A Runtime for Code Offloading on Modern Heterogeneous Platforms

Rogério Aparecido Gonçalves

Thesis submitted
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
for the
Doctoral Degree in Science

Program: Computer Science
Advisor: Prof. Dr. Alfredo Goldman

This research is supported by
Fundação Araucária (projeto DINTER UTFPR/IME-USP),
CAPES-COFECUB (Doutorado SDW 2015, proc. BEX 3401-15-4)

São Paulo, August, 2016

A Runtime for Code Offloading on Modern Heterogeneous Platforms

This is the original version of thesis written by (Rogério Aparecido Gonçalves), such as submitted to Committee.

Committee Members:

- Prof. Dr. Alfredo Goldman (supervisor) - IME-USP

Acknowledgements

I would like to thank my Advisor Prof. Alfredo Goldman for the guidance and good discussions during this period of study. He helped me and gave me good opportunities to learn more about the working in group and do research. I thank the LSS and CCSL colleagues, for helping me and your good company during the work in São Paulo.

I also thank all the IME professors that participate of DINTER project, for their cooperation and their commitment to execute the work with our group. I thank UTFPR, our directors and involved departments (DIRPPG, DIRGRAD, PROPPG) for the support during this project execution. I thank my colleagues of DACOM for their collaboration in my absence period to do this work.

I'm very grateful to the Inria Compiler Optimization and Runtime Systems Team (CORSE) that received me at Grenoble in my internship period as Visiting PhD Student. Fabrice Rastello and people of research group, thank you for your time and your help. I would like to thank my supervisors of there Professor Jean-François Mehaut (Université Grenoble Alpes - UGA, Laboratoire d'Informatique de Grenoble - LIG, Inria Grenoble Rhône-Alpes) and Professor Henri-Pierre Charles (Université Grenoble Alpes - UGA, LIST, CEA Grenoble) to receive me at Grenoble and for all discussions about the my research. Thank you for help me. I appreciated the mission in Grenoble and the chance of know the group people.

I thank my family for always support me, my wife Sabrina, my partner in this life journey, thanks for your love, patience and dedication to me during this study period.

Finally, I would like to thank the Fundação Araucária, the Departamento de Ciência, Tecnologia e de Ensino Superior do Governo do Estado do Paraná (SETI-PR) and State Government of Paraná for the financial support to make possible the DINTER UTFPR-USP project (CAPES registred DINTER 001-22/2011/CAA III/CGAA/DAV of 30/03/2011). I'm grateful to CAPES for the sandwich scholarship (Doutorado SWD 2015, BEX 3401/15-4, CAPES-COFECUB Project No. 828/15, *Choosing: Cooperation on Hybrid Computing Clouds for Energy Saving*) for the fellowships and the accomplishment of this work.

Abstract

GONÇALVES, R. A.. **A Runtime for Code Offloading on Modern Heterogeneous Platforms.** 2016. 94 f. Thesis(Doctoral) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2016.

The modern parallel processing platforms increasingly have brought to users multi-core systems with complex memory hierarchies organizations and new features as vectorization support. Moreover, these platforms congregate heterogeneous elements, from multi-core CPUs to many-cores GPUs. When these elements are integrated, the power processing of these platforms can are boosted. However, in the software side there is the need to modernize the code of legacy applications to use these new resources. Even when this is done, it may happen that the workload overcomes the capacity of multi-core systems usually there is the need to parallelize the applications code to use the computing power of coprocessors and accelerators devices. Programming for these new platforms is not a trivial task, either writing new code or transparently translation of legacy code are very complex tasks. These platforms with accelerators devices and multicore processors, even providing development kits, require the programmer to explicitly declare all data transfers between device memories. The programmer should specify the complete structure of grids and blocks of threads to launch the kernel execution on each specific device. To mitigate this condition, tools and approaches have been proposed to generate code for these platforms. Among the approaches that have excelled, one approach uses compilation directives and other approaches try to detect parallelizable code regions. In the first approach, compilation directives are used to guide the compilation process, where code transformations and modifications are applied on annotated regions to obtain a parallell code version. In the second approach the code translation should occur without modifications on original code and without programmer intervention. For this outcome, these approaches apply models and techniques to automatically detect which regions of code are parallelizable. In this thesis, we describe a runtime related with automatic code parallelization and offloading based on code versioning of parallel loops. The idea is that the OpenMP input code can be generated by compiler tool or written manually. Our runtime libraries are intercepting some applications calls for OpenMP runtime using a hooking technique. The decision about offloading have been taken automatically at runtime using the operational intensity that is obtained applying Roofline Model concepts. We are considering measures in all levels of memory hierarchy and the data transfers between host and devices.

Keywords: heterogeneous computing, automatic parallelization, code interception, operational intensity, offloading, accelerators devices, OpenMP, GPGPU, tools.

Resumo

GONÇALVES, R. A.. **Um Ambiente de Execução para Offloading de Código em Plataformas Heterogêneas Modernas.** 2016. 94 f. Thesis (Doctoral) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2016.

As plataformas modernas de processamento paralelo cada vez trazem aos usuários sistemas *multi-core* com organizações de hierarquia de memória complexas e novos recursos como o suporte à vetorização. Além disso, essas plataformas reúnem elementos heterogêneos, que vão de CPUs *multi-core* a GPUs *manycores*. Quando esses elementos são integrados, o poder de processamento dessas plataformas pode ser potencializado. Entretando, do lado do software há a necessidade de modernização de aplicações de código legado para que usem esses novos recursos. Mesmo quando isso é feito, pode acontecer que a carga de trabalho ultrapasse a capacidade de sistemas *multi-core*, usualmente há a necessidade de paralelizar o código dessas aplicações para usarem o poder de processamento de coprocessadores e dispositivos aceleradores. Programar para estas novas plataformas não é uma tarefa trivial, quer seja escrever um novo código ou a tradução de código legado de maneira transparente são tarefas muito complexas. Estas plataformas com dispositivos aceleradores e processadores multi-core, mesmo fornecendo kits de desenvolvimento, requerem que o programador declare explicitamente todas as transferências de dados entre as memórias dos dispositivos. O programador deve especificar a estrutura completa de grids e blocos de threads para lançar a execução de kernels em cada dispositivo. Para amemizar essa condição, ferramentas e abordagens tem sido propostas para gerar código para essas plataformas. Entre as abordagens que tem se destacado, uma usa diretivas de compilação e a outra tenta detectar automaticamente regiões de código paralelizáveis. Na primeira abordagem, diretivas de compilação são usadas para guiar o processo de compilação, no qual transformações e modificações de código são aplicadas às regiões anotadas para obtenção da versão paralela de código. Na segunda abordagem a tradução de código deveria ocorrer sem modificações no código original e sem a intervenção do programador. Para se alcançar esses resultados, trabalhos nessa categoria aplicam modelos e técnicas para detectar automaticamente quais regiões de código são paralelizáveis. Neste trabalho, apresentamos um *runtime* relacionado com paralelização automática e *offloading* de código baseado em versões de código para laços paralelos. A ideia é que o código de entrada OpenMP seja gerado por uma ferramenta de compilação ou escrito manualmente. As bibliotecas do *runtime* são capazes de interceptar, usando uma técnica de *hooking*, algumas chamadas que as aplicações fazem ao *runtime* do OpenMP. A decisão de *offloading* tem sido tomada automaticamente em tempo de execução usando a intensidade operacional que é obtida aplicando-se conceitos do Modelo Roofline. Estamos considerando medidas em todos os níveis da hierarquia de memória e das transferências de dados entre o *host* e os dipositivos aceleradores.

Palavras-chave: computação heterogênea, paralelização automática, interceptação de código, intensidade operacional, offloading, aceleradores, OpenMP, GPGPU, ferramentas.

Contents

List of Abbreviations and Acronyms **vii**

List of Figures **viii**

List of Tables **x**

1 Introduction **1**

1.1 Research Problem 3

1.2 Proposed Approach and Methodology 4

1.3 Objectives 5

1.4 Main Contributions 6

1.5 Thesis Organization 6

2 Related Works **7**

2.1 Compilation Tools and Runtimes for Offloading 7

2.2 Roofline Model 9

2.3 Performance Counters 12

2.4 Hooking 13

2.5 OpenMP Support for Offloading and Accelerators 13

2.6 Final Considerations 14

3 HOOKOMP: Hooking calls to OpenMP Runtime **15**

3.1 Study of Compilation Directives and Expanded OpenMP Code 15

3.1.1 Parallel Regions: `parallel` constructor 16

3.1.2 Loops: `for` constructor 18

3.2 Interceptable OpenMP Code Format 33

3.3 Interception and Offloading Mechanisms 36

3.4 The runtime input code format 37

3.5 Application and Runtime Libraries Interaction 38

3.6 Creating Hooks for `OpenMP` Functions 40

3.7 Final Considerations 46

4 Offloading Decision using Operational Intensity **47**

4.1 Operational Intensity 47

4.2 Events Sets and Hardware Restrictions 49

4.3 Measuring Chunks 50

4.4 Decision Model 50
4.5 Final Considerations 54

5 Experiments and Results 55
5.1 Execution Platform 55
5.2 Benchmarks 57
5.3 Overhead Analysis 57
5.4 Offloading and Chunk Size Evaluation Analysis 59
5.5 Final Considerations 62

6 Future Work 63
6.1 Compilation Tool Implementation 63
6.2 Runtime Improvements 64
6.2.1 Modify the strategy of collecting the values of Performance Counters 64
6.2.2 Consider Data Placement issues 65
6.2.3 Speculative Data Copy to device 65
6.2.4 Data Synchronization issues 65
6.2.5 The use of other decision models 65
6.2.6 Support other accelerators devices and other platforms 65
6.2.7 Use of Optimized Vendor Libraries 66
6.3 Final Considerations 66

7 Conclusions 67

A Creating Hooks for OpenMP Functions 68
A.1 Expanded code for parallel region with two loops 68
A.2 Generic next chunk function 70

References 74

List of Abbreviations and Acronyms

ABI	Application Binary Interface
API	Application Programming Interface
cuBLAS	CUDA Basic Linear Algebra Subroutines
DVFS	Dynamic Voltage and Frequency Scaling
OS	Operating System

List of Figures

2.1 Roofline Graph Example. Adapted from Williams *et al.* (2009) 10

3.1 Visualization of generated code format that uses static scheduling and loop upper bound with numeric value 23
3.2 Visualization of generated code format that uses static scheduling and loop upper bound with variable 24
3.3 Visualization of generated code format that uses dynamic scheduling (first format) . 29
3.4 Visualization of generated code format that uses dynamic scheduling (second format) 30
3.5 Two loops inside the same parallel region 31
3.6 Graphical representation of two loops inside the same parallel region 32
3.7 Loop iterations execution by threads 33
3.8 First format with dynamic scheduling 34
3.9 Second format with dynamic scheduling 35
3.10 Structure of loop formats 35
3.11 Threads executing loop iterations (Version 1) 36
3.12 Threads executing loop iterations (Current version) 37
3.13 Use of the alternative functions table 38
3.14 Application and Runtime Libraries Interaction 39
3.15 Mapping functions libraries called by applications 39

4.1 Memory Hierarchy Levels 48
4.2 Mechanism for measuring chunks 51

5.1 Xeon E5-2630 v2 (Intel, 2014) 56
5.2 Complete platform 56
5.3 *Overhead of decision code.* The runtime collect and calculate the decision, but force the execution in CPU side: *no offloading* 58
5.4 *Overhead of decision code.* The runtime collect and calculate the decision, but force the execution in CPU side: *no offloading* 58
5.5 *Chunk_size evaluation.* Configuration *chunk_size* = 16 59
5.6 *Chunk_size evaluation.* Configuration *chunk_size* = 32 60
5.7 *Chunk_size evaluation.* Configuration *chunk_size* = 64 60
5.8 *Chunk_size evaluation.* Configuration *chunk_size* = 128 61
5.9 *Chunk_size evaluation.* Configuration *chunk_size* = 256 61
5.10 Execution configuration that reaches the offloading decision 62

6.1 Compilation process workflow 63
6.2 Interleaved measurements strategy (Third version) 64

List of Tables

3.1 Possible combinations for *upper bound* and *chunk_size* definitions 25

4.1 Events Sets 49

5.1 Platform of Execution description 57
5.2 Benchmarks useds in experiments 57

Chapter 1

Introduction

There is a growing evolution of parallel processing platforms. The modern platforms have provided new features in multi-core systems as complex memory hierarchies organizations and vectorization support. Moreover, these platforms are composed by heterogeneous elements as coprocessors and accelerators devices. In fact, coprocessors and GPUs have a great processing power and the possibility to explore this potential has made that these elements are used in the composition of the platforms of the most powerful supercomputers in the world and appear on first positions on the rank list top500.org (Dongarra *et al.*, 1994, 1996) (Top500, 2013, 2014, 2015, 2016).

In november 2010, the chinese supercomputer `Tianhe-1A` was the first in the `top500` list. It was using GPUs of *Fermi* architecture (NVIDIA Tesla 2050) (Glaskowsky, 2009; NVIDIA, 2009; Patterson, 2009). In the following year it stayed among the first positions in the list. In november 2012, the american supercomputer `Jaguar`, which occupied the third position in the previous list, was upgraded with *Kepler* GPUs (NVIDIA K20x) (NVIDIA, 2012) becoming the `Titan`, the new features ensured that it was in the first position. The chinese supercomputer `Tianhe-2` reassumed the ranking first position in november 2013 using Intel Xeon Phi coprocessors (31S1P) (Intel, 2013), staying in this position until November 2015. This suggests that accelerator devices tend to remain in the base composition of these heterogeneous hardware platforms.

The quest for high performance in general-purpose and scientific applications, have turned to the exploration of new resources provided by platforms and their efficient use. In the software side there is needed to modernize the code of legacy applications to use the new features in the several processing elements (Intel, 2015b). If the workload overcomes the capacity of multi-core systems there is the need to parallelize the applications code to use the computing power of coprocessors and accelerators devices.

Usually, the languages C/C++ and Fortran are used to develop applications on these platforms, that follow the historical use of these languages in High Performance Computing applications. Many of the available Software Development Kits have support for C/C++ and Fortran. Among GPUs, the NVIDIA GPUs use CUDA (NVIDIA, 2015a, 2014a,c, 2015c) and AMD/ATI GPUs use OpenCL (Khronos, 2013), that also has been used by the Intel coprocessors (Intel, 2013; Newburn *et al.* , 2013).

There are also other approaches that have excelled in the context of parallel computing: *the compilation directives* and *automatic parallelization*. In the first approach, compilation directives are used to guide the compilation process, where code transformations and modifications are applied on annotated code regions to obtain a parallel code version for a target platform.

In compilation directives category, the OpenMP (Dagum e Menon, 1998) (OpenMP-ARB, 2011, 2013, 2015) (OpenMP API Site, 2012) (Chapman *et al.*, 2007) is the most known and used in multi-core applications. Multiple compilers have support to OpenMP implementations, such as GCC (GCC, 2015; GNU Libgomp, 2015c), Intel icc (Intel, 2016a,b) and LLVM clang (Lattner e Adve , 2004; LLVM Clang, 2015) (LLVM OpenMP, 2015). The OpenMP 4.0 specification covers the code offloading to accelerators devices (OpenMP-ARB, 2013). The GCC libgomp (GNU Libgomp, 2015a,b, 2016a,b) have generated code using the OpenACC standard (OpenACC, 2012, 2015b).

Among the tools that make use of compilation directives to transform and generate code are OpenMC (Lee e Eigenmann, 2010), hiCUDA (Han e Abdelrahman, 2009) (Han e Abdelrahman, 2011) (hiCUDA Project, 2012), CGCM (Jablin *et al.*, 2011) that optimizes communication and data transfers, PGI compilers (PGROUP, 2015, 2013, 2010) OpenHMPP (CAPS, 2012) tools and accULL (Reyes *et al.*, 2012) are compilers that implement the OpenACC standard (OpenACC, 2012, 2015b) (OpenACC, 2011, 2013, 2015a).

The automatic code parallelization is an approach where the code translation should occur without modifications on original code and without programmer intervention. For this outcome, these approaches apply models and techniques to automatically detect which regions of code are parallelizable. Most of the tools use the *Polyhedral Model* (Bastoul, 2004) (Benabderrahmane *et al.*, 2010) (Grosser e Simbürger, 2014) or equivalent concepts. Some examples of tools in this category are C-to-CUDA (Baskaran *et al.*, 2010), Par4All (Amini *et al.*, 2012; Par4All Site, 2012), PoLLy (Grosser e Zheng, 2010; Grosser *et al.*, 2011, 2012), PPCG (Verdoolaege *et al.*, 2013), KernelGen (Mikushin *et al.*, 2014; Mikushin e Likhogrud, 2012; Mikushin *et al.*, 2013) and Polly-ACC (Grosser e Hoefler, 2016).

With the goal of facilitate the development and attract users from other contexts and languages, and for their applications can use the available resources on these platforms, *bindings* and libraries have been developed. Among the *bindings* are PyCUDA and PyOpenCL (Klöckner *et al.*, 2012; PyCUDA, 2012) for Python, and JCuda (JCuda, 2012; Yan *et al.*, 2009) and JOCL (JOCL, 2012) that are *bindings* for Java. Besides these options, the manufacturers have invested on own solutions that facilitate the use of their platforms, Aparapi (Aparapi, 2011) is a API that convert at runtime Java bytecode to OpenCL, enabling code execution on CPUs and GPUs. The CUDA has been widely used in heterogeneous platforms with GPUs, which motivates the development of tools such as CU2CL (CUDA-to-OpenCL) (Martinez *et al.*, 2011) that are the source-to-source compilers, which do the code translation to other models, in this case OpenCL.

In this context, our work is directly related to the automatic parallelization approach and is composed of two parts: *compilation tool* and *runtime*. The focus this thesis is presents the runtime whose main goal is to provide support to OpenMP applications, choosing according to the *operational intensity* the appropriate code version in the offloading process to other devices at runtime.

The following section discusses the research problem and the context of this work. The proposed approach and methodology are described in the Section 1.2. The objectives of this thesis are presented in Section 1.3 and the main contributions of this work are in the Section 1.4. Finally, the thesis organization is described in the Section 1.5.

1.1 Research Problem

Nowadays the programming is not more restricted to Computer Science contexts. Applications are present in all fields of Science ranging from engineering to medicine. In the context of these areas, especially in engineering, there are several legacy code applications that were written in C or Fortran languages and for others platforms such as MPI.

There is a wide gap between the applications needs and the potential for parallel execution introduced by modern platforms. These platforms bring together different processing elements and various technologies, which makes these platforms increasingly heterogeneous.

The current scenario has in one side platforms with multi-core systems, coprocessors (Intel , 2013; Newburn *et al.*, 2013) and GPUs (NVIDIA, 2013) that provide great processing power, and in the other side legacy code applications that are still used. One of the main challenges of these modern platforms is the coexistence with other platforms, technologies and programming languages that are consolidated and widely used.

From the point of view of the applications, whether a new application or legacy code applications, they will always require improvement in performance. Improvements that can be achieved by exploiting the potential parallelism hidden in their code by using the resources available on these platforms. But these computing resources are almost inaccessible in some cases, due to the related complexity to develop new applications or translate existing applications code for these modern platforms.

This scenario was also described by Asanovic *et al.* (2009), the authors defend the need for Structural and Computational Patterns in Parallel Computing. These patterns improve the portability of parallel programming applications, while also allowing specialization of applications with the same behavior. In some situations it is not a matter of achieving the highest possible performance, but instead of making the application achieve acceptable performance, benefiting from a new parallel computing platform (Asanovic *et al.*, 2009).

In the same way this situation is reported as needed for code modernization (Intel, 2015b). Normally, the legacy code applications have no advantages in the use of new available resources when executing in these modern platforms. The authors give reasons and present advantages and a methodology to reach this goal.

Given these new platforms and programming models, there few direct support for the code of existing applications to be rewritten. The code rewriting process is very expensive in terms of time and it is often impractical. The coexistence with legacy code applications requires solutions that support the execution of applications in new or preexisting models and platforms.

Writing new applications or translating legacy code applications for these new platforms is a non trivial task. In order to facilitate the development and attracting developers from other contexts, languages bindings and libraries have been developed, enabling the use of accelerators in the widely used programming languages.

In general, developers for these platforms experience the same difficulties found in other parallel programming models. The hardware available does not always provide a simple and friendly Application Programming Interface (API), on contrary it usually requires the learning a new language and a new development paradigm. The main difficulty in some cases is on the provided parallel programming model.

Aiming to overcome this difficulty, projects has moved efforts to provide mechanisms that facilitate the development of applications. Tools generate code for multi-core or GPUs based on OpenMP or OpenACC directives that are added in the original code, or discovering the parallelizable regions automatically. These tools help the developer in the abstraction of solutions and the obtained results have been satisfactory, but are far from ideal.

Ideally, the connection between these contexts should be made by compilation tools that transform and translate the code applications in more efficiently versions that works using new features of these platforms. Working in code modernization to use the vectorization support in basic platform, or generating a parallel versions in the cases that the workload exceeds its capacity.

In the context of our runtime, there are versions of the parallelizable code regions and we need to choose the better or appropriate device to execute. The code offloading decision is taken at *runtime* using the *operational intensity* that is calculated during the part of code execution.

1.2 Proposed Approach and Methodology

Our proposal is related with automatic parallelization and offloading of legacy code applications. Normally, they are applications that were written in C/C++ and Fortran. Parallelization of this kind of applications consists in trying to parallelize code that may have not been written considering a parallel model. For obtain this result our proposal includes a parallelizing compiler (Wolfe, 1996) that should be able to detect parallelizable regions in the code and generate parallel versions for it. Moreover, for code offloading we need a runtime to execute the code and choose the best version of code considering the available devices on platform.

Thereby our proposal is divided naturally into two basic parts: *compilation tool* to prepare the input code with multiple versions for parallel regions, and a *runtime* that can explore heterogeneous elements using operational intensity to decide about code offloading. These two parts are described in following sections, we present the compilation tool work flow idea, which will be discussed in future works. The focus is on the *runtime* behavior, how it works intercepting calls to OpenMP runtime and how to take decisions in the code offloading process.

The main goal of runtime is to provide support to OpenMP code execution that cover the multi-core systems and try code offloading to accelerators devices, as coprocessors or GPUs. The generated OpenMP code have functions with code versions of parallel loops and a table with pointers to these alternatives functions. The appropriate version will be chosen possibly doing offloading for other devices at runtime.

The decision between code offloading or continuing the execution in multi-core CPUs is based in *operational intensity* that is calculated using measures of performance counters.

For offloading proposes, the runtime needed to control the execution of threads, and decide when to make the offloading and what is the better device to execute the code. Initially, the execution starts with OpenMP code, which will have some of their calls intercepted by the control of the execution of threads in CPU.

The runtime libraries intercept calls made by application to *OpenMP* runtime. For *hooking* OpenMP calls we developed a library *(libhookomp)* that is described in Chapter 3.

The interception mechanism used is a hooking technique applied at Application Binary Interface (ABI) level. Our library can be pre-loaded and it supports the interception of some kinds of calls to OpenMP runtime. It control the threads execution interacting with the other runtime libraries.

For each parallel region a time of threads is created by OpenMP runtime. During the execution one thread of OpenMP threads team is registered when it enter in loop. The thread collects the values of Performance Counters at runtime and take the decision about code offloading.

The offloading is decide by another library *(librooftine)* that implements the Roofline Model (Williams *et al.*, 2009) concepts and it has being used to decide about the code offloading. A basic concept in the *Roofline Model* is the *operational intensity*, which is defined as the relation between operations and memory traffic in bytes (number of operations divided by memory traffic) (Williams *et al.*, 2009).

With these collected performance counters values are calculated the *operational intensity*, *attainable performance* and the *execution time* for each considered device. Our decision model takes the decision about the code offloading and choose the better device to execute the code.

Currently, our runtime is able to choose the better device (multi-core or GPU) according the operational intensity of code. We are getting measures in all levels of memory hierarchy to know the amount of bytes that are moved to execute the code. As our work try offloading to GPU, it is needed to consider the data transfers between host memory and GPU global memory.

1.3 Objectives

This work has as main objectives:

Study the use of performance counters and operational intensity on offloading decision

The most tools that try to make the code offloading are compilation tools. They use the size of data for decide about the offloading. The decision is normally a static decision that is taken in compilation time for generate the code for the target or using the data sizes during the execution.

This work aims using the operating intensity that is calculated dynamically at run time based on performance counters.

Provide mechanisms for offloading in heterogeneous platforms

Modern platforms become increasingly heterogeneous, being composed of the various processing elements, so the exploration and effective use of available resources is needed for performance improvements.

The study of strategies and mechanisms that can decide about the offloading at run time using the dynamic behavior of applications execution on this platforms are welcome in this new scenario.

Create a runtime able to decide about code offloading at run time

The libraries of runtime are working transparently, if the input code is prepared with the alternative functions for devices. The interception library together the roofline library is able to control the threads and take the offloading decision.

1.4 Main Contributions

Our contribution is in the use of operational intensity and a simple decision model at run time. The runtime libraries are intercepting the applications calls to OpenMP runtime, controlling the execution of threads and collecting values of performance counters to calculate the operational intensity. The main contributions of this work are:

The use of operational intensity in offloading decision at run time

The experiments have shown that the operational intensity can be used in decision about offloading. Even with some hardware restrictions, it is possible the use of a strategy for get measures of performance counter values.

The development of runtime library for hooking the OpenMP applications and support the code offloading

In addition to using the library for offloading, the strategy can be used for control the OpenMP threads for monitoring and create traces.

The development of runtime library to collect and implements a simple model to make the decision about offloading

The library that implements Roofline Model concepts use a simple model to make the decision about the offloading. As the decision is taken at run time overhead of model must be minimum.

1.5 Thesis Organization

In this chapter we introduced our research problem, the context of our proposal, as well as the objectives and contributions of this work. The related works are in Chapter 2. In the Chapter 3 is showed the method for intercepting calls to OpenMP runtime that was used on hooking library development of the our runtime. The Chapter 4 describes how to the operational intensity can be used in making decision on code offloading for accelerators devices. The experiments and results about the using of our runtime are presented in the Chapter 5. We present the next steps and future works in the Chapter 6. Finally, in Chapter 7 are discussed some conclusions obtained with the implementation and experiments with our runtime.

Chapter 2

Related Works

This chapter presents the works that are related to the research developed in this thesis. Most of the tools that make offloading or generate code to execution of parallel phases on accelerators devices are in the compilation, optimization and code generation area, are *Compilers*.

Some compilation tools and runtimes make code offloading to accelerators devices. These include the `Par4All` (Amini *et al.*, 2012), the `PPCG` (Verdoolaege *et al.*, 2013), the `KernelGen` (Mikushin *et al.*, 2014; Mikushin e Likhogrud, 2012; Mikushin *et al.*, 2013) and the `Polly-ACC` (Grosser e Hoefler, 2016). These tools generate parallel code for `GPU` detecting parallelizable regions without the needed to do any modifications in the original code. Some of them have used compilation directives and other tools try to detect automatically the *SCoPs* that are targets of optimization.

The main idea of our approach is that values of performance counters can be collected at run time and applied to a decision model then this model is able to decide on the code offloading based on *operational intensity*. The operational intensity has been used to study the behavior of programs on platforms, and which optimizations can be applied in the code to improve the performance (Williams *et al.*, 2009).

The hooking technique that we are using to intercept the `OpenMP` calls is also used by others works for logging and trace proposes (Trahay *et al.*, 2011).

The tools `StarPU` (Augonnet *et al.*, 2010) and `Apollo` (Sukumaran-Rajam *et al.*, 2014), which are tools that try to execute code in platforms multi-core with `GPUs`, also use alternative functions for the code offloading.

The code offloading to accelerators devices has been covered in `OpenMP 4.0` specification (OpenMP-ARB, 2013, 2015) and in the `OpenACC` standard (OpenACC, 2011, 2013, 2015a). But, the offloading must be defined explicitly by developer using compilation directives.

2.1 Compilation Tools and Runtimes for Offloading

Most of the works that concern the use of accelerators devices are compilation tools. Many make the source-to-source translation C or Fortran code to parallel versions to `CUDA` (NVIDIA, 2015a, 2014a) and `OpenCL` (Khronos, 2013).

These tools are related with the automatic code parallelization approach. Where these tools apply models and techniques to automatically detect which regions of code are parallelizable. Most of the tools use the *Polyhedral Model* (Bastoul, 2004) (Benabderrahmane *et al.*, 2010) (Grosser e Simbürger , 2014) or equivalent concepts.

The C-to-CUDA (Baskaran *et al.*, 2010) is a implementation of a transformation system that generates parallel CUDA code, which is optimized for data accesses, from input sequential C code, for regular (affine) programs.

Par4All (Amini *et al.*, 2012) is a source-to-source compiler that integrate several tools for optimize and parallelize sequential code written in C and Fortran languages. It is not based on *Polyhedral Model* but it uses an abstract interpretation for array regions (Creusillet e Irigoin, 1996). The tools are integrated using Python scripts, the main script p4a implements a optimization and parallelization process flow using the PIPS (Amini *et al.*, 2011) that available in the PYPS binding for python (Guelton *et al.*, 2011). The Par4All generates code for multi-core (OpenMP), GPUs and some embedded systems platforms. In the case of the generated GPU code the NVIDIA compiler (nvcc) and libraries of CUDA SDK are used to generate the final code.

The PPCG *(Polyhedral Parallel Code Generation)* (Verdoolaege *et al.*, 2013) is other *source-to-source* compiler that generates OpenMP, CUDA and OpenCL code using *polyhedral* model concepts. To extract the model characteristics (iteration domain, accesses relations and schedule), the PPCG uses the pet (Verdoolaege e Grosser, 2012). It uses the library isl *(Integer Set Library)* (Verdoolaege, 2010) to create the representations and apply operations on the *polyhedral model*. The PPCG creates a new *schedule* for the code exploring the *tiling* transformation, it uses in this process an algorithm based on PLuTo (Bondhugula, 2012; Bondhugula *et al.*, 2007, 2008) that is implemented using the isl libraries.

The code generated by PPCG is divided em two parts, one is the main code to execute in the host and other part is the GPU kernels. The loop iterations mapping on grids, blocks and threads are made according the parameter tile size. Nested loops are considered by pairs, the out most external (tile loop) iterates on *tiles* and the inner most (point loop) iterates inside the tiles. The external loops are mapped to blocks of a grid, while the internal loops are mapped to threads of on block (Verdoolaege *et al.*, 2013).

