UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTE OF INFORMATICS
COMPUTER ENGINEERING

RAFAEL LEITES LUCHESE

Runtime library for parallel programming in MPPA®-256 manycore processor

Final report presented in partial fulfillment of the requirements for the degree of Computer Engineering.

Prof. Dr. Alexandre Carissimi
Advisor

Porto Alegre, December 2013

CIP – CATALOGING-IN-PUBLICATION

Luchese, Rafael Leites

Runtime library for parallel programming in MPPA®-256 manycore processor / Rafael Leites Luchese. – Porto Alegre: PPGC da UFRGS, 2013.

70 f.: il.

Final Report (Master) – Universidade Federal do Rio Grande do Sul. Computer Engineering, Porto Alegre, BR–RS, 2013. Advisor: Alexandre Carissimi.

1. Heterogeneous computing. 2. Manycore processors. 3. MPPA®-256. I. Carissimi, Alexandre. II. Título.

The only way to do great work is to love what you do. If you haven't found it yet, keep looking. Don't settle. As with all matters of the heart, you'll know when you find it. — STEVE JOBS

ACKNOWLEDGEMENT

I owe a great deal to my teacher and advisor, Mr Alexandre Carissimi, who, with his help, my monograph took its actual form and who served as a mentor for this work.

I also thank all the members of the Nanosim team, from the Université de Grenoble, involved in this work, who have provided me with interesting references and have helped me as far as the distance allowed.

I cannot forget, either, my familly and friends, which provided all the support I needed for this work to become true.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	*Arithmetic and Logic Unit*
API	*Application Programming Interface*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecturek*
DSP	*Digital Signal Processing*
FPGA	*Field Programmable Gate Array*
GCC	*GNU Compiler Collection*
GPGPU	*General Purpose GPU*
GPU	*Graphic Processing Unit*
HAS Foundation	*Heterogeneous System Architecture Foundation*
IDE	*Integrated Development Environment*
ISA	*Instruction Set Architecture*
MIMD	*Multiple Instruction Multiple Data*
NoC	*Network-on-Chip*
NODEOS	*Node Operational System*
OpenCL	*Open Computing Language*
OpenMP	*Open Multi-Processing*
OS	*Operational System*
PCIe	*Peripheral Component Interconnect Express*
Pocl	*Portable Computing Language*
POSIX threads	*Portable Operating System Interface threads*
RTEMS	*Real-Time Executive for Multiprocessor Systems*
SCC	*Single-Chip Cloud Computer*
SDK	*Software Development Kit*
SIMD	*Single Instruction Multiple Data*
TBB	*Threading Building Blocks*
VLIW	*Very Long Intruction Word*

LIST OF FIGURES

ABSTRACT

The demand for high processing power together with the actual concern about power consumption has led the industry towards heterogeneous solutions, which present a good compromise between these two factors. The challenge involving this approach relies on how to develop efficient and portable applications to take advantage of the parallelism available in such architectures.

Some software solutions appeared to help programmers developing code for these new class of heterogeneous processors. These solutions aim to provide an easy and flexible way to express parallelism in the applications. Some solutions such as CUDA and OpenCL present a software model and architecture that has proved to be very effective for general purpose programming on these devices.

Among the architectures that have been used for heterogeneous programming, manycore processors and CPUs combined with GPGPUs stand out as the most expressive solutions, obtaining the best results. The focus of this work is one specific solution, the MPPA®-256 manycore processor, which targets high parallel processing and low power consumption. This processor integrates 256 calculating cores for general purpose programming providing good processing capabilities.

As any other solution for heterogeneous computing available in the market, porting applications and developing new ones for this processor can be difficulty. As MPPA processor has unique hardware and software architectures, it increases the complexity of programming on it.

The focus of this work is to study and analyze the viability of implementing a runtime to be linked to the MPPA processor, with some high level functions aiming to help programmers with the task of easily setting up the programming context and necessary features to produce software in a faster way. This study contemplates the implementation and tests realized to validate the proposed solution.

Keywords: Heterogeneous computing, manycore processors, MPPA®-256.

1 INTRODUCTION

An important number of technological and scientific advances have been lately achieved in many different areas of the human knowledge because of the computational resources available worldwide. But the demand for very high processing performance continues to increase. Some applications such as image processing and analysis, signal processing control, security (cryptography) and telecommunications naturally demand computers with high calculation capabilities.

As the limits of hardware improvement are starting to be reached, making it impossible for the hardware components to follow the Moore's law, the solution adopted by the industry was to agregate more hardware resources to allow parallel processing. So instead of improving the processor's performance, the solution found was to integrate more processors into the systems. In this purpose, clusters and supercomputers appeared as possibilities to bring more computational power, by agregating more computers. However, to achieve such processing capabilities, clusters and supercomputers approaches present high energy consumption, their biggest disadvantage.

The most promising approach to provide both low power consumption and high processing power is to integrate several cores into one single chip, with multicore processors and heterogeneous architectures. The popularity of these processors naturally arose in the last years, becoming nowadays the standard of the industry even for personal computers, tablets and smartphones.

The problem is that, in general, the more sophisticated the processor is, the higher is its energy consumption. Furthermore, we achieved a level where this sophistication and consequent increased energy demand of these processors no longer brings extraordinary performance results, but only regular improvements. In an era where computers have achieved petaflop/s, and breaking the barrier of exaflop/s doesn't seem to be far, a special concern for power consumption raises, as it becomes more important to have a balanced ratio between processing and consumption power than only care about performance.

All this demand for very high processing capabilities comes at a price, and leads to a global concern about the power consumption related to these architectures. The use of simpler cores to perform on SIMD (Single Instruction Multiple Data) (FLYNN, 1972) model, along with a certain number of more sophisticated cores seems to be the natural tendency for the next years.

This approach of mixing devices with different levels of sophistication and different target niches has been characterized as heterogeneous computing, which is lately becoming very popular among programmers, and is one of the alternatives to achieve this balance ratio between performance and consumption. This technique consists in using different kinds of processors in a system, aiming to provide flexibility and better performances for applications running on these systems, generally improving energy management.

Among all the possibilities for heterogeneous computing, Graphic Processing Units (GPUs) and manycores stand out as the best solutions available on the market, providing the best processing power versus energy consumption ratios. General Purpose GPUs (GPGPUs) have efficiently been used for heterogeneous computing as well as manycores processors, with up to a hundred cores.

These devices used for heterogeneous computing certainly bring more processing power, but they also bring new challenges for programmers, which have to search for algorithms which can take full advantage of the hardware resources available on the system. When the gains in performance were achieved only by reducing the transistor size, it was not necessary to change the application's source code to see the result. But programming these new architectures requires from the programmers a new redesign of their applications to achieve the desired result.

As the industry has been capable of producing processors with hundreds of cores, it is now the programmers' responsibility to rewrite old applications and produce new ones for these extremely parallel devices, which is certainly not an easy task. However, this new way of thinking doesn't bring only problems, but brings also new opportunities for programmers to develop new skills and propose a whole new branch of solutions.

1.1 Objectives of this work

The focus of this work is the MPPA®-256 manycore processor, developed by the French company Kalray. This processor was developed targeting mainly high processing performance and low power consumption. This processor presents a parallel architecture with 256 processing cores targeting the embedded computing market, in applications such as transport, data security, image processing, telecommunications, network appliances, etc.

The objectives of this work are to study and implement a runtime system with high level functions, to facilitate the programmers' task when programming MPPA. In other words, the main goal is to provide some high level functions which will mask the low level API calls from the programmer. To achieve this goal, a study of the MPPA processor will be necessary, considering both hardware and software aspects.

The result of this study is to analyze the viability and complexity of implementing this runtime, containing some basic high level functions aiming to decrease the complexity of interacting with MPPA for new programmers. Being OpenCL and CUDA worldwide spread standards framework largely used for heterogeneous computing, the runtime will be developed aiming to have a programming model that keeps similarities with OpenCL and CUDA whenever possible.

This study is part of a scientific cooperation work between GPPD team (Grupo de Processamento Paralelo e Distribuído) of the Instituto de Informática UFRGS (Brazil) and Nanosim team, Université de Grenoble (France). The MPPA processor, which will be used to develop this work, is physically located in Grenoble, France. Meanwhile the tests will be realized accessing the machine remotely, from Porto Alegre, Brazil.

1.2 Organization

This work is organized in 6 chapters, including this introduction. In chapter 2 it is discussed heterogeneous computing, presenting the related architectures, examples of commercial devices and today's most popular development environments for this approach. Chapter 3 describes the MPPA processor architecture, highlighting some software and hardware aspects as well as its programming models, operational systems aspects and software development for MPPA. It is presented, in chapter 4, the scope of this work, defining its main purpose, the development of a runtime for MPPA processor and presenting aspects of the specification and implementation of the runtime. In chapter 5, we present an evaluation of the solution built, covering functional and complexity aspects. Finally, chapter 6 contains a conclusion about what was developed and possible future work.

2 HETEROGENEOUS COMPUTING

The term *Heterogeneous Computing* (KHOKHAR et al., 1993) refers to systems that have different kind of processors in their architecture collaborating to produce better results of the applications running in such systems. In other words, it is possible to describe a heterogeneous computing system by having in its architecture processors with distinct set of instructions, or more formally, processors with distinct ISAs (Instruction Set Architectures).

These processors can be, for example, CPUs (Central Processing Units), GPUs (Graphic Processing Units), DSPs (Digital Signal Processor), ASICs (Application-Specific Integrated Circuit) or even FPGAs (Field Programmable Gate Array). The most common and efficient heterogeneous architectures use GPUs though, being often called as accelerators. More recently, manycore processors, which also often mix different modules, have also being successfully used for heterogeneous programming, because of their flexibility and high processing power.

The main motivation behind the use of heterogeneous architectures relies in their commitment between processing power and energy consumption. In multicore architectures, the extra cores and cache memories required to fuel their instructions pipeline leads to a big silicon area and high power consumption, elevating the cost per computation. Clusters turn out to be a very expensive solution, as to achieve high processing power it is necessary to aggregate a considerable number of machines, which together spend much more energy than intrachip solutions.

The idea behind heterogeneous architectures is that even if one specific core is not nearly as capable as a sophisticated CPU core, there are so many cores available that together they can provide more calculating power spending less energy. Programming such architectures is not an easy task though, because each kind of processor will perform some part of the code better than the others, being the programmer often responsible for designating the tasks to the right hardware. The gain in performance comes mainly by exploring all the hardware resources available on the system.

The focus on the next sections will be at GPUs and manycore processors, highlighting the biggest differences between them and exposing aspects of their architectures, development environments available and contemplating some commercial examples of each one.

2.1 General Purpose Graphic Processing Units (GPGPUs)

In the early days of computing, graphic processing was consuming a lot of CPU time, which at the time was very expensive. Looking into the code of graphic processing algorithms it was very simple to identify these codes as being very easy to parallelize, because of its intrinsic characteristics. As spending CPU time wasn't the best deal back then and as the operations involved in this kind of applications were to be repeated several times for different data, the idea of creating a dedicated hardware became desirable.

Graphic Processing Units (GPUs) were born to free the CPU of the graphic tasks as a specialized hardware for this kind of algorithm. GPUs are strictly related to the SIMD (Single Instruction Multiple Data) paradigm, which means a same operation will be applied to lots of distinct data (one usual operation found in graphic processing is, for example, matrix multiplication). The idea behind this hardware is that you can take advantage of the application's parallelism by performing the same operation at the same time in different hardware resources. As the hardware was very specific, it was not expensive, and it could contain lots of replicated modules within a chip.

2.1.1 General architecture and GPGPU programming

In opposition to CPUs, which have big memories and control units, the GPUs are hardware containing mainly a same module replied several times, with very small memories, very simple control units and lots of arithmetic and logic units (ALUs), usually providing support for floating point operations, very common in graphic processing. Their high processing capability comes from the fact that they have much more processing elements than a CPU, being especially good at dealing with vectors and matrix. Figure 2.1 illustrates the connection between CPUs and GPUs, showing the differences in the architectural aspects of each module.

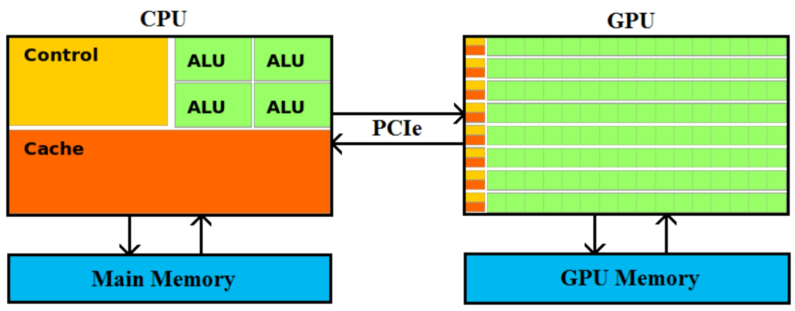


Figure 2.1: Connection between CPU and GPU.

Appearing in the late 90s, GPUs became popular in the beginning of this century, being very fast adopted as a standard among all computer fabricants in the market, which from now on would produce computers with separated CPU and GPU modules, usually interconnected by a PCIe (Peripheral Component Interconnect Express) bus allowing them to communicate. This bus is used by the CPU to send the control commands and blocks of data for the GPU, which by its turn executes the task received and send back by this same bus the result of the operations, as illustrated in figure 2.1.

When they first appeared, GPUs used to have a small memory generally shared between all threads executing on the device. However, more recent GPUs already have caches of L1 and L2 levels, aiming to improve the performance. Figure 2.2 exemplifies a generic memory model for a GPU, having 3 levels of memory: global, cache and local memories. Global memory is accessible for all threads, while a cache memory is shared only among some cores, gathered in processing groups (PG). Local memories (LM) are related to each core, a specific processing unit (PU).

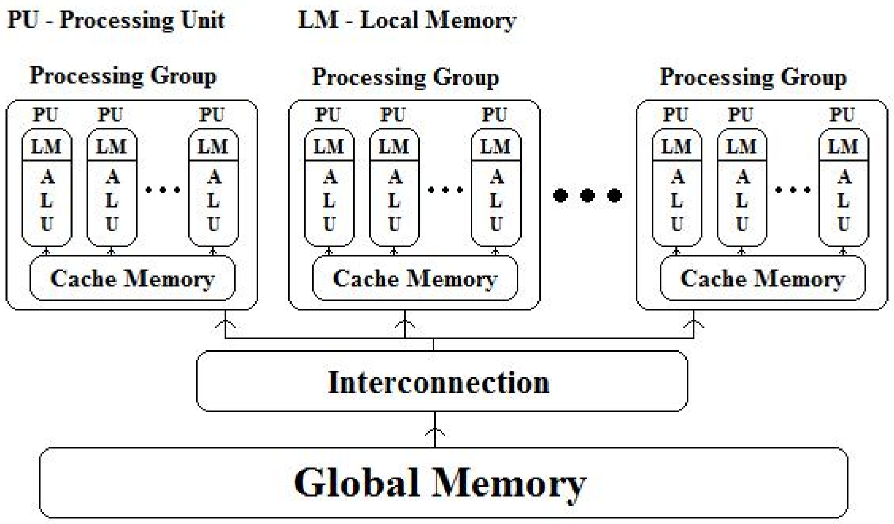


Figure 2.2: Generic GPU memory models.

Using GPUs was a huge improvement by the time they appeared, allowing the CPU time to be used mainly with effective processing, and letting the graphic processing tasks to be done by the GPU. As a dedicated hardware GPUs even improved computing time of applications by accelerating graphic processing. However, it didn't take too much time for programmers to realize that some parts of their programs were very similar to graphic processing, meaning that the same operation was performed over different data, several times. Iterative commands, such as *for* and *while* are often linked to such behavior.

At the same time, the advances achieved in some areas such as signal processing and real-time systems were demanding for very high processing power, meaning it was necessary to find a way to speed up computing. Taking all this into account, programmers came up with the idea of using GPUs for not only graphic processing but also general purpose programming, giving birth to what is called today as GPGPU (General Purpose Graphic Processing Unit) programming model.

In the last ten years GPGPU programming has become more popular among programmers. It suits perfectly applications with data parallelism, exploiting the SIMD paradigm, leading to a high throughput. In the beginning of the GPU era, these hardware used to be very simple, allowing no personalization and supporting only a few set of functions. Today, these architectures present a more complex instruction set with a high programmability inside their pipelines, making it easy to use GPUs for general applications.

Another advantage of using GPGPU programming is that GPUs usually present lower consumption when compared to multicore CPUs. As the hardware is simpler, naturally it consumes less energy than a CPU running the same task. That is an important characteristic of GPUs, as the power management is one of the main concerns nowadays.

When talking about GPGPU programming the concept of kernels is often used to refer to the parts of the code which actually execute on the GPU device. It is syntactically similar to a standard function, but differentiates from it because of some keywords that will indicate to the compiler to execute that specific function on the GPU. If we take a traditional loop as example, the kernel can be seen as the body of the loop, being the iteration process intrinsic. So when programming with GPUs, the programmer specifies the kernel and the data to loop over, being the compiler able to detect that part of code should execute in parallel in the GPU.

Nowadays, the most popular programming environments to program GPUs are CUDA (NICKOLLS et al., 2008) and OpenCL (MUNSHI, 2009), and the largest manufactures of GPUs are NVIDIA (NVIDIA, 2013) and AMD (AMD, 2013). While CUDA is a proprietary solution, developed only to be used with NVIDIA GPUs, OpenCL is an open source standard for heterogeneous programming, and can be used with any GPU. GPUs are still largely used for graphic processing in workstations, videogames, personal computers, and more recently it is possible to find it even on smartphones. Some of the most powerful supercomputers in the world also take advantage of GPU acceleration (TOP500, 2013).

2.1.2 Developing environments

The apparition of GPGPU programming brought also the need of developing programming languages which could express parallelism in a simple but still powerful way. In the beginning it was neither easy nor practical to transform the code for the older architectures to run in these new architectures. If programmers are unable to extract the applications' parallelism it will not show any improvement of performance when running in such architectures.

Some frameworks appeared to fill this gap, each one having some specific characteristic which differentiates one from the others. There is no right or better language to chose, it will strongly depend on the target architecture and on the application. Today's most popular frameworks for this purpose are CUDA and OpenCL, which will be presented in this section.

CUDA (Compute Unified Device Architecture): First issued in 2006, CUDA was introduced by NVIDIA. This parallel computing platform was invented with a new set of instructions and a new programming model allowing parallel processing to provide GPU programmers an efficient way to solve many complex computational problems they were facing.

CUDA is a proprietary solution targeting exclusively NVIDIA devices. With different instructions available, CUDA has been successfully used for GPGPU programming, performing applications such as cryptography and computational biology.

The idea when programming with CUDA is to divide the application into sub problems that can be solved in parallel, by grids (blocks of threads). If possible, it is also desirable that this sub problem could also be divided to be solved by a single thread. This way, each core contributes to the solution of the problem, cooperating with the others to maximize the throughput, even if the individual performance of each core is not the best.

