FEDERAL UNIVERSITY OF PELOTAS
Technology Development Center
Postgraduate Program in Computing

Thesis

A Model for Software Measurement Aiming to Guide Evaluations and Comparisons between Programming Tools to Implement GPU Applications

Daniel Di Domenico

Pelotas, 2022

Daniel Di Domenico

A Model for Software Measurement Aiming to Guide Evaluations and Comparisons between Programming Tools to Implement GPU Applications

Thesis presented to the Postgraduate Program in Computing at the Technology Development Center of the Federal University of Pelotas, as a partial requirement to achieve the PhD degree in Computer Science.

Advisor: Prof. Dr. Gerson Geraldo Homrich Cavalheiro
Coadvisor: Prof. Dr. João Vicente Ferreira Lima

Pelotas, 2022

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

D668m Domenico, Daniel Di

A model for software measurement aiming to guide evaluations and comparisons between programming tools to implement GPU applications / Daniel Di Domenico ; Gerson Geraldo Homrich Cavalheiro, orientador ; João Vicente Ferreira Lima, coorientador. — Pelotas, 2022.

103 f. : il.

Tese (Doutorado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2022.

1. GPU. 2. Programming tool. 3. Software measurement. 4. NPB. I. Cavalheiro, Gerson Geraldo Homrich, orient. II. Lima, João Vicente Ferreira, coorient. III. Título.

CDD : 005

Elaborada por Simone Godinho Maisonave CRB: 10/1733

Daniel Di Domenico

A Model for Software Measurement Aiming to Guide Evaluations and Comparisons between Programming Tools to Implement GPU Applications

Thesis approved, as a partial requirement, to achieve the PhD degree in Computer Science, Postgraduate Program in Computing, Technology Development Center, Federal University of Pelotas.

Defense Date: October 11th, 2022

Examining Committee:

Prof. Dr. Gerson Geraldo Homrich Cavalheiro (advisor)
PhD in IT Systems and Communications at Grenoble Institute of Technology, France.

Prof. Dr. André Rauber Du Bois.
PhD in Computer Science at Heriot-Watt University, Scotland.

Prof. Dr. Cristiana Barbosa Bentes
PhD in Systems Engineering and Computer Science at Federal University of Rio de Janeiro, Brazil.

Prof. Dr. Luiz Gustavo Leão Fernandes
PhD in Computer Science at Grenoble Institute of Technology, France.

To my wife Cristina and my parents who supported me during the way to develop this study.

ACKNOWLEDGEMENTS

I would like to thank, at the first place, the Lord, since He gives me a life with opportunities to achieve accomplishments like that.

I am profoundly grateful to my parents and sister for their love throughout my life. Thanks to them to encourage me pursuing my dreams.

I also would like to acknowledge my thesis advisers, Prof. Gerson Geraldo Homrich Cavalheiro and Prof. João Vicente Ferreira Lima, for their guidance and support throughout this work. Another acknowledgments for Prof. André Rauber Du Bois, Prof. Cristiana Barbosa Bentes and Prof. Luiz Gustavo Leão Fernandes, as the committee examiners of this doctorate defense. I am gratefully to them for their valuable comments on this Thesis.

Futher, I am thankful to the support received from FAPERGS and CNPq Brazil, program PRONEX 12/2014, by the project “GREEN-CLOUD”. The UFSM and IME/USP also contributed to this work offering platforms equipped with GPUs that were applied to execute experiments. Hence, I would like to thank this institutions for their assistance.

Last but not least, I am thankful for having my wife Cristina during all this time, sharing the good and bad moments, always motivating me to keep studying hard toward my goals. Your love and partnership is what makes me a better person.

It's your father's lightsaber. This is the weapon of a Jedi Knight. Not as clumsy or random as a blaster. An elegant weapon for a more civilized age.
— OBI-WAN KENOBI

ABSTRACT

DI DOMENICO, Daniel. **A Model for Software Measurement Aiming to Guide Evaluations and Comparisons between Programming Tools to Implement GPU Applications**. Advisor: Gerson Geraldo Homrich Cavalheiro. 2022. 103 f. Thesis (Doctorate in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2022.

Programming tools for GPUs are frameworks that offer resources to explore the massive parallelism power provided by these devices. Nowadays, they are being extensively applied for HPC purposes. Despite the existence of many tools that can be used to encode a GPU program, programming targeting GPUs is still seen as challenging, requiring the use of specialized frameworks to deal with the heterogeneous environment demanded by the GPU architecture. Also, there isn't a commonly accepted definition of a standard framework for GPU programming. So, the process to select a tool in order to implement a GPU program is not simple, especially when this choice can impact the performance and programming effort required to implement it.

Regarding that, this Thesis proposes a model to guide evaluations and comparisons between frameworks for GPUs. This model was designed based on the GQM method for software measurement and, because of that, was formulated using goals, questions and metrics to analyze three different aspects about the frameworks: programming expressiveness, programming effort and performance. As a result, the model seeks to offer a perspective including the characteristics, strengths and weaknesses about programming tools. We have in mind that such perspective can support the process of choosing a framework to develop a GPU program. Experiments guided by the proposed GQM model were conducted applying the NAS Parallel Benchmarks implemented with CUDA, OpenACC and Python/Numba. The experimental results contemplated the three aspects defined in the model, showing the similarities and differences regarding the tested APIs. Further, these results were employed to compose the perspective considering each framework. Due that, we believe this study contributes to improving the available knowledge about these tools, as well as advancing research on programming tools for GPUs.

Keywords: GPU. Programming tool. Software measurement. NPB.

RESUMO

DI DOMENICO, Daniel. **Um Modelo de Métrica de Software para Conduzir Avaliações e Comparações entre Ferramentas de Programação para Implementar Aplicações para GPU**. Orientador: Gerson Geraldo Homrich Cavalheiro. 2022. 103 f. Tese (Doutorado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2022.

Ferramentas de programação para GPUs são *frameworks* que possuem recursos para explorar o paralelismo massivo oferecido por tais dispositivos. Atualmente, estes dispositivos estão sendo amplamente empregados na Computação de Alto Desempenho. Apesar de existirem diversas ferramentas que podem ser utilizadas para codificar uma aplicação para GPU, o processo de programação requerido para esta finalidade ainda é visto como desafiador, demandando o uso de *frameworks* específicos para lidar com ambiente heterogêneo existente na arquitetura de GPU. Além disso, não há um *framework* amplamente aceito como o padrão para explorar GPUs. Neste sentido, a escolha de uma ferramenta para implementar um programa para GPUs não é simples, visto que tal decisão pode impactar o desempenho e o esforço de programação necessário para desenvolvê-lo.

Diante disto, esta Tese propõe um modelo para conduzir avaliações e comparações entre *frameworks* para GPUs. Este modelo foi formulado baseando-se na metodologia GQM para métrica de software e, devido a isto, utiliza *Goals* (Objetivos), *Questions* (Questões) e *Metrics* (Métricas) para analisar três aspectos referente aos *frameworks*: expressividade de programação, esforço de programação e desempenho. Como resultado, o modelo visa oferecer uma perspectiva considerando as características, pontos fortes e pontos fracos das ferramentas de programação. Nossa ideia é que tal perspectiva seja capaz de auxiliar na escolha de um *framework* a fim de desenvolver um programa para GPU. Experimentos conduzidos a partir do modelo GQM proposto foram realizados aplicando o conjunto de benchmarks do “NAS Parallel Benchmarks” implementados com CUDA, OpenACC e Python/Numba. Os resultados experimentais contemplaram os três aspectos definidos no modelo, mostrando as semelhanças e diferenças das APIs utilizadas. Ademais, os resultados foram empregados para construir a perspectiva sobre cada *framework*. Desta forma, nós acreditamos que este estudo contribui para aprimorar o conhecimento disponível sobre tais ferramentas, bem como no avanço das pesquisas relacionadas a ferramentas de programação para GPUs.

Palavras-chave: GPU. Ferramenta de programação. Métrica de software. NPB.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
C++ AMP	C++ Accelerated Massive Parallelism
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphical Processing Unit
GPGPU	General Purpose Graphical Processing Unit
GQM	Goal Question Metric
HPC	High Performance Computing
KS	Kolmogorov–Smirnov
LOC	Lines of Code
MPI	Message Passing Interface
NPB	NAS Parallel Benchmarks
NUMA	Non-Uniform Memory Access
OpenACC	Open Accelerators
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
STD	Standard Deviation
STL	C++ Standard Template Library

CONTENTS

1 INTRODUCTION

Heterogeneous computing systems, like multiprocessor processing servers equipped with GPUs, have been employed as a resource to achieve performance gains in different research fields (machine learning, bioinformatics, computational fluid dynamics and others) executing programs in parallel. GPUs, which are seen as the main heterogeneous device for High Performance Computing (HPC) (BRODTKORB; HAGEN; SÆTRA, 2013), offer a different platform in comparison with multiprocessors based on classic von Neumann architecture, providing an environment where applications can be performed on a massively-parallelized way. However, this contrasting architecture makes GPU programming challenging since it requires additional expertise to employ specialized programming tools to encode codes capable to explore them (LIMA et al., 2013; BALDINI; FINK; ALTMAN, 2014; ASTORGA et al., 2017).

Over the years, hardware vendors provided frameworks allowing programmers to implement codes targeting GPUs, such as CUDA (CUDA C++ PROGRAMMING GUIDE, 2021) and OpenCL (THE OPENCL SPECIFICATION, 2019). Advances were also proposed by these vendors intending to offer application programming interfaces (APIs) that are similar (OpenACC (OPENACC SPECIFICATION, 2020)) or equal (OpenMP 4.0 (OPENMP SPECIFICATION, 2020)) to the ones used to program multicore CPUs. OpenACC and OpenMP are based on compiling directives and allow the incremental parallelism, that is, a parallel application can be implemented extending its sequential version employing annotations in the code. This approach is a commercial success, contributing to make OpenMP the *de facto* standard API to multicore platforms. Besides the mentioned commercial GPU programming frameworks, some academic initiatives also have specified interfaces to heterogeneous environments providing different abstractions for GPU (e.g. implicit data handling and code portability). Some examples of them are Kokkos (EDWARDS et al., 2012), SkePU (ENMYREN; KESSLER, 2010), HPSM (LIMA; DI DOMENICO, 2017), StarPU (AUGONNET et al., 2011) and OmpSS (SAINZ et al., 2014).

Nowadays, GPUs don't just play a key role in HPC, but also have being applied for parallel processing in industries of different fields, as manufacture of equipment,

telecommunications and artificial intelligence innovations (ANDRADE et al., 2019). However, the programming tools focusing these devices still need improvements. A fact to prove this point is: there isn't a well accepted definition about the standard framework targeting GPUs. In comparison, parallel programming for cluster or multi-processor environments already have their consolidated tools (MPI and OpenMP, respectively). For GPUs, despite the many options available, none of them has a status like MPI or OpenMP.

Another aspect related to frameworks for GPUs is that most of them, including CUDA, OpenCL and OpenACC, were released supporting just compiled languages (such as C/C++ or Fortran) as third-party libraries (HOLM; BRODTKORB; SÆTRA, 2020). Although these languages are commonly known for their capacity to deliver high performance, they usually offer low-level APIs, a fact that can make them hard to be used. More recently, there were made available some options for GPU programming supporting high-level languages. One example is the Python language, which offers environments (like Numba (NUMBA DOCUMENTATION, 2022) and CuPy (CUPY API REFERENCE, 2022)) and packages (including PyCUDA and PyOpenCL (KLÖCKNER et al., 2012)) to enable GPU programming. Therefore, high-level programming languages can also be employed to develop GPU codes.

Considering the available frameworks for GPUs enabling distinct approaches of APIs for programming and the fact that there isn't a standard tool to implement GPU applications, an issue is faced: which framework is the most proper tool to encode some GPU program? Indeed, this question does not have a simple answer and was also reported as a problem by Astorga et al. (2017). Selecting a framework to develop an application to explore GPUs can impact its performance. Furthermore, each programming interface provides different concepts and resources to implement a source code. So, those aspects can also influence the required programming effort. Regardless of many factors that should be considered to make this choice, we suppose that evaluations and comparisons between programming tools could offer a perspective about them and, with that, assisting this process of decision.

There are in literature some studies that proceeded evaluations and comparisons between frameworks for GPUs. Most of them focused on executing these comparisons considering performance and programming effort aspects. However, these works proceeded analysis regarding programming tools just applying metrics as the approach to quantify, evaluate and perform comparisons between them. Hence, they did not follow a model of software measurement with predefined goals, rules and interpreting mechanisms in order to guide such process. Besides, these studies did not consider well-known benchmarks for HPC to execute the framework evaluations, since most of them performed such analysis using specific applications.

1.1 Objective

Taking into account the exposed concerning programming tools for GPUs, the **Objective** of this Thesis is to propose a model, based on the GQM (Goal, Question, Metric) method for software measurement, to evaluate and compare programming tools for GPUs. These comparisons aim to provide, considering programming expressiveness, programming effort and performance aspects, a perspective regarding the characteristics, strengths and weaknesses about each of the compared frameworks.

To accomplish the objective of this Thesis, we propose to reach the following **Specific Objectives** (**SO**):

SO1: Defining a model with goals, questions and metrics, based on the GQM method for software measurement, to guide the evaluation of programming tools for GPUs and, with that, comparing distinct aspects of such frameworks, as programming expressiveness, programming effort and performance.

SO2: Implementing versions of a set of GPU applications from a benchmark suite (NAS Parallel Benchmarks - NPB) employing different programming tools (CUDA, OpenACC and Python/Numba).

SO3: Proceeding programming expressiveness, programming effort and performance experiments guided by the proposed GQM model to collect information intending to evaluate and compare the frameworks for GPUs.

SO4: Presenting a perspective, based on the results of experiments guided by the proposed GQM model, regarding the characteristics, strengths and weaknesses about each of the evaluated frameworks for GPU programming (CUDA, OpenACC and Python/Numba).

1.2 Methodology

The methodology to develop this Thesis addresses the steps to achieve the specific objectives and, as a consequence of that, to accomplish the main objective of this study. Firstly, after a bibliography research, we defined which aspects about programming tools for GPUs would be evaluated applying our proposed model for software measurement. Most of the related studies employed programming effort and performance as goals for these analysis. As a complementary aspect, we included programming expressiveness since it can be applied to evaluate the resources of a framework by itself, that is, such evaluations do not need to be related to implemented applications.

After defining the aspects of evaluation, we were able to delineate the proposed model for software measurement based on the GQM method. The proposed GQM

model intends to describe how these defined aspects would be applied to evaluate and compare the programming tools for GPUs following the approach of goals, questions and metrics.

The set of GPU applications that were used during the experiments guided by the proposed GQM model was assigned at this stage. Previously to the definition of these programs, we were demanded to determine which frameworks would be employed to encode them. Among the available options, Andrade et al. (2019) highlighted CUDA as the most common choice from industrial companies as a programming tool to explore GPUs, since this framework has official technical support offered by Nvidia. Considering this fact and the massive application of Nvidia GPUs for HPC, CUDA was chosen as a tool to be used as a case study on this work. Hence, OpenCL was not selected once its API offers an approach similar to CUDA.

Having in mind to provide a contrasting point of view, OpenACC was chosen as a second tool to be compared to CUDA. The compiler directives approach employed by OpenACC aims to offer an easier programming environment than CUDA (which requires the specification of several computing details explicitly in the application code). Also, OpenACC is the most consolidated model based on compiler directives for GPUs and has official support from Nvidia. OpenMP was discarded by the same reason as OpenCL, since it offers a programming approach similar to OpenACC.

As a third framework to be compared to CUDA and OpenACC, we selected the Numba environment, a tool that enables GPU support in Python. With Numba, a GPU application can be encoded with pure Python code. The reason for including Python in this study is offering a counterpoint in relation to low-level compiled languages (like C/C++) which are required to make use of CUDA and OpenACC. Python is a high-level language commonly known to offer a great development productivity compared to complied ones, also being easy to learn and use (HOLM; BRODTKORB; SÆTRA, 2020). Besides, Python has been employed to develop scientific applications (ZIOGAS et al., 2021). Hence, the use of an API supporting pure Python to encode GPU programs can be advantageous since an application implemented with sequential Python may be ported to explore GPU devices without changing the programming language.

Regarding CUDA, OpenACC and Python/Numba as the programming tools of this work, we decided to employ the NPB programs in the experiments. NPB is composed of 5 kernels and 3 applications, totalizing 8 programs representing different domains of computation. NPB was applied to other works that focused on experiments regarding HPC aiming to analyze performance and programming effort aspects, including for GPUs. Considering that, there are implementations of NPB for GPUs with CUDA (ARAUJO et al., 2020), OpenCL (SEO; JO; LEE, 2011), and OpenACC (XU et al., 2015). This is another factor that contributed to the choice of NPB, since these third-party codes have also been confirmed and accepted by the HPC community. So,

we were required to implement the Python/Numba versions of the programs only.

The next stage of this work contemplated the execution of experiments guided by the proposed GQM model. They were performed using the programming tools (CUDA, OpenACC and Python/Numba) and the benchmarks developed employing them (NPB suite). The proposed model evaluates three aspects of a framework: programming expressiveness, programming effort and performance. Therefore, we achieved results for each of them. To finish this study, a final analysis regarding the experimental results was conducted. Such analysis intended to provide a perspective about each of the used programming tools considering their characteristics, strengths and weaknesses.

1.3 Scientific contributions

This Thesis makes the following **Scientific Contributions** (**SC**):

SC1: We provide a model for software measurement based on the GQM method that guides the comparison of programming tools to implement GPU programs employing different aspects of evaluation: programming expressiveness, programming effort and performance.

SC2: We make available the implementations of NPB five kernels and three applications with Python on serial and GPU (pure Python code applying Numba environment) versions. These codes are the first implementations of the NPB programs using Python.

SC3: We proceeded programming expressiveness experiments based on case studies regarding common GPU routines: Memory Allocation, Memory Transfer: Host to Device, Memory Transfer: Device to Host, Kernel Invocation and Using of Shared Memory. These experiments evaluated how the ideas can be represented in a code using a specific programming tool (CUDA, OpenACC and Python/Numba).

SC4: We performed code comparison using NPB programs intending to evaluate the programming effort required by each framework for GPU (CUDA, OpenACC and Python/Numba).

SC5: We executed performance experiments comparing GPU programs (NPB suite) developed with CUDA, OpenACC and Python/Numba, evaluating the differences in performance achieved by each framework.

SC6: Based on the experimental results considering programming expressiveness, programming effort and performance aspects, we presented a perspective about each of the evaluated programming tools for GPU (CUDA, OpenACC and Python/Numba) regarding their characteristics, strengths and weaknesses.

1.4 Meet briefly

When scanning this Thesis, the reader is embodied in a research addressing programming tools for GPUs. In the text, he or she is going to find information regarding GPU hardware, as well as details concerning programming tools which can be applied to explore this sort of device. After that, the reader will meet a contextualization about software measurement, focusing on topics like the GQM method, programming expressiveness and programming effort. These concepts were used to propose a GQM model aiming to guide evaluations and comparisons considering frameworks to explore GPUs. The goals, questions and metrics of such GQM model are detailed in order to specify how it was applied. The NPB suite is also illustrated by this study, presenting a description about each of its kernels and applications. In the sequence, our NPB implementations with Python and the Numba environment are highlighted, showing the advantages and limitations of applying Python targeting GPUs. The experiments are the next topic faced by the reader, when he or she will be aware of programming expressiveness, programming effort and performance results. These results were achieved guided by the proposed GQM model and employing the NPB benchmarks, leading us to a discussion where the evaluated frameworks (CUDA, OpenACC and Python/Numba) were analyzed. Finally, the reader faces a perspective about each of the programming tools for GPUs, being able to better understand their characteristics, strengths and weaknesses. Hence, this perspective can be useful to support the proper selection of a framework which suits more for some specific scenario or project. Further, the reader can apply the GQM model proposed by this study to evaluate other programming tools, also being possible, after understanding which resources such frameworks fail to provide, to propose improvements or new tool options for GPU programming.

1.5 Text organization

The remaining of this Thesis is organized as follows:

- Chapter 2 presents concepts about heterogeneous architectures in HPC, focusing on GPU devices and programming tools that can be applied to explore them.
- Chapter 3 addresses the related work considering other studies that proceeded comparisons between programming tools for GPUs.
- Chapter 4 introduces a context regarding comparisons between frameworks and applications, detailing programming aspects and the GQM method for software measurement. Next, it formalizes our proposed model to guide comparisons between programming tools for GPUs evaluating programming expressiveness, pro-

gramming effort and performance aspects.

- Chapter 5 describes the applications employed during the experiments of this Thesis, also detailing the implementations of NPB programs that we proceeded with Python.
- Chapter 6 presents the experimental results of this Thesis. They were executed guided by the proposed GQM model and applied the previously defined programs (NPB suite) and programming tools (CUDA, OpenACC and Python/Numba). After the evaluation of the proposed aspects about frameworks for GPUs, a discussion and final analysis details a perspective about each of them.
- Chapter 7 completes this work, presenting final remarks related to this Thesis and enumerating future works.

2 HETEROGENEOUS ARCHITECTURES AND GPUS

Parallel processing is a way that can be used to fulfill the increasing demand for more computational performance. This approach has been employed along the years through clusters executing the same job over many machines (such as supercomputers or mainframes). However, the parallel processing has gained more emphasis since 2004, when, due to problems related to heat dissipation, the new transistors available for CPU chips were used by hardware vendors to add more than one processing core in a single chip (ESMAEILZADEH et al., 2013). This breakthrough started a multicore era, considerably changing the way regular and mainly HPC applications are implemented, since they had to be adapted to explore the parallelism available on processors.

Despite the high performance that can be achieved by parallel applications running on multicore CPUs, the requirement for improvements in computing, especially in the HPC field, is constant. Regarding that, in the last decade new devices with different architectures have begun to be employed to execute parallel programs. These devices can be classified as heterogeneous architectures, and as faced by multicore ones, require parallel processing to deliver performance gains.

Heterogeneous architectures, also called accelerators in the HPC area, are a sort of hardware specifically developed to achieve high performance executing parallel codes (FENG; MANOCHA, 2007). So, these platforms are different from multicore CPUs which just replicate the cores for serial processing (FENG; MANOCHA, 2007). Embedded SoC (ARM Mali), Cell/BE, FPGAs boards and manycore architectures (GPUs and Intel Xeon Phi) are examples of devices that are considered accelerators (BRODTKORB; HAGEN; SÆTRA, 2013; SAINZ et al., 2014).

Nowadays, the trend topics for HPC are accelerators and heterogeneous computing (LIM; KIM, 2014; MENDONCA et al., 2016). For instance, twelve of the fifteen first places in 2022 Top 500 supercomputer list[1] already explore this sort of device. Hence, heterogeneous architectures play a key role in order to, not just improve executions from regular and HPC applications, but also in the path to build an exascale machine (MANN, 2020).

[1] TOP500 The List: https://www.top500.org/lists/top500/list/2022/06/

For HPC purposes, GPUs are seen as the main heterogeneous device (BRODTKORB; HAGEN; SÆTRA, 2013). Considering that and the focus of this work, the following sections of this Chapter are going to target GPUs, describing details of this kind of architecture and some programming tools that can be applied to explore them for parallel processing.

2.1 GPUs

GPU is an architecture originally employed for graphical processing that has evolved into a general purpose programmable accelerator (MANAVSKI, 2007; KINDRATENKO et al., 2009; BALDINI; FINK; ALTMAN, 2014). When a GPU is employed to a task different from graphical processing, it can be called GPGPU (General Purpose GPU). Today, most GPUs devices, including the ones equipping personal computers, are able to perform general purpose applications. However, there are GPUs designed exclusively to this goal that are applied by high investment environments totally focused on performance.

Regarding GPUs as a general purpose accelerator, it is possible to find a bunch of works published exploring their high computational power to improve performance. The main characteristic that make GPUs suitable for HPC is their efficiency for parallel processing using several threads (FENG; MANOCHA, 2007), additionally to a design that allows them to perform hundreds of billions of floating point operations per second on their large bandwidth on-board memory (MANAVSKI, 2007). In general, applications that have a large degree of data parallelism can properly explore the massive parallelism offered by GPUs (CARVALHO et al., 2020). Another advantage of applying GPUs for HPC is their better energy efficiency (performance-per-watt) compared to CPU devices (ABE et al., 2012; GOWDA; RAMAPRASAD, 2020). This can be seen as a benefit, since energy issues concentrate big concerns in computing nowadays.

To better understand the advantages related to parallel processing offered by GPUs, a comparison can be proceeded with CPUs. Basically, GPUs are different from CPUs in their purpose. A CPU intends to reduce the latency (execution time) on processing a single instruction, executing it as fast as possible. On the other hand, a GPU aims to raise the throughput of this processing, that is, it focuses on executing a large number of instructions at the same time. GPU cores commonly have a lower computational power comparing to CPU cores, resulting in a higher latency to perform just one operation. However, this latency can be overcome with the throughput (CUDA C++ PROGRAMMING GUIDE, 2021). According to this scenario, achieving a satisfying performance using GPUs requires applications able to make use of a great number of threads exploring the parallel processing (KIRK; HWU, 2013). Otherwise, executing the program on a CPU or multicore system will probably be faster.

In Figure 1, we can see models for CPU and GPU architectures. Each square/rectangle represents a processing unit, where their sizes are scaled accordingly to their computational power. GPU cores, despite having less power (smaller than CPU cores), are more numerous and can perform several instructions in parallel (massively parallelism). This is what makes it possible for a GPU to achieve a better performance than CPUs.