The KernelGen (Mikushin *et al.*, 2014; Mikushin e Likhogrud, 2012; Mikushin *et al.*, 2013) integrates other tools to generate GPU kernels automatically. This tool is based on LLVM projects (Lattner, 2008; Lattner e Adve, 2004), among the used projects are PoLLy (Grosser *et al.*, 2012) and the NVPTX *backend* (NVPTX, 2013) that generates PTX code. The KernelGen is part of the automatic detection of parallelizable regions approach.

In contrast to the traditional execution model, in which the code runs in the CPU is responsible for controlling the execution flow launching the execution of kernels and the data transfer between the host memory and the device memory, the KernelGen model is centered in the GPU, the main code executes in GPU side.

According the Mikushin *et al.* (2014) when the application is starting all the control and kernels are loaded to GPU, because the authors consider that the execution is more efficient in the GPU, even serial code parts. In this execution model takes advantage of the availability of data in the GPU memory, the data transfers costs are avoided. However the launch of the others kernels execution and the code execution in CPU side is made using callbacks that triggers the execution and launch other functions.

The PoLLy (Grosser e Zheng, 2010; Grosser *et al.*, 2011, 2012) is a polyhedral model implementation for the LLVM project. PoLLy implements a set of passes for the LLVM optimizer that detects and translate parts of the code to the polyhedral model representation. In this representation the analysis and code transformations can be applied and from this representation the LLVM-IR code is optimized and parallel versions can be generated. PoLLy is able to generate OpenMP code versions of parallelizable loops. The generated code have the two versions and according of the size of data the execution is switched.

Polly-ACC (Grosser e Hoefler, 2016) is the most recent work related with PoLLy, it is a heterogeneous compiler that starts the compilation from a sequential code and automatically generate a hybrid executable that transparently offloads suitable code regions.

The idea of provide alternative functions have been used by StarPU (Augonnet *et al.*, 2010) and Apollo (Sukumaran-Rajam *et al.*, 2014), which are tools that try to execute code in platforms multi-core with GPUs.

2.2 Roofline Model

The *operational intensity* is a concept well known and has been used to study the behavior of programs code on platforms, and which optimizations can be applied in the code to improve the performance.

The original *Roofline Model* (Williams *et al.*, 2009) proposes to calculate the operational intensity (I) using measures of number of arithmetic instructions (W) and memory traffic (Q) between Main Memory and the Last Level Cache (LLC). What may be not appropriate for on the current processors architectures. For multi-core and different organizations of memory is necessary to obtain measures considering all levels of the memory hierarchy (LLC, L2 and L1).

The original *Roofline Model* (Williams *et al.*, 2009) and the extensions use the *Operational Intensity* (I) that is the relation between *Work* (W) in terms of floating pointing operations and DRAM *traffic in bytes* (Q), where $Q = Q_{read} + Q_{write}$ (Equation 2.1).

$$I = \frac{W}{Q} \tag{2.1}$$

With the operational intensity is possible to calculate the Attainable Performance using the Equation 2.2.

$$AP = \min\{FLOPS, MB \times I\} \tag{2.2}$$

Where:

AP: Attainable Performance in GFlops/sec.

FLOPS: Peak Floating Point Performance.

MB: Peak Memory Bandwidth.

I: Operational Intensity.

The Peak Floating Point Performance and Peak Memory Bandwidth can be obtained from devices manuals or using benchmarks.

The Roofline Model Graph shows Figure 2.1.

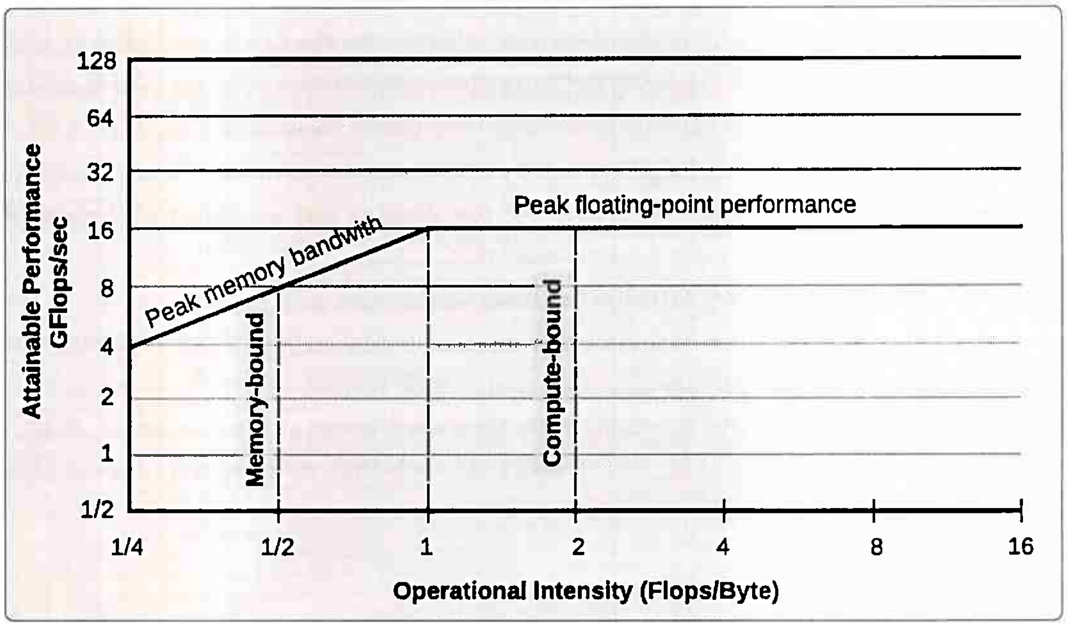


Figure 2.1: *Roofline Graph Example. Adapted from Williams et al. (2009)*

In this graph when the application code have operational intensity equal 1 this means that the amount of bytes that were moved in memory hierarchy is equal of the number os floating point instructions executed in the code.

Operational intensity less than 1 characterizes the code as *memory-bound.* Otherwise, when the operational intensity is more than 1 the code is considered *compute-bound*, i.e have more instructions than memory operations.

The work of Ofenbeck *et al.* (2014) shows how to apply the Roofline Model in modern processors. The authors assert that the W is a property of the chosen algorithm and does not depend on the platform. And Q is dependent of platform features such as cache levels (memory hierarchy), and in the most cases Q can be only estimated, the exact values have to determined by measurements.

The gemm can be used to exemplify the idea. The code is showed on Code 2.1, the code execute the $C \leftarrow \alpha AB + \beta C$.

```
DATA_TYPE *A = (DATA_TYPE *) malloc(NI * NK * sizeof(DATA_TYPE));
DATA_TYPE *B = (DATA_TYPE *) malloc(NK * NJ * sizeof(DATA_TYPE));
DATA_TYPE *C = (DATA_TYPE *) malloc(NI * NJ * sizeof(DATA_TYPE));

void gemm(DATA_TYPE * A, DATA_TYPE * B, DATA_TYPE * C) {
  int i, j, k;
  for (i = 0; i < NI; i++) {
    for (j = 0; j < NJ; j++) {
      C[i * NJ + j] *= BETA;
      for (k = 0; k < NK; ++k) {
        C[i * NJ + j] += ALPHA * A[i * NK + k] * B[k * NJ + j];
      }
    }
  }
}
```

Código 2.1: *The Sample code gemm*

In this code we have three matrices (A, B and C). Each matrix have N^2 elements, considering $NI = NJ = NK$. So N^2 elements of $DATA\_TYPE = double$ (8 bytes) is $8 \times N^2$, that is $24N^2$ of space in memory to store three structures. The A, B and C are *read* and only C is *read* and *written*.

The `line 9` C is updated with β constant ($C[i*NJ+j] = BETA*C[i*NJ+j]$), this line is executed $NI \times NJ$ times, this results in $NI \times NJ$ *reads* and $NI \times NJ$ *writes* to cache. Depending of cache write policies, at this point only reads (compulsory reads) are made from main memory to read C elements ($NI \times NJ$ reads) or writes to main memory are made ($NI \times NJ$ writes to update C elements with β constant multiply).

To complete the operation, at `line 11` the C is in cache, and the A and B elements need to be read. The expanded operation ($C[i*NJ+j] = ALPHA*A[i*NK+k]*B[k*NJ+j]+C[i*NJ+j]$) use the current element of C in cache and read the A and B elements. The value is computed and stored in C.

In general, looking at the code statically the operation of $C \leftarrow \alpha AB + \beta C$ the code read the matrices A, B and C, and write elements in C. The amount of bytes on memory traffic between main memory and the last level cache will be $3 \times N^2 \times 8 = 24N^2$, considering the size of double as 8 bytes. The total of arithmetic operations is 3 *multiplies* and 1 *add*.

$$\begin{aligned} W(n) &= 2N^3 + 2N^2 \\ Q_r &= 24N^2 \\ Q_w &= 8N^2 \\ Q(n) &= Q_r + Q_w \\ Q(n) &= 32N^2 \\ I &= \frac{2N^3 + 2N^2}{32N^2} \end{aligned} \quad \Longrightarrow \quad \begin{aligned} I &= \frac{2(N \times N^2) + 2N^2}{32N^2} \\ I &= \frac{2(N \times \cancel{N^2}) + 2\cancel{N^2}}{32\cancel{N^2}} \\ I &= \frac{2N + 2}{32} \\ I &= \frac{N + 1}{16} \end{aligned}$$

In this sample code, $N \geq 15$ is enough to the code be classified as *compute-bound* in the considered device.

It is correct, but to obtain more accurate measures is necessary to know about the final code generated to target platform, considering platforms that have different kind of instructions sets. For example, the multiplication instruction can not exist on a target platform, being necessary to carry out the operation with successive additions. Which leads to a mapping instruction $1 : N$ (one instruction on algorithm level, generate N instructions on target machine code).

Statically can be difficult to know about all the memory access, mainly in loops with unknown limits. In some cases, it is not known the amount of iterations to be executed by loop. This information only appear during the execution and when you have the dynamic behavior of code.

The cache policies to write data can be:

Write-Trough. The data is updated in the cache and in the main memory. The performance may be degraded due to latency Main Memory, as each written in the cache generates a write in the Memory that is slower.

Write-Around. is a similar to write-through, but write I/O is written directly to Main Memory, bypassing the cache.

Write-back or Copy-back. The data is written directly to cache and at the final the data is updated on Main Memory.

Konstantinidis e Cotronis (2015) introduced the *Quadrant-Split* model as a derivation of Roofline Model and an alternative that provide insights on the performance limiting factors of multiple devices to the same kernel in the same graph.

Another approach used by Lorenzo *et al.* (2013) was named of *Dynamic Roofline Model* (DyRM) and divide the code in slices to measure each one, and creates rooflines to each part and combines all at the final in one graph trying to capture the code behavior in a more dynamic manner. Measuring parts of the code show the code evolution during the execution (Lorenzo *et al.*, 2014).

The recent works have proposed extensions to original Roofline Model, which consider all levels in the memory hierarchy to get the traffic, and calculate the operational intensity counting other instructions in addition to arithmetic instructions (Aleksandar Ilic e Sousa, 2015; Ilic *et al.* , 2014) (Lo *et al.*, 2015). The work of (Caparrós Cabezas e Püschel, 2014) presents an extension of the roofline model that provides a more detailed bottleneck analysis considering other hardware parameters.

To access events and metrics to create the rooflines graphs, the most of works have been used performance counters that are captured during the program execution. The performance counters values are processed and analyzed using the Roofline Model concepts after the execution and statically.

2.3 Performance Counters

To calculate the amount of floating point instructions that were executed and the memory traffic, we must have access to devices performance counters. These counters are available as events and metrics. To access events and metrics on Intel CPUs we can use the PCM (Intel, 2015c) and for NVIDIA GPUs the CUPTI can be used (NVIDIA, 2014b, 2015b). In addition to these libraries of manufacturers there are other tools such as PAPI (Mucci *et al.*, 1999) which help in abstraction and facilitate the capture and use of performance counters across systems as different processors.

The IA-64 and IA-32 architecture processors have a Performance Monitoring Unit (PMU). This unit have a set of Model Specific Register (MSRs). The PMU counters and counter control registers are implemented using these registers. The access is made by special instructions (RDMSR and WRMSR) and according of the privilege level in which the application is running. The sampling features have been improved with Precise Event-Based Sampling (PEBS) (Intel, 2015a) in PMU. The mechanism enables the PMU to capture the architectural state and IP at the completion of the instruction that caused the event.

In the processors of Intel Sandy Bridge and Ivy Bridge architectures have available 11 hardware performance counters per core. Of these 3 are fixed counters for core cycles, reference cycles and core instructions executed. The other 8 are programmable counters with minimal restrictions.

In the processors with enabled Hyper Threading support the 8 programmable counters are divided and turn into 4 per thread. Since each logical processor is considered by the Operating System as a physical processor and each thread must maintain the execution context, the measures per thread will use own hardware counters. Further, when the hardware platform run with non-maskable interrupt (NMI) timer active, one of the four remaining counters can be used for this control. At the final the number os available counter is 3 per thread. In the use of PAPI is only guaranteed 3 programmable counters at a given time, in addition to the 3 fixed counters.

Most tools utilizes the same method to obtain the measurements. The code is instrumented with functions calls that are inserted by programmer to initialization of counters before the region of interest and after others calls are inserted to retrieve the events and metric values.

The GNU/Linux `perf` is currently implemented as a kernel module that permits non intrusive measurements because the application code is not instrumented to make measurements. The events and metrics are observed by the kernel during the application execution.

In terms of portability, the PAPI is presented as a good option, because it covers CPUs of different architectures and manufacturers, and in addition it has support for components. It is able to access the measures of native events available in the `perf`. It have another component for CUDA (cuda), which it uses the CUPTI to retrieve counters, metrics and events of NVIDIA GPUs.

2.4 Hooking

The hooking technique that we are using to intercept the OpenMP calls is also used by others works for many proposes, including debugging and monitoring devices and Operating System modules, creating logs and traces of runtimes and applications execution (Trahay *et al.*, 2011) (Mohr *et al.*, 2002). Basically, it can be used to extend functionalities by code injection loading shared libraries.

The EZTrace (Trahay *et al.*, 2011) intercepts by hooking not only OpenMP libraries. It is a trace generator for applications of HPC context, including hybrid applications with MPI, OpenMP and CUDA. The EZtrace generates input files for trace visualization tools.

Mohr *et al.* (2002) proposes the POMP interface with directives and functions to allow OpenMP code instrumentation catching events of parallel regions, loops, synchronization and runtime libraries. But this work does not use interception, it is need to use directives to annotate the code.

2.5 OpenMP Support for Offloading and Accelerators

The OpenMP 4.0 specification (OpenMP-ARB, 2013, 2015) covers the use of accelerators. The GNU `libgomp` (GNU Libgomp, 2015a,b, 2016a,b) implements the support for offloading devices using the OpenACC (OpenACC, 2011, 2013, 2015a).

The execution model is like a classical launching of GPU kernels, it is host-centric execution and the host device offloads target regions to target devices. Threads can not migrate from one to another device.

When a target construct is encountered, the target region demarcated with compilation directive `#pragma omp target` is executed by the implicit device task. The other tasks that encounter the target construct waits at the end of the construct until execution of the region completes. In the same way of the directives if the implementation have no support to target device or the target device does not exist, the generated code for all target regions are executed by the host device.

This execution model behavior and characteristics are the same used in OpenACC.

2.6 Final Considerations

Regarding that generate code for heterogeneous devices, the most part are compilers. The final code is generated in most of cases separately or code to CPU or to GPU. The works that try to decide in execution time have considered only the size of data.

The Roofline Model concepts as operational intensity are being used statically, in most cases they are used in the study of programs behavior during the execution. The results of execution are processed and the optimizations are applied in the code. The optimized code is executed again to obtain new results. These works do not use operational intensity dynamically and at run time.

The programming models of `OpenMP` and `OpenACC` that cover the offloading for legacy code need the annotations in the code using compilation directives. The user needs to declare explicitly the code regions that will be offloaded.

Chapter 3

HOOKOMP: Hooking calls to OpenMP Runtime

In this chapter are presented the concepts and the study about `OpenMP` compilation directives implementation that were used for development of our interception library – `HOOKOMP`. The main goal this part of work is provides support to `OpenMP` applications execution. The idea is that this part of the model, which is under the responsibility of the proposed runtime, is able to execute the prepared input code and intercept the application calls to `OpenMP` runtime.

We have adopted the use of `OpenMP` code format for runtime input code. This decision satisfies one of the initial project requirements, that was to generate code for *multicore* and accelerator devices. This decision complies with the requirements and it has contributed with a well structured and well defined code format. The format facilitates the dynamic identification of code blocks such as *parallel regions* and *loops*. These blocks are the structures that are intercepted by our library. This library works together the `libroofline` library that is described in the Chapter 4.

The input code format is flexible enough to allow that it can be generated by the compilation tool or written manually using `OpenMP` directives. It only needs to be prepared with alternative functions for parallelizable regions.

The *interceptable format* is an `OpenMP` code post directives expansion. This kind of code have no compilation directives, it is the application code before the linking process. The application code with calls to defined functions of the `OpenMP` runtime Application Binary Interface (ABI).

Considering that the application have alternatives versions of parallelizable code regions. Our runtime will control the execution intercepting the code and getting the measures of the performance counters. So, it will can decide about the offloading to other devices according the *operational intensity* of the considered code. The better device and the appropriate version will be chosen in the offloading process *at runtime*. We are considering accelerators devices like GPUs or coprocessors.

3.1 Study of Compilation Directives and Expanded OpenMP Code

The compilation directives is one of the approaches that has excelled in the context of parallel computing, because annotating code is usually easier than rewriting it. `OpenMP` implements the `fork-join` model, in which multiple threads of execution perform tasks defined implicitly or explicitly by `OpenMP` directives (OpenMP-ARB, 2011, 2013, 2015).

Compilation directives are used as annotations that provide tips about the original code and guide the compiler in the parallelization process of candidate regions. These directives are commonly implemented using the pre-processing directives `#pragma`, in `C/C++`, and sentinels `!$`, in the `Fortran` language. During the pre-processing, the *tokens* that compose the directive are replaced by the expanded code with calls to `OpenMP` runtime, if the annotated code is analyzed by a compiler that supports `OpenMP` (Dagum e Menon, 1998; OpenMP API Site, 2012; OpenMP-ARB, 2011, 2013, 2015), such as `GCC` (GCC, 2015; GNU Libgomp, 2015a) and LLVM `clang` (Lattner e Adve, 2004; LLVM Clang, 2015; LLVM OpenMP, 2015). Each directive is expanded in a code with specific format.

From the annotated blocks, the runtime can generate implicit threads to execute the parallel regions or assign to some other thread the execution of explicit tasks defined by the programmer using the `task` directive.

As we want to intercept `OpenMP` code, a study about the implementation of compilation directives was made in the `libgomp` (GNU Libgomp, 2015a,b, 2016a,b, 2015c), the `OpenMP` implementation for `GCC`. The library recently had the name changed of `GNU OpenMP Runtime Library` to `GNU Offloading and Multi Processing Runtime Library` and it is able to make code offloading using `OpenACC` (OpenACC, 2012, 2015b) (OpenACC, 2011, 2013, 2015a).

We verified the format of code generated by `GCC` with `libgomp` library. As loops parallelization is target of the work, the constructors of *parallel region* and work sharing *(parallel for)* were studied separately and combined. We generated the code in several `GCC` versions (4.8, 4.9, 5.3, 6.1) to verify the format. In this text we adopted the `GCC 4.8`, because this version presented a more consistent format for interception.

Furthermore, as one of the future goals of this research is to develop a compilation tool using `LLVM` projects, we verified the `LLVM PoLLy` (Grosser *et al.*, 2011, 2012) support to generating parallel code for `OpenMP` target. Currently, only the loop schedule of the *runtime* type is generated by `PoLLy`.

3.1.1 Parallel Regions: **`parallel`** constructor

Threads are created from annotated code blocks. The *master thread* executes sequentially until it finds the first parallel region. The similar `fork` operation creates a team of threads to execute a parallel region, the *master thread* is part of the team. Nested parallel regions are allowed. When all threads in a team reach the barrier that marks the end of the parallel region, the team is destroyed and the master thread continues the execution alone until it finds a new parallel region.

Parallel regions are created in `OpenMP` using the `#pragma omp parallel` constructor. This is the most important directive in `OpenMP`, it is responsible for indicating parallelizable code regions. When a parallel region is found, the group of threads that will execute the code in parallel is created. However, this constructor does not divide the work among the threads, it only creates the threads for parallel region. The code format of parallel construct is showed in Code 3.1.

```
#pragma omp parallel
{
  body;
}
```

Código 3.1: *Parallel directive format*

This directive is implemented with the creation of a outlined function using the source code contained in `body`. The `libgomp` (GNU Libgomp, 2015a,c) uses the functions to delimit the code region. The two functions related with parallel regions in `libgomp ABI` are listed in Box 3.1.1, these functions are called to delimit the parallel regions.

Box 3.1.1: The LibGOMP ABI – Functions related with parallel directive

```
void GOMP_parallel_start(void (*fn)(void *), void *data, unsigned num_threads)
void GOMP_parallel_end(void)
```

After the code processing, the expanded code assumes the format showed in Code 3.2.

```
/* A new function is created. */
void subfunction (void *data){
  use data;
  body;
}

/* Directive is replaced by calls to runtime for create parallel region */
setup data;
GOMP_parallel_start(subfunction, &data, num threads);
subfunction(&data);
GOMP_parallel_end();
```

Código 3.2: *Expanded code format for parallel directive*

The expanded code generated by GCC for `parallel` directive is showed in Code 3.3.

```
/* A new function is created. */
main._omp_fn.0 (struct .omp_data_s.0 * .omp_data_i) {
  return;
}

main () {
  int i;
  int D.1804;
  struct .omp_data_s.0 .omp_data_o.1;

<bb 2>:
  .omp_data_o.1.i = i;
  __builtin_GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, 0);
  main._omp_fn.0 (&.omp_data_o.1);
  __builtin_GOMP_parallel_end ();
  i = .omp_data_o.1.i;
  D.1804 = 0;

<L0>:
  return D.1804;
}
```

Código 3.3: *Expanded code generated by GCC for parallel directive*

A struct `omp_data` is created to pass arguments for outlined function that implements the parallel region.

3.1.2 Loops: `for` constructor

A team of threads is created when a parallel region is reached, but with only a parallel region constructor all threads will execute the same code. It is necessary distribute and coordinate the parallel execution. The `for` constructor is used to share work among the threads of team that was created in parallel region. The Code 3.4 shows a parallel region with a loop inside, which is equivalent to combined mode of constructors showed in the Code 3.5.

```
#pragma omp parallel
{
  #pragma omp for
  for (i = lb; i <= ub; i++){
    body;
  }
}
```

Código 3.4: *For directive inside a parallel region*

```
#pragma omp parallel for
for (i = lb; i <= ub; i++){
  body;
}
```

Código 3.5: *Combined parallel for directive*

The work is divided according of the adopted scheduling of loop iterations. The scheduling type is defined using *schedule* clause. This clause define how iterations of the associated loops are divided into contiguous non-empty subsets, called chunks (OpenMP-ARB, 2011, 2013, 2015). The scheduling is a manner how these chunks are distributed among threads of the team.

The OpenMP allow the use of some scheduling types: *static, auto, runtime, dynamic and guided.* In addition, we are considering in our analysis the *unspecified* type for situations when the schedule is not specified using the *schedule* clause.

Using the scheduling type *static* the iterations set is divided into sub-sets (chunks) according the *chunk_size* value. The *chunks* are assigned to the threads of team in a *round-robin* fashion respecting the order of the thread number. When the *chunk_size* value is not specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. In this case the *chunk_size* is defined by *number of iterations* divided by *number of threads.* The schedule clause declaration for this type is `schedule(static[, chunk_size])`.

In the scheduling type *dynamic* the iterations are distributed to threads by chunks considering the availability of each thread. The threads request new chunks to execute when finish the previous work. Each thread executes a chunk of iterations, then requests another chunk, until there is no more work to be distributed. The schedule clause declaration for dynamic type is `schedule(dynamic[, chunk_size])`. Each chunk contains *chunk_size* iterations, except for the last chunk, which may have fewer iterations. When no *chunk_size* is specified, the default value is 1.

In the scheduling *guided* the iterations are assigned to threads in chunks as in the dynamic type. The different behavior in the distribution is determined by the *chunk_sizevalue* value that is passed to the OpenMP runtime. If the *chunk_size* is 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. For a *chunk_size* with a *k* value greater than 1, the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than *k* iterations (except for the last chunk, which may have fewer than k iterations). When the *chunk_size* value is not specified, it defaults to 1. The schedule clause declaration for guided type is `schedule(guided[, chunk_size])`.

In the scheduling type *auto* the decision regarding scheduling is delegated to the compiler and/or runtime system. The schedule clause declaration for auto type is `schedule(auto)`.

In the same way, in the scheduling *runtime* the decision about the scheduling is deferred until run time, and the schedule and *chunk_size* are taken from the internal control variable *run-sched-var* (ICV). The schedule clause declaration for runtime type is `schedule(runtime)`. For both *auto* and *runtime* schedules the *chunk_size* value must not be specified.

Similarly to processing of parallel constructor, the `parallel for` directive is also implemented by creating a new outlined function with the source code of the loop. The format of combined directive *parallel for* is showed in the Code 3.6.

```
#pragma omp parallel for num_threads (number_of_threads) schedule ({auto, static
    , dynamic, guided, runtime}, {variable/expression | numerical value/constant
    })
```

Código 3.6: *Combined parallel for directive with schedule, chunk size and number of threds definitions*

When the scheduling algorithm is not specified the compiler generates the code using `libgomp` ABI functions for *static* scheduling. The function calls that delimit the parallel region will surround the outlined function call. The used functions to delimit the parallel code region and construct the loop format are listed in the Box 3.1.2.

Box 3.1.2: The LibGOMP ABI – Functions used on parallel for implementation

```
bool GOMP_loop_static_next(long *, long *)
void GOMP_loop_end_nowait(void)
void GOMP_parallel_loop_static(void (*)(void *), void *,unsigned, long, long, long,
    long, unsigned)
void GOMP_parallel_end(void)
```

According with the `libgomp` manual (GNU Libgomp, 2015c) the expanded code that replace the parallel loop declaration is composed of an outlined function and calls to create the parallel region. The control of loop iterations *chunks* that are executed by each thread is inside the new function. The expanded code format is showed in the Code 3.7.

```
void subfunction (void *data){
  long _s0, _e0;
  while (GOMP_loop_static_next (&_s0, &_e0)){
    long _e1 = _e0, i;
    for (i = _s0; i < _e1; i++)
      body;
  }
  GOMP_loop_end_nowait ();
}

/* Directive is replaced. */
GOMP_parallel_loop_static (subfunction, NULL, 0, lb, ub+1, 1, 0);
subfunction (NULL);
GOMP_parallel_end ();
```

Código 3.7: *Expanded code for parallel for directive*

The Code 3.8 and Code 3.9 show the use of *static* scheduling in two loops. The first loop with loop upper bound assuming a numerical value and the second one using a variable to define the upper bound.

```
#pragma omp parallel for schedule(
    static)
for (i = 0; i < 1024; i++){
  // body.
}
```

Código 3.8: *Static loop with upper bound using value*

```
n = 1024;
#pragma omp parallel for schedule(
    static)
for (i = 0; i < n; i++){
  // body.
}
```

Código 3.9: *Static loop with upper bound using variable*

The code that was generated by GCC is showed in Code 3.10 and in the Code 3.11 that are very similar, but they are differing only in the use of a `struct` to pass the data in the outlined function call.

```
main () {
  /* Variables declaration was suppressed. */

<bb 2>:
  __builtin_GOMP_parallel_start (main._omp_fn.0, 0B, 0);
  main._omp_fn.0 (0B);
  __builtin_GOMP_parallel_end ();
  return;
}

main._omp_fn.0 (void * .omp_data_i) {
  /* Variables declaration was suppressed. */

<bb 11>:

<bb 3>:
  D.1816 = __builtin_omp_get_num_threads ();
  D.1817 = __builtin_omp_get_thread_num ();
  q.1 = 1024 / D.1816;
  tt.2 = 1024 % D.1816;
  if (D.1817 < tt.2)
    goto <bb 9>;
  else
    goto <bb 10>;

<bb 10>:
  D.1820 = q.1 * D.1817;
  D.1821 = D.1820 + tt.2;
  D.1822 = D.1821 + q.1;
  if (D.1821 >= D.1822)
    goto <bb 5>;
  else
    goto <bb 8>;

<bb 8>:
  i = D.1821;

<bb 4>:
  i = i + 1;
  if (i < D.1822)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 5>:

<bb 6>:
  return;

<bb 9>:
  tt.2 = 0;
  q.1 = q.1 + 1;
  goto <bb 10>;
}
```

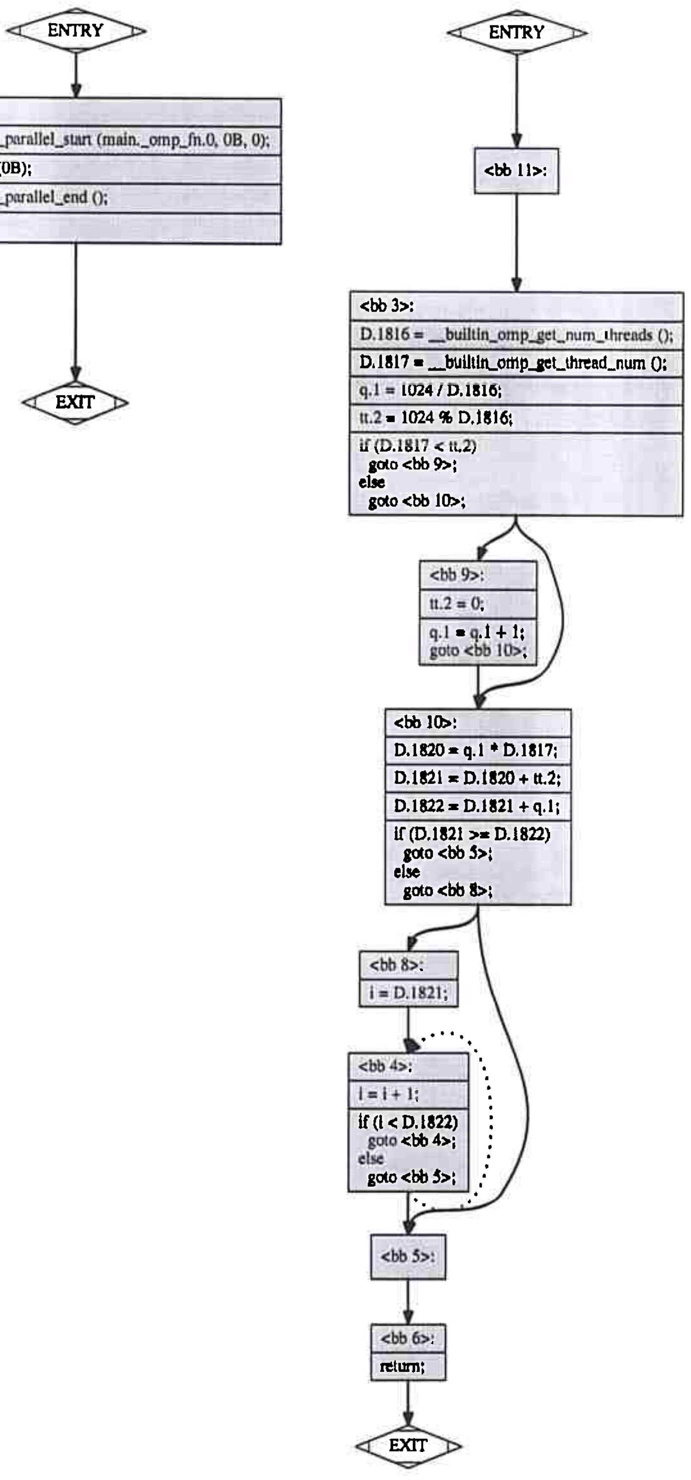
Código 3.10: *Expanded code for parallel for directive*

The code that was generated by GCC for the Code 3.9 is showed in Code 3.11.

```
main () {
  /* Variables declaration was suppressed. */
  struct .omp_data_s.0 .omp_data_o.1;

<bb 2>:
  n = 1024;
  .omp_data_o.1.n = n;
  __builtin_GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, 0);
  main._omp_fn.0 (&.omp_data_o.1);
  __builtin_GOMP_parallel_end ();
  n = .omp_data_o.1.n;
  return;
}

main._omp_fn.0 (struct .omp_data_s.0 * .omp_data_i) {
  /* Variables declaration was suppressed. */
  int n [value-expr: .omp_data_i->n];

<bb 11>:

<bb 3>:
  D.1821 = .omp_data_i->n;
  D.1822 = __builtin_omp_get_num_threads ();
  D.1823 = __builtin_omp_get_thread_num ();
  q.2 = D.1821 / D.1822;
  tt.3 = D.1821 % D.1822;
  if (D.1823 < tt.3)
    goto <bb 9>;
  else
    goto <bb 10>;

<bb 10>:
  D.1826 = q.2 * D.1823;
  D.1827 = D.1826 + tt.3;
  D.1828 = D.1827 + q.2;
  if (D.1827 >= D.1828)
    goto <bb 5>;
  else
    goto <bb 8>;

<bb 8>:
  i = D.1827;

<bb 4>:
  i = i + 1;
  if (i < D.1828)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 5>:

<bb 6>:
  return;

<bb 9>:
  tt.3 = 0;
  q.2 = q.2 + 1;
  goto <bb 10>;
}
```

Código 3.11: *Expanded code for parallel for directive*

The Figure 3.1 is a graphical representation of the Code 3.10.