CUDA's programming model involves the concepts of grids and blocks. To execute kernels, the task is divided into several threads, which execute concurrently. A kernel is executed as a block of threads, physically corresponding to a variable number of cores, which is given by the user as a parameter in the kernel call. CUDA can handle the number of cores/blocks available for processing by mapping each kernel to a block on run time, abstracting the process of delegating a task to a specific core/block. As soon as all threads of a block finish executing, a new block is launched.

From the programmer point of view, a grid is divided into equally shaped blocks, and blocks are composed by threads. A block has a unique identifier and all threads within a block execute the instruction specified in the kernel command sent by the host workstation. Threads in one block share the same local memory, and all threads have access to the global memory. Depending on the way threads, blocks and grids are grouped it is possible to have one-, two- or three-dimensional computing. This model is illustrated by figure 2.3, highlighting the hierarchy among grids, blocks and threads.

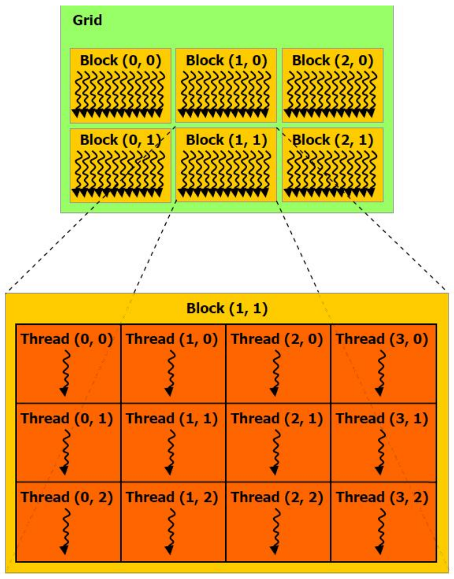


Figure 2.3: CUDA programming model.

CUDA can provide instructions for not only threads but also blocks. So when a task needs some kind of thread synchronization, it is desirable all threads to take approximately the same time to execute, to avoid a thread to wait for a long time. This is exactly the sort of instructions available in CUDA that help programmers with GPGPU programming. Besides of synchronization primitives and thread grouping, it has also instructions to manage the device memory.

CUDA is a very established platform, being compatible with the most popular operational systems, having a SDK (Software Development Kit) with a low and high level API, full support from NVIDIA and extension to standard computer languages such as C/C++ and Fortran. It is largely spread among programmers for GPGPU and heterogeneous programming, presenting generally good performance results. However, CUDA can be only used with NVIDIA hardware, not being possible to use it among heterogeneous platforms from other fabricants. Anyway, CUDA presents some very interesting aspects, and inspired the community to develop a similar solution, but which could be used for every platform. This solution is the OpenCL standard.

OpenCL (Open Computing Language): OpenCL (OPENCL, 2013) is a royalty-free standard first released in November 2008. It was started by Apple and it is maintained nowadays by the Khronos Group (KHRONOS, 2013), having some important companies collaborating with the OpenCL standard development, such as Intel, AMD and even NVIDIA. OpenCL is a specification latest released in July 2013 (2.0 release) and is already very popular among GPGPU programmers, being supported by most expressive GPU and manycore fabricants. OpenCL is also used for heterogeneous programming, because it can easily address heterogeneous platforms such as CPUs, GPUs and DSPs.

The OpenCL framework described in (GROUP et al., 2008) contains three components: a platform layer, a runtime and a compiler. The platform layer allows OpenCL to discover the device capabilities and to create an execution context, a very important task as it targets heterogeneous hardware. The runtime allows the host device to manipulate the context created. Finally the compiler creates executable programs which contain C commands and OpenCL kernels. In the Khronos Group documentation an OpenCL language is also specified, called OpenCL C, which is used to create kernels in the application's source code.

OpenCL programming model involves a host/device communication, being host and device not necessarily different hardware devices. A traditional OpenCL program contains a host and a device side. The host program executes on the host device issuing kernels to be executed in the target hardware. Kernels are the fundamental concept involving OpenCL programming and its execution model. They are sent by the host program and are then queued to be executed on the device, according to the resources available. This behavior is illustrated by figure 2.4.

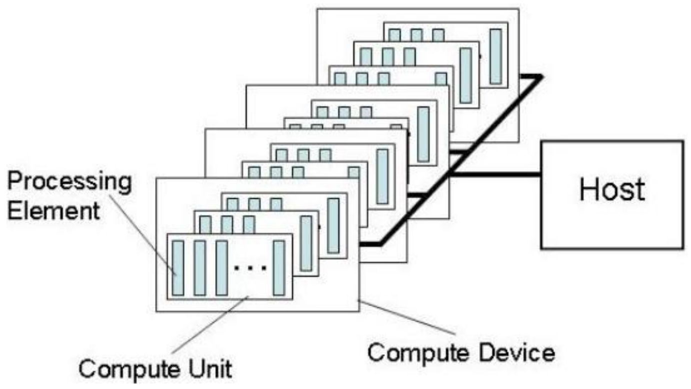


Figure 2.4: OpenCL execution model.

The host sends commands to one or more Compute Devices (CDs), which can be a CPU, GPU, etc. Each CD can be divided into Computing Units (CUs) and each CU can also be divided into Processing Elements (PEs). Kernels are split into uni-, two- or three-dimensional ranges that are called work-groups, which are mapped to Compute Units. Each processing element is one instance of a kernel and can be also called as a work-item, which belong to a work-group.

There are two main programming models for OpenCL: data and task parallel. In data parallel model, all work-items execute the same task over different data, operating in a similar way to the SIMD paradigm. This model is mostly used for programming GPUs, which are good with SIMD tasks. In the other hand, task parallel model works like a trapdoor to run arbitrary code from the command queue, allowing heterogeneous computing.

Figure 2.5 presents the OpenCL software architecture. The application communicates with the hardware through a set of tools including an OpenCL library, a Runtime, and an OpenCL device driver, which are involved in the process of building executable OpenCL code to run in a target device. The Platform Layer also helps in this process by providing information about the hardware to the application.

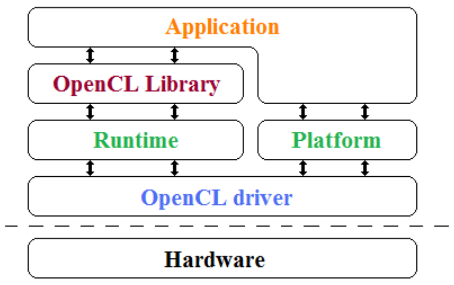


Figure 2.5: OpenCL software architecture.

In a higher abstraction level it is possible to describe the process of building and executing OpenCL program as follow. A source code is built, containing mainly C commands and kernels (OpenCL directives). The compiler generates the object code, which will be linked by a runtime library, building the executable file taking into account the target hardware. The executable contains both C and OpenCL directives, and will be loaded to the device memory to be executed.

OpenCL presents some similarities and some differences with CUDA. Their programming models are remarkable alike, as well as memory and execution model. Both work with host-device model, allowing memory management and similar set of instructions, presenting almost the same software architecture. This convergence is intentional, once OpenCL has a similar purpose of CUDA.

But they also have some differences. As an NVIDIA proprietary solution, CUDA targets only NVIDIA GPUs, while OpenCL was developed targeting heterogeneous platforms. Both of them have been successfully used for GPGPU programming, and some performance comparisons (SU et al., 2012) between them have been made lately. CUDA usually provides better results especially because of its specificity for NVIDIA devices, although OpenCL is more flexible, losing some performance to provide portability.

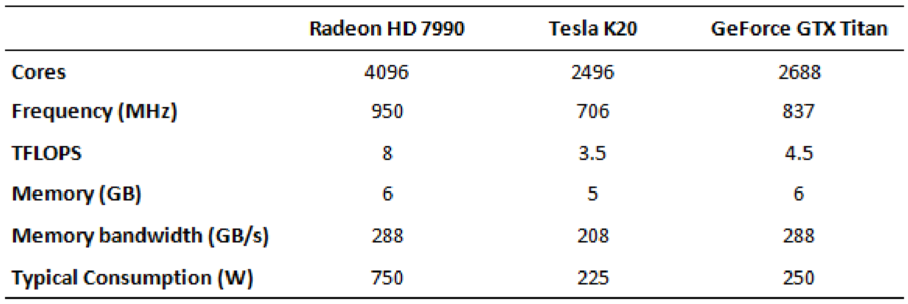
Pocl (Portable Computing Language): The OpenCL framework is a standard, meaning no real implementation is available. Being just a specification, it is necessary to implement OpenCL to run it on a new device. Companies such as ARM, IBM, Intel and NVIDIA have their own proprietary implementations of OpenCL. There is also an open source implementation of OpenCL, the Portable Computing Language (POCL, 2013a) with a MIT-license aiming to be an efficient OpenCL implementation and to be easily adaptable for new target devices.

In November 2013, the most recent stable release of the Pocl implementation is the 0.8 version, but continuous improvement are constantly been done. It uses Clang (CLANG, 2013) as an OpenCL front-end language, used to interpret kernels, and LLVM (LLVM, 2013) as back-end for the kernel compiler implementation, both also open source projects. Besides of providing an open source solution, the Pocl project aims to improve portability of OpenCL programs, avoiding manual optimizations depending on the target device, so if the hardware has a LLVM back-end, it should be easy to adapt Pocl to the target device.

Pocl can be used for heterogeneous architectures, and aims to exploit all kinds of parallel hardware resources, such as multicore, VLIW, superscalar and SIMD. The Pocl implementation does not yet fully supports the OpenCL standard specification, and there are some recognized bugs, but it is capable of successfully running several known examples (POCL, 2013b).

2.1.3 Commercial GPU examples

As GPUs are not a recent solution many different models are available in a market containing several fabricants. Among all GPUs fabricants, AMD and NVIDIA stand out as the most expressive companies with the biggest market shares. Both are extremely widespread for gaming and GPGPU programming, providing always a healthy competition for market dominance. AMD solutions include the well known family Radeon meanwhile NVIDIA most known families of GPUs are Tesla and GeForce. The GPUs models presented here are the most recent ones for each brand, by the actual date of October 2013. Figure 2.6 gives an overview of some of each GPU's parameters.



	Radeon HD 7990	Tesla K20	GeForce GTX Titan
Cores	4096	2496	2688
Frequency (MHz)	950	706	837
TFLOPS	8	3.5	4.5
Memory (GB)	6	5	6
Memory bandwidth (GB/s)	288	208	288
Typical Consumption (W)	750	225	250

Figure 2.6: Commercial GPUs compairison.

Radeon: First launched in 2000, the Radeon family is the most important GPU brand of AMD. The *AMD Radeon HD 7990 Graphics* model operates with a clock frequency of 950 MHz and 4096 stream processors, providing over 8 TFLOPS. There are 6 GB of DDR memory system available with a 288 GB/s maximum memory bandwidth. Because of its elevated processing capabilities, the requirements of the host system are very high, and the typical power consumption is also very high, about 750 W.
Tesla: The Tesla family, by NVIDIA, contemplates several different models of GPU, being present in the market for over 10 years. The *Tesla K20* model has 2496 processing cores operating in a clock rate of 706 MHz, achieving around 3.5 TFLOPS. It has a maximum memory bandwidth of 208 GB/s and 5 GB of total memory available on the board, while the typical power consumption is about 225 W.
GeForce: The GeForce GPU family, also fabricated by NVIDIA, is the biggest concurrent of AMD Radeon for gaming. The *GeForce GTX Titan* presents 2688 calculating cores, operating in a frequency of 837 MHz, providing 4.5 TFLOPS. There are 6 GB of DDR memory available, for a maximum bandwidth of 288 GB/s, being the typical power consumption around 250 W.

2.2 Manycore architectures

As the limits of hardware were being reached, the solution was to aggregate more than one core into a single silicon chip, the beginning of the multicore era, which we are still living. Multicore processors became very popular as soon as they started to appear in the market, but the market of computers doesn't restrict itself to personal computers. Some industries such as telecommunications, aerospatiale, transport, medical and data security often demand for very high processing, much more than personal computers can provide. For these niches it was necessary to provide solutions with more computational power, which can be translated into more cores. So instead of having processors with two, four or eight cores, why not having a processor with twenty cores, or even a hundred cores?

With this increasing numbers of cores in multicore systems, another terminology appeared to reference these massively parallel devices, the *manycore* processors. Another designation also used when referring to manycore can be *highly-parallel devices*, kind of describing the purpose of these processors, which clearly target parallel processing. These terms often refers to processors with at least dozens to hundreds of cores, presenting some aspects which differentiate them from multicore and GPU architectures.

They are characterized for having lots of cores, which are usually a middle term between the sophisticated CPUs and the simple GPUs cores, providing a good compromise between processing power and energy management. Most remarkably, manycore processors aim to maximize throughput, deemphasizing individual core performance.

2.2.1 General architecture

An important aspect of manycore architectures is the constant presence of a NoC (Network-on-Chip), responsible for interconnecting the different modules of the entire system. NoCs are structures used for intrachip communication as a solution to the problem brought by the amount of cores in the chips. As the tasks, even if parallel, tend to work in cooperation to solve problems, it is important to provide them some way to communicate.

The natural solution would be to interconnect all cores among a bus, but as the number of cores increase, it also increases the capacitance of the bus, leading to much more power consumption. Another parameter that also gets increased is the distances for the data to travel in the bus to reach all the receivers, keeping the bus busy for more time, leading to an important decrease of the communication efficiency.

NoCs are one of the most interesting solutions developed to avoid these problems. Importing concepts of computers networks and adapting it to the intrachip level, it is possible to provide an efficient and low power consumption alternative to system buses. In the other hand, as any network communication, it is vulnerable to high traffic or even to packet loss, depending on the way it is implemented.

It means the NoC plays a role in performance, when different modules need to exchange messages among themselves. The network's traffic, the protocol used and the routing algorithm are examples of the NoC's characteristics which might influence in the performance. Some variants in NoCs implementation, such as statics or dynamic routing algorithms, full or half duplex links, multicast and broadcast support and switching algorithms are examples of parameters which will certainly make a difference in the network performance. Another factor which plays a role in performance is the topology of the modules.

A very flexible and fluid way to map threads strongly differentiates manycore from other architectures. Another characteristic of manycores processors is that, because of the amount of cores, they usually present some special hierarchy among the processors. The way this hierarchy is built may vary from processor to processor, but it helps to better divide the work to be done into the cores.

Of course manycore processors aren't always the best choice to run an application. Depending on how the application is built, it may have a better result in a single or dual core processor, for example. Manycore processors usually operate in a lower frequency than single cores. It means that it is pointless to have such hardware resources available to not spend it carefully, taking as most advantage as possible of each core.

2.2.2 Developing environments

As manycore architectures are recent in the market and the fact that each processor usually presents some unique characteristics, programming them is not very easy. The alternatives to program these architectures are either to adapt one existing language or to use some feature provided by their fabricants.

Some fabricants, knowing the difficulties of programming them, also develop a set of tools to help programmers to develop their codes. But this solution leads to a non portable source code, meaning that when changing the target hardware, it is necessary to redesign the application. The other alternative is to adapt known languages and frameworks for this purpose. Some languages such as TBB, Cilk/Cilk Plus, Dataflow languages and Smyle OpenCL have been used to program manycore architectures and are briefly presented here.

TBB (Threading Building Blocks): The TBB (PHEATT, 2008) is a C++ template library developed by Intel, aiming to facilitate the task of writing code for parallel architectures, such as multicore and manycore processors. It provides both data structures and special algorithms to avoid some intrinsic complications of parallel programming. It treats the threads as tasks, easily dynamically allocating them to a free core to be performed, respecting graph dependencies and abstracting these low level details from the programmer. An automated management of the processors cache is available to improve performance.

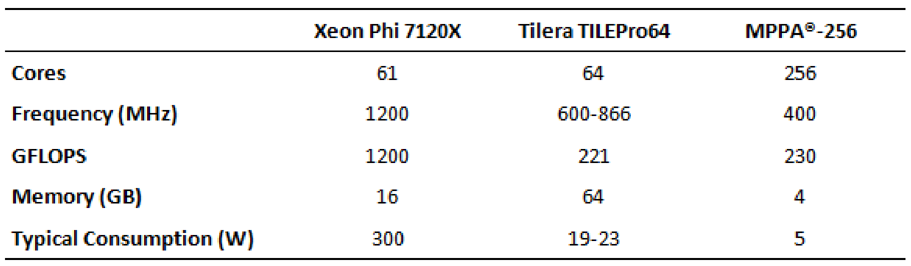
Cilk/Cilk Plus: The Cilk (BLUMOFE et al., 1995) programming language, nowadays developed also by Intel but first started by MIT, was designed to be used for multithreaded parallel architectures targeting general purpose programming. It is in fact an extension of the C and C++ languages, being based in the ANSI C standard language and the Cilk Plus version inspired in the C++ language. The programmer is the responsible for exposing the application's parallelism, by using two basic primitives: *spawn* and *sync*. The *spawn* primitive indicates some task can be executed in parallel without any problem, while the *sync* primitive is used for synchronization. As C/C++ extensions, Cilk/Cilk Plus allow programmers to easily add parallelism to their application's source code, not requiring significantly changes.

Dataflow: The Dataflow (UUSTALU; VENE, 2006) programming paradigm changes the way programmers are used to build applications by modeling the program as a directed graph of the data flowing between operations. In Dataflow oriented languages, an operation can run as soon as all inputs are available, providing this way parallelism to the application. A very important concept in this model is the idea of state, being a sort of portrait of the actual system situation, as the program progress over the graph is achieved by moving from one state to the next. Several languages are available to program in Dataflow paradigm, such as SigmaC, SISAL, Quartz Composer, LUSTRE, etc.

Smyle OpenCL: The Smyle OpenCL framework (INOUE, 2013) consists in an independent implementation of the OpenCL standard, targeting embedded multicore and manycore systems, developed at the Japanese Ritsumeikan University. The biggest difference founded in this implementation comparing to the OpenCL standard is that here the threads and objects are statically mapped to the architecture's cores, doing it at design time instead of at runtime, aiming to minimize the runtime overhead. The framework developed was tested and runs fine for the tested applications. However, it is not yet fully tested for heterogeneous architectures and a complete set of applications, being more a research than a commercial solution. The target device of this study was the SCC (Single-Chip Cloud Computer), an Intel experimental processor.

2.2.3 Commercial manycore processors examples

Most of manycore processors available nowadays are used mainly in researches, such as the Polaris processor, developed by Intel, which integrates 80 cores in a single chip. But it is possible to find some commercial manycores, such as the Intel solutions Larrabee (SEILER et al., 2008) and Xeon Phi (CHRYSOS; ENGINEER, 2012). The Tile64Pro, manufactured by Tilera (TILERA, 2013), together with the MPPA processor are also good examples of commercial manycores available nowadays in the market. Differently from the GPUs market, very well established and with big players, manycores are still an open market, full of possibilities. The models presented here are the most recent ones for the mentioned brands, by the date of October 2013. Figure 2.7 gives an overview comparison between the different commercial manycores.



	Xeon Phi 7120X	Tilera TILEPro64	MPPA®-256
Cores	61	64	256
Frequency (MHz)	1200	600-866	400
GFLOPS	1200	221	230
Memory (GB)	16	64	4
Typical Consumption (W)	300	19-23	5

Figure 2.7: Commercial manycores comparison.