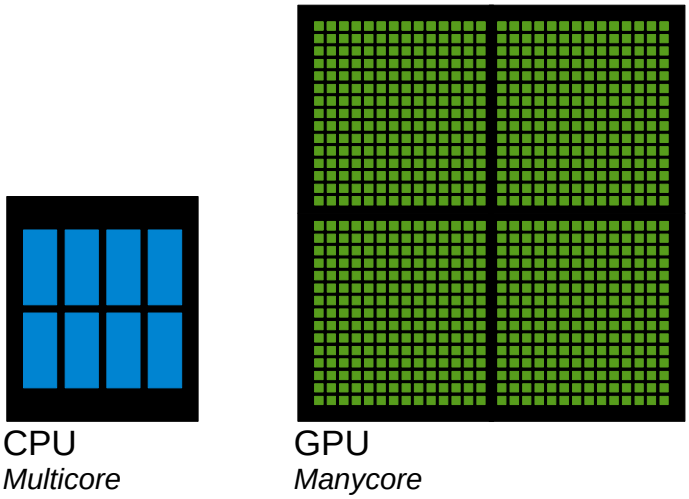

Figure 1 – CPU and GPU architecture models (CUDA C++ PROGRAMMING GUIDE, 2021).

As mentioned before, applications can take advantage of the massively parallelism provided by GPUs. For that, the program has to provide a lot of threads to be performed by the GPU. This great number of threads is necessary because, besides parallel processing, a GPU can also deliver performance hiding memory latencies with computation. Such fact is possible since a GPU can automatically switch a thread that is being executed and needs to fetch data from main GPU memory by another one. However, to completely hide the memory latency, enough available threads are required to be executed (BRODTKORB; HAGEN; SÆTRA, 2013).

As a result of this hiding of memory latencies with computation, a GPU core devotes more transistors to data processing, e.g., floating-point computations, which is beneficial for highly parallel computations. Hence, most parts of a GPU core is composed of an arithmetic logic unit. A CPU core, on the other hand, does not provide this advantage, since it relies on large data caches and complex flow control to avoid long memory access latencies (CUDA C++ PROGRAMMING GUIDE, 2021). Figure 2 depicts models showing the distinction between a CPU and a GPU core.

Currently, the main vendor of GPU devices for HPC purposes, as well as academic and industrial environments, is Nvidia. This company offers hardware and software (drivers and programming tools) allowing the implementation of general purpose parallel applications to run on their GPUs, providing a complete solution in order to explore

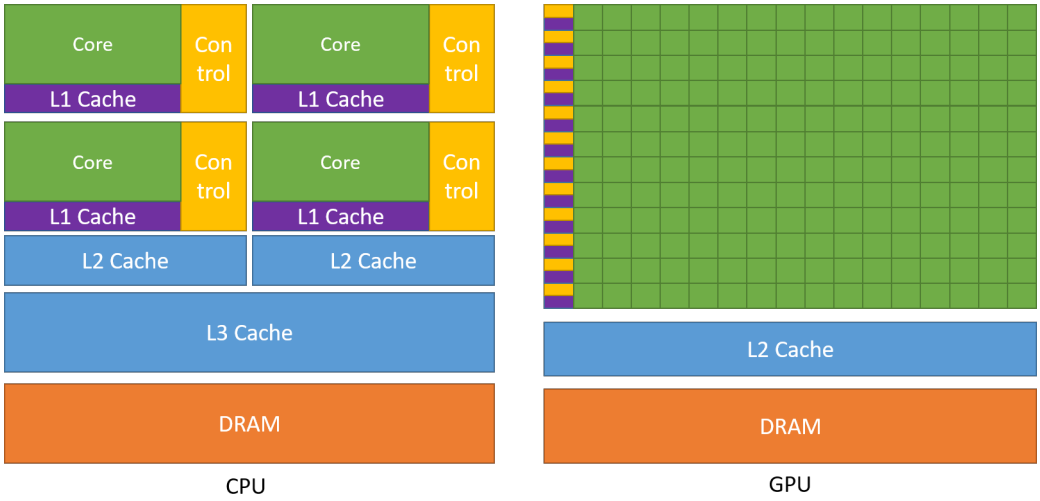

Figure 2 – CPU and GPU core models (CUDA C++ PROGRAMMING GUIDE, 2021).

their manufactured devices. For this reason, we chose to focus on Nvidia GPUs in the following of this work.

2.2 Programming tools to explore GPUs

In a computational system, GPUs role as coprocessors (being CPUs the main processing units), working with a private memory space apart from the main memory node. So, an application is required to manage memory transfers between the main memory and the GPU memory, once the GPU can only access data allocated in its own memory space. Furthermore, the architecture of a GPU is different from a CPU, usually requiring a specific version of code to be explored.

Programming for GPUs is based on an execution model where *kernels*[2] are launched to be executed on a GPU device. This means that just a few computationally demanding parts of the application code (the *kernels*) are performed on GPUs (CARVALHO et al., 2020). The remainder of the source is executed on CPUs, which are responsible for tasks like starting and finishing the program, managing inputs and outputs, as well as launching *kernels* to the GPUs.

Accordingly to their heterogeneous characteristics, GPUs require specific resources to be properly explored by software (LIMA et al., 2013). To achieve this demand, some frameworks were developed offering features to support them. This Section intends to describe some of these programming tools. We addressed the frameworks considering their importance in the HPC area. An aspect that illustrates this is that such tools are suggested and maintained by hardware vendors, a factor that contributes to their relevance, even though this fact limits their portability. Also, we

[2] A kernel is a GPU program that typically executes in a data-parallel fashion (BRODTKORB; HAGEN; SÆTRA, 2013)

regard the employment of such programming tools in the context of this study.

2.2.1 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and a programming model that aims to explore GPUs manufactured by Nvidia. This model allows to implement applications to CPU-GPU environments (SANDERS; KANDROT, 2010). CUDA is considered hard to be used, since several computing details must be defined explicitly by the programmer. These details, like memory allocations, memory transfers and configurations of GPU resources employed for processing, are written employing constructs available in the CUDA library. So, the development of a CUDA program can be a tough and tedious task (BUENO et al., 2013). Since CUDA was launched by Nvidia in 2007, new versions were made available. However, they didn't bring many improvements related to this aspect. Until this moment, the CUDA API offers practically the same features as when it was first released.

In a nutshell, a CUDA application is almost a C/C++ ordinary code. However, it has specific sections expressed to control the GPU processing and to be performed on a Nvidia GPU device. The code parts delineated to be performed on the GPU are called *kernels*. When processed, a *kernel* is launched as a task to be executed in parallel by a GPU according to parameters previously defined in the code (KIRK; HWU, 2013). Applications developed applying CUDA must be compiled employing a compiler that supports it. The official Nvidia compiler for this purpose is called NVCC.

2.2.2 OpenCL

OpenCL, short for Open Computing Language, is a language specification maintained by Khronos Group for heterogeneous platforms (independent of manufacturer) to implement parallel applications focusing on accelerators (like Nvidia, AMD ou ARM GPUs), multicore processors and other kind of devices (FGPAs, DSPs...) (WEBER et al., 2011). According to Kirk; Hwu (2013), OpenCL provides a programming interface similar to CUDA, but its resources have a greater programming complexity, since OpenCL offers portability to different architectures. The difficulty imposed by its API can be seen as a factor to explain why CUDA is more employed than OpenCL.

The OpenCL framework consists of three components that are used as a way to provide portability. First, the platform layer can be used to gather information about OpenCL-capable devices. Next, the runtime offers functions for managing device memory, running *kernels* (OpenCL parallel code written in C or C++) and transferring data to devices. Finally, the compiler maps abstract *kernels* onto a device-specific architecture to be executed. This compiling process is done during application's runtime by a specific compiler (like CLCC) (WEBER et al., 2011).

2.2.3 OpenMP

OpenMP (Open Multi-Processing) is considered the *de facto* standard to develop parallel algorithms intended to explore shared memory platforms, like multicore environments (ADCOCK et al., 2013). An OpenMP application is implemented through compiling directives called `pragmas`, requiring for that a compiler which supports its specification (DONGARRA et al., 2003). These directives denote to a compiler how the program can run sections of the application code in parallel.

One of the advantages of using OpenMP, which contributed to its commercial success, is the incremental parallelism, that is, a sequential program (written in C, C++ or Fortran language) can be extended, employing the `pragmas`, to be executed in a parallel way. This approach makes easier to apply OpenMP as a parallel model whereas another paradigms use the all-or-nothing conversion of an entire program (CHAPMAN; JOST; PAS, 2007).

OpenMP has evolved to keep up-to-date according to the new technologies. The OpenMP 4.0 has included support for accelerator devices. Some compilers already offer resources to explore GPUs using OpenMP, such as IBM XL (LI, 2017) and LLVM/Clang (CLANG 12 DOCUMENTATION, 2021).

2.2.4 OpenACC

Open Accelerators API, also called OpenACC, allows the implementation of applications focusing on exploring accelerator devices. Expressing parallelism with OpenACC is done employing `pragmas` (compiler directives) in the same way as OpenMP framework (OPENACC SPECIFICATION, 2020). One motivation that leads to the development of OpenACC is to simplify low-level parallel languages such as CUDA (LI et al., 2016).

According to Weber et al. (2011), OpenACC was the first initiative that proposes a standard based on compiler directives to explore accelerators. This fact can be seen as a change of paradigms, since it made possible to explore an accelerator architecture (like a GPU) just adding annotations on the sequential code. In an OpenACC program, ordinary tasks that must be handled in the accelerator programming, as data transfers between memory nodes, *kernel* implementation, parallelism mapping and work scheduling, are treated by OpenACC compiler and runtime (KIRK; HWU, 2013). Compared to CUDA, the programmer usually only has to express in the code which loops have to be accelerated and about the involved variables (KUAN et al., 2014). Thus, the barrier to program heterogeneous platforms was reduced, since, as previously mentioned, the programmer just has to determine which pieces of code can run in parallel.

2.2.5 Python and Numba

Launched in 1994, Python has achieved significant popularity in recent years. This high-level programming language can significantly increase development productivity compared to C/C++ and Fortran offering a powerful and easy-to-learn environment that focuses on readability of code and allows an efficient prototyping of it (HOLM; BRODTKORB; SÆTRA, 2020). Nowadays, Python is increasing its importance in scientific applications, being applied on many domains such as molecular dynamics, climate codes and machine learning (ZIOGAS et al., 2021). One of the reasons for that is the libraries providing a rich set of methods for scientific computing (ODEN, 2020). In addition, Python offers programming environments (like Numba (NUMBA DOCUMENTATION, 2022) and CuPy (CUPY API REFERENCE, 2022)) and packages (PyCUDA and PyOpenCL (KLÖCKNER et al., 2012)) to explore GPU devices. Hence, Python can also be considered as an option to develop applications targeting GPU platforms, also having an official suggestion[3] of Nvidia for that.

From the listed tools to explore GPUs with Python, Numba is the only one which supports implementing GPU code (including *kernel*) with pure Python. Unlike CuPy, PyCUDA and PyOpenCL, Numba does not require the use of native low-level CUDA/OpenCL calls specified in C/C++. Numba can be characterized as a just-in-time compiler and it is recommended for functions that use NumPy (NUMPY DOCUMENTATION, 2022) arrays and loops. A Python function annotated with a Numba decorator is compiled to machine code before execution. At compile time Numba reads Python bytecode, loads information about types of the input arguments and optimizes the function. Next, Numba uses the LLVM compiler library to generate machine code. This process allows code execution at native speed on every call (NUMBA DOCUMENTATION, 2022), both for CPU or GPU. Specifically for GPU code, the Numba compiling process generates PTX instructions to be processed by a CUDA-capable GPU device. Hence, Numba enables CUDA support for Python.

Numba for GPUs offers a programming model with almost all resources available in native C++ CUDA but dynamic parallelism and access to texture memory. A Numba application for GPU must define host and device codes, GPU *kernels*, as well as memory transfers. However, Numba also provides automatic data transfers between CPU and GPU for NumPy arrays. Additionally, several Python constructs, built-in types, built-in functions, standard library modules and NumPy arrays are supported and can be used to implement the GPU code (NUMBA DOCUMENTATION, 2022).

[3]GPU-Accelerated Computing with Python: https://developer.nvidia.com/how-to-cuda-python

2.2.6 Other tools

Regardless of the existence of the programming tools described in the last sections, there are other frameworks that can be used to explore GPUs. Most of them were developed by academic researches and commonly offer programming resources in order to be suitable alternatives to the “official” tools (the ones suggested and maintained by hardware vendors).

Some examples of these frameworks are StarPU (AUGONNET et al., 2011) and XKaapi (GAUTIER et al., 2013), which provide as main feature implicit data transfers between CPU and GPU memory nodes. These transfers are executed automatically according to a data dependency graph. Kokkos (EDWARDS et al., 2012) is another API that offers resources to data movement. Despite not offering implicit data transfers, Kokkos offers abstractions to assist the programmer with this assignment.

Additionally, Kokkos provides code portability and parallel skeletons/patterns, resources that are also available with HPSM (LIMA; DI DOMENICO, 2017) and SkePU (ENMYREN; KESSLER, 2010). The first feature indicates that the same version of a parallel code can perform on both CPU and GPU devices. This may be an advantage when the application targets both platforms or heterogeneous processing (CPU+GPU). Parallel skeletons and patterns offer to programmers an easier way to implement parallel code, a resource that is also enabled by APIs based on compiler directives (like OpenMP and OpenACC).

Regarding these previous described tools, the parallelism for a GPU program must be delineated using a library. This is the same approach applied by CUDA, OpenCL and Python/Numba. Another APIs that offer libraries for GPU programming are C++ AMP (Accelerated Massive Parallelism) (WYNTERS, 2016) and Thrust (THRUST, 2021), both providing STL-like algorithms and containers of C++ for data parallelism on GPU. On the other hand, some tools offer other approaches to describe the parallelism. PARRAY (CHEN; CUI; MEI, 2012) and PACXX (HAIDL; GORLATCH, 2014) use annotations for compiler transformations. Similar as OpenMP and OpenACC, OmpSS (SAINZ et al., 2014) and OMPi (NOAJE; JAILLET; KRAJECKI, 2011) employ compiler directives as an abstraction to model GPU parallelism.

This study will not apply any of the tools described by this Section in the experiments. Despite that, as we are proposing a model to evaluate and compare programming tools for GPUs, future evaluations can be proceeded applying them in order to analyze their characteristics related to expressiveness, programming effort and performance.

2.3 Chapter overview

This Chapter presented characteristics and purposes about heterogeneous architectures in computing, also called accelerators in the HPC area. The focus was given to GPUs, once it is the main heterogeneous device employed for HPC purposes. Hence, the GPU hardware was described, pointing differences between a CPU and a GPU. By the end, some programming tools to explore GPUs were characterized, highlighting the frameworks suggested and maintained by hardware vendors.

3 RELATED WORK

Some research has been conducted to evaluate and compare different frameworks to explore GPU devices. These works have different approaches, since they can focus on the developer's sight related to some API (how hard/easy to be used or productive it can be) or they can analyze the potential of some tool to deliver performance. The following sections of this Chapter present studies that previously proceeded comparisons between programming tools for GPUs. We limited as scope articles which applied CUDA, OpenACC or Python for GPUs, once these are the tools we employed as case studies of this work.

3.1 Studies regarding comparisons of GPU programs implemented with CUDA, OpenACC and Python

The current Section intends to describe the works where comparisons between CUDA, OpenACC and Python implementations for GPUs were performed. The focus is to illustrate the different aspects employed to proceed such comparisons, like performance, energy consumption and programming effort.

RS1 *- Accelerating Phylogenetic Inference on GPUs: an OpenACC and CUDA comparison (KUAN et al., 2014)*

This manuscript presents comparisons between OpenACC and CUDA implementations of MrBayes application to execute a phylogenetic inference.[1] This kind of programs have a high computational burden, so GPU processing has been applied to overcome this issue. After proceeded the OpenACC (version 1.0 with PGI 12.6 compiler) and CUDA (version 4.2) implementations, performance comparisons were executed on a GPU device (Nvidia GTX 580) using simulated DNA data sets of various sizes. The speedups obtained by OpenACC and CUDA versions over the sequential one have fluctuated according to the inputs, ranging from 2.5 to 9 times. However,

[1] Phylogenetic inference employs DNA sequences to reconstruct the evolutionary history of species.

CUDA implementations achieved better performances for all scenarios.

The authors proceeded a second analysis related to the employing of OpenACC as a framework targeting GPUs. They recommended the use of OpenACC, since it offers a great potential to tune applications without the huge programming effort required by CUDA and without adding a high penalty on performance (only 10% of slowdown for the implemented application). Despite that, the study does not present a detailed evaluation quantifying how less programming effort is required by OpenACC.

RS2 *- A comparison of CUDA and OpenACC: Accelerating the Tsunami Simulation EasyWave (CHRISTGAU et al., 2014)*

This paper presents results for the tsunami simulation EasyWave executed on GPUs. The authors performed experiments with this application on two Nvidia GPUS (Tesla C1060 and Tesla C2075), employing versions developed with CUDA 5.0 (with optimizations regarding the GPU hardware) and OpenACC (using PGI 13.6 compiler). Considering the implementation process, an evaluation was proceeded counting the lines added to the sequential version of the program, which originally has 462 lines. The OpenACC version required 21 additional lines, while the CUDA one, where optimizations to deal with hardware details of each GPU were implemented, demanded 248 additional lines.

Regarding performance, the optimized CUDA application performed better on both GPU devices employed by the simulations (just one input size was applied), achieving speedups of 10.7 times over sequential version. The OpenACC program had a disappointed result, with a maximum speedup of only 2.67 times. The authors also gauged data from different parts of the application, figuring out that for some routines, the OpenACC application can perform as fast as the CUDA one. After a deeper analysis, they assumed that this difference can be a consequence of compiler issues, once the structure of the functions is very similar and only differs in the memory access pattern.

RS3 *- An Early Performance Comparison of CUDA and OpenACC (LI; SHIH, 2018)*

This article presents a comparison between CUDA and OpenACC applications focusing just on performance aspects. All experimental results were obtained from executions performed on a Nvidia Tesla K40c, employing CUDA 7.0 and the PGI 15.4 compiler for OpenACC. The applications applied for the tests (10 in total) were selected from the Rodinia suite (8 applications), including also a Jacobi benchmark made available by Nvidia and a Matrix Multiplication algorithm developed by the authors. As input sizes, they employed different values to evaluate the performance impact of changes in data size.

The results for the selected algorithms showed that OpenACC performance is sim-

ilar to CUDA. However, when different input sizes were applied for each application, OpenACC versions registered more variations for processing time than CUDA optimized versions. Accordingly to the authors, this means that OpenACC programs are more sensitive to changes in data size than the equivalent CUDA programs with optimizations. By the end, they concluded that OpenACC is a reliable programming model and a good alternative to CUDA for accelerator devices.

RS4 *- Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption (MEMETI et al., 2017)*

This study proceeds a detailed evaluation of frameworks targeting accelerators, including programming productivity, performance, and energy consumption aspects. One interesting point in this work is the proposed metric to analyze the programming productivity. It applied a parallelization effort formula to evaluate the programming tools (OpenCL, OpenACC, OpenMP, and CUDA) regarding lines of code written during the implementation of an application.

As parallel applications for the experiments, the work employed algorithms extracted from SPEC Accel (OpenACC and OpenCL implementations) and Rodinia (OpenMP, OpenCL and CUDA implementations) benchmark suites. Regardless of the presentation of results that led to some observations about the programming models, no direct comparison was proceeded between CUDA and OpenACC applications, since the benchmark suites employed do not offer both versions of the same program.

As presented results, the study claimed that OpenACC and CUDA required less programming effort than OpenCL (about 6.7 and 2.0 times less on the average, respectively). The same scenario was presented to OpenMP, which demanded on the average 3.6 and 3.1 times less programming effort than OpenCL and CUDA, respectively. Considering performance and energy consumption behavior, the results varied according to the application. However, for most of them, OpenCL and CUDA achieved the best results (better performance and energy efficiency) than OpenACC and OpenMP.

RS5 *- Comparing Programmer Productivity in OpenACC and CUDA: an Empirical Investigation (LI et al., 2016)*

This paper presents an investigation of programming productivity comparing the time required to develop GPU applications using OpenACC and CUDA. The evaluation was based on human subjects, since it measured the time spent for undergraduate-level and graduate-level students to implement two programs applying both frameworks.

As results, the authors presented three conclusions. First, the study showed that OpenACC enables users to finish a parallel solution in a shorter time than CUDA (at

least 37% faster). Second, despite the programming productivity of OpenACC model was better than CUDA one, CUDA applications achieved, on average, an execution time 9 times shorter than OpenACC versions. By the end, the authors also noted that previous CUDA experience did not affect the amount of work required to implement an application employing OpenACC.

RS6 *- CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application (HOSHINO et al., 2013)*

This manuscript intends to compare the performance of two microbenchmarks (matrix multiplication and 7-point stencil) and one real-world computational fluid dynamics (CFD) application implemented with OpenACC and CUDA running on a GPU device (Nvidia M2050). The compilations were proceeded with PGI CUDA Fortran 12.10 for CUDA programs. For OpenACC, three different compilers (PGI 12.10, HMPP 3.2.4, and Cray 8.1.0.165) with CUDA 4.1 were applied, except for the CFD application, where just the PGI tool was used.

The experimental scenarios indicated that OpenACC applications achieved approximately 50% of performance from CUDA versions. The reason pointed by the authors for such difference was the memory access optimizations proceeded in CUDA codes. However, the performance can fluctuate according to the selected compiler. Depending on the chosen tool, the OpenACC version can reach a performance up to 98% from CUDA one. The authors also presented some numbers related to the lines of code added for CUDA and OpenACC versions over the base implementations of each application. The number of lines included in CUDA codes were always greater than the ones added for OpenACC codes, but it was not proceeded an analysis to quantify or to interpret this information.

RS7 *- Evaluating the Performance and Cost of Accelerating Seismic Processing with CUDA, OpenCL, OpenACC, and OpenMP (GIMENES; PISANI; BORIN, 2018)*

This paper presents an evaluation of performance and cost-benefit (in terms of hardware performance and energy efficiency) using two seismic processing methods (Common Midpoint and Common Reflection Surface). For that, the authors implemented parallel applications with both methods employing CUDA, OpenCL, OpenACC, and OpenMP frameworks aiming to explore CPUs and GPUs. Regarding GPUs, 6 Nvidia devices (K40, K80, M40, M60, P100, P102) were applied to the experiments, with CUDA 8.0.61 and PGI compiler as software tools. The authors also used two different data sets as inputs for the seismic simulations.

Before the experiments intending to evaluate performance, it was depicted some information about the number of directives and API calls required for each implemen-

tation. This aimed to expose an indication of programming effort demanded by each programming tool. According to the displayed data, OpenCL and CUDA implementations presented numbers about 5.7 and 4.7 times greater than the ones exposed by OpenACC, respectively. Regarding performance outcomes, CUDA and OpenCL achieved, in general, similar results. OpenACC underperformed OpenCL and CUDA, reaching an average slowdown of 9%, with the maximum being 28%. As a conclusion, the authors pointed OpenACC as an interesting approach for the tested seismic processing methods, since it provides a friendlier API to create GPU-accelerated applications with less programming effort, as well as a small decrease in performance when compared to the other two approaches (OpenCL and CUDA).

RS8 *- GPU Computing with Python: Performance, Energy Efficiency and Usability (HOLM; BRODTKORB; SÆTRA, 2020)*

This work presented an evaluation considering GPU programs in C++, with CUDA and OpenCL toolkits, and in Python, with PyCUDA and PyOpenCL packages. The authors, who have extensive experience working with C++ and Python for GPUs, implemented three benchmarks employing these tools to compare performance, energy efficiency and usability. Also, they executed experiments on seven different models of Nvidia GPUs.

Regarding performance, the results showed that Python versions are as efficient as C++ ones in many cases, as well as the energy efficiency is proportionally related to the improvement of performance. Through the counting of lines of code of a single program (Mandelbrot set), it was shown that Python implementations are smaller than C++ ones for both OpenCL and CUDA. Hence, the authors subjectively, as themselves admit, classified the development time with Python as “faster” and with C++ as “medium”. Nevertheless, the size reduction with Python is achieved on serial (host) code, once PyCUDA/PyOpenCL parallel code must be written in native low-level CUDA/OpenCL. By this reason, the performance results are similar using both tools, since the most computing processing is achieved by the GPU and the host code does not significantly affect the final performance. To conclude, the authors claimed that using Python can be preferable to C++ and that using CUDA can be preferable to using OpenCL.

RS9 *- Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing (ODEN, 2020)*

This paper presented a performance comparison of Numba-CUDA (Python) and C-CUDA for different kinds of programs, such as micro benchmarks and a mini application. The author sought to understand the differences between C-CUDA and Numba-CUDA in terms of performance. Once Numba allows the implementation of GPU code

with pure Python, the paper also aims to provide tips to improve applications written with this framework.

The experimental results showed a performance penalty when applying Numba to explore a GPU. Numba versions achieved 75% to 86% of the performance reached by CUDA. The main reason for that is the reduction of GPU utilization with Numba. This happens because some parts of the application, usually serial codes, are executed by the Python interpreter. Hence, these non-GPUs parts, which were not accelerated, became a bottleneck and negatively affected performance.

RS10 *- Parallel Computation of Aerial Target Reflection of Background Infrared Radiation: Performance Comparison of OpenMP, OpenACC, and CUDA Implementations (GUO et al., 2016)*

This study focuses on a computation of the Infrared Radiation signature of an aerial target. The authors developed these parallel algorithms with OpenMP (for CPUs) and with OpenACC and CUDA (for GPUs). Considering GPU implementations, it was applied CUDA 6.0 and PGI 15.5 compiler (for OpenACC). The experiments were executed on an Nvidia Tesla K20c GPU.

Results regarding GPUs presented a speedup of 140 and 426 times over the sequential version for OpenACC and CUDA, respectively. So, the speedup obtained in the OpenACC implementation was 33% of that in CUDA. The authors explained this difference through a more effective use of GPU memory by the CUDA version. They also claimed that, from a point of view considering the number of rewriting lines, OpenACC improves the programmer's productivity, since its directives are defined without changing the original sequential code. On the other hand, the CUDA approach required the rewriting of several lines during the implementation, most of them related to the definition of kernel functions, shared memory usage, device memory allocation, data transfers between host and device, and necessary synchronization. So, OpenACC can be seen as an option to make GPU programming easier and to obtain a reasonable speedup.