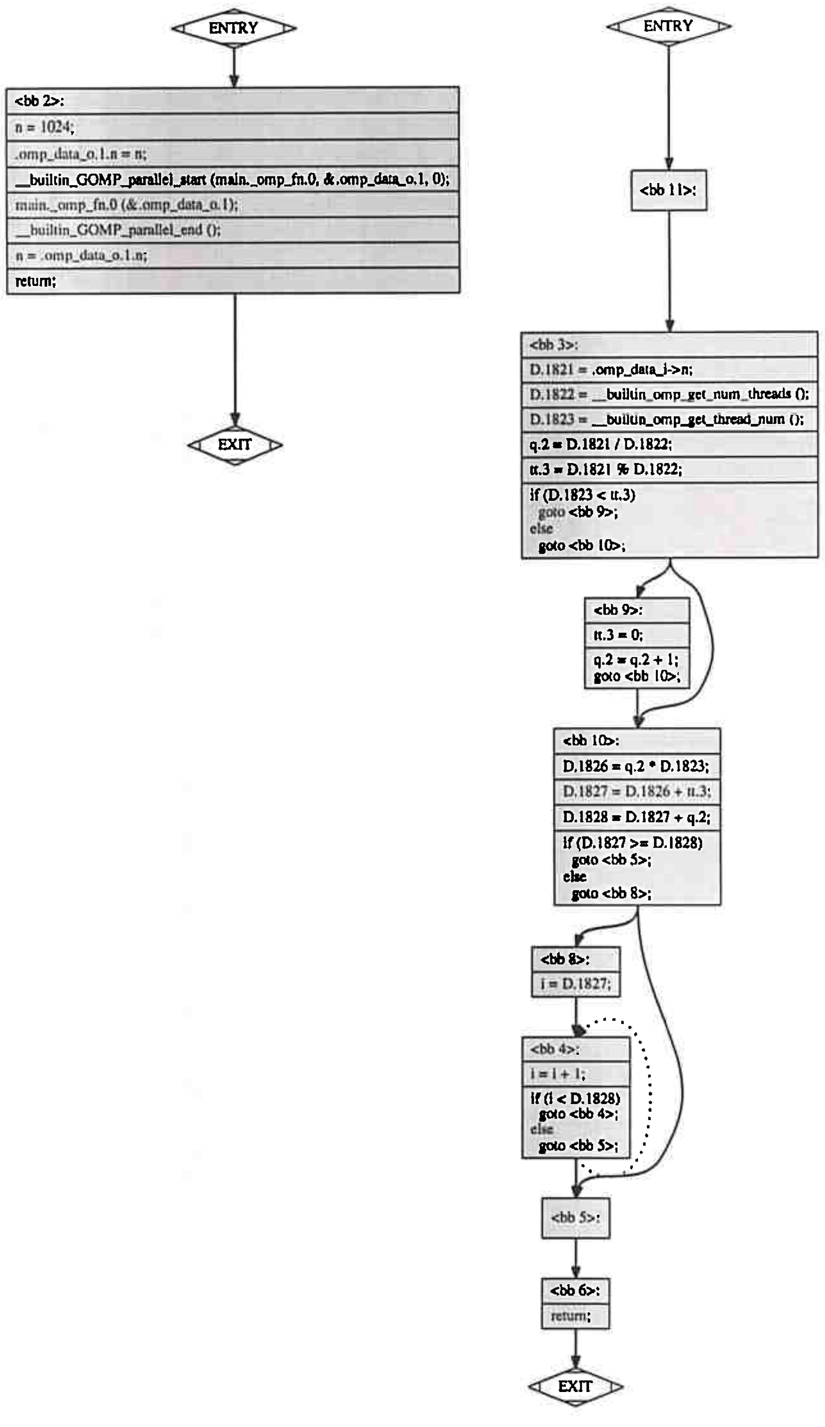


```
<bb 2>:
__builtin_GOMP_parallel_start (main._omp_fn.0, 0B, 0);
main._omp_fn.0 (0B);
__builtin_GOMP_parallel_end ();
return;
```

```
<bb 11>:

<bb 3>:
D.1816 = __builtin_omp_get_num_threads ();
D.1817 = __builtin_omp_get_thread_num ();
q.1 = 1024 / D.1816;
tt.2 = 1024 % D.1816;
if (D.1817 < tt.2)
  goto <bb 9>;
else
  goto <bb 10>;

<bb 9>:
tt.2 = 0;
q.1 = q.1 + 1;
goto <bb 10>;

<bb 10>:
D.1820 = q.1 * D.1817;
D.1821 = D.1820 + tt.2;
D.1822 = D.1821 + q.1;
if (D.1821 >= D.1822)
  goto <bb 5>;
else
  goto <bb 8>;

<bb 8>:
i = D.1821;

<bb 4>:
i = i + 1;
if (i < D.1822)
  goto <bb 4>;
else
  goto <bb 5>;

<bb 5>:

<bb 6>:
return;
```

Figure 3.1: *Visualization of generated code format that uses static scheduling and loop upper bound with numeric value*

The Code 3.11 can be represented graphically by Figure 3.2.



```
<bb 2>:
n = 1024;
.omp_data_o.1.n = n;
__builtin_GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, 0);
main._omp_fn.0 (&.omp_data_o.1);
__builtin_GOMP_parallel_end ();
n = .omp_data_o.1.n;
return;
```

```
<bb 11>:

<bb 3>:
D.1821 = .omp_data_i->n;
D.1822 = __builtin_omp_get_num_threads ();
D.1823 = __builtin_omp_get_thread_num ();
q.2 = D.1821 / D.1822;
tt.3 = D.1821 % D.1822;
if (D.1823 < tt.3)
  goto <bb 9>;
else
  goto <bb 10>;

<bb 9>:
tt.3 = 0;
q.2 = q.2 + 1;
goto <bb 10>;

<bb 10>:
D.1826 = q.2 * D.1823;
D.1827 = D.1826 + tt.3;
D.1828 = D.1827 + q.2;
if (D.1827 >= D.1828)
  goto <bb 5>;
else
  goto <bb 8>;

<bb 8>:
i = D.1827;

<bb 4>:
i = i + 1;
if (i < D.1828)
  goto <bb 4>;
else
  goto <bb 5>;

<bb 5>:

<bb 6>:
return;
```

Figure 3.2: *Visualization of generated code format that uses static scheduling and loop upper bound with variable*

For *unspecified* or *auto* schedule type, the GCC generates the same format that is generated for static *static*.

Using the others schedule types *runtime, dynamic* or *guided* the format of generated code among them is the same, but have two formats depending of loop *upper bound* and *chunk_size* definitions.

If the code have numerical values on these definitions the code have a format. Otherwise, if variables or expressions that need evaluation are used, then the generated code format is another.

```
#pragma omp parallel for schedule(
    dynamic)
for (i = 0; i < 1024; i++){
  // body.
}
```

Código 3.12: *Dynamic scheduling loop with upper bound using value*

```
n = 1024;
#pragma omp parallel for schedule(
    dynamic)
for (i = 0; i < n; i++){
  // body.
}
```

Código 3.13: *Dynamic scheduling with loop upper bound using variable*

The Table 3.1 summarizes the combinations for *upper bound* and *chunk_size* definitions and the code format that is generated. We considered when the assumed value is a *numerical value* or a *constant value* as *value*. And when is a *variable* or an *expression* that need of evaluation as *variable*.

Table 3.1: *Possible combinations for* upper bound *and* chunk_size *definitions*

Upper Bound	chunk_size	Format
value	value	First
variable	value	Second
value	variable	Second
variable	variable	Second
value	unspecified	First
variable	unspecified	Second

The pattern described in the Table 3.1 tells us that the presence of some one definition using a variable or expression is an enough condition for the second code format to be generated. This is due to the need for evaluation to discover of the final value assumed by the variables.

For loops using the *dynamic, runtime* and *guided* schedule types – «schedule_type», the GCC and libgomp use the functions listed in Box 3.1.3 to delimit the parallel region code and to create the *first* loop format.

Box 3.1.3: The LibGOMP ABI – Functions used on *parallel for* first format

```
void GOMP_parallel_loop_<<schedule_type>>_start (void (*fn) (void *), void *data,
    unsigned num_threads, long start , long end, long incr);
void GOMP_parallel_end (void);
bool GOMP_loop_<<schedule_type>>_next (long *istart , long *iend);
void GOMP_loop_end_nowait (void);
```

As it is needed in the *second* format the evaluation of variable or expression the parallel region call is used. The functions that used in second format are presented in Box 3.1.4.

Box 3.1.4: The LibGOMP ABI – Functions used on *parallel for* second format

```
void GOMP_parallel_start (void (*fn) (void *), void *data, unsigned num_threads);
void GOMP_parallel_end (void);
void GOMP_parallel_loop_<<schedule_type>>_start (void (*fn) (void *), void *data,
    unsigned num_threads, long start, long end, long incr);
bool GOMP_loop_<<schedule_type>>_next (long *istart, long *iend);
void GOMP_loop_end_nowait (void);
```

The first format code is showed in Code 3.14.

```
void subfunction (void *data){
  long _s0, _e0;
  while (GOMP_loop_runtime_next (&_s0, &_e0)){
    long _e1 = _e0, i;
    for (i = _s0; i < _e1; i++){
      body;
    }
  }
  GOMP_loop_end_nowait ();
}

/* The annoted loop is replaced. */
GOMP_parallel_loop_runtime_start (subfunction, NULL, 0, lb, ub+1, 1, 0);
subfunction (NULL);
GOMP_parallel_end ();
```

Código 3.14: *Expanded code for dynamic, runtime and guided schedule in the first format*

The second code format is showed in the Code 3.15.

```
void subfunction (void *data){
  long i, _s0, _e0;
  if (GOMP_loop_runtime_start (0, n, 1, &_s0, &_e0)){
    do {
          long _e1 = _e0;
          for (i = _s0; i < _e0; i++) {
            body;
          }
    } while (GOMP_loop_runtime_next (&_s0, &_e0));
  }
  GOMP_loop_end ();
}

/* The annoted loop is replaced. */
GOMP_parallel_start (subfunction, NULL, 0);
subfunction (NULL);
GOMP_parallel_end ();
```

Código 3.15: *Expanded code for dynamic, runtime and guided schedule in the second format*

First format is shown in the Code 3.16.

```
main () {
  /* Variables declaration was suppressed. */
<bb 2>:
  __builtin_GOMP_parallel_loop_dynamic_start (main._omp_fn.0, 0B, 0, 0,
     1024, 1, 1);
  main._omp_fn.0 (0B);
  __builtin_GOMP_parallel_end ();
  return;
}

main._omp_fn.0 (void * .omp_data_i) {
  /* Variables declaration was suppressed. */
<bb 10>:
<bb 3>:
  D.1818 = __builtin_GOMP_loop_dynamic_next (&.istart0.1, &.iend0.2);
  if (D.1818 != 0)
    goto <bb 8>;
  else
    goto <bb 5>;

<bb 8>:
  .istart0.3 = .istart0.1;
  i = (int) .istart0.3;
  .iend0.4 = .iend0.2;
  D.1822 = (int) .iend0.4;

<bb 4>:
  i = i + 1;
  if (i < D.1822)
    goto <bb 4>;
  else
    goto <bb 9>;

<bb 9>:
  D.1823 = __builtin_GOMP_loop_dynamic_next (&.istart0.1, &.iend0.2);
  if (D.1823 != 0)
    goto <bb 8>;
  else
    goto <bb 5>;

<bb 5>:
  __builtin_GOMP_loop_end_nowait ();

<bb 6>:
  return;
}
```

Código 3.16: *Expanded code for parallel for directive with dynamic schedule (first format)*

Second format is shown in the Code 3.17.

```
main () {
  /* Variables declaration was suppressed. */
  struct .omp_data_s.0 .omp_data_o.1;

<bb 2>:
  n = 1024;
  .omp_data_o.1.n = n;
  __builtin_GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, 0);
  main._omp_fn.0 (&.omp_data_o.1);
  __builtin_GOMP_parallel_end ();
  n = .omp_data_o.1.n;
  return;
}

main._omp_fn.0 (struct .omp_data_s.0 * .omp_data_i) {
  /* Variables declaration was suppressed. */
<bb 10>:
<bb 3>:
  D.1822 = .omp_data_i->n;
  D.1823 = (long int) D.1822;
  D.1826 = __builtin_GOMP_loop_dynamic_start (0, D.1823, 1, 1, &.istart0
      .2, &.iend0.3);
  if (D.1826 != 0)
    goto <bb 8>;
  else
    goto <bb 5>;

<bb 8>:
  .istart0.4 = .istart0.2;
  i = (int) .istart0.4;
  .iend0.5 = .iend0.3;
  D.1830 = (int) .iend0.5;

<bb 4>:
  i = i + 1;
  if (i < D.1830)
    goto <bb 4>;
  else
    goto <bb 9>;

<bb 9>:
  D.1831 = __builtin_GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);
  if (D.1831 != 0)
    goto <bb 8>;
  else
    goto <bb 5>;

<bb 5>:
    __builtin_GOMP_loop_end_nowait ();

<bb 6>:
    return;
}
```

Código 3.17: *Expanded code for parallel for directive with dynamic schedule (second format)*

Figure 3.3 shows the graphical representation for the first format showed in Figure 3.16.

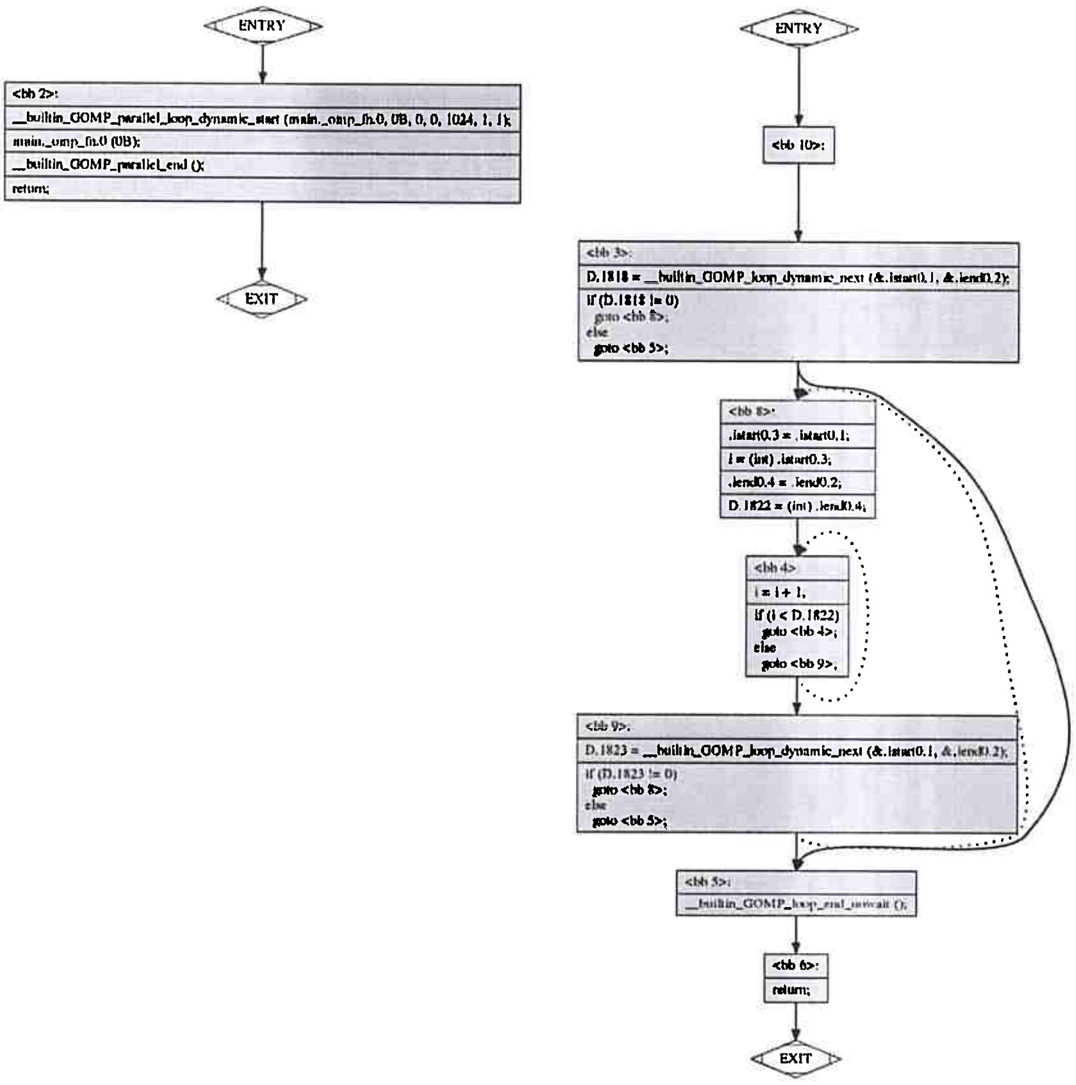
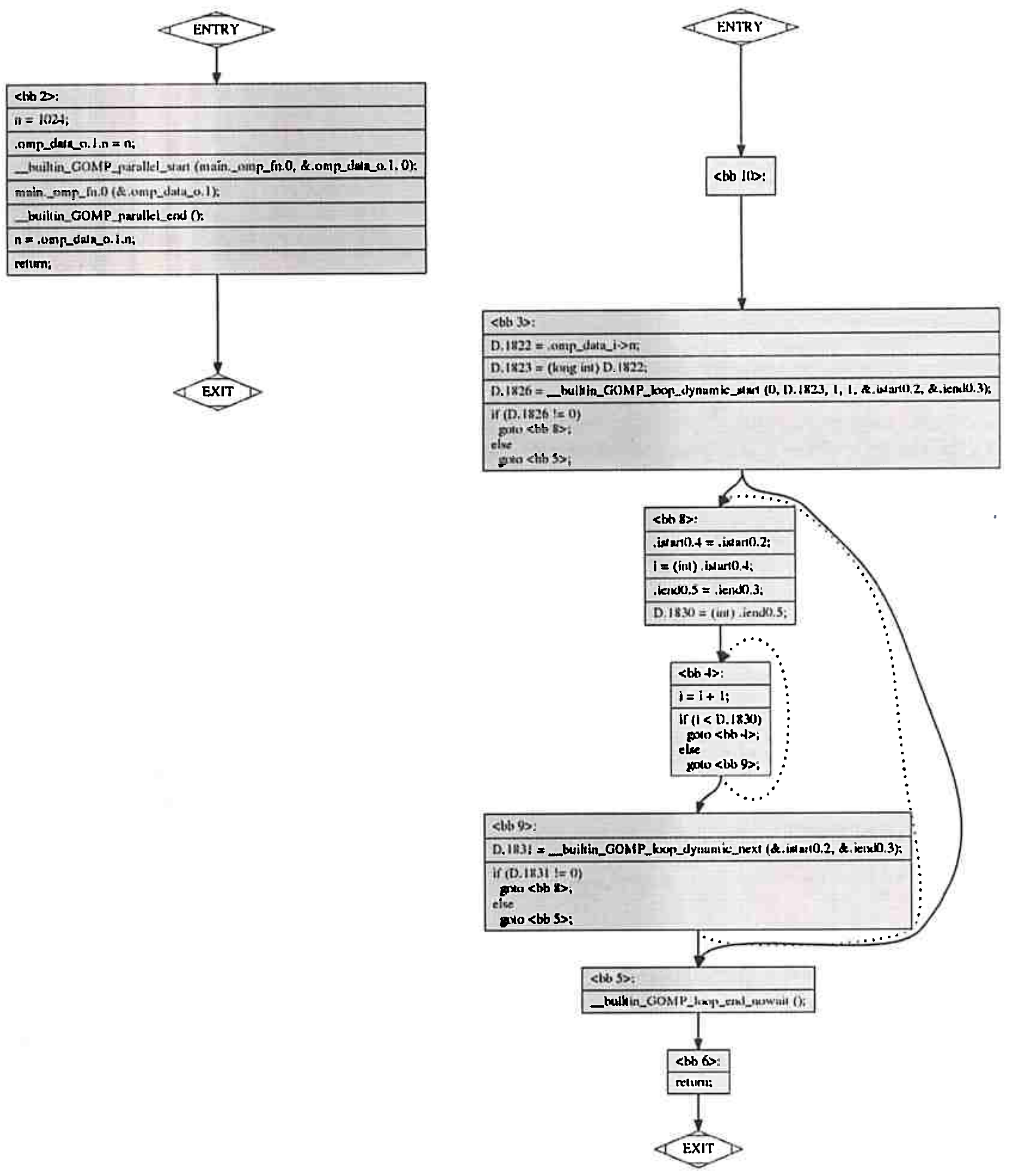


Figure 3.3: *Visualization of generated code format that uses dynamic scheduling (first format)*

The graphical representation for second format that was showed in Figure 3.17 is presented in the Figure 3.4.



```
<bb 2>:
n = 1024;
.omp_data_o.1.n = n;
__builtin_GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, 0);
main._omp_fn.0 (&.omp_data_o.1);
__builtin_GOMP_parallel_end ();
n = .omp_data_o.1.n;
return;
```

```
<bb 3>:
D.1822 = .omp_data_i->n;
D.1823 = (long int) D.1822;
D.1826 = __builtin_GOMP_loop_dynamic_start (0, D.1823, 1, 1, &.istart0.2, &.iend0.3);
if (D.1826 != 0)
  goto <bb 8>;
else
  goto <bb 5>;

<bb 8>:
.istart0.4 = .istart0.2;
i = (int) .istart0.4;
.iend0.5 = .iend0.3;
D.1830 = (int) .iend0.5;

<bb 4>:
i = i + 1;
if (i < D.1830)
  goto <bb 4>;
else
  goto <bb 9>;

<bb 9>:
D.1831 = __builtin_GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);
if (D.1831 != 0)
  goto <bb 8>;
else
  goto <bb 5>;

<bb 5>:
__builtin_GOMP_loop_end_nowait ();

<bb 6>:
return;
```

Figure 3.4: *Visualization of generated code format that uses dynamic scheduling (second format)*

The study of the expanded code of directives showed that there are at least two loop formats, which are generated according of the use of variables definitions. Variables that represent the loop iterations upper bound or the chunk size influence the code generation. If numerical values or constants are used, the GCC + libgomp generate one format and if values are variables or expressions that require evaluation for the discover of final value, the second format is generated.

Another interesting point to check is how execution of adjacent loops inside the same parallel region is made. The Code 3.18 has two loops with different schedules and upper bounds and *chunk_size* definitions, they are sharing the same parallel region.

```
num_t = 8;
#pragma omp parallel num_threads(num_t)
{
  #pragma omp for schedule(runtime)
  for (i = 0; i < 1024; i++){
    // body_1;
  }

  #pragma omp for schedule(dynamic, 32)
  for (j = 0; j < n; j++){
    // body_2;
  }
}
```

Código 3.18: *Parallel region code with two loops inside, loops with different configurations*

In the intermediate representation generated by GCC that is shown in the Code A.1, we can see that the generated code follows the formats to parallel region, in which an *outlined function* is created to handle the code inside the demarcated parallel region. The Figure 3.5 shows the idea, in the end of the first loop the threads are find an implicit barrier in the call `GOMP_loop_end()`.