Larrabee: Larrabee is a hybrid processor developed by Intel, being considered as a first draft for manycore architectures. It is considered as a manycore processor, but keeps lots of similarities with GPUs, including a special module for processing 3D graphics. It is built with a 32 nm technology, with a variable number of x86 cores integrated in the chip, depending on the model (from 8 to 48 cores), operating at the clock rate of 1 GHz, producing around 1 TFLOPS.

Xeon Phi: Intel MIC (Many Integrated Core Architecture) is a manycore computer architecture designed by Intel, which incorporates some previous work of the Larrabee project. The great advantage of the MIC architecture is the compatibility it presents with previous Intel devices (Xeon processor's family), something rare in the manycore world. The Intel products that actually implement the MIC architecture is named Xeon Phi. The *Intel Xeon Phi 7100 series*, more specifically the *7120X* model integrates 61 cores into the chip. It was conceived using a 22 nm technology, operating with clock rate around 1.2GHz providing 1.2 TFLOPS. The maximum memory bandwidth is about 350 GB/s for a RAM memory of 16 GB and 30 MB of cache, with typical power consumption of 300 W.

TilePro64: Tilera is a semiconductor company specialized in producing scalable many-core processors. Among its solutions one of the most expressive is the *TILEPro64*, integrating 64 RISC general purpose processors, each of them having its own cache memory and implementing cache coherence. They are interconnected through a mesh network linking all the 64 cores, being each a node of the network with a non-blocking router. The communication with outside modules can be done by DDR memories, PCIe, Ethernet and flexible I/O interfaces (which can be software-configured to handle several protocols). The processor is fabricated with a 90 nm technology, operating with a frequency between 600 MHz to 800 MHz and running GNU/Linux. The typical power consumption of the processor is about 19 W to 23 W.
MPPA®-256: The MPPA®-256 (DINECHIN et al., 2013) processor produced by Kalray, a French company specialized in developing manycore processors for high performance applications, was built with a CMOS 28nm technology and integrates 288 VLIW (Very Long Instruction Word) cores in a single chip, having 32 cores dedicated to resource management and 256 calculating cores, dedicated to computational processing. The cores are divided into 16 clusters of 16 processing elements, totalizing 256 cores. Inside a cluster, a shared memory paradigm is used, allowing inter process communication.

MPPA operates with a clock rate of 400 MHz and can provide 700 Giga operations per second and 230 GFLOPS, when extracting the most of its 256 cores. A 4 GB global memory is available, with 32 MB of internal memory, besides of L1 and L2 caches of each processor. Also the improvement in power consumption when running typical applications, comparing to other processors, is about 10 times, having a 5 Watts typical consumption.

2.3 Final Considerations

Knowing the difficult of efficiently developing software for these new parallel devices, the Heterogeneous System Architecture Foundation (HSA Foundation) (FOUNDATION, 2013) was created. HSA Foundation is a non-profit consortium formed by some of the most expressive hardware developers in the world, such as AMD, ARM, Texas Instruments, and more. Their goal is to forward the industry progress in heterogeneous architectures to prepare the programming tools to address these architectures, making it easier for programmers to build parallel applications. They work in building a standard for heterogeneous computing that could provide high application performance at low energy cost. Their solution is to produce an interface for parallel computation supporting a diverse set of high level languages used for general purpose programming, targeting heterogeneous platforms such as CPUs, GPUs, DSPs and manycores.

The apparition of such organizations such as HAS Foundation demonstrates the difficulties that have been faced in the late years to produce a portable solution capable of abstracting hardware details without losing too much performance. The manycores market together with heterogeneous computing concepts present a gap of programming tools and languages. For some, this gap might be seen as a problem, but for others it might also be seen as a great opportunity for developing new skills and new tools.

The target processor of this study is the MPPA architecture, which is a manycore processor presenting relevant similarities with GPUs. It is desirable to offer MPPA programmers a library of functions that could allow them to easily develop code. In cooperation with the Grenoble team this work aims to expand the possibilities of programming MPPA by mixing the concepts of manycores, OpenCL, Pocl and heterogeneous computing.

3 MPPA®-256 PROCESSOR

The MPPA®-256 manycore processor was conceived and produced by the Kalray society (KALRAY, 2013b) as a solution to high performance computing with low power consumption, being software programmable. Kalray is a French company specialized in developing manycore processors for high performance applications.

Knowing the difficulties usually involved in programming manycore architectures, especially a totally new one, and also the problems in porting applications' code to new target devices, Kalray provides a whole set of tools to facilitate MPPA's learning experience. Kalray's global solution includes not only the processor itself but also a SDK (Software Development Kit), designed boards to MPPA and tools to simulate, execute, trace and debug applications. These tools allow the user to maximize the performance of their applications, improving MPPA's time to market.

3.1 MPPA hardware architecture

The MPPA processor is composed, in a macro level, by four Input/Output subsystems, 16 computing clusters and a NoC (Network-on-Chip), responsible for connecting these elements. There are four Input/Output subsystems, one in each side of the chip, which are referenced as north, east, west and south. They are fully responsible for any communication with elements outside the MPPA processor, including the host workstation. Each I/O subsystem has 4 cores, totalizing 16 cores dedicated to PCIe, Ethernet, Interlaken and other I/O devices. The 4 cores in a same I/O subsystem share the same address space, and all of them are directly connect to 2 clusters by the NoC. Figure 3.1 shows a global view of MPPA hardware architecture, where *C* represents a cluster.

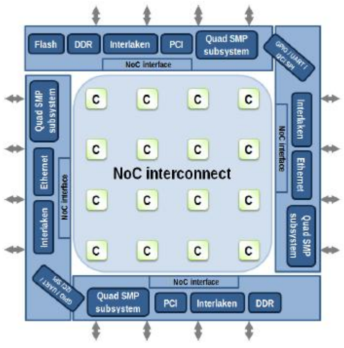


Figure 3.1: MPPA general view (KALRAY, 2013a).

MPPA is connected to a host CPU by a PCIe interface, similarly to a GPU. The host, which is actually an x86 CPU running GNU/Linux, sends code to be executed on the MPPA processor through the I/O subsystems modules. The MPPA then executes the received task and write the output data in a 4GByte DDR3 RAM memory, which is connected to an I/O subsystem and can be accessed by the host CPU. Each core in the I/O subsystem has its own 8-way set associative instruction cache memory, with 32KB. They have also a data cache memory, of 128KB, shared between the four cores.

The communication between clusters and I/O subsystems is assured by two high speed low latency NoCs (Network-on-Chips), one used for data transmission (the D-NoC) and the other is used for control (the C-NoC). The D-NoC presents a high bandwidth as it transfers data between the clusters and I/O nodes, while the C-NoC presents a low bandwidth. It was designed this way to avoid controls commands to be stuck because of data transfers, which tend to be more frequent. These NoCs have a 2D torus topology, integrating the 16 clusters and the 4 I/O subsystems. Each cluster has a special core for communication that is associated with a node of the NoC as well as each core in the I/O subsystems, totalizing 32 nodes (16 clusters plus 16 I/O nodes), disposed as shown in figure 3.2. One important point to highlight concerning the NoCs is that they provide for all clusters a direct link with two I/O subsystems, including the insider clusters.

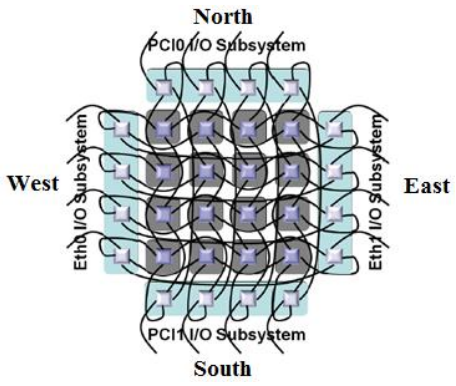


Figure 3.2: NoC's architecture (DINECHIN et al., 2013).

MPPA has 16 clusters organized in a matrix of 4x4, totalizing 256 calculating cores. Each cluster has 16 calculating cores, called processing elements (PE), and 1 resource manager (RM), also called *system* core, used for control and communication, besides of a local shared memory. This memory is a 2MB low latency cache for instructions and data, and it is shared by all the 16 cores and the RM core in a single cluster, which is a huge advantage of MPPA processor, enabling a high bandwidth and throughput between PEs on a same cluster. Each PE has their own 2-way set associative L1 caches for data and control, both with 8KBytes. For simplicity, the MPPA processor does not implements cache coherence between L1 caches of its PEs. Figure 3.3 gives an overview of cluster's structure.

One important characteristic found in the MPPA processor is its homogeneous approach considering its 288 cores (256 calculating cores, 16 resource manager cores and 16 I/O subsystem cores), which are all 32 bit VLIW processors. All of them have the same architecture, including the ones used for resource management, with seven pipeline stages for high power efficiency. It was a particular choice of Kalray experts because of its deterministic execution time and low power consumption compared to others possible choices.

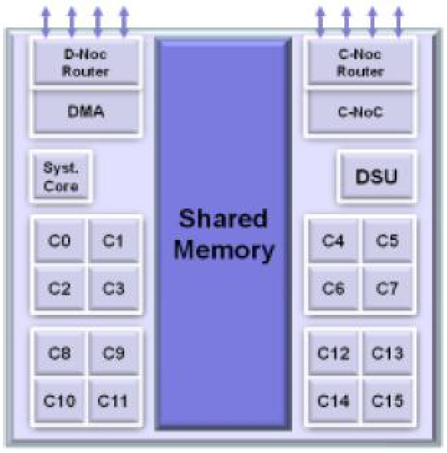


Figure 3.3: Cluster internal structure (DINECHIN et al., 2013).

3.2 MPPA software architecture

Besides of hardware aspects, some software aspects are also important to understand how MPPA processor works. Each core runs its own operational system and depending on which module this processor belongs to (I/O node or clusters), it can run a RTEMS or NODEOS operational system. Both are modified Linux operational systems and are compatible with the POSIX threads (DREPPER; MOLNAR, 2003) library, allowing programmers to use pthreads calls without any restriction.

The RTEMS (Real-Time Executive for Multiprocessor Systems) operational system runs on the cores belonging to the I/O subsystems. This operational system is used mainly for interface management, once it is helps in the communication between host and MPPA device, but also between I/O node and clusters.

The NODEOS (Node Operational System) runs on the RM core, the one belonging to the clusters. The main goal is to help with the real execution of the task by the PEs in each cluster. This operational system is very light weight, once it is stored on the cluster's 2MB local shared memory, along with data and code from the programs executing.

Besides of these operational systems running on the MPPA processor, we also have the native operational system running on the host machine, which is a GNU/Linux. Kalray provides some APIs to allow the different operational systems to interect between them. These APIs will be detailed later on this chapter.

3.3 Programming models

For programming MPPA processor, there are two possible approaches: dataflow and shared memory programming models. Each one of them presents some particularities, being impossible to say if one is better than the other. Because of their specific characteristics it will depend on the application we want to build and the level of abstraction we search to determine which approach suits better the situation.

3.3.1 Dataflow programming model

One of the possibilities when programming MPPA processor is to use the dataflow programming model. This model is characterized for modeling the program as a graph that specifies the data flowing, being a very different way to program from the imperative and object oriented languages that most programmers are used to. The MPPA dataflow programming model fully supports data and task parallelism. One example of programming with dataflow approach in MPPA processor using the Sigma C language is described in (AUBRY et al., 2013).

For Dataflow programming model, Kalray has developed a set of tools available on a graphic IDE (Integrated Development Environment) based in C/C++ called MPPA ACCESSCORE. It integrates a compiler, a simulator, a profiler and a debug platform that allows programmers to develop parallel applications without any special background on the hardware architecture, providing even power consumption information. As Kalray provides strong suport on the dataflow programming model, it is out of the scope of this work.

3.3.2 Explicit parallel programming model

Another way to program MPPA processor is using the explicitly parallel programming model with shared memory paradigm that allows a multithread programming. Using this model, there are new possibilities that can be used to program MPPA, such as OpenMP and POSIX threads. Both approaches are fully supported in MPPA, being up to the programmer to decide which one suits better his application.

OpenMP (Open Multi-Processing): OpenMP (DAGUM; MENON, 1998) is an API for shared memory multiprocessing programming. It is managed by the OpenMP ARB technology consortium (ARB, 2013), a nonprofit organization formed by several important hardware and software industry, which have the role of defining, producing and approving the new versions of the OpenMP specification. It is a portable and scalable model aiming to provide programmers flexible interface for developing parallel applications for multi-platforms, being possible to use C/C++ and Fortran for programming.

OpenMP in MPPA performs a *fork-join* model, where a master thread is responsible for creating a group of threads that will execute in parallel. After each thread completes its task, they synchronize and end (they *join*), leaving only the master thread executing again. MPPA is already prepared to run OpenMP code and no special feature is needed to do it. The only restriction for OpenMP is the maximum number of 16 threads per cluster, not being possible to manage threads creation.

POSIX threads: The pthreads (POSIX threads) libraries are a standard thread based API for C/C++ widely used in multithread applications. The programmer can write programs in C/C++ and use the GCC compiler with some special directives to build the executable code. In this programming model, it is recommendable for the programmer to have some knowledge of the processor's architecture, to take advantage of the locality factor.

The programmer has total access to the architecture with the MPPA APIs allowing low level calls. With these APIs it is possible, for example, to send a task from the I/O node to a specific cluster or to create a communication channel between clusters and I/O subsystem. Also inside a cluster it is also possible to use the pthreads library without any restriction, creating as many threads as desirable, and using elements of pthread library such as the commands join, exit, semaphores, conditional variables and more.

MPPA fully supports POSIX pthreads programming, where a process run in a cluster and threads run on cores belonging to this cluster, sharing the cluster's memory (the 2MB memory). Inter-processes communication is done through the NoC, using the NoC Inter Process Communication library. It is possible to have different codes running on each cluster of MPPA, allowing heterogeneous computing.

When developing code for MPPA processor using the POSIX threads library it is necessary to split our code into two parts called master and slave. This is necessary exactly because we have different operational systems running on different modules. The master code runs on the I/O subsystems while the slave code runs in clusters.

As this model uses a memory shared paradigm, all cores in a cluster share the same address space, leading to some intrinsic difficulties due to the model, such as false sharing (TORRELLAS; LAM; HENNESSY, 1994) and cache coherence which may also apply when programming MPPA processor. It is entirely up to the programmer to be aware of such characteristics and take it into account when programming.

Programming with pthreads libraries for MPPA can be a little tricky. As the application's code is written in C/C++, it is totally natural for programmers. However, some tasks such as sending code to execute on a cluster, passing data between clusters and synchronizing, for example, claim for reasonable knowledge of the APIs' functions and how to use it, making it difficult in a first moment.

3.4 MPPA's execution model

Executing applications built with the POSIX threads explicit parallel programming model in MPPA brings different possibilities concerning the platforms to be used and the possibility of using a SIMD and MIMD (Multiple Instruction Multiple Data) (FLYNN, 1972) programming paradigms. It is possible to use MPPA to run in all cores the same task exploring the SIMD paradigm. But this is a well known approach that GPUs often obtain better results, because they are specialized hardware to operate under these conditions. In this work the focus will remain in the possibility of using MPPA with the MIMD paradigm, allowing heterogeneous computing. Over the next paragraph it will be discussed about MIMD programming in MPPA, as well as compilation aspects and communication connectors.

3.4.1 MIMD execution model

MPPA processor can easily suit the MIMD paradigm with its execution model. As the code is divided into master and slave parts, it is likely to have more than one slave in the application, being possible to have different instructions being executed at the same time in different PEs of each cluster. The master source file is unique for each application, and will be responsible for managing the slave side and for some processing. But the slave side can be composed by more than one file. In other words, the slave side can be seen as functions, which are sent to the clusters to be executed.

Being possible to send the same slave code to different cluster, or to send different slave codes to each cluster, the compiled source file is composed by only one master binary and one or more slave binaries, being called multibinary because of this behavior. The structures of the master and slave files are very similar, both having their own list of libraries, functions, a *main* function, exactly as a C/C++ source code. The difference is that slave side will typically contain mainly calculating operations while master side will typically contain more control and communication commands.

The master-slave model of programming MPPA leads to an intrinsic need of providing means for different modules to communicate. The master program will need to send data to the slave, which will perform some operation and send the result back. This process can happen as many times as necessary during the execution of an application.

The general structure of an MPPA program using the POSIX thread model is very simple. The master usually creates all the features it need for communication, divide the work by sending slaves to be executed on the clusters, and wait for them to return the result. The slave side will also initialize the features it needs for communicating, create threads to perform the task, waits for all threads to finish, and send back the result. This is a very generic way to describe MPPA programs, but it often apply to most applications, with some minor changes.

Data can be transmitted during the creation of a cluster process, through the parameters sent with the *mppa_spawn* function. But it is often desirable to be able to send and receive data after sending the task to a cluster, not having to wait until it finishes. For this purpose the MPPA APIs provide programmers connectors and read/write functions to be used with them, allowing such transfers to be done. The different connectors available on MPPA will be detailed later. Inside a cluster, it is also possible to create threads and split once again the workload. Thread management is fully supported using the MPPA APIs.

3.4.2 Compilation and execution aspects

Focusing on the explicit parallel programming model, and more specifically in the POSIX threads libraries, there are some relevant information to be discussed for this work about the compiler and execution tools of MPPA. Aspects of the compilation and how the application is actually executed in the processor are discussed in detail in the next paragraphs.

After writing the code of the application, naturally, it is necessary to compile it. To facilitate the programmers' adaptation and keep it portable, Kalray choose to use the GCC compiler (STALLMAN, 2002), which is a widely known standard compiler. GCC (GNU Compiler Collection) is an open source compiler of the GNU project, which played an incredible role in free software development. The advantage of using GCC is that almost all programmers are pretty familiar with it, being not necessary to dedicate any time in a new compiler.

Kalray provides a special makefile file that can be used by MPPA programmers, being necessary only to adapt it with your application name and files, and choosing the right options as desired, making it very easy to use. The programmer may also choose to use his own makefile file. The file structure is very similar to a generic makefile, with minor changes. Flags to indicate whether the code should be compiled for RTEMS or NODEOS operational systems are necessary. It is important to include the MPPA tool chain files, which contain the directives of MPPA APIs, libraries and all the needed information to be linked with the binary files. To facilitate the programmer task, it can also include the directives to run the application on MPPA or in a simulator.

The Kalray makefile file already contains the call to a linker that is responsible for linking all the sources, master and slave(s) into one single binary file, called multibinary. With the applications source code compiled and the multibinary file ready, it is possible to launch this program to be executed on the supported platforms.

Kalray provides the possibility of executing MPPA applications in a simulator, which will closely emulates the behavior of MPPA. It allows multiuser executions, meaning each user of the MPPA workstation can launch applications to run in the same time. When using the real processor, one of the I/O subsystems is used together with the DDR memory, responsible for charging the application to be executed.

3.4.3 Host-MPPA communication

To send a task to the MPPA processor it is necessary to do it through the host workstation, being necessary to provide the means for them to communicate. The communication between host and MPPA is done with the help of drivers (host and MPPA side), an API with low level functions and some connectors. More specifically, the MPPA side is represented by the I/O subsystem, which is the module responsible for dealing with host communication. Physically, it is done by a PCI Express interface, very similarly to GPU connection.