RS11 *- Productivity of GPUs under different programming paradigms (MALIK et al., 2012)*

In this work the authors encoded multiple versions of four NPB kernels (CG, EP, FT and MG) with CUDA, OpenCL, PGI Accelerator (precursor of OpenACC) and MATLAB. Despite no details about the implementations were given, the study evaluated performance and ease of use intending to measure the impact of high-level frameworks (PGI Accelerator and MATLAB) compared to CUDA and OpenCL.

Performance results (using classes S, W, A and B of NPB) pointed CUDA and

OpenCL with the best speedups over sequential versions. On the other hand, PGI Accelerator and MATLAB achieved poor and similar speedups for most cases. Besides that, the paper also presented a detailed programming effort analysis comparing the source codes implemented using these four tools. The proceeded analysis showed several perspectives about the code developed with each programming tool, evaluating manual effort counting lines of codes and conceptual programming effort counting specific key terms of the parallel programming model. The authors concluded that parallelism-centric abstractions, like OpenACC and MATLAB, can provide better productivity than library approaches closer to the GPU architecture, like CUDA and OpenCL. However, this advantage usually sacrifices performance.

3.2 Overview about the related studies

After presenting the studies regarding evaluations and comparisons between CUDA, OpenACC and Python implementations for GPUs, this Section intends to provide an overview about the resources and metrics applied by each article to make such differentiation. This overview considers performance and programming effort as subjects for the executed comparisons since these are two of the aspects evaluated by our work. Programming expressiveness was not included once none study proceeded such analysis. Table 1 depicts some summarized information about each article described in the previous Section (3.1).

Table 1 shows that 10 from 11 of the described studies focus on performance evaluations considering applications implemented with at least two of the three tools defined as scope (CUDA, OpenACC and Python). The only article that does not contemplate this question is **RS4**. Actually, the whole paper presents evaluations using CUDA and OpenACC programs over performance and programming productivity subjects, although it did not proceed with a straight comparison employing these two frameworks. For most of the executed experiments detailed by the papers, CUDA implementations outperform OpenACC ones. The main factor pointed for this behavior are the optimizations that can be developed employing CUDA. It seems that improvements regarding memory access on the GPU device are a key factor that leads CUDA applications for better results. However, for some specific scenarios, like executing just parts of a program (**RS2**), changing the compiler (**RS6**) or according to the application (**RS3**), OpenACC implementations achieved performances similar to CUDA versions.

The performance achieved by Python for GPUs is similar to CUDA and OpenCL, as pointed out by studies **RS8** and **RS9**. Python, as an interpreted language, has to apply compiling in order to accelerate the code for GPUs. This process generates programs which are executed using CUDA or OpenCL runtime libraries. Hence, this compiling is suggested by both papers as the cause for the similar performance between Python

Study			Comparisons between CUDA, OpenACC and Python		
ID	**Title**	**Reference**	**Performance**	**Programming effort**	**Resource/metric to evaluate programming effort**
RS1	Accelerating Phylogenetic Inference on GPUs: an OpenACC and CUDA comparison	(KUAN et al., 2014)	✓		Obs: claims that OpenACC requires less programming effort than CUDA, even without showing any information about it
RS2	A comparison of CUDA and OpenACC: Accelerating the Tsunami Simulation EasyWave	(CHRISTGAU et al., 2014)	✓	✓	Counting lines of code added to the sequential version of a program
RS3	An Early Performance Comparison of CUDA and OpenACC	(LI; SHIH, 2018)	✓		
RS4	Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption	(MEMETI et al., 2017)			Obs.: despite of evaluating programming productivity applying a programming effort formula, none comparison regarding CUDA, OpenACC or Python was proceeded
RS5	Comparing Programmer Productivity in OpenACC and CUDA: an Empirical Investigation	(LI et al., 2016)	✓	✓	Evaluation based on human subjects, comparing the time spent for students to implement two programs applying both frameworks
RS6	CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application	(HOSHINO et al., 2013)	✓	✓	Counting lines of code added to the base version of a program
RS7	Evaluating the Performance and Cost of Accelerating Seismic Processing with CUDA, OpenCL, OpenACC, and OpenMP	(GIMENES; PISANI; BORIN, 2018)	✓	✓	Counting directives and API calls, showing the difference between required lines for OpenACC and CUDA
RS8	GPU Computing with Python: Performance, Energy Efficiency and Usability	(HOLM; BRODTKORB; SÆTRA, 2020)	✓	✓	Counting all lines of code of a program
RS9	Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing	(ODEN, 2020)	✓		
RS10	Parallel Computation of Aerial Target Reflection of Background Infrared Radiation: Performance Comparison of OpenMP, OpenACC, and CUDA Implementations	(GUO et al., 2016)	✓		Obs: claims that OpenACC requires less programming effort than CUDA since its directives are defined without changing the original sequential code
RS11	Productivity of GPUs under different programming paradigms	(MALIK et al., 2012)	✓	✓	Evaluations based on manual programming effort, where all lines of code were counted, and conceptual programming effort, where just specific terms of the framework were counted

Table 1 – Overview from studies comparing GPU applications implemented with CUDA, OpenACC and Python.

and CUDA/OpenCL. Despite that, **RS9** detected some performance slowdowns when applying Python with Numba. The reduction of GPU utilization is the reason for such fact and it happens because the non-GPU parts of the code are executed by the Python interpreter.

Considering programming effort, 5 of the 11 studies performed an analysis proposing to evaluate the differences between CUDA and OpenACC frameworks. Studies **RS2**, **RS6**, **RS7** and **RS11** used a metric where lines of code (LOC) were counted to

claim that OpenACC can be used in a simpler way to implement an application targeting GPUs, since it requires fewer lines than CUDA. An approach based on human subjects was employed by **RS5**, where the authors concluded that OpenACC enables users to finish a parallel solution in a shorter time than CUDA. To achieve this conclusion, they evaluated the time spent for students to implement two programs applying both frameworks.

Analyzing the differences between OpenACC and CUDA implementations, two well-known factors were confirmed. First, OpenACC framework requires less programming effort than CUDA. Second, CUDA can deliver, in general, better performance than OpenACC. Nevertheless, it seems that this second fact is not true for all scenarios, since for some specific experiments, OpenACC applications performed almost as good as CUDA ones.

Regarding Python, study **RS8** also applied LOC counting to claim that Python's implementations are smaller than CUDA ones. This size reduction is achieved regarding the host code of the application, since the GPU code (kernel) was implemented with native C++ CUDA in such paper. Further, this study recommended Python instead of C++ for GPUs because it offers a simpler and high-level interface which does not penalize performance for most cases.

As a conclusion of this overview about the presented studies, we can firstly point out that most of the works executed experiments with specific applications. Just **RS3**, **RS4** (despite no comparison between CUDA, OpenACC or Python was proceeded) and **RS11** applied well-known benchmarks to evaluate the programming tools. Second, the evaluations and comparisons between frameworks and applications were executed using just metrics, that is, they were not guided by a model or a method with predefined goals, rules and interpreting directives. Third, works **RS4** and **RS11** proposed to evaluate programming productivity based on counting lines of code, a metric that is not proper to proceed with such analysis since, in our view, it must involve more variables. Fourth, we haven't found any study comparing Python for GPUs with OpenACC, a comparison that seems relevant because both tools offer higher-level approaches of programming than CUDA. Besides, the studies did not display data to attest the reliability of their performance results employing statistical methods. By the end, **RS5** presented a research considering human subjects (programmers) to evaluate the productivity of OpenACC and CUDA. Although this can be an interesting approach, it is difficult to be executed and can be influenced by external conditions that can affect the results, like the participants' previous programming experience and their effective concern to collaborate with experimentation.

3.3 Chapter overview

This Chapter described related works which previously executed comparisons between CUDA, OpenACC and Python targeting GPU devices. The focus was to detail the proceeded comparisons, analyzing the metrics and resources employed to evaluate performance and programming effort, once programming expressiveness was not regarded by any of the manuscripts. To finalize, it was presented an overview comparing these related studies, concluding with an analysis concerning the executed evaluations.

4 A MODEL TO COMPARE PROGRAMMING TOOLS TARGETING GPUS BASED ON GQM METHOD

In order to proceed comparisons between programming tools, a model is required to guide such process. The purpose of this Chapter is to present the model we are proposing to accomplish this guidance. Based on the GQM method for software measurement (detailed by Section 4.1), this model aims to provide resources to guide the comparison of frameworks targeting GPUs. Before its description, this Chapter introduces a background related to programming expressiveness and programming effort (Section 4.2). Both concepts were employed to define the model. After that, the GQM model itself is expounded, detailing their goals, questions and metrics. By the end, a discussion describes how the model will be applied and which questions about programming tools for GPU devices it is expected to answer.

4.1 GQM method for software measurement

There are in literature several methods that can be applied to execute software measurements, a task which contemplates the means to quantify characteristics of software products or software processes. Software measurement allows to determine the software strengths and weaknesses, to provide a rationale for adopting/refining techniques (e.g., *What is the impact of the tool X on the effort?*), or even to evaluate software quality (BASILI; CALDIERA; ROMBACH, 1994; ERGASHEVA; KRUGLOV; SHULHAN, 2019). Software measurement is usually employed in the software engineering area. However, HPC also can make use of this kind of process, for instance, to compare different parallel programming tools.

The GQM (short for Goal Question Metric) is a method for software measurement based on a paradigm proposed by Basili and Weiss first presented in 1984. It was originally applied for evaluating defects for a set of projects in the NASA Goddard Space Flight Center environment. In a nutshell, GQM is a mechanism for defining and interpreting operational and measurable software. For that, the GQM paradigm defines and evaluates operational goals through the use of measurement. GQM also

represents a systematic approach for tailoring and integrating goals with models to evaluate software processes, products and quality perspectives of interest (BASILI, 1992).

Commonly, studies make use of software metrics to evaluate applications. Depending on the reason for these evaluations, metrics like number of failures, lines of code, number of defects and total effort required for implementation can be applied. The GQM method employs such metrics surrounded with the appropriate models and goals in order to make clear which metrics to use, as well as how to properly interpret them (BASILI, 1992). Hence, GQM focuses on eliciting goals (and also questions) as a way to define which metrics are necessary to be collected. This means that each defined measurement has always a related purpose (BERANDER; JÖNSSON, 2006). Due to this approach, the results achieved applying the GQM method are usually clear and solid since the measurements are rooted from the goal and objectives (SETIAWAN; RASJID; EFFENDI, 2018).

The process for using the GQM approach follows the application of a hierarchical structure composed by goals, questions and metrics. Figure 3 depicts the structure of a GMQ model.

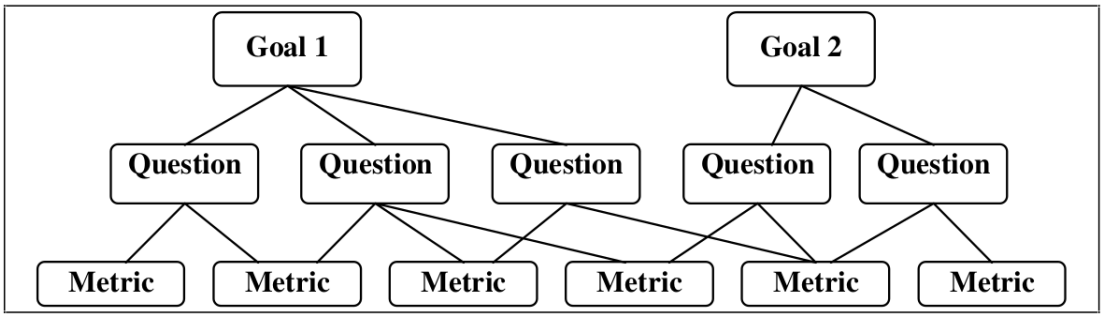

Figure 3 – GQM model hierarchical structure (BASILI; CALDIERA; ROMBACH, 1994).

A GQM model is based on goals, where the purpose of measurement, object and issue to be measured, as well as a viewpoint describing why a measure is taken must be specified. Each goal is refined into several questions to divide the issue into its major components. A question, in its turn, is refined into metrics. The metrics can be objective or subjective and are used in order to provide answers to the questions. By answering the questions, the measured data can be analyzed to check if the goal were reached. Questions can be applied by different goals and metrics can be shared by distinct questions (BASILI; CALDIERA; ROMBACH, 1994). By using GQM, metrics are defined from a top-down perspective and analyzed and interpreted bottom-up (SOLINGEN; BERGHOUT, 2001).

According to Basili; Caldiera; Rombach (1994), the result of applying the GQM method is the specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data. The specific

results of each level that composes the GQM model are detailed by the following topics:

- **Goal:** Conceptual level. A goal is defined for an object of measurement (product, process or resource), for a variety of reasons, with respect to various models of quality, from various points of view, relative to a particular environment. Also, a goal has to specify the objective of each measure;
- **Question:** Operational level. A questions characterizes how the goal(s) should be attained/achieved and the way it is going to be performed;
- **Metric:** Quantitative level. A metric represents the set of data associated with a question in order to answer it in a quantitative way.

Along the years, some GQM models have been proposed offering resources to guide evaluations about software. For instance, Setiawan; Rasjid; Effendi (2018) proposed a model to generate indicators in order to improve quality on software development. Ramadan; Abd ellatif (2010) applied the GQM approach to define models to assess the performance of softwares for project management. Meng et al. (2017) defined a GQM model intending to recommend patterns for software processes. A model for recommendation of software design patterns based on GQM was addressed by Youssef et al. (2021). Fuggetta et al. (1998) and Solingen; Berghout (2001) employed the GQM to evaluate industrial software processes, where both studies detailed the results and experiences regarding the applied methodologies. A Human Computer Interaction assessment was proceeded by Al-nanih; Al-nuaim; Ormandjieva (2009) focusing on quantifying the quality to use Health Information Systems. A GQM model was also defined to evaluate the usability of mobile applications by Hussain; Ferneley (2008). By the end, Hernandes et al. (2012) evaluated the usefulness and ease of use from a tool to assist systematic literature reviews employing a GQM model.

The choice of GQM as a paradigm to proceed evaluations related to software was based on the advantages this methodology can provide. The cited studies that have employed GQM highlighted favorable aspects about it, as follows:

- The GQM is a suitable method to organize software measurement programs (SOLINGEN; BERGHOUT, 2001);
- The GQM methodology can be applied for multi-purpose evaluation of software (RAMADAN; ABD ELLATIF, 2010);
- The GQM is a quite well-known method which supports quantitative evaluation of software processes and products (FUGGETTA et al., 1998). Also, the GQM is seen as one of the most popular measurement approaches (HUSSAIN; FERNELEY, 2008);

- The GQM paradigm is adaptable to distinct environments. It has already been applied by several companies, as Philips, Siemens and NASA (HUSSAIN; FERNELEY, 2008);

- The GQM approach has been implemented targeting distinct purposes, such as evaluation about the characteristics of software products, assessing security in cloud environments and gathering non-functional requirements, as well as assessing software reliability (SETIAWAN; RASJID; EFFENDI, 2018; ASKARBEKULY; SADOVYKH; MAZZARA, 2020).

A GQM model can be used in a wide variety of contexts because it allows the choosing of appropriate software metrics according to the circumstances of applicability (ASKARBEKULY; SADOVYKH; MAZZARA, 2020). Also, it can have goals defined intending to evaluate software products (AL-NANIH; AL-NUAIM; ORMANDJIEVA, 2009; HUSSAIN; FERNELEY, 2008; RAMADAN; ABD ELLATIF, 2010; HERNANDES et al., 2012). A GPU application is a product developed using a programming framework. Regarding that and the advantages emphasized by the cited studies that already have employed the GQM, we selected it to be used is this Thesis. The GQM is seen as a properly approach for our objective which concerns on evaluating and comparing programming tools to implement GPU programs. Thus, a GQM method will be applied to define a model aiming to evaluate GPU applications and, as consequence, the tools employed to implement them.

4.2 Programming aspects

This Section describes concepts related to programming aspects employed to delineate a GQM model that seeks to evaluate and compare programming tools that offer resources to explore GPU devices. The following topics address two concepts: programming expressiveness (Topic 4.2.1) and programming effort (Topic 4.2.2).

4.2.1 Programming expressiveness

According to Higuera (2019), expressiveness is a word applied in the context of logic and computer science to define the breadth of ideas that can be represented using a logic, language or algebra. The terms *expressivity* and *expressive power* can also be employed to reference expressiveness. In essence, the more expressive a language is, the greater the variety and quantify of ideas it can be used to represent (HIGUERA, 2019). This concept can be detailed with the following example when different languages called A and B are compared:

- **Language A:** offers natural numbers and sum operator;

- **Language B:** offers natural numbers, as well as sum and multiplication operators.

Computing the square of a number is possible in both A and B. However, A requires to proceed the calculus with just sum, not offering a fixed formula for calculating the square of this number. Considering B, the square can be expressible as `n X n`. Hence, it can be concluded that B has a greater expressive power than A. Furthermore, B captures the expressive power of A since we can express everything of A in B (HIGUERA, 2019). The same view is shared by Felleisen (1990), who claims that a language X is more expressive than a language Y whether X can express all the facilities Y can express in a given language universe.

Following these concepts, a programming tool with more expressive power offers more resources to delineate the ideas in a source code than a tool with less power. With more programming resources, this framework can provide different ways to define some operation, offering, with that, more control over the execution of a code and opportunities for optimizations. However, more expressive capacity is not necessarily related to the "ease of use" of such tool, since we have distinct kinds of expressivity.

The expressiveness of a language can be seen with two different meanings (FARMER, 2007; HIGUERA, 2019):

- **Theoretical expressivity:** the measure of what ideas can be expressed without regard to how the ideas are expressed. This concept is commonly applied in areas of mathematics and logic that deal with the formal description of languages and their meaning, *e.g.* formal language theory, mathematical logic and process algebra.

- **Practical expressivity:** the measure of how readily ideas can be expressed. This abstraction can be useful considering programming languages, where the offered code readability can be a key factor for choosing some specific tool to implement applications.

Regarding the relations between theoretical and practical expressivity, a **Language X** can have a high practical expressiveness since it provides a great set of mathematical functions. The theoretical expressiveness of **Language X** could also be seen as high if it also offers resources to express low-level mathematical operations. Despite that, whether this language does not make low-level operations available, its theoretical expressiveness could be seen as low. Therefore, theoretical and practical expressivity are not opposite concepts because one language can provide resources considering both aspects.

The definition of an expressivity relationship leads to a natural measure of expressive power between programming languages (FELLEISEN, 1990). However, we perceived a lack of frameworks with resources to compare them using such characteristic.

Some studies have presented and surveyed comparisons focusing on low-level logics and mathematics proves to point out expressiveness differences applying a conceptual approach (DANTSIN et al., 2001; FELLEISEN, 1991; FARMER, 2007; HIGUERA, 2019). The focus of our work is to compare specific frameworks to develop applications targeting GPUs. Hence, these previous studies cannot be used as a basis to our comparisons because they commonly focus on the semantics of a logic, while to evaluate the programming expressiveness of a framework we will focus on the syntax.

4.2.2 Programming effort

The process to evaluate the programming effort of some application demands a metric. As described in Section 4.1, such metric can be quantified applying one or more measurements. As a complementary notion regarding software, measurements can be employed to describe, following previously defined rules, the properties of a program (FENTON; BIEMAN, 2014). One of these properties can be its source code. Therefore, the definition of measurements and metrics, as well as the properties to be described, are essential to proceed a programming effort evaluation.

In literature, counting lines of code (LOC) is seen as a suitable metric to evaluate programming effort. A shorter code is easier to be written, tested, maintained and less vulnerable to failure than a larger code (FENTON; BIEMAN, 2014). Additionally, Jones (2010) claims that the effort or cost of some application is related to the prediction or measure of its size, an aspect that can be analyzed from the source code of a program. Besides that, LOC metric was previously applied for this purpose by studies that proceeded similar evaluations considering programs for GPUs (MALIK et al., 2012; CHRISTGAU et al., 2014; MEMETI et al., 2017; HOSHINO et al., 2013; GIMENES; PISANI; BORIN, 2018; LIMA; DI DOMENICO, 2019; HOLM; BRODTKORB; SÆTRA, 2020). These aspects suggest that counting LOC is a good alternative in order to proceed a programming effort analysis.

Despite the status of LOC metric regarding programming effort evaluations, it is not a proper metric to measure programming productivity, since this is a topic that involves more variables, like language, programmer skills and code complexity. Another aspect that must be taken into account to count lines of codes is the programming model offered by a framework. Some approaches allow to implement more than one instruction in just one line. Models based on compiler directives are examples of that, being possible to declare several directives and clauses within one instruction. So, a strategy to balance the counting of lines has to be employed when models like that are included in the evaluations.

Using the LOC metric also requires a definition about which lines will be counted. A source code contains diverse types of instructions, like executable statements, declarations of variables, headers and comments. In a code targeting GPU applications,

more classifications are present, once there are two kinds of statements: the ones related to *host* and the ones related to *device* (GPU). Before starting the measurement process applying LOC, the sorts of lines that will be counted must be characterized to be followed as a standard.

4.3 Proposed GQM model

In this Section, we propose a model, based on the GQM method, that offers resources to guide evaluations and comparisons between programming tools applied to encode GPU applications. Firstly, we describe the goals, questions and metrics which compose the proposed GQM model. Following that, an overview about this model is introduced and its main aspects are discussed.

4.3.1 Goals, questions as metrics

Aiming to regard different perspectives in order to properly compare frameworks for GPUs, we defined three goals for our proposed GQM model. Each of them focus on an aspect that we consider relevant for the implementation and execution of a GPU program: programming expressiveness, programming effort and performance. In the sequence, these goals are characterized, also presenting their questions and metrics. The descriptions for questions and metrics which compose the model are presented following a similar approach employed by (JARDIM et al., 2021).

GO1 *- Quantify the programming expressiveness offered by a programming tool that focuses on GPU devices.*

This goal has as purpose to evaluate the **programming expressiveness** offered by a tool to encode a GPU application. This analysis is proceeded considering the programmer's point of view when implementing an application and employing the resources offered by a framework. The intention with this goal is to measure how the ideas can be represented using a programming tool to develop a GPU program, also analyzing how the resources offered by a framework to define such ideas can impact the programming effort and performance of an application. For that, the evaluations will be executed over common GPU routines, like allocation of memory or kernel invocation.

As questions for this goal, we defined three:

- **Q1.1: How much work (programming effort) the expressivity provided by the programming tool requires?**
  - **M1.1.1:**

    * Name: OPER (number of operations)
    * Definition: Counts the operations demanded to implement a specific routine related to the GPU.
    * Comment: This metric addresses the number of operations required to specify a GPU routine, from its beginning to its end.
  - **M1.1.2:**
    * Name: INDI (number of indirections)
    * Definition: Counts the indirections required to specify the operations of a specific routine related to the GPU.
    * Comment: This metric addresses the number of indirections specified in the operations counted by metric M.1.1.1 (OPER). One indirection is a token used in the source code to reference something using a name. Examples of indirections are variables, functions, data types and reserved words.

- **Q1.2: What is the potential for optimizations offered by the expressivity of a programming tool?**
  - **M1.2.1:**
    * Name: EEXE (are the operations explicitly executed?)
    * Definition: Evaluates if the GPU operations will be executed explicitly as they were specified in the source code or if they require to be transformed before processing. This metric is measured with values Yes and No.
    * Comment: Considers the source code execution of a GPU operation. Some operations can be performed following the statements defined in the code. Others can require a previous transformation to be processed instead. These transformations are usually proceeded by a compiler.
  - **M1.2.2:**
    * Name: MEXE (are the operations executed following the order which they have been disposed in the source code?)
    * Definition: Analyzes when the operations for GPU are executed. They can be processed following the exact flow defined in the source code or they can be performed in a different order (previous or future moment). This metric is measured with values Yes and No.
    * Comment: When an operation is executed in a moment different from the one defined in source code, this moment is usually defined by a compiler.

This question also applies metrics M1.1.1 (OPER) and M1.1.2 (INDI) to be answered.

- **Q1.3: What is the compatibility between the implemented and the executed code?**

  Compatibility references the relation between what is implemented by the programmer and what is effectively executed by a runtime on GPU. Evaluating this aspect is relevant to quantify the control a programmer has over the execution of an application. Also, the compatibility affects the understanding about the proceeded implementation, once greater is such compatibility, greater is the programmer's comprehension about what will be performed on GPU just reading the source code. This question is answered employing metrics M1.2.1 (EXEX) and M1.2.2 (MXEX), since they measure aspects regarding the implemented and executed code.

As an interpreting rule to guide the evaluations of this Goal, the greater is the value achieved by a metric or a question, the greater is the expressivity power measured, since the use of more operations and indirections can allow a better description of a routine. This generates opportunities to specify ideas of different manners in the code, increases the programmer's control over the application and also makes feasible the encoding of improvements and optimizations.

GO2 *- Quantify the programming effort required by a programming tool to develop an application targeting GPU.*

The present goal consists of evaluating the **programming effort** required by a framework to develop an application to explore GPUs. As defined in Goal **GO1**, this evaluation also regards the perspective of a programmer implementing a GPU application employing a programming tool. However, it focuses just on the amount of work required by a coder to implement a program, discarding how the ideas are expressed. The metrics delineated to quantify this goal are based on counting lines of code (LOC). As an evaluation rule, the bigger is the LOC number, the bigger is the required programming effort.