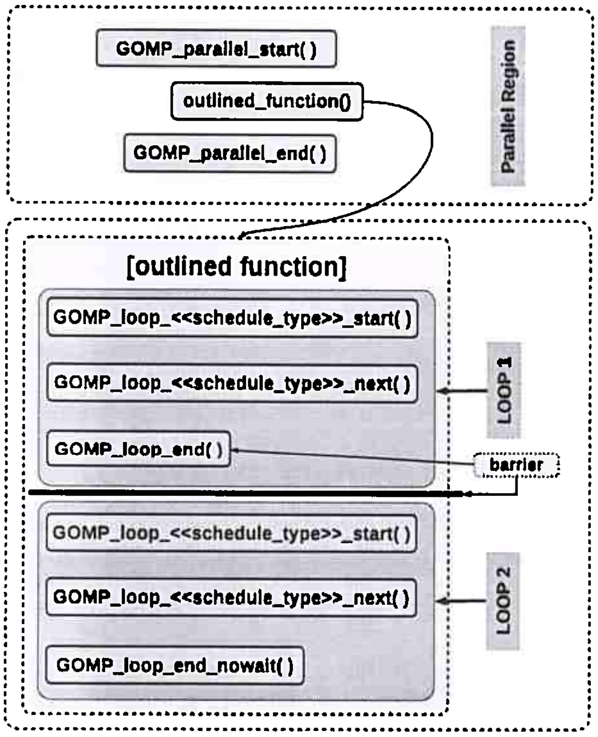
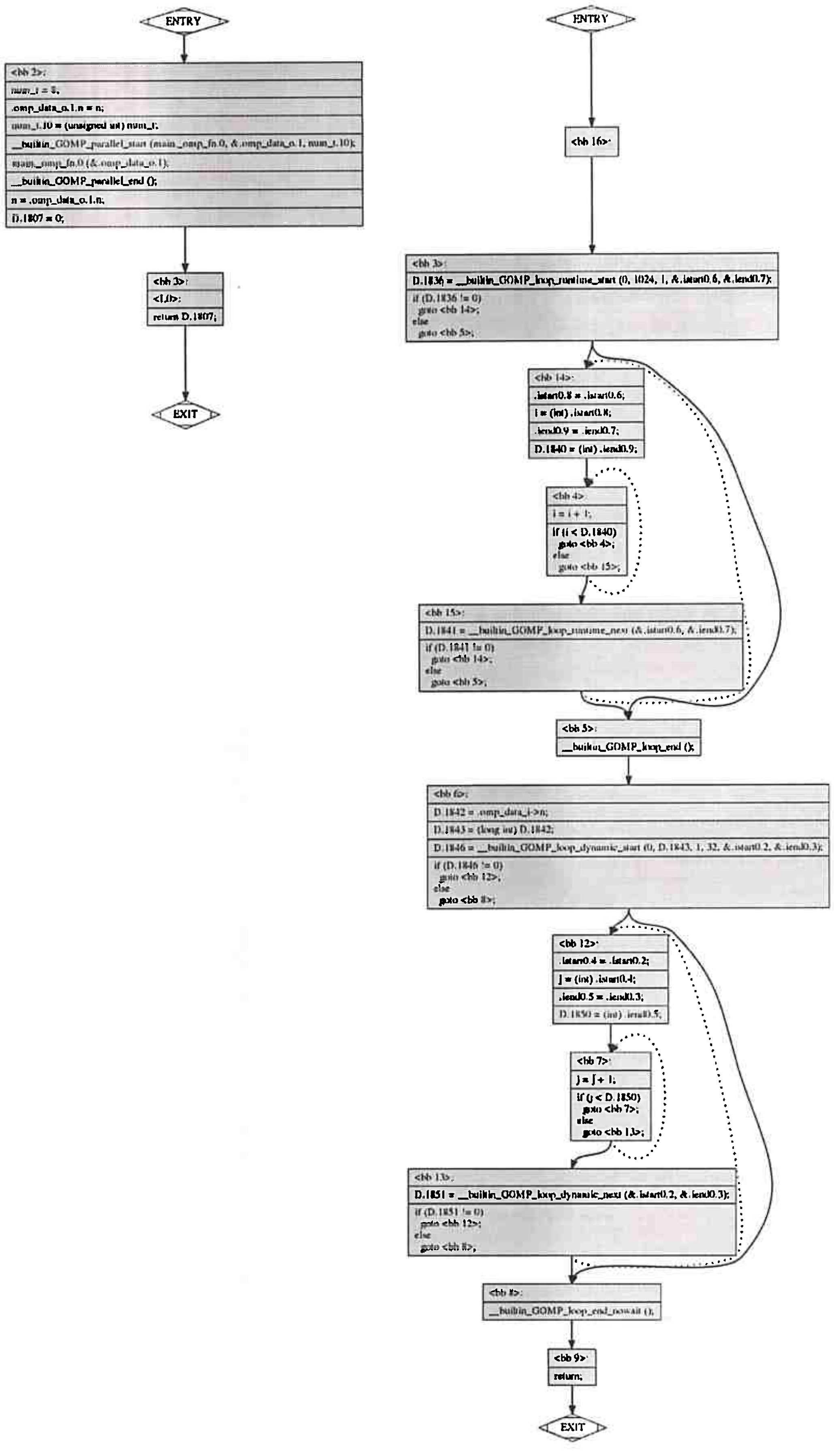


Figure 3.5: *Two loops inside the same parallel region*

The difference in comparing with the original format is in the finalization of the first loop. Instead of the first loop being finalized with the `GOMP_loop_end_nowait()` he is finished with the call to the function `GOMP_loop_end()`, which has an implicit barrier. All threads will wait the end of the execution of the first loop and then start the execution of the second loop. The graphical representation of the code is showed in th Figure 3.6.



```
<bb 2>:
num_t = 8;
.omp_data_o.1.n = n;
num_t.10 = (unsigned int) num_t;
__builtin_GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, num_t.10);
main._omp_fn.0 (&.omp_data_o.1);
__builtin_GOMP_parallel_end ();
n = .omp_data_o.1.n;
D.1807 = 0;

<bb 3>:
<L0>:
return D.1807;
```

```
<bb 16>:

<bb 3>:
D.1836 = __builtin_GOMP_loop_runtime_start (0, 1024, 1, &.istart0.6, &.iend0.7);
if (D.1836 != 0)
  goto <bb 14>;
else
  goto <bb 5>;

<bb 14>:
.istart0.8 = .istart0.6;
i = (int) .istart0.8;
.iend0.9 = .iend0.7;
D.1840 = (int) .iend0.9;

<bb 4>:
i = i + 1;
if (i < D.1840)
  goto <bb 4>;
else
  goto <bb 15>;

<bb 15>:
D.1841 = __builtin_GOMP_loop_runtime_next (&.istart0.6, &.iend0.7);
if (D.1841 != 0)
  goto <bb 14>;
else
  goto <bb 5>;

<bb 5>:
__builtin_GOMP_loop_end ();

<bb 6>:
D.1842 = .omp_data_i->n;
D.1843 = (long int) D.1842;
D.1846 = __builtin_GOMP_loop_dynamic_start (0, D.1843, 1, 32, &.istart0.2, &.iend0.3);
if (D.1846 != 0)
  goto <bb 12>;
else
  goto <bb 8>;

<bb 12>:
.istart0.4 = .istart0.2;
j = (int) .istart0.4;
.iend0.5 = .iend0.3;
D.1850 = (int) .iend0.5;

<bb 7>:
j = j + 1;
if (j < D.1850)
  goto <bb 7>;
else
  goto <bb 13>;

<bb 13>:
D.1851 = __builtin_GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);
if (D.1851 != 0)
  goto <bb 12>;
else
  goto <bb 8>;

<bb 8>:
__builtin_GOMP_loop_end_nowait ();

<bb 9>:
return;
```

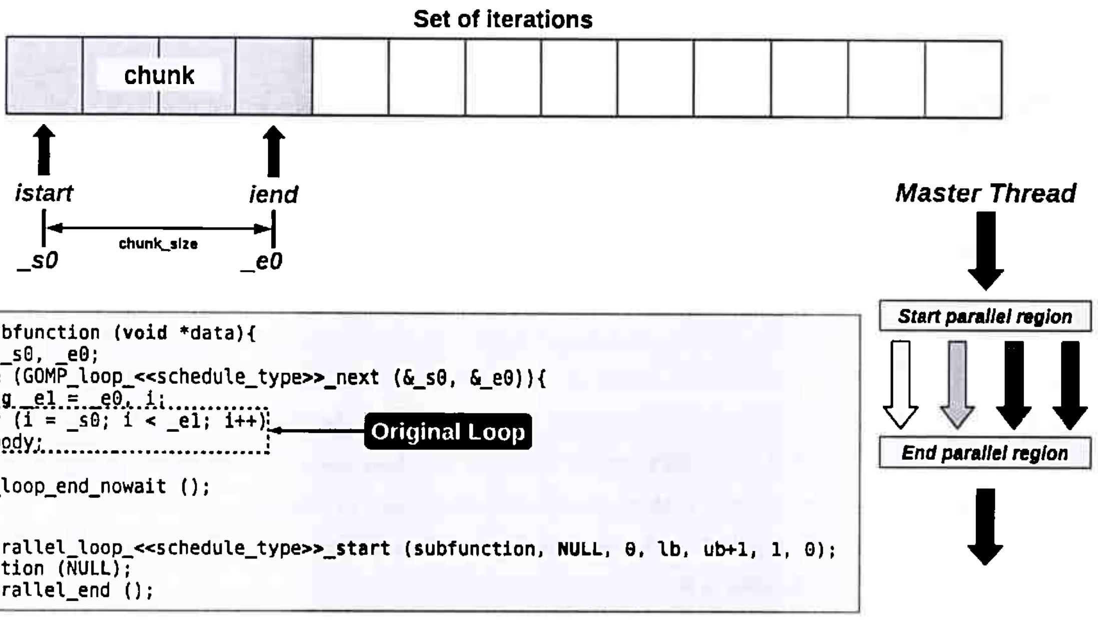
Figure 3.6: *Graphical representation of two loops inside the same parallel region*

The handling function that was created with the parallel region code contains the code of two adjacent loops. As can be seen in the code, There were not created two functions, one for each loop. The function handles both loops inside the same parallel region.

3.2 Interceptable OpenMP Code Format

The interceptable structures in our library are parallel regions and loops. The loops need to have the schedule type runtime, dynamic or guided. The format for static scheduling only divide the iterations by number of threads. It do not have an intermediate function in the format that allows to control how much of code will execute to get measures.

The function `GOMP_loop_«schedule_type»_next(...)` is the most important in the interceptable formats, because this function get the lower and upper index that delimit the sub-set of iterations – a *chunk*. In our interception mechanism when the certain percentage of code is reached, the interception of this function allows that the execution to be stopped. And if the offloading decision is positive, it allows to set that other threads have no more work to do. The Figure 3.7 shows the schema of threads getting chunks and executing these sub-set of iterations.



```
void subfunction (void *data){
  long _s0, _e0;
  while (GOMP_loop_<<schedule_type>>_next (&_s0, &_e0)){
    long _e1 = _e0, i;
    for (i = _s0; i < _e1; i++)
      body;
  }
  GOMP_loop_end_nowait ();
}

GOMP_parallel_loop_<<schedule_type>>_start (subfunction, NULL, 0, lb, ub+1, 1, 0);
subfunction (NULL);
GOMP_parallel_end ();
```

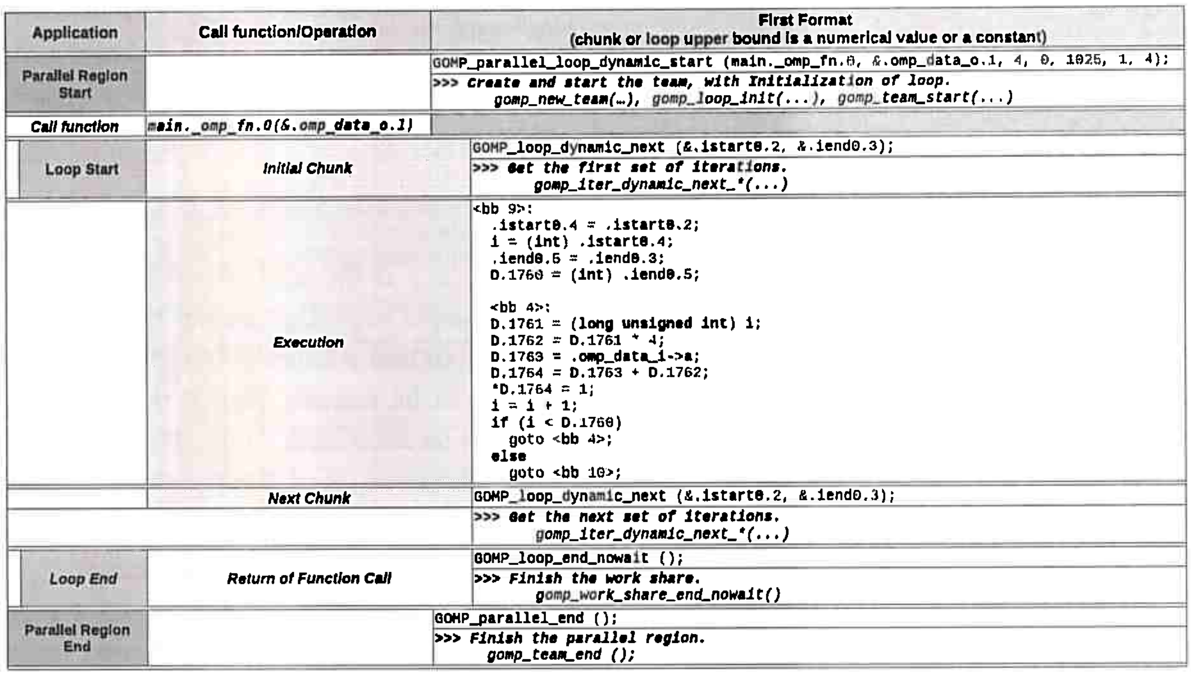
Figure 3.7: *Loop iterations execution by threads*

The threads in the team execute the same outlined function. During the execution each thread request to runtime more iterations (chunks) to execute.

When we are generating the GCC code using the `libgomp` we can verify that the code format for loops are affected by the manner how some variables are defined in the compilation directives and loop limits. Although, they are equivalent codes, they have different structures and they make calls to different functions.

The *chunk_size* value can be defined putting a numerical value directly in the code. A defined constant can be used too and during the pre-processing it will have the same effect. An expression that results in a value after the evaluation or a single variable can be used in this definition. The same rules are applied to upper bound loops definitions.

The generated code can have two formats according to the definition, the Figure 3.8 shows the first and the Figure 3.9 the second. In these figures the *dynamic* can be replaced by *runtime* or *guided*, because they have the same format.



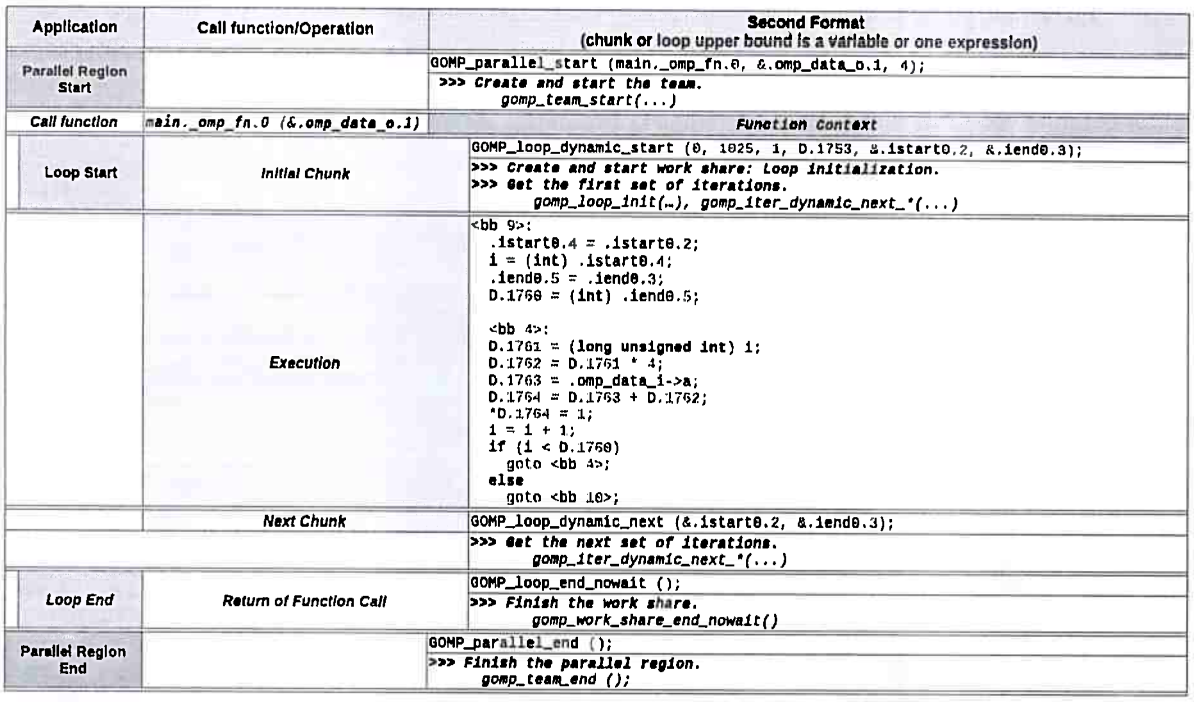
Application	Call function/Operation	First Format (chunk or loop upper bound is a numerical value or a constant)
Parallel Region Start		`GOMP_parallel_loop_dynamic_start (main._omp_fn.0, &.omp_data_o.1, 4, 0, 1025, 1, 4);`
		`>>> Create and start the team, with Initialization of loop.`
		`gomp_new_team(…), gomp_loop_init(...), gomp_team_start(...)`
Call function	`main._omp_fn.0(&.omp_data_o.1)`	
Loop Start	Initial Chunk	`GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);`
		`>>> Get the first set of iterations.`
		`gomp_iter_dynamic_next_*(...)`
	Execution	`<bb 9>:`
		`.istart0.4 = .istart0.2;`
		`i = (int) .istart0.4;`
		`.iend0.5 = .iend0.3;`
		`D.1760 = (int) .iend0.5;`
		`<bb 4>:`
		`D.1761 = (long unsigned int) i;`
		`D.1762 = D.1761 * 4;`
		`D.1763 = .omp_data_i->a;`
		`D.1764 = D.1763 + D.1762;`
		`*D.1764 = 1;`
		`i = i + 1;`
		`if (i < D.1760)`
		`goto <bb 4>;`
		`else`
		`goto <bb 10>;`
	Next Chunk	`GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);`
		`>>> Get the next set of iterations.`
		`gomp_iter_dynamic_next_*(...)`
Loop End	Return of Function Call	`GOMP_loop_end_nowait ();`
		`>>> Finish the work share.`
		`gomp_work_share_end_nowait()`
Parallel Region End		`GOMP_parallel_end ();`
		`>>> Finish the parallel region.`
		`gomp_team_end ();`

Figure 3.8: *First format with dynamic scheduling*

In the first format the call to function GOMP_parallel_loop_«schedule_type»_start() starts the parallel region. It creates the team of threads and initialize the loop execution controls. Inside the outlined function GOMP_loop_«schedule_type»_next(...) is used by threads to get the first chunk. The thread execute this first chunk and after retrieves in loop the next chunk to execute until have no more work to do. When the thread finishes the loop calling *GOMP_loop_end_nowait()* the loop work share is finished too. The parallel region is ended in the call GOMP_parallel_end() that deallocates the team of threads.

The same semantic is applied in the second format, but different functions are used. The second format is showed in the Figure 3.9.

In the second format the parallel region is started with the call GOMP_parallel_start(...) in this call only the team of threads is created. The loop work share initialization is made inside the outlined function and the call to GOMP_loop_«schedule_type»_start(...) is used to retrieves the first chunk. The threads that can get the first chunk execute it and use the GOMP_loop_«schedule_type»_next(...) to get the next until the finish the loop execution calling the GOMP_loop_end_nowait(). The parallel region is finished using the same function GOMP_parallel_end() that deallocates the team of threads.



Application	Call function/Operation	Second Format (chunk or loop upper bound is a variable or one expression)
Parallel Region Start		`GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, 4);`
		`>>> Create and start the team.`
		`gomp_team_start(...)`
Call function	`main._omp_fn.0 (&.omp_data_o.1)`	Function Context
Loop Start	Initial Chunk	`GOMP_loop_dynamic_start (0, 1025, 1, D.1753, &.istart0.2, &.iend0.3);`
		`>>> Create and start work share: Loop initialization.`
		`>>> Get the first set of iterations.`
		`gomp_loop_init(..), gomp_iter_dynamic_next_*(...)`
	Execution	`<bb 9>:`
		`.istart0.4 = .istart0.2;`
		`i = (int) .istart0.4;`
		`.iend0.5 = .iend0.3;`
		`D.1760 = (int) .iend0.5;`
		`<bb 4>:`
		`D.1761 = (long unsigned int) i;`
		`D.1762 = D.1761 * 4;`
		`D.1763 = .omp_data_i->a;`
		`D.1764 = D.1763 + D.1762;`
		`*D.1764 = 1;`
		`i = i + 1;`
		`if (i < D.1760)`
		`goto <bb 4>;`
		`else`
		`goto <bb 10>;`
	Next Chunk	`GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);`
		`>>> Get the next set of iterations.`
		`gomp_iter_dynamic_next_*(...)`
Loop End	Return of Function Call	`GOMP_loop_end_nowait ();`
		`>>> Finish the work share.`
		`gomp_work_share_end_nowait()`
Parallel Region End		`GOMP_parallel_end ();`
		`>>> Finish the parallel region.`
		`gomp_team_end ();`

Figure 3.9: *Second format with dynamic scheduling*

The Figure 3.10 summarize the two formats that were identified and that are interceptable by our library.

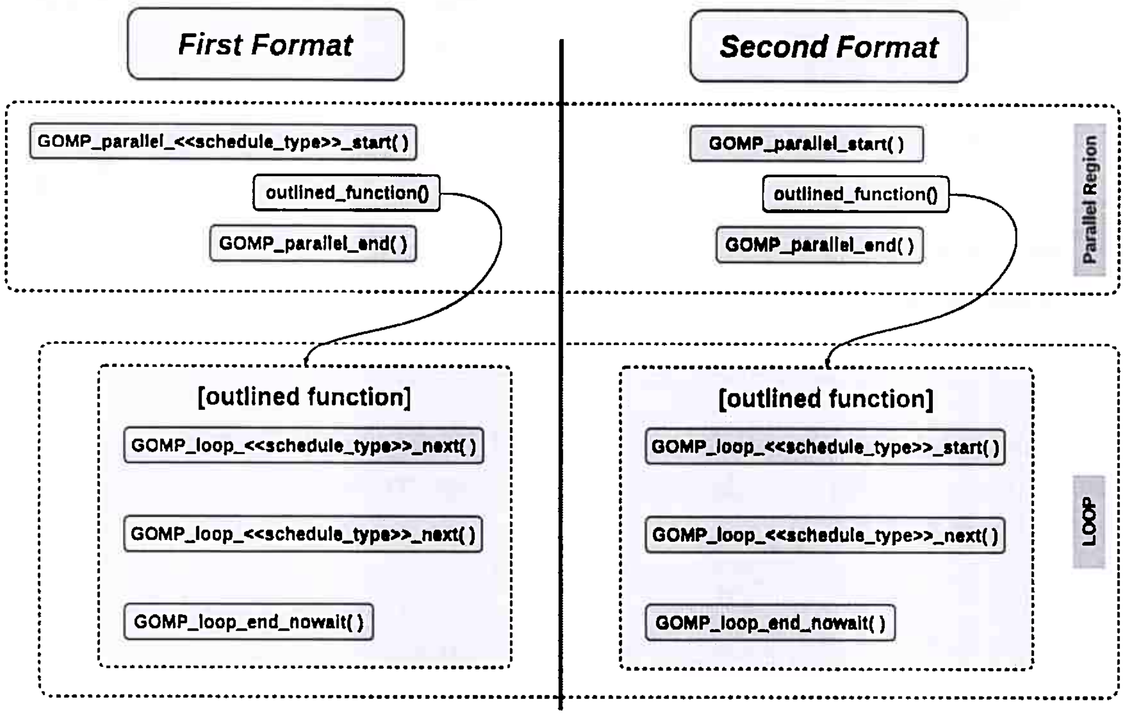


Figure 3.10: *Structure of loop formats*

3.3 Interception and Offloading Mechanisms

For offloading purposes, the runtime needed to control the execution of threads, and decide when to make the code offloading and what is the better device to execute the code. Initially, the execution starts with OpenMP code, which will have some of their calls intercepted by the execution control of threads in the CPU.

The Figure 3.11 shows the initial version that was using one collector thread to get measures of performance counters and the others were blocked. This approach proved to be inefficient, because just only one thread was getting the measures, so for a period of time the application run in single thread execution mode. The poor performance was evident in contrast with the normal OpenMP version that was executing with all threads.

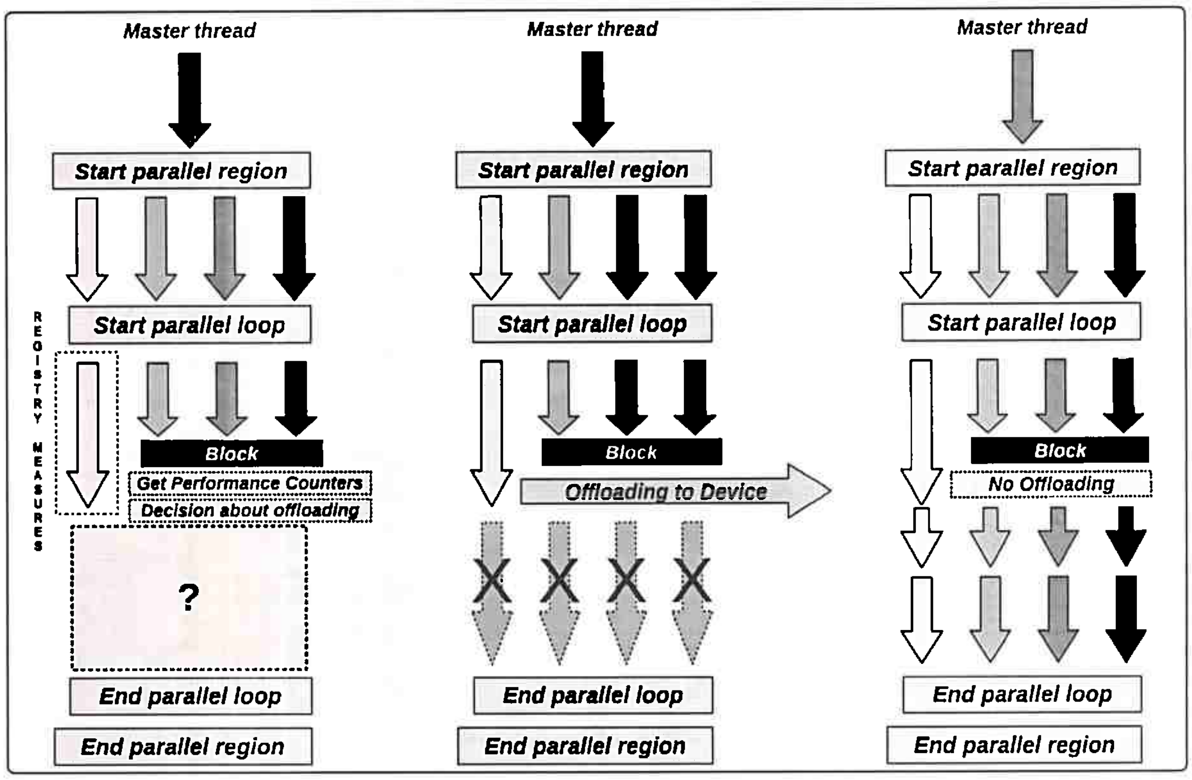


Figure 3.11: *Threads executing loop iterations (Version 1)*

The second and current approach uses one thread in the process to collect measures of performance counters and the other threads can execute freely. The threads execution mechanism is showed in Figure 3.12.

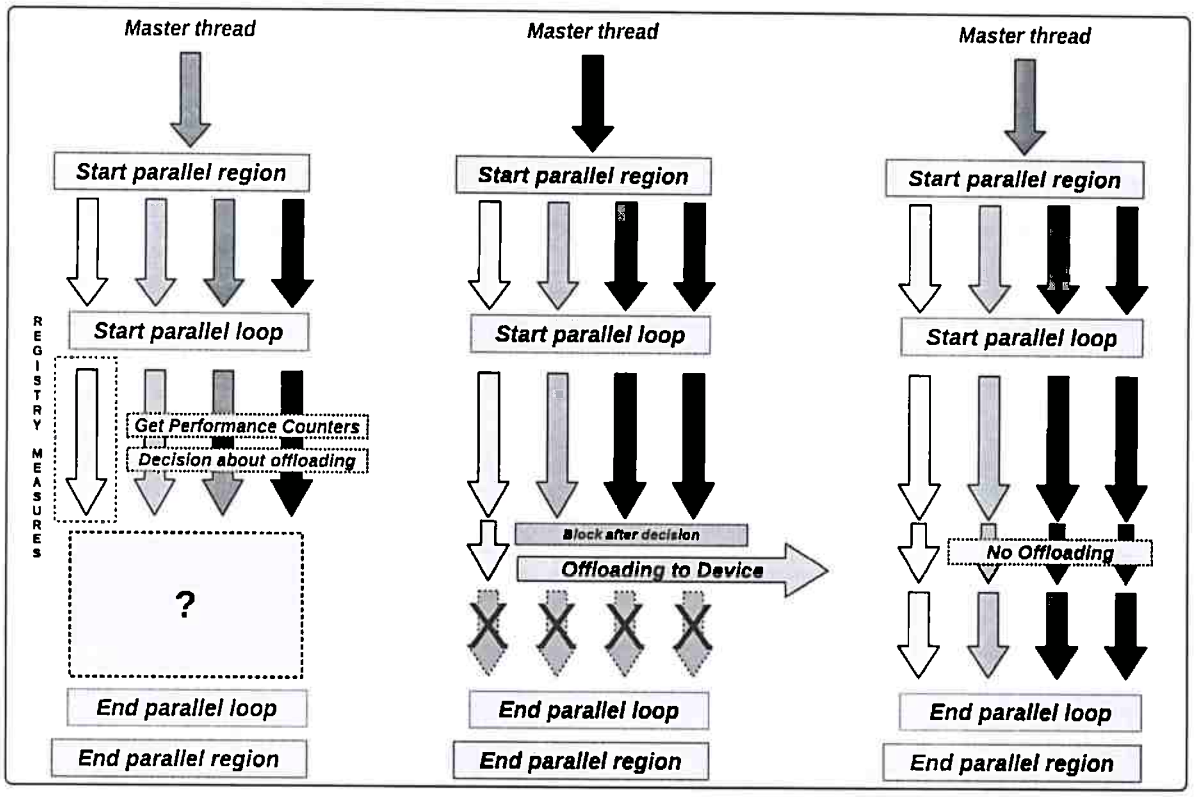


Figure 3.12: *Threads executing loop iterations (Current version)*

For each parallel region a set of threads is created by OpenMP runtime. When the threads start the loop execution. One thread is registered to collect measures of performance counters and take the decision about the code offloading.

If the runtime decision is positive for offloading, the other threads are blocked and the appropriated version *[loop index, device]* function is launched using the alternative functions table. The work shared is defined as completed when the offloading is made. The get a next chunk function (`GOMP_loop_*_next(...)`) will return *false* for blocked threads, and all the threads will be terminated by the OpenMP runtime. Otherwise, the threads continue the regular execution on OpenMP execution model.

The decision between makes the code offloading or continuing the execution in multi-core CPU is returned by the model implemented in `libroofline` using the *operational intensity*.

3.4 The runtime input code format

The input code used by our runtime is a OpenMP code with a alternative functions table. This table is shared with the runtime libraries for the interception control can call the appropriate alternate function based on the offloading decision. The Figure 3.13 shows the use of alternative functions table.

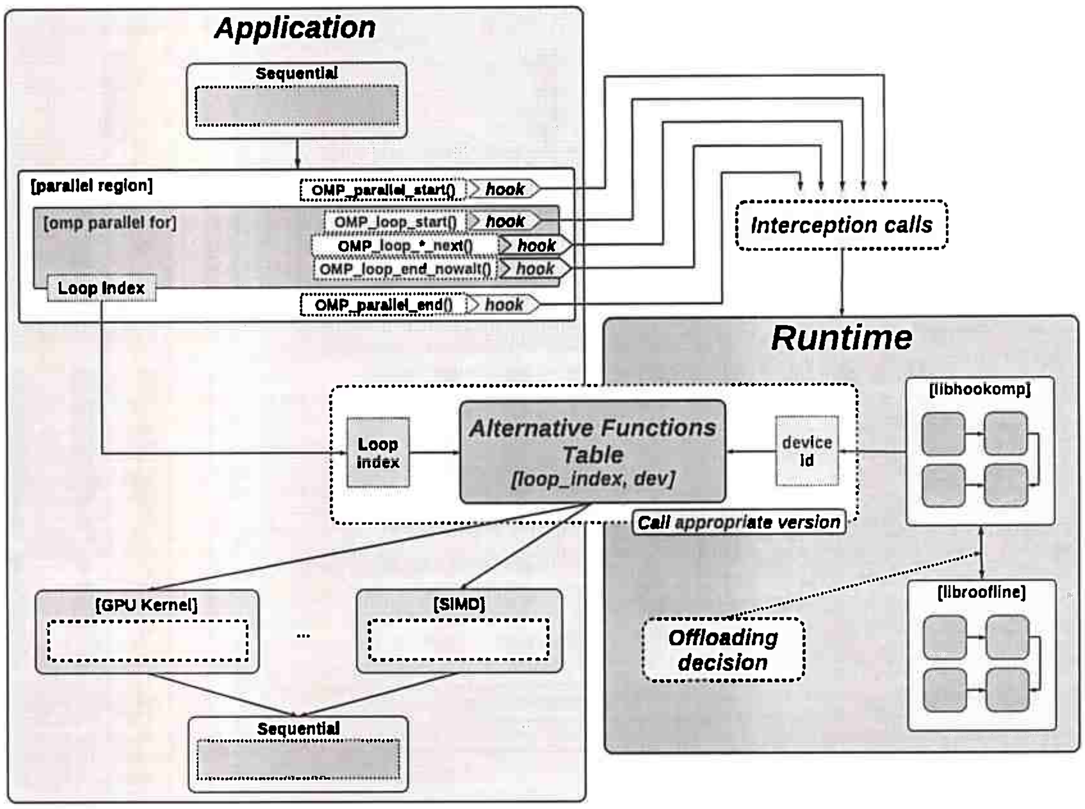


Figure 3.13: *Use of the alternative functions table*

The `librofline` returns the device *id* together the offloading decision. If the runtime decision is positive for offloading the `libhookomp` switch the execution to a alternative function indexed by *[loop_ index, dev_ id]*.