There are two connectors, Buffer and MQueue, available for making this link and they are manageable by the API functions. The communication between two systems always involves a sender or transmitter and a receiver, being always unidirectional. To facilitate the reading, for now on the transmitter process will be called *Tx* and the receiver process will be called *Rx*.

Buffer: The Buffer connector is in fact a buffer in the *Rx* process where a number N of *Tx* processes have access to write. A trigger helps the *Rx* process to know when something was written in the buffer. This trigger can be set with the help of the API functions, and is the only mean the *Rx* processes has to be aware of new data available.

MQueue: The MQueue connector implements a message passing interface between one *Rx* process and one *Tx* process. It is possible to set the size of the queue, being the maximum number of messages that can be queued avoiding message loss. The size of the message can be also defined by the programmer.

3.4.4 I/O Subsystem – Cluster communication

The communication between I/O subsystem and clusters is assured by a set of tools. The C-NoC and the D-NoC are used as the physical path between them. To link the physical world with the software, some connectors were developed by Kalray, are they: Sync, Portal, RQueue and Channel. The difference between these connectors and the ones presented in the previous section is that for I/O subsystem-cluster communication, the C-NoC and D-NoC are used, something that not happens with host-MPPA connectors.

Together with an API providing low level functions, these connectors allow the communication cluster-to-cluster and cluster-to-I/O subsystems. Each feature connects with the NoCs in a special way, creating different modes of communication, for distinct purposes.

Sync: The Sync connector is used exclusively for synchronization and consists in a 64 bit word in the Rx process accessible for a number N of Tx process for reading. It allows one process to know the state of the other process to perform the synchronization. It is implemented using only the C-NoC.

Portal: The Portal connector is in fact a memory area in the RX process where a number N of Tx processes have access to write. When the communication is established, it is necessary to set a trigger, so the Rx process can be aware when some Tx process makes a write operation in the memory area. Only the D-NoC is used in this connector, and the RX process needs to perform asynchronous read operations.

RQueue: The RQueue connector implements a queued message passing communication. It is possible to set both the size of the queue and the size of the messages exchanged, being the maximum size of the message 120 bytes. It links one Rx processes to a number N of Tx processes and both NoCs are used. A system of credits in the Tx processes is implemented to managing the blockage of the process if necessary.
Channel: The Channel connector links one specific Rx to one specific Tx process, being a used for both communication and synchronization. Both D-NoC and C-NoC are used, being often used to provide a rendez-vous behavior between both processes. It is possible to use asynchronous operations in the Rx process.

The choice of which one of them should be used will depend on the application and how it communicates with other modules. Also, using one of them doesn't eliminate the possibility of using the others, being possible to mix them in a same application.

3.5 MPPA communication APIs

The MPPA processor involves different operational systems and different hardware resources in each module. The host-device model implies that host and MPPA processor need to communicate, running different OS, through some physical feature (PCIe, Ethernet, etc), but also some software support is needed, to treat hardware events.

Once the host sends to MPPA the application to be executed, it is received in the MPPA by the I/O subsystems, which often need to send it for the clusters, to be calculated there. Inside a cluster, it is also necessary to provide tools for the different PEs to communicate and to ensure the creation and management of the different threads.

It means that these different modules need to communicate and not always use the same mechanisms for doing it. Being aware of that, Kalray developed a set of functions to provide all necessary features for MPPA programmers. These functions compose three different APIs, totalizing more than a hundred functions including memory management, message passing, threads management, and more.

The *PCI Express API* assures the communication between host and MPPA. The *Process Management and Communication API* (PM&C) provides the means for I/O subsystem and clusters to communicate. And finally, the *NODEOS API* supplies the clusters with the necessary tools for assuring threads management. Figure 3.4 illustrates the different APIs and where they fit in MPPAs architecture.

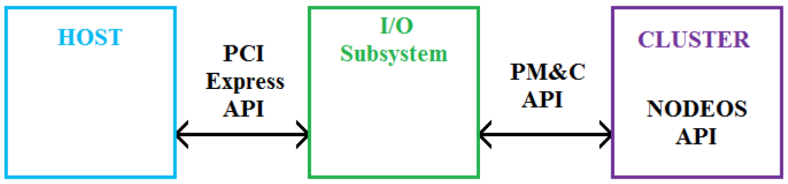


Figure 3.4: MPPA and different APIs for programming it.

3.5.1 PCI Express API

The PCI Express API allows the MPPA processor to communicate with the host workstation. As the applications are launched from the host, it is necessary to provide the tools for programmers to manage memory access, synchronization, etc. The feature responsible for assuring this communication is composed by three different modules: a host-side device driver, a host-side low level communication library and a MPPA communication library.

The host-side device driver allows direct memory accesses by the host on the 4GB memory that the I/O subsystems also have access, through a set of special primitives for memory management by handling interruptions. The host-side low level communication library is responsible for masking the primitives used by the device driver to implement a message passing interface for communication, memory management functions, to boot and to shutdown MPPA, etc. The MPPA communication library is very similar to the host-side low level communication library, being the differences occasioned by the necessity that MPPA has of allocating memory. In fact, the device drivers used in MPPA and in the host are different, leading to different libraries for managing it.

This API has around 30 functions for the tasks of: initialization and boot of the system, host-MPPA message passing, dealing with errors, loading and starting applications, multiple devices support, synchronization, message queue management (MPPA and host sides) and memory transfers. Being this API the one responsible for launching an application from the host to the MPPA processor, it is also responsible for launching an application to the simulator. To assure the API works fine, it is necessary to install and load the device drivers to the host and MPPA, besides of linking the host and MPPA sides with the respective libraries.

3.5.2 Process Management and Communication API

The Process Management and Communication API is responsible for process creation and for assuring the communication inter-process. It is based on the classic POSIX threads libraries with synchronous and asynchronous operations, adapted for MPPA environment, including the NoC and PCI features.

The master side of the application's code runs on the RTEMS operational system. In the other hand, the slave side of the application's code runs on the NODEOS operational system. It implies in different address spaces, different primitives, different system calls and interruptions, etc. It means that these two distinct operational systems do not automatically communicate between them. But master and slave codes need to communicate, at list two times in an application: for launching the salves in the clusters and for synchronizing when the slaves finish executing.

Often, the applications exchange messages with data and control between master and slaves, being necessary to create a communication mean among them. Or, even sometimes, it is necessary to communicate with another cluster, to cooperate for solving a task. These are some practical examples programmers face when programming parallel architectures. The Process Management and Communication API aims to provide the necessary tools to assure this communication.

This API was built following some guide lines to simplify and to make it as natural as possible for programmers. It was decided to stay very close of the POSIX inter-process communication, a well known standard. The communication between different modules is assured by a sort of pipe, identifying a receiver and a sender process. Each pipe is designed to be either read only or write only, being always an unidirectional link. The communication features available for intercluster communication were presented in sections 3.4.3 and 3.4.4. These communication features are typically used for creating software barriers, data exchange, massage passing and synchronization.

The API defines types and functions that can be used for both RTEMS and NODEOS systems, totalizing over 35 functions in the API. For process management, this API provides the *mppa_spawn* function, one of the most important functions, being used in the master side of the application to create multi-clusters programs. Some other functions to manage the processes identifier are also available, such as in the POSIX libraries.

The rest of the functions provided by the API are mainly used for synchronous and asynchronous communication between clusters and I/O subsystems. Read and write operations can be performed, for all different communication features, being also possible to set triggers and signals, deal with errors, etc.

3.5.3 NODEOS API

The NODEOS API was designed to run on the clusters, and to be as much similar as possible with the POSIX thread standard library, implementing the well known *pthread_* functions. The API contains over 60 functions used for setting threads attributes, thread management, thread mutex (mutual exclusion), thread conditional variables, semaphores, synchronization and time operations.

The 16 PEs responsible for actually executing the slave code sent by the master share the 2 MB memory available on each cluster, meaning all of them share also the same address space. This leads to a concern about memory management inside the cluster, being one of the functions of this API to provide the necessary tools to outline these problems. POSIX standard mutex, conditional variables and semaphores are available to be used carefully by programmers to avoid all sorts of memory problems, such as race conditions.

The thread management contemplates all functions for creating, finishing and joining threads. As all PEs share the same memory, the thread creating process is very light in terms of system requirements. After the *mppa_spawn* function is called in the master side, the slave code passed as an argument of the function will be loaded in the PE 0 of the cluster, chosen to execute the main thread of the slave side. If none thread is created, the slave code will perform entirely in the PE 0. However if a thread is created in the slave side, it will be sent to be executed in the first free PE. Figure 3.5 illustrates this behavior.

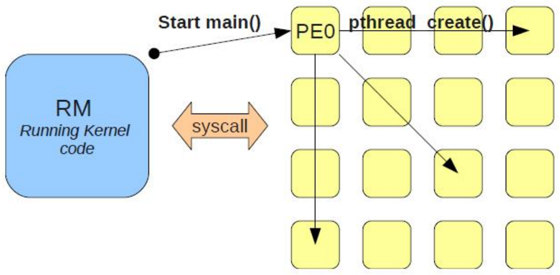


Figure 3.5: PEs using the POSIX threads standards (KALRAY, 2013a).

3.6 Final considerations

Programming MPPA processor using the POSIX Threads programming model is, in most cases, not obvious, being necessary to study its hardware and software architectures, to understand the expected behavior. The details involved in this task, such as different operational systems, master-slave model, inter-modules communication APIs, take time to learn, making it harder to start writing code for MPPA rapidly.

Even writing a simple *Hello World* application may take a while to understand the necessary structures and API calls the programmer has to make to assure everything works fine. Going deeper on programming MPPA is even more challenging, as new features and possibilities may appear, transforming it into a complex task. Even if the API functions were developed following the industry standards, as the number of functions overpass a hundred and most of them have a considerable number parameters, it turns out that programming MPPA with the POSIX threads explicit parallel programming model is not easy at all, despite the flexibility it provides.

Being aware of the difficulties in starting to program MPPA, the main goal of this work is to provide a higher level interface, freeing the programmer the task to go deep into the details of the hardware and software architectures providing a simpler and faster way to develop code for MPPA. The purpose and implementation of this solution are discussed in the next chapter.

4 SPECIFICATION AND IMPLEMENTATION OF A RUNTIME FOR MPPA

The MPPA manycore processor is a very recent architecture, with great potential and also very promising for high performance parallel computing. However, as any other massively parallel architecture, if the programmers are not able to fully use the hardware resources available, writing optimized parallel applications, it won't present any special gain of performance. Instead of forcing users to learn a new language or a new programming paradigm, it is desirable to provide them the opportunity of choosing the language and paradigm they feel more comfortable, and which may possibly be the language their applications are already written, being only necessary to port them.

As the OpenCL framework has being successfully used to program some multicore and manycore CPUs and for GPGPU programming, it will then serve as an inspiration to develop a runtime for MPPA, aiming to help programmers to develop code for MPPA. This runtime will fit the explicit parallel programming model, more specifically the POSIX threads model. OpenCL software architecture and execution model are intended to serve as guide lines for the architecture of the runtime. The objective is to remain close of OpenCL standard in terms of abstraction and functionality.

The main motivation is given by the fact that the POSIX threads model used together with C/C++ are a world-wide spread standard that most programmers, if not learned programming with it, are at least accustomed to it. But programming it in MPPA may not be as easy as it is for other architectures, relying on the numerous and essentials MPPA APIs' functions. The main goal is then to build a runtime library, with a few number of functions that could mask these details from the programmer and help them to develop code for MPPA with only few information about the architecture and MPPA APIs.

It is intended, as the expected results of this work, to verify the limitations and difficulties to build such library and to evaluate the complexity decrease of programming MPPA with this library instead of using MPPA API's calls directly.

4.1 Runtime specification

The runtime to be developed has to be capable, in a first moment, of loading tasks to MPPA, in the I/O subsystems modules and set up the communication features needed for communicating with the other modules. After the master code is loaded and executing in the I/O subsystem, it is desirable to send the slave codes to be executed on the clusters, with data and parameters. Once the slave code is received in the clusters, it should also bind the communication features with the ones created in the master and passed as a parameter to the slaves. Once the slaves' codes are executing, it is important to provide the means to create and manage the threads. During all the execution, master and slave need to be able to communicate among themselves. Finally, master and slave have to synchronize, aggregating the results computed by each module.

The activities mentioned in the previous paragraph cover all the main tasks every single application running on MPPA often has to do. To provide these basic functionalities we were able to synthesize these common tasks into five main actions, and this runtime should implement the following functionalities:

- Initialization and communication set up;
- Launching a task in a cluster;
- Assuring communication between master and slaves;
- Redistribute this task into the cores belonging to that cluster;
- Final synchronization to send the results back.

4.1.1 User library function requirements

This work will be then based on the five actions mentioned, which contemplate the most usual API calls when programming the MPPA processor. Of course complex applications might use other MPPA API calls, more specific ones, but it is not a goal of this work to exhaust all functions and built a complete library, but only to analyze the most essential functionalities, serving as a proof of concept. The five actions presented in a macro level in the previous section are here explained in a MPPA point of view, with more details.

The initialization action should be capable of creating all the communication features for clusters to communicate with the I/O node and prepare the execution context for the task execution. It must create all the structures and variables needed, letting the clusters ready to receive a task and perform it.

Next action, for launching a task in a cluster, should provide the users all the support for launching a job to the MPPA clusters initialized with the previous function. It will send the name of the function that will be executed in the given cluster, besides of some arguments, including data and necessary information for the clusters to perform correctly. It must be possible to indicate which clusters to send the task.

As different modules of MPPA are involved when executing an application, it is necessary to provide them the means to communicate and exchange data and controls. The communication action should be responsible for providing all the necessary features to assure this connection between master and slaves.

Once the task is sent to the cluster, it is now desirable to have it divided into smaller pieces, which can be executed by each PE inside the clusters. The next action should be responsible for assuring this behavior, providing all the necessary features to create the threads responsible for calculating each little piece of the whole task. It is similar to the launching cluster action in terms of parameters. As far as possible, it is also desirable to choose which PE the function would be executed.

Finally, master-slave synchronization is desired to allow all different clusters to send back their results and permitting the program to finish correctly. After sending the tasks to be executed on the clusters, the master code waits for the results from each cluster, creating a point of synchronization in the code. After the results are received, the program can properly finish.

4.1.2 Runtime architecture

The runtime developed will be accessible for the final users to build their applications. The user can then make calls to the runtime, but it can also make directly calls to the MPPA APIs. Once the objective was not fully cover all functions in the APIs, only the most common ones, it will mask some of the APIs calls, but some more complex applications might need to use the full APIs' specification.

As the runtime covers all the usual tasks a MPPA application have to perform, it is entirely possible to build applications only using the runtime library, not being necessary to use the other API calls, which is the case of the examples that will be presented latter in this work. This way, the runtime is an intermediate feature between the operational systems that have access to the MPPA processor and the application itself. The runtime architecture is presented in figure 4.1, illustrating this behavior.

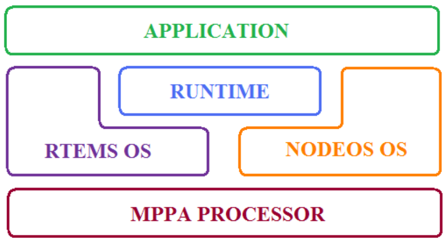


Figure 4.1: Runtime software architecture.

4.1.3 Runtime API specification

From the five main actions identified in the requirements, it was possible to synthesize it into 9 functions that will be the visible for the user. They are the interface for the runtime API. These 9 functions are presented in figure 4.2, and are detailed in the next paragraphs.

```
[1] void mppa_read_portal (portal_t *portal, void *r_buffer,
        unsigned long r_buffer_size, void *destination_buffer, int rank)

[2] void mppa_write_portal (portal_t *portal, void *w_buffer,
        unsigned long w_buffer_size, int rank)

[3] void init_cluster (rtems_t *master_fd, void *r_buffer,
        unsigned long r_buffer_size)

[4] void mppa_launch_cluster (rtems_t *master_fd, int rank,
        char *cluster_executable, char *cluster_args)

[5] void sync_clusters (rtems_t *master_fd)

[6] void init_kernel (nodeos_t *slave_fd, char *argv[], char **master_args,
        void *r_buffer, unsigned long r_buffer_size)

[7] void launch_kernel (nodeos_t *slave_fd, pthread_t *th_id,
        int core_no, void *function, void *param)

[8] void* sync_thread (pthread_t th_id)

[9] void close_portals (nodeos_t *slave_fd)
```

Figure 4.2: Runtime API functions' headers.

The functions available in the runtime API can be divided into three main types of functions: communication, master and slave functions. The communication functions are used by both the master and slaves sides, while master and slave functions are used only in the master and slave side, respectively.

Communication functions: the communications functions are the ones responsible for assuring the communication between master and slave(s). They are the functions number 1 and 2 in figure 4.2. In the communication, a buffer is used to allow master and slave to interact.

The *mppa_read_portal* function is responsible for reading data from a buffer. It takes as parameters a descriptor of the communication feature used, the associated buffer and its size, the destination variable and the identifier of the transmitter. It is used always when module wants to perform a read operation.

The *mppa_write_portal* function is responsible for the other side of the communication, by writing data in the buffer. It is used to any time a module needs to write something to another module. It takes as parameters the descriptor of the communication feature, the buffer associated and its size as well as the identifier of the module we want to write in.

Master functions: the master functions are used in the master side mainly to prepare the context, to launch task to clusters and to allow different clusters to synchronize. They are the functions number 3, 4 and 5 in figure 4.2.

The *init_cluster* function is responsible for initializing the master side of the application. It should create the necessary features for communication as well as preparing the execution context in the master side. It takes as parameters a descriptor of the master instance and a pointer to the buffer that will be used for communication during the execution, as well as the buffer size.

The *mppa_launch_cluster* function is responsible for launching tasks to be executed on clusters. It takes as parameters the descriptor of the master instance, the identifier of the clusters we want to send the task, a pointer to the function to be executed on that cluster and the arguments we want to send.

The *sync_clusters* function is responsible for synchronizing the different clusters in execution, performing a barrier. It takes as a parameter the descriptor of the master instance allowing to identify the clusters in use.

Slave functions: the slave functions are the ones used in the slave side. They are mainly used for preparing the context in the slave side, creating threads of execution, synchronizing those threads and properly finishing the slave instance. They are the functions number 6, 7, 8 and 9 in figure 4.2.

The *init_kernel* function is responsible for preparing the execution context in the slave side. Its task is to receive the arguments sent by the master and creating and binding the communication features. It takes as parameters the descriptor of the slave instance, a pointer to where store the master arguments and information (pointer and size) about the buffer to be used.

The *launch_kernel* function is called to create threads of execution in the slave. This function is used to divide the work into the different cores available in the slave side. It takes as parameters the slave instance descriptor, a pointer to the thread that will be created, the identifier of the core we want to launch this thread in, the name of the function to be performed and finally the arguments to be sent for the new thread.