- **Q2.1: How much programming effort does the GPU code, including kernel implementation, require?**
  - **M2.1.1**
    - Name: PTO (programming tool operations in device code)

    * Definition: Counts the number of lines with operations related to the programming tool in device code.
    * Comment: Considers operations such as accesses to index positions and GPU shared memory.
  - **M2.1.2**
    * Name: MM (memory management)
    * Definition: Counts the number of lines with operations related to memory management.
    * Comment: Regards operations like allocations and deallocations.
  - **M2.1.3**
    * Name: CG (GPU communication)
    * Definition: Number of lines with operations for GPU communication.
    * Comment: Operations like memory transfers are taken into account here.
  - **M2.1.4**
    * Name: KIS (kernel invocation and synchronization)
    * Definition: Computes the number of lines with operations employed to invoke and synchronize kernels.
    * Comment: Considers the operations to launch and synchronize a kernel after its execution.

- **Q2.2: How much programming effort does the host code require?**
  - **M2.2.1**
    * Name: OG (operations related to GPU computation)
    * Definition: Counts lines related to operations executed on host that are useful to launch a GPU kernel.
    * Comment: Includes instructions like selection of a GPU device, definition of sizes for memory transfers and setting of blocks and threads for a kernel invocation.
  - **M2.2.2**
    * Name: OC (operations related to CPU computation)
    * Definition: Computes lines with operations encoded in host code that are applied to conclude the GPU processing using the CPU, e. g., a reduction of some value to a *host* variable.
    * Comment: This metric is related to operations that are executed on host. However, they involve data which were or will be processed on the GPU.

GO3 *- Quantify the performance delivered by an application executed on a GPU according to the programming tool employed to implement it.*

With this goal, we intend to evaluate the impact of the applied programming tool on **performance**. As a point of view to execute such evaluation, we must consider the platform where the GPU application was performed, specially the GPU specifications. Performance is a key aspect regarding GPU since this sort of device is widely employed to high-performance processing. So, the comparison of frameworks targeting GPUs must focus on performance.

- **Q3.1: How much time does the application execution require?**
  - **M3.1.1**
    - Name: TGPU (GPU execution time)
    - Definition: Computes the time for GPU execution considering a mean of 30 executions with at least 95% of confidence.
    - Comment: This metric regards just GPU performance.
- **Q3.2: What is the speedup achieved comparing sequential with GPU execution time (Ts / $Tgpu$)?**
  - **M3.2.1**
    - Name: TSER (serial execution time)
    - Definition: Computes the time for a serial execution considering a mean of 30 executions with at least 95% of confidence.
    - Comment: This metric regards executions of applications encoded with the host programming language from the framework targeting GPU.
  - **M3.2.2**
    - Name: TSPU (GPU speedup)
    - Definition: Computes the speedup dividing the results from metrics M3.2.1 (TSER) over M3.1.1 (TGPU).
    - Comment: This metric offers a perspective about the improvement achieved using a GPU in comparison with the serial performance.

4.3.2 Model overview and discussion

The proposed GQM model is composed of three goals, seven questions (two each goal but **GO1** that has three) and thirteen metrics. Goal **GO2** (Programming effort) has

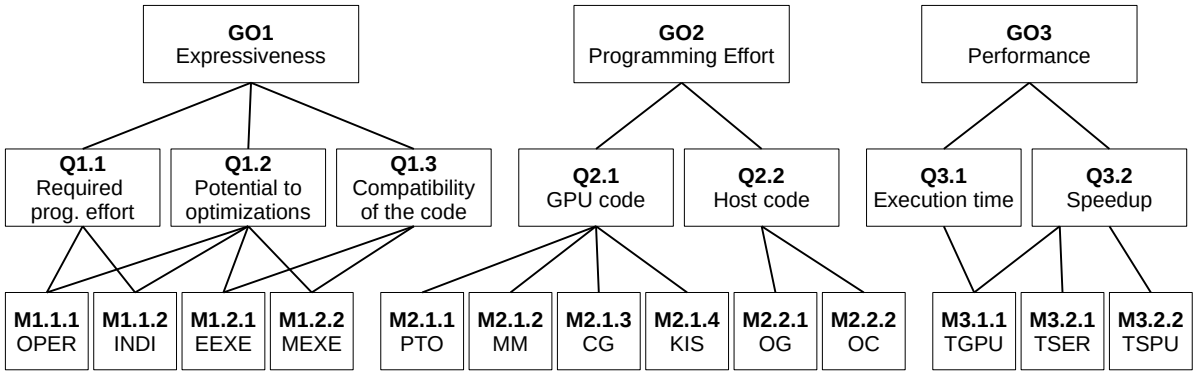

Figure 4 – Proposed GQM model to evaluate and compare programming tools targeting GPUs.

more metrics than the other ones (six). Goal **GO1** has four metrics, while goal **GO3** is composed of three metrics. Figure 4 depicts the GQM model hierarchically organized.

Goal **GO1** is defined with three questions focusing on aspects of the expressivity that can influence the implementation (**Q1.1**) and the performance of an application (**Q1.2** and **Q1.3**). Firstly, we had in mind to propose questions related to theoretical and practical expressivity. However, as long as we were developing this model, we concluded that would not be possible, since the comprehension about theoretical and practical expressivity of a programming tool is only possible after the analysis of the values obtained for the proposed metrics. Hence, this classification will be addressed with the experiments of this work (Chapter 6).

The programming effort evaluation proposed by goal **GO2** will be proceeded making a clear division about GPU (device) and host codes. This approach allows profiling the code required by a programming tool, making possible to analyze, considering effort aspects, which kinds of operations are critical by each of them. Another factor related to goal **GO2** is an analysis about the influence on effort exercised by expressiveness (contemplated by goal **GO1**), since both features take into account the implementation of an application executed by a programmer. Therefore, the impact (if there is one) that **GO1** can produce on **GO2** also will be analyzed employing our proposed model.

Goal **GO3** focuses on performance evaluations. These evaluations must quantify the influence on performance according to the framework chosen to implement a specific application over distinct hardware platforms.

In general terms, it is commonly accepted in parallel programming, specially using compiled languages like C/C++ and Fortran, that approaches based on compiler directives (like OpenMP and OpenACC) require less programming effort than other strategies (such as CUDA and OpenCL) where the programmer is responsible to define more details in the source code. This fact is intensified when we consider programming for GPUs, once the heterogeneous environment requires the management of more ar-

chitectural features to delineate the parallelism. Even with this advantage, employing compiler directives can limit the implementation of optimizations, a point that can result in a performance penalty.

Following these facts, we intend that our proposed must be capable of providing a better understanding about this relation between programming effort and performance considering GPU applications. For that, we also included other variables to be analyzed beyond them, such as programming expressiveness and the use of a high-level programming language (Python).

By the end, our GQM model was developed to be applied using a sequence of tasks. Figure 5 presents a flow detailing how to employ it. The process to evaluate a programming tool with resources for GPUs starts using as input the framework itself and applications or case studies regarding it. Next, the GQM model is applied to guide the experiments, resulting in expressiveness, programming effort and performance evaluations. Finally, these three sorts of results are summarized with a final analysis. From the outcomes of final analysis we must be able to compare several versions of one application developed with different frameworks.

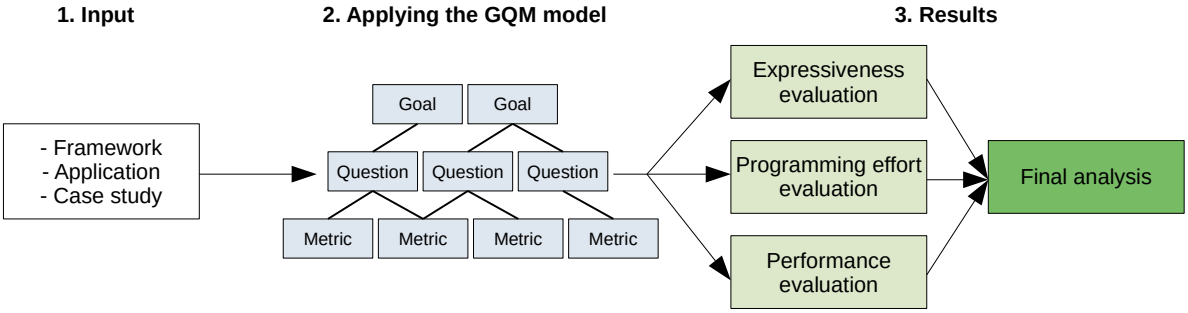

Figure 5 – Flow to apply the proposed GQM model.

In summary, we expect that the results achieved applying the proposed model according to the flow defined in Figure 5 make possible to offer a perspective about programming tools for GPUs. Such perspective must regard the characteristics, strengths and weaknesses about each of the compared frameworks. Besides, it is supposed that these results able us to answer the following questions:

1. Regarding a specific programming tool, is there an influence between the offered programming expressiveness over the required programming effort to encode a GPU application applying it?

2. Is there a relation between the programming expressiveness offered by the programming tool and the delivered performance of an application?

3. Considering performance, what are the factors that lead to a higher/lower penalty according to the programming effort demanded by the framework?

4. Are there any other aspects besides programming expressiveness and programming effort that can impact the performance of a GPU application?

4.4 Chapter overview

This Chapter presented a proposition of a model, based on the GQM method for software measurement, to evaluate and compare programming tools targeting GPUs. The proposed model was defined with three goals regarding features that we considered important for the development and execution of a GPU program: (**GO1**) programming expressiveness, (**GO2**) programming effort and (**GO3**) performance. These three goals were detailed with seven questions (three for **GO1** and two for both **GO2** and **GO3**), as well as thirteen metrics (four, six and three for **GO1**, **GO2** and **GO3** respectively). Also, it was described how the model is going to be applied and which results are expected to be achieved employing it. As background, this Chapter also addressed the context about aspects used to define the proposed GQM model, such as programming expressiveness and programming effort.

5 APPLICATIONS

In order to apply the proposed GQM model (defined in Section 4.3) to evaluate and compare programming tools with resources to explore GPUs, it is required some applications implemented employing such tools. The set of these parallel programs must be suitable for GPU processing and, to ensure a proper comparison, must provide different characteristics. To accomplish that, we selected the NAS Parallel Benchmarks (NPB) as applications to proceed experiments regarding the objective of this study.

The following sections describe the chosen applications. The Python/Numba versions of these programs were developed by us. For them, we also detailed how we proceeded such implementations.

5.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks,[1] commonly known as NPB, are a set of programs developed by NASA Advanced Supercomputing Division composed of 5 micro computational kernels and 3 simulated CFD (Computational Fluid Dynamics) applications based on important classes of aerophysics programs (BAILEY et al., 1994). The purpose of NPB is to proceed performance evaluations of highly parallel architectures, like multicore or manycore. The benchmarks, originally implemented with Fortran, are currently available employing different frameworks for parallelism. Most of them were developed with C++, probably due to the popularity of this language in the scientific programming context, being used to explore distinct kinds of platforms, like cluster, multicore, multi-zone, accelerator, and GPU.

We chose NPB for this Thesis since its programs are well-known in the parallel computing field and representative in different domains of computation. Besides that, NPB has been extensively applied on experiments regarding performance evaluations, as well as has available GPU implementations with CUDA (ARAUJO et al., 2020), OpenCL (SEO; JO; LEE, 2011) and OpenACC (XU et al., 2015). Others advantages offered by NPB are a set of predefined input sizes which can be selected as workload

[1] NAS Parallel Benchmarks: https://www.nas.nasa.gov/software/npb.html

classes, in addition to routines for verifying the outputs delivered by the execution of an application. This last factor contributes to implementing new versions of the benchmarks, once their output results can be validated.

NPB employs the word “kernel” to designate an algorithm that executes a specific transformation over some input data. This term is not related to “*kernel*” defining functions targeting devices on GPU programs. Hence, a NPB kernel is not necessarily implemented using just one GPU kernel since the ‘kernel” word is applied with different meanings for both contexts. This point must be clarified intending to avoid misinterpretations during the remainder of this work.

The following topics enumerate the five NPB kernels, as described by Bailey et al. (1994). Such topics also specify the number of CUDA kernels developed for each benchmark to explore GPUs using CUDA (regarding the implementations proceeded by (ARAUJO et al., 2020)). The number of CUDA kernels is an indication about the amount of GPU code developed and executed by each NPB program, an aspect that has relevance to analyze the results presented by this study.

- **CG**: This benchmark is called Conjugate Gradient. It calculates an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix using the inverse power method. The conjugate gradient method is applied as a subroutine for solving equations of a large, sparse, and unstructured matrix linear system. The CUDA implementation of CG is composed of 13 CUDA kernels.

- **EP**: The Embarrassingly Parallel benchmark has as goal to generate independent pairs of Gaussian random deviates. This type of problem is commonly seen in Monte Carlo simulation applications, evolutionary algorithms, artificial neural networks and many other computational intelligence techniques. Another characteristic is that the EP kernel requires few communication during computation. By the end, EP was modeled to GPU applying just 1 CUDA kernel.

- **FT**: It is a Fast Fourier Transform algorithm. The benchmark numerically solves a three-dimensional partial differential equation (PDE) using forward and inverse Fast Fourier Transforms (FFTs). The 3D FFTs are a key routine for some CFD applications (like large eddy turbulence simulations) and the steps of their computations require considerable communication. The CUDA version of this program has 14 CUDA kernels.

- **IS**: The Integer Sort kernel executes a sort over small integers. The sorting method applied by the benchmark is based on the bucket sorting approach. This algorithm was adapted to CUDA employing 12 CUDA kernels.

- **MG**: This kernel is a simple 3D MultiGrid benchmark. It computes an approximate

solution to a three-dimensional discrete Poisson problem. 9 CUDA kernels were implemented to the GPU version with CUDA.

NPB also consists of three applications. Accordingly to Bailey et al. (1994), this set of programs mimics the computation and data movement characteristics of large scale CFD applications. In summary, they solve a system of nonlinear partial differential equations applying three different algorithms, as follows:

- **BT**: Block Tridiagonal algorithm, designed targeting GPUs with 19 CUDA kernels.
- **LU**: Lower-Upper Symmetric-Gauss-Seidel algorithm, implemented for GPUs with 23 CUDA kernels.
- **SP**: Scalar Pentadiagonal algorithm, which was developed for GPUs employing 16 CUDA kernels.

Considering NPB version 3, NASA Advanced Supercomputing Division included more applications to the benchmark suite. Despite that, this study will focus just on the kernels and applications previously described that were part of NPB version 1.

5.2 Implementations with Python

The current available implementations of NPB have as main objective to deliver performance. Due that, these codes are mostly developed with low-level compiled languages. However, the present work has defined as a goal to include a perspective about the use of high-level programming languages for GPUs. Hence, we had to implement a version of NPB employing a framework which can provide such feature. To achieve that, this version was encoded with Python.

We developed a GPU version with Python for each of the five NPB kernels: CG, EP, FT, IS and MG, as well as each of the three CFD applications: BT, LU and SP. The implemented codes kept the features offered by original NPB implementations, including workload options defined by classes (from small to very large sizes), verification routines to check the correctness of an execution, also including options to time and profile operations. As part of the contributions of this Thesis (**SC2**), we made our codes available in a Git[2] repository.

The Numba environment was employed to add GPU resources to our implementations with Python, a tool that enables CUDA support for this language. More details about Python for GPUs and Numba were presented by Topic 2.2.5. The NPB programs were implemented with Python targeting GPUs based on the CUDA versions presented

[2]NPB with PYTHON: https://github.com/danidomenico/NPB-PYTHON

by Araujo et al. (2020, 2021). With the employment of Python and Numba for GPU applications, we were able to reduce the number of lines of code with operations related to the GPU framework of most NPB programs in comparison to CUDA. This evidence was confirmed in our programming effort analysis at Section 6.2. Nevertheless, we also noted that Numba did not reduce GPU *kernel* code.

We also could take advantage of the high-level programming features of Python on the development task, like operations to declare and initialize arrays with a single instruction using NumPy (NUMPY DOCUMENTATION, 2022) methods. For C/C++ and native CUDA codes, arrays must be allocated to memory, initialized in an explicit way and released from memory after the end of computing. Another advantage is native data types, specially Python complex numbers that were applied to implement the FT kernel. It allows the execution of operations (e.g.: add, divide, conjugate) without auxiliary functions as required by C++ structures.

Figures 6 and 7 present two versions of a matrix-vector multiplication algorithm for GPU encoded with native C++ CUDA and Numba, respectively. Their objective is to illustrate the differences between the two versions. The kernel implementations are similar in both codes but with different language syntax. On the other hand, the host code suffered significant changes. CUDA in Figure 6 requires several operations to allocate, initialize, copy to/from GPU and release arrays. These instructions were specified from line 16 to 34 and from line 38 to 42. NumPy and Numba methods simplify operations in Figure 7 dealing with arrays and memory copies. Hence, the high-level operations offered by Python reduce code size and managing arrays in host, demanding only 6 lines of code (22, 23, 26, 27, 30 and 40).

A number of other specific CUDA features were used for our NPB implementations such as local and shared memory resources, and atomic operations. Python built-in functions and mathematical methods from modules supported by Numba were also applied and contributed to optimize the kernels. Finally, Python's support for complex numbers eased the implementation of FT kernel, once this feature also includes GPU code compiled with Numba.

Despite the advantages in performance added by Numba to a Python code, applying it limited some operations during the implementations. One of these limitations is the use of global variables (not supported by Numba), which required the employment of more parameters to declare and call functions and GPU kernels. Particularly related to GPU codes, Numba does not support the use of constant memory in the same way of native CUDA. NPB defines some of their constant values according to the input, but Numba allows only immutable variables to be set to the GPU constant memory. This fact required the usage of global memory and/or parameters to replace such feature. Finally, as a technical restriction, some kernels of BT and SP applications couldn't be launched to execution with the same configurations of blocks and threads as the na-

```
#define N 1200 //...Imports ommited
#define M 900

__global__ //CUDA kernel
void mult_vet_mat(double *v, double *m, double *res) {
   int i = threadIdx.x + blockIdx.x * blockDim.x;

   if(i < N) {
      res[i] = 0;
      for(int j=0; j<M; j++)
         res[i] += (v[i] * m[i*M+j]);
   }
}

int main() {
   double *vet = (double*) malloc(N * sizeof(double));
   double *mat = (double*) malloc(N * M * sizeof(double));
   double *result = (double*) malloc(N * sizeof(double));

   for(int i=0; i<N; i++) { //Initialize arrays
      vet[i] = i+1;
      for(int j=0; j<M; j++)
         mat[i*M+j] = i*M+j + 1;
   }

   double *dev_v, *dev_m, *dev_r;
   cudaMalloc(&dev_v, N * sizeof(double));
   cudaMalloc(&dev_m, N * M * sizeof(double));
   cudaMalloc(&dev_r, N * sizeof(double));

   cudaMemcpy(dev_v, vet, N*sizeof(double),
         cudaMemcpyHostToDevice);
   cudaMemcpy(dev_m, mat, N*M*sizeof(double),
         cudaMemcpyHostToDevice);

   mult_vet_mat<<<(N+31)/32, 32>>>(dev_v, dev_m, dev_r);

   cudaMemcpy(result, dev_r, N*sizeof(double),
         cudaMemcpyDeviceToHost);
   // Using result in host....
   cudaFree(dev_v); cudaFree(dev_m); cudaFree(dev_r);
   free(vet); free(mat); free(result);
}
```

Figure 6 – Matrix-vector multiplication in C/C++ with CUDA.

```
from numba import cuda
import numpy

N = 1200
M = 900

@cuda.jit #CUDA kernel with Numba decorator
def mult_vet_mat(v, m, res):
   i = ( cuda.threadIdx.x +
      cuda.blockIdx.x * cuda.blockDim.x )

   if i < N:
      res[i] = 0
      for j in range(M):
         res[i] += (v[i] * m[i*M+j])
# END mult_vet_mat()


#### Host code ####

# Declare and initialize arrays
vet = numpy.arange(N)+1.0
mat = numpy.arange(N * M)+1.0

# Copy from host to device
dev_v = cuda.to_device(vet)
dev_m = cuda.to_device(mat)

# Declare dev_r on device
dev_r = cuda.device_array(N, numpy.float64)


# Call CUDA kernel
mult_vet_mat[int((N+31)/32), 32](dev_v,
                          dev_m,
                          dev_r)


# Copy from device to host
result = dev_r.copy_to_host()


# Using result in host....
```

Figure 7 – Matrix-vector multiplication in Python with Numba.

tive CUDA version. Therefore, we had to reduce the number of threads to avoid a `CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES` exception at runtime.

Another noted limitation of Python/Numba to encode GPU programs is related to automatic data transfers between CPU and GPU for NumPy arrays. This feature simplifies the code implementation, reducing the number of lines and releasing programmers to deal with memory management. Despite that, automatic data transfers should be applied carefully. Our preliminary experiments showed a considerable performance penalty when an application alternates its execution flow from host to device many times, since in each access to an array a memory transfer is executed. Thus, this automation can be seen as a systematic capability and the Numba runtime needs to be improved to proceed data transfers just when the access of some data is mandatory.

By the end, we also implemented the NPB kernels with Python to define a baseline employing sequential codes that could be compared to parallel versions. This can contribute to analyze the differences in performance between both versions (sequential and parallel). Our serial implementations were based on the C++ version presented by Löff et al. (2021). For the sequential codes, we had to deal with performance issues, since Python is an interpreted language and it is not able to deliver compu-

tational speed compared to C/C++ and Fortran, especially to process loops (HOLM; BRODTKORB; SÆTRA, 2020). The solution for that was to apply the Numba environment, since this tool can optimize Python programs adding the speed of a compiled code (NUMBA DOCUMENTATION, 2022). Although there were other tools able to proceed such optimizations (like Cython (BEHNEL; BRADSHAW; SELJEBOTN, 2009)), we employed Numba for our Python sequential codes since we already had applied it for our GPU codes.

5.3 Chapter overview

This Chapter described the GPU applications we chose intending to apply our GQM model designed to evaluate and compare GPU frameworks. The programs from NPB suite were selected since they have been largely applied on experiments related to HPC. NPB has available implementations for GPU with CUDA and OpenACC, a factor that also contributed to its choice for this work. Hence, we had to develop a NPB version with Python and Numba environment. Details about the proceeded implementations were reported by this Chapter regarding the features, advantages and limitations of Python/Numba.

Table 2 summarizes information about each NPB version employed by this study.

Version		Source	Codes available at:
Serial	C++	Löff et al. (2021)	https://github.com/GMAP/NPB-CPP
	Python	Own implementation	https://github.com/danidomenico/NPB-PYTHON
GPU	CUDA*	Araujo et al. (2020)	https://github.com/GMAP/NPB-GPU
	OpenACC*	Xu et al. (2015)	https://github.com/uhhpctools/openacc-npb
	Python/Numba	Own implementation	https://github.com/danidomenico/NPB-PYTHON

*Implemented with C/C++ language

Table 2 – NPB versions applied to the experiments.

6 RESULTS COMPARING PROGRAMMING TOOLS FOR GPUS EMPLOYING THE PROPOSED GQM MODEL

This Chapter presents results regarding experiments and evaluations aiming to compare programming tools to implement GPU applications. These results were achieved guided by the proposed GQM model delineated at Section 4.3. The GQM model considers distinct perspectives about programming, like expressiveness and effort, as well as the impact of a programming tool on performance.

We applied three programming tools during our experiments: CUDA, OpenACC and Python (using the Numba environment to enable CUDA support to this language). These three frameworks were employed to encode GPU programs: BT, CG, EP, FT, IS,[1] MG, LU and SP from NPB suite. In Chapter 5 we described such benchmarks and pointed where their source codes are available. The Python versions (serial and GPU) of these programs were developed by us. Such implementations were also detailed in Chapter 5.

The following sections present the results related to each of the aspects defined by the applied GQM model (a Section to each aspect). By the end, another Section (6.4) discusses and analyzes these results intending to provide an overview about the proceeded evaluations and comparisons, also intending to formulate a perspective regarding the characteristics, strengths and weaknesses of each programming tool.

6.1 Programming expressiveness

Results related to the executed programming expressiveness evaluations are presented here. These results are summarized in Table 3 and also discussed at the end of this Section. Our evaluations were guided by Goal **GO1** of the proposed GQM model which defined three questions with metrics to proceed the expressivity analysis, as detailed by the following topics:

[1] Results for IS implemented with OpenACC were not computed since the source code of this benchmark was not made available by the authors of the code. According to them, IS requires prefix-sum (scan) operation that is not supported by OpenACC standard yet.

- **Q1.1** (Required programming effort):
  - **M1.1.1** - *OPER*: Number of operations required to implement a GPU task.
  - **M1.1.2** - *INDI*: Number of indirections demanded by the operations counted by metric *OPER*.
- **Q1.2** (Potential to optimizations):
  - **M1.2.1** - *EEXE*: Are the operations executed as defined in the source code?
  - **M1.2.2** - *MEXE*: Do the order of execution reflects the source code implementation?
  - Metrics *OPER* and *INDI*.
- **Q1.3** (Compatibility between implemented and executed code):
  - Metrics *EEXE* and *MEXE*.

Our proposed expressivity evaluation must regard common routines that exist in a GPU application. Aiming to evaluate these routines, we represented them as case studies. Hence, our expressiveness analysis is based on five routines: Memory Allocation, Memory Transfer: Host to Device, Memory Transfer: Device to Host, Kernel Invocation and Using of Shared Memory. The following topics detail the proceeded evaluations for each of the case studies.

Each case study is represented by code examples to illustrate how the metrics of the GQM model were employed. For them, we applied some patterns:

- `v_host`: reference to a vector variable allocated on the host (main) memory.
- `v_dev`: reference to a vector variable allocated on the device memory node.
- `size_v`: variable indicating the number of elements of `v_host` or `v_dev`.

6.1.1 Memory Allocation

Figure 8 depicts pieces of code expressing a memory allocation routine on device using CUDA, OpenACC and Python/Numba.

```
1 // CUDA
2
3 cudaMalloc(&v_dev,
4     size_v * sizeof(double));
5 // ..... (code ommited)
6 cudaFree(v_dev);
```

```
1 // OpenACC
2
3 #pragma acc data create(v_host)
4
5
6 //
```

```
1 # Python/Numba
2
3 v_dev = cuda.device_array(
4     size_v, numpy.float64)
5
6 #
```

Figure 8 – Programming expressiveness: case study for **Memory Allocation**.