Following the idea of code modernization using the vendor libraries. Another possibility is the use of optimized functions available in software development kits like a cuBLAS (NVIDIA, 2014d, 2015d) to create the alternative functions table.

3.5 Application and Runtime Libraries Interaction

For *hooking* OpenMP calls we developed a library *(libhookomp)* using the `LD_PRELOAD` technique (Linux Man-pages, 2016). This library can be pre-loaded and linked to the code before the OpenMP runtime libraries replacing the code of original calls for proxy functions with the same names. It supports the interception of some kinds of calls that application makes to the OpenMP runtime.

This mechanism is used to detect at runtime *parallel regions* and *parallel loops*. Then our runtime functions can be execute an additional code before or after the original calls to OpenMP runtime, according the context execution. The Application and runtime libraries interaction is showed in the Figure 3.14.

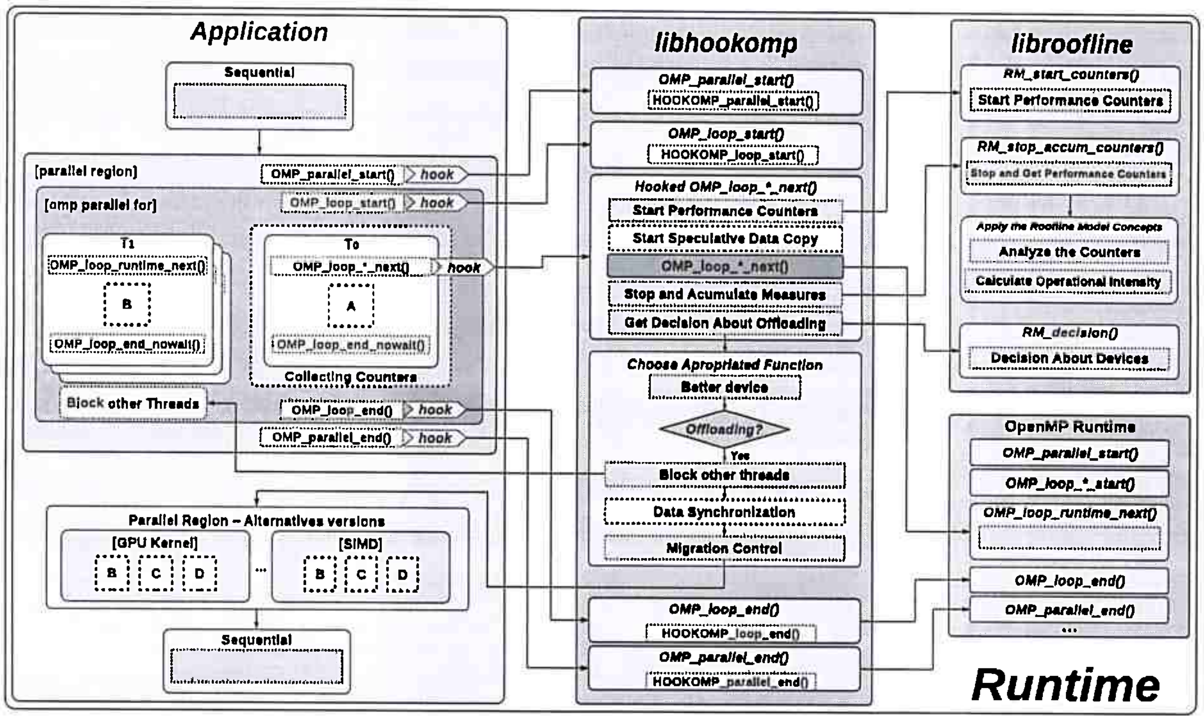
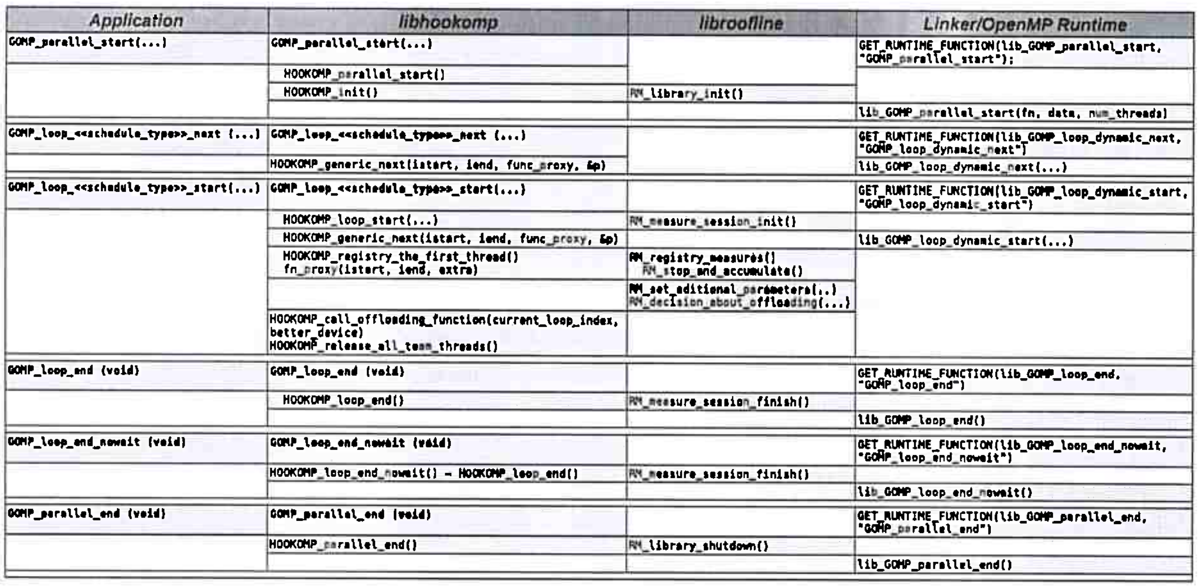


Figure 3.14: *Application and Runtime Libraries Interaction*

The interaction between applications and libraries is detailed in Figure 3.15. We mapped the function calls through the libraries, starting in application code until the OpenMP runtime.



Application	libhookomp	libroofline	Linker/OpenMP Runtime
GOMP_parallel_start(...)	GOMP_parallel_start(...)		GET_RUNTIME_FUNCTION(lib_GOMP_parallel_start, "GOMP_parallel_start");
	HOOKOMP_parallel_start()		
	HOOKOMP_init()	RM_library_init()	
			lib_GOMP_parallel_start(fn, data, num_threads)
GOMP_loop_<<schedule_type>>_next (...)	GOMP_loop_<<schedule_type>>_next (...)		GET_RUNTIME_FUNCTION(lib_GOMP_loop_dynamic_next, "GOMP_loop_dynamic_next")
	HOOKOMP_generic_next(istart, iend, func_proxy, &p)		lib_GOMP_loop_dynamic_next(...)
GOMP_loop_<<schedule_type>>_start(...)	GOMP_loop_<<schedule_type>>_start(...)		GET_RUNTIME_FUNCTION(lib_GOMP_loop_dynamic_start, "GOMP_loop_dynamic_start")
	HOOKOMP_loop_start(...)	RM_measure_session_init()	
	HOOKOMP_generic_next(istart, iend, func_proxy, &p)		lib_GOMP_loop_dynamic_start(...)

	HOOKOMP_registry_the_first_thread()
	fn_proxy(istart, iend, extra)
		RM_registry_measures()
		RM_stop_and_accumulate()	

		RM_set_aditional_parameters(..)
		RM_decision_about_offloading(...)	

	HOOKOMP_call_offloading_function(current_loop_index, better_device)
	HOOKOMP_release_all_team_threads()		

GOMP_loop_end (void)	GOMP_loop_end (void)		GET_RUNTIME_FUNCTION(lib_GOMP_loop_end, "GOMP_loop_end")
	HOOKOMP_loop_end()	RM_measure_session_finish()	
			lib_GOMP_loop_end()
GOMP_loop_end_nowait (void)	GOMP_loop_end_nowait (void)		GET_RUNTIME_FUNCTION(lib_GOMP_loop_end_nowait, "GOMP_loop_end_nowait")
	HOOKOMP_loop_end_nowait() - HOOKOMP_loop_end()	RM_measure_session_finish()	
			lib_GOMP_loop_end_nowait()
GOMP_parallel_end (void)	GOMP_parallel_end (void)		GET_RUNTIME_FUNCTION(lib_GOMP_parallel_end, "GOMP_parallel_end")
	HOOKOMP_parallel_end()	RM_library_shutdown()	
			lib_GOMP_parallel_end()

Figure 3.15: *Mapping functions libraries called by applications*

In this mapping we can see the sequencing of the calls among contexts (application code until OpenMP runtime). That shows important points in hooking mechanism, for example, what is made at the beginning of a *parallel region* or when a *loop* is started.

In addition to proxy functions definition contained in libhookomp we defined initialization and shutdown functions for the libraries. When the parallel region is intercepted in the hook library will be started an internal parallel region control using HOOKOMP_parallel_start() and HOOKOMP_init(). In the same way, this control will be finalized when the parallel region in application code reach the end where the interception code will call internally the `HOOKOMP_parallel_end()`. The controls in `libroofline` will be initialized and finished together this calls using the functions `RM_library_init()` and `RM_library_shutdown()` where the `PAPI` libraries and its controls are initialized.

The measures are made by loop and the loop start initialize the loop control in `libhookomp` that initialize the a measure session in `libroofline` library. The interception of function that gets next chunks is made using a generic function that treats the two loops formats. The `HOOKOMP_generic_next(...)` is prepared to call the next chunk function by a pointer, because in the first format it is the `GOMP_loop_«schedule_type»_next(...)` and in the second format the next chunk is retrived together with the loop initialization by function `GOMP_loop_«schedule_type»_start(...)`.

The `HOOKOMP_generic_next(...)` is very important, because is in this function that the threads execution is controlled. The measures are registered and the offloading is made calling the alternative function for the current loop.

3.6 Creating Hooks for OpenMP Functions

To create *hooks* for `libgomp` functions it is need to create a library with proxy functions that have the same signature of the available functions via `libgomp` ABI. The *hooking* library is loaded before of original `libgomp` during the dynamic linking. So calls to functions of the `OpenMP` runtime are linked to symbols of hooking library. The proxy functions will request to *linker* using `dlsym` the next symbol that represent the original function pointer. This is the interception mechanism used by our runtime is showed by a sample in Code 3.21 that shows the interception for `GOMP_parallel_start()` function.

```
void GOMP_parallel_start (void (*fn) (void *), void *data, unsigned
    num_threads){
  PRINT_FUNC_NAME;

  /* Retrieve the OpenMP runtime function. */
  typedef void (*func_t) (void (*fn) (void *), void *, unsigned);
  func_t lib_GOMP_parallel_start = (func_t) dlsym(RTLD_NEXT, "
      GOMP_parallel_start");

  lib_GOMP_parallel_start(fn, data, num_threads);
}
```

Código 3.19: *Exemplo de criação de uma hook para a função* GOMP_parallel_start

The idea is retrieve the pointer for call the original function inside the proxy function after or before the execution of some specific code.

To facilitate the writing of code a macro was defined to retrieve the original functions pointers. The macro code is showed in the Code 3.20.

```
#define GET_RUNTIME_FUNCTION(hook_func_pointer,func_name) \
  do {  \
    if (hook_func_pointer) break; \
    void *__handle = RTLD_NEXT; \
    hook_func_pointer = (typeof(hook_func_pointer)) (uintptr_t) dlsym(
        __handle, func_name); \
    PRINT_ERROR();  \
} while(0)
```

Código 3.20: *Definição de macro para recuperar o ponteiro para a função original*

The same proxy function to `GOMP_parallel_start()` can be rewritten using the macro. The Code 3.21 presents the rewritten function.

```
void GOMP_parallel_start (void (*fn) (void *), void *data, unsigned
    num_threads){
  PRINT_FUNC_NAME;
  /* Retrieve the OpenMP runtime function. */
  GET_RUNTIME_FUNCTION(lib_GOMP_parallel_start, "GOMP_parallel_start");

  HOOKOMP_parallel_start();
  HOOKOMP_init();

  lib_GOMP_parallel_start(fn, data, num_threads);
}
```

Código 3.21: *Proxy function to parallel region start*

Similarly, a proxy function to `GOMP_parallel_end()` is defined in the Code 3.22.

```
void GOMP_parallel_end (void){
  PRINT_FUNC_NAME;
  /* Retrieve the OpenMP runtime function. */
  GET_RUNTIME_FUNCTION(lib_GOMP_parallel_end, "GOMP_parallel_end");

  /* In cases of benchmark have two loops inside the same parallel region.
      The second was ignored, because the control had no reinitilized. */
  if(is_hookomp_initialized){
    HOOKOMP_parallel_end();
  }

  lib_GOMP_parallel_end();
}
```

Código 3.22: *Fucntion to intercepts the GOMP_parallel_end*

The function that starts the parallel region in the loop *first format* is showed in the Code 3.23. It should be noted that both parallel start functions do the same internal controls initializations in `HOOKOMP`.

```
void GOMP_parallel_loop_dynamic_start (void (*fn) (void *), void *data,
unsigned num_threads, long start , long end,
long incr , long chunk_size){
  PRINT_FUNC_NAME;
  /* Retrieve the OpenMP runtime function. */
  GET_RUNTIME_FUNCTION(lib_GOMP_parallel_loop_dynamic_start, "
      GOMP_parallel_loop_dynamic_start");

  HOOKOMP_parallel_start();

  /* Initializations. */
  HOOKOMP_init();
  HOOKOMP_loop_start(start , end, num_threads, chunk_size);

  lib_GOMP_parallel_loop_dynamic_start(fn, data, num_threads, start , end,
      incr , chunk_size);
}
```

Código 3.23: *Fucntion to intercepts the GOMP_parallel_loop_dynamic_start*

In the Code 3.24 is showed the interception function for `GOMP_loop_end()`. The proxy function finalize the loop controls in HOOKOMP library.

```
void GOMP_loop_end (void){
  PRINT_FUNC_NAME;
  /* Retrieve the OpenMP runtime function. */
  GET_RUNTIME_FUNCTION(lib_GOMP_loop_end, "GOMP_loop_end");

  HOOKOMP_loop_end();
  lib_GOMP_loop_end();
  TRACE("End of loop: %d\n", current_loop_index);
}
```

Código 3.24: *Function to intercepts the GOMP_loop_end*

The function to close the loop execution in which the threads continue the execution without wait by others threads is presented in the Code 3.25.

```
void GOMP_loop_end_nowait (void){
  PRINT_FUNC_NAME;
  /* Retrieve the OpenMP runtime function. */
  GET_RUNTIME_FUNCTION(lib_GOMP_loop_end_nowait, "GOMP_loop_end_nowait");

  HOOKOMP_loop_end_nowait();
  lib_GOMP_loop_end_nowait();

  TRACE("End of loop nowait: %d\n", current_loop_index);
}
```

Código 3.25: *Function to intercepts the GOMP_loop_end_nowait*

The proxy functions as `GOMP_loop_dynamic_start(...)` because the different number of parameters are using the generic functions to the both formats can be supported. The Code 3.26 shows this use with the encapsulation of the parameters.

```
bool GOMP_loop_dynamic_start (long start , long end, long incr , long
    chunk_size ,
long *istart , long *iend){
  PRINT_FUNC_NAME;
  /* Retrieve the OpenMP runtime function. */
  GET_RUNTIME_FUNCTION(lib_GOMP_loop_dynamic_start, "
      GOMP_loop_dynamic_start");

  TRACE("Starting with %d threads.\n", omp_get_num_threads());
  /* Initializations. */
  HOOKOMP_loop_start(start , end, omp_get_num_threads() , chunk_size);

  chunk_next_fn func_proxy;
  Params p;
  p._0 = start;
  p._1 = end;
  p._2 = incr;
  p._3 = chunk_size;

  p.func_start_next = lib_GOMP_loop_dynamic_start;
  p.func_type = FUN_START_NEXT;
  func_proxy = &HOOKOMP_proxy_function_start_next;
  bool result = HOOKOMP_generic_next(istart , iend, func_proxy, &p);

  return result;
}
```

Código 3.26: *Function to intercepts the GOMP_loop_dynamic_start*

Considering the loop formats that were identified some generic functions were created to provide support the both, because depending of format the the loop work share initialization and the first chunk can be obtained in different points of execution or parallel region initialization or inside the loop format function. The Code 3.27 shows a function that is used by function calls for initialize the work share and retrieve the first chunk.

```
/* Proxy function to *_start */
bool HOOKOMP_proxy_function_start_next (long* istart , long* iend, void*
    extra) {
  PRINT_FUNC_NAME;
  Params *params = (Params*) extra;
  bool result = params->func_start_next(params->_0, params->_1, params->_2
      , params->_3, istart , iend);

  return result;
}
```

Código 3.27: *Proxy function to functions that execute loop start and get next chunk*

For the next chunk functions another proxy function was created, because the different number of parametes. This function is showed in the Code 3.28.

```
bool GOMP_loop_dynamic_next (long *istart , long *iend){
  PRINT_FUNC_NAME;

  /* Retrieve the OpenMP runtime function. */
  GET_RUNTIME_FUNCTION(lib_GOMP_loop_dynamic_next, "GOMP_loop_dynamic_next
      ");

  chunk_next_fn func_proxy;
  Params p;
  p._0 = 0;
  p._1 = 0;
  p._2 = 0;
  p._3 = 0;
  p.func_next = lib_GOMP_loop_dynamic_next;
  p.func_type = FUN_NEXT;
  func_proxy = &HOOKOMP_proxy_function_next;
  bool result = HOOKOMP_generic_next(istart , iend , func_proxy , &p);

  return result;
}
```

Código 3.28: *Proxy function to functions that get next chunk*

The used functions have a different number os parameters, so it was need to create generic functions. The function that support the next chunk functions is showed in the Code 3.29.

```
/* Proxy function to *_next */
bool HOOKOMP_proxy_function_next (long* istart , long* iend , void* extra) {
  PRINT_FUNC_NAME;
  Params *params = (Params*) extra;
  bool result = params->func_next(istart , iend);

  return result;
}
```

Código 3.29: *Proxy function to functions that get next chunk*

In the table of alternative functions are stored a structure with pointers to function, list of parameters values and types. The runtime is using the library libffi to recreate the target function call using these elements. The structure and the function to recreate the call is showed in Code 3.30.

```
/* Struct to store pointer and arguments to alternative functions. */
typedef struct Func {
  void *f;
  int nargs;
  ffi_type** arg_types;
  void** arg_values;
  ffi_type* ret_type;
  void* ret_value;
```

```c
} Func;

/* Call the target function. */
bool HOOKOMP_call_function_ffi(Func* ff) {
  PRINT_FUNC_NAME;
  ffi_cif cif;
  int retval = 0;

  if ((retval = ffi_prep_cif(&cif, FFI_DEFAULT_ABI, ff->nargs, ff->
      ret_type, ff->arg_types)) != FFI_OK){
    TRACE("Error ffi_prep_cif.\n");
  }
  else{
    TRACE("Calling the target function.\n");
    ffi_call(&cif, FFI_FN(ff->f), ff->ret_value, ff->arg_values);
    TRACE("The target function was called.\n");
  }

  return (retval == FFI_OK);
}
```

Código 3.30: *Function to remake the target function call*

It is in the function listed in the Code 3.31 that the appropriate function is called according the *loop_index* and *device_id*. The element in the alternative functions table is passed to `HOOKOMP_call_function_ffi(...)` that will reconstruct the function call.

```c
/* Call the appropriated function. */
bool HOOKOMP_call_offloading_function(long int loop_index, long int
    device_id){
  PRINT_FUNC_NAME;
  bool retval = false;

  if(TablePointerFunctions == NULL){
    TRACE("TablePointerFunctions is not defined.\n");
    return retval;
  }
  if((TablePointerFunctions != NULL) && (TablePointerFunctions[loop_index
      ][device_id] != NULL) && ((TablePointerFunctions[loop_index][
      device_id])->f != NULL)){
    TRACE("Offloading function for loop index: %d, device id: %d.\n",
        loop_index, device_id);
    retval = HOOKOMP_call_function_ffi(TablePointerFunctions[loop_index][
        device_id]);
  }
  else{
    TRACE("Offloading function not defined in TablePointerFunctions.\n");
  }

  return retval;
}
```

Código 3.31: *Function to try call offloading function*

Knowing that such functions can vary depending the loop format, a generic function was created to handle the both formats. Practically, all the execution control is made in this function. As a percentage of the code must be executed to collect performance counters values to be made, this function is very important, because the collector thread retrieves chunks and computes how many iterations were executed. The generic function is showed in the Code A.2. The offloading decision is taken after to get measures of all event sets ou after reach the percentage of executed code.

3.7 Final Considerations

The developed runtime is able to intercept the applications call to `OpenMP` runtime, it can control the threads execution and make offloading according the operational intensity of code.

The current version of interception mechanism registry a thread member of `OpenMP` threads team to collect the values of performance counters during the application execution.

During the experiments this approach proved to be thread scheduling sensitive, because is highly dependent of the scheduling made by the system. For some combinations of chunk sizes and number of threads, is possible that the thread that is the collector do not finish the measures before the work is finished be other threads. Maybe another approach can be solve this problem using all the threads to measure the performance counters.

Chapter 4

Offloading Decision using Operational Intensity

In this chapter are presented the model and the library that implements the use of operational intensity to decide about the offloading – *librooftine*. This library implements the *Roofline Model* (Williams *et al.*, 2009) concepts and it is being used to decide about the code offloading.

Make a decision about the code offloading the library calculates the operational intensity, attainable performance and estimate the time of execution. As this work targets the offloading for GPUs, the time of data transfers to and from device have been considered in the execution time estimation, because devices like GPUs have private memory. In this case, it is needed to copy input . data to launch a kernel execution and copy output data for retrieve the results.

The memory traffic is measured in all memory levels. The library collects the values of Performance Counters at runtime. These measures are collected at runtime using the `PAPI` (Mucci *et al.* , 1999). We are using the `PAPI` to get performance counters because it works across systems as different processors and it has components that include the Linux `perf` and `cuda` that uses `CUPTI` (NVIDIA, 2013, 2014b).

4.1 Operational Intensity

A basic concept in the *Roofline Model* is the *operational intensity*. The *operational intensity* is the relation between *Work* in terms of floating point operations (W) and the *number of bytes* in DRAM traffic (Q) (Williams *et al.*, 2009) and it can be calculated using the Equation 4.1.

$$I = \frac{W}{Q} \tag{4.1}$$

The number of `READ` and `WRITE` accesses to the RAM Memory (MEM), Last Level Cache (LLC), Level 2 Cache (L2), Level 1 (L1) and the number of Floating Point operations (FP) are collected on a code percentage.

To address the need to make measures at runtime we adopted the `PAPI` (Mucci *et al.*, 1999). The `PAPI` is widely used and have support for several platforms and components, these features that motivated the decision to choose it.

The `PAPI` works with two categories of events, the *PRESET* and *NATIVE*, that allows you to get values of both event types. The number of floating point operations is calculated using others 3 native events that are applied in the Equation 4.2 available in `PAPI` as a *preset event*

PAPI_DP_OPS.

$$\begin{aligned} W = FP\_COMP\_OPS\_EXE : SSE\_SCALAR\_DOUBLE+ \\ 2 * (FP\_COMP\_OPS\_EXE : SSE\_FP\_PACKED\_DOUBLE)+ \\ 4 * (SIMD\_FP\_256 : PACKED\_DOUBLE) \end{aligned} \tag{4.2}$$

As the number of floating point instructions that were executed is available in a preset event, the PAPI_DP_OPS is derived of the 3 natives and it is not necessary to collect these events and calculate the total value. Just use the value provided by PAPI.

The counters with PACKED represent more than one floating point operation each, so it is needed multiply for a factor 2 or 4 depending of number operations executed by instructions in each category. These instructions are related with vectorization support (AVX), in which floating point operations can combine additions and multiplications. In this case, AVX addition instructions that work on four doubles are count as four operations (PAPI, 2015).

All levels in memory hierarchy are being considered, the Figure 4.1 shows the levels.

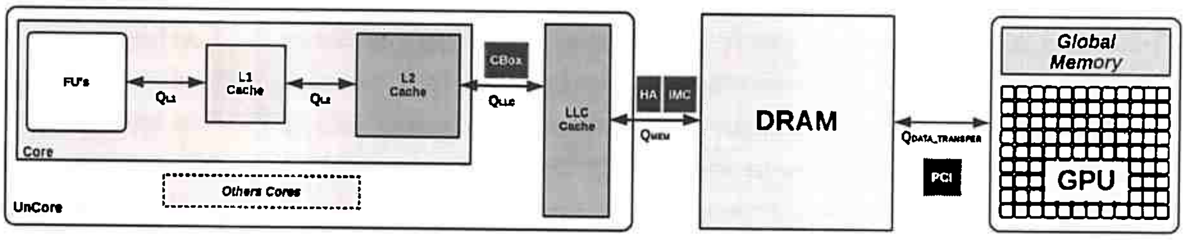


Figure 4.1: *Memory Hierarchy Levels*

Each Q in determined level can be divided in two classes to distinguish the kind or nature of accesses. Q_{write} represents write accesses and Q_{read} denotes the read accesses. The Equation 4.3 presents how to Q_{level} is calculated.

$$Q_{level} = Q_{read} + Q_{write} \tag{4.3}$$

The measured values for all memory levels accesses are used to calculate the total memory traffic in bytes (Q) using the Equation 4.4.

$$Q_{total} = Q_{L1} + Q_{L2} + Q_{LLC} + Q_{MEM} \tag{4.4}$$

The number of reads and writes accesses are accumulated and multiplied by 64 bytes. The value CACHE_LINE_SIZE is a parameter to the decision model and it is obtained from the processor manual.

Our model is considering the transfer of data to devices as another level in the memory hierarchy. The $Q_{DATA\_TRANSFER}$ is an additional parameters given by input code (Application). It is the amount of bytes of the input and output data structures that are needed to launch the kernel execution on device and get the execution results.

4.2 Events Sets and Hardware Restrictions

To cover measures in all the levels, we need to collect values for 5 events sets: *Main Memory, Last Level Cache (LLC)*, *Level 2 Cache*, *Level 1 Cache* and *Floating Point Instructions*. We are using a combination of *native* and *preset* PAPI events.

We are taking as a basis the event set used in the work of Ofenbeck *et al.* (2014), in which the authors apply the Roofline Model concepts on newer processors and they present a list of native events that were used with Intel PCM. We mapped these events to use them with PAPI. The defined events sets for the model are presented in Table 4.1.

Table 4.1: *Events Sets*

Set	Event	Native
MEM	UNC_H_IMC_READS	ivbep_unc_ha0::UNC_H_IMC_READS:cpu=0
	UNC_H_IMC_WRITES	ivbep_unc_ha0::UNC_H_IMC_WRITES:cpu=0
LLC	PAPI_TOT_CYC	UNHALTED_CORE_CYCLES
	PAPI_REF_CYC	UNHALTED_REFERENCE_CYCLES
	PAPI_L3_DCR	OFFCORE_REQUESTS:DEMAND_DATA_RD
	PAPI_L3_DCW	L2_RQSTS:RFO_MISS
L2	PAPI_TOT_CYC	UNHALTED_CORE_CYCLES
	PAPI_REF_CYC	UNHALTED_REFERENCE_CYCLES
	PAPI_L2_DCR	L2_RQSTS:ALL_DEMAND_DATA_RD
	PAPI_L2_DCW	L2_STORE_LOCK_RQSTS:ALL
L1	UNHALTED_CORE_CYCLES	UNHALTED_CORE_CYCLES
	UNHALTED_REFERENCE_CYCLES	UNHALTED_REFERENCE_CYCLES
	perf::PERF_COUNT_HW_CACHE_L1D:READ	perf::PERF_COUNT_HW_CACHE_L1D:READ
	perf::PERF_COUNT_HW_CACHE_L1D:WRITE	perf::PERF_COUNT_HW_CACHE_L1D:WRITE
FP	PAPI_TOT_CYC	UNHALTED_CORE_CYCLES
	PAPI_REF_CYC	UNHALTED_REFERENCE_CYCLES
	PAPI_DP_OPS	FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE + 2*(FP_COMP_OPS_EXE:SSE_FP_PACKED_DOUBLE) + 4*(SIMD_FP_256:PACKED_DOUBLE)

We verify the events of the list that can be measured simultaneously. At this point we found the first restrictions, the hardware has some restrictions related to number of available counters.

The hardware supports 11 counters. The processor uses 3 fixed counters to measure *core cycles*, *reference cycles* and *executed instructions*. The remaining 8 are programmable counters, which are under other restrictions. If hyperthreading (Marr *et al.*, 2002) is enabled the number is divided between thread, 4 per thread. If the NMI timer is active more one is used, that remains 3 per thread. PAPI assures 3 fixed + 3 programmable counters. Only 3 programmable counters are not enough to map all the events simultaneously.

To overcome this problem it was necessary to design a strategy to make the measures of all event sets into sub parts of the region code. We tried to use PAPI in multiplexed mode to collect all the events. The multiplexed mode in PAPI performs a round-robin on events set. It is need that the program execution time is enough to be divided into intervals that PAPI can switch at all events subsets. PAPI uses the *ITIMER* as measurement period and to reduce the overhead swaps the subset of events process context switch. Events that are not measured in a particular interval are estimated.

The problem in this approach is that the code fraction that we wanted measure was not enough to sustain the program execution by N intervals to measure the N subsets of events in multiplexed mode. In most cases measured only one or two events, the other events were with zero value.

We solved this problem making measures by *chunks* of loop iterations. In each subset of iterations (chunk), the library performs measures of an events set. The measures are interleaved on defined event sets in *round-robin* scheduling until that a percentage of executed code is reached.

4.3 Measuring Chunks

The measurement of events sets is made by *chunks* of iterations loop set. During the execution of chunk iterations one of the event sets will be measured. We have 5 events sets, so the minimum number of *chunks* needed to collect measures is 5 too. But if the *chunk_size* value allows the execution of more chunks to collect measures before reaching the code percentage to be executed, the collected values are accumulated and at the end of measuring phase is done the average by the number of measurements.