The *sync_thread* is used to synchronize a thread when it finishes executing. It takes as a parameter only a pointer to the thread we want to synchronize. It is called each time a thread ends executing, so it can properly finish.

The *close_portals* function is responsible for closing all the connectors created for master-slave interaction. It should free all the memory allocated for this communication, as it will no longer be necessary. This function should be called at the end of the slave code, allowing it to properly end.

4.2 Runtime implementation

With a well defined purpose, the next step in the project consists in implementing the runtime library discussed in the previous sections. This step consists in taking all the requirements stated transforming it into code, picking the right data structures, functions and types to be used in the library. It may also be necessary to adapt some requirements to fit in MPPA programming models and the MPPA API's functions available, but always trying to stay as close as possible from the specification.

Next sections will discuss about the communication features used as well as the data structures, explaining the choices made. The library implementation, detailing a bit each of the functions is also presented, highlighting important aspects of MPPA limitations and possibilities.

4.2.1 Programming environment

All the applications used for tests during the development were written in the C programming language, using only standard C commands and MPPA API calls. They where first developed in C, then ported to the MPPA processor with all APIs calls and then adapted to use the developed library. Only standard C libraries, such as *stdio*, *string*, *assert*, *time* were used. The POSIX threads library used is in fact inside the MPPA libraries, *mppaipc.h* and *mppa/osconfig.h*, which contain the MPPA APIs functions, constants and types' declarations.

The runtime to be implemented targets C/C++ programming with the explicit parallel POSIX threads programming model of MPPA. It will contain only standard ANSI C code and MPPA's APIs calls, not being necessary any other feature than it already is to compile and execute it in the processor.

4.2.2 Communication features

MPPA presents distinct possibilities for the different modules to interact between themselves, are they: MQueue, Buffer, Sync, Portal, RQueue and Channel. These features were presented in detail previously in this work, and here our interest relies mainly in the last four features, which are used for I/O subsystems and clusters communication.

The Sync feature is the most simple, and because of it can't be used to exchange data, being only a 64-bit word, used for synchronization. The RQueue feature implements a message passing interface, with limited size of message (128 bytes), which is a huge disadvantage for this approach, once the idea is to be able to exchange data. It is most indicated to exchange controls, which tend to be small messages. Finally the Channel feature links one specific receiver process to one specific transmitter process, which would imply in creating and managing different structures for each module, with an unnecessary overhead.

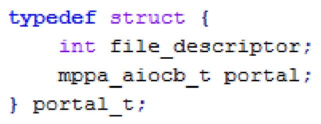
The Portal feature was in fact the most simple and the one that best fit the requirements of this project. The choice of using it instead of any other structure relies on the fact that it can multiplex several transmitters' processes to a single receiver process. Another important point is that it actually allocates a memory area for the communication, being possible to set the size of this area depending on the application, which is a really interesting approach. Also, for being a memory area, it can be simply seen as bytes in a low level, allowing the user to decide what type of data he wants to exchange.

After some tests, the portal feature turned out to be also very simple to use, but still flexible enough to fit the requirements. Choosing it means all the functions in the library use this kind feature, but it does not prevent the final user of building an application which uses other communication feature, as they can coexist in a same program.

4.2.3 Data structures

As the library masks some important information from the final user, it is necessary to store it somewhere, so when it is needed in another part of the program it may be available. Some data structures were designed to help to concentrate the required information for each module avoiding also the library functions to contain too many parameters, making it harder to use. Three data structures were built to help programming this interface: one for the portal communication, one for describing the master side and one for describing the slave side.

The first structure was created to concentrate the needed information of the portals. Each time a read and write operation is done it is necessary to inform which is the portal it should read from or write in. When creating portal communicators, each of them has a sort of identifier that is used latter to designate the right portal for each operation. The *mppa_aiocb_t* and the *file_descriptor* field allow to correctly identifying the portal structure targeted. Those are values that are returned by the portal creation functions and are lately used by the read, write and close portal functions. The structure is presented in figure 4.3.



```
typedef struct {
    int file_descriptor;
    mppa_aiocb_t portal;
} portal_t;
```

Figure 4.3: Portal struct.

Both processes, the slave and the master, have some characteristics which need to be used to describe them and store important information about them, which might be frequently used. To allow such behavior, two structures of data were created to serve as descriptors of the processes there are related, one for each side of the code. As the master code runs on the I/O nodes, running the RTEMS OS, and the slave code runs on the NODEOS OS, the names of the structures are: *rtems_t* and *nodeos_t*.

The *rtems_t* structure is used to describe the master process, and contain the necessary information which will be used by the runtime library functions. It contains four fields: *pid*, *r_portal*, *w_portal* and *clusters*. The pid field is a vector of *mppa_pid_t*, which is the value returned by the *mppa_spawn* API call when spawning a cluster. Its size is 16 (one for each cluster) and it stores the pid of the spawned process. The *r_portal* and *w_portal* fields store the information about the read and write portals, respectively. Only one read portal is necessary, as all clusters write on it, but an array of 16 write portals is necessary, once it is necessary to identify which cluster we are going to write in. Finally the clusters field is used to control which clusters are being used or are available, also being an important field for the synchronization of the different clusters.

```
typedef struct {
    mppa_pid_t pid[CLUSTER_NO];
    portal_t *r_portal;
    portal_t *w_portal[CLUSTER_NO];
    int clusters[CLUSTER_NO];
} rtems_t;
```

Figure 4.4: RTEMS struct.

The *nodeos_t* structure is used to describe the slave process, and it is a little bit simpler than the master process. It has in fact three fields: *rank*, *r_portal* and *w_portal*. The *rank* is the number of the cluster that specific slave is running in. The *r_portal* and *w_portal* have the same function of the ones in the master side, holding the necessary information about the communication features. In the slave side only one write portal is necessary, once the I/O node is the only node we can write in.

```
typedef struct {
    int rank;
    portal_t *r_portal;
    portal_t *w_portal;
} nodeos_t;
```

Figure 4.5: NODEOS struct.

4.2.4 Functions implementation

In this section we detail the implementation of each function, highlighting the most important aspects of each one of them. To provide the 9 functions that are visible for the user in the runtime API interface, it was necessary to develop 6 more functions that are not visible for them, but perform important tasks to allow them to correctly be implemented, totalizing 15 functions. The headers of the additional functions that are not visible are presented in figure 4.6.

```
[10]  portal_t *mppa_create_read_portal (char *path, void* r_buffer,
              unsigned long r_buffer_size, int trigger)

[11]  portal_t *mppa_create_write_portal (char *path)

[12]  void mppa_close_portal (portal_t *portal)

[13]  void set_path_name (int rank, char *path)

[14]  int set_cluster_id (int rank)

[15]  void copy_buffer (void *r_buffer, unsigned long r_buffer_size,
          void *destination_buffer, int rank)
```

Figure 4.6: Additional functions.

Communication functions: these functions are responsible for creating the portal communication between I/O nodes and clusters. They are also responsible for reading and writing in the desired buffer. All the additional functions are used for the communication.

The *mppa_create_read_portal* function (number 10 in figure 4.6) is responsible for creating a read portal, by associating a portal structure to a buffer. To create a portal it is necessary to specify its path, which contains information of which modules are involved in the communication. It allocates the memory for a portal structure, and opens the uni-directional channel to receive data. It sets a trigger used for notifying the receiver about available data in the buffer and it is also responsible for binding the created portal to the D-NoC, which will be in the interaction. It returns the portal descriptor created. This function masks 6 MPPA API calls.

The *mppa_create_write_portal* function (number 11 in figure 4.6) is used to create a write portal. Creating a write portal is simpler than creating a read portal and it is only necessary to send the path as a parameter, as it contains the information about the modules involved in the communication. It creates the portal structure allocating memory and binding it to D-NoC. This function masks 2 MPPA API calls.

The *mppa_read_portal* function is visible for the user, and is responsible for reading from a buffer. It is necessary to inform which portal we want to read from, by sending the right portal descriptor as parameter. The portal descriptors are stored in the structures that describe both master and slave instances. It is necessary to send the pointer of the buffer associated to the communication as well as its size. The function copies the data in the buffer to a variable, also passed as a parameter. The rank parameter allows to identify which cluster we have to read from. Internally, the identifier of the cluster is used as an offset to the buffer, positioning the reader in the right place. Basically this function waits for the trigger and read the buffer when data is available, masking 2 MPPA API calls.

The *mppa_write_portal* function is also visible for the user and is responsible for writing something in the buffer. As in the read function, it is also necessary to specify the cluster's identifier, so the write pointer is positioned in the right place for writing in the buffer. When calling the function, it is necessary to specify the portal we want to write in and the buffer which actually contains the data we want to write. This function is responsible for 1 MPPA API call.

The *mppa_close_portal* function (number 12 in figure 4.6) is used to properly close a portal structure. It is responsible for freeing the allocated memory for the structures and guarantying the communication channel returns the right value when finishing, indicating it ended well. This function masks 1 MPPA API call.

The *set_path_name* function (number 13 in figure 4.6) is used to set a path name based on the module involved in the communication. This function generates a unique path for each module and for each type of portal (read and write). These values need to be unique as they are the identifiers of the communicators used in the previous functions to create the portals. The path received as a parameter is a standard path that will be modified according to the rank, to produce the unique path. It returns a string which contains the path for each module.

The *set_cluster_id* function (number 14 in figure 4.6) is responsible for setting a unique identifier for each cluster, given its rank. It is used by the function *set_path_name* to create the unique identifier path.

The *copy_buffer* function (number 15 in figure 4.6) is very simple, only used for memory copping. It performs a single instruction which copies a variable to another. It is used in the read portal function to copy the area in the buffer we want to read to the variable. This function is responsible for managing the offset in reading, using the rank parameter for such purpose.

Master functions: the master functions are responsible for creating the portal structures for communication and for launching and synchronizing clusters.

The *ïnit_cluster* function is responsible for initializing the barrier structure, setting all clusters to available. It will also be responsible for creating the read portal for the master side. As we don't know yet which clusters will be used, it is not yet necessary to create the write portals.

The *mppa_launch_cluster* function is responsible for launching a slave to be executed on a cluster. The cluster's identifier is passed as a parameter, and it is necessary to spawn the slave in that cluster. The function tests whether the cluster identifier is within the range allowed (0 to 15) and if the cluster is available. If it is possible to spawn the slave to that cluster, the function creates the arguments to pass to the slave. In the arguments it is necessary to send, besides of user's arguments, the cluster number and the path of the master portal to be linked in the slave. It spawns the slave and sets that cluster to unavailable, finishing by creating the write portal for that cluster in the master side. This function masks 2 MPPA API calls.

The *sync_clusters* function is responsible for performing the barrier that will synchronize clusters. It will search in the array for the clusters in use and will wait them to finish. By the time each cluster finishes, it sets the cluster to available again, and closes the write portal. Once all clusters have finished, it closes also the read portal in the master side. This function masks 1 MPPA API call.

Slave functions: the slave functions are used in the slave side and have almost the same functionality of the master functions, only adapting it to the slave context. They will create the portal structures needed to provide the communication with the master and also for managing threads creation.

The *init_kernel* function will be responsible for, in a first moment, receiving the master arguments and parsing it (as it contains in fact three parameters – cluster identifier, master path and master arguments). It will be also responsible for creating the read and write portals used in the slave side.

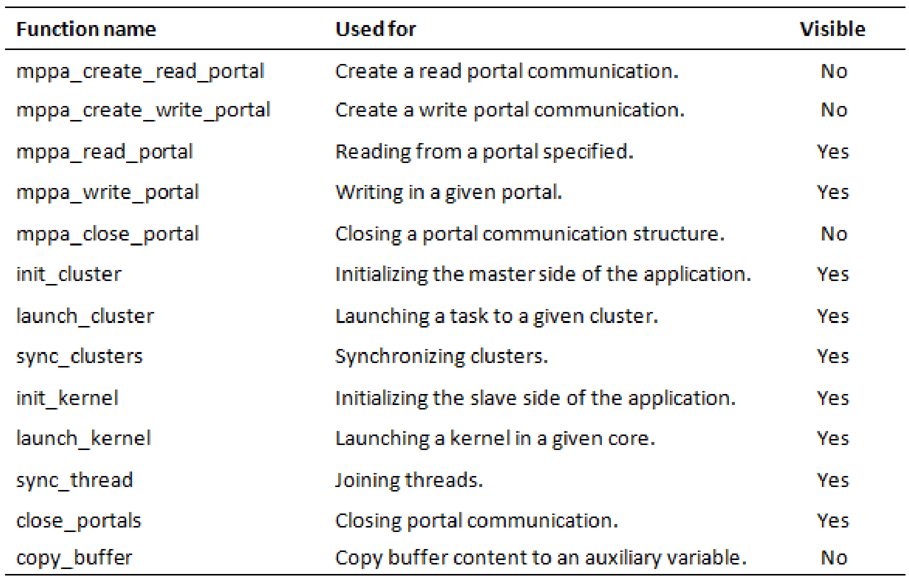
The *launch_kernel* function is called each time the user wants to create a thread. The number of the core we want to use for that thread is passed as a parameter. After verifying it contains a valid value (0 to 15), the function is responsible for setting the *pthread_affinity* property to spawn it to the right core. It is done by using a mask of 16 bits, each bit indicating one core. We set to '1' the bit in the mask corresponding to the core we want to use, and to '0' all the other bits. For example, to create a thread on core 4, the mask used will be '0000000000010000', corresponding to the decimal number 16. The function will try to create the thread in the target core using the *pthread_create* primitive, but if it is being used a special error message is sent by the core and is treated by the function, sending it to the next available core. If all cores are in use, the function sleeps for 0.1 seconds and tries again. This function masks 6 MPPA API calls.

The *sync_thread* is used to synchronize a thread in the slave. As the user was not directly responsible for creating the thread, it should also not be directly responsible for joining the thread. This function is used only to perform a *pthread_join* call, masking 1 MPPA API call.

The *close_portals* function is necessary because it is not possible to close the portal structures when joining the threads in the slave, as it is done in the master side with the *sync_clusters* function. It is due to the fact that joining the threads does not end the slave side, as the main thread continues to execute and might want to send and receive data to and from the master. It closes read and write portals and allows slave to properly finish.

4.3 Final considerations

We were able to fully implement the specification with the help of some additional functions invisible for the user. The final API containing 9 functions fits all the requirements and provides all the functionalities defined in the five actions. A recapitulative of the developed functions is presented in figure 4.7, with a brief description.



Function name	Used for	Visible
mppa_create_read_portal	Create a read portal communication.	No
mppa_create_write_portal	Create a write portal communication.	No
mppa_read_portal	Reading from a portal specified.	Yes
mppa_write_portal	Writing in a given portal.	Yes
mppa_close_portal	Closing a portal communication structure.	No
init_cluster	Initializing the master side of the application.	Yes
launch_cluster	Launching a task to a given cluster.	Yes
sync_clusters	Synchronizing clusters.	Yes
init_kernel	Initializing the slave side of the application.	Yes
launch_kernel	Launching a kernel in a given core.	Yes
sync_thread	Joining threads.	Yes
close_portals	Closing portal communication.	Yes
copy_buffer	Copy buffer content to an auxiliary variable.	No

Figure 4.7: API functions.

The functions implemented for the runtime library are responsible for masking over 20 MPPA API calls, synthesized into 15 functions, where only 9 are visible for the user in the runtime API interface. Each of these function not only replace the MPPA calls, but also reduces the number of lines in the user's code, as each of them do a lot of internal control such as tests, assertions, etc. The structures implemented also facilitate the user's task, as all the important information is centralized in one single structure.

Besides of simplifying the common tasks, the runtime library hides from the user some controls such as managing buffers, setting pthread affinity attribute to run on the target core, managing arguments exchange between master and slave, etc. These tasks may also require some effort from users when programming MPPA and are offered by the runtime library.

5 EVALUATION

After defining the requirements for the runtime library and implementing it, it is necessary to use it in some applications and test it. In this chapter it will be described the parameters used to evaluate our solution as well as the applications tests made to validate the library functions and its performance. Finally, an evaluation of the solution proposed is done based on the results obtained for each application, taking into account the relevant aspects for this work.

5.1 Methodology and evaluation criteria

The methodology used to evaluate the correctness and the performance of the library developed is to execute and compare some parameters of two different versions of the same application: one version using only the MPPA API calls and one version using the runtime library. These versions will be referred in the text by *Kalray API* and *Runtime API*, for the version using only the direct MPPA API calls and for the version using the runtime library developed, respectively. The parameters picked to help to evaluate the proposed solution are: functionality, execution time and number of functions.
Functionality: it is important to make sure that the output of both versions of the application is the same. It means to evaluate whether they are functionally equivalent or not. This parameter will permit us to be sure the runtime correctly performs its tasks, not influencing in the result.
Execution time: as using the developed runtime library implies in using another software layer between the application and MPPA APIs, another parameter involved in evaluating the solution is the possible overhead caused by this new layer in the software architecture. In other words, it is necessary to measure the time the application using the runtime library takes to run and compare it to the application which does not use the library, allowing to see the overhead caused by the developed library.
Number of functions: the number of functions parameter aims to evaluate the decrease in complexity involved in the proposed solution. A smaller number of functions implies less previous knowledge on the details of the processor, such as API specific calls, hardware and software architectures, etc. We will be looking at the number of specific MPPA functions necessary to perform each version of the application, which allows us to have an idea of how much learning effort would be required for someone to build that application. All functions which are not from the standard C language will be counted. In the original application, these functions are the MPPA API calls, while in the runtime library we have developed they are the library API calls.

These parameters serve as touchable comparators between the two versions of the same application, allowing us to analyze and conclude about the viability of this implementation. They will also help us to estimate the decrease of complexity brought by the runtime library functions.

All the results that involve time are an average value from 10 different executions. The number of executions used was 10, since the values barely varied, allowing a certain confiability.

5.2 Application examples

To validate the correctness and effectiveness of the developed function, and verify if it was really doing what it was meant to do, some applications were needed. It was necessary to choose applications with some meaning, but at the same time not too much complex, as the objective of these tests were not the application's performances itself, but the fact that the library could work fine and also analyzing the complexity decrease of programming these applications to MPPA using the library.

The applications choose to test the library are: one application for calculating π with the Monte-Carlo method, one application to calculate the Mandelbrot set and finally one for calculating a low pass filter. It was important to pick different applications to test the flexibility of the solution proposed, instead of restricting it to one single solution.

5.2.1 Calculating π by the Monte-Carlo method

One of the applications that were used to validate the library works fine was the method of Monte-Carlo to calculate the number π. This application can be easily parallelized to produce a more precise result, as a bigger number of points leads to a more homogeneous random distribution.

The method of Monte-Carlo for calculating π consists in a statistic solving problem method, based on random numbers. The principle is to inscribe a circle of radius R inside a square with side length of 2*R, both shapes having their center in the origin. The area of the circle will be π * R^2. The area of the square will be (2 * R)^2 = 4 * R^2. So the ratio between the areas of the circle and the square is π/4.