Accordingly to Figure 8, each of the programming tools require different operations to allocate memory on device:

- **CUDA:** requires 2 operations. To allocate memory is demanded 1 function passing 2 parameters. The second parameter is composed of 1 variable, 1 data type and 1 function. So, this operation totalizes 5 indirections. To deallocate memory, 1 function must be called passing 1 parameter, which represents 2 indirections. Both operations demand 7 indirections in total and are processed explicitly as defined in the code.
- **OpenACC:** demands just 1 operation passing the variable which must be allocated on device (1 indirection) to the `create` clause. How and when this memory allocation is proceeded relies on the compiler's decisions.
- **Python/Numba:** requires 1 operation to allocate memory. This operation calls a method which demands 4 indirections: the method itself and 3 variables. It is executed explicitly as defined in the code, despite the memory deallocation is handled implicitly by the runtime.

6.1.2 Memory Transfer: Host to Device

The operations to proceed a copy of memory from host to device in a GPU application implemented with CUDA, OpenACC and Python/Numba are illustrated by Figure 9.

```
1 // CUDA
2
3 cudaMalloc(&v_dev,
4     size_v * sizeof(double));
5 cudaMemcpy(v_dev, v_host,
6     size_v * sizeof(double),
7     cudaMemcpyHostToDevice);
8 // ..... (code ommited)
9 cudaFree(v_dev);
```

```
1 // OpenACC
2
3
4
5 #pragma acc data copyin(v_host)
6
7
8
9 //
```

```
1 # Python/Numba
2
3
4
5 v_dev = cuda.to_device(v_host)
6
7
8
9 #
```

Figure 9 – Programming expressiveness: case study for **Memory Transfer: Host to Device**.

Following the examples presented by Figure 9, each programming tool must execute:

- **CUDA:** requires a memory allocation before transferring information to GPU. After that, the data can be copied from host to GPU memory. By the end, this data must be freed. The 3 operations require 14 indirections in total: 5 to allocate memory, 7 to execute the copy and 2 to deallocate the data. All of them are performed explicitly as defined in the code.
- **OpenACC:** demands just 1 operation passing the variable which must be copied to device to the `copyin` clause (1 indirection). How and when this data is copied, as well as allocated and deallocated, depends on the compiler's decisions.
- **Python/Numba:** the memory copy is defined using just 1 operation. The method to perform such task requires 3 indirections: the method itself and 2 variables.

This operation is executed explicitly as defined in the code and, if necessary, the runtime implicitly allocates and deallocates the data.

6.1.3 Memory Transfer: Device to Host

This Topic regards operations related to memory transfers from device to host for the three programming tools that are being evaluated. Figure 10 depicts codes defining these operations for each of the APIs.

```cuda
// CUDA

cudaMemcpy(v_host, v_dev,
    size_v * sizeof(double),
    cudaMemcpyDeviceToHost);
```

```openacc
// OpenACC

#pragma acc data copyout(v_host)

//
```

```python
# Python/Numba

v_host = v_dev.copy_to_host()

#
```

Figure 10 – Programming expressiveness: case study for **Memory Transfer: Device to Host**.

Considering the codes detailed by Figure 10, it is possible to imply:

- **CUDA:** a memory transfer from GPU to host can be executed by itself, since the allocation and deallocation of host data is not related to a GPU operation. Hence, CUDA requires just 1 operation with 7 indirections: 1 function call with 4 parameters (the third parameter is composed of 3 indirections). Such operation is executed explicitly as defined in the code.
- **OpenACC:** just 1 operation must be defined, passing the variable that has to be copied to host to the `copyout` clause. This results in 1 indirection. How and when this memory allocation is proceeded relies on the compiler's decisions.
- **Python/Numba:** a memory transfer demands just 1 operation calling a method which has 3 indirections: the method itself and 2 variables. This operation is executed explicitly as defined in the code.

6.1.4 Kernel Invocation

This evaluation regards the simplest way a GPU kernel can be explicitly invoked with CUDA, OpenACC, and Python/Numba. Figure 11 displays the source codes to execute such task for these three frameworks.

```cuda
// CUDA

kernel<<<blocks, threads>>>(
    p1, p2, ..., pn);
```

```openacc
// OpenACC

#pragma acc parallel loop
//
```

```python
# Python/Numba

kernel[blocks, threads](
    p1, p2, ..., pn)
```

Figure 11 – Programming expressiveness: case study for **Kernel Invocation**.

The code for each of the programming tools can be summarized as follows:

- **CUDA** and **Python/Numba:** a kernel invocation is defined in the same way for both APIs. This task requires 1 operation with at least 3 mandatory indirections for both programming tools: `blocks` (number of blocks) and `threads` (number of threads), as well as the kernel name. Furthermore, 0 or more parameters should be passed accordingly to the kernel implementation. This operation is executed as defined in the code.

- **OpenACC:** a kernel can be explicitly called combining the directives `parallel` and `loop` in 1 operation. No specific parameter (indirection) is required, since the number of blocks and threads are defined by the compiler. OpenACC offers a way to define levels of parallelism like CUDA and Python/Numba. Nevertheless, OpenACC is a framework for generic accelerators. So, there is no direct mapping to CUDA's threads and blocks. If necessary, these levels can be defined using `gang`, `worker` and `vector` clauses (from outermost to innermost level).

The kernel parameters can also be seen as indirections for this case studies. However, we will not compute them to evaluate expressiveness since it depends on the application characteristics. Hence, they won't be counted as operations or indirections.

6.1.5 Using of Shared Memory

Figure 12 shows examples of code with operations to apply the GPU shared memory on kernel computations.

```
1 // CUDA
2
3 __shared__ double v[size_v];
4 // ..... (code ommited)
5 __syncthreads();
6 //
```

```
1 // OpenACC
2
3 #pragma acc parallel \
4         private(v_host)
5 #pragma acc loop
6 // ..... (code/loop ommited)
```

```
1 # Python/Numba
2
3 v = cuda.shared.array(size_v,
4     numba.float64)
5 # ..... (code ommited)
6 cuda.syncthreads()
```

Figure 12 – Programming expressiveness: case study for **Using of Shared Memory**.

The depicted codes illustrate different ways to deal with shared memory. The following topics details each of these codes:

- **CUDA:** a shared memory variable must be declared using CUDA's `__shared__` construct. This operation requires 4 indirections: 1 reserved word, 1 data type, 1 variable name and 1 array size. Also, it is required another operation (with 1 indirection) to synchronize the threads that are writing/reading to/from the shared variable. When processing the code, both operations, which totalizes 5 indirections, are executed explicitly as defined in the code.

- **OpenACC:** this framework demands at least 2 operations to define a shared variable (`parallel` and `loop` directives). The variable must be declared as private in the `parallel` directive. So, for each parallel loop inside the parallel section,

such variable can be defined as shared by the compiler. However, this decision is taken by the compiling tool. Both operations required just 1 indirection, that is, the variable which will be allocated on shared memory.

- **Python/Numba:** shared memory variables are defined in the same way as CUDA. Two operations with a total of 5 indirections are required to handle them: 1 to declare the variable, which demands 4 indirections, and 1 to synchronize the threads, which adds another indirection.

6.1.6 Programming expressiveness overview

Table 3 summarizes the metrics to evaluate programming expressiveness. These metrics were applied over the case studies detailed in the previous topics of this Section. The Python/Numba programming tool is referenced as just Numba in the table. We also translated the Yes and No values for metrics *EEXE* and *MEXE* to 1 and 0, respectively. This was necessary in order to compute the expressivity values for each of the questions defined by Goal **GO1** of the GQM model.

Case study	Program. tool	Metrics and Questions							
		OPER	INDI	EEXE	MEXE	**Q1.1**$^{+}$	**Q1.2**$^{+}$	**Q1.3**$^{+}$	**Total***
Memory Allocation	CUDA	2	7	1	1	**9**	**11**	**2**	22
	OpenACC	1	1	0	0	**2**	**2**	**0**	4
	Numba	1	4	1	1	**5**	**7**	**2**	14
Memory Transfer: Host to Device	CUDA	3	14	1	1	**17**	**19**	**2**	38
	OpenACC	1	1	0	0	**2**	**2**	**0**	4
	Numba	1	3	1	1	**4**	**6**	**2**	12
Memory Transfer: Device to Host	CUDA	1	7	1	1	**8**	**11**	**2**	21
	OpenACC	1	1	0	0	**2**	**2**	**0**	4
	Numba	1	3	1	1	**4**	**6**	**2**	12
Kernel Invocation	CUDA	1	3	1	1	**4**	**6**	**2**	12
	OpenACC	1	0	0	1	**1**	**2**	**1**	4
	Numba	1	3	1	1	**4**	**6**	**2**	12
Using of Shared Memory	CUDA	2	5	1	1	**7**	**9**	**2**	18
	OpenACC	2	1	0	0	**3**	**3**	**0**	6
	Numba	2	5	1	1	**7**	**9**	**2**	18

$^{+}$**Q1.1**=*OPER* + *INDI*, **Q1.2**=*OPER* + *INDI* + *EEXE* + *MEXE*, **Q1.3**=*EEXE* + *MEXE*
***Total** = Q1.1 + Q1.2 + Q1.3

Table 3 – Programming expressiveness evaluation.

Each question of Table 3 was totalized using their own metrics. As introduced by Goal **GO1**, the greater is the displayed number by a metric or a question, the greater is the expressivity power of the programming tool regarding a case study. More operations and indirections means more ways to describe a routine, increasing the programmer's control over the application and also creating opportunities to implement improvements and optimizations.

As a result of the evaluations, OpenACC seems to provide a simpler way to express GPU operations in the source code for all case studies. Question **Q1.1** pointed out that

OpenACC required less operations and indirections than CUDA and Numba. However, based on the interpreting mechanism defined by our proposed GQM model, this simplicity can prevent the encoding of optimizations, since less options and parameters are available to the encoder implementing them. The execution of OpenACC code is also dependent on compiling. All the case studies rely on the compiler to be effectively performed, decreasing the compatibility between the implemented and the executed code (Question **Q1.3**). Hence, the programmer, just reading the source code, can't be totally sure that the program will be executed as the way he/she has implemented it, reducing the control that he/she has over the application.

CUDA results show a framework which offers an environment with great expressivity, since the GPU code is implemented using more operations and indirections than OpenACC and Numba. This greater expressivity was achieved for all three questions and five case studies in Table 3, implying that CUDA is a tool that provides good control over the application execution (compatibility between implemented and executed code), as well as resources to implement optimizations.

The high-level approach offered by Numba due to Python language can also be seen in Table 3. According to the expressivity evaluation, this tool delivers the same compatibility between source and executed code as CUDA (Question **Q1.3**), but the GPU routines can be implemented with less operations and indirections for most of the case studies. The results also show that Numba and CUDA provide a greater power to develop optimizations in the code than OpenACC. Additionally, as showed by Question **Q1.3**, Numba offers this resource requiring simpler routines than the ones demanded by CUDA.

By the end, this programming expressivity evaluation indicates some patterns about the frameworks that can impact performance and programming effort. So, these results motivated the execution of the new studies regarding such aspects. They will be presented in the next sections of this Chapter.

6.2 Programming effort

This Section addresses the results considering the proceeded programming effort evaluations, comparing the effort required by frameworks for GPUs as defined by Goal **GO2** of the proposed GQM model. As detailed in Section 4.3, this goal intends to answer two questions about the code of a GPU application: **Q2.1** (programming effort required by GPU code) and **Q2.1** (programming effort required by host code).

We employed the counting of lines of code (LOC) to quantify the programming effort. LOC counting was detailed in Section 4.2.2. We only counted the executable statements of the source code, that is, blank and comment lines, as well as variable declarations and headers were discarded. Therefore, this evaluation focused on pro-

gram's operations that are effectively executed. Due to the choice of LOC, we limited our analysis to effort. Also as mentioned in Section 4.2.2, the evaluation of programming productivity aspects require more variables to be properly executed.

The programming effort evaluations are based on applications. Thus, three versions of the NPB programs were applied for the experiments implemented with CUDA, OpenACC and Python/Numba. The following items enumerate the proposed metrics for Goal **GO2**. They represent types of operations. OpenACC is a framework that uses an approach regarding compiler directives. As pointed in Section 4.2.2, some strategy can be used to balance the counting of lines, since such approach allows the specification of several operations in just one line of code. Hence, the following items also details which kinds of instructions were considered for OpenACC by each of the metrics.

- **Q2.1:**
  - **M2.1.1** - *PTO*: Operations related to the programming tool in device code. For OpenACC, counts clauses which describe loops (`gang`, `vector` and `worker`) and reductions.
  - **M2.1.2** - *MM*: Memory allocations and deallocations. Variables listed in the `create` clause of `data` directive, as well as `acc_malloc` and `acc_free` calls were classified as MM operations for OpenACC.
  - **M2.1.3** - *CG*: GPU communication (data transfers). OpenACC allows the definition of these operations through clauses like `present`, `copyin`, `copyout` and `deviceptr`, including also directive `update`.
  - **M2.1.4** - *KIS*: Kernel invocations and synchronization. OpenACC allows defining these operations using `kernels` and `parallel` directives.
- **Q2.2:**
  - **M2.2.1** - *OG*: Host code to setup GPU kernels and data transfers.
  - **M2.2.2** - *OC*: Host code to conclude GPU processing (e.g. reducing. some variable on host).

Table 4 shows the number of lines of code with operations required by each programming tool for all applications grouped by type/metric. In the table, Python/Numba is referenced as just Numba. As can be seen, OpenACC demanded less operations than Numba and CUDA for all applications. In the analyzed OpenACC codes, this number of operations could be even lower, since many directives and clauses were used to optimize performance, adding instructions to improve memory management, data transfers and loops. This explains the large number for *PTO*, *MM* and *GC* operations. For the MG kernel, there are more operations of these types than specified

with Numba and CUDA. For FT and LU, *GC* operations also have a great number with OpenACC. Despite that, it is pertinent to mention that such programs could run with less or even without these optimizations reducing the necessary effort. For BT, LU, MG and SP, *KIS* operations for OpenACC can also be highlighted due to the greater number of `parallel` directives on the codes.

NPB application	Program. tool	Number of lines with operations (Metrics and Questions)								
		PTO	MM	GC	KIS	OG	OC	**Q2.1**$^{+}$	**Q2.2**$^{+}$	**Total***
BT	CUDA	193	24	108	19	367	0	**344**	**367**	711
	OpenACC	110	18	28	46	2	0	**202**	**2**	204
	Numba	204	10	17	19	248	0	**250**	**248**	498
CG	CUDA	76	26	14	13	96	17	**129**	**113**	242
	OpenACC	21	5	11	16	2	0	**53**	**2**	55
	Numba	76	5	14	13	77	17	**108**	**94**	202
EP	CUDA	19	6	3	1	19	0	**29**	**19**	48
	OpenACC	9	2	7	4	1	0	**22**	**1**	23
	Numba	19	3	3	1	18	0	**26**	**18**	44
FT	CUDA	50	16	3	25	89	0	**94**	**89**	183
	OpenACC	25	11	69	12	1	0	**117**	**1**	118
	Numba	50	8	3	25	85	2	**86**	**87**	173
IS$^{\&}$	CUDA	44	20	5	15	62	2	**84**	**64**	148
	Numba	45	7	6	15	54	1	**73**	**55**	128
LU	CUDA	527	12	34	24	247	4	**597**	**251**	848
	OpenACC	149	17	67	58	2	0	**291**	**2**	293
	Numba	523	6	5	24	230	4	**558**	**234**	792
MG	CUDA	64	12	6	9	116	11	**91**	**127**	218
	OpenACC	69	21	22	24	2	0	**136**	**2**	138
	Numba	64	1	6	9	100	9	**80**	**109**	189
SP	CUDA	109	26	98	16	151	0	**249**	**151**	400
	OpenACC	154	23	44	65	2	0	**286**	**2**	288
	Numba	169	13	10	16	131	4	**208**	**135**	343

$^{+}$**Q2.1**=*PTO* + *MM* + *CG* + *KIS*, **Q2.2**=*OG* + *OC*
*__Total__ = Q2.1 + Q2.2
$^{\&}$**IS** = Results for OpenACC were not shown since the source code of the program is not available.

Table 4 – Programming effort evaluation according to operation type.

Numba and CUDA required a similar number of operations for most metrics, including the ones related to GPU device code (metrics of Question **Q2.1**). The exceptions are *MM* and *OG*. Numba required half or less *MM* operations than CUDA with an interface that does not demand memory deallocations. The code examples in Section 5.2 illustrate the reasons why Numba demanded less *MM* operations than CUDA code. *OG* also presents a scenario where CUDA needed more operations. The specification of the amount of data to be transferred between memory nodes is the explanation for that. Finally, *OG* operations for OpenACC presented a smaller number than Numba and CUDA. However, these implementations have features to configure GPU execution using parameters, while OpenACC just applied constant values for that.

Considering programming effort results, Numba seems to be similar to CUDA, although it demanded less operations in total for most applications (except SP). As an

advantage, Numba offers Python's high-level approach to implement GPU applications that contributes to reducing the number of required operations to deal with allocation and movement of memory. Memory operations are usually seen as the main programming bottleneck for many applications (LI; KESSLER, 2017). Additionally, for the FT kernel, Python's support for complex numbers reduced operations on GPU code from 265 to 237 in comparison with CUDA. Numba provides many control features to develop code optimizations at the same level as CUDA. Such aspect requires more operations to manage several details about GPU execution, which can be seen as a disadvantage in terms of effort, especially when compared with OpenACC. The NPB implementations with OpenACC employed many operations intending to optimize their performance. Even with that, OpenACC code required less statements than Numba and CUDA.

Regarding questions **Q2.1** and **Q2.2**, it seems that OpenACC required less lines than CUDA and Numba for both questions in the implemented applications. Also, Numba demanded fewer lines of code than CUDA. Hence, the effort required by the experimented programming tools follows the same behavior for both device (**Q2.1**) and host (**Q2.2**) code.

6.3 Performance

After presenting the results about programming expressiveness and programming effort, this Section introduces results regarding performance. As defined by the proposed GQM model, the performance evaluation references Goal **GO3**. This goal comprises two questions: **Q3.1** (GPU execution time) and **Q3.2** (speedup). To answer them, three metrics must be computed:

- **M3.1.1** - *TGPU*: Execution time on a GPU platform (T_{GPU});
- **M3.2.1** - *TSER*: Execution time on a CPU platform (serial or T_S);
- **M3.2.2** - *TSPU*: Speedup, calculated dividing serial execution time by GPU execution time (T_S / T_{GPU});

We proceeded experiments on a GPU device aiming to quantify these metrics and, with that, answering questions **Q3.1** and **Q3.2**. These experiments were performed on a Dell T420 machine composed of two Intel Xeon E5-2420 Sandy Bridge processors and 6 cores per processor (totalizing 12 cores and 24 threads contexts) running at 1.9 GHz and 80 GB main memory (NUMA access). The machine is enhanced with two Nvidia Titan V GPUs with 5,120 CUDA cores at 1,200 MHz and a 12 GB of HBM2 memory. Our experiments were restricted to 1 CPU core and 1 GPU without CPU parallelism. The software environment used was Debian 10 operating system, CUDA

11.2, GCC 8.3.0, PGCC 21.2, OpenACC 2.7, Python 3.8.8, Numba 0.53.1 and LLVM 10.0.1.

Regarding this section, each presented result was a mean of 30 executions. To ensure a confidence of at least 95% we performed a statistical analysis over the experimental data. This analysis contemplates Kolmogorov–Smirnov Test to evaluate whether the samples are normally distributed. Besides, to validate the comparisons between different versions of the same benchmark, we applied the Student's T-Test and, for non-distributed samples, the Mann–Whitney U-Test. A detailed report containing all executed statistical evaluations is available in Appendix A and also in this Git[2] repository. To reduce the impact of an unexpected operating system interference or other unexpected situations, we ran all programs on a dedicated computer with unused daemons disabled. We ran the benchmarks in an interleaved manner, mixing programming tools and input sizes, reducing the probability of interference that could affect the quality of results collected for a given combination of parameters.

The programs of NPB suite (BT, CG, EP, FT, IS, LU, MG and SP) were executed with classes B and C as input sizes. Both classes are fixed and preset within NPB source codes. Accordingly to NPB documentation, class C has a workload about 4 times larger than class B. For each of these programs and input sizes, we performed experiments with 5 versions, defined as:

- **C++**: serial C++ code compiled with GCC using `-O3` optimization flag. As previously mentioned, these codes were presented by (LÖFF et al., 2021).
- **Python**: serial Python code developed during the term of this Thesis. The critical performance sections were compiled on-the-fly through the Numba environment.
- **CUDA**: GPU code developed with C++ and CUDA. The kernels were compiled with NVCC employing the `-O3` optimization flag. CUDA implementations were developed by (ARAUJO et al., 2020).
- **OpenACC**: GPU code developed with C applying OpenACC directives targeting Nvidia GPUs. These codes were originally implemented by (XU et al., 2015) and adjusted to be compiled with PGCC by (ARAUJO et al., 2020). To build them, it was applied `-O3 -mcmodel=medium` compiling flags. To execute them, it was required to use the command `ulimit -s unlimited` to increase the memory available in the stack. Obs.: as previously mentioned, the IS implementation was not made available by their authors, so their results were not shown.
- **Numba**: GPU code implemented by us with Python. CUDA support was enabled with the Numba environment.

[2]Statistical analysis report: https://github.com/danidomenico/NPB-PYTHON-stat-analysis

As a first result for our performance experiments, Figure 13 depicts charts for the NPB programs regarding GPU time (T_{GPU}). These results are the data employed to compute metric *TGPU* and to answer question **Q3.1**.

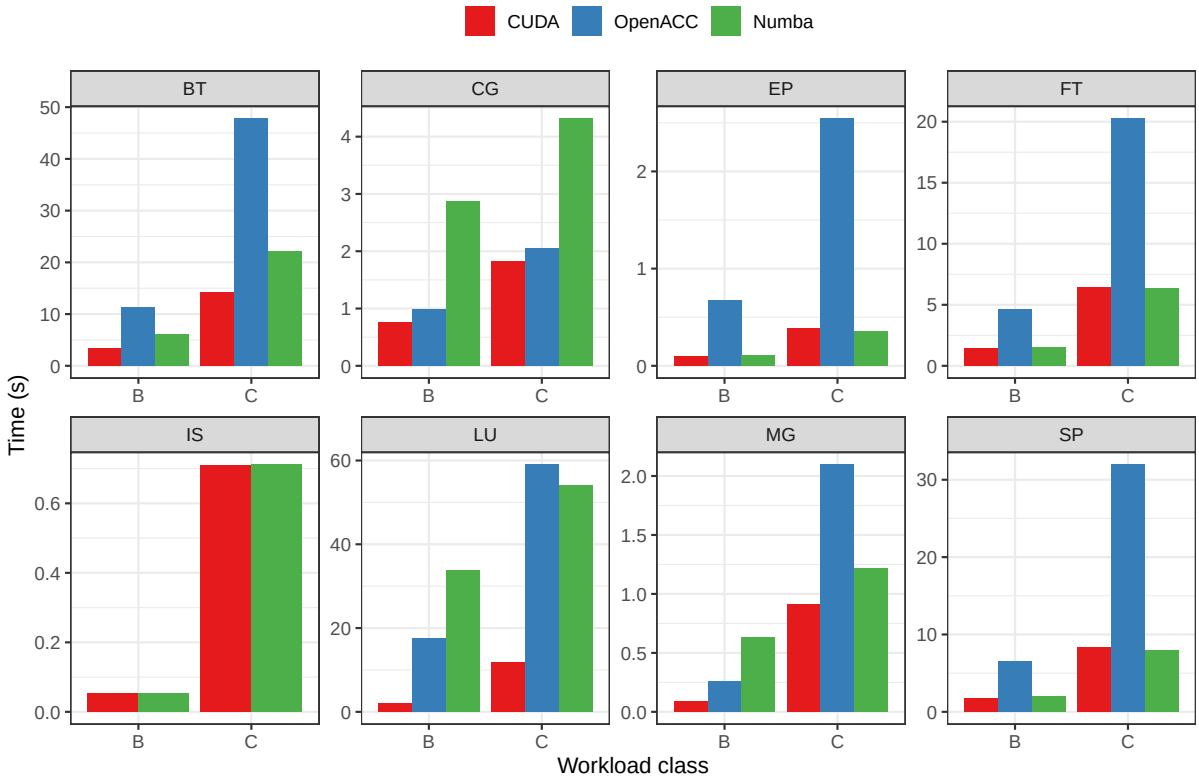

Figure 13 – Execution time for NPB programs on GPU (T_{GPU}). (Results for IS implemented with OpenACC were not shown since the source code of the program is not available.)

For most cases, CUDA and Numba outperformed OpenACC, as well as CUDA and Numba achieved similar results. Exceptions were the CG kernel, where CUDA and OpenACC performed better than Numba, and MG kernel and LU application, where CUDA version performed better than Numba and OpenACC version was faster than Numba just with class B. CUDA results were faster than OpenACC as previously reported by Araujo et al. (2020).

The second result considering performance is related to executions on a CPU processor (metric *TSER* from the proposed GQM model). Table 5 shows the serial time (T_s) obtained by C++ and Python versions for the NPB suite. Sequential times with C++ were faster than Python, except for EP kernel, where the Python version was faster for both classes. EP is the simplest kernel of the NPB suite and also has the shortest code. It seems that Python and Numba are able to optimize programs with less complexity (such as EP) when the code is compiled employing the LLVM.