Chunks have the same number of iterations and the same code that allows separated measures for each subset of events. The Figure 4.2 shows the measurement mechanism idea. In the original code to get measures for the original loop is needed to surround the for loop with calls to *start* and *stop* measures.

The *collector thread* is registered in the beginning of each loop. This thread needs to measure an event set for each chunk that represent the original loop for a subset of iterations. So, it is need to measure the chunk and in the same way surrounding the code with start and stop controls.

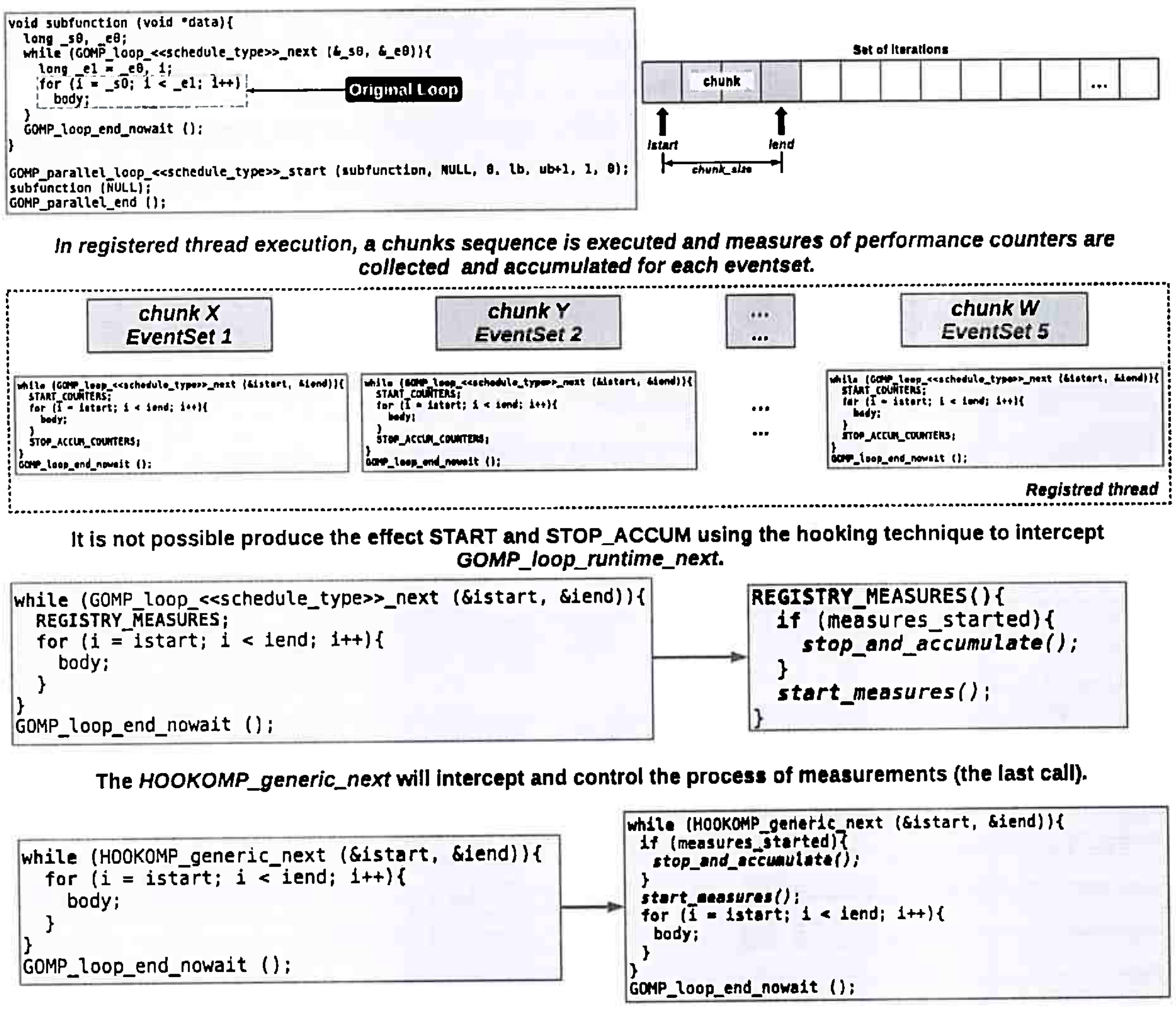
Using the hooking technique described in Chapter 3, we are not modifying the code. We do not modify the application code or the OpenMP runtime. So, it is not possible to put START_COUNTERS and STOP_ACCUM_COUNTERS surrounding the loop code.

We created one function REGISTRY_MEASURES that is able to start and stop/accumulate the measures. A *chunk* starts the measures and it measuring period finish at the begin of next *chunk*.

4.4 Decision Model

We will use a top-down approach to presenting the decision model. It is a very simple model, because at run time we need a model to decide about the offloading quickly.

Starting with the high level function that returns if the offloading is possible and which is the best device for offloading. The offloading decision is taken with basis in Algorithm 1.



```
void subfunction (void *data){
  long _s0, _e0;
  while (GOMP_loop_<<schedule_type>>_next (&_s0, &_e0)){
    long _e1 = _e0, i;
    for (i = _s0; i < _e1; i++)
      body;
  }
  GOMP_loop_end_nowait ();
}

GOMP_parallel_loop_<<schedule_type>>_start (subfunction, NULL, 0, lb, ub+1, 1, 0);
subfunction (NULL);
GOMP_parallel_end ();
```

In registered thread execution, a chunks sequence is executed and measures of performance counters are collected and accumulated for each eventset.

```
while (GOMP_loop_<<schedule_type>>_next (&istart, &iend)){
  START_COUNTERS;
  for (i = istart; i < iend; i++){
    body;
  }
  STOP_ACCUM_COUNTERS;
}
GOMP_loop_end_nowait ();
```

```
while (GOMP_loop_<<schedule_type>>_next (&istart, &iend)){
  START_COUNTERS;
  for (i = istart; i < iend; i++){
    body;
  }
  STOP_ACCUM_COUNTERS;
}
GOMP_loop_end_nowait ();
```

```
while (GOMP_loop_<<schedule_type>>_next (&istart, &iend)){
  START_COUNTERS;
  for (i = istart; i < iend; i++){
    body;
  }
  STOP_ACCUM_COUNTERS;
}
GOMP_loop_end_nowait ();
```

It is not possible produce the effect START and STOP_ACCUM using the hooking technique to intercept *GOMP_loop_runtime_next.*

```
while (GOMP_loop_<<schedule_type>>_next (&istart, &iend)){
  REGISTRY_MEASURES;
  for (i = istart; i < iend; i++){
    body;
  }
}
GOMP_loop_end_nowait ();
```

```
REGISTRY_MEASURES(){
  if (measures_started){
    stop_and_accumulate();
  }
  start_measures();
}
```

The *HOOKOMP_generic_next* will intercept and control the process of measurements (the last call).

```
while (HOOKOMP_generic_next (&istart, &iend)){
  for (i = istart; i < iend; i++){
    body;
  }
}
GOMP_loop_end_nowait ();
```

```
while (HOOKOMP_generic_next (&istart, &iend)){
  if (measures_started){
    stop_and_accumulate();
  }
  start_measures();
  for (i = istart; i < iend; i++){
    body;
  }
}
GOMP_loop_end_nowait ();
```

The first chunk starts the measures and stop at the begin of next chunk.

Figure 4.2: *Mechanism for measuring chunks*

Algorithm 1 $RM\_decision\_about\_offloading()$

```
1: INPUT: * $better\_device\_index$
2: OUTPUT: * $better\_device\_index, offload\_decison$
3: $offload\_decision \leftarrow true$
4: if ($offload\_decision \leftarrow$ RM_check_all_eventsets_was_collected()) then
5:     $oi \leftarrow$ RM_get_operational_intensity()
6:     $better\_device\_index \leftarrow$ RM_get_better_device_to_execution($oi$)
7: else
8:     $better\_device\_index \leftarrow 0$
9: end if
10: return $offload\_decision$
```

Our model verify if all event sets were collected and try to decide about the better device to execute the code based on *operational intensity.* If have a positive decision for offloading it returns to hook library the better device id. The *hook library* that is controlling the threads execution and is able to choose the appropriate code in table of alternative functions. Otherwise, returns that the better device is a CPU (device: 0) and the execution of multicore version continues to be made by the OpenMP threads.

The *operational intensity* that is used as reference in the model is calculated considering the memory hierarchy levels of processor side. How it is calculated is presented in the Algorithm 2.

Algorithm 2 *RM_get_operational_intensity()*

```
1: INPUT:$\emptyset$
2: OUTPUT:$I$
3: $I \leftarrow 0.0$
4: $W \leftarrow 0.0$
5: $Q, Q_{MEM}, Q_{LLC}, Q_{L2}, Q_{L1} \leftarrow 0.0$
6: $W \leftarrow \mathbf{work()}$
7: $Q_{L1} \leftarrow Q_{level}(IDX\_L1)$
8: $Q_{L2} \leftarrow Q_{L1} + Q_{level}(IDX\_L2)$
9: $Q_{LLC} \leftarrow Q_{L2} + Q_{level}(IDX\_LLC)$
10: $Q_{MEM} \leftarrow Q_{LLC} + Q_{level}(IDX\_MEM)$
11: $Q \leftarrow Q_{MEM}$
12: $I \leftarrow \frac{W}{Q}$
13: return $I$
```

The better device to execute the code is chosen using the Algorithm 3. For each selected device, the *attainable performance* is calculated based on the Equation 4.5 considering values for *performance, memory bandwidth* and the *calculated operational intensity.*

Algorithm 3 *RM_get_better_device_to_execution()*

```
1: INPUT:$I$
2: OUTPUT:$dev$
3: $I_{GPU} \leftarrow$ RM_get_operational_intensity_in_GPU()
4: $dev \leftarrow 0,$
5: $T_{exec} \leftarrow DBL\_MAX$
6: $AP, AP' \leftarrow 0.0$
7: $W \leftarrow$ work()
8: for $d = 0$ to $NUM\_DEVICES - 1$ do
9:     $I_{dev} \leftarrow select(I, I_{GPU})$
10:    $AP' \leftarrow$ RM_attainable_performance($d, I_{dev}$)
11:    $T'_{exec} \leftarrow$ RM_execution_time(d, W, AP')
12:    if $(T'_{exec} < T_{exec})$ then
13:        $(T_{exec}, AP, dev) \leftarrow (T'_{exec}, AP', d)$
14:    end if
15: end for
16: return $dev$
```

The device with the better execution time is chosen and its *id* is returned for the offloading decision. The attainable performance is calculated using the Equation 4.5 for each device and the better device is chosen.

$$AP = MIN(FLOPS_{dev}, (MB_{dev} * I)) \tag{4.5}$$

The attainable performance calculus is implemented in the Algorithm 4. The values of performance (FLOPS) and Memory Bandwidth (MB) can be retrieved from device manuals or can be calculated using benchmarks. In this model we are using both sources to define parameters to represent these values.

Algorithm 4 *RM_attainable_performance()*

```
1: INPUT:$dev, I$
2: OUTPUT:$AP$
3: $AP \leftarrow MIN(FLOPS_{dev}, (MB_{dev} * I))$
4: return $AP$
```

After obtaining the value of the attainable performance, the model calculates the time execution in each device using the Equation 4.6. The execution time (T_{exec}) is calculated applying the *computing time* (T_{comp}) and the data transfers time of Algorithm 7.

$$T_{exec} \leftarrow T_{comp} + \textbf{RM\_time\_data\_transfers(dev)} \tag{4.6}$$

The Algorithm 5 calculates the execution time. It is considering the *computing time* added to time spends in data transfers, if executions in the current device needs to transfer data from main memory to device memory, for example.

Algorithm 5 *RM_execution_time()*

```
1: INPUT:$dev, W, P$
2: OUTPUT:$T_{exec}$
3: $T'_{exec} \leftarrow$ RM_computing_time(W, P)
4: $T_{exec} \leftarrow T'_{exec} +$ RM_time_data_transfers(dev)
5: return $T_{exec}$
```

The *computing time* is calculated using the Equation 4.7 with the number of operations (W) and the calculated attainable performance on device ($AP = P$).

$$T_{comp} \leftarrow \frac{W}{P} \tag{4.7}$$

The time spent in computations is calculated using the Algorithm 6. The parameters are the number of point floating operations (W) and the performance of device (P).

Algorithm 6 *RM_computing_time()*

```
1: INPUT:$W, P$
2: OUTPUT:$T_{comp}$
3: $T_{comp} \leftarrow \frac{W}{P}$
4: return $T_{comp}$
```

As our work tries the offloading to GPU, it is needed to also consider the data transfers between host memory and GPU global memory. These transfers between host and GPU may be considered as one more cache level to launch kernels execution on GPU, and in some way align logically the architectures to compare the attainable performance with the calculated operational intensity. The $Q_{DATA\_TRANSFER}$ is used to calculate the time of data transfers to device, and it is defined on the Equation 4.8.

$$Q_{DATA\_TRANSFER} = Q_{DATA\_TRANSFER\_WRITE} + Q_{DATA\_TRANSFER\_READ} \tag{4.8}$$

The data transfers directions are treated separately, with different latencies. If the host send data to device (H2D) the transfer is considered as WRITE and if the host receive data from device (D2H), as READ. The data transfer time is calculated using the Algorithm 7, the different latency rates are considered.

Algorithm 7 *RM_time_data_transfer()*

```
 1: INPUT:$dev$
 2: OUTPUT:$T_{comp}$
 3: $T_{DATA\_TRANSFER} \leftarrow 0.0$
 4: if ($Type_{dev}$ <> $T_{CPU}$) then
 5:     $DW \leftarrow Q_{DATA\_TRANSFER\_WRITE} * BYTES\_TO\_GB$
 6:     $DR \leftarrow Q_{DATA\_TRANSFER\_READ} * BYTES\_TO\_GB$
 7:     $T_{DATA\_TRANSFER\_WRITE} \leftarrow LAT\_W_{dev} + (DW/EF\_MB\_W_{dev})$
 8:     $T_{DATA\_TRANSFER\_READ} \leftarrow LAT\_R_{dev} + (DR/EF\_MB\_R_{dev})$
 9:     $T_{DATA\_TRANSFER} \leftarrow T_{DATA\_TRANSFER\_READ} + T_{DATA\_TRANSFER\_WRITE}$
10: end if
11: return $T_{DATA\_TRANSFER}$
```

The Algorithm 8 presents the estimative for operational intensity in GPU. The operational intensity is calculated considering the data transfers as an other memory hierarchy level.

Algorithm 8 *RM_get_operational_intensity_in_GPU()*

```
 1: INPUT:$\emptyset$
 2: OUTPUT:$I$
 3: $I \leftarrow 0.0$
 4: $W \leftarrow 0.0$
 5: $Q_{total}, Q_{data\_transfer} \leftarrow 0.0$
 6: $W \leftarrow$ work()
 7: $Q \leftarrow$ Q_total()
 8: $Q_{data\_transfers} \leftarrow Q_{data\_transfer\_read} + Q_{data\_transfer\_write}$
 9: $I \leftarrow \frac{W}{Q_{total}+Q_{data\_tranfers}}$
10: return $I$
```

4.5 Final Considerations

Currently, our runtime is able to choose the more appropriate device (multi-core or GPU) according the operational intensity of the code. We are getting measures in all levels of memory hierarchy to know the amount of bytes that are moved to execute the code.

Chapter 5

Experiments and Results

In this chapter we describe our execution platform, the benchmarks used in the experiments and we present the obtained results with these experiments. We executed some experiments using the `Polybench` (Pouchet *et al.*, 2012) benchmark suite.

According with the presented discussion in the Chapter 3 the input code expected by our runtime is a `OpenMP` prepared with a alternative functions table. For this we modified the benchmarks used in the experiments preparing each one using the `OpenMP` and adding the `CUDA` version. Our `OpenMP` with offloading version was created manually based on these others versions, basically we join the `OpenMP` and `CUDA` versions and mounting the table of pointers to alternative functions. The versions of `GPU` and others targets can be found in `Polybench GPU` (Grauer-Gray *et al.* , 2012b) and `Polybench ACC` (Cavazos *et al.*, 2012).

We executed experiments to verify the *Overhead* of our runtime and *Offloading* decision and performance.

5.1 Execution Platform

The Execution Platform used in experiments is a machine equipped with 2 Intel `Xeon(R)` processors (CPU E5-2630 v2 @ 2.60GHz). Each processor has 6 cores, which results in 24 threads (2×6 *cores* $\times 2$ *threads*). The memory of system has 64GB of RAM, L1 cache of 32KB data, 32KB instruction per core, L2 cache of 256KB per core and L3 cache with 24MB accessible by all cores. The processor architecture is presented in the Figure 5.1.

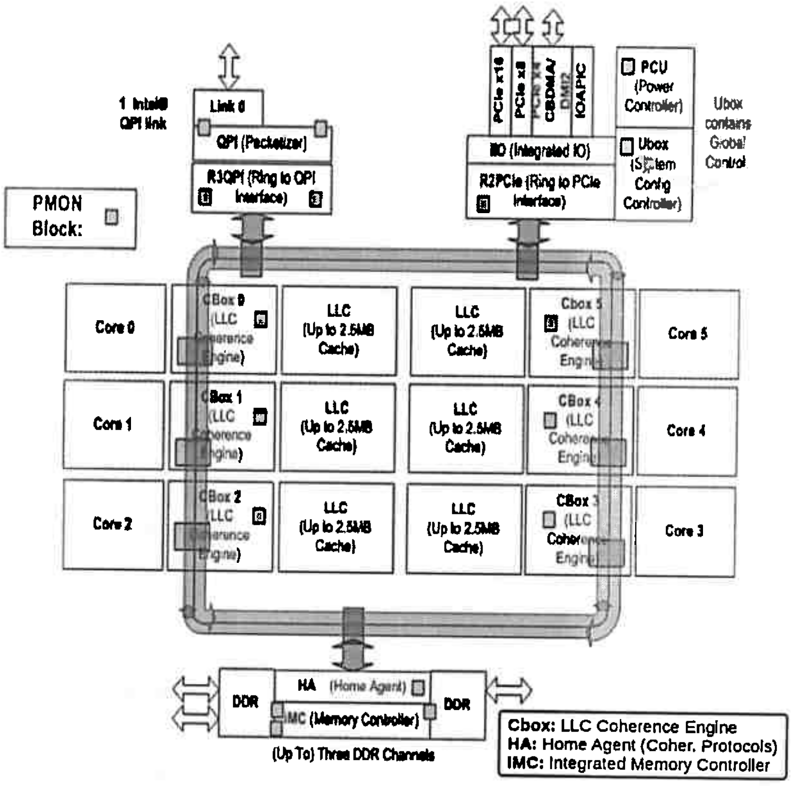

1 Intel® QPI link
Link 0
QPI (Packetizer)
R3QPI (Ring to QPI Interface)
PMON Block:
PCIe x16
PCIe x8
PCIe x4
CBDMA/DMI2
IOAPIC
IIO (Integrated IO)
R2PCIe (Ring to PCIe Interface)
PCU (Power Controller)
Ubox (System Config Controller)
Ubox contains Global Control
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
CBox 0 (LLC Coherence Engine)
CBox 1 (LLC Coherence Engine)
CBox 2 (LLC Coherence Engine)
CBox 3 (LLC Coherence Engine)
CBox 4 (LLC Coherence Engine)
Cbox 5 (LLC Coherence Engine)
LLC (Up to 2.5MB Cache)
HA (Home Agent)
IMC (Memory Controller)
DDR
(Up To) Three DDR Channels
Cbox: LLC Coherence Engine
HA: Home Agent (Coher. Protocols)
IMC: Integrated Memory Controller

Figure 5.1: *Xeon E5-2630 v2 (Intel, 2014)*

This machine has also 2 NVIDIA GPUs `Tesla K40c`, one of which was used in the experiments. The Figure 5.2 presents a complete schema of the available platform. 8

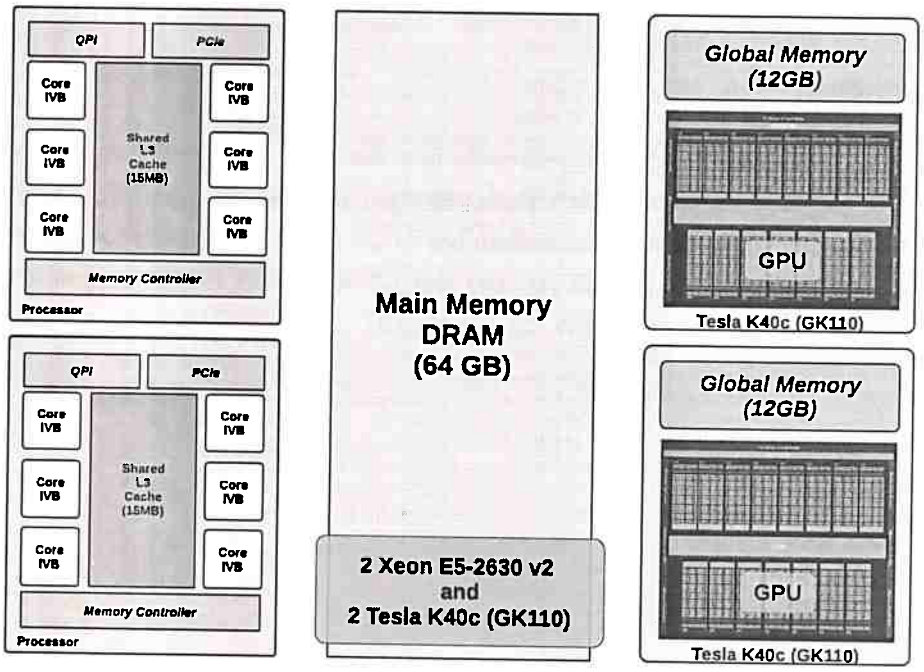


Figure 5.2: *Complete platform*

Detailed informations about the platform are presented in Table 5.1.

Table 5.1: *Platform of Execution description*

Component	Description
CPU	2 Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz. 2 x 6 cores x 2 threads = 24 threads. L1 cache: 32KB data, 32KB instruction per core, L2 cache: 256KB per core, L3 cache: 24 MB accessible by all cores, QPI: 6.4GT/sec
RAM	64GB
2 GPUs	Device [0,1]: "Tesla K40c", (15) Multiprocessors, (192) CUDA Cores/MP: 2880 CUDA Cores, Total amount of global memory: 11520 MBytes (12079136768 bytes), L2 Cache Size: 1572864 bytes, Total number of registers per block: 65536, Warp size: 32, Max. number of threads per MP: 2048, Max. number of threads per block: 1024

5.2 Benchmarks

The benchmarks used in the experiments were prepared using the Polybench 3.2 (Pouchet *et al.*, 2012) OpenMP and CUDA versions. The versions of GPU and others targets can be found in Polybench GPU (Grauer-Gray *et al.*, 2012b) and Polybench ACC (Cavazos *et al.*, 2012; Grauer-Gray *et al.* , 2012a). Our OpenMP with offloading version was created manually based on these others versions, basically we join the OpenMP and CUDA versions and create the table of alternative functions.

The Table 5.2 shows the list of used benchmarks.

Table 5.2: *Benchmarks useds in experiments*

Name	Application	Versions
gemm	Matrix-multiply $C = alpha.A.B + beta.C$	SEQ, OMP, OMP+OFF CUDA

For each of versions (OMP, CUDA, OMP+OFF) was executed the sequential version (SEQ) for comparison.

5.3 Overhead Analysis

To analyze the overhead of our runtime libraries and its impact of the offloading decision in performance, we execute the gemm (Pouchet *et al.*, 2012), a matrix-multiply ($C = alpha.A.B.{+}beta.C$). In this experiment the runtime was forced to decide for not offloading of code. The benchmarks were executed with *num_threads* in $\{1, 2, 4, 8, 10, 12\}$ to compare with the original version (SEQ) and OpenMP version (OMP). As our version of OpenMP with offloading is denoted in experiments as OMP+OFF.

The Figure 5.3 and Figure 5.4 show the results for gemm benchmark in an experiment in which the runtime was forced to decide to not do code offloading. The *chunk_size* was defined as 32 for Figure 5.3 and 64 for Figure 5.4. Each configuration was executed 30 times to calculate the mean and standard deviation values.

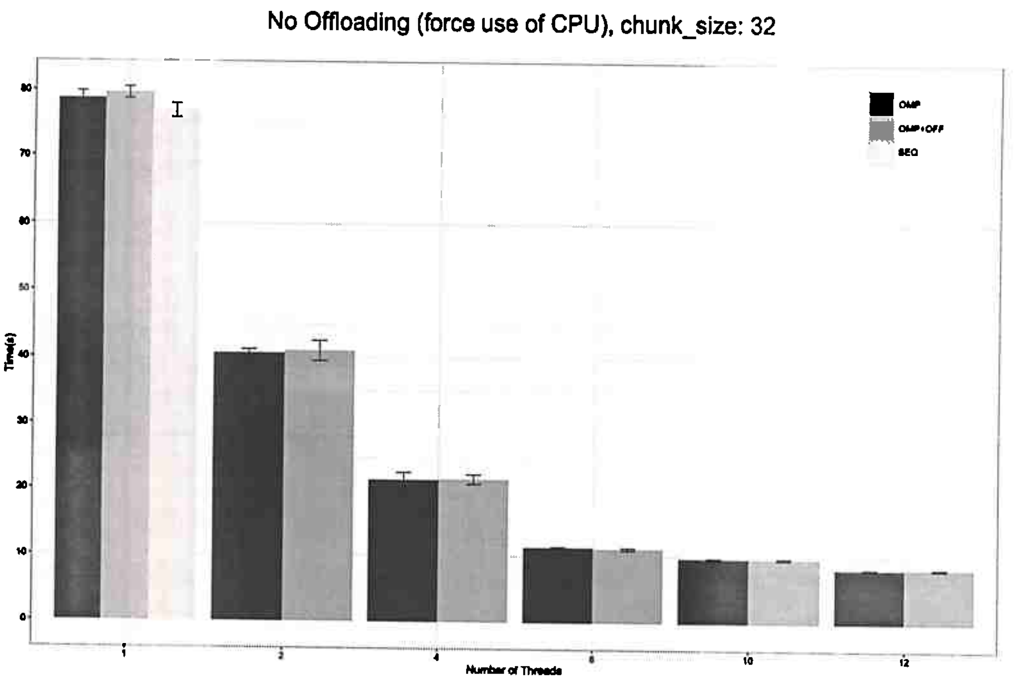


Figure 5.3: Overhead of decision code. *The runtime collect and calculate the decision, but force the execution in CPU side:* no offloading

The overhead of our runtime is relatively small either for single thread as for the other configurations when compared with OMP version.

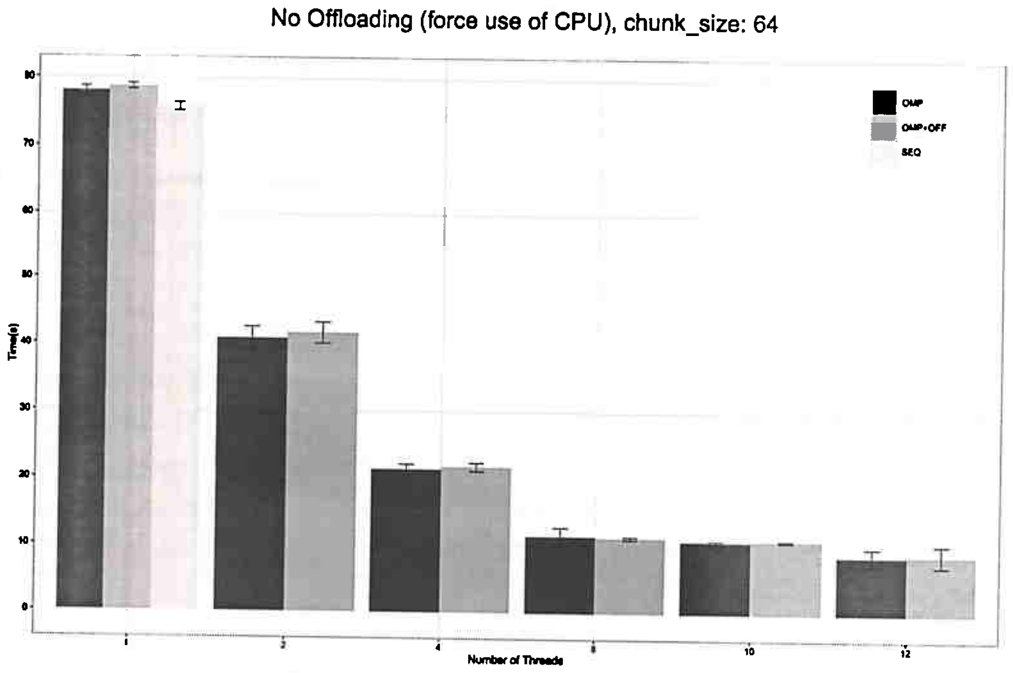


Figure 5.4: Overhead of decision code. *The runtime collect and calculate the decision, but force the execution in CPU side:* no offloading

The obtained overhead is of 1.25% on average when executing in single thread mode. And the difference between OMP and OMP+OFF is minimal when executing other configurations.

5.4 Offloading and Chunk Size Evaluation Analysis

Some executions allowing offloading are shown on Figures 5.5, 5.6, 5.7, 5.8 and 5.9. The runtime use the *operational intensity* to decide on the use of the accelerator.

The second experiment targets the *chunk_size evaluation*. What is the influence of *chunk_size*[1] in the offloading decision? The size of dimensions was $LARGE\_DATASET = 2048$. The Figure 5.5 shows the results for execution with $chunk\_size = 16$, in this case the execution reached the offloading decision point only up to 10 threads. Taking the decision positive for offloading and made the offloading to GPU.