It means that if you generate N random numbers within the square, (N *π)/4 of them tend to be also inside the circle. If you know the total number of points (N points), and the number of points inside the circle (M points), it is possible to calculate the number π as follow:

$$\pi = (4 * M)/N \tag{5.1}$$

Figure 5.1 illustrates the behavior of the algorithm, where the random points are generated and spread in the square. The points inside the circle (to simplify, we take radius R=1) are marked as red, meanwhile the ones outside the circle are marked as blue. To simplify, you can use only positive numbers, within the first quadrant, which will provide the same result.

With a number N big enough to assure a good probability distribution, this method can provide a good estimative of the π number. It is possible to know whether the generated point is inside or outside the circle by calculating and analyzing the distance of this point to the origin, with the help of Cartesian theorems. If bigger than the radius, it is outside the circle; otherwise, it is inside the circle.

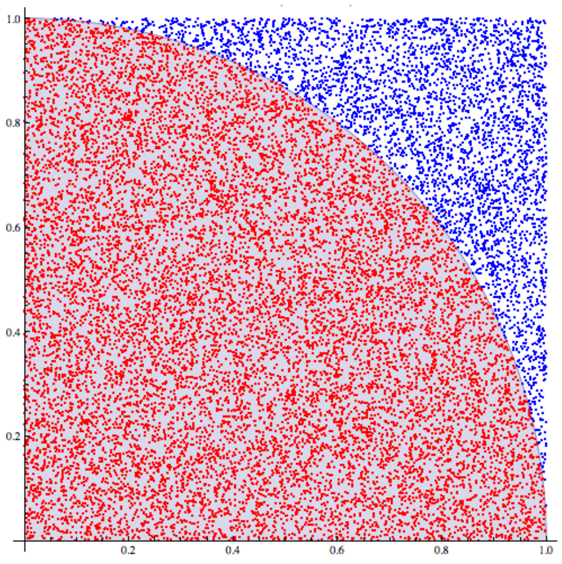


Figure 5.1: Monte-Carlo method for calculating π.

Four versions of this application were built: one using direct MPPA API calls and focusing on threads, one also using direct MPPA calls but focusing on clusters, one using the library and focusing on thread parallelism and finally one using the library focusing in cluster parallelism.

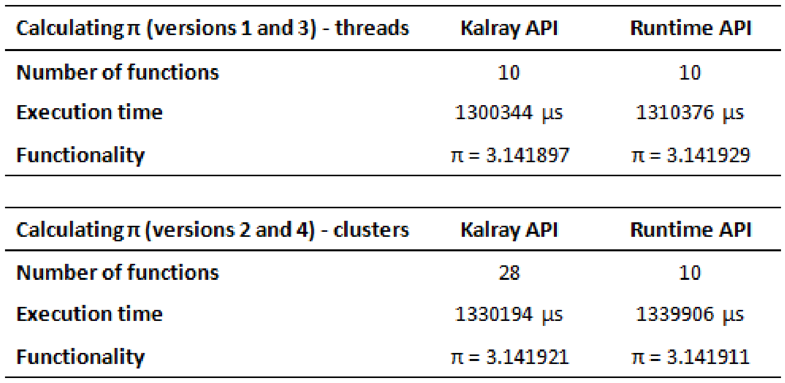
The first and second versions, referenced respectively as *Kalray API - threads* and *Kalray API - clusters*, are the application in pure C language, ported to MPPA and containing only direct MPPA API calls. The first version uses only one cluster and 15 threads, and the second version uses the 16 clusters, with one thread in each cluster.

The third and fourth versions, referenced respectively as *Runtime API - threads* and *Runtime API - clusters*, use the functions of the developed library to execute. In the third version, only one cluster is used to execute and inside this cluster, 15 threads are created to calculate the algorithm. In the fourth version, instead of creating threads inside the clusters, the 16 clusters are used, performing each one thread. A little description about the approaches used is given in the next paragraphs.

In the first and third versions, the *Kalray API - threads* and *Runtime API - threads*, the master sends the task to be executed on the slave side, which will be responsible for creating as many threads as desired to calculate the π number. Each thread calculates a given number of points (in our case, one million points), and update a shared variable with the number of points inside the circle. As they share the same memory space, this communication has no overhead, once it can be easily done by a global variable, accessible for all threads. As each thread access a separated memory area for writing the number of points they have calculated, there are no memory concurrency. The cluster itself is responsible for calculating and printing the number π, after the threads synchronize.

In the second and fourth versions, the *Kalray API - clusters* and *Runtime API - clusters*, the master side sends each cluster the task to perform and wait for the results of the calculation. Again, one million points were used, but this time it is necessary to send it through the communication features, once the clusters don't share the same memory. To do so, the buffer used in this version is a buffer of integers, and write/read operations are performed at the end of the calculation to send back the results. After all clusters synchronize, it is possible to process the individual results and converge to one single value of the number π.

Figure 5.2 shows the results obtained by executing all versions of the application. The *Kalray API - threads* and *Runtime API - threads* are comparable between themselves, as they execute the same operations. Equivalently, the *Kalray API - clusters* and *Runtime API - clusters* versions can be compared between themselves as well.



Calculating π (versions 1 and 3) - threads	Kalray API	Runtime API
Number of functions	10	10
Execution time	1300344 µs	1310376 µs
Functionality	π = 3.141897	π = 3.141929

Calculating π (versions 2 and 4) - clusters	Kalray API	Runtime API
Number of functions	28	10
Execution time	1330194 µs	1339906 µs
Functionality	π = 3.141921	π = 3.141911

Figure 5.2: Calculating π application results.

It is possible to see from figure 5.2 that all versions produced a similar result, showing they are functionally equivalent. It is possible to see a little overhead caused by the library, as both versions using the library took a little bit more time to execute. However, this overhead was about 0.01 seconds for a total execution time of around 1.3 seconds, not being very critical. The number of functions in the versions using clusters is drastically reduced from 28 to 10 functions. In the version using threads it does not happens, as both use 10 functions. It is due to the fact that in the thread version there is no communication between master and slave, making the original version simpler.

The two approaches used explore the opposite situations, calculating only with clusters or threads. But it is completely possible to build a hybrid solution, using as many clusters and as many threads as desired. Both approaches performed perfectly, producing the expected result, proving the runtime library worked fine for this application. The solution using clusters evidence it is possible to use a buffer of integers, avoiding the necessity of converting integers to characters and vice versa, in case the buffer was only for characters. The communication between clusters and I/O node performed correctly.

5.2.2 Calculating the Mandelbrot set fractal

Another simple application used for testing the functionality of the runtime library is the Mandelbrot set fractal. As well as the Monte-Carlo method for calculating π, the Mandelbrot set can be easily parallelized using threads to calculate it.

Fractals are shapes belonging to the non-Euclidian geometry, being used exactly in situations where the Euclidian geometry can't be used or can't be easily explained by the traditional geometry. They usually can be expressed through an iterative or recurrent process, not depending on the scale used. The two-dimension Mandelbrot set is one of the most known examples of fractal shape.

Basically, it samples complex numbers and evaluate whether the result of a mathematic operation iterated in it tends towards infinity or not. In fact, it analyzes if a complex number, when applied the quadratic polynomial operation a number N of times, remain bounded or tend towards infinity, no matter how large N gets. The complex quadratic polynomial is given by the formula below:

$$Zn + 1 = c + (Zn)^2 \tag{5.2}$$

It means that the next number of the sequence (Zn+1) will be given by the addition of the complex number c with the previous number of the sequence, starting with Z0 = 0. Taking a number complex c, if the sequence generated is bounded, it belongs to the Mandelbrot set; however if not bounded, then it does not belong to the Mandelbrot set. An image file can be generated to analyze if the result of calculating Mandelbrot fractal is correct. If correct, it should produce an image as Figure 5.3, with 540x540 pixels.

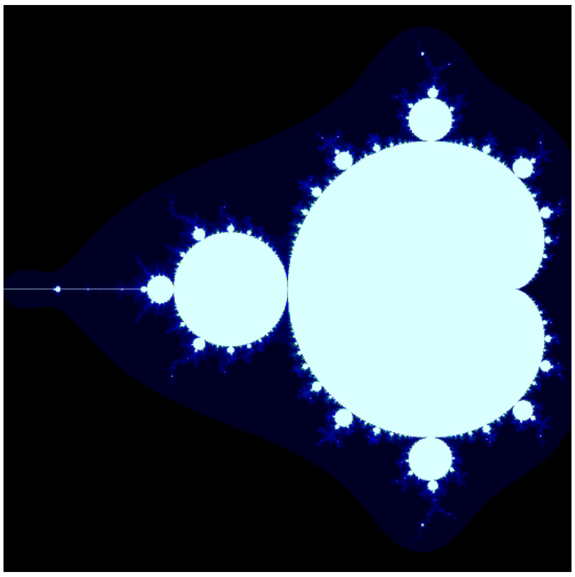
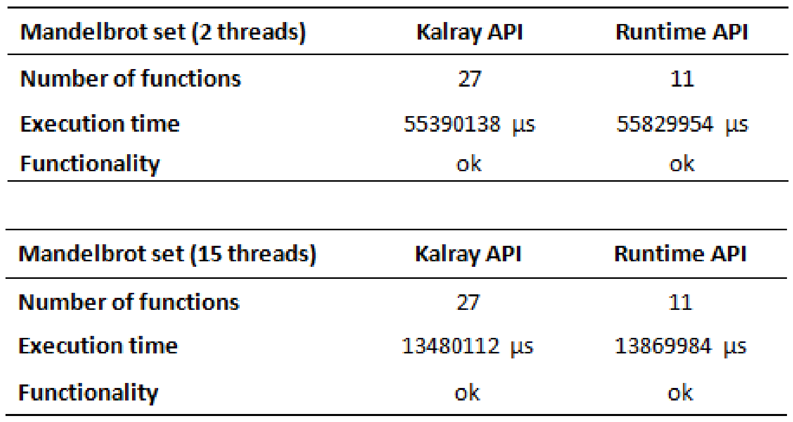


Figure 5.3: Mandelbrot set output.

Again, this application was ported to MPPA, and two versions of the application were built, helping us in the evaluation. A first version, referenced as *Kalray API*, using only direct MPPA API calls and pure C language was used to compare with the one using the library, referenced as *Runtime API*. Both versions use threads inside a cluster to execute the parallel processing, being their master code only responsible for launching the clusters. No master-slave communication was needed in this application, as all calculation is done in the slave side, which is also responsible for printing the output image.

The number of threads used directly influence the time it spends to calculate the Mandelbrot fractal. As the number of threads grows, the execution time decreases. Because of that, two opposite tests were used to verify the parameters: one with 2 threads and one with 15 threads. Figure 5.4 shows the results obtained. In all executions, the image produced was the same figure 5.3.



Mandelbrot set (2 threads)	Kalray API	Runtime API
Number of functions	27	11
Execution time	55390138 µs	55829954 µs
Functionality	ok	ok

Mandelbrot set (15 threads)	Kalray API	Runtime API
Number of functions	27	11
Execution time	13480112 µs	13869984 µs
Functionality	ok	ok

Figure 5.4: Mandelbrot set application results.

The results obtained in figure 5.4 shows the applications successfully produced the right image, being functionally equivalent. The time spent in each version is very similar, but it is possible to identify the overhead caused by the library. In the versions with 15 threads, the overhead was about 0.4 seconds, from a total execution time of 13.87 seconds, around 3% of the total time. In the version with 2 threads, the overhead was of 0.43 seconds, from a total around 55.83 seconds of execution, representing less than 1% of the time. It is possible to see that the number of functions used reduces considerably. Both versions have the same number of functions as we only vary the number of threads used, and not the source code itself.

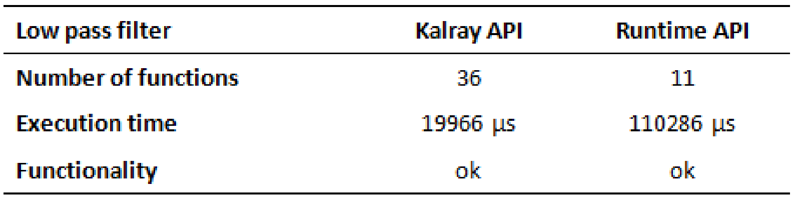
5.2.3 Calculating a low pass filter

The last application used to validate the developed library is used to calculate the convolution product between a sample input generated and a low pass filter. The first version of this application, referenced as *Kalray API*, was already available for MPPA users as an example for channel communication, using only MPPA API calls. The second version of the application, referenced as *Runtime API*, was ported to use the developed library, being only necessary to adapt it to use the runtime library functions.

The application is responsible for producing an input signal in the master side, which is passed to the slave side using the communication features. In the slave side, inside the cluster, functions are responsible for first calculating the filter coefficients, which will be then used to calculate the convolution product between signal and filter. It produces an output signal, which is the signal after passing for the filter, with only low components.

The application uses integers of 2 bytes (int16_t), so the buffer in the runtime library was declared also as a 2 bytes integer, avoiding the conversion problems it could present if the buffer was for characters. The output signal is sent back to the I/O node, through the same kind of buffer (int16_t), to be printed there.

Figure 5.5 shows the comparison between the two versions of the application. As the output consists in an array of more than a hundred components, it is not showed here, but both applications produced the same result.



Low pass filter	Kalray API	Runtime API
Number of functions	36	11
Execution time	19966 μs	110286 μs
Functionality	ok	ok

Figure 5.5: Low pass filter application results.

From figure 5.5 it is possible to conclude that both versions have achieved the same results, producing the same array in the output, proving their functionality. The time of execution is very different from one version to the other, being the original version around 5 times faster. It is due to the fact that the original application uses a channel communication, with dedicated link between master and slave, speeding up the communication overhead the library introduces. The number of functions, however, is reduced from 36 to 11 functions, evidencing the runtime API functions provide a simpler interface.

As the application was first built using the channel communication, it was interesting to verify that the library worked fine using portal instead of channel. Another important point was to verify that the buffer used in the library can be of any type, including characters, integers and even structures, giving it more flexibility.

5.3 Final considerations

With the example applications presented on the previous sections we could in a first moment verify that the runtime library developed was working fine, as all the examples produced the same result using the implemented library and in their *original* versions. It was possible to see that the execution time was close in both versions, with the overhead caused by the library being very small or even negligible due the total execution time.

It was possible to verify that in most cases the number of functions needed from the MPPA API was bigger than the number of functions needed when using the runtime library. Decreasing the number of functions implies in easier and faster development for the final user. As a consequence of decreasing the number of functions used, it was also possible to observe in these applications that the runtime library decreases the number of lines in the source code by 10% to 30%, depending on how much communication the application uses. As more communication is used, more evident are the runtime library gains in source code lines, as the communication functions are alone responsible for more than 10 MPPA API calls.

The developed library stay closer to what is well known in the C language, masking MPPA structures and API calls, which might not seem very natural for programmers at a first moment. Providing functions with a more recognizable face help them to quickly code applications, not having to spend hours reading documents and looking into examples to build a simple program.

Because of that, choosing solutions which can decrease the complexity of the project, being faster to learn, can contribute to improve the project pay back, always providing a comparable if not similar level of performance. Going further on the concepts of software development, programmers always look for a way to build reusable and portable code, avoiding to be obliged to rewrite the code if some change occurs. A high level library can avoid such situations by masking the MPPA API calls. So if a specific function changes, it could be absorbed by the more high level function and be transparent for the final user.

6 CONCLUSION

The industry trends towards heterogeneous computing and manycore architectures are an important indicator that providing tools for developing and porting applications to these processors is a growing market. However, programming manycore architectures are not an easy task, as there is no consolidate solution in the market targeting all architectures available. It appears as a great opportunity for companies to focus on new and efficient solutions and for developers to build and improve their applications' performance.

Developing code for MPPA processor is not trivial, as there is a lot of information that need to be taken into account. Among the aspects necessary for programmers to learn about MPPA before programming we can highlight: the hardware architecture of the processor, its programming model (master-slave model, different operational systems) and the numerous functions in the MPPA APIs. It is necessary to provide higher level primitives to abstract these details from the programmers.

This work contributed, in a first place, for the whole group to improve their expertise about the MPPA processor and how to program it, as it was necessary to study the processor and understand its behavior. Another contribution is the actual runtime solution, which is ready for use, and can be used for developing applications for MPPA using the POSIX threads libraries and master-slave paradigm. An application written in C can be easily ported to run on MPPA by adapting it with the runtime library developed.

We can also conclude, from the results obtained in this work, that investing time and human resources in developing a higher abstraction layer for MPPA is a good idea, once it decreases the complexity of programming MPPA reducing the number of specific functions used, almost at no overhead cost. Using the runtime library prevents the programmer of having to deeply study software and hardware details of the processor. Of course it is still necessary to have some knowledge in the global architecture, the master-slave model, etc. However, details such as how master and slave communicate, how to set a task to a specific core, how to synchronize threads, are invisible for the final user, allowing them to get quickly started developing applications to run on MPPA. An improvement of the current solution or even a solution going towards the OpenCL and Pocl models is a promising idea as an extension of this work.

It is possible to observe also that using the runtime library to build applications decreases the flexibility the programmer has over the application, specially regarding the communication aspects. Another limitation of this work is that the applications used to test don't represent the whole universe of applications in the parallel world, as many applications need to cooperate using inter-cluster communication, which was not in the scope of the runtime implemented.

As this work served as a proof of concept, with a positive result, a possible future work would be to implement a more complex and complete library for the runtime. It could contemplate more specific functions, especially regarding the communication aspects. A possibility is to aggregate more functions to help in synchronizing threads and clusters and using different types of communication.

The solution proposed used only the portal feature for communication among the different possibilities. This choice was made thinking on the requirements defined in the beginning of the project. But providing the users the possibility of choosing another communication type could be very interesting, mainly because it would be possible to mix them. For example, an application could use the Sync feature for synchronization, the Rqueue feature for control and flags exchange (with message passing), and data exchange with portal and channel.

Another possibility would be to explore aspects of the cluster-to-cluster communication, which are not in the scope of this work. This is an interesting idea, as many applications may need to use all different clusters, but still working in a colaborative way. It would require functions to inter-cluster communication and synchronization. A library with cluster-to-cluster communication would reach a new branch of applications.

Finally, the solution proposed contemplates exclusively the Process Management and Communication API and the NODEOS API. Another alternative would be to develop also high level primitives for the PCI Express API, providing tools for the user to easily communicate with the host machine. It could be interesting in applications where the host needs to communicate with other networks, including the Internet, or when other hardware could be integrated to a system, such as a GPU or a DSP.

REFERENCES

AMD. **AMD company**. URL: <www.amd.com>.

ARB, O. **OpenMP ARB consortium**. URL: <http://openmp.org/wp/about-openmp/>.

AUBRY, P.; BEAUCAMPS, P.-E.; BLANC, F.; BODIN, B.; CARPOV, S.; CUDENNEC, L.; DAVID, V.; DORE, P.; DUBRULLE, P.; DINECHIN, B. D. d. et al. Extended Cyclo-static Dataflow Program Compilation and Execution for an Integrated Manycore Processor. **Procedia Computer Science**, [S.l.], v.18, p.1624–1633, 2013.

BLUMOFE, R. D.; JOERG, C. F.; KUSZMAUL, B. C.; LEISERSON, C. E.; RANDALL, K. H.; ZHOU, Y. **Cilk**: an efficient multithreaded runtime system. [S.l.]: ACM, 1995. v.30, n.8.