Applying the results displayed by Table 5, we were able to calculate metric *TSPU* (speedup) and answer question **Q3.2**. Hence, Figure 14 introduces performance results of GPU executions (T_{GPU}) over sequential times (T_s) aiming to show the speedup

NPB program	Time (T_s)			
	Class B		Class C	
	C++	Python	C++	Python
BT	281.101	366.054	1,167.523	1,507.663
CG	112.241	163.602	318.593	427.222
EP	142.433	129.839	569.267	518.092
FT	125.279	151.234	372.670	622.605
IS	4.354	7.510	19.381	30.977
LU	209.341	246.204	1,008.549	1,141.680
MG	5.759	20.457	51.517	162.779
SP	208.363	229.561	843.760	934.838

Table 5 – Serial execution time in seconds.

values (T_s / T_{GPU}). We employed the execution times achieved by C++ serial versions to determine CUDA and OpenACC speedups, while it was used the times obtained by Python sequential executions to compute Numba speedups. Using this approach, improvements of GPU versions over the base programming language (C/C++ or Python) are going to be highlighted.

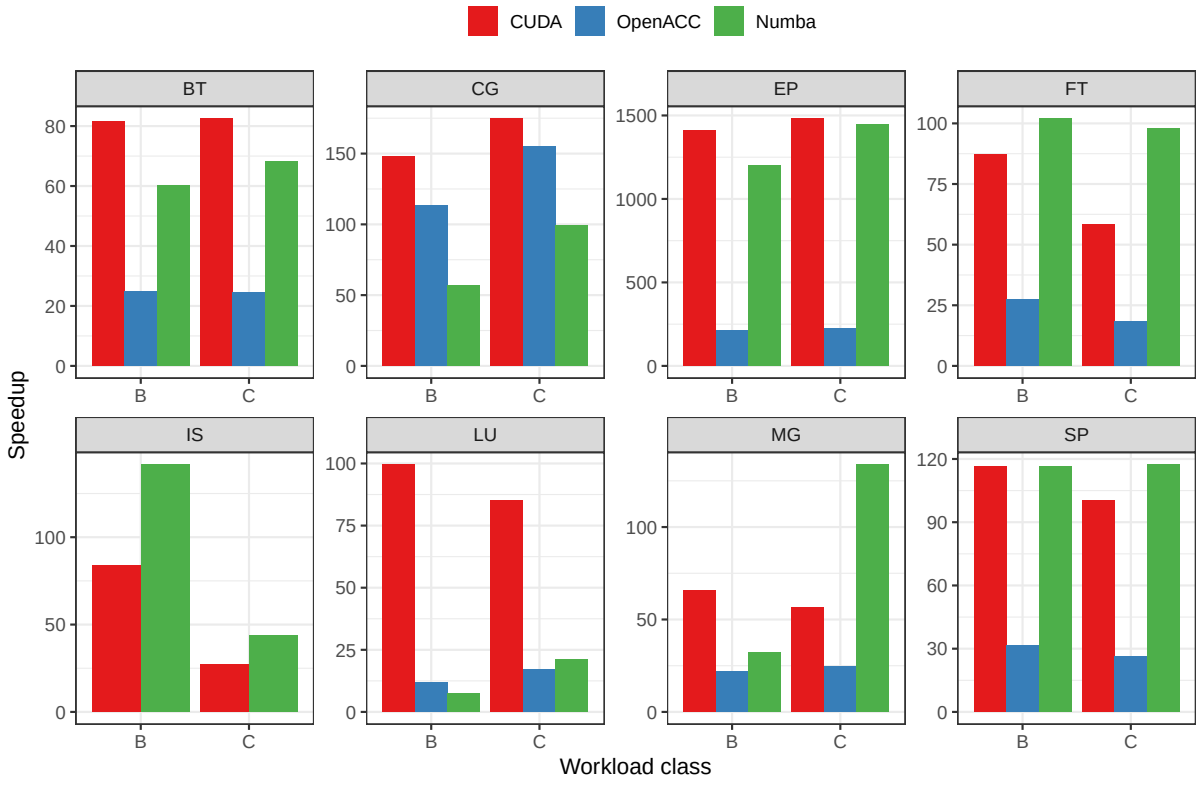

Figure 14 – Speedup results for NPB programs (T_S / T_{GPU}). (Results for IS implemented with OpenACC were not shown since the source code of the program is not available.)

CUDA reached the best speedups for BT, CG, EP and LU, while Numba had the best results for FT and IS. The best speedups for MG and SP were achieved by CUDA with class B and by Numba with class C. Numba and CUDA outperformed OpenACC results for almost all kernels. The exceptions were CG and LU (only from class B)

where OpenACC had a better speedup than Numba. These comparisons between versions were validated applying the Student's T-Test and the Mann–Whitney U-Test (for non-distributed samples). More details about these statistical tests can be seen in Appendix A.

As a general overview about the experiments, the best performance results were achieved by CUDA. However, Numba reached a performance similar to CUDA for BT, EP, FT, IS, and SP, while OpenACC had the worst results for most of the programs. The exceptions were CG, LU and MG benchmarks with Numba which performed considerably slower than CUDA and even OpenACC.

The presented results showed that CUDA outperformed OpenACC for all the NPB programs. The reasons for this fact were already reported and discussed by previous works (Araujo et al. (2020, 2021)), where the authors emphasized factors like non-isolation of irregular computations, low GPU usage and the grain of parallelism applied to justify the poor results reached by OpenACC versions. Numba and CUDA codes were implemented employing the same parallelism strategies and applied the same optimizations for GPUs. So, it was predicted that they would achieve similar performances, as well as it was expected that Numba would perform better than OpenACC. Regarding that, the next chapters are going to focus on the performance differences between Numba and CUDA, since for some programs the Numba version did not behave as expected.

We proceeded an analysis to verify the reasons which caused the Numba drawbacks for CG, LU and MG. GPU traces generated with `nvprof`[3] tool showed two: (1) Numba requires more time to execute some GPU kernels than CUDA and (2) Numba executions presented a low percentage of GPU utilization. For CG, the `gpu_kernel_three()` routine, which represents more than 80% of the workload executed on GPU, is performed 14.0% and 11.9% slower with Numba than with CUDA for classes B and C, respectively. For MG, the `resid_gpu_kernel()` and `psinv_gpu_kernel()` GPU kernels are the main tasks, together representing the processing of more than 60% of one MG interaction. Numba performs both tasks 23.3% and 12.0% slower than CUDA for classes B and C, respectively. The distinctions between CUDA and Numba were not related to the number of blocks or threads per block to invoke the GPU kernels because we used the same values for both versions in all programs. One aspect that can justify the slower executions of the GPU kernels with Numba is the different compiler employed for each version. CUDA programs were compiled by us with NVCC, while Numba ones were automatically compiled with the LLVM library. A difference considering the applied optimizations by both compiling tools related to the loop unrolling was already described by Oden (2020).

[3] `nvprof` is a tool offerred by Nvida to collect and view profiling data of CUDA-related activities on both CPU and GPU.

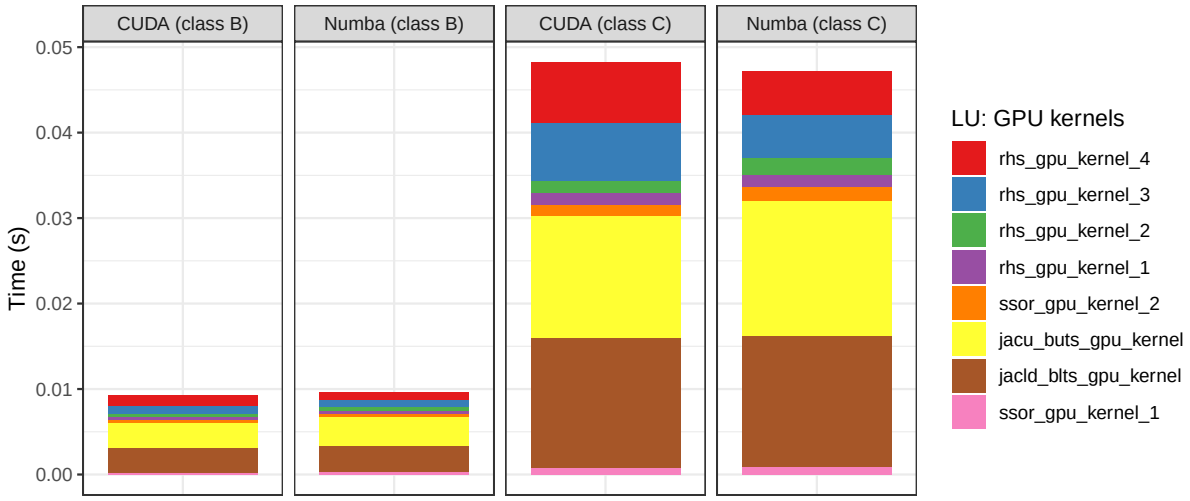

Figure 15 – Traces for LU application regarding the execution of one interaction on GPU - classes B and C for both CUDA and Numba.

Table 6 displays the percentage of GPU utilization for CG, LU and MG programs developed with CUDA and Numba, detailing the ratio between the processing time just on the GPU and the total time required by the whole application. As can be seen, the three benchmarks presented a lower GPU utilization with Numba. For CG and MG kernels, this fact impacted performance along with the slower executions of the GPU routines previously detailed. For LU application, this is the key aspect to explain the performance penalty with Numba, since both CUDA and Numba achieved similar times executing routines on the GPU (as showed by Figure 15). The low percentage of GPU utilization with Numba happens because some routines of the benchmarks cannot be accelerated to GPU, being performed sequentially by Python on the CPU. Python, as an interpreted language, executes serial code much slower than compiled languages. Hence, these non-GPU parts became a bottleneck and negatively affected performance for Numba applications. This behavior was also reported by Oden (2020). The performance drawback was higher on class B than class C (Figure 13) since the GPU utilization on class C increases with Numba.

	Percentage of GPU utilization*			
	Class B		Class C	
NPB program	CUDA	Numba	CUDA	Numba
CG	95.18	29.32	96.94	47.05
LU	94.95	6.53	99.11	20.43
MG	99.99	16.60	99.40	79.69

*Time on GPU / Total time of the application

Table 6 – GPU utilization: percentage of execution exclusively on the GPU device.

For Numba and CUDA implementations, the LU is a benchmark composed by various GPU kernels launched several times from the same CPU routine called `ssor_gpu`.

This routine consists of some loops (including nested ones) to invoke the GPU kernels and to compute the amount of blocks and threads per block before each kernel launching. The heuristic to calculate the number of blocks and threads based on the workload that was applied by the CUDA version was also employed by our Numba code. Although the entire LU application is performed on the GPU, the processing time of `ssor_gpu` routine on CPU is also gauged as part of the total time achieved by the program. Therefore, the `ssor_gpu` function is the main aspect for the Numba drawbacks caused by a low percentage of GPU utilization, since this function is executed by Python interpreter. We were not able to encode optimizations for the Numba version to increase the speed of serial processing on GPU programs because Numba does not support the launching of GPU kernels from optimized routines.

6.4 Discussion and final analysis

This Section discusses and proceeds a final analysis regarding the executed experiments intending to achieve goals **GO1** (expressiveness), **GO2** (effort) and **GO3** (performance) of the proposed GQM model. Analyzing the results about each of the goals, this Section aims to build a perspective addressing the three programming tools applied during the experiments (CUDA, OpenACC and Python/Numba).

Firstly, the results suggest a relation between Goals **GO1** and **GO2**. The *OPER* metric from Goal **GO1**, which maps the number of operations to implement a GPU routine, is related to the LOC metric used during the programming effort evaluation. Thus, if a framework offers an expressiveness that demands more operations to specify the ideas in the code, it will probably require more lines of code to develop an application. Nonetheless, this relation between both Goals can be seen as incomplete, once the programming effort metric does not map the indirections required by the expressivity. If so, it would be possible to differentiate programming models which require the same number of operations but a distinct number of indirections. In a situation like that, the effort could be higher or lower depending on the number of indirections.

Goals **GO1** and **GO3** are related since we evaluated the potential for optimizations of a programming tool offered by its expressiveness. The results of expressivity and performance over CUDA, OpenACC and Numba suggest that frameworks which offer an interface requiring more operations and indirections have a high potential to provide better performance. A factor that contributes for this resource considers the GPU routines which are executed as explicitly as implemented (high compatibility between source and executed code) following the order in which these routines have been disposed in the source code. On the other hand, the programming tool itself can add performance penalties which are not related to the expressiveness. This was observed analyzing the performance results of some Python/Numba applications, which

presented slowdowns related to limitations from Numba environment and Python language.

A relation between Goals **GO2** and **GO3** can also be implied from the results. OpenACC, despite requiring less lines of code than CUDA and Python/Numba, suffered from huge slowdowns for most of the executed benchmarks. Regardless that, this relation can also be associated with the expressivity offered by a programming model. For example, CUDA and Python/Numba achieved similar performance for most of the applications, even with Python/Numba requiring less effort than CUDA. It seems that the expressivity provided by Python/Numba allows to implement a GPU code with less operations than CUDA. Hence, the results suggest that programming effort itself can be employed as an indicator about the performance delivered by an application. However, a proper analysis about the impact of a framework on performance should also consider the programming expressiveness evaluation.

Regarding just Goal **GO1**, it seems that the proceeded programming expressiveness evaluation (Section 6.1) did not converge to a classification contemplating both theoretical and practical expressivity. The applied metrics can better characterize the notion of practical expressivity, since they measure how ideas can be expressed employing a framework. Theoretical expressivity focuses on what ideas can be expressed, an aspect not addressed by these metrics. Following this premise, the proposed GQM model able us to conclude that the greater is the expressiveness value achieved by a programming tool, the greater is the practical expressivity of this framework. Hence, the ideas expressed with its API seems to offer a better readability in comparison to a framework that scored less points for the applied metrics and questions.

As a complementary perspective, APIs are conventionally classified as high-level or low-level considering the programming abstractions provided by them to develop an application. After concluding the experiments with GPUs, our evaluations suggest the requirement of two variables to properly classify these tools considering this characteristic: the programming language (C/C++ or Python) and the framework (CUDA, OpenACC and Numba). So, regarding the employed programming resources (languages and frameworks), we have for CUDA a low/low scenario, since both language and framework offer low-level abstractions. For OpenACC we have a low/high situation and for Numba a high/low case. A general categorization concerning the abstraction of each language/framework is depicted by Table 7. Unfortunately, we were not able to proceed experiments with a high-level framework with Python (high/high scenario) since we could not find any tool offering features similar to OpenACC available for it. We found a project with such features in this repository,[4] but it is still a prototype.

To finalize, it seems that, considering CUDA, OpenACC and Python/Numba, the

[4]pyACC: https://github.com/MaxStrange/pyACC

	Framework	
Language	CUDA*	OpenACC
C/C++	Low Low	Low High
Python	High Low	–

*Numba enables CUDA support for GPU programming with Python

Table 7 – Programming abstraction level offered by the evaluated APIs considering language and framework.

conducted analysis guided by our proposed GQM model suggests:

- **CUDA** has the harder API to encode a GPU program. Its interface requires a greater programming effort than the other tools with an expressivity where the operations are commonly encoded using several indirections. As advantages of such approach, CUDA offers great control over the code execution, high compatibility between implemented and executed code, also providing opportunities to develop optimizations. Therefore, these aspects result in a high performance.

- **OpenACC** is the easiest way to develop an application to GPU, offering a simpler approach which demands a low number of operations and indirections to express the GPU routines. OpenACC also requires low programming effort, since the parallel code is developed incrementing the serial one. Despite these advantages, OpenACC usually penalizes the performance of the applications. A reason for such fact is that the programmer does not have much resources to implement optimizations. Further, the execution of the implemented program relies on compiler decisions, a factor that reduces the compatibility between implemented and performed code.

- **Phyton/Numba** has an API that offers compatibility between source and executed code similar to CUDA, even though requiring less operations and indirections to specify the GPU routines. This is a result of the high-level approach offered by Python language. So, the programming effort demanded by Phyton/Numba is usually less than that required by CUDA. These aspects did not impacted performance significantly, since Phyton/Numba and CUDA achieved, considering the executed experiments, similar results for most of the benchmarks. The experiments also suggest which Phyton/Numba can reach performance improvements proportionally greater in comparison with CUDA for some programs. However, the Phyton/Numba results presented slowdowns for some specific applications. These penalties derived from the use of Python language and Numba environment, once the programs can take longer to execute GPU routines and the serial tasks are performed on CPU in an interpreted way. Both factors were not related to expressivity and programming effort, but they can negatively impact performance depending on the program's characteristics (e.g. applications with

routines that cannot be parallelized for GPU).

6.5 Hardware platform influence

As a complement to the results described by Section 6.3, the Appendix B presents additional performance results achieved executing the NPB versions implemented with CUDA, OpenACC and Numba on a different platform. We only executed performance experiments, since the programming expressiveness and programming effort results are not related to a hardware device.

In summary, the results illustrated by Appendix B (experiments executed on a Nvidia GTX Titan X GPU device) showed that CUDA and Numba achieved a compatible and constant performance with relation to the results described by Section 6.3. Additionally, the proceeded experiments suggest that Numba can be affected by the hardware platform since CG, LU, MG and SP presented differences comparing the experiments performed on both environments. Despite that, these distinctions did not seem to affect the gains with Python for GPUs because the reached speedups for Numba were similar to CUDA and greater than OpenACC for most of the NPB programs.

6.6 Chapter overview

This Chapter introduced experimental results regarding the comparison of programming tools to implement GPU applications. These results were obtained applying a GQM model over case studies and GPU applications developed with CUDA, OpenACC and Python/Numba, contemplating three goals of evaluation: programming expressiveness, programming effort and performance. After each goal has been addressed individually, a discussion and final analysis was presented in order to provide a general overview about the achieved results, as well as an evaluation considering the influence of the goals over each other. By the end, a comparing perspective was presented taking into account the characteristics, strengths and weaknesses of each programming tool.

7 CONCLUSION

This Chapter presents the final remarks about this Thesis, also addressing the publications proceeded during the development of this study and the future works that we foreseen to continue our researches regarding programming tools for GPUs.

7.1 Final remarks

The use of GPUs to achieve performance gains is increasing for both HPC and commercial purposes. Despite that, the encoding of programs targeting these devices is considered challenging due to the heterogeneous environment which a programmer is required to deal with during the development of a GPU code. Also, there isn't a standard framework that can be used as a "safe choice" in order to proceed an implementation targeting GPUs, a factor that can cause doubts in the process to select a programming tool. This fact is intensified since the chosen framework can impact different points of the implemented application, like performance and programming effort.

The exposed concerns about GPU programming were the focus of this Thesis. We had in mind that evaluations and comparisons between programming tools could offer a perspective about them, providing elements to support the choosing of a framework to develop a GPU program. Regarding that, this work formalized a model to evaluate and compare programming tools for GPUs. This model, based on the GQM method for software measurement, proposes to analyze distinct aspects of a framework: programming expressiveness, programming effort and performance. Each of these aspects was characterized as a goal representing a purpose of evaluation, being composed by their own questions and metrics. Although other studies have executed comparisons between frameworks for GPUs, none of them proceeded such task guided by a predefined model, since they usually employed just metrics to quantify and expose information about a programming tool.

The proposed GQM model was applied using three programming tools targeting GPUs: CUDA, OpenACC and Python/Numba. The first two frameworks were selected once they offer distinct approaches of programming (C/C++ library for CUDA and com-

piler directives for OpenACC), as well as both have official support from Nvidia to explore their GPUs. We also employed Numba as a third framework of comparison, a tool that enables GPU support in Python language and allows to encode a GPU program with pure Python code. Python is a high-level language while C/C++, which was used to explore CUDA and OpenACC as third-party libraries, is considered low-level in comparison to Python. Therefore, applying Python and C/C++ in this work offered a point of view about languages of different programming-abstraction levels (low and high).

For the experiments, we employed the set of programs from the NPB suite implemented with CUDA, OpenACC and Python/Numba. The programs with Python/Numba were encoded by us and are available at this Git[1] repository. As defined by our proposed GQM model, such experiments focused on three aspects. Firstly, it was evaluated the programming expressiveness for each of the frameworks based on case studies contemplating common GPU operations: Memory Allocation, Memory Transfer: Host to Device, Memory Transfer: Device to Host, Kernel Invocation and Using of Shared Memory. Next, the required programming effort and achieved performance for each version of the NPB programs were also gauged based on their source code and executions on a GPU device, respectively. Finally and as delineated by the GQM model, these results were discussed and analyzed to provide a perspective about the frameworks considering their characteristics, strengths and weaknesses. This discussion also contemplated the influence between the evaluated aspects over each other.

In summary, the executed experiments, evaluations and comparisons suggested that CUDA has an API which offers great expressivity, resulting in great control over the executed code, opportunities to encode optimizations and high performance. However, these advantages also require high programming effort load, since the GPU routines are usually implemented using several operations and indirections. On the other hand, OpenACC provides an API demanding less lines of code with operations related to GPU than the other analyzed frameworks. This happens due to use of an approach relying on compiler directives. Regardless that, the programmer doesn't have much features to implement optimizations, as well as the OpenACC code depends on compiler decisions to be executed on GPU. These factors usually contributed to add performance penalties to the applications which achieved the worst results for most cases. By the end, Python/Numba offers similar expressivity than CUDA considering the resources to specify GPU routines and to implement optimizations in a source code. As an advantage, Python/Numba implementations require a lower programming effort in comparison with CUDA as a result of the Python's high-level language. The performance delivered for most NPB programs also achieved an equivalent level than CUDA. Nevertheless, the use of Python/Numba presented performance slowdowns for a part

[1] NPB with PYTHON: https://github.com/danidomenico/NPB-PYTHON

of the benchmarks, some of them caused by the characteristics of Python language and Numba environment.

The main and the specific objectives delineated for this Thesis could be achieved during the development of this study. Firstly, we defined the proposed GQM model accomplishing the **SO1**. With the implementation of NPB with Python (serial and GPU versions), we also reached **SO2**. Applying the outputs from **SO1** and **SO2**, we were able to conduct experiments regarding programming expressiveness, programming effort and performance. Therefore, we proceeded evaluations and comparisons between the frameworks (CUDA, OpenACC and Python/Numba), leading us to achieve **SO3**. By the end, **SO4** was reached presenting a perspective contemplating each of the three frameworks. This perspective was defined using the experimental results related to **SO3**.

We suppose that our Thesis contributed to the research of programming tools for GPUs since it offers a model to guide evaluations and comparisons of frameworks. The executed comparisons were employed to formulate a perspective considering the characteristics, strengths and weaknesses of a programming tool. Besides, the presented experimental results improve the knowledge about CUDA, OpenACC and Python/Numba, providing resources that can be applied to assist the processes to choose a programming tool to implement GPU applications. Another applicability for this study is related to the improvement of the frameworks, since the results achieved guided by the proposed model allow a better understanding about the weak points of a tool. Thus, such points can be advanced, not only to optimize a framework, but also be considered during the design of new programming tools targeting GPUs.

7.2 Publications

During the term applied to develop this Thesis, we proceeded some publications regarding APIs for parallel programming, as well as comparisons between frameworks for GPUs. The published papers are enumerated as follows:

1- Daniel Di Domenico and Gerson Geraldo H. Cavalheiro. **JAMPI: A C++ Parallel Programming Interface Allowing the Implementation of Custom and Generic Scheduling Mechanisms**, 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2020, pp. 273-280.

- This study introduces the design and implementation of JAMPI, a generic parallel programming interface focused on code reuse, productivity and high-level abstraction to enable the construction of parallel applications targeting multicore platforms. With this work, we delved deeper into the concepts

of programming tools for parallel environments, a fact that contributed to developing this Thesis.

2- Andre D. Jardim, Kevin Oliveira, Diogo J. Cardoso, Daniel Di Domenico, Andre R. Du Bois, and Gerson Geraldo H. Cavalheiro. **An extension for Transactional Memory in OpenMP**, 25th Brazilian Symposium on Programming Languages (SBLP'21), 2021, pp. 58–65.

- In this work, we applied the GQM method for software measurement to analyze OpenMP codes aiming to evaluate distinct extensions of transactional memories, including one that had been proposed by ourselves. Hence, this work was our first experience dealing with the GQM method to evaluate and compare parallel programming tools.

3- Daniel Di Domenico, Gerson Geraldo H. Cavalheiro and João V. F. Lima. **NAS Parallel Benchmark Kernels with Python: A and programming effort analysis focusing on GPUs**, 2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2022, pp. 26-33.

- This paper presents the implementation of the 5 NPB kernels with Python targeting CPU (serial) and GPU. We executed performance and programming effort experiments using these benchmarks, comparing the achieved results to other GPU versions developed with CUDA and OpenACC. This manuscript is the source of some results that were employed by this Thesis to analyze frameworks for GPUs.

4- Daniel Di Domenico, João V. F. Lima and Gerson Geraldo H. Cavalheiro. **NAS Parallel Benchmarks with Python: A performance and programming effort analysis focusing on GPUs**, The Journal of Supercomputing (*submitted*).

- This study is an extended version of paper number 3 published in PDP 2022. As original content, we included the implementation of three NPB applications with Python (both serial and GPU versions). We also conducted experiments with CUDA, OpenACC and Python considering performance and programming effort. The results of such experiments were also presented by this Thesis.

7.3 Future works

As future works, we can focus on different topics intending to improve the current results, also defining more resources to complement the objectives delineated to this

study. Regarding the proposed GQM model, new questions and metrics can be formulated to advance on the evaluations about programming tools for GPUs. For example, a new metric could be added to measure specific tasks related to the execution of a GPU program, such as (1) the cost of data transfers between memory nodes and (2) the execution time of the main GPU kernels of the application. These metrics could be used to identify some bottleneck that penalizes performance. Besides, the programming effort evaluation can be improved including a metric to compute the indirections of an operation, being possible to differentiate frameworks which require the same number of operations but a distinct number of indirections.