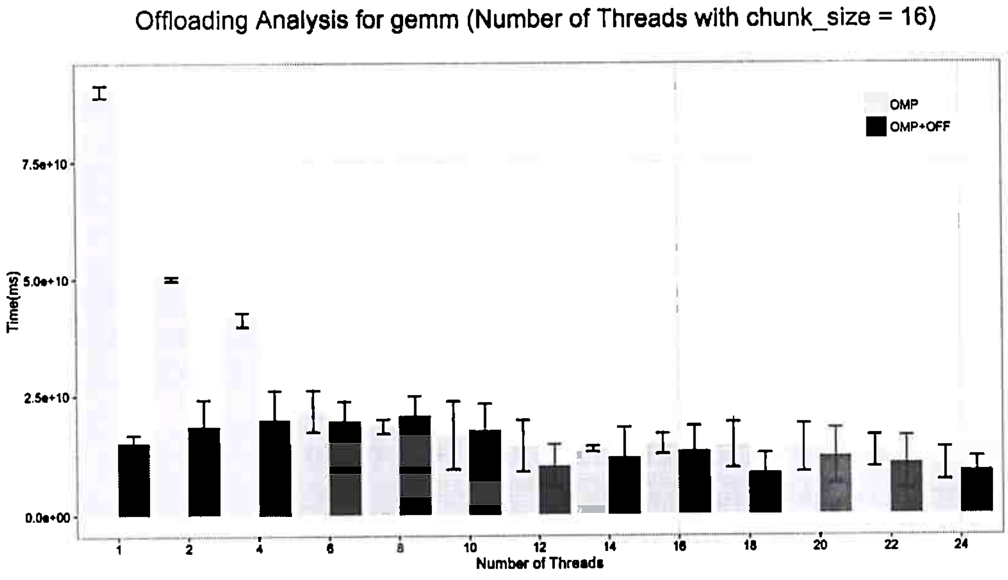


Figure 5.5: Chunk_size evaluation. *Configuration* $chunk\_size = 16$

Still considering the Figure 5.5, in others executions between 12 and 24 threads, the thread that collects the performance counters values does not reach the decision point. The work is finish by others threads favored in system scheduling, then the offloading does not occur.

The same situation happen in execution described by Figure 5.6 for execution with $chunk\_size = 32$, the work is finished by others threads when the program is using more than 10 threads.

[1]subset of loop iterations

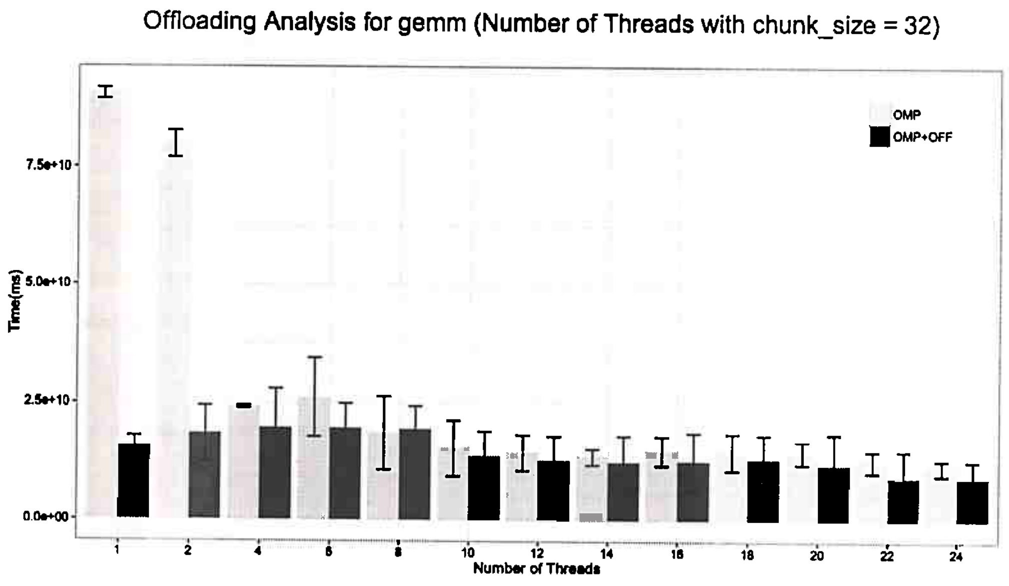


Figure 5.6: Chunk_size evaluation. *Configuration chunk_size* = 32

With the data dimensions size and a fixed *chunk_size* value, if the number of threads is increasing will exist more threads to share and consume the work. So, it is possible that the collector thread had no enough time to get the values for all defined events sets. But if this situation happened is because the data size is no enough to offloading purposes.

The Figure 5.7 shows the values for *chunk_size* = 64. In this execution only 4 threads reachs the offloading decision point. In the executions with 6 threads, the work was finished by the others threads in in 8 of 10 executions and the decision point was not reached. Then the offloading decision was taken and the offloading made only for 2 executions of experiment.

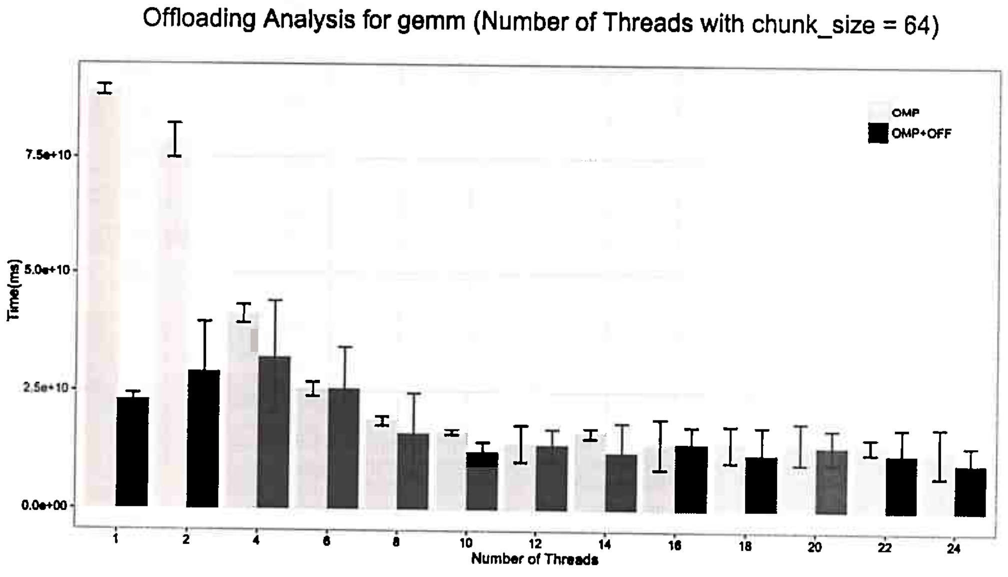


Figure 5.7: Chunk_size evaluation. *Configuration chunk_size* = 64

The values for *chunk_size* = 128 is shown in Figure 5.8. The number of threads which reached the decision point and made the offloading was only 2 threads.

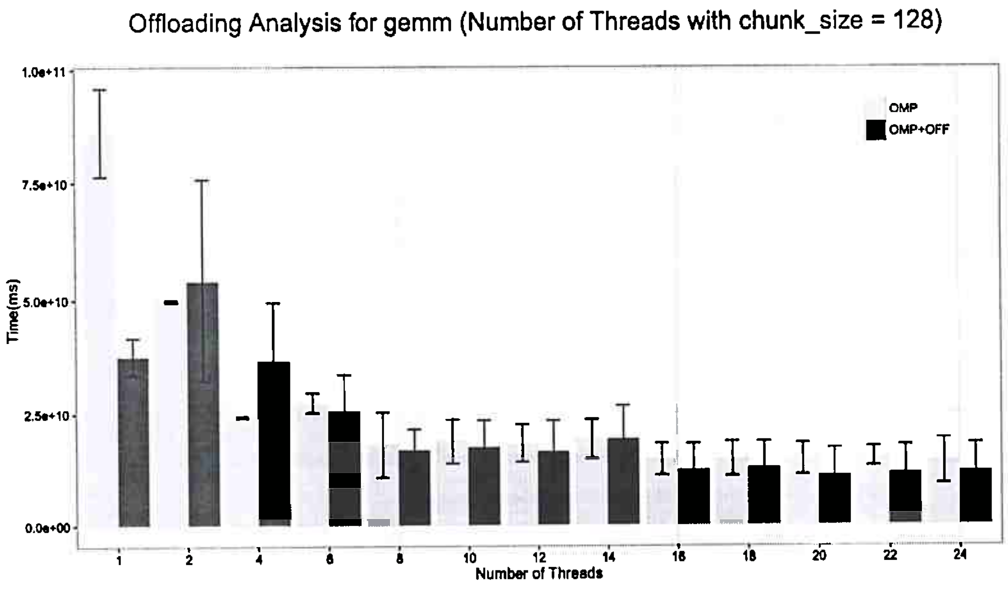


Figure 5.8: Chunk_size evaluation. *Configuration chunk_size* = 128

The Figure 5.9 represents the execution with the last tested configuration, the *chunk_size* = 256. Just the execution with one thread is able to do the collect and make the decision, because the *chunk_size* = 256 is proportionally big considering the dimensions of data. With two or more threads the work is consumed quickly by the threads and the collector thread have not enough chunks of the loop iterations set to get measures on all events sets.

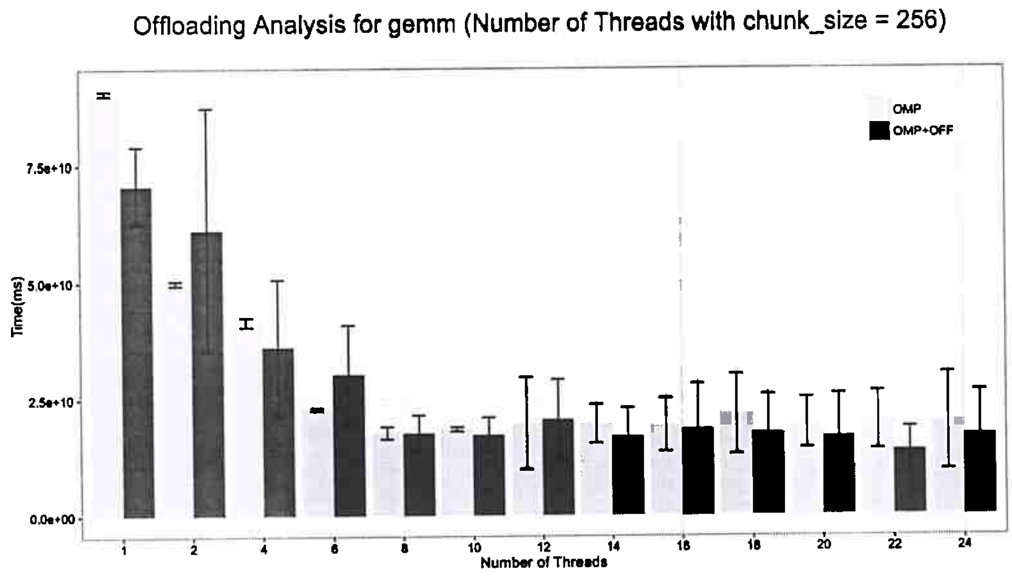


Figure 5.9: Chunk_size evaluation. *Configuration chunk_size* = 256

The Figure 5.10 summarizes the configurations that reaches or not the decision point during the execution.

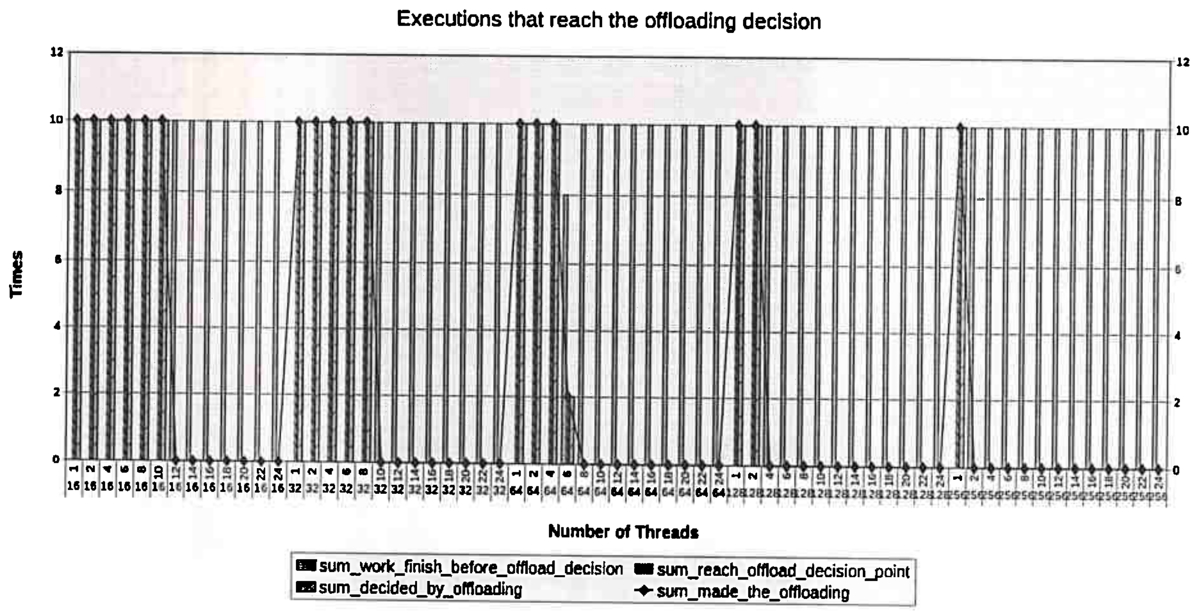


Figure 5.10: *Execution configuration that reaches the offloading decision*

During these executions, the decrease in the number of threads is related to the increase of chunk size. An rate is followed like a pattern.

The relation between the number of threads and the size chunk is $(10,16)$, $(8,32)$, $(4,64)$, $(2,128)$, $(1,256)$. The values in the Figure 5.10 was calculated adding the flag about the offloading occurrence. Being 10 executions for each configuration, the 0 is no made offloading and 10 is o number of execution that took the decision.

5.5 Final Considerations

The experiments results showed that the number of threads in CPU and the *chunk size* value are crucial. We can see this in the Figure 5.10, if the number of threads is increasing the number os times that the collector thread reached the decision point is decreased.

Chapter 6

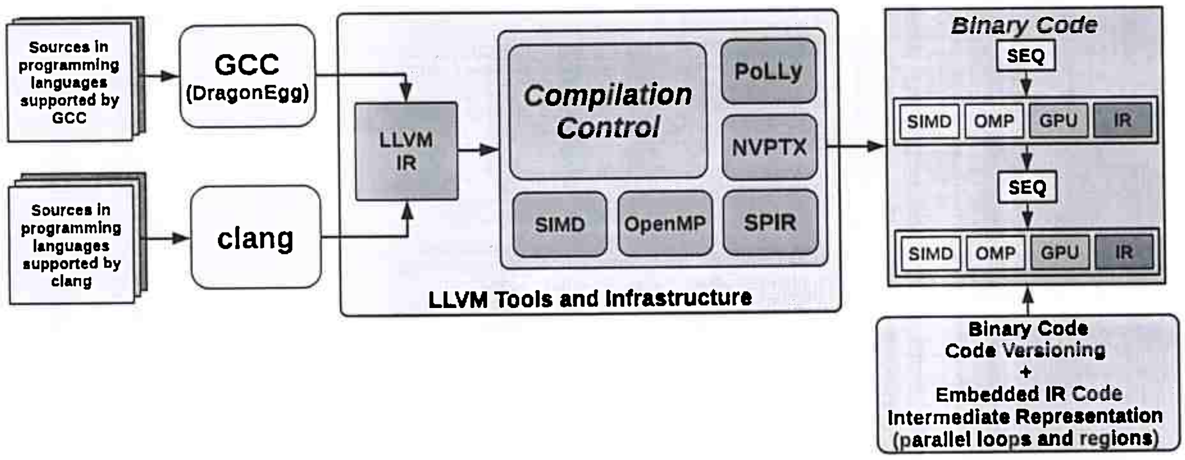
Future Work

In this chapter are presented the future works related with our thesis project. As mentioned previously our approach is related with automatic parallelization and it is divided into two parts: *compilation tool* to prepare the input code with multiple versions for parallel regions, and a *runtime* that can explore heterogeneous elements using operational intensity to decide about code offloading.

The *runtime* part is the focus of this thesis and it was described in the Chapters 3 and 4. The compilation tool will be developed as future work together with other improvements to runtime.

6.1 Compilation Tool Implementation

To build the *compilation tool* that detects parallelizable regions and create the input code to *runtime*, we use the LLVM (Lattner e Adve, 2004) compiler infrastructure. As we want to serve as legacy code applications, in specific various applications written in Fortran and C, languages supported by the GCC compiler and clang (LLVM Clang, 2015). In the cases of Fortran code the DragonEgg plugin (DragonEgg, 2013) can be used in the translation of GCC intermediate code (GIMPLE) to LLVM intermediate representation (LLVM-IR). This step of the compilation process allows bringing code written in different languages to LLVM-IR. When the code is compiled with clang the LLVM-IR is created directly. The Figure 6.1 shows the work flow of compilation process.



Sources in programming languages supported by GCC
GCC (DragonEgg)
Sources in programming languages supported by clang
clang
LLVM IR
Compilation Control
PoLLy
NVPTX
SIMD
OpenMP
SPIR
LLVM Tools and Infrastructure
Binary Code
SEQ
SIMD
OMP
GPU
IR
SEQ
SIMD
OMP
GPU
IR
Binary Code
Code Versioning
+
Embedded IR Code
Intermediate Representation
(parallel loops and regions)

Figure 6.1: *Compilation process workflow*

The compilation tool will find parallelizable regions as loops, and will apply code optimizations and transformations in this regions to generate parallel versions. In this phase, the PoLLy (Grosser e Zheng, 2010; Grosser *et al.*, 2011, 2012) can be used to detect parallel regions. Parallel regions will be extracted and placed inside functions. The modules containing these functions will be used to generate versions, as *code versioning*.

The prepared code for the runtime will be a OpenMP code with an alternative devices functions table. For each parallel loop there will be a handler function for every considered device on the platform. That includes a version able to retrieve the intermediate code for compilation at runtime.

The format of input code is very simple, which allows the runtime to accept generated code by the compilation tool, or code written manually for OpenMP applications that have defined the table of alternatives functions.

6.2 Runtime Improvements

6.2.1 Modify the strategy of collecting the values of Performance Counters

The current approach of measurements strategy collects the values of performance counters during the execution of a code percentage. A detected problem is that depending of threads scheduling on the system, the other threads, which are not collecting the counters, can finish their work before the collector thread have the values to make the decision.

An alternative approach would be switch between the threads registering the thread that make the collect of events set needed to model calculations. The Figure 6.2 shows the idea of interleaved measurements strategy.

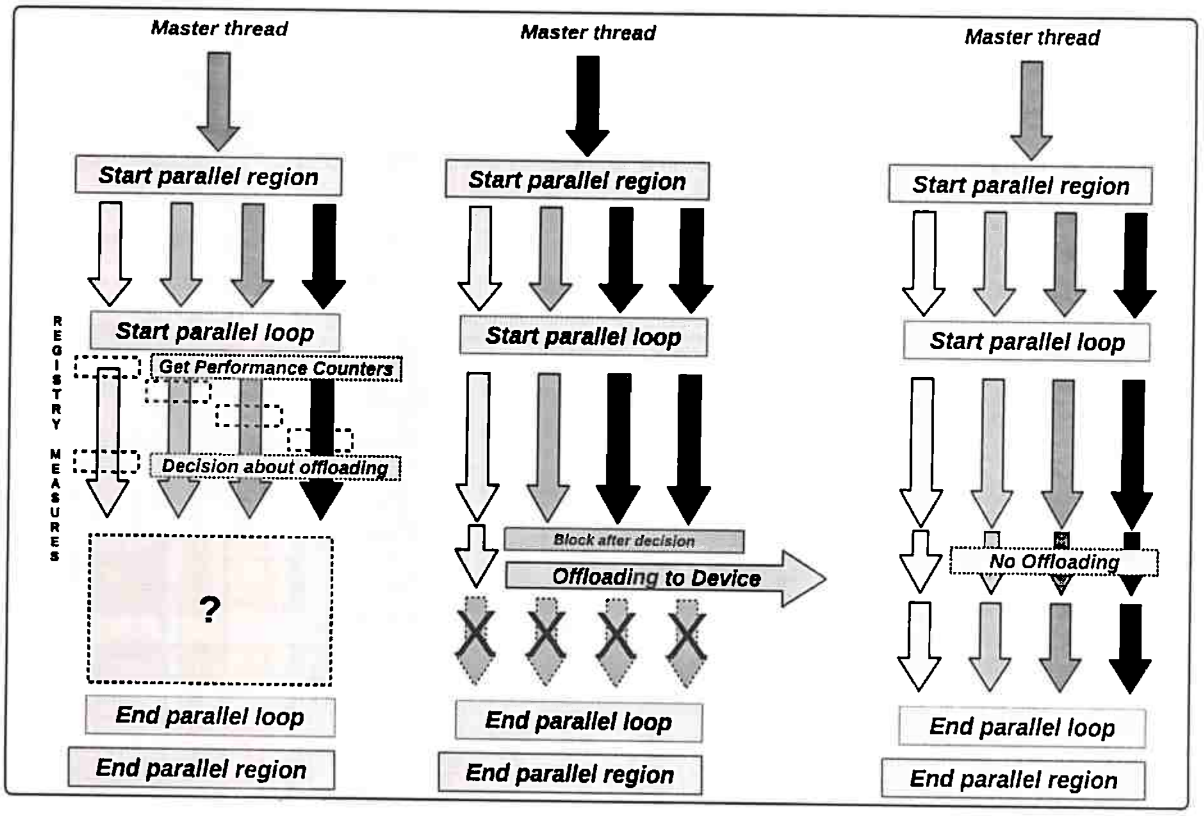


Figure 6.2: *Interleaved measurements strategy (Third version)*

6.2.2 Consider Data Placement issues

The decision model in the current version, calculates the execution time and the offloading decision for each parallelizable loop, considering that the data are always in host memory. Because it use the time of input and output data transfers to make the decision.

Assuming that the code has two adjacent loops that are using the same data, or the second loop uses the results of the first. If model decided to make the offloading of the first loop, the data were transferred to device memory. The decision of launching the execution of second kernel in the same device that run the first kernel or if it is better to launch it on another device or return execution to the CPU, must consider the data placement trying to minimize the data transfer.

If the better execution time is related to do not move the data from memory device that run the first kernel, the second kernel must also have their execution launched on the same device for it can take advantage in the sharing of resident data.

6.2.3 Speculative Data Copy to device

In the current version, the data that is used by the kernel are transfered on demand, when the decision is taken in favour of the offloading.The idea is to copy data to memory of device before of the decision, in a speculative way.

Data copies can be made at the beginning in overlap with the measurement session execution. In this case, if the decision is positive for code offloading the data would be residents in device memory.

6.2.4 Data Synchronization issues

The current version of runtime re-execute the code launching the kernel on GPU when it has a positive decision for offloading. Besides of data copy issues (copy on demand or in advance), the *synchronization* of data that were updated by OpenMP threads during the session of measurements may be useful in some context.

6.2.5 The use of other decision models

The interface between the libraries (interception and library that implements the decision model using of operational intensity) allows the model to be modified or replaced by another model.

The function that intercepts the next chunk retrieval by collector thread is implemented with an unique entry point for the measures controls, the function `registry_measures()` (Section 4.3) that can allow the replacement of the model.

6.2.6 Support other accelerators devices and other platforms

In the current version we have offloading support only for GPUs, but it is possible that the input code has versions using OpenCL to try the offloading to other devices. We have interests on provide support for devices as FPGAs (Bacon *et al.*, 2013) and Intel Xeon Phi co-processors (Intel, 2013).

An another interesting platform is the Juno ARM (ARM, 2016) that includes the ***big.LITTLE*** scheduling and power management. This platform have two ARM processors (Cortex-A57 and Cortex-A53 MP Core) that share the same main memory and a Mali GPU. The Cortex-A57 is dual core and Cortex-A53 is a quadcore actually, but the number of cores can vary. Normally, the Operating System (OS) (Distro Linaro Linux) detects 6 cores that can be used at the same time.

The OS implements a module that switches between CPUs based on frequency *(cpufreq)* like a Dynamic Voltage and Frequency Scaling (DVFS). The main idea is start the execution in low power core and If the workload increase the OS switch the execution for the powerful processor. The processors share the same main memory, which is good for avoiding data transfers.

6.2.7 Use of Optimized Vendor Libraries

Standard libraries with optimized versions of functions can be used to mount the alternative functions table. Optimized manufacturers or vendor libraries have been made available for the efficient use of target platforms and making the modernized code (Intel, 2015b). The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) (NVIDIA, 2015d) is a complete standard BLAS library for GPUs.

6.3 Final Considerations

In this chapter it was presented the possible improvements in the runtime and the idea of process flow of compilation tool to generate the input code.

In cases that the collector threads have no enough time the collect all event sets, the problem is related to scheduling on system, but depend of the data size. If the data is very small, probably the offloading must not be made. We need to execute more experiments with benchmark that use data with bigger dimensions and try to implement the interleaved measures.

Chapter 7

Conclusions

Considering the runtime and the experiments results. We can conclude that the *operational intensity* can be used at runtime to take decisions about the code offloading to device accelerators. The use of it proved to be appropriate for rapid decisions at runtime, in accordance com a original idea of Roofline Model of give insights about the performance and about the code behavior.

We are considering all levels of memory hierarchy and data transfers between host memory and device memory. The data transfers are measured and considered on the decision in an attempt to cover data transfers and to align the different architectures through the levels of the memory hierarchy.

Our runtime need of improvements and adjusts in the precision of measures and in the model of decision, because the hardware have limitations to measure events and we are using five events sets to get measures of performance counters.

The measures are made on a minimum percentage of the code for that all events sets have collected values. The *chunk_size* becomes a problem, our measures are made by chunks of loops iterations and the OpenMP runtime not allow modifications in the number of threads and in the *chunk_size* after the start of parallel region. So, it is not possible to adjust the *chunk_size* to make quick measures over small chunks and after increase to desired value to execution. If the *chunk_size* is too large, it can make the measurements suffer a delay until all event sets are measured.

Appendix A

Creating Hooks for OpenMP Functions

In this appendix are available some expanded codes of OpenMP directives. It is a complementary material for the Chapter 3, which presents a study about the OpenMP directives and how their implementation is made.

A.1 Expanded code for parallel region with two loops

In the intermediate representation generated by GCC that is shown in the Code A.1, we can see that the generated code follows the formats to parallel region, in which an outlined function is created to handle the code inside the demarcated parallel region.

```
main () {
  /* Variables declaration was suppressed. */
<bb 2>:
   num_t = 8;
   .omp_data_o.1.n = n;
   num_t.10 = (unsigned int) num_t;
   __builtin_GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, num_t.10)
       ;
   main._omp_fn.0 (&.omp_data_o.1);
   __builtin_GOMP_parallel_end ();
   n = .omp_data_o.1.n;
D.1807 = 0;

<L0>:
   return D.1807;
}

main._omp_fn.0 (struct .omp_data_s.0 * .omp_data_i) {
  /* Variables declaration was suppressed. */

<bb 16>:

<bb 3>:
  D.1836 = __builtin_GOMP_loop_runtime_start (0, 1024, 1, &.istart0.6, &.
      iend0.7);
  if (D.1836 != 0)
    goto <bb 14>;
else
    goto <bb 5>;

<bb 14>:
  .istart0.8 = .istart0.6;
  i = (int) .istart0.8;
  .iend0.9 = .iend0.7;
  D.1840 = (int) .iend0.9;

<bb 4>:
  i = i + 1;
  if (i < D.1840)
    goto <bb 4>;
else
    goto <bb 15>;

<bb 15>:
  D.1841 = __builtin_GOMP_loop_runtime_next (&.istart0.6, &.iend0.7);
  if (D.1841 != 0)
    goto <bb 14>;
else
    goto <bb 5>;

<bb 5>:
  __builtin_GOMP_loop_end ();

<bb 6>:
  D.1842 = .omp_data_i->n;
  D.1843 = (long int) D.1842;
  D.1846 = __builtin_GOMP_loop_dynamic_start (0, D.1843, 1, 32, &.istart0
      .2, &.iend0.3);
  if (D.1846 != 0)
    goto <bb 12>;
else
    goto <bb 8>;

<bb 12>:
  .istart0.4 = .istart0.2;
  j = (int) .istart0.4;
  .iend0.5 = .iend0.3;
  D.1850 = (int) .iend0.5;

<bb 7>:
  j = j + 1;
  if (j < D.1850)
    goto <bb 7>;
  else
```

```
    goto <bb 13>;

<bb 13>:
  D.1851 = __builtin_GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);
  if (D.1851 != 0)
    goto <bb 12>;
  else
    goto <bb 8>;

<bb 8>:
  __builtin_GOMP_loop_end_nowait ();

<bb 9>:
  return;
}
```

Código A.1: *Parallel Regions code with two different loops definitions inside*

A.2 Generic next chunk function

An generic function was created to handle the both loops formats, because we detected two formats that are generated according some code features. The generic function is showed in the Code A.2.

```
bool HOOKOMP_generic_next(long* istart, long* iend, chunk_next_fn fn_proxy
    , void* extra) {
  PRINT_FUNC_NAME;

  /* The first thing is stop the last chunk measure. */
  if(registred_thread_executing_function_next == (long int) pthread_self()
      ){
    RM_registry_measures();
  }

  bool result = false;

  /* Registry the thread which will be execute and get measures. */
  if(!thread_was_registred_to_execute_measures){
    /* Calculate the max iterations need to measures. */
    max_loops_iterations_for_measures = ((total_of_iterations *
        percentual_of_code) / 100);

    HOOKOMP_registry_the_first_thread();
  }

  /* If is not getting measures execute directly. */
  if(!is_executing_measures_section){ /* Call directly. */
    result = fn_proxy(istart, iend, extra);
  }
```

```
else{ /* Measuring session. */
  if(registred_thread_executing_function_next == (long int) pthread_self
     ()){ /* Registered thread. */
    if(executed_loop_iterations < max_loops_iterations_for_measures){ /*
       No reached the percentual. */
      result = fn_proxy(istart, iend, extra);
      // No more work to do.
      if(!result){
        /* The shared work finish before the decision. */
        reach_offload_decision_point = false;
      }
      else {
        /* Update the number of iterations executed by this thread. */
        executed_loop_iterations += (*iend - *istart);
        /* Starting the registry on RM library. Is necessary partial
           measures each chunk.
        Switching to do not get measures considering control code. */
        RM_registry_measures();
      }
    }
    else{ /* Offloading decision. */
      TRACE("[HOOKOMP]: Trying to make decision about offloading.\n");

      /* Reach the offloading decision point.*/
      reach_offload_decision_point = true;

      long better_device = 0;

      TRACE("Defining additional parameters.\n");
      /* N: total of iterations, Number of executed iterations (
         percentual), last chunk_size. */
      RM_set_aditional_parameters(total_of_iterations,
         executed_loop_iterations, (*iend - *istart),
         q_data_transfer_write, q_data_transfer_read,
         type_of_data_allocation);

      RM_print_counters_values_csv();

      TRACE("Getting decision about offloading.\n");
      if((decided_by_offloading = RM_decision_about_offloading(&
         better_device)) != 0){
        /* Launch appropriated function. */
        TRACE("Trying to launch appropriated function to loop %d on
           device: %d.\n", current_loop_index, better_device);

        if((made_the_offloading = HOOKOMP_call_offloading_function(
           current_loop_index, better_device)) == 0){
          TRACE("The function offloading was not done.\n");
        }
        else{
          TRACE("The offloading was done launching of appropriated function to loop %d on device: %d.\n", current_loop_index, better_device);
        }
      }
      else {
        TRACE("Is not possible decide about offloading.\n");
      }

      TRACE("After decision about offloading.\n");

      /* Continue execution. */
      if(!(decided_by_offloading && made_the_offloading)){
        TRACE("[HOOKOMP]: [CONTINUE] Calling next function after offloading decision about.\n");
        result = fn_proxy(istart, iend, extra);
      }

      /* Mark that is no more in section of measurements. */
      is_executing_measures_section = false;
      executed_loop_iterations = 0;

      /* Release all blocked team threads. */
      if(number_of_blocked_threads > 0){
        HOOKOMP_release_all_team_threads();
      }
    }

  }
  else { /* Others threads. */
    /* If decide by offloading: block the other threads to wait. */
    if(decided_by_offloading){
      sem_wait(&mutex_verify_number_of_blocked_threads);

      if(number_of_blocked_threads < (number_of_threads_in_team - 1)) {
        number_of_blocked_threads++;

        sem_post(&mutex_verify_number_of_blocked_threads);
        TRACE("[HOOKOMP]: Thread [%lu] will be blocked.\n", (long int) pthread_self());
        sem_wait(&sem_blocks_other_team_threads);
        TRACE("[HOOKOMP]: Thread [%lu] is waking up of block.\n", (long int) pthread_self());
      }
    }
    TRACE("[HOOKOMP]: Other thread in execution, verifying if made by offloading: %d\n", made_the_offloading);