CHRYSOS, G.; ENGINEER, S. P. Intel® Xeon Phi™ coprocessor (codename Knights Corner). In: HOT CHIPS SYMPOSIUM, HC, 24. **Proceedings** [S.l.: s.n.], 2012.

CLANG. **clang**: a c language family frontend for llvm. URL: <clang.llvm.org>.

DAGUM, L.; MENON, R. OpenMP: an industry standard api for shared-memory programming. **Computational Science & Engineering, IEEE**, [S.l.], v.5, n.1, p.46–55, 1998.

DINECHIN, B. Dupont de; AYRIGNAC, R.; BRAULT, F.; BRUNIE, N.; RAY, V.; VILLETTE, J. Conception et mise en oeuvre d'une architecture VLIW pour le calcul embarqué. **ComPAS'2013**, [S.l.], 2013.

DINECHIN, B. Dupont de; MASSASA, P. G. de; LAGERA, G.; LéGER, C.; ORGOGOZOA, B.; REYBERTA, J.; STRUDELA, T. A Distributed Run-Time Environment for the Kalray MPPAR-256 Integrated Manycore Processor. **Procedia Computer Science**, [S.l.], v.18, 2013.

DREPPER, U.; MOLNAR, I. The native POSIX thread library for Linux. **White Paper, Red Hat Inc**, [S.l.], 2003.

FLYNN, M. J. Some computer organizations and their effectiveness. **Computers, IEEE Transactions on**, [S.l.], v.100, n.9, p.948–960, 1972.

FOUNDATION, H. **HAS Foundation**. URL: <http://hsafoundation.com/>.

GROUP, K. O. W. et al. The opencl specification. **A. Munshi, Ed**, [S.l.], 2008.

INOUE, K. SMYLE Project: toward high-performance, low-power computing on manycore-processor socs. In: ASP-DAC. **Proceedings...** [S.l.: s.n.], 2013. p.558–560.

KALRAY. **Kalray Internal Documentation - Specifications**.

KALRAY. **Kalray Society**. URL: <www.kalray.eu>.

KHOKHAR, A. A.; PRASANNA, V. K.; SHAABAN, M. E.; WANG, C.-L. Heterogeneous computing: challenges and opportunities. **Computer**, [S.l.], v.26, n.6, p.18–27, 1993.

KHRONOS. **The Khronos Group Inc.** URL: <www.khronos.org>.

LLVM. **The LLVM Compiler Infrastructure Project**. URL: <llvm.org>.

MUNSHI, A. The OpenCL Specification v. 1.0. **Khronos OpenCL Working Group**, [S.l.], 2009.

NICKOLLS, J.; BUCK, I.; GARLAND, M.; SKADRON, K. Scalable parallel programming with CUDA. **Queue**, [S.l.], v.6, n.2, p.40–53, 2008.

NVIDIA. **NVIDIA company**. URL: <www.nvidia.com>.

OPENCL. **OpenCL**. URL: <http://www.khronos.org/opencl/>.

PHEATT, C. Intel® threading building blocks. **Journal of Computing Sciences in Colleges**, [S.l.], v.23, n.4, p.298–298, 2008.

POCL. **Portable Computing Language**. URL: <pocl.sourceforge.net>.

POCL. **Pocl**. URL: <http://pocl.sourceforge.net/pocl-0.8.html>.

SEILER, L.; CARMEAN, D.; SPRANGLE, E.; FORSYTH, T.; ABRASH, M.; DUBEY, P.; JUNKINS, S.; LAKE, A.; SUGERMAN, J.; CAVIN, R. et al. Larrabee: a many-core x86 architecture for visual computing. In: ACM TRANSACTIONS ON GRAPHICS (TOG). **Proceedings...** [S.l.: s.n.], 2008. v.27, n.3, p.18.

STALLMAN, R. M. GNU compiler collection internals. **Free Software Foundation**, [S.l.], 2002.

SU, C.-L.; CHEN, P.-Y.; LAN, C.-C.; HUANG, L.-S.; WU, K.-H. Overview and comparison of OpenCL and CUDA technology for GPGPU. In: CIRCUITS AND SYSTEMS (APCCAS), 2012 IEEE ASIA PACIFIC CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2012. p.448–451.

TILERA. **Tilera**. URL: <http://www.tilera.com/products/processors>.

TOP500. **TOP500 supercomputers**. URL: <http://www.top500.org>.

TORRELLAS, J.; LAM, H.; HENNESSY, J. L. False sharing and spatial locality in multiprocessor caches. **Computers, IEEE Transactions on**, [S.l.], v.43, n.6, p.651–663, 1994.

UUSTALU, T.; VENE, V. The essence of dataflow programming. In: **Central European Functional Programming School**. [S.l.]: Springer, 2006. p.135–167.

ANNEX A: TRABALHO DE GRADUAÇÃO I

Adapting *pocl* for manycore processor MPPA®-256

Rafael Leites Luchese[1], **Alexandre Carissimi**[1]

[1]Instituto de Informática - Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Porto Alegre, RS - Brasil

{rlluchese,asc}@inf.ufrgs.br

Abstract. *The recent increase in the number of manycore processors available in the market, due to its high computing power and low power consumption, has been raising new challenges in the parallel computing area. One example of these processors is the Kalray MPPA®-256 manycore processor that provides massively parallel processing. The main challenge is how to efficiently program taking advantage of the high parallelism level manycore processors can supply. Aiming to provide portable and flexible support for programming manycore processors, one emerging solution is the OpenCL framework, a standard specification targeting heterogeneous hardware platforms that allows programmers to easily express parallelism. One available implementation of OpenCL is the Portable Computing Language (pocl), an easily-portable open source implementation. The idea behind this work is to analyze the viability of adapting the pocl language for MPPA®-256 processor.*

1. Introduction

In the past few years the hardware industry has been facing several problems to keep the same improvements of performance we had in the years before. In the same way, the scientific advances achieved in several areas of the human knowledge are requiring more and more computational power. Factors, such as power consumption and reliability, which were not considered very important, are now being the main directives of projects.

The solution adopted by the hardware industry to improve performance is to use parallel architectures by integrating several cores within a single chip. Since the beginning of the 21th century, the multicore processors are becoming more numerous, and attending different types of applications like smartphones, image and audio applications, signal processing, and more. Some of the most interesting advantages of using this approach are speeding-up processing and low power consumption.

On the other hand, the generalization of embedded multicore system solutions has also brought new challenges for programmers, raising the complexity of programming due to the inherent applications' parallelism and the memory architecture used (shared or distributed). Programmers are now expected to think parallel and build solutions which take the most advantage of the hardware resources they have in hands.

This article focuses in one specific manycore processor, the Kalray MPPA®-256, a proprietary Multi-Purpose Parallel Architecture. This processor is one of the solutions proposed by Kalray for high performance computing and low power consumption, and supports two different programming models: dataflow model and POSIX model.

One of the emerging possibilities for programming in shared memory architectures is the OpenCL framework, a standard specification which aims to allow programmers to easily express parallelism. As its name suggests, OpenCL is an open, royalty-free standard which was created to execute across heterogeneous platforms, such as multicore CPUs, GPUs and DSPs. The Portable Computing Language (*pocl*) is an open source MIT-licensed implementation of the OpenCL standard, which aims to provide easy adaptation for heterogeneous hardware.

The main goal of this study is to analyze the constraints existing to adapt *pocl* language to the MPPA®-256 processor. The MPPA processor, the OpenCL framework and the *pocl* language will be presented and discussed in this paper highlighting their relevant characteristics for general comprehension and for the purpose of this study.

In section 2 it will be discussed today's parallel processing architectures, emphasizing NUMA and GPGPUs architectures, which have some similar characteristics with MPPA processor. Section 3 briefly describes MPPA processor architecture, highlighting its most relevant aspects. It will be presented, in section 4, the OpenCL framework and the *pocl* language. Section 5 gives an initial approach to analyze the viability to adapt and use *pocl* to program MPPA processor. Finally, in section 6, it will be presented the expected schedule for developing the studies' activities.

2. Parallel processing architectures

The most traditional and natural programming model is to build sequential programs, where the instructions are executed one after another. They were the first approach used in computer science and are very easy to understand. Nevertheless, executing a program sequentially has been proven to waste processing time, especially during peripherals access. Because of the CPU time wasted, programmers started to search for solutions that could eliminate this loss. The first solution was to overlap I/O operations and processing. With the help of a scheduler and context switching it was possible to execute one process while another one was blocked waiting for I/O operations. This mechanism is called Multitasking.

However having several processes running, sharing common resources and sometimes cooperating with each other, also brought the necessity of providing tools to allow processes to efficiently exchange data. So the next step taken, also done in software, was to use threads. Threads are different execution flows of a same process, which run in the same memory context. As long as switching between threads does not involve a change in the execution context, they are described as light-weight processes.

But these solutions were not, in fact, executing simultaneously in a given time. The parallelism until this time was logical and not physical. With the apparition of multicore processors, it became possible to have two processes (or threads) executing at the same time, using different hardware resources.

The parallel architectures were rapidly absorbed by the users and the industry, becoming more and more common. When the hardware parallelism became true, it brought also new possibilities to programmers. It was now possible to execute one same instruction for multiple input data or even executing different instructions for multiple input data in parallel. These situations mentioned before can be described respectively by the SIMD (Single Instruction Multiple Data) and the MIMD (Multiple Instruction Multiple Data) paradigms [Flynn 1972].

Based on those two paradigms, new classes of machines appeared. The SIMD machines became very popular for processing video applications, due to the applications inherent parallelism. They originated the GPUs (Graphic Processing Units). The MIMD machines are today's multicore processors, used for general processing purpose.

Different classes of MIMD machines exist, and are classified according to its memory access. It is found three main classes of MIMD machines, which are: UMA (Uniform Memory Access) machines, NUMA (Non-Uniform Memory Access) machines and NORMA (Non-Remote Memory Access) machines.

UMA and NUMA machines have both shared memory and a unique address space, and the different between them is that UMA machines share the same physical memory (with uniform access), while NUMA machines have the memory physically spread on the system (implying in non-uniform access). NUMA will be discussed on more details in section 2.1.

NORMA machines are composed by a set of nodes interconnected by a network. Each node is a complete computer system and no process is allowed to access memory modules of other processors. The consequence is that NORMA machines have multiple address spaces and distributed memory. The message passing paradigm is used to program this kind of architecture. NORMA machines are out of our study scope.

The MPPA®-256 manycore processor, target processor for this study, presents some similarity with both NUMA machines and GPUs. That is the reason why this study will focus on these two classes of multicore processors, besides of manycore architectures.

2.1. NUMA architectures

NUMA (Non-Uniform Memory Access) architectures are characterized for having different accesses times depending on the data location on memory, once the memory is spread on the system. The memory is not a single module, but is now divided in several modules. A global view of a general NUMA architecture is shown in Figure 1. Even though, the natural programming paradigm is shared memory, meaning a CPU can access a memory belonging to another module, because all cores still share the same address space. Currently, the most widely used programming languages for programming NUMA machines are OpenMP [Dagum and Menon 1998], Pthreads [Drepper and Molnar 2003], TBB [Pheatt 2008], Cilk [Blumofe et al. 1995] and StreamIt [Thies et al. 2002].

Two new concepts are very important to understand the NUMA structure: local and remote access. A local access occurs when a CPU tries to access the memory within its own module while a remote access occurs when a CPU tries to access memory addresses belonging to another module besides its own. This leads to an important characteristic of NUMA machines: the NUMA factor. It represents the ratio in latency between access times in remote and local accesses (a 10 NUMA factor means a remote access takes 10 times longer than a local access). These two concepts are exemplified in Figure 1.

Local accesses are faster than remote accesses, once the last one needs to pass for the Communication Network to get the data. A very important point linked to local and remote access is the performance achieved by a NUMA machine running a specific application: reducing the number of remote accesses will increase the performance, as it will take less time to execute. That is why it is important to store data and instructions in the memory module associated with the CPU the program will be executed at. In other words, it is very desirable to exploit locality when using NUMA machines. This usually can be done with support from the operational systems, both by the scheduler and by using system calls.

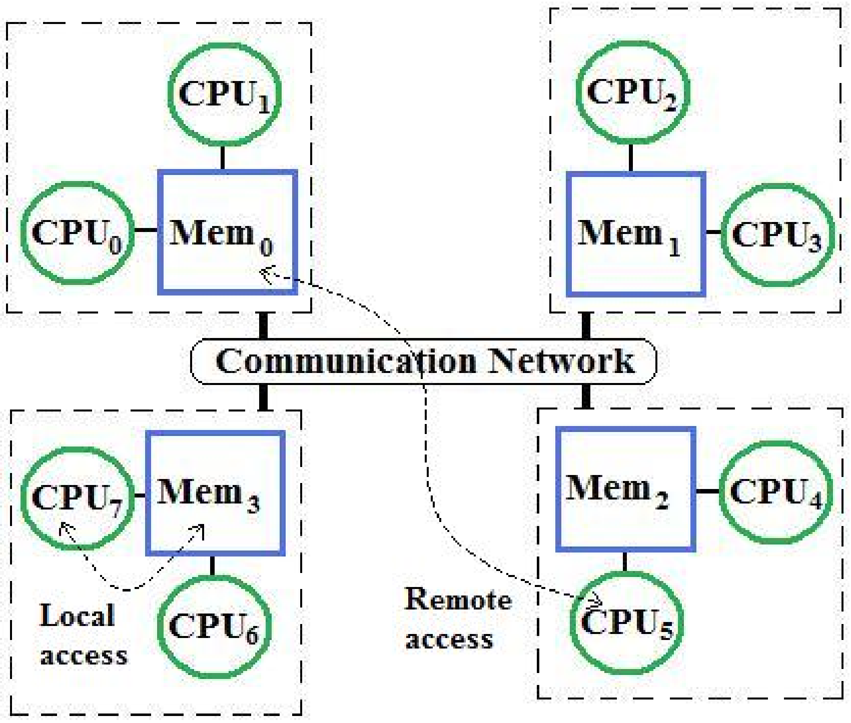


Figure 1. General NUMA architecture.

Usually each core in a NUMA machine has its own cache memory, to speed up memory access and reduce execution time. But cache memories are local, and other cores don't have access to it. It leads us to a common problem that programmers face when programming NUMA machines: the cache coherence problem. It refers to the consistency of data stored in the main shared memory address space and local cache memories. It is possible that some piece of data is present in the cache memory of two or more processors at the same time. When one of these copies is changed, it may not be carried to the main memory right away, making other cores caches inconsistent, which might be a problem in some cases. Some architectures provide hardware support to guarantee cache coherence, but some don't provide any support. In this case, programmers have either to do it by software themselves, or they can ignore the problem, but being aware of it.

Another difficulty related to NUMA machines and cache memories is the false sharing problem. It happens when two processor from different modules share the same cache line, without necessarily sharing common data in that line. So every time a write operation is performed, it invalidates the others cores cache line, even if this operation was not done in the data used by the other cores. It increases memory accesses, as each operation will invalidate the cache line, and the cores will be forced to re-read the cache line for each new operation. Because of the difficulties mentioned above, it is very complex to write efficiency programs for NUMA machines.

2.2. GPGPUs architectures

In the early years of computing, all processing was done by the CPU, including graphics' processing. With the apparition of multicore architectures and using the SIMD paradigm it was possible to create dedicate hardware to accelerate graphic processing, once these class of application had an inherent parallelism (the same instruction must be performed over lots of data). This was the beginning of GPUs' industry, which has being developing very fast since the beginning of the 21th century.

When comparing CPUs and GPUs architectures it is remarkable that CPUs have most of its silicon surface area dedicated to memory (cache) and control, while GPUs have smaller memories and control units, but usually have several arithmetic and logic units (ALUs). This is the main factor which allows GPUs to massively process a given input, extracting its parallelism. Control task is facilitated in GPUs because it executes the same operation for all input elements.

GPUs are modules which are usually connected on a system bus of a conventional computer. From this information it is important to highlight that for using GPUs it is necessary to send and receive blocks of data between CPU and the GPU. This data exchange between CPUs and GPUs is usually done by PCI express bus. The memory hierarchy in GPUs usually presents global, cache and local memories. Global (or device) memory is accessible for all processing groups. Cache memory is usually a L1 cache and is shared by all processing units (PUs) in a same group. Finally, local memory refers to the registers, and each processing unit has its own local memory. The traditional memory hierarchy is shown in Figure 2.

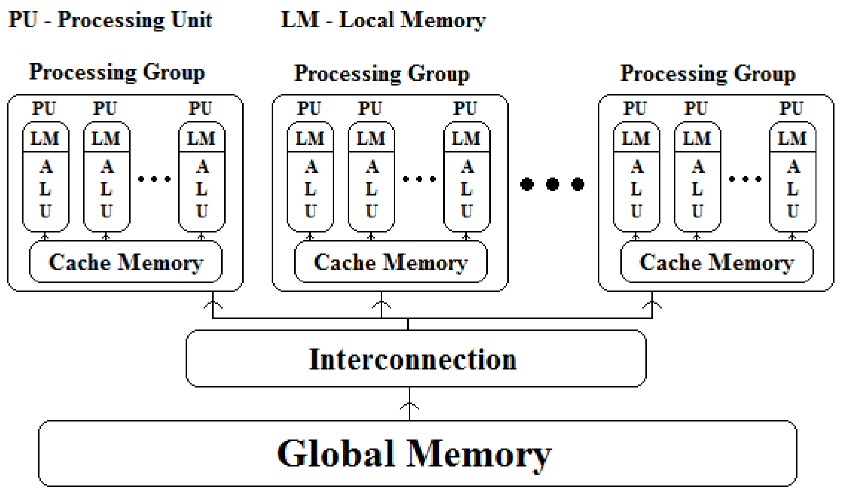


Figure 2. Memory hierarchy in GPUs.

Due to the increasing demand for tremendous computational power in some applications like signal processing and real-time systems, soon programmers came up with the idea of using GPUs to improve CPU processing, and not only to process graphics. This was the beginning of GPGPUs (General Purpose Graphic Processing Units) programming model. The paradigm used to program GPU, which is closely related to the SIMD paradigm, is the stream processing paradigm [Buck et al. 2004].

Another important concept when programming GPUs is the notion of kernels. Kernels are the part of a program that actually execute on a device (GPU). A kernel is syntactically similar to a standard C function. The key difference is a set of additional keywords and the execution model. For example, a kernel can be seen as the body of a loop where each iteration is affected to a program unit.

The two main frameworks used nowadays for GPGPU programming are CUDA [Nickolls et al. 2008] and OpenCL. CUDA is a proprietary solution, developed and maintained by Nvidia, which targets only Nvidia GPUs. OpenCL is currently the open dominant framework for general purpose programming and is maintained by the non-profit technology consortium Khronos Group [Khronos 2013].

2.3. Manycore architectures

With the increasing number of cores in multicore systems, another terminology appeared to describe processors with several cores: the manycore processors. These processors are the hardware purpose to sustain Moore's law in the near future. It refers usually to processors with at least dozens to hundreds cores. It differs from multicore architectures, mainly because of its intrinsic parallelism, showing some similarity with both GPUs and multicore CPUs. Manycore processors aim to maximize throughput, deemphasizing individual core performance.