Another future assignment that we foreseen is to perform experiments using other programming tools. The proposed GQM model was designed to evaluate any framework which targets GPUs. It just requires that applications are implemented employing this tool since these programs are demanded to proceed the programming effort and performance evaluations. Hence, a possibility is to execute experiments employing the frameworks listed on Topic 2.2.6. These tools offer distinct programming resources in comparison to the ones we have applied on our experiments. We seem that experiments guided by our model can be useful to offer a point of view about these other frameworks, not just to evaluate and compare them, but also to provide a better understanding about how they can be improved.

Lastly, we also have in mind to execute evaluations with CUDA, OpenACC and Python/Numba using applications from other benchmark suites focused on GPUs. Microbenchmarks could also be employed on these new experiments. They will generate more information about each of the frameworks, complementing the analysis performed about them. Such factor could improve the presented perspective regarding programming expressiveness, programming effort and performance aspects.

REFERENCES

ABE, Y. et al. Power and Performance Analysis of GPU-Accelerated Systems. In: WORKSHOP ON POWER-AWARE COMPUTING AND SYSTEMS (HOTPOWER 12), 2012., 2012, Hollywood, CA. **Proceedings...** USENIX Association, 2012.

ADCOCK, A. B.; SULLIVAN, B. D.; HERNANDEZ, O. R.; MAHONEY, M. W. Evaluating OpenMP Tasking at Scale for the Computation of Graph Hyperbolicity. In: OpenMP IN THE ERA OF LOW POWER DEVICES AND ACCELERATORS, 2013, Canberra, ACT, Australia. **Proceedings...** Springer Berlin Heidelberg, 2013. p.71–83. (9th International Workshop on OpenMP, IWOMP 2013).

AL-NANIH, R.; AL-NUAIM, H.; ORMANDJIEVA, O. New Health Information Systems (HIS) Quality-in-Use Model Based on the GQM Approach and HCI Principles. In: INTERNATIONAL CONFERENCE ON HUMAN-COMPUTER INTERACTION. PART IV: INTERACTING IN VARIOUS APPLICATION DOMAINS, 13., 2009, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2009. p.429–438.

ANDRADE, H.; LWAKATARE, L. E.; CRNKOVIC, I.; BOSCH, J. Software Challenges in Heterogeneous Computing: A Multiple Case Study in Industry. In: EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS (SEAA), 2019., 2019. **Proceedings...** [S.l.: s.n.], 2019. p.148–155.

ARAUJO, G. A. d.; GRIEBLER, D.; DANELUTTO, M.; FERNANDES, L. G. Efficient NAS Parallel Benchmark Kernels with CUDA. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2020., 2020. **Proceedings...** [S.l.: s.n.], 2020. p.9–16.

ARAUJO, G. et al. NAS Parallel Benchmarks with CUDA and beyond. **Software: Practice and Experience**, [S.l.], p.1–28, 2021.

ASKARBEKULY, N.; SADOVYKH, A.; MAZZARA, M. Combining Two Modelling Approaches: GQM and KAOS in an Open Source Project. In: OPEN SOURCE SYSTEMS, 2020, Cham. **Proceedings...** Springer International Publishing, 2020. p.106–119.

ASTORGA, D. d. R.; DOLZ, M. F.; FERNÁNDEZ, J.; SÁNCHEZ, J. D. G. Supporting Advanced Patterns in GrPPI, a Generic Parallel Pattern Interface. In: EURO-PAR WORKSHOPS, 2017. **Proceedings...** [S.l.: s.n.], 2017.

AUGONNET, C.; THIBAULT, S.; NAMYST, R.; WACRENIER, P.-A. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. **Concurr. Comput. : Pract. Exper.**, Chichester, UK, v.23, n.2, p.187–198, Feb. 2011.

BAILEY, D. H. et al. **The NAS Parallel Benchmarks RNR-94-007**. [S.l.]: NASA Advanced Supercomputing Division, 1994.

BALDINI, I.; FINK, S. J.; ALTMAN, E. Predicting GPU Performance from CPU Runs Using Machine Learning. In: IEEE 26TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2014., 2014. **Proceedings...** [S.l.: s.n.], 2014. p.254–261.

BASILI, V. R. **Software Modeling and Measurement**: The Goal/Question/Metric Paradigm. USA: [s.n.], 1992.

BASILI, V. R.; CALDIERA, G.; ROMBACH, D. H. **The Goal Question Metric Approach**. [S.l.]: John Wiley & Sons, 1994. v.I.

BEHNEL, S.; BRADSHAW, R. W.; SELJEBOTN, D. S. Cython tutorial. In: PYTHON IN SCIENCE CONFERENCE, 8., 2009, Pasadena, CA USA. **Proceedings...** [S.l.: s.n.], 2009. p.4 – 14.

BERANDER, P.; JöNSSON, P. A Goal Question Metric Based Approach for Efficient Measurement Framework Definition. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING, 2006., 2006, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2006. p.316–325. (ISESE '06).

BRODTKORB, A. R.; HAGEN, T. R.; SÆTRA, M. L. Graphics processing unit (GPU) programming strategies and trends in GPU computing. **Journal of Parallel and Distributed Computing**, [S.l.], v.73, n.1, p.4–13, 2013. Metaheuristics on GPUs.

BUENO, J. et al. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 27., 2013, Eugene, Oregon, USA. **Proceedings...** ACM, 2013. p.359–368. (ICS '13).

CARVALHO, P. et al. Kernel concurrency opportunities based on GPU benchmarks characterization. **Cluster Computing**, [S.l.], v.23, 03 2020.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP**: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). Cambridge, MA, USA: The MIT Press, 2007.

CHEN, Y.; CUI, X.; MEI, H. PARRAY: A Unifying Array Representation for Heterogeneous Parallelism. In: ACM SIGPLAN - SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 17., 2012, New York, NY, USA. **Proceedings...** ACM, 2012. p.171–180. (PPoPP '12).

CHRISTGAU, S. et al. A comparison of CUDA and OpenACC: Accelerating the Tsunami Simulation EasyWave. In: ARCS 2014; 2014 WORKSHOP PROCEEDINGS ON ARCHITECTURE OF COMPUTING SYSTEMS, 2014. **Proceedings...** [S.l.: s.n.], 2014. p.1–5.

Clang 12 Documentation. **OpenMP Support**. Available at: <https://clang.llvm.org/docs/OpenMPSupport.html>.

CUDA C++ Programming Guide. **Version 11.2.1. Nvidia. 2021**.

CuPy API Reference. **Version 11.4. Preferred Infrastructure, Inc. and Preferred Networks, Inc.**

DANTSIN, E.; EITER, T.; GOTTLOB, G.; VORONKOV, A. Complexity and Expressive Power of Logic Programming. **ACM Comput. Surv.**, New York, NY, USA, v.33, n.3, p.374–425, sep 2001.

DONGARRA, J. et al. (Ed.). **Sourcebook of Parallel Computing**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

EDWARDS, H. C. et al. Manycore performance-portability: Kokkos multidimensional array library. **Scientific Programming**, [S.l.], v.20, n.2, p.89–114, 2012.

ENMYREN, J.; KESSLER, C. W. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In: FOURTH INTERNATIONAL WORKSHOP ON HIGH-LEVEL PARALLEL PROGRAMMING AND APPLICATIONS, 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.5–14. (HLPP '10).

ERGASHEVA, S.; KRUGLOV, A.; SHULHAN, I. Development and evaluation of GQM method to improve adaptive systems. In: ITTCS, 2019. **Proceedings...** [S.l.: s.n.], 2019.

ESMAEILZADEH, H. et al. Power Challenges May End the Multicore Era. **Commun. ACM**, New York, NY, USA, v.56, n.2, p.93–102, Feb. 2013.

FARMER, W. M. Chiron: A Multi-Paradigm Logic. In: 2007. **Proceedings...** [S.l.: s.n.], 2007.

FELLEISEN, M. On the Expressive Power of Programming Languages. In: SCIENCE OF COMPUTER PROGRAMMING, 1990. **Proceedings...** Springer-Verlag, 1990. p.134–151.

FELLEISEN, M. On the expressive power of programming languages. **Science of Computer Programming**, [S.l.], v.17, n.1, p.35–75, 1991.

FENG, W. chun; MANOCHA, D. High-performance computing using accelerators. **Parallel Computing**, [S.l.], v.33, n.10–11, p.645 – 647, 2007. High-Performance Computing Using Accelerators.

FENTON, N. E.; BIEMAN, J. **Software Metrics: A Rigorous and Practical Approach**. 3.ed. [S.l.]: CRC Press, 2014. (Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series).

FUGGETTA, A. et al. Applying GQM in an Industrial Software Factory. **ACM Trans. Softw. Eng. Methodol.**, New York, NY, USA, v.7, n.4, p.411–448, oct 1998.

GAUTIER, T.; LIMA, J. V. F.; MAILLARD, N.; RAFFIN, B. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In: IEEE 27TH INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, 2013., 2013, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2013. p.1299–1308. (IPDPS '13).

GIMENES, T. L.; PISANI, F.; BORIN, E. Evaluating the Performance and Cost of Accelerating Seismic Processing with CUDA, OpenCL, OpenACC, and OpenMP. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS), 2018., 2018. **Proceedings...** [S.l.: s.n.], 2018. p.399–408.

GOWDA, K. N.; RAMAPRASAD, H. Dynamic schedule management framework for aperiodic soft-real-time jobs on GPU based architectures. In: IEEE INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE AND SYSTEMS (ICESS), 2020., 2020. **Proceedings...** [S.l.: s.n.], 2020. p.1–10.

GUO, X.; WU, J.; WU, Z.; HUANG, B. Parallel Computation of Aerial Target Reflection of Background Infrared Radiation: Performance Comparison of OpenMP, OpenACC, and CUDA Implementations. **IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing**, [S.l.], v.9, n.4, p.1653–1662, 2016.

HAIDL, M.; GORLATCH, S. PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14. In: LLVM COMPILER INFRASTRUCTURE IN

HPC, 2014., 2014, Piscataway, NJ, USA. **Proceedings...** IEEE Press, 2014. p.1–11. (LLVM-HPC '14).

HERNANDES, E.; ZAMBONI, A.; FABBRI, S.; THOMMAZO, A. di. Using GQM and TAM to evaluate StArt – a tool that supports Systematic Review. **CLEI Electronic Journal**, [S.l.], v.15, 04 2012.

HIGUERA, N. N. R. **On the expressiveness of LARA**: a unified language for linear and relational algebra. Santiago, Chile: [s.n.], 2019.

HOLM, H. H.; BRODTKORB, A. R.; SæTRA, M. L. GPU Computing with Python: Performance, Energy Efficiency and Usability. **Computation**, [S.l.], v.8, n.1, 2020.

HOSHINO, T.; MARUYAMA, N.; MATSUOKA, S.; TAKAKI, R. CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD, AND GRID COMPUTING, 2013., 2013. **Proceedings...** [S.l.: s.n.], 2013. p.136–143.

HUSSAIN, A.; FERNELEY, E. Usability Metric for Mobile Application: A Goal Question Metric (GQM) Approach. In: INTERNATIONAL CONFERENCE ON INFORMATION INTEGRATION AND WEB-BASED APPLICATIONS AND SERVICES, 10., 2008, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2008. p.567–570. (iiWAS '08).

JARDIM, A. D. et al. An Extension for Transactional Memory in OpenMP. In: BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 25., 2021, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2021. p.58–65. (SBLP'21).

JONES, C. **Software Engineering Best Practices**. 1.ed. New York, NY, USA: McGraw-Hill, Inc., 2010.

KINDRATENKO, V. V. et al. GPU clusters for high-performance computing. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING AND WORKSHOPS, 2009., 2009. **Proceedings...** [S.l.: s.n.], 2009. p.1–8.

KIRK, D. B.; HWU, W. W. **Programming Massively Parallel Processors**: A Hands-on Approach. 2nd.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

KLöCKNER, A. et al. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. **Parallel Computing**, [S.l.], v.38, n.3, p.157–174, 2012.

KUAN, L. et al. Accelerating Phylogenetic Inference on GPUs: an OpenACC and CUDA comparison. In: INTERNATIONAL WORK-CONFERENCE ON BIOINFORMATICS AND BIOMEDICAL ENGINEERING, IWBBIO 2014, GRANADA, SPAIN, APRIL 7-9, 2014, 2014. **Proceedings...** Copicentro Editorial, 2014. p.589–600.

LI, K. **OpenMP Accelerator Support for GPUs. 2017**. Available at: <https://www.openmp.org/updates/openmp-accelerator-support-gpus>.

LI, L.; KESSLER, C. VectorPU: A Generic and Efficient Data-container and Component Model for Transparent Data Transfer on GPU-based Heterogeneous Systems. In: WORKSHOP AND 6TH WORKSHOP ON PARALLEL PROGRAMMING AND RUNTIME MANAGEMENT TECHNIQUES FOR MANY-CORE ARCHITECTURES AND DESIGN TOOLS AND ARCHITECTURES FOR MULTICORE EMBEDDED COMPUTING PLATFORMS, 8., 2017, New York, NY, USA. **Proceedings...** ACM, 2017. p.7–12. (PARMA-DITAM '17).

LI, X. et al. Comparing Programmer Productivity in OpenACC and CUDA: an Empirical Investigation. **International Journal of Computer Science, Engineering and Applications**, [S.l.], v.6, p.1–15, 2016.

LI, X.; SHIH, P.-C. An Early Performance Comparison of CUDA and OpenACC. **MATEC Web Conf.**, [S.l.], v.208, p.05002, 2018.

LIM, J.; KIM, H. Design Space Exploration of Memory Model for Heterogeneous Computing. In: IEEE 26TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2014., 2014. **Proceedings...** [S.l.: s.n.], 2014. p.160–167.

LIMA, J. V. F.; BROQUEDIS, F.; GAUTIER, T.; RAFFIN, B. Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2013., 2013. **Proceedings...** [S.l.: s.n.], 2013. p.105–112.

LIMA, J. V. F.; DI DOMENICO, D. HPSM: A Programming Framework for Multi-CPU and Multi-GPU Systems. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING WORKSHOPS (SBAC-PADW), 2017., 2017. **Proceedings...** [S.l.: s.n.], 2017. p.31–36.

LIMA, J. V. F.; DI DOMENICO, D. HPSM: a programming framework to exploit multi-CPU and multi-GPU systems simultaneously. **International Journal of Grid and Utility Computing**, [S.l.], v.10, p.201, 01 2019.

LöFF, J. et al. The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. **Future Generation Computer Systems**, [S.l.], v.125, p.743–757, 2021.

MALIK, M. et al. Productivity of GPUs under different programming paradigms. **Concurrency and Computation: Practice and Experience**, [S.l.], v.24, 2012.

MANAVSKI, S. A. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In: IEEE INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING AND COMMUNICATIONS, 2007., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.65–68.

MANN, A. Core Concept: Nascent exascale supercomputers offer promise, present challenges. **Proceedings of the National Academy of Sciences**, [S.l.], v.117, n.37, p.22623–22625, 2020.

MEMETI, S. et al. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption. In: WORKSHOP ON ADAPTIVE RESOURCE MANAGEMENT AND SCHEDULING FOR CLOUD COMPUTING, 2017., 2017, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2017. p.1–6. (ARMS-CC '17).

MENDONCA, G. S. D. et al. Automatic Insertion of Copy Annotation in Data-Parallel Programs. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD), 2016., 2016. **Proceedings...** [S.l.: s.n.], 2016. p.34–41.

MENG, Z.; ZHANG, C.; SHEN, B.; WEI, Y. A GQM-based Approach for Software Process Patterns Recommendation. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, 29., 2017. **Proceedings...** [S.l.: s.n.], 2017. p.370–375.

NOAJE, G.; JAILLET, C.; KRAJECKI, M. Source-to-Source Code Translator: OpenMP C to CUDA. In: IEEE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS, 2011., 2011. **Proceedings...** [S.l.: s.n.], 2011. p.512–519.

Numba Documentation. **Version 0.50. Anaconda, Inc. and others**.

NumPy Documentation. **Version 1.21. The NumPy community**.

ODEN, L. Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2020., 2020. **Proceedings...** [S.l.: s.n.], 2020. p.216–223.

OpenACC Specification. **Version 3.1. OpenACC.org. 2020**.

OpenMP Specification. **Version 5.1. The OpenMP Architecture Review Board. 2020**.

RAMADAN, N.; ABD ELLATIF, M. Adaptation of GQM Method for Evaluating the Performance of Software Project Manager. **International Journal of Computer Science and Information Security,**, [S.l.], v.8, p.304–307, 01 2010.

SAINZ, F. et al. Leveraging OmpSs to Exploit Hardware Accelerators. In: IEEE 26TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2014., 2014, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2014. p.112–119. (SBAC-PAD '14).

SANDERS, J.; KANDROT, E. **CUDA by Example**: An Introduction to General-Purpose GPU Programming. 1st.ed. [S.l.]: Addison-Wesley Professional, 2010.

SEO, S.; JO, G.; LEE, J. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In: IEEE INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION (IISWC), 2011., 2011. **Proceedings...** [S.l.: s.n.], 2011. p.137–148.

SETIAWAN, R.; RASJID, Z. E.; EFFENDI, A. Design Metric Indicator to Improve Quality Software Development (Study Case: Student Desk Portal). **Procedia Computer Science**, [S.l.], v.135, p.616–623, 2018. The 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life.

SOLINGEN, R. van; BERGHOUT, E. Integrating goal-oriented measurement in industrial software engineering: industrial experiences with and additions to the Goal/Question/Metric method (GQM). In: SEVENTH INTERNATIONAL SOFTWARE METRICS SYMPOSIUM, 2001. **Proceedings...** [S.l.: s.n.], 2001. p.246–258.

The OpenCL Specification. **Version 2.2. Khronos OpenCL Working Group. 2019**.

Thrust. **What is Thrust?** Available at: <https://thrust.github.io>.

WEBER, R.; GOTHANDARAMAN, A.; HINDE, R.; PETERSON, G. Comparing Hardware Accelerators in Scientific Applications: A Case Study. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.22, n.1, p.58–68, Jan 2011.

WYNTERS, E. Fast and Easy Parallel Processing on GPUs Using C++ AMP. **J. Comput. Sci. Coll.**, Evansville, IN, USA, v.31, n.6, p.27–33, jun 2016.

XU, R. et al. NAS Parallel Benchmarks for GPGPUs Using a Directive-Based Programming Model. In: LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING, 2015, Cham. **Proceedings...** Springer Int. Publishing, 2015. p.67–81.

YOUSSEF, C. K. et al. GQM-Based Tree Model for Automatic Recommendation of Design Pattern Category. In: INTERNATIONAL CONFERENCE ON SOFTWARE AND

INFORMATION ENGINEERING (ICSIE), 2020., 2021, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2021. p.126–130. (ICSIE 2020).

ZIOGAS, A. N.; BEN-NUN, T.; SCHNEIDER, T.; HOEFLER, T. NPBench: A Benchmarking Suite for High-Performance NumPy. In: ACM INT. CONF. ON SUPERCOMPUTING, 2021, New York, NY, USA. **Proceedings...** ACM, 2021. p.63–74. (ICS '21).

Appendices

APPENDIX A – Statistical analysis over performance results

This Appendix focuses on providing information regarding a statistical analysis executed over the performance results presented by Section 6.3. The objective here is to ensure a confidence of at least 95% to the experimental data set. The versions of programs contemplated by this Appendix are:

- **C++**: serial C++ code.
- **Python**: serial Python code.
- **CUDA**: GPU code developed with C++ and CUDA.
- **OpenACC**: GPU code developed with C applying OpenACC directives targeting Nvidia GPUs.
- **Numba**: GPU code implemented with Python and CUDA support enabled with Numba environment.

To accomplish the goal for this Appendix, we firstly proceeded the Kolmogorov–Smirnov Test (KS) aiming to check whether the samples are normally distributed. Table 8 depicts the results achieved for each of the executed versions and input sizes including mean, standard deviation (STD) and KS Test. If column **KS p-value** displays a value greater than 0.05, the sample is normally distributed considering a significance level of 95%. The non-distributed samples are highlighted with red in Table 8.

Most of the samples showed by Table 8 are normally distributed. Just CG (Python for class B), EP (C++ for class B and Numba for classes B and C), FT and MG (both with CUDA for class C) kernels presented some versions where the samples did not achieve a normal distribution.

The second statistical analysis proceeded aimed to validate the significance of comparisons between different versions of the same benchmark. For that, we applied the Student's T-Test (or just T-Test) and, for non-distributed samples, the Mann–Whitney U-Test (or just U-Test). Table 9 shows data contemplating T-Test and U-Test values for each of the GPU versions compared during the performance experiments. If the displayed number in column **p-value** has a value less than 0.05, the comparison between Version A and Version B have a significance level of 95% to occur by chance.

NPB Application	Version	Kolmogorov–Smirnov Test for each input size									
		Class B					**Class C**				
		Sample size	Mean	STD	KS p-value	**Normally distrib.?**	Sample size	Mean	STD	KS p-value	**Normally distrib.?**
BT	C++	30	281.101	1.333	0.344	✓	30	1,167.523	9.371	0.187	✓
	Python	30	366.054	2.258	0.179	✓	30	1,507.663	7.126	0.694	✓
	CUDA	30	3.445	0.002	0.508	✓	30	14.158	0.003	0.546	✓
	OpenACC	30	11.379	0.021	0.336	✓	30	47.788	0.033	0.201	✓
	Numba	30	6.078	0.002	0.244	✓	30	22.099	0.003	0.838	✓
CG	C++	30	112.241	0.414	0.095	✓	30	318.593	0.459	0.407	✓
	Python	30	163.602	1.728	0.004	–	30	427.222	6.426	0.143	✓
	CUDA	30	0.757	0.005	0.451	✓	30	1.824	0.005	0.373	✓
	OpenACC	30	0.989	0.012	0.113	✓	30	2.055	0.004	0.997	✓
	Numba	30	2.879	0.050	0.203	✓	30	4.315	0.028	0.879	✓
EP	C++	30	142.433	0.122	0.001	–	30	569.268	0.194	0.395	✓
	Python	30	129.839	0.245	0.398	✓	30	518.092	0.854	0.102	✓
	CUDA	30	0.101	0.001	0.203	✓	30	0.384	0.000	0.839	✓
	OpenACC	30	0.669	0.002	0.052	✓	30	2.543	0.003	0.084	✓
	Numba	30	0.108	0.001	0.007	–	30	0.358	0.002	0.017	–
FT	C++	30	125.279	3.442	0.211	✓	30	372.670	1.783	0.819	✓
	Python	30	151.234	2.753	0.395	✓	30	622.651	5.398	0.153	✓
	CUDA	30	1.439	0.002	0.835	✓	30	6.399	0.005	0.030	–
	OpenACC	30	4.589	0.018	0.265	✓	30	20.253	0.070	0.063	✓
	Numba	30	1.484	0.003	0.563	✓	30	6.354	0.003	0.593	✓
IS	C++	30	4.354	0.050	0.239	✓	30	19.381	0.144	0.284	✓
	Python	30	7.510	0.017	0.670	✓	30	30.977	0.019	0.366	✓
	CUDA	30	0.052	0.000	0.988	✓	30	0.710	0.000	0.851	✓
	Numba	30	0.053	0.000	0.457	✓	30	0.711	0.000	0.932	✓
LU	C++	30	209.341	0.717	0.554	✓	30	1,008.549	2.974	0.629	✓
	Python	30	246.204	1.200	0.769	✓	30	1,141.680	3.324	0.981	✓
	CUDA	30	2.104	0.005	0.525	✓	30	11.842	0.008	0.927	✓
	OpenACC	30	17.451	0.144	0.449	✓	30	59.012	0.178	0.815	✓
	Numba	30	33.738	0.437	0.727	✓	30	53.972	0.832	0.984	✓
MG	C++	30	5.759	0.028	0.852	✓	30	51.517	0.110	0.115	✓
	Python	30	20.457	0.102	0.727	✓	30	162.779	0.750	0.311	✓
	CUDA	30	0.087	0.002	0.308	✓	30	0.914	0.001	0.008	–
	OpenACC	30	0.260	0.000	0.855	✓	30	2.096	0.001	0.788	✓
	Numba	30	0.633	0.009	0.578	✓	30	1.217	0.011	0.686	✓
SP	C++	30	208.364	0.582	0.889	✓	30	843.760	3.075	0.879	✓
	Python	30	229.561	0.649	0.882	✓	30	934.839	2.349	0.874	✓
	CUDA	30	1.786	0.001	0.190	✓	30	8.400	0.003	0.179	✓
	OpenACC	30	6.590	0.035	0.377	✓	30	31.955	0.058	0.750	✓
	Numba	30	1.975	0.002	0.680	✓	30	7.971	0.009	0.246	✓

Table 8 – Kolmogorov–Smirnov Test for performance results (significance level of 95%).

All the different samples compared during our experiments are significant according to the applied T and U tests. Hence, the comparisons have a high probability to express reality.