```

```
        /* After the wakeup of blocked or while the offloading was not made.
            */
        if(!made_the_offloading){
          TRACE("[HOOKOMP]: [OTHERS/WAKE UP] Calling next function out of
              measures section after wake up.\n");
          result = fn_proxy(istart, iend, extra);
        }
        else{ /* Indicates have no more work to do. */
          result = false;
        }
      }

    }
    TRACE("[HOOKOMP]: Leaving the %s.\n", __FUNCTION__);
    return result;
}
```

Código A.2: *The generic function to handle the next chunk functions*

References

Aleksandar Ilic e Sousa (2015) Frederico Pratas Aleksandar Ilic and Leonel Sousa. CARM: Cache-Aware Performance, Power and Energy-Efficiency Roofline Modeling. Em *Compiler, Architecture and Tools Conference (CATC 2015)*, CATC'15. Intel. Cited on page 12

Amini *et al.* **(2011)** Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell and Pierre Villalon. PIPS is not (just) Polyhedral Software. Em *International Workshop on Polyhedral Compilation Techniques (IMPACT'11), Chamonix, France*, páginas 1–6. Cited on page 8

Amini *et al.* **(2012)** Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian Mcmahon, François-Xavier Pasquier, Grégoire Péan and Pierre Villalon. Par4All: From Convex Array Regions to Heterogeneous Computing. Em *IMPACT 2012 : 2nd International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, Paris, France. URL: https://hal-mines-paristech.archives-ouvertes.fr/hal-00744733. 2 pages. Cited on page 2, 7, 8

Aparapi (2011) Aparapi. Aparapi. API for data parallel Java, sep 2011. URL: https://code.google.com/p/aparapi/. Cited on page 2

ARM (2016) ARM. Juno ARM Development Platform, January 2016. URL: https://www.arm.com/products/tools/development-boards/versatile-express/juno-arm-development-platform.php. Cited on page 65

Asanovic *et al.* **(2009)** Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel and Katherine Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52(10):56–67. ISSN 0001-0782. doi: 10.1145/1562764.1562783. URL: http://dl.acm.org/citation.cfm?id=1562783. Cited on page 3

Augonnet *et al.* **(2010)** Cédric Augonnet, Samuel Thibault and Raymond Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Research Report RR-7240, INRIA. URL: https://hal.inria.fr/inria-00467677. Cited on page 7, 9

Bacon *et al.* **(2013)** David Bacon, Rodric Rabbah and Sunil Shukla. FPGA Programming for the Masses. *Queue*, 11(2):40:40–40:52. ISSN 1542-7730. doi: 10.1145/2436696.2443836. URL: http://doi.acm.org/10.1145/2436696.2443836. Cited on page 65

Baskaran *et al.* **(2010)** Muthu Manikandan Baskaran, J. Ramanujam and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. Em *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, páginas 244–263, Berlin, Heidelberg. Springer-Verlag. ISBN 3-642-11969-7, 978-3-642-11969-9. doi: 10.1007/978-3-642-11970-5_14. URL: http://dx.doi.org/10.1007/978-3-642-11970-5_14. Cited on page 2, 8

Bastoul (2004) Cédric Bastoul. Code generation in the polyhedral model is easier than you think. Em *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, páginas 7–16, Juan-les-Pins, France. Cited on page 2, 8

Benabderrahmane *et al.* (2010) Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen and Cédric Bastoul. The Polyhedral Model is More Widely Applicable Than You Think. Em *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, páginas 283–303, Berlin, Heidelberg. Springer-Verlag. ISBN 3-642-11969-7, 978-3-642-11969-9. doi: 10.1007/978-3-642-11970-5_16. URL: http://dx.doi.org/10.1007/978-3-642-11970-5_16. Cited on page 2, 8

Bondhugula (2012) Uday Bondhugula. Pluto - an automatic parallelizer and locality optimizer for multicores {@online}, 2012. URL: http://pluto-compiler.sourceforge.net/. Cited on page 8

Bondhugula *et al.* (2007) Uday Bondhugula, J. Ramanujam and P. Sadayappan. PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University. Cited on page 8

Bondhugula *et al.* (2008) Uday Bondhugula, Albert Hartono, J. Ramanujam and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113. ISSN 0362-1340. doi: 10.1145/1375581.1375595. URL: http://doi.acm.org/10.1145/1375581.1375595. Cited on page 8

Caparrós Cabezas e Püschel (2014) Victoria Caparrós Cabezas and Markus Püschel. Extending the Roofline Model: Bottleneck Analysis with Microarchitectural Constraints. Em *2014 IEEE International Symposium on Workload Characterization (IISWC)*, páginas 222–231. doi: 10.1109/IISWC.2014.6983061. Cited on page 12

CAPS (2012) CAPS. OpenHMPP Directives, November 2012. URL: http://www.caps-entreprise.com/openhmpp-directives/. Cited on page 2

Cavazos *et al.* (2012) John Cavazos, Scott Grauer-Gray, Robert Searles, William Killian, Lifan Xu and Sudhee Ayalasomayajula. PolyBench/ACC The Polyhedral Benchmark Suite Targeting Multicore CPUs, GPUs, and Accelerators, May 2012. URL: http://cavazos-lab.github.io/PolyBench-ACC/. Cited on page 55, 57

Chapman *et al.* (2007) Barbara Chapman, Gabriele Jost and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press. ISBN 0262533022, 9780262533027. Cited on page 2

Creusillet e Irigoin (1996) Béatrice Creusillet and François Irigoin. Interprocedural Array Region Analyses. *Languages and Compilers for Parallel Computing*, páginas 46–60. Cited on page 8

Dagum e Menon (1998) Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55. ISSN 1070-9924. doi: 10.1109/99.660313. URL: http://dl.acm.org/citation.cfm?id=615255.615542. Cited on page 2, 16

Dongarra *et al.* (1994) Jack J Dongarra, Hans W Meuer and Erich Strohmaier. TOP500 Supercomputer Sites. Technical Report, RUM 37/94, University of Mannheim, June 30, 1994. URL: www.netlib.org/benchmark/top500/lists/rum3894.ps. Cited on page 1

Dongarra *et al.* (1996) Jack J Dongarra, Hans W Meuer and Erich Strohmaier. TOP500 Supercomputer Sites. Technical Report, RUM 48/96, University of Mannheim, November 18, 1996. URL: http://www.netlib.org/utk/people/JackDongarra/pdf/top500_9611.pdf. Cited on page 1

DragonEgg (2013) DragonEgg. DragonEgg - Using LLVM as a GCC backend, September 2013. URL: http://dragonegg.llvm.org/. Cited on page 63

GCC (2015) GCC. Gcc, the gnu compiler collection, 2015. URL: https://gcc.gnu.org/. Cited on page 2, 16

Glaskowsky (2009) Peter N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. Whitepaper, NVIDIA. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA%27s_Fermi-The_First_Complete_GPU_Architecture.pdf. Cited on page 1

GNU Libgomp (2015a) GNU Libgomp. GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation. Technical Report, GNU. URL: https://gcc.gnu.org/onlinedocs/gcc-5.2.0/libgomp.pdf. Cited on page 2, 13, 16, 17

GNU Libgomp (2015b) GNU Libgomp. GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation. Technical Report, GNU libgomp. URL: https://gcc.gnu.org/onlinedocs/gcc-5.3.0/libgomp.pdf. Cited on page 2, 13, 16

GNU Libgomp (2016a) GNU Libgomp. GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation. Technical Report, GNU libgomp. URL: https://gcc.gnu.org/onlinedocs/gcc-6.1.0/libgomp.pdf. Cited on page 2, 13, 16

GNU Libgomp (2016b) GNU Libgomp. GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation. Technical Report, GNU libgomp. URL: https://gcc.gnu.org/onlinedocs/gcc-5.4.0/libgomp.pdf. Cited on page 2, 13, 16

GNU Libgomp (2015c) GNU Libgomp. GNU libgomp, GNU Offloading and Multi Processing Runtime Library documentation (Online manual), Aug 2015c. URL: https://gcc.gnu.org/onlinedocs/libgomp/. Cited on page 2, 16, 17, 19

Grauer-Gray *et al.* (2012a) S. Grauer-Gray, Lifan Xu, R. Searles, S. Ayalasomayajula and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. Em *Innovative Parallel Computing (InPar), 2012*, páginas 1–10. doi: 10.1109/InPar.2012.6339595. Cited on page 57

Grauer-Gray *et al.* (2012b) Scott Grauer-Gray, Will Killian and Louis-Noël Pouchet. PolyBench/GPU Implementation of PolyBench codes for GPU processing, March 2012b. URL: http://web.cse.ohio-state.edu/~pouchet/software/polybench/GPU/. Cited on page 55, 57

Grosser e Hoefler (2016) Tobias Grosser and Torsten Hoefler. Polly-ACC Transparent Compilation to Heterogeneous Hardware. Em *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, páginas 1:1–1:13, New York, NY, USA. ACM. ISBN 978-1-4503-4361-9. doi: 10.1145/2925426.2926286. URL: http://doi.acm.org/10.1145/2925426.2926286. Cited on page 2, 7, 9

Grosser e Simbürger (2014) Tobias Grosser and Andreas Simbürger. Polyhedral.info, Fev 2014. URL: http://polyhedral.info/. Cited on page 2, 8

Grosser e Zheng (2010) Tobias Grosser and Hongbin Zheng. Polly - Polyhedral Transformations for LLVM. LLVM Developer Meeting 2010. Cited on page 2, 8, 64

Grosser *et al.* (2011) Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger and Louis-Noël Pouchet. Polly - polyhedral optimization in llvm. Em *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011. URL: http://perso.ens-lyon.fr/christophe.alias/impact2011/impact-07.pdf. Cited on page 2, 8, 16, 64

Grosser *et al.* (2012) Tobias Grosser, Armin Groesslinger and Christian Lengauer. POLLY - Performing Polyhedral Optimizations On A Low-level Intermediate Representation. *Parallel Processing Letters*, 22(04):1250010. ISSN 0129-6264. doi: 10.1142/S0129626412500107. URL: http://www.worldscientific.com/doi/abs/10.1142/S0129626412500107. Cited on page 2, 8, 16, 64

Guelton *et al.* (2011) Serge Guelton, Mehdi Amini, Ronan Keryell and Béatrice Creusillet. PyPS a programmable pass manager. The 24th International Workshop on Languages and Compilers for Parallel Computing, September 2011. URL: https://hal-mines-paristech.archives-ouvertes.fr/hal-01087303. Poster. Cited on page 8

Han e Abdelrahman (2009) Tianyi David Han and Tarek S Abdelrahman. hiCUDA: A High-level Directive-based Language for GPU Programming. Em *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, páginas 52–61, New York, NY, USA. ACM. ISBN 978-1-60558-517-8. doi: http://doi.acm.org/10.1145/1513895.1513902. Cited on page 2

Han e Abdelrahman (2011) Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1): 78–90. ISSN 1045-9219. doi: 10.1109/TPDS.2010.62. URL: http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.62. Cited on page 2

hiCUDA Project (2012) hiCUDA Project. The hicuda project site, oct 2012. URL: http://www.eecg.utoronto.ca/~tsa/hicuda/. Cited on page 2

Ilic *et al.* (2014) Aleksandar Ilic, Frederico Pratas and Leonel Sousa. Cache-aware Roofline Model: Upgrading the Loft. *IEEE Computer Architecture Letters*, 13(1):21–24. ISSN 1556–6056. doi: 10.1109/L-CA.2013.6. Cited on page 12

Intel (2015a) Intel. Intel(r) 64 and IA-32 Architectures Software Developer's Manual. Volume 3B:System Programming Guide, Part 2. Manual, Intel. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf. Cited on page 12

Intel (2015b) Intel. What is Code Modernization?, July 2015b. URL: https://software.intel.com/en-us/articles/what-is-code-modernization. [Online; posted by Michel P. at 08-July-2015]. Cited on page 1, 3, 66

Intel (2016a) Intel. Openmp* support, Jan 2016a. URL: https://software.intel.com/pt-br/node/522678. Cited on page 2

Intel (2016b) Intel. IntelÂ® OpenMP* Runtime Library Interface. Technical Report, Intel. URL: https://www.openmprtl.org/sites/default/files/resources/libomp_20160322_manual.pdf. OpenMP* 4.5. Cited on page 2

Intel (2015c) Intel. Intel(r) Performance Counter Monitor - A better way to measure CPU utilization, June 2015c. URL: https://software.intel.com/en-us/articles/intel-performance-counter-monitor. Cited on page 12

Intel (2014) Intel. Intel(r) Xeon(r) Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual. Technical Report, Intel. URL: http://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-2600-v2-uncore-manual.html. Cited on page viii, 56

Intel (2013) Intel. The intel(r) xeon phi(tm) 5110p coprocessor, jan 2013. URL: http://www.intel.com.br/content/www/br/pt/processors/xeon/xeon-phi-detail.html. Cited on page 1, 3, 65

Jablin *et al.* (2011) Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard and David I. August. Automatic CPU-GPU Communication Management and Optimization. Em *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 46 of *PLDI '11*, páginas 142–151, New York, NY, USA. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993516. URL: http://doi.acm.org/10.1145/1993498.1993516. Cited on page 2

JCuda (2012) JCuda. Jcuda site, June 2012. URL: http://www.jcuda.org/. Cited on page 2

JOCL (2012) JOCL. Jocl site, June 2012. URL: http://www.jocl.org/documentation/documentation.html. Cited on page 2

Khronos (2013) Khronos. OpenCL - The open standard for parallel programming of heterogeneous systems, January 2013. URL: http://www.khronos.org/opencl/. Cited on page 1, 7

Klöckner *et al.* (2012) Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B Catanzaro, Paul Ivanov and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174. ISSN 0167-8191. doi: 10.1016/j.parco.2011.09.001. Cited on page 2

Konstantinidis e Cotronis (2015) Elias Konstantinidis and Yiannis Cotronis. A Practical Performance Model for Compute and Memory Bound GPU Kernels. Em *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, páginas 651–658. IEEE. ISBN 978-1-4799-8491-6. doi: 10.1109/PDP.2015.51. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7092788. Cited on page 12

Lattner (2008) Chris Lattner. LLVM and Clang: Next generation compiler technology. Em *The BSD Conference, Ottawa, Canada.* Cited on page 8

Lattner e Adve (2004) Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Em *Proceedings of the International Symposium on Code Generation and Optimization*, number c in CGO '04, páginas 75–86, Palo Alto, California. IEEE Computer Society. ISBN 0-7695-2102-9. doi: 10.1109/CGO.2004.1281665. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1281665. Cited on page 2, 8, 16, 63

Lee e Eigenmann (2010) Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. Em *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, páginas 1–11, Washington, DC, USA. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.36. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5644879. Cited on page 2

Linux Man-pages (2016) Linux Man-pages. The Linux man-pages project, March 2016. URL: https://www.kernel.org/doc/man-pages/. Cited on page 38

LLVM Clang (2015) LLVM Clang. clang: a C language family frontend for llvm, August 2015. URL: http://clang.llvm.org/. Cited on page 2, 16, 63

LLVM OpenMP (2015) LLVM OpenMP. OpenMP(r): Support for the OpenMP language, August 2015. URL: http://openmp.llvm.org/. Cited on page 2, 16

Lo *et al.* (2015) YuJung Lo, Samuel Williams, Brian Van Straalen, TerryJ. Ligocki, MatthewJ. Cordery, NicholasJ. Wright, MaryW. Hall and Leonid Oliker. Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis. Em Stephen A. Jarvis, Steven A. Wright and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, volume 8966 of *Lecture Notes in Computer Science*, páginas 129–148. Springer International Publishing. ISBN 978-3-319-17247-7. doi: 10.1007/978-3-319-17248-4_7. URL: http://dx.doi.org/10.1007/978-3-319-17248-4_7. Cited on page 12

Lorenzo *et al.* (2013) O G Lorenzo, T F Pena, J C Cabaleiro, J C Pichel and F F Rivera. DyRM: A Dynamic Roofline Model Based on Runtime Information. Em *2013 International Conference on Computational and Mathematical Methods in Science and Engineering*, páginas 965–967. ISBN 978-84-616-2723-3. URL: http://citius.usc.es/investigacion/publicacions/listado/490. Cited on page 12

Lorenzo *et al.* **(2014)** O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel and F. F. Rivera. 3DyRM: a dynamic roofline model including memory latency information. *The Journal of Supercomputing*, 70(2):696–708. ISSN 0920-8542, 1573-0484. doi: 10.1007/s11227-014-1163-4. URL: http://link.springer.com/10.1007/s11227-014-1163-4. Cited on page 12

Marr *et al.* **(2002)** Deborah T. Marr, David L. Binns, Frank adn Hill, Glenn Hinton, David A. Koufaty, J. Allan Miller and Michel Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):04–15. ISSN 1535766X. URL: http://www.ece.cmu.edu/~ece742/f12/lib/exe/fetch.php?media=marr_hyperthread02.pdf. Cited on page 49

Martinez *et al.* **(2011)** G. Martinez, M. Gardner and Wu chun Feng. CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures. Em *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, páginas 300 –307. doi: 10.1109/ICPADS.2011.48. Cited on page 2

Mikushin *et al.* **(2014)** D. Mikushin, N. Likhogrud, E.Z. Zhang and C. Bergstrom. KernelGen – The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs. Em *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, páginas 1011–1020. doi: 10.1109/IPDPSW.2014.115. Cited on page 2, 7, 8

Mikushin e Likhogrud (2012) Dmitry Mikushin and Nicolas Likhogrud. Kernelgen - a toolchain for automatic gpu-centric applications porting. URL: http://hgpu.org/?p=8313. Cited on page 2, 7, 8

Mikushin *et al.* **(2013)** Dmitry Mikushin, Nikolay Likhogrud and Eddy Zheng Zhang. KERNELGEN - the design and implementation of a next generation compiler platform for accelerating numerical models on GPUs. Technical Report, Rutgers. Cited on page 2, 7, 8

Mohr *et al.* **(2002)** Bernd Mohr, Allen D. Malony, Sameer Shende and Felix Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23(1): 105–128. ISSN 0920-8542, 1573-0484. doi: 10.1023/A:1015741304337. URL: http://dx.doi.org/10.1023/A:1015741304337. Cited on page 13

Mucci *et al.* **(1999)** Philip J. Mucci, Shirley Browne, Christine Deane and George Ho. PAPI: A Portable Interface to Hardware Performance Counters. Em *In Proceedings of the Department of Defense HPCMP Users Group Conference*, páginas 7–10. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.6801. Cited on page 12, 47

Newburn *et al.* **(2013)** Chris J. Newburn, Rajiv Deodhar, Serguei Dmitriev, Ravi Murty, Ravi Narayanaswamy, John Wiegert, Francisco Chinchilla and Russell McGuire. *Offload Compiler Runtime for the Intel(r) Xeon Phi(tm) Coprocessor*, páginas 239–254. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-38750-0. doi: 10.1007/978-3-642-38750-0_18. URL: http://dx.doi.org/10.1007/978-3-642-38750-0_18. Cited on page 1, 3

NVIDIA (2013) NVIDIA. NVIDIA CUDA C Programming Best Practices Guide. Technical Report, NVIDIA. Cited on page 3, 47

NVIDIA (2015a) NVIDIA. CUDA C Best Practices Guide. Technical Report, NVIDIA. URL: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide. Version 7.5. Cited on page 1, 7

NVIDIA (2014a) NVIDIA. CUDA C Best Practices Guide. Technical Report, NVIDIA. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf. Design Guide, DG-05603-001 Version 6.5. Cited on page 1, 7

NVIDIA (2014b) NVIDIA. CUPTI User's Guide (CUDA Performance Tools Interface). Technical Report, NVIDIA. URL: http://docs.nvidia.com/cuda/cupti. DA-05679-001 v6.0. Cited on page 12, 47

NVIDIA (2014c) NVIDIA. CUDA C Programming Guide. Technical Report PG-02829-001, NVIDIA. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Design Guide v6.5, posted at 2014-08-01 13:24:15. Cited on page 1

NVIDIA (2015b) NVIDIA. CUPTI User's Guide. Technical Report February, NVIDIA. URL: http://docs.nvidia.com/cuda/cupti. "DA-05679-001 v6.0". Cited on page 12

NVIDIA (2015c) NVIDIA. CUDA C Programming Guide. Technical Report PG-02829-001, NVIDIA. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Design Guide v7.5. Cited on page 1

NVIDIA (2014d) NVIDIA. cuBLAS-XT: Accelerate BLAS calls with multiple GPUs, May 2014d. URL: http://developer.nvidia.com/cublasxt. Cited on page 38

NVIDIA (2015d) NVIDIA. cuBLAS: NVIDIA CUDA Basic Linear Algebra Subroutines, November 2015d. URL: https://developer.nvidia.com/cublas. Cited on page 38, 66

NVIDIA (2009) NVIDIA. Whitepaper: Nvidia's next generation cuda(tm) compute architecture: Fermi(tm). Technical Report Version 1.1, NVIDIA. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Cited on page 1

NVIDIA (2012) NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Technical Report, NVIDIA Corporation. URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. Cited on page 1

NVPTX (2013) NVPTX. User Guide for NVPTX Back-end, July 2013. URL: http://llvm.org/docs/NVPTXUsage.html. Cited on page 8

Ofenbeck *et al.* (2014) Georg Ofenbeck, Ruedi Steinmann, Victoria Caparrós Cabezas, Daniele G. Spampinato and Markus Püschel. Applying the Roofline Model. Em *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) 2014, Monterey, CA, USA, March 23-25, 2014*, páginas 76–85. doi: 10.1109/ISPASS.2014.6844463. URL: http://dx.doi.org/10.1109/ISPASS.2014.6844463. Cited on page 10, 49

OpenACC (2011) OpenACC. OpenACC Application Programming Interface. Version 1.0., November 2011. URL: http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf. Cited on page 2, 7, 13, 16

OpenACC (2013) OpenACC. OpenACC Application Programming Interface. Version 2.0, June 2013. URL: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf. Cited on page 2, 7, 13, 16

OpenACC (2015a) OpenACC. OpenACC Application Programming Interface. Version 2.5, October 2015a. URL: http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf. Cited on page 2, 7, 13, 16

OpenACC (2012) OpenACC. OpenACC Directives for Accelerators Site, March 2012. URL: http://www.openacc-standard.org/. Cited on page 2, 16

OpenACC (2015b) OpenACC. OpenACC Directives for Accelerators, September 2015b. URL: http://www.openacc.org/. Cited on page 2, 16

OpenMP API Site (2012) OpenMP API Site. The OpenMP(r) API Specification for Parallel Programming, January 2012. URL: http://openmp.org. Cited on page 2, 16

OpenMP-ARB (2011) OpenMP-ARB. OpenMP Application Program Interface Version 3.1. Technical Report, OpenMP Architecture Review Board (ARB). URL: http://www.openmp.org/mp-documents/OpenMP3.1.pdf. Cited on page 2, 15, 16, 18

OpenMP-ARB (2013) OpenMP-ARB. OpenMP Application Program Interface Version 4.0. Technical Report, OpenMP Architecture Review Board (ARB). URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf. Cited on page 2, 7, 13, 15, 16, 18

OpenMP-ARB (2015) OpenMP-ARB. OpenMP Application Program Interface Version 4.5. Technical Report, OpenMP Architecture Review Board (ARB). URL: http://www.openmp.org/mp-documents/openmp-4.5.pdf. Version 4.5. Cited on page 2, 7, 13, 15, 16, 18

PAPI (2015) PAPI. Counting floating point operations on intel sandy bridge and ivy bridge, April 2015. URL: http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops. Cited on page 48

Par4All Site (2012) Par4All Site. Par4All Developer Guide, September 2012. URL: http://www.par4all.org/documentation/developers-guide/. Cited on page 2

Patterson (2009) David Patterson. The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges. *NVIDIA Whitepaper*, páginas 3–10. Cited on page 1

PGROUP (2015) The Portland Group PGROUP. PGI Accelerator Compilers with OpenACC Directives, December 2015. URL: http://www.pgroup.com/resources/accel.htm. Cited on page 2

PGROUP (2013) The Portland Group PGROUP. PGI(r) Compiler Reference Manual. Parallel Fortran, C and C++ for Scientists and Engineers, September 2013. URL: http://www.pgroup.com/resources/docs.htm. Cited on page 2

PGROUP (2010) The Portland Group PGROUP. PGI User's Guide, December 2010. URL: http://www.pgroup.com/resources/docs.htm. Release 11.0. Cited on page 2

Pouchet *et al.* (2012) Louis-Noël Pouchet, Uday Bondugula and Tomofumi Yuki. PolyBench/C the Polyhedral Benchmark suite, March 2012. URL: http://web.cse.ohio-state.edu/~pouchet/software/polybench/. Cited on page 55, 57

PyCUDA (2012) Site PyCUDA. PyCUDA Site, June 2012. URL: http://documen.tician.de/pycuda/. Cited on page 2

Reyes *et al.* (2012) Ruymán Reyes, Iván López-Rodríguez, Juan J. Fumero and Francisco de Sande. accULL: An OpenACC Implementation with CUDA and OpenCL Support. Em Christos Kaklamanis, Theodore Papatheodorou and Paul G. Spirakis, editors, *Proceedings of the 18th International Conference on Parallel Processing (Euro-Par 2012)*, volume 7484 of *Euro-Par'12*, páginas 871–882, Berlin, Heidelberg. Springer-Verlag. ISBN 978-3-642-32819-0. doi: 10.1007/978-3-642-32820-6_86. URL: http://dx.doi.org/10.1007/978-3-642-32820-6_86. Cited on page 2

Sukumaran-Rajam *et al.* (2014) Aravind Sukumaran-Rajam, Juan Manuel Martinez, Willy Wolff, Alexandra Jimborean and Philippe Clauss. Speculative Program Parallelization with Scalable and Decentralized Runtime Verification. Em Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734, páginas 124–139, Toronto, Canada. Springer. doi: 10.1007/978-3-319-11164-3_11. URL: https://hal.inria.fr/hal-01070610. Cited on page 7, 9

Top500 (2013) Top500. TOP500 List of Supercomputers Site, 2013. URL: http://www.top500.org/. Cited on page 1

Top500 (2014) Top500. TOP500 List of Supercomputers Site, 2014. URL: http://www.top500.org/. Cited on page 1

Top500 (2015) Top500. TOP500 List of Supercomputers Site, 2015. URL: http://www.top500.org/. Cited on page 1

Top500 (2016) Top500. TOP500 List of Supercomputers Site, 2016. URL: http://www.top500.org/. Cited on page 1

Trahay *et al.* (2011) François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst and Jack Dongarra. EZTrace: a generic framework for performance analysis. Em *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Newport Beach, CA, United States. URL: https://hal.inria.fr/inria-00587216. Poster Session. Cited on page 7, 13

Verdoolaege (2010) Sven Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. Em Komei Fukuda, Jorisvander Hoeven, Michael Joswig and Nobuki Takayama, editors, *Proceedings of the Third International Congress Conference on Mathematical Software - ICMS 2010*, volume 6327 of *ICMS'10*, páginas 299–302, Berlin, Heidelberg. Springer-Verlag. ISBN 3-642-15581-2, 978-3-642-15581-9. doi: 10.1007/978-3-642-15582-6_49. URL: http://dl.acm.org/citation.cfm?id=1888390.1888455. Cited on page 8

Verdoolaege e Grosser (2012) Sven Verdoolaege and Tobias Grosser. Polyhedral Extraction Tool. Em *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France. Cited on page 8

Verdoolaege *et al.* (2013) Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23. ISSN 1544-3566. doi: 10.1145/2400682.2400713. URL: http://dl.acm.org/citation.cfm?doid=2400682.2400713. Cited on page 2, 7, 8

Williams *et al.* (2009) Samuel Williams, Andrew Waterman and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65. ISSN 00010782. doi: 10.1145/1498765.1498785. URL: http://dl.acm.org/ft_gateway.cfm?id=1498785&type=html. Cited on page viii, 5, 7, 9, 10, 47

Wolfe (1996) Michael Wolfe. Parallelizing compilers. *ACM Comput. Surv.*, 28(1):261–262. ISSN 0360-0300. doi: 10.1145/234313.234417. URL: http://doi.acm.org/10.1145/234313.234417. Cited on page 4

Yan *et al.* (2009) Yonghong Yan, Max Grossman and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. Em *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, páginas 887–899, Berlin, Heidelberg. Springer-Verlag. ISBN 978-3-642-03868-6. doi: 10.1007/978-3-642-03869-3_82. URL: http://www.cs.rice.edu/~{}vs3/PDF/Yan-Grossman-Sarkar-Europar-2009.pdf. Cited on page 2