Manycore approach clearly targets very high parallel processing, being also referred as a highly-parallel device. The software running on manycore architectures is desirable to be parallel, to take advantage of the hardware resources available. Another important characteristic in this family of processor it is the usual presence of a NoC (Network-on-Chip) which allows the cores to communicate with each other.

Another difference between multicore and manycore architectures is in the way they map threads to their cores. While multicore architectures show a certain affinity when mapping threads to cores, manycore architectures mapping is more flexible and fluid. It leads to the biggest challenge faced when programming manycore processors, which is to build efficient parallel programs. Multi-threaded programming is inherently harder than single-threaded development, and it is also vulnerable to race conditions or synchronization errors that deliver unpredictable results and can be hard to debug.

Finally, manycore processors are becoming popular in a moment where power consumption is one of the main directives in the hardware development industry. That is why this is a key factor for this kind of architecture. Provide low power consumption in manycore systems is a very challenging task for hardware designers.

3. The MPPA®-256 processor

The MPPA®-256 manycore processor [Dupont de Dinechin et al. 2013a] was conceived and produced by the Kalray society [Kalray 2013] as a solution to high performance computing with low power consumption. Kalray is a French company specialized in developing manycore processors for high performance applications. It was built with a CMOS 28nm technology and integrates 288 cores in a single chip, having 32 cores dedicated to resource management and 256 calculating cores for computational processing.

Kalray provides some numbers which show its high computational capacity and low power consumption. The MPPA®-256 processor can operate 500 Giga operations per second and 230 GFLOPS, extracting the most of its 256 cores. Also the improvement in power consumption when running typical applications, comparing to other processors, is about 10 times, having a 5 Watts typical consumption. The next subsection will discusson more details the internal architecture of MPPA and its programming models.

3.1. MPPA internal architecture

One important characteristic found in the MPPA processor is its homogeneous approach considering its 288 cores, which are all VLIW (Very Long Instruction Word) processors. All of them have the same architecture, including the ones used for resource management. It was a particular choice of Kalray experts because of its deterministic execution time and low power consumption compared to others possible choices.

The MPPA's 256 calculating cores are divided into 16 groups of 16 cores, and each group is referred as a cluster. Each cluster has 17 cores: 16 calculating cores, called processing elements (PE), and 1 resource management (RM) core used for control, besides of a local shared memory. This memory is a 2MB low latency cache for instructions and data, and it is shared by all the 16 cores in a single cluster. This organization is a huge advantage of MPPA processor, enabling a high bandwidth and throughput between PEs on a same cluster. Each PE has its own L1 cache, and for simplicity, the MPPA processor does not implements cache coherence between L1 caches of its PEs.

We also find in the MPPA processor four Input/Output subsystems, one in each side of the chip, being referenced as north, east, west and south. Each I/O subsystem has 4 cores, totalizing 16 cores dedicated to PCI express, Ethernet, Interlaken and other I/O devices. The 4 cores inside a I/O subsystem share the same address space, and all of them connect directly to 2 clusters.

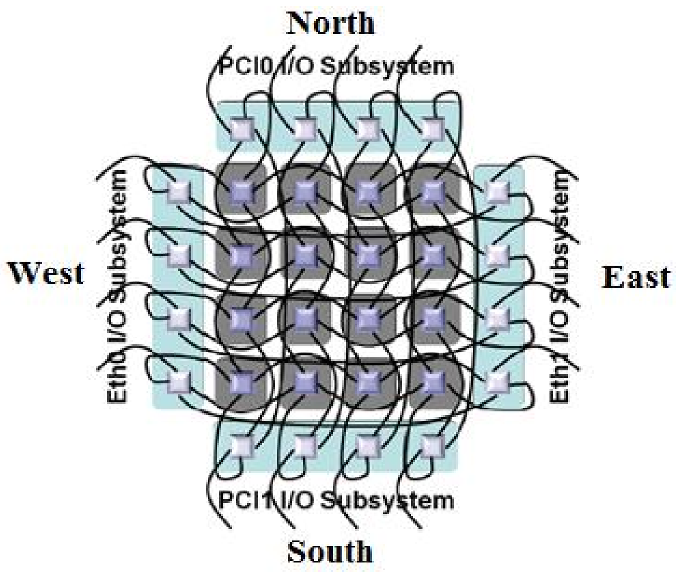


Figure 3. NoCs architecture, integrating clusters and I/O subsystems.

The communication of the clusters is assured by two high speed low latency NoCs (Network-on-Chips), one used for data transmission (the D-NoC) and the other is used for control (the C-NoC). These NoCs have a 2D torus topology, integrating the clusters and the I/O subsystems. Each cluster is associated with a node of the NoC as well as each I/O subsystem, disposed as shown in Figure 3, extracted from [Dupont de Dinechin et al. 2013b]. One important point to highlight concerning the NoCs is that they provide for all clusters a direct link with two I/O subsystems, including the insider clusters.

Today, the MPPA processor is being used connected to a host CPU, very similarly to a GPU connection. The host CPU sends code to be executed in the MPPA processor using the I/O subsystem. The MPPA then process the operations and write the output data in a 2GByte RAM memory, which is connected to an I/O subsystem and can be accessed by the CPU. All the transfers within the I/O subsystems and the MPPA clusters are done by the NoCs. For programming the MPPA processor, there are two different approaches: dataflow and shared memory programming models, which are discussed hereafter.

3.2. Dataflow programming model

One of the possibilities when programming MPPA processor is to use the dataflow programming model. This model is characterized for modeling the program as graph that specifies the data flowing, being a very different way to program from the imperative languages that most programmers are used to. One example of dataflow programming in the MPPA processor using the Sigma C language is described in [Aubry et al. 2013].

For Dataflow programming model, Kalray has developed a set of tools available on a graphic IDE based in C/C++ called MPPA ACCESSCORE IDE. It integrates a compiler, a simulator, a profiler and a debug platform that allows programmers to develop parallel applications without any special background on the hardware architecture. The dataflow programming model, despite its advantages, is not yet very natural for most programmers, which are accustomed to the POSIX programming model.

3.3. POSIX programming model

Another way to program MPPA processor is to use the POSIX programming model. The pthreads (POSIX threads) libraries are a standard thread based API for C/C++, widely used in multithread applications. The paradigm used in this programming model is shared memory, so all cores in a same cluster share the same address space. Given the similarities MPPA presents with NUMA architectures, difficulties mentioned in section 2.1 (such as false sharing, cache coherence, locality, and others) when programming NUMA machines may also apply when programming MPPA processor.

In this programming model, it is recommendable for the programmer to have some knowledge of the processor's architecture, to take full advantage of the locality factor. MPPA supports pthreads programming, where a process run in a cluster and threads run on cores belonging to this cluster, sharing the cluster's memory. Inter-processes communication is done through the NoC, using the NoC IPC library. In this context it is also possible to use C/C++ and OpenMP to program MPPA processor.

4. The OpenCL framework

In the last few years, the hardware resources available for programmers have become more heterogeneous, including multicore CPUs, GPUs and DSPs (Digital Signal Processors). It was desirable then to create a solution in software which could be able to run under the different hardware, without having to change the applications' source code. In this context an emerging solution is the OpenCL (Open Computing Language) framework, which aims to be a standard for programming heterogeneous platforms.

OpenCL [Khronos 2013] is a royalty-free standard started by Apple and maintained, nowadays, by the Khronos Group. Several expressive enterprises also collaborate with OpenCL standard development, such as Intel, AMD, Nvidia and others. OpenCL is a specification OpenCL is in its third release (OpenCL 1.2) from November 2011 and is becoming very popular among GPGPU programmers lately, being supported by most expressive GPUs fabricants.

The OpenCL framework described in [Munshi 2009] contains three components: a platform layer, a runtime and a compiler. The platform layer allows OpenCL to discover the device capabilities and to create an execution context. The runtime allows the host device to manipulate the context created. Finally the compiler creates executable programs which contain OpenCL kernels. In the Khronos Group documentation an OpenCL language is also specified, called OpenCL C, which is used to create kernels in the application's source code. The OpenCL architecture is composed by four models: programming model, platform model, execution model and memory model.

The OpenCL programming model involves a host/device communication. Nevertheless the host and the device don't need necessarily to be different hardware devices, being possible that both are on the same hardware. In a multicore processor, for example, it is possible to designate one core to be the host and the other to be the device. Anyway, its traditional programming model suits perfectly MPPA processor's connectivity, which is done in this exactly way nowadays, very similarly to a GPU connection.

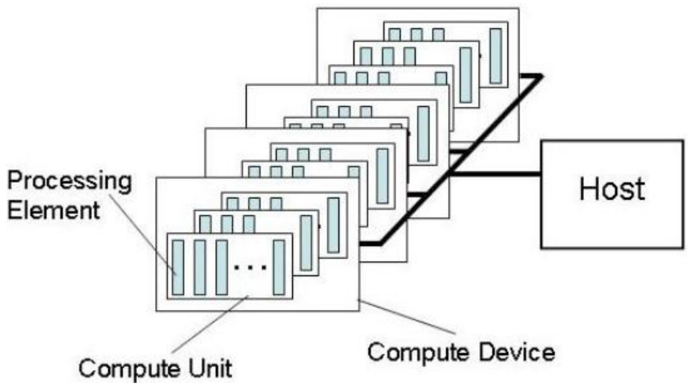


Figure 4. Platform model for OpenCL.

OpenCL presents a platform model very similar with GPUs, where we found a host who issues commands to one or more Compute Devices (CDs). Each CD can be divided into Computing Units (CUs) and each CU can also be divided into Processing Elements (PEs), as shown in Figure 4.

The execution model of OpenCL involves two parts: the host program and the kernels. Kernels are sent by the host program and are queued to be executed on the device. Each instance of a kernel corresponds to a work-item, and can be organized into work-groups. There are two main programming models for OpenCL: data and task parallel. In data parallel model, all work-items execute the same program while task parallel model works like a trap-door to run arbitrary code from the command queue.

The memory model found in OpenCL, shown in Figure 5, is somehow similar to GPU memory model explained on section 2.2. There are four hierarchical memories: global, constant, local and private memory. Global memory is accessible by all work-items belonging to any work-group, for read and write operations. Constant memory is a region of global memory that remains constant during the execution of a kernel. The local memory is a shared memory within a work-group while the private memory is not visible by any other work-item, being private for each work-item. The host has no access to local and private memories meanwhile the device has access to all memories.

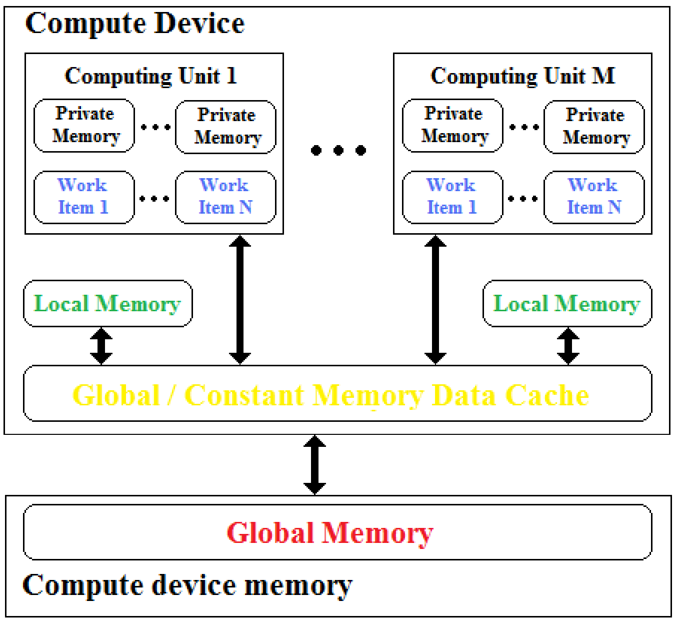


Figure 5. OpenCL memory model.

In a high abstraction level it is possible to describe the process of building and executing OpenCL program as follow. A source code is built, containing mainly C commands and kernels (OpenCL directives). The compiler generates the object code, which will be linked by a runtime library, building the executable file taking into account the target hardware. The executable contains both C and OpenCL directives, and will be loaded to the device memory to be executed.

Figure 6 presents the OpenCL software architecture. The application communicates with the hardware through a set of tools including an OpenCL library, a Runtime, and an OpenCL device driver, which are involved in the process of building executable OpenCL code to run in a target device. The Platform Layer also helps in this process by providing information about the hardware to the application.

OpenCL presents similarities and differences with CUDA. As a Nvidia proprietary solution, CUDA targets only Nvidia GPUs, while OpenCL was developed targeting heterogeneous platforms. Both of them have been used for GPGPU programming, and have similarities concerning their architecture and memory model. Performance comparison between them have been made lately and CUDA usually provides better results especially because of its specificity, although OpenCL is more flexible, losing some performance to provide portability [Su et al. 2012].

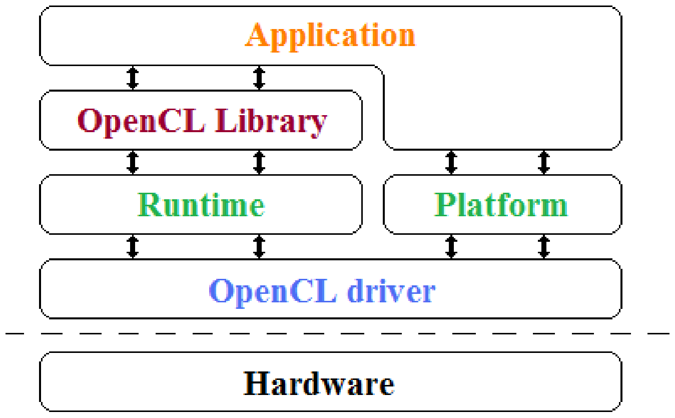


Figure 6. OpenCL software architecture.

Being just a specification, it is necessary to implement OpenCL to run it on a new device. Companies such as ARM, IBM, Intel and Nvidia have their own proprietary implementations of OpenCL. There is also an open source implementation of OpenCL, the Portable Computing Language [Pocl 2013] with an MIT-license aiming to be easily adaptable for new target devices.

The *pocl* implementation is nowadays in version 0.7. It uses Clang [Clang 2013] as an OpenCL front-end language, used to interpret kernels, and LLVM [LLVM 2013] as back-end for the kernel compiler implementation, both belonging to the same project, which is an open source project. The project aims to provide portability for *pocl*, so if the hardware has a LLVM back-end, it should be easy to adapt *pocl* to the target device. *Pocl* can be used for heterogeneous architectures, and aims to exploit all kinds of parallel hardware resources, such as: multicore, VLWI, SIMD and others.

5. Adapting *pocl* for MPPA®-256 processor

A possibility to program today's manycore processors is to adapt the existent frameworks. Our main goal in this study is to focus on OpenCL framework because it aims to provide portability and efficiency when programming through heterogeneous platforms beyond being a royalty free solution. Manycore MPPA processor is also very promising for high performance parallel computing, and an OpenCL implementation targeting this processor could bring a powerful tool to develop massively parallel and portable programs. It would give programmers one more possibility when programming MPPA processor, making it even more versatile.

The idea of combining both MPPA and *pocl* came up naturally. So the main goal of this work is to analyze the viability of adapting the *pocl* language to develop programs for the MPPA processor. It is intended, as the expected results of this work, to verify the limitations and difficulties to accomplish such adaptation. To achieve such objective, the purpose is to develop a runtime which will be linked with programs built using the *pocl* language for the MPPA processor. The choice of *pocl* was mainly based in the fact that it is an open source solution.

The runtime to be developed has to be capable, in a first moment, of loading OpenCL kernels for MPPA, in its resource management (RM) core of the I/O controllers.

This RM core should then redistribute the tasks to be executed in the different clusters of MPPA. The cluster's RM core, in its turn, will then be responsible for receiving the task to be executed and for creating work units for each processing core in that cluster. Finally, it should wait for all cores to finish processing, synchronizing and sending the results back to the I/O controller RM, which should then make it available in the MPPA global memory. Basically, the proposed runtime should provide the following functionalities:

- Initialization;
- Launching a task in a cluster;
- Redistribute this task into the cores belonging to that cluster;
- Final synchronization to send the results back.

To provide these functionalities, it will be necessary to study and modify the function calls that the OpenCL (*pocl*) library makes. The communication between the host CPU and MPPA processor will be also the runtime responsibility. It is, in fact, the exactly purpose of the runtime, to be a tool which will automate the communication between MPPA and the host CPU.

This work is part of a scientific cooperation between GPPD team (Grupo de Processamento Paralelo e Distribuído) of the Instituto de Informática UFRGS and Nanosim team, Université de Grenoble (France). The machine used to develop this work is located in Grenoble, France, meanwhile the tests will be realized accessing the machine remotely, from Porto Alegre, Brazil.

6. Project schedule

The activities to be done aiming to accomplish our goal of adapting *pocl* implementation for MPPA processor have been sliced into smaller steps and the expected schedule is presented in Table 1.

Table 1. Project schedule.

Activity	Aug	Sept	Oct	Nov	Dec
Runtime architecture design	X				
Runtime implementation		X	X		
Tests and debugging			X	X	
Results evaluation				X	
Monograph redaction	X	X	X	X	
Presentation					X

References

Aubry, P., Beaucamps, P.-E., Blanc, F., Bodin, B., Carpov, S., Cudennec, L., David, V., Dore, P., Dubrulle, P., Dinechin, B. D. d., et al. (2013). Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, 18:1624–1633.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). *Cilk: An efficient multithreaded runtime system*, volume 30. ACM.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for gpus: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM.

Clang (2013). clang: a c language family frontend for llvm. URL: <clang.llvm.org>.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55.

Drepper, U. and Molnar, I. (2003). The native posix thread library for linux. *White Paper, Red Hat Inc*.

Dupont de Dinechin, B., Ayrignac, R., Brault, F., Brunie, N., Ray, V., and Villette, J. (2013a). Conception et mise en oeuvre d'une architecture vliw pour le calcul embarqué. *ComPAS'2013*.

Dupont de Dinechin, B., de Massasa, P. G., Lagera, G., Léger, C., Orgogozoa, B., Reyberta, J., and Strudela, T. (2013b). A distributed run-time environment for the kalray mppar-256 integrated manycore processor. *Procedia Computer Science*, 18.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960.

Kalray (2013). Kalray society. URL: <www.kalray.eu>.

Khronos (2013). The khronos group inc. URL: <www.khronos.org>.

LLVM (2013). The llvm compiler infrastructure project. URL: <llvm.org>.

Munshi, A. (2009). The opencl specification v. 1.0. *Khronos OpenCL Working Group*.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with cuda. *Queue*, 6(2):40–53.

Pheatt, C. (2008). Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298.

Pocl (2013). Portable computing language. URL: <pocl.sourceforge.net>.

Su, C.-L., Chen, P.-Y., Lan, C.-C., Huang, L.-S., and Wu, K.-H. (2012). Overview and comparison of opencl and cuda technology for gpgpu. In *Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on*, pages 448–451. IEEE.

Thies, W., Karczmarek, M., and Amarasinghe, S. (2002). Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer.