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, Feb 2010.

[76] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An Experimental Study on Performance Portability of OpenCL Kernels. In *SAAHPC*, pages 3–13. IEEE Computer Society, 2010.

[77] Asadollah Shahbahrami, Ben Juurlink, and Stamatis Vassiliadis. A comparison between processor architectures for multimedia applications. In *ProRISC*, pages 138–152, 2004.

[78] C. Silvano, W. Fornaciari, S.C. Reghizzi, G. Agosta, G. Palermo, V. Zaccaria, P. Bellasi, F. Castro, S. Corbetta, E. Speziale, D. Melpignano, J.M. Zins, D. Siorpaes, H. Hubert, B. Stabernack, J. Brandenburg, M. Palkovic, P. Raghavan, C. Ykman-Couvreur, A. Bartzas, D. Soudris, T. Kempf, G. Ascheid, H. Meyr, J. Ansari, P. Mahonen, and B. Vanthournout. Parallel paradigms and run-time management techniques for many-core architectures: The 2parma approach. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pages 835–840, July 2011.

[79] C. Silvano *et al.* 2PARMA: Parallel Paradigms and Run-time Management Techniques for Many-Core Architectures. In *VLSI 2010 Annual Symposium*, volume 105 of *LNEE*, pages 65–79. Springer, 2011.

[80] John A. Stratton, Hee-Seok Kim, Thomas B. Jablin, and Wen-Mei W. Hwu. Performance Portability in Accelerated Parallel Kernels. Technical Report IMPACT-13-01, U. of Illinois at Urbana-Champaign, May 2013.

[81] John A. Stratton, Christopher Rodrigrues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, U. Illinois at Urbana-Champaign, March 2012.

[82] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014.

[83] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[84] Robert M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1), 1967.

[85] U.S. Department of Energy, Office of Science and National Nuclear Security Administration. Preliminary Conceptual Design for an Exascale Computing Initiative. http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20141121/Exascale_Preliminary_Plan_V11_sb03c.pdf, 2014.

[86] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, Jan 2008.

[87] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: A Multi-parallel Intrusion Detection Architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 297–308, New York, NY, USA, 2011. ACM.

[88] David W. Wall. Limits of Instruction-Level Parallelism. In *ASPLOS*, pages 176–188, 1991.

[89] William A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1), 1995.

[90] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. Fixing Performance Bugs: An Empirical Study of Open-Source GPGPU Programs. In *ICPP*, pages 329–339. IEEE Computer Society, 2012.

[91] Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Training Branch Prediction. In *MICRO*, pages 51–61, 1991.

[92] Yao Zhang, Mark Sinclair II, and Andrew A. Chien. Improving Performance Portability in OpenCL Programs. In *Proc. of 28th Int'l Supercomputing Conf., ISC 2013*, pages 136–150. 2013.