NPB Application	Version A	Version B	Significance test comparing two samples (versions) for each input size					
			Class B			**Class C**		
			Test	p-value	**Significant?**	Test	p-value	**Significant?**
BT	C++	CUDA	T-Test	8.464e-069	✓	T-Test	3.563e-062	✓
	C++	OpenACC	T-Test	1.843e-068	✓	T-Test	8.381e-062	✓
	Python	Numba	T-Test	1.967e-065	✓	T-Test	8.199e-069	✓
	CUDA	OpenACC	T-Test	1.070e-077	✓	T-Test	1.868e-090	✓
	CUDA	Numba	T-Test	1.437e-154	✓	T-Test	2.743e-177	✓
	OpenACC	Numba	T-Test	3.391e-073	✓	T-Test	1.527e-086	✓
CG	C++	CUDA	T-Test	4.552e-072	✓	T-Test	6.629e-084	✓
	C++	OpenACC	T-Test	3.97e-072	✓	T-Test	6.868e-084	✓
	Python	Numba	U-Test	2.872e-011	✓	T-Test	2.708e-054	✓
	CUDA	OpenACC	T-Test	1.163e-047	✓	T-Test	4.539e-079	✓
	CUDA	Numba	T-Test	2.041e-049	✓	T-Test	2.482e-061	✓
	OpenACC	Numba	T-Test	2.778e-051	✓	T-Test	5.295e-059	✓
EP	C++	CUDA	U-Test	2.872e-011	✓	T-Test	4.233e-102	✓
	C++	OpenACC	U-Test	2.872e-011	✓	T-Test	4.265e-102	✓
	Python	Numba	U-Test	2.868e-011	✓	U-Test	2.872e-011	✓
	CUDA	OpenACC	T-Test	5.520e-079	✓	T-Test	1.076e-084	✓
	CUDA	Numba	U-Test	2.868e-011	✓	U-Test	2.872e-011	✓
	OpenACC	Numba	U-Test	2.868e-011	✓	U-Test	2.872e-011	✓
FT	C++	CUDA	T-Test	1.085e-046	✓	U-Test	2.872e-011	✓
	C++	OpenACC	T-Test	2.280e-046	✓	T-Test	2.556e-068	✓
	Python	Numba	T-Test	6.798e-052	✓	T-Test	3.150e-061	✓
	CUDA	OpenACC	T-Test	7.601e-068	✓	U-Test	2.872e-011	✓
	CUDA	Numba	T-Test	3.294e-045	✓	U-Test	2.872e-011	✓
	OpenACC	Numba	T-Test	1.975e-070	✓	T-Test	2.231e-068	✓
IS	C++	CUDA	T-Test	9.861e-058	✓	T-Test	8.609e-063	✓
	Python	Numba	T-Test	2.549e-078	✓	T-Test	1.015e-094	✓
	CUDA	Numba	T-Test	3.032e-079	✓	T-Test	1.628e-071	✓
LU	C++	CUDA	T-Test	6.329e-073	✓	T-Test	8.625e-075	✓
	C++	OpenACC	T-Test	7.631e-077	✓	T-Test	1.241e-074	✓
	Python	Numba	T-Test	5.529e-081	✓	T-Test	2.425e-082	✓
	CUDA	OpenACC	T-Test	1.767e-060	✓	T-Test	4.412e-072	✓
	CUDA	Numba	T-Test	1.720e-055	✓	T-Test	5.258e-051	✓
	OpenACC	Numba	T-Test	1.144e-054	✓	T-Test	1.222e-025	✓
MG	C++	CUDA	T-Test	2.575e-069	✓	U-Test	2.872e-011	✓
	C++	OpenACC	T-Test	2.893e-068	✓	T-Test	1.531e-078	✓
	Python	Numba	T-Test	8.497e-069	✓	T-Test	2.988e-069	✓
	CUDA	OpenACC	T-Test	2.570e-060	✓	U-Test	2.872e-011	✓
	CUDA	Numba	T-Test	7.083e-059	✓	U-Test	2.872e-011	✓
	OpenACC	Numba	T-Test	3.895e-049	✓	T-Test	8.941e-058	✓
SP	C++	CUDA	T-Test	1.657e-075	✓	T-Test	3.815e-072	✓
	C++	OpenACC	T-Test	1.117e-075	✓	T-Test	7.913e-072	✓
	Python	Numba	T-Test	2.281e-075	✓	T-Test	7.548e-077	✓
	CUDA	OpenACC	T-Test	1.318e-063	✓	T-Test	1.243e-077	✓
	CUDA	Numba	T-Test	1.524e-095	✓	T-Test	6.546e-062	✓
	OpenACC	Numba	T-Test	3.171e-063	✓	T-Test	2.311e-080	✓

Table 9 – Significance test comparing versions from performance results. (significance level of 95%)

APPENDIX B – Additional performance experiments

This Appendix presents results regarding additional performance experiments on a new platform with GPU. Such results offer supplementary information to complement the performance evaluation presented by Section 6.3.

We proceeded experiments on a machine composed of one Intel Xeon E3-1230V2 processor equipped with 4 cores (8 threads contexts) running at 3.3 GHz and 24 GB main memory. The machine is enhanced with one Nvidia GTX Titan X GPU with 3,072 CUDA cores at 1,000 MHz and 12 GB of GDDR5 memory. As faced in Section 6.3, our experiments were restricted to 1 CPU core and 1 GPU without CPU parallelism. The software environment used was Ubuntu 21.10 operating system, CUDA 11.7, GCC 11.2.0, PGCC 22.5, OpenACC 2.7, Python 3.8.8, Numba 0.53.1 and LLVM 10.0.1.

From now on, this Appendix refers to the current machine as **USP**, while the platform employed in Section 6.3 will be referenced as **UFSM**. Comparing both environments, we perceive as main distinctions an increase in the frequency of CPUs and a decrease in the frequency of the GPU. Besides, there are variances in the number of available cores for processing. Table 10 shows an outline pointing the differences between both platforms.

	Platform	
Configuration	USP	UFSM
CPU device	Intel Xeon E3-1230V2	Intel Xeon E5-2420
CPU cores	4	6 (x2)
CPU freq.	3.3 GHz	1.9 GHZ
Main memory	24 GB	80 GB
GPU device	Nvidia Titan V	Nvidia GTX Titan X
GPU cores	3,072	5,120
GPU freq.	1,000 MHz	1,200 MHz
GPU memory	12 GB GDDR5	12 GB HBM2
Oper. system	Ubuntu 21.10	Debian 10
CUDA	11.7	11.2
GCC	11.2.0	8.3.0
PGCC	22.5	21.2

Table 10 – Additional experiments: Configuration comparison between applied platforms.

The versions of NPB programs contemplated by this Appendix are the same used in Section 6.3, as follows:

- **C++**: serial C++ code compiled with GCC using `-O3` optimization flag.
- **Python**: serial Python code optimized employing Numba environment.
- **CUDA**: GPU code developed with C++ and CUDA compiled by NVCC applying the `-O3` optimization flag.
- **OpenACC**: GPU code developed with C applying OpenACC directives targeting Nvidia GPUs. The compilation was proceeded using PGCC with `-O3 -mcmodel=medium` flags. Executing this version required the command `ulimit -s unlimited` to increase the memory available in the stack.
- **Numba**: GPU code implemented with Python and CUDA support enabled with the Numba environment.

The input sizes employed for the experiments were workload classes B and C offered by the NPB suit. We executed the GPU versions on **USP** with the same configurations used on **UFSM** regarding number of blocks and threads to invoke the GPU kernels.

B.1 Results

The results presented by this Section are related to Goal **GO3** of the proposed GQM model. Firstly, Figure 16 depicts the achieved values for the *TGPU* metric aiming to answer question **Q3.1** (GPU execution time) of the model.

For most of the benchmarks showed in Figure 16, the *TGPU* values obtained from executions on **USP** follow the same pattern of **UFSM**, that is, CUDA and Numba achieved similar results as well as both versions outperformed OpenACC. However, some differences in results can be highlighted. The *TGPU* values for **USP** are slower than **UFSM** ones as expected since the GPU that equips **USP** has a lower power than the ones available in **UFSM**. Other perceived variances are related to Numba versions, specially to LU and SP applications. The **USP** execution of LU achieved a better performance with Numba than with CUDA for class C. On **UFSM**, the CUDA version had performed faster. The **USP** environment, which provides a CPU with greater power and a GPU with less capacity than **UFSM**, seems to improve Numba performance for the LU benchmark. A reason that can explain such fact is the increasing of GPU utilization for Numba on **USP**, once the execution of non-accelerated codes on the CPU by Python were probably faster than on **UFSM**. So, the total time spent by LU on the CPU was lower. An equivalent trend can be seen for CG and MG. Both kernels were also affected by a low GPU utilization on **UFSM** and reached an improved performance on **USP**. Considering SP, Numba versions executed slower than CUDA ones for classes

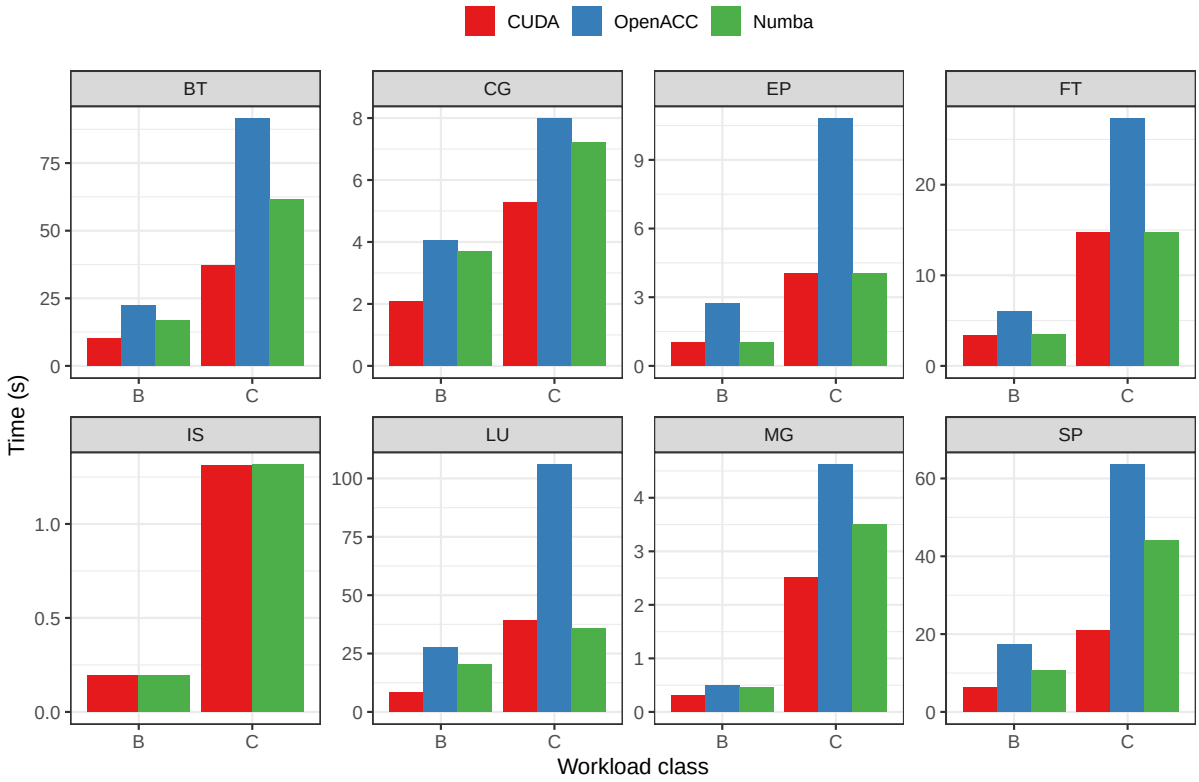

Figure 16 – Additional experiments: Execution time for NPB programs on GPU (T_{GPU}). (Results for IS implemented with OpenACC were not shown since the source code of the program is not available.)

B and C, showing a similar behavior to the BT application. Despite that, Numba outperformed OpenACC for SP using both workload classes.

The next result is related to question **Q3.2** (speedup) of the GQM model. Table 11 illustrates values for metric *TSER* on the **USP** platform (T_S). Similarly to **UFSM**, the C++ version achieved better execution times than Python for all benchmarks but EP. The main fact in the serial results are the improvements on performance, since the CPUs available on **USP** have a greater computational power than the ones on **UFSM**.

	Time (T_s)			
	Class B		Class C	
NPB program	C++	Python	C++	Python
BT	180.271	230.228	755.233	950.683
CG	69.449	104.822	193.652	292.356
EP	76.432	70.499	305.639	281.266
FT	56.080	101.744	248.270	564.849
IS	1.655	2.451	6.840	11.186
LU	139.953	163.469	722.452	801.043
MG	4.517	12.410	43.597	99.224
SP	128.136	148.019	550.291	615.027

Table 11 – Additional experiments: Serial execution time in seconds.

Combining metrics *TGPU* and *TSER*, we calculated metric *TSPU* intending to an-

swer question **Q3.2**. The speedup values are introduced by Figure 17. They were calculated dividing the serial time by the GPU time (T_S / T_{GPU}). As defined in Section 6.3, the serial time from C++ versions were used to compute the CUDA and OpenACC speedups. For Numba speedups, it was applied the Python sequential times.

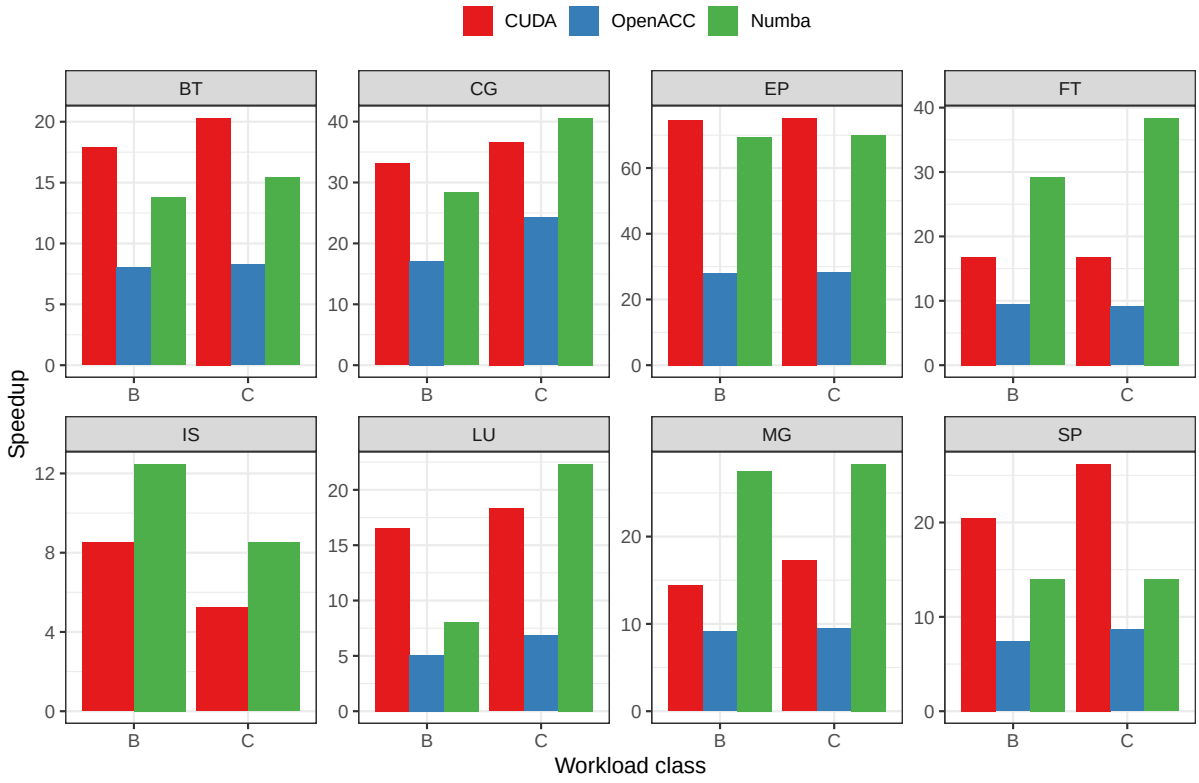

Figure 17 – Additional experiments: Speedup results for NPB programs (T_S / T_{GPU}). (Results for IS implemented with OpenACC were not shown since the source code of the program is not available.)

The speedups of all benchmarks were reduced on **USP** due to hardware differences which have already been mentioned. Comparing **USP** and **UFSM** results, CUDA and OpenACC speedups seem to keep stable. On the other hand, Numba speedups proportionally increased for CG, LU and MG when related to CUDA and OpenACC. These three programs were affected by a low GPU utilization with Numba on **UFSM**, a factor that does not seem to have the same impact on **USP**. Hence, as CG, LU and MG performances had improved with Numba, the Numba speedups also grew up. For the SP application, Numba speedups were smaller than CUDA ones on **USP** influenced by the slowdowns noted to this program with Numba on such platform.

The performance on **USP** suggests that Numba executions on GPU can be affected by the hardware available on a platform. CG, LU, MG and SP presented differences comparing **USP** and **UFSM** results for Numba. This behavior was not perceived for CUDA and OpenACC, which maintained a constant performance on both environments. However, the variances regarding Numba results does not appear to decrease

the gains with Python for GPUs, since the achieved speedups for Numba were similar to CUDA and greater than OpenACC for most cases on both platforms.

B.2 Statistical analysis over results

This Section has the same purpose of Appendix A, providing a statistical analysis to ensure a confidence of at least 95% to the experimental data set presented by Section B.1. Table 12 introduces statistical results for each of the samples. The **KS p-value** column (referencing the Kolmogorov–Smirnov Test) displaying a value greater than 0.05 means a normally-distributed sample considering a significance level of 95%. Otherwise, the value references a non-distributed sample, being highlighted with red.

		Kolmogorov–Smirnov Test for each input size									
		Class B					**Class C**				
NPB Application	Version	Sample size	Mean	STD*	KS p-value	**Normally distrib.?**	Sample size	Mean	STD*	KS p-value	**Normally distrib.?**
BT	C++	30	180.271	0.676	0.294	✓	30	755.233	4.542	0.451	✓
	Python	30	230.228	0.852	0.872	✓	30	950.683	3.076	0.293	✓
	CUDA	30	10.090	0.084	0.059	✓	30	37.186	0.419	0.058	✓
	OpenACC	30	22.444	0.051	0.210	✓	30	91.419	0.155	0.021	–
	Numba	30	16.705	0.044	0.073	✓	30	61.551	0.294	0.094	✓
CG	C++	30	69.449	0.364	0.131	✓	30	193.652	0.928	0.025	–
	Python	30	104.823	0.586	0.911	✓	30	292.356	1.619	0.852	✓
	CUDA	30	2.098	0.012	0.148	✓	30	5.284	0.010	0.072	✓
	OpenACC	30	4.052	0.011	0.613	✓	30	7.980	0.005	0.704	✓
	Numba	30	3.701	0.016	0.324	✓	30	7.202	0.012	0.698	✓
EP	C++	30	76.433	0.086	0.434	✓	30	305.639	0.200	0.280	✓
	Python	30	70.499	0.214	0.096	✓	30	281.266	0.857	0.256	✓
	CUDA	30	1.025	0.001	0.788	✓	30	4.063	0.012	0.533	✓
	OpenACC	30	2.720	0.012	0.389	✓	30	10.800	0.014	0.911	✓
	Numba	30	1.015	0.002	0.545	✓	30	4.026	0.014	0.128	✓
FT	C++	30	56.080	1.263	0.009	–	30	248.270	0.812	0.133	✓
	Python	30	101.744	1.408	0.063	✓	30	564.849	12.747	0.060	✓
	CUDA	30	3.334	0.004	0.324	✓	30	14.788	0.023	0.889	✓
	OpenACC	30	5.971	0.014	0.148	✓	30	27.290	0.178	0.003	–
	Numba	30	3.493	0.002	0.825	✓	30	14.713	0.007	0.115	✓
IS	C++	30	1.656	0.014	0.690	✓	30	6.840	0.072	0.875	✓
	Python	30	2.452	0.007	0.095	✓	30	11.187	0.015	0.351	✓
	CUDA	30	0.194	0.001	0.000	–	30	1.310	0.000	0.428	✓
	Numba	30	0.197	0.002	0.002	–	30	1.315	0.000	0.140	✓
LU	C++	30	139.953	2.489	0.188	✓	30	722.453	12.723	0.078	✓
	Python	30	163.469	1.926	0.277	✓	30	801.043	3.452	0.450	✓
	CUDA	30	8.463	0.006	0.404	✓	30	39.398	0.241	0.024	–
	OpenACC	30	27.578	0.032	0.698	✓	30	105.877	0.060	0.819	✓
	Numba	30	20.281	0.236	0.958	✓	30	35.866	0.101	0.001	–
MG	C++	30	4.518	0.036	0.064	✓	30	43.597	0.226	0.388	✓
	Python	30	12.410	0.029	0.653	✓	30	99.224	0.181	0.164	✓
	CUDA	30	0.314	0.009	0.001	–	30	2.522	0.003	0.000	–
	OpenACC	30	0.493	0.002	0.004	–	30	4.618	0.003	0.589	✓
	Numba	30	0.451	0.007	0.736	✓	30	3.505	0.010	0.471	✓
SP	C++	30	128.136	0.449	0.417	✓	30	550.291	1.187	0.732	✓
	Python	30	148.019	0.657	0.816	✓	30	615.027	1.881	0.612	✓
	CUDA	30	6.255	0.024	0.094	✓	30	21.006	0.054	0.003	–
	OpenACC	30	17.304	0.045	0.004	–	30	63.537	0.072	0.012	–
	Numba	30	10.576	0.032	0.148	✓	30	44.009	0.169	0.402	✓

*STD = Standard deviation

Table 12 – Additional experiments: Kolmogorov–Smirnov Test for performance results (significance level of 95%).

Table 13 depicts values for Student's T-Test (T-Test) and Mann–Whitney U-Test (U-

Test) regarding the comparisons between samples executed in the experiments. If the value in column **p-value** is lower than 0.05, the comparison between Version A and Version B have a significance level of 95% to occur by chance.

			Significance test comparing two samples (versions) for each input size					
			Class B			**Class C**		
NPB Application	Version A	Version B	Test	p-value	**Significant?**	Test	p-value	**Significant?**
BT	C++	CUDA	T-Test	4.716e-073	✓	T-Test	2.965e-066	✓
	C++	OpenACC	T-Test	6.621e-071	✓	U-Test	2.872e-011	✓
	Python	Numba	T-Test	1.866e-071	✓	T-Test	4.587e-074	✓
	CUDA	OpenACC	T-Test	1.772e-096	✓	U-Test	2.872e-011	✓
	CUDA	Numba	T-Test	1.349e-078	✓	T-Test	2.653e-082	✓
	OpenACC	Numba	T-Test	2.398e-103	✓	U-Test	2.872e-011	✓
CG	C++	CUDA	T-Test	1.900e-067	✓	U-Test	2.872e-011	✓
	C++	OpenACC	T-Test	4.693e-067	✓	U-Test	2.872e-011	✓
	Python	Numba	T-Test	1.639e-066	✓	T-Test	1.072e-066	✓
	CUDA	OpenACC	T-Test	1.036e-112	✓	T-Test	7.583e-101	✓
	CUDA	Numba	T-Test	7.078e-096	✓	T-Test	5.428e-110	✓
	OpenACC	Numba	T-Test	4.509e-059	✓	T-Test	1.549e-067	✓
EP	C++	CUDA	T-Test	5.745e-087	✓	T-Test	2.661e-094	✓
	C++	OpenACC	T-Test	8.990e-090	✓	T-Test	3.391e-094	✓
	Python	Numba	T-Test	2.261e-074	✓	T-Test	2.194e-074	✓
	CUDA	OpenACC	T-Test	3.246e-064	✓	T-Test	8.141e-140	✓
	CUDA	Numba	T-Test	5.796e-032	✓	T-Test	5.722e-015	✓
	OpenACC	Numba	T-Test	2.175e-065	✓	T-Test	1.169e-139	✓
FT	C++	CUDA	U-Test	2.872e-011	✓	T-Test	5.914e-073	✓
	C++	OpenACC	U-Test	2.872e-011	✓	U-Test	2.872e-011	✓
	Python	Numba	T-Test	4.991e-055	✓	T-Test	5.589e-049	✓
	CUDA	OpenACC	T-Test	1.232e-077	✓	U-Test	2.872e-011	✓
	CUDA	Numba	T-Test	4.014e-065	✓	T-Test	2.768e-018	✓
	OpenACC	Numba	T-Test	1.099e-069	✓	U-Test	2.872e-011	✓
IS	C++	CUDA	U-Test	2.867e-011	✓	T-Test	2.747e-056	✓
	Python	Numba	U-Test	2.872e-011	✓	T-Test	1.198e-083	✓
	CUDA	Numba	U-Test	1.792e-006	✓	T-Test	1.689e-074	✓
LU	C++	CUDA	T-Test	1.591e-051	✓	U-Test	2.872e-011	✓
	C++	OpenACC	T-Test	1.469e-049	✓	T-Test	1.935e-050	✓
	Python	Numba	T-Test	3.584e-057	✓	U-Test	2.872e-011	✓
	CUDA	OpenACC	T-Test	5.825e-088	✓	U-Test	2.872e-011	✓
	CUDA	Numba	T-Test	6.412e-051	✓	U-Test	2.872e-011	✓
	OpenACC	Numba	T-Test	5.006e-046	✓	U-Test	2.872e-011	✓
MG	C++	CUDA	U-Test	2.863e-011	✓	U-Test	2.872e-011	✓
	C++	OpenACC	U-Test	2.872e-011	✓	T-Test	1.885e-066	✓
	Python	Numba	T-Test	3.385e-085	✓	T-Test	6.234e-081	✓
	CUDA	OpenACC	U-Test	2.863e-011	✓	U-Test	2.872e-011	✓
	CUDA	Numba	U-Test	2.863e-011	✓	U-Test	2.872e-011	✓
	OpenACC	Numba	U-Test	2.872e-011	✓	T-Test	1.771e-068	✓
SP	C++	CUDA	T-Test	1.739e-072	✓	U-Test	2.872e-011	✓
	C++	OpenACC	U-Test	2.872e-011	✓	U-Test	2.872e-011	✓
	Python	Numba	T-Test	4.013e-069	✓	T-Test	1.470e-074	✓
	CUDA	OpenACC	U-Test	2.872e-011	✓	U-Test	2.872e-011	✓
	CUDA	Numba	T-Test	1.749e-103	✓	U-Test	2.872e-011	✓
	OpenACC	Numba	U-Test	2.872e-011	✓	U-Test	2.872e-011	✓

Table 13 – Additional experiments: Significance test comparing versions from performance results (significance level of 95%).

Regarding the applied statistical tests, most of the presented samples are normally distributed according to KS-Test results (Table 12). Also, the proceeded comparisons between these samples have high probability to express the reality, since the employed T-Test and U-Test detailed in Table 13 indicates they are significant.